Gap-free Processor Verification by S^{2}QED and Property Generation

Keerthikumara Devarajegowda*†, Mohammad Rahmani Fadiheh*, Eshan Singh‡
Clark Barrett‡, Subhasish Mitra‡,Wolfgang Ecker†§, Dominik Stoffel*, Wolfgang Kunz*

*Technische Universität Kaiserslautern, Germany, †Infineon Technologies AG, Germany
‡Stanford University, USA, §Technische Universität München, Germany

***Abstract*—The required manual effort and verification expertise are among the main hurdles for adopting formal verification in processor design flows. Developing a set of properties that fully covers all instruction behaviors is a laborious and challenging task. This paper proposes a highly automated and "complete" processor verification approach which requires considerably less manual effort and expertise compared to the state of the art.**

The proposed approach extends the S^{2}QED approach to cover both single and multiple instruction bugs and ensures that a design is completely verified according to a well-defined criterion. This makes the approach robust against human errors. The properties are simple and can be automatically generated from an ISA model with small manual effort. Furthermore, unlike in conventional property checking, the verification engineer does not need to explicitly specify the processor's behavior in different special scenarios, such as stalling, exception, or speculation, since these scenarios are taken care of implicitly by the proposed computational model. The great promise of the approach is shown by an industrial case study with a 5-stage RISC-V processor.

I. Introduction

The evolution of ecosystems for scalable and customizable computing based on the RISC-V instruction set architecture (ISA) along with the demand to deliver millions of computing devices to growing markets such as for the Internet-of-Things (IoT) have created a new and urgent need for highly efficient techniques of processor verification. Verifying the correctness of a processor's behavior after aggressive microarchitectural optimizations and for every possible instruction interleaving is a laborious and demanding task.

Formal verification (FV) by property checking is a powerful technique for exhaustive design analysis. However, developing a set of properties that fully covers all relevant behaviors of a design is known to be a hard problem and requires high formal verification expertise. Although there exist techniques that propose complete formal verification of processor cores [1], [2], they require high manual effort and are hard to integrate in many industrial verification flows.

Formal processor verification is a well researched topic. We provide a summary of the most related works: An approach for control path verification of pipelined processors is proposed in [3] using an efficient validity checker for the logic of uninterpreted functions with equality. This approach of abstracting from the datapath motivated other works with symbolic model checkers [4], [5]. Such methods target abstract models of the processor and can miss bugs in the RTL description. Moreover, developing the architectural description of the processor implementation and the refinement mapping between implementation and specification requires significant manual effort. In [6] an approach is presented which generates properties and verification IPs from an executable specification of the ISA. Being based on bounded proofs and not covering all corner cases of instruction execution, this work contributes to efficient bug hunting but fails to prove the absence of bugs.

A "complete verification" approach is proposed in [2], in which the authors used an industrial microcontroller to demonstrate the formal verification with a "complete set of properties". The proposed approach uses well-defined completeness checks [1] to determine the gaps in the property set, thus ensuring high design quality. Developing a complete set of properties and passing the completeness checks, however, requires a large amount of manual work and high expertise.

In [7], formal verification of processors with a complete set of properties is proposed, in which the properties are generated from an architectural description of the processor. A similar approach in which the properties are generated from semi-formal specifications is proposed in [8]. These approaches, in contrast to the work proposed here, do not follow a model-driven flow but derive design-specific properties based on the implemented architecture. This leads to higher manual efforts for property development, especially when migrating to a new processor architecture.

In [9] it is shown how formal methods can be used to leverage the benefits of quick error detection (QED) tests for pre-silicon verification. The technique termed as Symbolic QED (SQED) is based on Bounded Model Checking (BMC) [10]. It has been shown to be effective in revealing difficult-to-find bugs, is applicable to large designs and requires low manual effort. However, SQED fails to imply the absence of bugs in a processor due to its bounded proof and limited search space. This motivated the work on S^{2}QED [11] which is based on a bounded model with symbolic initial state. It provides an unbounded proof that every instruction executes uniquely independent of its context in the program. However, also S^{2}QED does not allow for a well-defined statement on the absence of all logic bugs in a processor. Therefore, in this paper, we propose *Complete S^{2}QED* (C-S^{2}QED), a gap-free processor verification targeting processor cores of medium complexity with in-order instruction execution. The C-S^{2}QED method has the following characteristics:

1) The proposed approach is integrated into an industrial model-driven design flow and is based on generating a set of S^{2}QED properties from a formal ISA model.
2) The C-S^{2}QED properties fulfill a well-defined completeness criterion derived from [1]. By merit of completeness, C-S^{2}QED exhaustively verifies whether or not the microarchitectural description of a processor at the RTL is a correct implementation of its ISA-level specification.
3) Our approach combines the generation of properties with proving the uniqueness of instruction execution by S^{2}QED. This simplifies both, creating properties and completeness checking. Consequently, when compared to previous complete verification approaches, such as [2], C-S^{2}QED considerably reduces the required manual effort and verification expertise.

