Template-based Parameterized Synthesis of Uniform Instruction-Level Abstractions for SoC Verification

Pramod Subramanyan, Bo-Yuan Huang, Yakir Vizel, Aarti Gupta, and Sharad Malik

***Abstract*—Modern system-on-chip (SoC) designs comprise programmable cores, application-specific accelerators and I/O devices. Accelerators are controlled by software/firmware and functionality is implemented by this combination of programmable cores, firmware, and accelerators. Verification of such SoCs is challenging, especially for system-level properties maintained by a combination of firmware and hardware. Attempting to formally verify the full SoC design with both firmware and hardware is not scalable, while separate verification can miss bugs.**

A general technique for scalable system-level verification is to construct an abstraction of SoC hardware and verify firmware/software using it. There are two challenges in applying this technique in practice. Constructing the abstraction to capture required details and interactions is error-prone and time-consuming. The second is ensuring abstraction correctness so that properties proven with it are valid.

This paper introduces a methodology for SoC design and verification based on the synthesis of instruction-level abstractions (ILAs). The ILA is an abstraction of SoC hardware which models updates to firmware-visible state at the granularity of instructions. For hardware accelerators, the ILA is analogous to the instruction-set architecture (ISA) definition for programmable processors and enables scalable verification of firmware interacting with hardware accelerators. To alleviate the disadvantages of manual construction of abstractions, we introduce two algorithms for synthesis of ILAs from partial description called templates. We then show how the ILA can be verified to be correct. We evaluate the methodology using a small SoC design consisting of the 8051 microcontroller and two cryptographic accelerators. The methodology uncovered 15 bugs.

***Index Terms*—system-on-chip, systems modeling, formal verification, accelerator architectures, model checking**

I. Introduction

The end of Dennard-scaling [8] has led to power and thermal constraints limiting performance of ICs. We are now in the era of "dark silicon" where significant parts of an IC must be powered-off in order to stay within its power and thermal budgets [10]. Despite the technological limitations imposed by the dark silicon era, the demand for increased performance and energy-efficiency has not subsided and this has led to rise of *accelerator-rich* system-on-chip (SoC) architectures [5]. Application-specific functionality is implemented using fixed-function or semi-programmable accelerators for increased performance and energy-efficiency. As a result, modern SoC designs contain a number of programmable cores, semi-programmable accelerators, I/O devices and memories. Modern SoCs also contain *firmware*; this executes on programmable cores, interacts closely with hardware and orchestrates operation of accelerators and I/O devices.

A. Challenges in SoC Verification

The prevalence of firmware and the emergence of accelerator-rich architectures have introduced new challenges in SoC verification. These are described below.

Challenges due to Firmware: Firmware lies between the operating system and hardware and interacts closely with the hardware. Firmware and hardware make many assumptions about the behavior of the other component. As a result, verifying the two separately requires explicitly enumerating these assumptions and verifying that the other component satisfies them. An example from a commercial SoC highlighting the importance of capturing these interactions is provided in [35]. A series of I/O write operations could be executed by malicious firmware leaving an accelerator in a "confused" state after which sensitive cryptographic keys could be exfiltrated. The bug was because implicit assumptions made by hardware about the timing of firmware I/O writes were violated by the malicious code. This points to the need for scalable co-verification of SoC hardware and firmware.

Challenges due to Accelerator-Rich SoCs: The emergence of accelerator-rich SoC architectures has obsoleted existing hardware/software (HW/SW) and hardware/firmware (HW/FW) abstractions. In the past, programmable hardware meant a programmable core and this was modeled using the core's instruction-set architecture (ISA) specification. Software could be compiled, verified, and reasoned about using this ISA-specification. However, with the proliferation of semi-programmable accelerators in today's SoCs, the ISA abstraction is inadequate at the system-level. System functionality may now be implemented using accelerators and so an ISA-centric view of execution is incomplete. Firmware typically controls and interacts with accelerators by executing memory-mapped I/O (MMIO) reads and writes. These MMIO reads and writes are commands to the accelerator to perform various functions. For example, a command could instruct an accelerator to fetch a block of data from memory, encrypt it using a specified key and write the result back to memory. However, from the perspective of the ISA, all that has occurred is an I/O write. Therefore, there is an important need for abstractions that model the HW/FW and HW/SW interfaces presented by accelerators in modern SoC designs.

Pramod Subramanyan is with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. Bo-Yuan Huang, Yakir Vizel, Sharad Malik are with the Department of Electrical Engineering, Princeton University. Aarti Gupta is with the Department of Computer Science, Princeton University.

This work was supported by in part by C-FAR, one of six SRC STARnet Centers, sponsored by MARCO and DARPA and a research gift from Intel Corporation.

B. Abstractions for SoC Verification

Aggressive time-to-market requirements mean firmware and software for SoCs must be designed before hardware is ready. This requires models of SoC hardware. In practice, such models are usually transaction-level models (TLMs) of SoC components written in SystemC [4, 28, 39]. TLMs divide SoC computations into transactions and application-level functionality is implemented as a sequence of transactions. While TLMs are important, it is difficult to assign system-level meaning and precise semantics to transactions written in SystemC. Furthermore, TLMs can be quite detailed and formal analysis of the TLM along with firmware is challenging.

TLMs illustrate one instantiation of a general solution to the modelling problem: construction of abstractions of SoC hardware. An *abstraction* of SoC functionality is constructed and when verifying properties involving firmware, the abstraction is used instead of the bit-precise cycle-accurate hardware model. Verification using the abstraction is more scalable because irrelevant firmware-invisible details are not included in the abstraction. While the general technique is well-known, we are aware of only a few efforts that have applied this to the co-verification of SoC hardware and firmware [27, 31, 45, 46].

Although the idea of constructing abstractions for firmware verification is attractive, it is challenging to apply in practice. Firmware interacts with hardware components in a myriad of ways. For the abstraction to be useful, it needs to model all interactions and capture all updates to firmware-visible state.

- Firmware usually controls accelerators in the SoC by writing to memory-mapped registers within the accelerators. These registers may set the mode of operation of the accelerator, the location of the data to be processed, or return the current state of the accelerator's operation. The abstraction needs to model these "special" reads and writes to the memory-mapped I/O space correctly.
- Once operation is initiated, accelerators step through a high-level state machine that implements the data processing functionality. Transitions of this state machine may depend on responses from other SoC components, the acquisition of semaphores, external inputs, etc. These state machines have to be modeled to ensure there are no bugs involving race conditions or malicious external input that cause unexpected transitions or deadlocks.
- Another concern is preventing compromised/malicious firmware from accessing sensitive data. To prove that such requirements are satisfied, the abstraction needs to capture issues such as a sensitive value being copied into a firmware-visible temporary register.

Manually constructing an abstraction which captures these details, as proposed for example in [45, 46], is not practical because it is error-prone, as well as tedious and very time-consuming. Abstractions that focus on specific types of properties, like the control flow graph from [27], can address certain verification concerns, but do not capture all of the above requirements. A third alternative is to verify the firmware using a software/SystemC model of the hardware [4, 18, 39]. This too misses bugs present in the hardware implementation but not the SystemC model. The underlying problem with these approaches is in correctness of the abstraction. If the hardware implementation is not consistent with the abstraction, properties proven using it are not valid.

C. Instruction-Level Abstractions for SoC Verification

In this paper, we propose a general methodology for SoC verification based on the construction of abstractions of hardware components that capture updates to all firmware-visible state. We call such abstractions *instruction-level abstractions* (ILAs) and propose techniques for semi-automatic synthesis of ILAs and verification of their correctness.

Instruction-Level Abstractions: An instruction-level abstraction (ILA) of a hardware component is an abstraction that models all firmware-visible state variables and associated state updates in that component. In programmable cores, the ILA models all architectural registers, and in accelerators it models all memory-mapped and I/O addressable registers. The insight underlying the ILA is that firmware only views changes in system state at the granularity of instructions. So hardware components need only be modelled at this granularity.

Uniform and Hierarchical ILAs: Accelerators in today's SoCs perform computation in response to commands sent by programmable cores [5]. This computation is typically bounded in length. Our insight is to view commands from the programmable cores to the accelerators as analogous to "instruction opcodes" and state-updates in response to these commands as "instruction execution." We propose a *uniform* instruction-level abstraction (ILA) which models accelerators using the same fetch/decode/execute sequence as a programmable core. The command is analogous to "fetch," the case-split determining how the command is processed is "decode," and the state update is "execute." Imposing this structure on an abstraction for accelerators allows firmware interactions with accelerators to be modeled using well-understood instruction-interleaving semantics, enabling use of standard tools like software model checkers. Verification of SoC hardware is also easier because conformance with an ILA can be checked compositionally on a "per-instruction" basis leveraging work in microprocessor verification [23, 25].

We also propose hierarchical instruction-level abstractions which allow the construction of compositional models of an accelerator as consisting of a macroILA and possibly several microILAs. The macroILA comprises a set of *macroinstructions*, each of which may be implemented by a sequence of microinstructions that comprise a microILA at a lower level of abstraction. This is analogous to CISC (complex instruction set computer) instructions being implemented as a series of microinstructions. Hierarchy helps manage complexity and models different levels of abstraction in hardware components.

ILA Synthesis: Manual construction of ILAs is tedious and error-prone. These challenges are exacerbated for third-party IPs as ILAs have to be constructed *post hoc* from existing implementations. Therefore, techniques for automated synthesis of ILAs are important. To address this challenge, we propose techniques for the synthesis of ILAs from partial descriptions known as templates. Instead of manually constructing the complete abstraction, the verification engineer

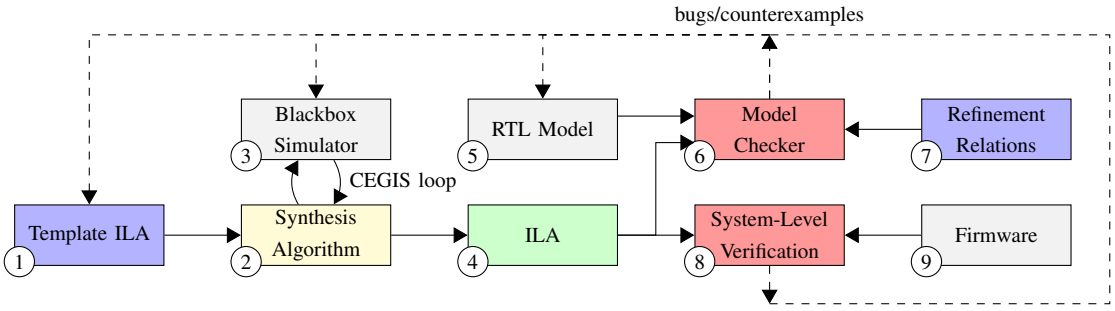

Fig. 1: Overview of the ILA-based SoC Verification Methodology.

now has an easier task of writing a *template* that partially defines the operation of hardware components. The synthesis framework infers the complete abstraction and fills in the missing details by using a *blackbox simulator*[1] of the hardware component. Simulators are often constructed for SoC design and validation, e.g., for simulation-based testing of firmware. In principle, it may be possible to extract abstractions through automated analysis of such simulators. However, in practice, the scale and complexity of simulator codebases make this infeasible. Our work helps constructs ILAs in this scenario. The template abstraction, synthesis framework and blackbox simulator are shown in boxes 1, 2 and 3 in Figure 1.