The effectiveness of the proposed method is demonstrated by verifying a 5-stage RISC-V processor implemented in an industrial System-on-Chip (SoC) previously verified with a complete property set. The experimental results show that all (previously known) bugs were found by the proposed method with drastically reduced manual effort.

978-3-9819263-4-7/DATE20/©2020 EDAA

The rest of the paper is organized as follows. Sec. II reviews the notion of a complete set of properties and its use in processor verification. Secs. III and IV discuss, in detail, processor verification by S^{2}QED and C-S^{2}QED. Sec. V presents C-S^{2}QED as part of an industrial model-driven flow. Our industrial case study is presented in Sec. VI.

II. Complete Property Set

Interval Property Checking (IPC) is a variant of BMC [10], in which the design behaviors can be specified in a special property format called *operation properties* [1], [12]. Each operation property describes a certain design behavior in a finite time interval in which the design starts and ends in a so called *important state* or *conceptual state*. An operation is defined as a set of finite sequences of state transitions between two important states such that only unimportant states are visited in between. An important state is an abstract state and corresponds to one or more concrete states. Each concrete state can only belong to one important state. Operation properties are supposed to be chained in the sense that the end state of one operation is the start state of the succeeding operation. These operations (transitions between conceptual states) can be viewed as forming a conceptual state machine (CSM). The CSM is a finite automaton describing the sequencing of operations and is close to the specification. The goal is to completely describe every input/output behavior by a sequence of operation properties without any gaps [1], [12]. Interval Property Checking fulfilling the following completeness criterion is called *Complete Interval Property Checking (C-IPC).*

Definition 1. (Complete Property Set): A property set is complete w.r.t. a Design Under Verification (DUV), if the sequence of output signal values over a period of time is uniquely defined by the property set according to a set of *determination requirements*. The *determination requirements* are specifications of signal behavior in the design, describing which output and state variables of the design are to be uniquely determined in each operation [1], [12]. □

The determination requirements define the input/output space considered by the property set.

Corollary 1. A property set $V = \{P_1, P_2, \ldots, P_n\}$ is complete if two arbitrary state machines satisfying all properties in the set are sequentially equivalent with respect to the determination requirements. □

The verification engineer makes a property set complete by ensuring that every possible operation is covered by an operation property and that all outputs and other signals referred to in the determination requirements are uniquely specified at every time point by the operation properties. Completeness of a set of properties can be checked automatically, and independent of a design. A complex sequential equivalence check, as in [13], is not needed. Instead, completeness can be established inductively by considering all pairs (P_i, P_j) of properties describing an operation P_i and a direct successor operation P_j and by performing a set of *completeness tests* [1], [12] described below.

A) **Case Split Test**: The case split test checks that all paths between the important states are covered by at least one property. In other words, it checks that at the ending important state of each operation, for every possible input combination, there exists at least one operation property which determines the next important state. This ensures that there is no input scenario missed in the property set.

B) **Successor Test**: The successor test checks for every operation whether the successor operation is uniquely determined. For every pair of predecessor/successor operations (P_i, P_j), the execution of P_j must be uniquely determined by the predecessor P_i. Passing successor and case split tests ensures that there exists a unique chain of operations for every input trace.

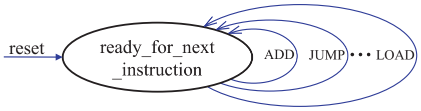

Fig. 1. Conceptual State Machine of a pipelined processor

C) **Determination Test**: The determination test checks whether a set of operation properties uniquely determine the outputs of a circuit (or other signals in determination requirements, e.g., general purpose registers in processors) at all time points.

D) **Reset Test**: The above three tests form an inductive proof stating that if an operation determines its ending important state and output, then there always exists a successor operation that uniquely determines the next important state and output. The validity of this inductive reasoning depends on the reset state (i.e., the induction base). The reset test checks whether the reset input sequence initializes the system deterministically to a unique important state and fulfills all determination requirements.

A. Complete Processor Verification

Pipelined processors execute several operations simultaneously such that different operations overlap each other by one or more clock cycles. Logically, the executions of instructions are independent of each other, even though, physically, they overlap each other in time as they pass through the pipeline. Based on this insight, operation properties can be written in such a fashion that the specified computations and the corresponding determination requirements overlap temporally. In this way, the operation properties provide an abstract view on instruction execution that is close to the ISA-level programmer's view on the processor.

A CSM to capture the operations in a pipeline is shown in Fig. 1. Each operation starts and ends in an important state called *ready_for_next_instruction*. The state transition graph (STG) of this finite state machine is of a special, "degenerate" form: it consists of a single state and many transitions beginning and ending at that state. Each transition is labeled with a different opcode and represents the execution of a specific instruction. The special structure of this STG has implications for checking the completeness of the property set, as discussed for our new approach in Sec. IV.

III. Processor Verification with S^{2}QED

A. Design Errors

Design errors in hardware, often called "bugs" or "logic bugs", lead to incorrect behavior of the implementation in certain scenarios.

Definition 2. (Error Scenario): In the context of processor design, an *error* is a deviation of the implementation's behavior from its specification, in a certain *error scenario*. An error scenario consists of (1) an instruction in which the error becomes observable, (2) the instruction's operands, and, (3) its *program context*, i.e., the sequence of previously executed instructions. □

We say, an error scenario *activates* a logic bug. Logic bugs can be categorized into *single-instruction bugs* or *multiple-instruction bugs*.

Definition 3. (Single-instruction bug): A bug is a *single-instruction bug* if there exists (1) an instruction opcode and (2) a set of operands such that the execution of the instruction leads to an error *in all program contexts*, i.e., independently of all previously executed instructions. □

Definition 4. (Multiple-instruction bug): A bug is a *multiple-instruction bug* if it is not a single-instruction bug and if there exists (1) an instruction opcode, (2) a set of operands, and, (3) a program context such that the execution of the instruction leads to an error. □

A multiple-instruction bug requires error scenarios consisting of specific instruction sequences that set up the micro-architecture of the processor such that the bug is activated. In contrast to a single-instruction bug, there are program contexts in which a multiple-instruction bug is not activated.

B. S^{2}QED Computational Model

S^{2}QED [11] is a formal processor verification approach targeting complex instruction pipelines. S^{2}QED proves that every instruction executes independently of the previous pending instructions in the pipeline, i.e., independently of its program context. The computational model of S^{2}QED consists of two identical and independent instances of the processor under verification which are constrained to execute the same instruction, at an arbitrary time point. Fig. 2 shows the computational model in which the two CPU instances of the same processor are unrolled for a time window as large as the upper bound of the execution time of an instruction in the pipeline.

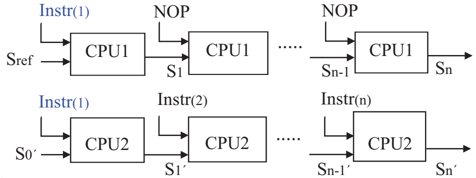

Fig. 2. S^{2}QED Computational model

Definition 5. (QED consistency): In the S^{2}QED computational model, the two CPU instances are *QED-consistent* at a time point t, if the corresponding architectural state elements of both instances at time point t hold the same values. □

For a processor with N registers a QED-consistent register state is characterized by the named logic expression:

$$qed\_consistent\_registers := \bigwedge_{i=0}^{N-1} \left(R^i_{cpu1} = R^i_{cpu2}\right) \qquad (1)$$

This expression is a Boolean predicate that is implemented as a *macro* in the property language of the verification tool. It represents architectural states in which the register files of both CPU instances have identical contents.

Consider an S^{2}QED computational model, in which R_{cpu1} and R_{cpu2} represent the general purpose register files of CPU 1 and CPU 2, respectively. Fig. 3 shows the S^{2}QED property that is to be proven on this model. The property specifies that if two independent CPU instances fetch the same instruction and the register files are consistent with each other before the write-back (the macro *qed_consistent_registers*() specifies this consistency), then the two CPU instances must be QED-consistent also after the write-back, independently of the pipeline context. The CPU 1 instance is constrained to start from a flushed-pipeline state S_{ref} and fetches only NOPs in the time frames for $t > 1$. A flushed-pipeline state S_{ref} is forced on the CPU 1 instance by letting it execute only NOPs for as many time frames before time point t as there are pipeline stages. This results in a significant reduction of proof complexity and excludes any false counterexample to the property that can result from an inconsistent pipeline register. The CPU 2 instance is left unconstrained to start from a symbolic initial state and is allowed to execute an arbitrary sequence of instructions for the time frames $t > 1$. In this computational model, the SAT solver compares the scenario 1, where the *Instruction Under Verification* (IUV) is executed in a flushed-pipeline context, with all scenarios 2 where the IUV is executed in an arbitrary context including the ones where bugs are activated and propagated.

```
assume:
    at $t_{\text{IF}}$:                  cpu2_fetched_instr() = cpu1_fetched_instr();
    during [$t_{\text{IF}}$+1, $t_{\text{WB}}$]:  cpu1_fetched_instr() = NOP;
    at $t_{\text{IF}}$:                  cpu1_state() = $S_{\text{ref}}$;
    at $t_{\text{WB}}$:                  qed_consistent_registers();
prove:
    at $t_{\text{WB}}$+1:                qed_consistent_registers();
```