ILA Verification: To validate the ILA and ensure that the hardware implementation is consistent with the ILA, a set of temporal refinement relations are defined by the verification engineer. These relations specify equivalence between ILA and the register-transfer level (RTL) hardware implementation. These refinement relations are verified using hardware model checking to ensure that RTL behavior matches the ILA. If the refinement relations are proven, we have a guarantee that the abstraction is a correct over-approximation of hardware and any properties proven using the abstraction are in fact valid. If the proof fails, we get counterexamples that can be used to fix either the implementation or the template. ILA verification is shown in boxes 5, 6 and 7 in Figure 1.

Methodology: Figure 1 is an overview of the methodology. Blue boxes (1 and 7) show the components that are provided by the verification engineer. We assume the RTL model and a simulator are already available; these are gray (boxes 3 and 5). Automatically generated artifacts are green (box 4) and off-the-shelf tools are red (boxes 6 and 8). The synthesis algorithm (box 2) is in yellow.

D. Contributions

We introduce a general methodology for template-based synthesis of instruction-level abstractions for SoC verification. The methodology has three advantages. It helps verification engineers *easily* construct *correct* abstractions that are *useful* in verifying system-level properties of SoCs.

We introduce a language for template-based synthesis that is tailored to modeling hardware components in modern SoCs. We introduce two synthesis algorithms based on the counter-example guided inductive synthesis (CEGIS) paradigm [21, 22]. Our first algorithm adapts CEGIS to our context – the synthesis of SoC abstractions. Our second algorithm improves upon this by taking advantage of the instruction-based structure of the ILA and is up to 18× faster than the first algorithm. Together, these algorithms enable scalable synthesis of SoC ILAs. Finally, we show how synthesized ILAs can be verified to be correct abstractions of SoC hardware.

We present a case study applying the methodology to the verification of a simple SoC design built from open-source components. The SoC consists of the 8051 microcontroller and two cryptographic accelerators. We discuss synthesis and verification of instruction-level abstractions in this SoC and describe the bugs found during verification. The methodology helped find a total of 15 bugs in the simulator and RTL.

This methodology was first presented in a conference article [38]. This journal paper introduces the notion of uniform and hierarchical ILAs (§II). While similar abstractions involving fetch, decode and execute have been proposed before, our contribution is in showing how a *uniform abstraction* can be used for both processors and semi-programmable acclerators. This in turn allows us to build on work in software verification to analyze programs that interact with accelerators. This paper also introduces a novel parameterized synthesis algorithm (§IV) which is up to 18× faster (geometric mean 1.9×) than the algorithm presented in [38]. We also include new experimental results (§VI) that evaluate applicability of uniform and hierarchical ILAs and the new synthesis algorithm.

This paper describes synthesis and verification of ILAs, as opposed to system-level verification *using* ILAs (boxes 8 and 9 in 1). An example of security-verification using ILAs in part of a commercial SoC with a 32-bit microcontroller and other peripherals is published in [37]. ILA-based verification was successful in finding bugs in this commercial SoC.

II. Instruction-Level Abstractions

This section provides an overview of uniform and hierarchical instruction-level abstractions (ILAs).

A. Architectural State and Inputs

The architectural state variables of an ILA are modelled as Boolean, bitvector or memory variables. As with ISAs, the

[1]The term *blackbox* simulator (also referred to as an I/O oracle [21]) means the simulator can be used to find the next state and outputs of the component given a specific current state and input value.

architectural state refers to state that is persistent and visible across instructions.

Let $\mathbb{B} = \{0, 1\}$ be the Boolean domain and let $bvec_l$ denote all bitvectors of width l. $M_{k\times l} : bvec_k \rightarrow bvec_l$ maps from bitvectors of width k to bitvectors of width l and represents memories of address width k bits and data width l bits. Booleans and bitvectors are used to model state registers while the memory variables model hardware structures like scratchpads and random access memories. A memory variable supports two operations: $read(mem, a)$ returns data stored at address a in memory mem. $write(mem, a, d)$ returns a new memory which is identical to mem except that address a maps to d, *i.e.,* $read(write(mem, a, d), a) = d$.

Let S represent the vector of state variables of an ILA consisting of Boolean, bitvector and memory variables. In an ILA for a microprocessor, S contains all the architectural registers, bit-level state (e.g., status flags), and data and instruction memories. In an accelerator, S contains all the software-visible registers and memory-structures. A state of an ILA is a valuation of the variables in S.

Let vector W represent the input variables of the ILA; these are Boolean and bitvector variables which model input ports of processors/accelerators.

Let $type_{S[i]}$ be the "type" of state variable $S[i]$; $type_{S[i]} = \mathbb{B}$ if $S[i]$ is Boolean, $type_{S[i]} = bvec_l$ if $S[i]$ is a bitvector of width l and $type_{S[i]} = M_{k\times l}$ if $S[i]$ is a memory.

B. Fetch/Decode/Execute

1) Fetching an Instruction: The result of fetching an instruction is an "opcode." This is modelled by the function $F_o : (S \times W) \rightarrow bvec_w$, where w is the width of the opcode. For instance, in the 8051 microcontroller, $F_o(S, W) \triangleq read(S[\mathrm{IMEM}], S[\mathrm{PC}])$ where $S[\mathrm{IMEM}]$ is the instruction memory and $S[\mathrm{PC}]$ is the program counter. We are using the notation $S[\mathrm{IMEM}]$ to denote the fact that IMEM is a member of the state vector S.

Programmable cores repeatedly fetch, decode and execute instructions, *i.e.*, they always "have" an instruction to execute. However, accelerators may be event-driven and execute an instruction only when a certain trigger occurs. This is modelled by the function $F_v : (S \times W) \rightarrow \mathbb{B}$. For example, suppose an accelerator executes an instruction when either $\mathrm{I[Cmd1Valid]}$ or $\mathrm{I[Cmd2Valid]}$ is asserted, then $F_v(S, W) \triangleq \mathrm{I[Cmd1Valid]} \vee \mathrm{I[Cmd2Valid]}$. Here $\mathrm{Cmd1Valid}$ refers to an input and the notation $\mathrm{I[Cmd1Valid]}$ denotes that this is a member of the input vector I.

2) Decoding an Instruction: Decoding an instruction involves examining an opcode and choosing the state update operation that will be performed. We represent the different choices by defining a set of functions $D = \{\delta_j \mid 1 \leq j \leq C\}$ for some constant C where each $\delta_j : bvec_w \rightarrow \mathbb{B}$. Recall $F_o : (S \times W) \rightarrow bvec_w$ is a function that returns the current opcode. Each δ_j is applied on the result of F_o. The functions δ_j must satisfy the condition: $\forall j, j', S, W : j \neq j' \implies \neg(\delta_j(F_o(S, W)) \wedge \delta_{j'}(F_o(S, W)))$; *i.e.*, the functions δ_j use a "one-hot" encoding.

For convenience let us also define the predicate $op_j \triangleq \delta_j(F_o(S, W))$. When op_j is 1, it selects the jth instruction. For example, in the case of the 8051 microcontroller, $D = \{\delta_1(f) \triangleq (f = 0), \delta_2(f) \triangleq (f = 1), \ldots, \delta_{256}(f) \triangleq f = 255\}$.[2] Recall we had defined F_o for this microcontroller as $F_o(S, W) \triangleq read(S[\mathrm{IMEM}], S[\mathrm{PC}])$. Therefore, $op_j \iff read(S[\mathrm{IMEM}], S[\mathrm{PC}]) = (j - 1)$. We are "case-splitting" on each of the 256 values taken by the opcode and each of these performs a different state update. The functions δ_j choose which of these updates is to be performed.

3) Executing an Instruction: For each state element $S[i]$ define the function $N_j[i] : (S \times W) \rightarrow type_{S[i]}$. $N_j[i]$ is the state update function for $S[i]$ when $op_j = 1$. For example, in the 8051 microcontroller, opcode `0x4` increments the accumulator. Therefore, $N_4[\mathrm{ACC}] = \mathrm{ACC} + 1$.

The complete next state function $N : (S \times W) \rightarrow S$ is defined in terms of the functions $N_j[i]$ over all i and j.

C. Hierarchical ILAs

Hierarchical ILAs handle different levels of abstraction in hardware components and allow ILAs to contain other ILAs. We call the inner ILAs microILAs while the outer/parent ILA is called a macroILA. Each microILA has its own state and input variables, fetch, decode and next state functions: $S^\mu, W^\mu, F_o^\mu, F_v^\mu, D^\mu$ and N^μ respectively. Each microILA is associated with a Valid function $V^\mu : S \times W \rightarrow B$ where S and W are the state and input variables of the macroILA containing it. $V^\mu = 1$ iff the valuation of the state variables S^μ is legal, and the microILA only executes when $V^\mu = 1$ MacroILAs and microILAs execute concurrently and asynchronously.

Communication between the macro and micro ILAs happens in two ways. The microILA can read all macro-state and use these in its next state computations. Similarly, the macroILA can read microILA state when $V^\mu = 1$ and use these variables in the macroILA's next state computation.

An example of hierarchical ILAs in our experimental platform is an ILA for an accelerator that implements the Advanced Encryption Standard (AES) algorithm. The AES accelerator has two parts. One part contains the configuration registers and the processor core interface. The other is a state machine that repeatedly fetches data from the shared RAM, encrypts it and writes back the encrypted data. We model the encryption state machine as a microILA, and each state in it is modeled as a "microinstruction." The interface that interacts with the processor and controls the state machine is the macroILA.