Fig. 3. Original S^{2}QED property

```
assume:
    at $t_{\text{IF}}$:                  cpu2_fetched_instr() =
                                  cpu1_fetched_instr();
    during [$t_{\text{IF}}$+1, $t_{\text{WB}}$]:  cpu1_fetched_instr() = NOP;
    at $t_{\text{IF}}$:                  cpu1_state() = $S_{\text{ref}}$;
    at $t_{\text{ID}}$:                  ready_for_next_instruction();
    at $t_{\text{ID}}$:                  instr_register_type();
    at $t_{\text{WB}}$:                  qed_consistent_registers();
prove:
    at $t_{\text{EX}}$:                  ready_for_next_instruction();
    at $t_{\text{WB}} + 1$:              qed_consistent_registers();
    at $t_{\text{WB}} + 1$:              cpu1_reg_value( reg_addr @ $t_{\text{ID}}$ ) =
                                  expected_value( funct_type @ $t_{\text{ID}}$ );
```

Fig. 4. S^{2}QED property for Register-type instructions

C. Extending S^{2}QED for single-instruction bugs

The S^{2}QED property shown in Fig. 3 can detect all logic bugs resulting in a QED-inconsistent state [11]. In other words, the instruction execution is verified to be independent of the program context or previously fetched and executed instructions. As a result, the property covers multiple-instruction bugs. However, single-instruction bugs can be masked due to the fact that "common-mode" bugs like a bug in the data path of the ALU have the same effect on both CPU instances and may not lead to a QED-inconsistent state.

In this paper, we extend the original S^{2}QED approach such that it detects all logic bugs in a processor including single-instruction bugs. With our new approach, a complete processor verification can be achieved with substantially less manual effort when compared to traditional C-IPC. This is possible since S^{2}QED "automatically" explores all possible program contexts so that only a much simplified property set is needed that covers just the single-instruction bugs.

Instead of one S^{2}QED property for the entire ISA, we develop an extended S^{2}QED property for each of the different instruction classes such as "register-type", "memory", "control flow". The extended S^{2}QED property is shown in Fig. 4 for register-type instructions in a 5-stage RISC processor. This is denoted by the macro *instr_register_type*() in the assumption. At time point t_{IF}, the same instruction is fetched by both CPU instances, 1 and 2. Just like in the original S^{2}QED property we assume that (i) CPU 1 starts from a flushed pipeline state, (ii) it fetches only NOPs after the time point t_{IF} and (iii) the previous instruction execution has resulted in a QED-consistent state. The macro *ready_for_next_instruction*() describes the state (cf. Fig. 4) of the CPU 1 and CPU 2 pipelines, when they are ready for the next instruction.

At time point t_{ID}, both pipelines begin processing the IUV. Because of pipelining, each processor may be ready to decode the next instruction in the program already one clock cycle later, at t_{EX} (if no stall occurs). In terms of executing the IUV, the conceptual starting and ending states of the operation are assumed at t_{ID} and t_{EX}, even though the processing of the IUV continues for several more clock cycles (until $t_{\text{WB}}+1$).

At time point $t_{\text{WB}}+1$, one clock cycle after the results of an instruction execution are committed, the QED consistency (macro *qed_consistent_registers*()) and the expected values of the instruction execution (macro *expected_value*(*funct_type* @ t_{ID})) are checked. These checks ensure that any logic bug in a processor is found by the new extended S^{2}QED property.