D. Putting it all Together

To summarize, an instruction-level abstraction (ILA) is the tuple: $\mathrm{A} = \langle S, W, F_o, F_v, D, N, L^\mu \rangle$. S and W are the state and input variables. F_o, F_v, D and N are the fetch, decode and next state functions respectively. $L^\mu = \{(V^{\mu_p}, A^{\mu_p}), ...\}$ is a set of microILAs contained within this ILA. $V^{\mu_p} : (S \times W) \rightarrow \mathbb{B}$ is the valid function associated with the microILA A^{μ_p}.

[2] We are abusing notation here by writing elements of $bvec_8$ as $0 \ldots 255$.

E. Syntax

The language of expressions allowed in F_o, F_v, D and $N_j[i]$ is shown in Figure 2. Expressions are quantifier-free formulas over the theories of bitvectors, bitvector arrays and uninterpreted functions. They can be of type Boolean, bitvector or memory, and each of these has the usual variables, constants and operators with standard interpretations. The synthesis primitives, shown in **bold**, will be described in §III.

⟨*exp*⟩ ::= ⟨*bv-exp*⟩ | ⟨*bool-exp*⟩ | ⟨*mem-exp*⟩ | ⟨*func-exp*⟩
  | ⟨*choice-exp*⟩

⟨*bv-exp*⟩ ::= var ⟨*id*⟩ *width* | cnst `val` *width*
  | bvop ⟨*exp*⟩ ...
  | read ⟨*memexp*⟩ ⟨*addrexp*⟩
  | read-block ⟨*memexp*⟩ ⟨*addrexp*⟩
  | apply ⟨*func*⟩ ⟨*bv-exp*⟩ ...
  | **extract-bitslice** ⟨*bv-exp*⟩ *width*
  | **extract-subword** ⟨*bv-exp*⟩ *width*
  | **replace-bitslice** ⟨*bv-exp*⟩ ⟨*bv-exp*⟩
  | **replace-subword** ⟨*bv-exp*⟩ ⟨*bv-exp*⟩
  | **in-range** ⟨*bv-exp*⟩ ⟨*bv-exp*⟩

⟨*bool-exp*⟩ ::= var | true | false
  | boolop ⟨*exp*⟩ ...

⟨*mem-exp*⟩ ::= ⟨*id*⟩ |
  | write ⟨*mem-exp*⟩ ⟨*bv-exp*⟩ ⟨*bv-exp*⟩
  | write-block ⟨*mem-exp*⟩ ⟨*bv-exp*⟩ ⟨*bv-exp*⟩

⟨*func-exp*⟩ ::= func ⟨*id*⟩ $width_{out}$ $width_{in_1}$ ...

⟨*choice-exp*⟩ ::= **choice** ⟨*exp*⟩ ⟨*exp*⟩ ...

Fig. 2: Syntax for Expressions

III. ILA Synthesis

The goal of ILA synthesis is to help semi-automatically synthesize the functions $N_j[i]$. To do this, we build on work in oracle-guided program synthesis [21, 22]. In particular, we assume availability of a simulator that models state updates performed by the accelerator. In practice, this simulator can be either an RTL description of the hardware component, or a high-level C/C++/SystemC simulator. Note this is a black-box simulator, also called an I/O oracle, which can be used to simulate the execution of a component given an initial state and an assignment to the component's inputs.

A. Notation and Problem Statement

Let $Sim : (S \times W) \rightarrow S$ be the I/O oracle for the next state function N. Define $Sim_i : (S \times W) \rightarrow type_{S[i]}$ to be the function that projects the state element $S[i]$ from Sim.[3] In order to help synthesize the function implemented by Sim_i, the SoC designers write a *template next state function*, denoted by $\mathcal{T}_i : (\Phi \times S \times W) \rightarrow type_{S[i]}$.

Φ is a set of *synthesis* variables, also referred to as "holes" [33], and different assignments to Φ result in different next state functions. Unlike $N[i]$, $\mathcal{T}_i$ is a partial description and is therefore easier to write. It can omit certain low-level details, such as the mapping between individual opcodes and operations, opcode bits and source and destination registers, etc. These details are filled-in by the counter-example guided inductive synthesis (CEGIS) algorithm by observing the output of Sim_i for carefully selected values of (S, W).

(Problem Statement: ILA Synthesis): For each state element $S[i]$ and each op_j, find an interpretation of Φ, Φ^i_j, such that $\forall S, W : op_j \implies (\mathcal{T}_i(\Phi^i_j, S, W) = Sim_i(S, W)))$.

Note the synthesis procedure is repeated for each instruction (each j) and each state element (each i), and the corresponding synthesis result is Φ^i_j.

[3] $Sim_i(S, W) = Sim(S, W)[i]$.

B. Template Language

The template identifies: (i) elements of architectural state, (ii) components of each instruction: the fields and ranges on/in which the instruction operates, (iii) a skeleton of possible state updates, and (iv) the direction of data-flow. Most importantly, the template implicitly decouples the orthogonal concerns of precisely matching low-level bitfields in order to identify opcodes and the state updates performed by each opcode. In our experience, these tightly-coupled concerns are often the most error-prone and tedious parts of manual model construction.

The synthesis primitives in the currently implemented template language are shown in bold in Figure 2. Note our algorithms/methodology are not dependent on these specific synthesis primitives. The only requirement placed on the primitives is that they can be "compiled" to some quantifier-free formula over bitvectors, arrays and uninterpreted functions. These theories are typically supported in all modern satisfiability modulo theory (SMT) solvers, *e.g.*, Z3 [7].

Synthesis Primitives: The expression **choice** ε_1 ε_2 asks the synthesizer to replace the **choice** primitive with either ε_1 or ε_2 based on simulation results. **choice** ε_1 ε_2 is translated to the formula $\text{ITE}(\phi_b, \varepsilon_1, \varepsilon_2)$ where $\phi_b \in \Phi$ is a new Boolean variable associated with this instance of the choice primitive. Its value is determined by the synthesis procedure.

The primitives **extract-slice** and **extract-subword**, synthesize bitvector extract operators. The synthesis process determines the indices to be extracted from. The **replace-slice** and **replace-subword** are the counterparts of these primitives; they replace a part of the bitvector with an argument expression. The primitive **in-range** synthesizes a bitvector constant that is within the specified range. Adding new synthesis primitives is easy and straightforward in our framework.

C. An Illustrative Example

We illustrate the definition of an ILA and template next state function using the processor shown in Figure 3. The instruction to be executed is read from the ROM. Its operands can either be an immediate value (data) from the ROM or from the 4-entry register file. For simplicity, we assume that the only two operations supported by the processor are addition and subtraction.

The architectural state for the processor is $S = \langle \text{ROM}, \text{PC}, \text{R}_0, \text{R}_1, \text{R}_2, \text{R}_3 \rangle$ and input set W is empty. $type_{\text{ROM}} = M_{8\times 8}$ while all the other variables are of type $bvec_8$. The opcode which determines the next instruction is

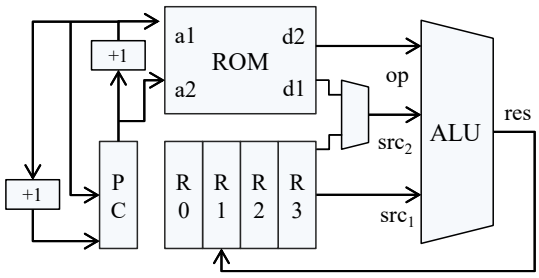

Fig. 3: A simple processor for illustration.

stored in the ROM and so $F_o \triangleq read(\mathrm{ROM}, \mathrm{PC})$; $F_v \triangleq 1$. $D = \{\delta_j \mid 1 \leq j \leq 256\}$ where each $\delta_j(f) \triangleq f = j - 1$.[4] The template next state functions, $\mathcal{T}_{\mathrm{PC}}$ and $\mathcal{T}_{\mathrm{R_i}}$, are as follows.

$$
\begin{array}{rcl}
\mathcal{T}_{\mathrm{PC}} & = & \textbf{choice } (\mathrm{PC}+1)\ (\mathrm{PC}+2) \\
imm & = & read(\mathrm{ROM}, \mathrm{PC}+1) \\
src_1 & = & \textbf{choice } \mathrm{R}_0\ \mathrm{R}_1\ \mathrm{R}_2\ \mathrm{R}_3 \\
src_2 & = & \textbf{choice } \mathrm{R}_0\ \mathrm{R}_1\ \mathrm{R}_2\ \mathrm{R}_3\ imm \\
res & = & \textbf{choice } (src_1 + src_2)\ (src_1 - src_2) \\
\mathcal{T}_{\mathrm{R_i}} & = & \textbf{choice } res\ R_i \quad (0 \leq i \leq 3)
\end{array}
$$

Algorithm 1 CEGIS Algorithm

```
1:  function SynthesizeAll(T, Sim)
2:      for all S[i] ∈ S do
3:          for all op_j do
4:              Φ^i_j ← SynCEGIS(op_j, T_i, Sim_i)
5:              N_j[i](S, W) ← T_i(Φ^i_j, S, W)
6:          end for
7:      end for
8:  end function
9:  function SynCEGIS(op_j, T_i, Sim_i)
10:     k ← 1
11:     R_1 ← op_j ∧ (θ ↔ (T_i(Φ_1, S, W) ≠ T_i(Φ_2, S, W)))
12:     while sat(R_k ∧ θ) do
13:         Δ ← Model_(S,W)(R_k ∧ θ)   ▷ get dist. input Δ
14:         O ← Sim_i(Δ)                ▷ simulate Δ
15:         O_1 ← (T_i(Φ_1, Δ) = O)
16:         O_2 ← (T_i(Φ_2, Δ) = O)
17:         R_{k+1} ← R_k ∧ O_1 ∧ O_2 ▷ enforce output O for Δ
18:         k ← k + 1
19:     end while
20:     if sat(R_k ∧ ¬θ) then
21:         return Model_Φ1(R_k ∧ ¬θ)
22:     end if
23:     return ⊥
24: end function
```

D. CEGIS Algorithm

The counter-example guided inductive synthesis (CEGIS) algorithm for synthesizing $N_j[i]$ from template next state function $\mathcal{T}_i$ and the simulator Sim is shown in Algorithm 1. The function SynthesizeALL calls SynCEGIS for each op_j (the different "opcodes") and each $S[i]$ (each of element of architectural state). In each case, SynCEGIS returns Φ^i_j which is used to compute the next state function as $N_j[i](S, W) = \mathcal{T}_i(\Phi^i_j, S, W)$.