The macro *expected_value* (*funct_type* @ t_{ID}) checks the correctness of results of the instruction execution in the CPU 1 instance, for the following reason: Since the CPU 1 instance fetches NOPs before and after the time point t_{IF}, the complex instruction interleaving scenarios such as forwarding or control transfers do not need to be considered. These scenarios are, instead, covered by checking QED consistency between the CPU instances. This leads to a simplification of the property and its generation from an ISA model (cf. Sec. V). Also, checking QED consistency ensures that the CPU 2 instance completes with the same, expected results. The property shown detects all logic bugs with respect to the execution of register type instructions. Similarly, an S^{2}QED property is developed for each instruction class of the ISA. The S^{2}QED approach is independent of the availability of formal specification.

IV. Completeness Check for C-S^{2}QED

Since the S^{2}QED property set of the previous section does not concern itself with analyzing all special contexts in which an instruction can be running, the conventional completeness check of C-IPC is bound to fail. In particular, it is the case split test that fails. On the other hand, the reset test, successor test and determination test do hold for the proposed S^{2}QED version of the property set if each property uniquely determines the outputs and architectural state variables of the processor. This must be ensured by the set of macros *expected_value*(*func_type*). Since these checks can be applied to our new approach in their conventional form [12], we do not further detail them in this paper.

Next, we present an adaptation of the case split test to the reasoning of the new formulation. We then prove that verifying the S^{2}QED properties under the adapted completeness check is equivalent to C-IPC in terms of covering all logic bugs.

A. Case split test

An operation property P is a property written in the form of an implication $A \implies C$, where the antecedent A is a set of *assumptions* and the consequent C is a set of *commitments*. An assumption or a commitment is an LTL formula where the only temporal operator allowed is X. Such a formula can be mapped to a finite unrolling of a transition structure. In the following, we use the notation *next*(A, l) to denote a "temporal shift" of the formula A by l clock cycles, i.e., *next*$(A, l) := \mathsf{X}^l A$. This adds a temporal offset l to all time points referred to in the formula A.

We first consider the case split test for the property suite of a general hardware design and then look at simplifications for a processor pipeline. Let a set of important states be given by the commitments $\{C_i\}$ of the properties $\{P_i\}$ for an arbitrary design. Then, for every important state (given by a commitment C_P) reached in an operation P it is checked whether the disjunction of the assumptions $\{A_{Q_j}\}$ of all successor properties Q_j completely covers the commitment C_P, i.e., for every path starting in a substate of the important state C_P there exists an operation property Q_j whose assumption A_{Q_j} describes the path. Let $\{A_{Q_1}, A_{Q_2}, \ldots\}$ be the set of assumptions of the successor properties, then the case split test checks whether

$$C_P \rightarrow next((A_{Q_1} \vee A_{Q_2} \vee A_{Q_3} \vee ....), l_P) \qquad (2)$$

where l_P is the length of the property P, i.e., the number of clock cycles between the starting and the ending state of P. The *next* operator aligns the starting state of property Q_j with the ending state of property P_i.

In a processor pipeline, the operation properties all begin and end in the same conceptual state, referred to as *ready_for_next_instruction*() in Fig. 1. In our formulation, we have grouped instructions into properties according to instruction classes, i.e., we have a total of n properties to consider when the processor core supports n instruction classes. Since there exists only a single conceptual state that is reached in every operation, we only need to verify for every property P_i that its commitment C_{P_i} is completely covered by the assumptions of the possible successor properties. Let $\{A^2_{Q_1}, A^2_{Q_2}, A^2_{Q_3}, \cdots\}$ and $\{A^1_{Q_1}, A^1_{Q_2}, A^1_{Q_3}, \cdots\}$ be the assumptions of the successor properties on the CPU 2 and CPU 1 instances, respectively. We need to prove:

$$\begin{aligned} C_{P_i} \rightarrow next(\ (\ (A^1_{Q_1} \wedge A^2_{Q_1}) \vee (A^1_{Q_2} \wedge A^2_{Q_2}) \\ \vee (A^1_{Q_3} \wedge A^2_{Q_3}) \vee \cdots), l_{P_i}) \end{aligned} \qquad (3)$$

Assuming that the S^{2}QED property set holds on the computational model, it is proven that the result of every operation is consistent between the two CPU instances, regardless of the predecessor operations. Therefore, it is sufficient to prove the case split test on one of the CPU instances. In other words, in order to prove the case split test, every possible instruction sequence is implicitly considered on the CPU 2 instance. The case split test for S^{2}QED is reduced to:

$$C_{P_i} \rightarrow next((A^2_{Q_1} \vee A^2_{Q_2} \vee A^2_{Q_3} \vee \cdots), l_{P_i}) \qquad (4)$$

If this test passes, it means that for every possible instruction sequence of the processor there exists a chain of properties that is executed. This means that every (cycle-accurate) execution trace of the processor can be partitioned into finite non-overlapping segments of behavior such that each segment is described by a property. In other words, every execution trace is "covered" by a sequence of operation properties.