SynCEGIS tries to find an interpretation of (S, W), say Δ, which for some two interpretations of Φ: Φ_1 and Φ_2, is such that $\mathcal{T}_i(\Phi_1, \Delta) \neq \mathcal{T}_i(\Phi_2, \Delta)$. To understand the algorithm, observe that $\mathcal{T}_i(\Phi, S, W)$ defines a *family* of next-state functions. Different functions are selected by different assignments to Φ. The key idea is to repeatedly find *distinguishing inputs* [21] while ensuring the simulation input/output values observed thus far are satisfied. A distinguishing input for Φ_1 and Φ_2 is an assignment to S and W such that the $\mathcal{T}_i(\Phi_1, S, W) \neq \mathcal{T}_i(\Phi_2, S, W)$. The distinguishing input Δ is found in line 13. Next we use the simulator Sim_i to find the correct output O and assert that the next distinguishing input must satisfy the condition that the output for Δ is O (lines 15, 16 and 17). When no more distinguishing inputs can be found, then all assignments to S define the same transition relation and we pick one of these assignments in line 21.

IV. Parameterized Synthesis

In this section, we present an improved synthesis algorithm. We start with a high-level discussion of how the algorithm works before presenting its details.

A. Motivating Parameterized Synthesis

As the inner loop of SynthesizeALL in Algorithm 1 shows, SynCEGIS is executed for each of the different decode functions represented by op_j. For the illustrative example from §III-C, this means that we execute Algorithm 1 for each opcode: $\{op_1 \iff read(\mathrm{ROM}, \mathrm{PC}) = 0, op_2 \iff read(\mathrm{ROM}, \mathrm{PC}) = 1, \ldots, op_{256} \iff read(\mathrm{ROM}, \mathrm{PC}) = 255\}$. Consider the synthesis of one element of architectural state: PC. Excluding a few instructions that operate on an immediate value, for most opcodes, the next state function for the PC is PC + 1.[5] However, Algorithm 1 repeatedly rediscovers the same next state function (PC+1) by computing new distinguishing inputs and then pruning the search space according to the corresponding simulator outputs. Finding distinguishing inputs is the most computationally expensive part of synthesis. The parameterized synthesis algorithm attempts to avoid this computation as much as possible.

To understand how parameterized synthesis works, let us first consider a strawman proposal. For the first opcode, *i.e.*, when executing $\mathrm{SynCEGIS}(op_1, \mathcal{T}_i, Sim_i)$ we will execute SynCEGIS as usual. However, we will record the distinguishing inputs, the corresponding simulator outputs and next state function computed by the algorithm. Let the sequence of pairs of distinguishing inputs and simulator outputs be $\langle(\Delta^1, O^1), \ldots, (\Delta^k, O^k)\rangle$. Now suppose we are executing

[4] In the rest of this paper, we shall refer to the elements of the state vector as ROM, PC etc., instead of $S[ROM]$ or $S[PC]$ in order to keep notation uncluttered.

[5] This is a common scenario. For all elements of architectural state, there are often a few next state functions that occur across many different opcodes. For example, in the case of R_0, the most common next state function is the identity function R_0.

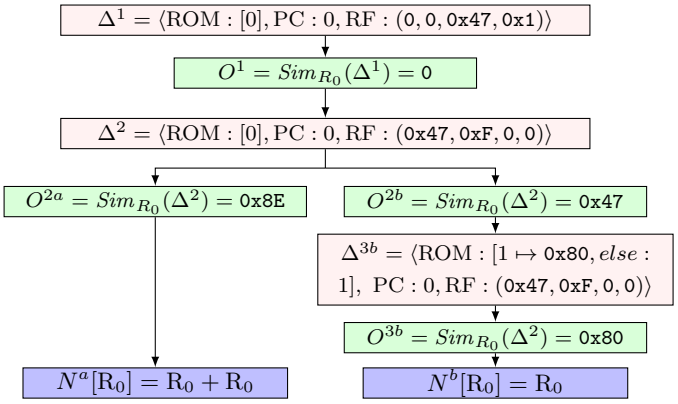

Fig. 4: Distinguishing Input Tree (DIT). The notation $\text{ROM} = [0]$ means that all entries in the ROM map to value zero. The notation $\text{ROM} = [0 \mapsto 1, else : 0\text{x}0]$ means that ROM address 0 maps to 1, but all other addresses map to 0. The notation $RF = (0, 0, 0\text{x}47, 0\text{x}1)$ refers to the valuation of the register file; it means that $\text{R}_0 = \text{R}_1 = 0$, $\text{R}_2 = 0\text{x}47$ and $\text{R}_3 = 0\text{x}1$.

SYNCEGIS$(op_2, \mathcal{T}_i, Sim_i)$. The strawman proposal does not recompute distinguishing inputs but instead presents the distinguishing inputs computed for op_1: $\langle \Delta^1, \ldots, \Delta^k \rangle$ to the simulator and evaluates its output. Suppose these outputs are the same as those observed for op_1: $\langle O^1, \ldots, O^k \rangle$. This means the search space has been pruned identically and so the next state functions are the same for both op_2 and op_1. If this occurs, the SMT solver need never be invoked for the computation of distinguishing inputs!

Unfortunately, this strawman proposal is flawed. This is because the distinguishing inputs are computed for a specific op_j: $\Delta \models op_j \wedge (\theta \leftrightarrow (\mathcal{T}_i(\Phi_1, S, W) \neq \mathcal{T}_i(\Phi_2, S, W)))$. However, $\forall j, j' : j \neq j' \implies \neg(op_j \wedge op_{j'})$. The distinguishing inputs for op_j and op'_j *must* be different. To make this idea work, we need a way of transforming distinguishing inputs computed for op_j into distinguishing inputs for $op_{j'}$. If we could do this, then we would not need to recompute distinguishing inputs if the next state function was "seen" before.

B. An Intuitive Explanation

We start with an explanation of the algorithm using a worked out example for the processor shown in §III-C. Figure 4 shows a *distinguishing input tree (DIT)*. A distinguishing input tree (DIT) consists of three types of nodes: (i) distinguishing input nodes (shown in red), (ii) simulator outputs (green), and (iii) symbolic expressions for the next state (blue). Each path from the root to a leaf node in the tree represents an equivalence class of semantically equivalent next state functions ($N_j[i]$) and the distinguishing inputs and corresponding outputs that occur along this path uniquely identify this next state function.

The tree shown in Figure 4 shows the computation of $N_j[\text{R}_0]$ over different op_j. Consider the path $\langle(\Delta^1, O^1), (\Delta^2, O^{2a}), N^a[\text{R}_0]\rangle$ in Figure 4. This path starts at the root (Δ^1) and terminates at a leaf ($N^a[R_A]$). It corresponds to the computation of $N_1[\text{R}_0]$ where $op_1 \iff read(\text{ROM}, \text{PC}) = 0\text{x}0$. For the first opcode ($op_1$), parameterized synthesis works identically to SYNCEGIS with the only difference being that distinguishing inputs, simulator outputs and the next state function are inserted into the DIT.

Now suppose we are synthesizing $N_2[\text{R}_0]$. Recollect that $op_2 \iff read(\text{ROM}, \text{PC}) = 1$. As described earlier, our goal is to reuse the previously computed distinguishing input Δ^1. The distinguishing characteristic of Δ^1 is in the assignments to $\text{R}_0, \text{R}_1, \text{R}_2, \text{R}_3$ and $read(\text{ROM}, \text{PC}+1)$. The distinguishing nature does not depend on $read(\text{ROM}, \text{PC})$ which contains the opcode. We would be able to reuse this distinguishing input for op_2 if we changed the assignment to ROM such that $read(\text{ROM}, 0) = 0\text{x}1$ while "keeping everything else the same." The tricky part here is formalizing "keeping everything else the same."

Let us consider a strawman proposal to achieve this. Suppose we use an SMT solver and find an assignment $\texttt{ROM}'$ such that $\texttt{ROM}' \models op_2$. One such assignment is all ROM addresses in $\texttt{ROM}'$ map to 1: $\texttt{ROM}' = [1]$. Now consider the distinguishing input $\Delta^{1'} \triangleq \langle \texttt{ROM}' = [1], \text{PC} = 0, \text{RF} = (0, 0, 0\text{x}47, 0\text{x}7)\rangle$. $\Delta^{1'}$ is exactly the same as Δ^1, except that $\texttt{ROM}'$ has been changed so that $\texttt{ROM}' \models op_2$. But $\Delta^{1'}$ and Δ^1 are not equivalent in terms of distinguishing power. Δ^1 can distinguish between the functions $\text{R}_3 + \text{R}_3$ and $\text{R}_3 + read(\text{ROM}, \text{PC}+1)$. The former evaluates to 2 while the latter evaluates to 1 under the assignment Δ^1. However, $\Delta^{1'}$ cannot distinguish between these functions (both evaluate to 2) and is weaker in terms of distinguishing power than Δ^1. Therefore, a more precise formulation of "keeping everything else the same" involves showing that the new input does not weaken the original assignment's distinguishing power. The algorithm introduced in this section allows us to reuse distinguishing inputs by making simple syntactic substitutions while retaining the same distinguishing power.

Let $\Delta^{1'} = \langle \text{ROM} : [0 \mapsto 1, else : 0], \text{PC} : 0, \text{RF} : (0, 0, 0\text{x}47, 0\text{x}1)\rangle$ be a "minimally changed" version of Δ^1. Note $\Delta^{1'}$ retains the distinguishing power of Δ^1 while also satisfying op_2. We call $\Delta^{1'}$ an S_D-variant interpretation of Δ^1. A formal definition of S_D-variant interpretations is in the next section.[6] *Our algorithm computes S_D-variant distinguishing inputs by solving a simple SMT instance for the predicate op_j, and then performs a syntactic substitution on the distinguishing inputs in the DIT.*

Now suppose $\Delta^{1'}$ results in the same output $O^{1'} = 0$ from the simulator. This means that the search space can be pruned in the same was with O^1. Therefore, we continue following this path in the DIT and compute the next S_D-variant distinguishing input of Δ^2. Now the simulator returns an output we have not seen before. We insert this into the tree as node O^{2b}. We are now on a new path in the tree and use the SMT solver to compute distinguishing input Δ^{3b}. At this point, we do not have any reusable information and the algorithm devolves into CEGIS. And this eventually results in the computation of $N_2[\text{R}_0]$ as R_0. *Now for all the other opcodes which have the same next state function* R_0*, we can*

[6] S stands for support and D is the set of decode functions, so an S_D-variant interpretation varies only over the support of the decode functions.

just follow this path in the DIT instead of solving many more new SMT instances.

C. Definitions

(**Definition 1: Supporting Subexpressions**) Let ε be an expression and σ be some subexpression (some node in the abstract syntax tree) of ε. We say σ is a *supporting subexpression* of ε if there exist ν_1 and ν_2 such that substituting σ with ν_1 and ν_2 in ε causes ε to differ: *i.e.*, $\varepsilon[\sigma/\nu_1] \neq \varepsilon[\sigma/\nu_2]$. In other words, σ is a subexpression that affects the value of ε. For example, let $\varepsilon \triangleq read(\mathrm{ROM}, \mathrm{PC}+1) = 0$. Then $\nu_1 \triangleq \mathrm{PC}+1$ is a supporting subexpression of ε.

(**Definition 2: Complete Set of Supporting Subexpressions**) Let $S_\varepsilon = \{\sigma_1, \ldots, \sigma_p, \ldots, \sigma_L\}$ be a set of supporting subexpressions of ε. We say S_ε is a *complete set* of supporting subexpressions, if for all interpretations I_1 and I_2 of ε, if $[\![\varepsilon]\!]_{I_1} \neq [\![\varepsilon]\!]_{I_2}$ then there exists some $\sigma_p \in S_\varepsilon$ such that $[\![\sigma_p]\!]_{I_1} \neq [\![\sigma_p]\!]_{I_2}$.[7] In other words, if two interpretations differ in the value of ε, then at least one of the expressions in a complete set of supporting expressions of ε must also differ. Let us return to example of $\varepsilon \triangleq read(\mathrm{ROM}, \mathrm{PC}+1) = 0$. The singleton set $\{read(\mathrm{ROM}, \mathrm{PC}+1)\}$ is in fact a complete set of supporting subexpressions for ε. This is because ε is a Boolean, and if its truth value changes from 0 to 1 or vice versa, it must be because $read(\mathrm{ROM}, \mathrm{PC}+1)$ changed.

We can extend this definition to a set of expressions $E = \{\varepsilon_1, \ldots, \varepsilon_q, \ldots, \varepsilon_Q\}$. $S_E = \{\sigma_1, \ldots, \sigma_p, \ldots, \sigma_L\}$ is a complete set of supporting subexpressions for the set E if for all interpretations I_1, I_2 and all $\varepsilon_q \in E$, if $[\![\varepsilon_q]\!]_{I_1} \neq [\![\varepsilon_q]\!]_{I_2}$ then there exists some $\sigma_p \in S_E$ such that $[\![\sigma_p]\!]_{I_1} \neq [\![\sigma_p]\!]_{I_2}$. Consider the set of expression $E \triangleq \{read(\mathrm{ROM}, \mathrm{PC}) = 0, read(\mathrm{ROM}, \mathrm{PC}) = 1, \ldots, read(\mathrm{ROM}, \mathrm{PC}) = 255\}$. The singleton set $S_E \triangleq \{read(\mathrm{ROM}, \mathrm{PC})\}$ is a complete set of supporting expressions for E. Any change in the truth value of any element of E must necessarily be accompanied by a change in the valuation of the sole member of S_E.

(**Definition 3: σ_p-variant Interpretations**) Given a subexpression σ_p, we say that two interpretations I_1 and I_2 are σ_p-variant if for all expressions ε, $[\![\varepsilon]\!]_{I_1} \neq [\![\varepsilon]\!]_{I_2} \implies [\![\sigma_p]\!]_{I_1} \neq [\![\sigma_p]\!]_{I_2}$. In other words, if two interpretations are σ_p-variant, then they differ only in their assignments to σ_p and expressions that depend on σ_p and nothing else. Let $\sigma_p \triangleq read(\mathrm{ROM}, \mathrm{PC})$. Then the assignments $\Delta^1 = \langle \mathrm{ROM} : [0], \mathrm{PC} : 0, \mathrm{RF} : (\mathtt{0}, \mathtt{0}, \mathtt{0x47}, \mathtt{0x1})\rangle$ and $\Delta^{1'} = \langle \mathrm{ROM} : [0 \mapsto 1, else : 0], \mathrm{PC} : 0, \mathrm{RF} : (\mathtt{0}, \mathtt{0}, \mathtt{0x47}, \mathtt{0x1})\rangle$ are σ_p-variant.

This definition can be extended to a set of subexpressions S_E. Two interpretations are S_E-variant if for all expressions ε, if $[\![\varepsilon]\!]_{I_1} \neq [\![\varepsilon]\!]_{I_2}$, there exists $\sigma_p \in S_E$ such that $[\![\sigma_p]\!]_{I_1} \neq [\![\sigma_p]\!]_{I_2}$. If we define S_E as the singleton set $\{read(\mathrm{ROM}, \mathrm{PC})\}$, then Δ^1 and $\Delta^{1'}$ as defined in the previous paragraph are S_E-variant.

D. Sketch of the Algorithm

Suppose we have a complete set of supporting subexpressions S_D for the set of decode predicates $\{op_j \mid 1 \leq j \leq C\}$.

[7] Notation $[\![\varepsilon]\!]_I$ refers to expression ε evaluated under interpretation I.

Algorithm 2 Parameterized synthesis

```
1:  procedure SynthesizeAll(D, 𝒯, Sim)
2:      for all S[i] ∈ S do
3:          S_D ← GetSuppSet(D)
4:          reextract ← CheckSuppInvariant(𝒯_i, S_D)
5:          Δ_t ← ⊥
6:          for all op_j do
7:              if S_D is non-empty then
8:                  N_j[i] ← SynParam(op_j, 𝒯_i, Sim_i, Δ_t)
9:              else
10:                 N_j[i] ← SynCEGIS(op_j, 𝒯_i, Sim_i)
11:             end if
12:         end for
13:     end for
14: end procedure
1:  function SynParam(op_j, 𝒯_i, Sim_i, Δ_t)
2:      k ← 1
3:      R_1 ← op_j ∧ (θ ↔ (𝒯_i(Φ_1, S, W) ≠ 𝒯_i(Φ_2, S, W)))
4:      while true do
5:          if Δ′ found in Δ_t then
6:              Δ ← Fixup(Δ′, op_j)
7:          else
8:              Δ ← Model_(S,W)(S_k ∧ θ)
9:          end if
10:         if Δ = ⊥ then
11:             return Extract(R_k)
12:         end if
13:         O ← Sim_i(Δ)
14:         O_1 ← (𝒯_i(Φ_1, Δ) = O)
15:         O_2 ← (𝒯_i(Φ_2, Δ) = O)
16:         R_{k+1} ← R_k ∧ O_1 ∧ O_2
17:         if O not in Δ_t then
18:             Insert(Δ_t, Δ, O)
19:         end if
20:     end while
21: end function
```

Now consider the template next state function $\mathcal{T}_i$. Suppose for all S_D-variant interpretations I_1 and I_2, $[\![\mathcal{T}_i]\!]_{I_1} = [\![\mathcal{T}_i]\!]_{I_2}$, then we say $\mathcal{T}_i$ is S_D-invariant. $\mathcal{T}_{R_i}$ are all S_D invariant for the definitions given in §III-C.

The key insight is the following: if $\mathcal{T}_i$ is S_D-invariant, given a set of distinguishing inputs $\Delta_1, \Delta_2, \ldots, \Delta_k$, all S_D-variants of these inputs will also prune the search space in the same way (assuming of course that the simulator outputs are the same for each of these inputs). $\mathcal{T}_i$ is S_D-invariant, so it does not "depend" on the interpretation of the predicates op_j. The distinguishing nature of Δ_k is not "affected" by S_D, therefore it can be replaced by an S_D-variant of itself.

The above suggests Algorithm 2. It starts by computing a complete set of supporting expressions S_D using GetSuppSet in line 3. In our current implementation, GetSuppSet returns a set containing bitvector and boolean variables and expressions of the form $read(M, addr)$ occurring in the op_j's. The computation of S_D need not always succeed. In our implementation, we do not handle the case when the op-

code involves expressions involving modified memories; *e.g.*, $read(write(M, addr_1), addr_2)$ and failover to SYNCEGIS.[8]

Line 4 checks if the template $\mathcal{T}_i$ is S_D-invariant for the S_D we computed. This is stored as the flag $reextract$. It is important to note that even if $\mathcal{T}_i$ is not S_D-invariant, we can still speculatively reuse distinguishing inputs. However, in this case, we do need to verify that $R_k \wedge \theta$ is unsatisfiable when we reach the leaf node of the DIT. We then use either SYNPARAM or SYNCEGIS (if computation of S_D failed) to compute the $N_j[i]$ for each op_j.

Line 5 of SYNPARAM checks if we already have a distinguishing input in the DIT Δ_t. If so, we find a S_D-variant interpretation such that $\Delta \models op_j$. This is done by the procedure FIXUP in line 6. FIXUP finds a model for op_j and then performs a syntactic substitution on the distinguishing input Δ' in the tree. If we have reached a leaf node in the tree, we use EXTRACT to get the result of the synthesis. In most cases, EXTRACT just returns the function stored in the leaf node of the tree. However, if $\mathcal{T}_i$ was not S_D-invariant ($reextract = false$) or if we are on a new path in the tree, EXTRACT uses the SMT solver to compute the result. tree. Note even if we are just following an existing path in the DIT, we update the formula R_k in lines 14-16. This is just the construction of syntax trees for these formulas. The SMT solver is not used to compute distuishing inputs using the R_k unless we diverge from the outputs stored in the tree. Procedure INSERT adds a new output node to the DIT.

V. ILA CORRECTNESS AND VERIFICATION

In this section, we discuss correctness of the ILA synthesized by the algorithms presented in Sections III and IV, describe why additional verification of the ILA may be necessary and then describe how this verification is done.