The case split test is easy to satisfy for a set of S^{2}QED properties. For a processor that implements n instruction classes, we have n S^{2}QED properties. By ensuring the correctness of the macros that capture the opcode of the instruction classes, such as *instr_register_type*() in Fig. 4, the case split can be easily proven.

B. Complete S^{2}QED

Based on completeness checking with the S^{2}QED-adapted case split test, we can formulate the following theorem.

Theorem 1. *Given an S^{2}QED property set* $V = \{P_1, P_2, \cdots, P_n\}$ *in which* P_i *is a property for an instruction class, created for a given ISA, as described in Sec. III-C. If the property set fulfills the completeness checks for S^{2}QED, then it detects all logic bugs in the ISA implementation of a processor core.*

Proof. Assume that there is a bug in the processor. If the bug is a single-instruction bug, it means that there is an erroneous instruction that produces a wrong result (even) if it is executed in a flushed pipeline. Because the property set

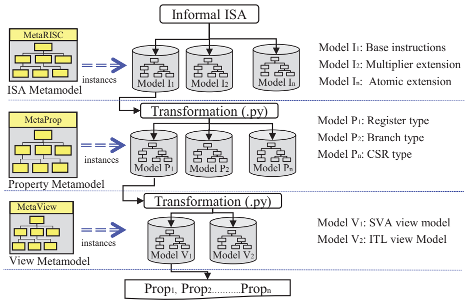

Fig. 5. C-S^{2}QED property generation flow

passes the case split test and the successor test, there exists a single S^{2}QED property targeting the instruction. Since the property set passes the determination test, there exists a macro *expected_value*(*funct_type*) which is called by the commitment of the property and which verifies the computation result of the instruction. According to this macro, the property fails for the expected architectural state in the CPU 1 instance.

If the bug is a multiple-instruction bug, two cases have to be distinguished: (1) The bug is activated in a flushed-pipeline context. Then, the bug will be detected in the same way as described above for the single-instruction bug. (2) The bug is not activated in a flushed-pipeline context. Then, there exists another program context that activates the bug. Because the property set passes the case split test and the successor test, there exists a unique sequence of operation properties covering the program context, i.e., the sequence of instructions in the pipeline. Hence, there exists an S^{2}QED property that covers the program context in the CPU 2 instance of the model and has the instruction exposing the bug as the IUV. Because the property set passes the determination test, the macro *qed_consistent_registers*() determines the full architectural state including the state variables exposing the bug. Since the bug is not activated in the CPU 1 instance, it is detected as an inconsistency between CPU 1 and CPU 2 architectural states. □

In the following, S^{2}QED fulfilling the completeness checks is referred to as *Complete S^{2}QED* (C-S^{2}QED).

V. Automatic Generation of C-S^{2}QED Properties

Property development is considered as one of the major hurdles to the widespread adoption of formal methods in real-life designs. With the proposed C-S^{2}QED approach, the number of properties required to completely verify a processor are effectively reduced to the number of instruction classes supported in the processor. In addition, the C-S^{2}QED properties together with the macros can be generated from a formal ISA model. Automation ensures the correct-by-construction paradigm, and increases the overall verification productivity.

At Infineon, an in-house automation framework based on metamodeling is widely employed for code generation applications. Automatic generation of properties from abstract specifications is one such application [14], which follows the Model Driven Architecture® principle proposed by the Object Management Group® [15]. The generation of C-S^{2}QED properties is implemented following the model-driven flow elaborated in [14]. The framework uses a subset of Unified Modeling Language (UML) class diagrams to represent metamodel definitions. Python is used as the (generation) language to automate the property generation. For efficiently building the code generators for a given metamodel, the metamodeling framework provides an extensive infrastructure including APIs, API extensions, input/output plugins, GUI, input readers, output writers, etc.

The generation flow is illustrated in Fig. 5. The uppermost layer involves translating the informal ISA to a formal ISA model. In this layer, high-level information such as system configurations, e.g., support for multiply/division unit, and intended behaviors, e.g., ADD instruction execution, are considered.

A metamodel, *MetaRISC* in Fig. 5, is created using a UML class diagram to capture the basic structure of the ISA. Each model instance of the *MetaRISC* metamodel is a formal specification model of the ISA, which will be implemented in hardware. A model instance is created with all the instructions (and corresponding details) of the given ISA. For example, RISC-V is a modular ISA and allows for adding several extensions, e.g., support for multiplication/division, atomic execution, etc., on top of the base instruction set. In Fig. 5, *Model* I_1 is a formal specification model of the base instruction set and *Model* I_2 is a formal specification model of the multiplier extension. These model instances are created manually using a GUI provided by the framework.

The main task of the intermediate layer is to transform the specification model to property models. The transformation script, which is manually written in Python, performs two major tasks. First, it extracts the expected behavior of each instruction class from the specification model and invokes the APIs provided by the automation framework to facilitate the generation of macros. Second, it creates a property model for each instruction class in the specification model. During this step, microarchitectural details are considered, such as pipeline length and interface signals to the register file. Using the APIs provided by the framework for the property metamodel *MetaProp*, a property model similar to the one shown in Fig. 4 is written in Python.

The transformation in the bottom layer, metamodel *MetaView* in Fig. 5, maps each property model to a corresponding view model and generates the property in the specified target language, such as SVA.

As described in previous sections, the advantage of the proposed C-S^{2}QED approach results from the fact that the creation of a complete set of properties is drastically simplified. Without exploiting the contribution of this paper, the above model-driven generation flow can also be used to create a complete set of properties based on the classical C-IPC approach. However, when compared to C-S^{2}QED, the transformation script in the intermediate layer is more complex and requires significantly more manual development effort. The reason is that the macro generation needs to consider different special scenarios such as forwarding, stalling, exception, etc., so that more microarchitectural details need to be considered in the transformation script. The transformation script describes a property model for each instruction, explicitly considering interleaving of instructions. Although describing one property model for one instruction class is possible when following the C-IPC approach, the transformation script becomes even more complex and the corresponding generated properties are hard to read and difficult to debug. By contrast, the macro generation for C-S^{2}QED is straightforward, as macros need to be generated only for a flushed-pipeline context (cf. Sec. III-C).

VI. Experiments

The effectiveness of C-S^{2}QED is demonstrated by verifying a RISC-V processor implemented in an industrial SoC for a safety-critical application. The RISC-V core implements a Harvard architecture and supports the RV32I base integer instruction set from the RISC-V ISA. The 5-stage pipelined core has an in-order fetch unit, a 32-bit ALU, a configurable multiply/division unit and an exception handler.

TABLE I
BUG DETECTION RESULTS

	SIC	S^{2}QED	C-S^{2}QED
Finds single-instruction bugs	yes	no	yes
Finds multiple-instruction bugs	yes	yes	yes
Effort for base instr. set (person days)	15	2	5
Effort for new extension (person days)	5	1	1
Runtime (with bugs)	< 30 s	< 60 s	< 30 s
Runtime (without bugs)	27 min	6 min	18 min
CEX length ([min, max] instructions)	[1, 5]	[2, 5]	[1, 5]

The processor core has been previously verified with a complete set of properties generated using a model-driven flow based on the one described in Sec. V. However, this verification approach, called *Spec Implemented Correctly* (SIC) in Table I, is based on conventional C-IPC [2], [12]. 14 logic bugs comprising both single-instruction bugs and multiple-instruction bugs were found during verification. In our experiments for C-S^{2}QED, all these logic bug scenarios were injected into the RTL. For property checking, we used the commercial tool OneSpin 360 DV-Verify on an Intel® Xeon® E5-2690 v3 @2.6GHz with 32 GB RAM.

All 14 logic bugs that were previously detected by the SIC method are also detected by C-S^{2}QED properties within 30 s of computation time. Additionally, two error scenarios were detected by the C-S^{2}QED properties. The reported errors are activated in a flushed-pipeline context, in which the failing properties identified the unnecessary stalling of the pipeline. As the CPU 1 instance is fetching NOPs before and after the time point t_{IF}, the results of the instruction (fetched at t_{IF}) execution are expected at specific time points. Because of this, any unnecessary stalls (which result in performance loss) associated with any instruction class are identified by the C-S^{2}QED property set.

Table I summarizes the results from the C-S^{2}QED experiments (last column) and compares them with other approaches applied to verify the RISC-V core. Columns 1 and 2 correspond to SIC and the original S^{2}QED method [11], respectively. Rows 1 and 2 report on the categories of bugs the different approaches were able to detect. Rows 3 and 4 show the manual effort required to develop the verification setup and the properties for the base integer instruction set and to support the new ISA extension (e.g., mult/div instructions), respectively. For C-S^{2}QED the manual effort mainly consists of developing the ISA model and the transformation scripts in the generation framework. It should be noted that only the transformation script for the property models is design-dependent and the rest can be reused for other designs with the same ISA. "Runtime" refers to the computation time spent to detect a bug (row 5) or to prove its absence (row 6). In case of a bug being reported by the formal tool, the length of the counterexample (CEX) is also reported for each method.