A. Correctness of Synthesized ILA

The template next state function $\mathcal{T}_i$ represents a *family* of possible next state functions. The synthesis algorithms pick a function from this family consistent with the I/O relations exhibited by the simulator Sim. The synthesized result N is guaranteed to be correct if the next state function implemented by the simulator Sim is one of functions represented by the family $\mathcal{T}_i$. We now formalize this notion of correctness.

1) Template Bugs: Consider the ILA A $= \langle S, W, F_o, F_v, D, N, L^\mu \rangle$ and the template next state function $\mathcal{T}_i$. We say that $\mathcal{T}_i$ *can express* N if for all state elements $S[i]$ and each op_j, there exists Φ^i_j such that $op_j \implies \mathcal{T}_i(\Phi^i_j, S, W) = N_j[i](S, W)$.

We refer to the scenario when $\mathcal{T}_i$ *cannot express* N as a template bug because this occurs when the template next state function $\mathcal{T}_i$ has not been constructed correctly by the verification engineer. A template bug may result in the SMT solver returning an unsatisfiable result when attempting to find a distinguishing input. When this happens, our synthesis framework prints out the unsat core of R_k. In our experience, examining the simulation inputs and outputs present in the unsat core is sufficient to identify the bug.

Unfortunately, a unsatisfiable result from the SMT solver is not guaranteed if $\mathcal{T}_i$ cannot express N. In such a scenario, the algorithm may also return an incorrect transition relation and this will be discovered when verifying the ILA (see §V-B1).

2) Simulator Bugs: Since Sim models a simulator and real-world simulators may contain bugs, it is possible that Sim is not equivalent to the idealized transition relation N, *i.e.*, $Sim(S, W) = N(S, W)$ does not hold for all S and W. This will also either cause an unsatisfiable result or an incorrect transition relation. The former can be debugged using the unsat core of R_k while the latter will be detected during verification.

3) Correctness of Synthesis: In the absence of template and simulator bugs, we have the following result about correctness of the synthesized next state functions.[9]

(Theorem 1) If $\mathcal{T}_i$ can express N and $\forall S, W : Sim(S, W) = N(S, W)$ then for each state element $S[i]$ and for each op_j, Algorithms 1 and 2 will terminate with result Φ^i_j and $\forall S, W : op_j \implies \mathcal{T}_i(\Phi^i_j, S, W) = N_j[i](S, W)$.

B. Verification of Synthesized ILAs

Once we have an ILA, the next step is to verify that it correctly abstracts the hardware implementation. This is required because Theorem 1 only guarantees correctness of synthesis in the absence of template and simulator bugs. For strong guarantees of correctness, including correctness in the presence of potential template and simulator bugs, we need to verify correctness of the synthesized ILA against the RTL hardware implementation.

1) Verifying Abstraction Correctness: For state variables that model outputs of hardware components, we expect that ILA outputs always match implementation outputs. In this case, refinement relations are of the form $\mathbf{G}(x_{ILA} = x_{RTL})$.

However, if we are considering internal state variables of hardware components, the above property is likely to be false. For example, consider a pipelined microprocessor with branch prediction. The processor may mispredict a branch and execute "wrong-path" instructions. Although these instructions will eventually be flushed, while they are being executed registers in the RTL will contain the results of these "wrong-path" instructions and so x_{RTL} will not match x_{ILA}. Therefore, we consider refinement relations of the following form: $\mathbf{G}(cond_{ij} \implies x_{ILA} = x_{RTL})$ [25]. The predicate $cond_{ij}$ specifies when the equivalence between state in the ILA and the corresponding state in the implementation holds; *e.g.*, in a pipelined microprocessor, we might expect that when an instruction commits, the architectural state of the implementation matches the ILA.

2) Compositional Verification: Defining the refinement relations as above allows compositional verification [23]. Consider the property $\neg(\phi \mathbf{U} (cond_{ij} \wedge (x_{ILA} \neq x_{RTL})))$ where ϕ states that all refinement relations hold until time $t-1$. This is equivalent to $\mathbf{G}(cond_{ij} \implies x_{ILA} = x_{RTL})$, but we can now abstract away irrelevant parts of ϕ when proving equivalence

[8]Note, this restriction only applies to the opcode (F_o from §II), not to the next state function. This corner case does not occur in any of the designs in our evaluation.

[9]The absence of template and simulator bugs corresponds to the notion of a *valid structure hypothesis* in the terminology of [32].

of x_{ILA} and x_{RTL}. For example, when considering op_j, we can abstract away the implementation of other opcodes $op_{j'}$ and assume these are implemented correctly. This simplifies the model and verifies each opcode separately.

3) Examples of Refinement Relations: One part of our case study is a pipelined microcontroller with limited speculative execution. Here the refinement relations are of the form $\mathbf{G}(inst\_finished \implies (x_{ILA} = x_{RTL}))$. These relations state that the ILA state variables and implementation state variables must match when each instruction completes.

The other part of our case study involves the verification of two cryptographic accelerators. Here the refinement relations are of the following form: $\mathbf{G}(hlsm\_state\_changed \implies x_{ILA} = x_{RTL})$. The predicate $hlsm\_state\_changed$ is $true$ whenever the high-level state machine in the accelerator changes state. This refinement relation states that the high-level state machines of the ILA and RTL have the same transitions. The RTL state machine also has some "low-level" states but such states do not exist in the ILA and are not visible to the firmware, and hence do not need to match ILA state.

4) Verification Correctness: If we prove the refinement relations for all outputs of the ILA and implementation, then we know that the ILA and implementation have identical externally-visible behavior. Hence any properties proven about the behavior of the external inputs and outputs of the ILA are also valid for the implementation.

Proving the property $\mathbf{G}(x_{ILA} = x_{RTL})$ for all external outputs may not be scalable, so we adopt McMillan's compositional approach. We prove refinement relations of the form $\neg(\phi\ \mathbf{U}\ (cond_{ij} \wedge x_{ILA} \neq x_{RTL}))$ for internal state and use these to prove the equivalence of the outputs.

If such compositional refinement relations are proven for all firmware-visible state in the ILA and implementation, this shows that all firmware-visible state updates are equivalent between the ILA and the implementation. Further, transitions of the high-level of state machines in the ILA are equivalent to those in the implementation. This guarantees that firmware/hardware interactions in the ILA are equivalent to the implementation, thus ensuring correctness of the abstraction.

VI. Evaluation

This section presents an evaluation of the proposed methodology and algorithms. We describe the evaluation methodology, the example SoC used as a case study, and then presents the synthesis and verification results.

A. Methodology

This section describes our implementation and the libraries and tools used, the structure of the example SoC, its firmware programming interface and our verification objectives.

1) Implementation Details, Tools and Libraries: Our synthesis framework (box 2 in Figure 1) was implemented in C++ using the Z3 SMT solver [7]. The synthesis framework can be invoked using a domain-specific language (DSL) embedded in Python. This DSL is used to describe ILAs and template next state functions (box 1 in Figure 1). ILA verification (box 6 in Figure 1) was done using ABC's hardware model checker [40]. ABC performs verification on gate-level netlists, while the RTL description (box 5 in Figure 1) of our SoC is in behavioral Verilog. We used a modified version of Yosys [44] to synthesize netlists from behavioral Verilog. Experiments were run on a machine with an Intel Xeon E3-1230 CPU and 32 GB of RAM. Our synthesis framework, templates and synthesized ILAs are available at [36].

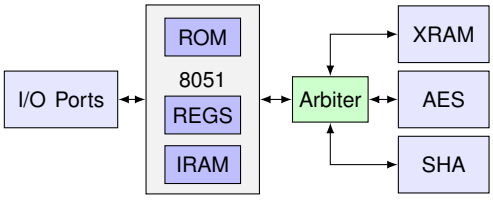

Fig. 5: Example SoC Block Diagram

2) Example SoC Structure: We evaluate this methodology using an SoC design consisting of the 8051 microcontroller and two cryptographic accelerators. A block diagram of design is shown in Figure 5. The RTL (Verilog) implementation of the 8051 is from OpenCores.org [41]. We used *i8051sim* from UC Riverside as a blackbox instruction-level simulator of the 8051 [24]. One accelerator [19] implements encryption/decryption using the Advanced Encryption Standard (AES) [11]. The second accelerator [34] implements the SHA-1 cryptographic hash function [12]. We wrote interface modules that exposed the AES and SHA-1 accelerators to the 8051 using a memory-mapped I/O interface. The accelerators and microcontroller share access to the XRAM, which stores input and output data for the accelerators. We implemented high-level simulators in Python for the two accelerators.

3) Firmware Programming Interface: Firmware running on the 8051 configures the accelerators by writing to memory-mapped registers. Operation is started by writing to the start register which is also memory-mapped. The accelerators use direct memory access (DMA) to fetch the data from the external memory (XRAM), perform the operation and write the result back to XRAM. Firmware determines completion by polling a memory-mapped status register.

4) Verification Objectives: In this work we focus on producing a verified ILA of the SoCs hardware components. The objectives here are to verify that: (i) each instruction in the 8051 is executed according to the ILA, (ii) firmware programming the cryptographic accelerators by reading/writing to appropriate memory-mapped registers produces the expected results, and (iii) ensure that implementation of the cryptographic accelerators matches the high-level state machines in the ILA. We do not verify correctness of encryption/hashing and model these as uninterpreted functions.

B. Synthesis and Verification of the 8051 ILA

This section describes synthesis of the 8051 ILA from its template and verification of the ILA against RTL.

1) Synthesizing the 8051 ILA: We constructed a template ILA of the 8051 which models all opcodes and elements of architectural state. We used *i8051sim* as the blackbox simulator. Note this is equivalent to synthesizing the instruction

set architecture (ISA) of the 8051. Our methodology ensures that the constructed ILA specification is precisely-defined and correct; this is a significant challenge in practice. For example, Godefroid *et al.* [14] report that ISA documents only partially define some instructions and leave some state undefined. They also report instances where implementation behavior contradicts the ISA document and cases where implementation behavior changes between different generations of the same processor-family. Our methodology avoids these pitfalls.

Model	LoC	Size
Template ILA	≈ 500	22 KB
C++ instruction-level simulator	≈ 3000	106 KB
Behavioral Verilog implementation	≈ 9600	360 KB

TABLE I: Lines of code (LoC) and size in bytes of each model.

As one indication of the effort involved in building the model, Table I compares the size of the template ILA with the simulator and the RTL implementation. The template ILA is smaller than the high-level simulator (i8051sim) by a factor of 5. Note the simulator is much smaller than the RTL. This supports our claim that ILAs can be synthesized with lesser effort than manual construction.

Figure 6(a) shows the execution times for synthesis of each element of architectural in the ILA for the 8051. The blue bars show the execution time for Algorithm 1 (SYNCEGIS) while the yellow bars show the execution time for Algorithm 2 (SYNPARAM), which improves upon SYNCEGIS using the distinguishing-input tree. Note the y-axis is in log-scale. We see that for the challenging synthesis problems, *e.g.*, the IRAM, SYNPARAM is about 18× faster than SYNCEGIS. Similarly for PSW, SYNPARAM is about 2× faster. Overall, SYNPARAM is significantly faster than SYNCEGIS, with speedup increasing for challenging instances. Average and geometric mean speedups of SYNPARAM over SYNCEGIS are 2.6× and 2.0× respectively.

2) Monolithic Verification of 8051 ILA: We first attempted to verify the 8051 by generating a large monolithic Verilog model from the ILA that implemented the entire functionality of the processor in a single cycle. The IRAM in this model was abstracted from a size of 256 bytes to 16 bytes. This abstracted model was generated automatically using the synthesis library. We manually implemented the abstraction reducing the size of the IRAM in the RTL implementation.

We used this model to verify properties of the form $\mathbf{G}(inst\_finished \implies x_{ILA} = x_{RTL})$. For the external outputs of the processor, e.g., the external ram address and data outputs, the properties were of the form $\mathbf{G}(output\_valid \implies x_{ILA} = x_{RTL})$. Verification was initially done using bounded model checking (BMC) with ABC using the `bmc3` command. After fixing some bugs and disabling the remaining (17) buggy instructions, we were able to reach a bound of 17 cycles after 5 hours of execution.

3) Compositional Verification of 8051 ILA: To improve scalability, we generated a set of "per-instruction" models which only implement the state updates for one of the 256 opcodes, the implementation of the other 255 opcodes is abstracted away. We then verified a set of properties of the form: $\neg(\phi\ \mathbf{U}(inst\_finished \wedge opcode = o_i \wedge x_{ILA} \neq x_{RTL}))$. Here ϕ states that all architectural state matches until time $t-1$. We then attempted to verify five important properties stating that: (i) PC, (ii) accumulator, (iii) the IRAM, (iv) XRAM data output and (iv) XRAM address must be equal for the ILA and the implementation.

Property	BMC bounds					Proofs
	CEX	≤ 20	≤ 25	≤ 30	≤ 35	
PC	0	0	25	10	204	96
ACC	1	0	8	39	191	56
IRAM	0	0	10	36	193	1
XRAM/dataout	0	0	0	0	239	238
XRAM/addr	0	0	0	0	239	239

TABLE II: Results with per-instruction model.

Results for these verification experiments are shown in Table II. Each row of the table corresponds to a particular property. Columns 2-6 show the bounds reached by BMC within 2000 seconds. For example, the first row shows that for 25 instructions, the BMC was able to reach a bound between 21 to 25 cycles without a counterexample; for 10 instructions, it achieved a bound between 26 to 30 cycles and for the remaining 204 instructions, the BMC reached a bound between 31 and 35 cycles. The last column shows the number of instructions for which we could **prove** the property. These proofs were done using the `pdr` command which implements the IC3 unbounded model checking algorithm [3] with a time limit of 1950 seconds. Before running `pdr`, we preprocessed the netlists using the gate-level abstraction [26] technique with a time limit of 450 seconds.

4) Bugs Found During 8051 Verification: Seven bugs were found in the simulator during ILA synthesis. Bugs in `CJNE`, `DA`, `MUL` and `DIV` instructions were due to signed integers being used where unsigned values were expected. Another was a typo in `AJMP` and the last was a mismatch between RTL and the simulator when dividing by zero.

An interesting bug in the template was for the `POP` instruction. The `POP <operand>` instruction updates two items of state: (1) `<operand> = RAM[SP]` and (2) `SP = SP - 1`. But what if operand is `SP`? The RTL set `SP` using (1) while the ILA used (2). This was discovered during model checking and the ILA was changed to match the RTL. This shows one of the key benefits of our methodology: there are no undefined corner cases and all state updates are precisely-defined and consistent between the ILA and RTL.

In the RTL model, we found a total of 7+1 bugs. One of these is an entire class of bugs related to the forwarding of special function register (SFR) values from an in-flight instruction to its successor. This affects 17 different instructions and all bit-addressable architectural state. We partially fixed this. A complete fix appears to require significant effort. Another interesting issue was due to reads from reserved/undefined SFR addresses. The RTL returned the previous value stored in a temporary buffer which could potentially have security implications and result in unintended leakage of information through undefined state. Various corner-case bugs were found in the `AJMP`, `JB`, `JNB`, `JBC`, `AJMP`, `DA` and `POP` instructions.

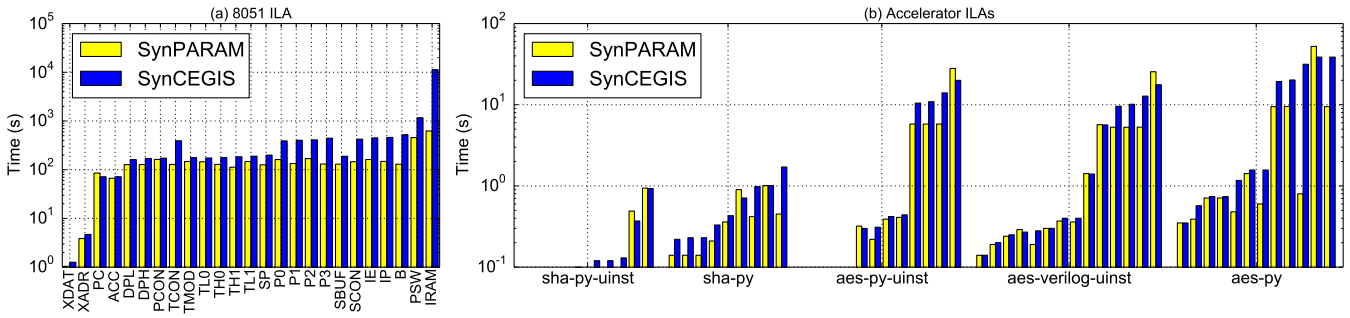

Fig. 6: Execution time: Baseline vs. Parameterized Synthesis.

C. Synthesis and Verification of Accelerator ILAs

We constructed 5 ILAs for the accelerators. One ILA for the AES accelerator is a monolithic ILA (aes-py), the two others are hierarchical and contain a macroILA that responds to commands from the processor core and a microILA for encryption state machine. These hierarchical ILAs were synthesized using a high-level python simulator (aes-py-uinst) and from the RTL (aes-verilog-uinst). Similarly, two SHA-1 ILAs were synthesized: a monolithic ILA (sha1-py) and a hierarchical ILA containing a microILA similar to the AES accelerator (sha1-py-uinst).

Model	Template Size		Simulator Size	
	LoC	kB	LoC	kB
aes-py	163	5.4	225	6.5
aes-py-uinst	176	5.5	235	7.6
aes-rtl-uinst	203	7.1	1905	58
sha-py	126	4.7	207	6.5
sha-py-uinst	157	5.4	231	7.1

TABLE III: Lines of code and size of each model.

1) Synthesis Results: Table III compares the sizes of the template ILA with the simulators. The template ILA is again smaller in size than the simulator, but the difference in size is not as pronounced as with the 8051. This is mainly because the accelerators are simpler than the 8051 and so the python simulators constructed for them are also small. However, these results again demonstrate that ILAs can be synthesized for non-trivial accelerators fairly easily.

Figure 6(b) shows the execution time for the two synthesis algorithms – SYNCEGIS and SYNPARAM. Except for a few outliers, SYNPARAM is faster than SYNCEGIS with average and geometric mean speedups of 2.1× and 1.4× respectively. These synthesis instances are easier those for the 8051 and so the potential speedup is lower.

2) Verifying Accelerator ILAs: To simplify verification, we reduced the size of the XRAM to just one byte as we were not looking to prove correctness of reads and writes to XRAM. We then examined set of properties of the form $\mathbf{G}(hlsm\_state\_change \implies (x_{ILA} = x_{RTL}))$. We were able to prove that the *AES:State*, *AES:Addr*, and *AES:Len* in the implementation matched the ILA using the `pdr` command. For other firmware-visible state, BMC found no property violation up to 199 cycles with a time limit of one hour.

D. Scaling ILA-Based Verification to Larger Designs

Experimental results in this paper and the case study in [37] show that ILAs can be constructed for non-trivial designs. We now discuss the challenges in applying this methodology on larger SoCs. Our methodology consists of two parts: synthesis and verification. A complex processor, such as an x86 processor, has thousands of instructions and hundreds of architectural state variables. Constructing a template for such a processor will be challenging. However, this is known to be a difficult problem and [14, 17] have shown that synthesis is very helpful in constructing models of ISAs.

Turning to verification, while a more complex processor would indeed be harder to verify, the ILA does not add new additional complexity here. If the design is too large for formal verification, techniques like randomized testing and simulation-based verification may be used. Since the ILA is a precise machine-readable description of SoC hardware, it is amenable to such semi-formal verification techniques.

VII. RELATED WORK

Abstraction Synthesis: We build on recent progress in syntax-guided synthesis [1, 33]. Our synthesis algorithm is based on oracle-guided synthesis [21], the theoretical underpinnings of which are studied in [22]. Our contribution is the application of synthesis to constructing abstractions for SoC verification and the parameterized formulation which makes ILA synthesis tractable. Godefroid et al. applied Oracle-guided synthesis to construct a model for a subset of x86 ALU instructions [14]. Heule et al. [17] also tackled the same problem but combined stochastic search techniques with modern constraint solvers. Both [14] and [17] require processor-specific knowledge of the opcode format and argument format and associated manual effort to encode instruction functionality in templates. This manual effort may be acceptable when building a single model, such as the target of their work: part of an x86 CPU. Unlike [14] and [17], we are interested in constructing complete ILAs for diverse accelerators and processor cores and repetitive manual analysis can be a significant bottleneck in this.