These observations have been made in our experiments:

Observation 1: A complete set of C-S^{2}QED properties can detect all logic bugs in a processor irrespective of their context in the program within a reasonable amount of time. The length of the counterexample is of significant importance to identify the root cause of a bug and denotes the number of instructions that need to be executed to trigger a certain bug scenario. C-S^{2}QED significantly reduces the debugging time due to short counterexamples. The minimum counterexample length for an C-S^{2}QED property is 1 instruction, when a single-instruction bug is detected.

Observation 2: The verification sign-off in industrial flows includes manual review of the property suite (4-eyes principle), comparing it against the (often informal) specification. This can be tedious for a conventional C-IPC approach like SIC. In C-S^{2}QED this review process is much simpler because the properties themselves are simple and the special execution scenarios are taken care of by S^{2}QED.

Observation 3: C-S^{2}QED requires no modification in the RTL code of the design and it has no restriction on the type of instructions it can consider. Developing the C-S^{2}QED property for each instruction class is straightforward and, by employing a generation flow, the manual effort for various extensions in an ISA is drastically reduced.

VII. CONCLUSION

This paper has shown that S^{2}QED and model-driven generation of properties can complement each other nicely, leading to a highly automated approach to complete processor verification. By merit of the proposed computational model, the verification engineer is relieved from analyzing microarchitectural details of instruction execution. C-S^{2}QED is able to capture both single and multiple instruction bugs and fulfills the well-established completeness criterion provided by C-IPC. Our current approach is applicable to the verification of medium complexity in-order processor cores. Extending the approach to out-of-order processor cores and uncore components, such as memory controller, peripherals, etc., is subject to future work.

REFERENCES

[1] J. Bormann and H. Busch, "Verfahren zur Bestimmung der Güte einer Menge von Eigenschaften (Method for determining the quality of a set of properties)," European Patent Application, EP1764715, 09 2005.

[2] J. Bormann, S. Beyer, A. Maggiore, M. Siegel, S. Skalberg, T. Blackmore, and F. Bruno, "Complete formal verification of TriCore2 and other processors," in *Design & Verification Conference (DVCon)*, 2007.

[3] R. B. Jones, D. L. Dill, and J. R. Burch, "Efficient validity checking for processor verification," in *Proc. of IEEE/ACM Intl. Conference on Computer-aided Design (ICCAD)*, 1995, pp. 2–6.

[4] W. Damm, A. Pnueli, and S. Ruah, "Herbrand automata for hardware verification," in *Proc. of Intl. Conference on Concurrency Theory*, 1998, pp. 67–83.

[5] S. Berezin, A. Biere, E. M. Clarke, and Y. Zhu, "Combining symbolic model checking with uninterpreted functions for out-of-order processor verification," in *Proc. of Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 1998, pp. 369–386.

[6] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi, "End-to-end verification of ARM ® processors with ISA-Formal," in *Proceedings of 28th International Conference on Computer Aided Verification*, 2016.

[7] U. Kühne, S. Beyer, J. Bormann, and J. Barstow, "Automated formal verification of processors based on architectural models," in *Proc. of Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2010, pp. 129–136.

[8] R. Baranowski and M. Trunzer, "Complete formal verification of a family of automotive DSPs," in *Proc. Design and Verification Conference Europe (DVCON-Europe)*, 2016.

[9] 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*, 2018.

[10] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Form. Methods Syst. Des.*, vol. 19, no. 1, pp. 7–34, Jul. 2001.

[11] 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 *Design, Automation & Test in Europe Conference (DATE)*. IEEE, 2018, pp. 55–60.

[12] J. Urdahl, D. Stoffel, and W. Kunz, "Path predicate abstraction for sound system-level models of RT-level circuit designs," *IEEE Transactions On Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 33, no. 2, pp. 291–304, Feb. 2014.

[13] K. Claessen, "A coverage analysis for safety property lists," in *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. IEEE Computer Society, 2007, pp. 139–145.

[14] K. Devarajegowda and W. Ecker, "Meta-model based automation of properties for pre-silicon verification," in *IFIP/IEEE Intl. Conference on Very Large Scale Integration (VLSI-SoC)*, 2018, pp. 231–236.

[15] J. M. Siegel, "Model driven architecture ® (mda): The mda guide rev 2.0," june 2014. [Online]. Available: https://www.omg.org/mda/presentations.htm