Processor Modeling and Verification: Formal modeling of ISAs for processors is now a well-studied topic. An early effort was the construction of a specification for and formal verification of the FM8501 microprocessor by Hunt [20]. More recently, Fox and Myreen [13] as well as the ISA-Formal

project at ARM have constructed formal specifications of ARM ISAs [29, 30]. Goel and colleagues [15] constructed a specification of both user-level and system-level instructions in the x86 ISA. All these specifications can be used to reason about software and also to verify that hardware correctly implements the ISA. While our goals for the ILA are similar, we wish to go beyond modeling programmable cores and also model application-specific accelerators. A second difference is our use of synthesis for semi-automatic construction.

The *refinement relations* we use in proving that the abstraction and the implementation match are based on the refinement relations for processor verification presented in [23, 25]. Also helpful in our verification effort were techniques for memory modeling and abstraction in model checking, such as Velev's memory model [43]. While these verification techniques are very important, these are not the focus of our paper. We focus on *synthesizing* abstractions. To verify their correctness, we can leverage the rich body of work in hardware verification.

SoC Verification: A number of efforts have studied transaction-level modeling (TLM) of SoCs using SystemC [4, 16, 28, 39, 42] and the Spec-C language [9]. A key difference between ILAs and TLMs is that ILAs seek to precisely delineate the HW/FW interface while showing refinement between the ILA and SoC hardware. Both of these remain challenging with TLMs. Also, ILA synthesis can help construct models bottom-up for existing legacy SoC IPs.

Although many studies in recent years have investigated the problems of firmware and hardware verification, most of these studies have typically focused on separate verification of hardware and firmware. Examples include [2, 6, 18, 31], all of which use symbolic execution to analyze firmware. These efforts do not address co-verification of hardware and firmware, a critical requirement for SoC verification. One approach to compositional SoC co-verification of hardware and firmware is by Xie *et al*. [45, 46] which involves the construction of "bridge" specifications. Our methodology makes it easy to construct the equivalent of the bridge specifications while also ensuring this specification (abstraction) is correct.

VIII. Conclusion

Modern SoCs consist of programmable cores, accelerators and peripheral devices as well as firmware running on the programmable cores. Functionality of the SoC is derived by a combination of firmware and hardware. Verifying such SoCs is challenging because formally verifying a unified SoC description with firmware and hardware is not scalable, while verifying the two components separately may miss bugs.

In this paper, we introduced a methodology for SoC verification based on *synthesizing instruction-level abstractions* (ILA) of SoCs. The ILA captures updates to all firmware-accessible states in the SoC and can be used instead of the bit-precise cycle-accurate hardware model while proving system-level properties involving firmware and hardware. One advantage of our methodology is that the ILA is verifiably correct: we prove that the behavior of the ILA matches the implementation. Another advantage is that instead of specifying the complete ILA, the verification engineer has an easier task of writing a template ILA which partially defines the operation of the hardware components, and our synthesis algorithm reconstructs the missing details. We demonstrated the applicability of our methodology by using it to verify a small SoC consisting of the 8051 microcontroller and two cryptographic accelerators. The verification process uncovered several bugs substantiating our claim that the methodology is effective.

References

[1] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design*, pages 1–8, October 2013.

[2] O. Bazhaniuk, J. Loucaides, L. Rosenbaum, M. R. Tuttle, and V. Zimmer. Symbolic Execution for BIOS Security. In *Proceedings of the 9th USENIX Conference on Offensive Technologies*, 2015.

[3] A. R. Bradley. SAT-Based Model Checking without Unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 70–87, January 2011.

[4] M. Caldari, M. Conti, M. Coppola, S. Curaba, L. Pieralisi, and C. Turchetti. Transaction-Level Models for AMBA Bus Architecture Using SystemC 2.0. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 26–31, March 2003.

[5] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman. Accelerator-rich architectures: Opportunities and progresses. In *Proceedings of the 51st ACM/IEEE Design Automation Conference*, pages 1–6, June 2014.

[6] D. Davidson, B. Moench, S. Jha, and T. Ristenpart. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proceedings of the 22nd USENIX Conference on Security*, pages 463–478, 2013.

[7] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, March 2008.

[8] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, October 1974.

[9] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. D. Gajski. System-on-Chip Environment: A SpecC-Based Framework for Heterogeneous MPSoC Design. *EURASIP Journal of Embedded Systems*, pages 5:1–5:13, June 2008.

[10] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, pages 365–376, June 2011.

[11] National Institute for Standards and Technology. Federal Information Processing Standards Publication 197: Announcing the Advanced Encryption Standard. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, November 2001.

[12] National Institute for Standards and Technology. Federal Information Processing Standards Publication 180-2: Announcing the Secure Hash Standard. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf, August 2002.

[13] A. C. J. Fox and M. O. Myreen. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *First International Conference on Interactive Theorem Proving*, pages 243–258. Springer, July 2010.

[14] P. Godefroid and A. Taly. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 441–452, June 2012.

[15] S. Goel, W. A. Hunt Jr., M. Kaufmann, and S. Ghosh. Simulation and formal verification of x86 machine-code programs that make system calls. In *Proceedings of the 14th International Conference on Formal Methods in Computer-Aided Design*, pages 91–98. IEEE, October 2014.

[16] C. Helmstetter and O. Ponsini. A Comparison of Two SystemC/TLM Semantics for Formal Verification. In *6th ACM & IEEE International Conference on Formal Methods and Models for Co-Design*, pages 59–68, June 2008.

[17] S. Heule, E. Schkufza, R. Sharma, and A. Aiken. Stratified Synthesis: Automatically Learning the x86-64 Instruction Set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 237–250, June 2016.

[18] A. Horn, M. Tautschnig, C. Val, L. Liang, T. Melham, J. Grundy, and D. Kroening. Formal co-validation of low-level hardware/software interfaces. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design*, pages 121–128, October 2013.

[19] H. Hsing. OpenCores.org Tiny_AES project page. http://opencores.org/project,tiny_aes, 2014.

[20] W. A. Hunt Jr. *FM8501: A Verified Microprocessor*. 1994.

[21] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided Component-based Program Synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 215–224, May 2010.

[22] S. Jha and S. A. Seshia. A Theory of Formal Synthesis via Inductive Learning. *CoRR*, abs/1505.03953, 2015.

[23] R. Jhala and K. L. McMillan. Microarchitecture Verification by Compositional Model Checking. In *13th International Conference on Computer-Aided Verification*, pages 396–410, July 2001.

[24] R. Lysecky, T. Givargis, G. Stitt, A. Gordon-Ross, and K. Miller. Intel 8051 Simulator. http://www.cs.ucr.edu/~dalton/i8051/i8051sim/, 2001.

[25] K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *11th IFIP Conference on Correct Hardware Design and Verification Methods*, pages 179–195. September 2001.

[26] A. Mishchenko, N. Een, R. Brayton, J. Baumgartner, H. Mony, and P. Nalla. GLA: Gate-level Abstraction Revisited. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1399–1404, 2013.

[27] M. D. Nguyen, M. Wedler, D. Stoffel, and W. Kunz. Formal Hardware/Software Co-verification by Interval Property Checking with Abstraction. In *Proceedings of the 48th Design Automation Conference*, pages 510–515, 2011.

[28] P. R. Panda. SystemC - a modeling platform supporting multiple design abstractions. In *The 14th International Symposium on System Synthesis*, pages 75–80, 2001.

[29] A. Reid. Trustworthy specifications of ARMv8-A and v8-M system level architecture. In *Proceedings of 16th International Conference on Formal Methods in Computer-Aided Design*, pages 161–168. IEEE, 2016.

[30] 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 processors with ISA-Formal. In *Proceedings of the International Conference on Computer Aided Verification*, pages 42–58. Springer, 2016.

[31] B. Schmidt, C. Villarraga, T. Fehmel, J. Bormann, M. Wedler, M. Nguyen, D. Stoffel, and W. Kunz. A New Formal Verification Approach for Hardware-dependent Embedded System Software. *IPSJ Transactions on System LSI Design Methodology*, 6:135–145, 2013.

[32] S. A. Seshia. Combining Induction, Deduction, and Structure for Verification and Synthesis. *Proceedings of the IEEE*, 103(11):2036–2051, 2015.

[33] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference Architectural Support for Programming Languages and Operating Systems*, pages 404–415, 2006.

[34] J. Strömbergson. https://github.com/secworks/sha1, 2014.

[35] P. Subramanyan and D. Arora. Formal Verification of Taint-Propagation Security Properties in a Commercial SoC Design. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1–2, 2014.

[36] P. Subramanyan, B. Huang, Y. Vizel, A. Gupta, and S. Malik. Experimental artifacts and synthesis framework source code. https://bitbucket.org/spramod/tcad-ila-synthesis, 2015.

[37] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung. Verifying Information Flow Properties of Firmware using Symbolic Execution. In *Proceedings of the Conference on Design Automation and Test in Europe*, pages 337–342, 2016.

[38] P. Subramanyan, Y. Vizel, S. Ray, and S. Malik. Template-based Synthesis of Instruction-Level Abstrac-

tions for SoC Verification. In *Proceedings of the 15th International Conference on Formal Methods in Computer-Aided Design*, pages 160–167, September 2015.

[39] S. Swan. SystemC Transaction Level Models and RTL Verification. In *Proceedings of the 43rd Annual Design Automation Conference*, pages 90–92, 2006.

[40] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/~alanmi/abc/, April 2015.

[41] S. Teran and J. Simsic. http://opencores.org/project,8051, 2013.

[42] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi. A SystemC/TLM Semantics in Promelaand Its Possible Applications. In *Proceedings of the 14th International SPIN Workshop Model*, pages 204–222, July 2007.

[43] M. N. Velev, R. E. Bryant, and A. Jain. Efficient Modeling of Memory Arrays in Symbolic Simulation. In *Proceedings of the 9th International Conference on Computer Aided Verification*, pages 388–399, June 1997.

[44] C. Wolf. http://www.clifford.at/yosys/, 2015.

[45] F. Xie, X. Song, H. Chung, and R. N. Translation-based Co-verification. In *3rd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, pages 111–120, July 2005.

[46] F. Xie, G. Yang, and X. Song. Component-based Hardware/Software Co-verification for Building Trustworthy Embedded Systems. *Journal of System Software*, 80(5):643–654, May 2007.