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

Pramod Subramanyan, Yakir Vizel, Sayak Ray and Sharad Malik
Department of Electrical Engineering, Princeton University.

***Abstract*—Contemporary integrated circuits are complex system-on-chip (SoC) designs consisting of programmable cores along with accelerators and peripherals controlled by firmware running on the cores. The functionality of the SoC is implemented by a combination of firmware and hardware components. As a result, verifying these two components separately can miss bugs while attempting to formally verify the full SoC design considering both firmware and hardware is not scalable.**

An abstraction that can be used instead of the cycle-accurate and bit-precise hardware implementation can be helpful in scalably verifying system-level properties of SoCs. However, constructing such an abstraction to capture all the required details and interactions is error-prone, tedious and time-consuming. Another challenge is ensuring correctness of the abstraction so that properties proven using it are valid.

In this paper, we introduce a methodology for SoC verification. We *synthesize* an *instruction-level abstraction* (ILA) that precisely captures updates to all firmware-accessible states spanning the cores, accelerators and peripherals. The synthesis algorithm uses a blackbox simulator to synthesize the ILA from a *template* specification. A "golden-model" generated from the ILA is used to verify whether the hardware implementation matches the ILA. We demonstrate the methodology using a small SoC design consisting of the 8051 microcontroller and two cryptographic accelerators. The methodology uncovered 14 bugs.

I. Introduction

Today's integrated circuits are complex system-on-chip (SoC) designs consisting of one or more programmable cores, several accelerators and peripheral devices [17]. The overall functionality of the SoC is determined by *firmware* that runs on the cores and orchestrates the operation of the accelerators and peripheral devices. Attempting to formally verify the complete SoC with all its hardware components and firmware is not scalable for even very small designs.

Firmware sits "below" the operating system and interacts closely with the hardware. Both firmware and hardware make many assumptions about the behavior of the other component. As a result, verifying the two components separately requires explicitly enumerating these assumptions and verifying that the other component satisfies these assumptions. An example from a commercial SoC highlighting the importance of capturing these interactions is provided in [21]. A series of I/O write operations could be executed by malicious firmware leaving a cryptographic accelerator in a "confused" state after which sensitive cryptographic keys could be exfiltrated. The bug was due to certain implicit assumptions made by hardware about the timing of firmware I/O writes. These were violated by the malicious code sequence.

This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

A. Abstractions for SoC Verification

A general technique for making SoC verification tractable is to use an abstraction that accurately models all updates to firmware-accessible hardware states [9, 16, 25, 26]. When verifying properties involving firmware, the abstraction is used instead of the bit-precise cycle-accurate hardware model.

Although the idea of constructing abstractions for firmware verification is attractive, there are several challenges in applying the technique in practice. Firmware interacts with hardware components in a myriad of ways. For the abstraction to be useful, it needs to model all these interactions and capture all updates to firmware-accessible states.

- 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, the 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-accessible temporary register.

We argue that manually constructing an abstraction which captures these details, as proposed for example in [25, 26], 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 [16], can ease certain verification concerns, but this does not capture all the requirements mentioned above. A third alternative is to verify the firmware using a software model of the hardware [9]. This too misses bugs present in the hardware implementation but not the software model.

The problem with these approaches is correctness of the abstraction. If the hardware implementation is not consistent with the abstraction, properties proven using it are not valid.

B. Synthesizing Instruction-Level Abstractions

In this paper, we propose a general methodology for constructing correct abstractions for SoC verification. The abstrac-

ISBN: 978-0-9835678-5-1.

tion captures all updates to firmware-accessible states which includes the architectural state of the cores, memory-mapped and I/O addressable registers in the accelerators and peripheral devices as well as high-level state machines that model the operation of the cores and other hardware components.

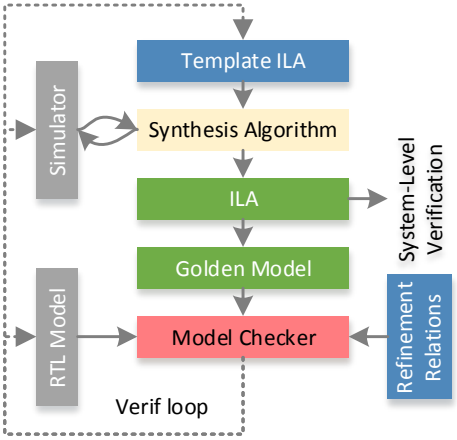

Fig. 1: Block Diagram of Template-Based Synthesis of Instruction-Level Abstractions

We call this an *instruction-level abstraction* or *ILA*. The insight is that firmware can only view changes in system state at the granularity of instructions. So it is sufficient to model hardware components of the SoC at this granularity. Then, the ILA of an SoC is a product of deterministic finite state transition systems that are abstractions of each of the SoCs hardware components constructed at the granularity of instructions. For example, Figure 2 shows an ILA that is a product of three finite state transition systems: a processor, accelerator and an I/O peripheral. The ILA for the processor is analogous to an instruction-granularity control flow graph, while for the accelerator it is an instruction-granularity high-level state machine. If we construct an ILA and prove it is an overapproximation of the hardware components, system-level properties proven using the ILA will be valid.

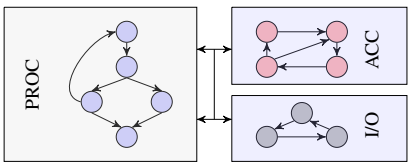

Fig. 2: Instruction-Level Abstraction

To enable the abstraction to be easily constructed in a semi-automated manner, we build on recent progress in syntax-guided synthesis [1, 11, 19]. We propose *synthesizing the ILA* from a *template*. Instead of manually constructing the complete abstraction, the verification engineer now has the much easier task of writing a *template* that partially defines the operation of the hardware components. The synthesis framework infers the complete abstraction and fills in the missing details by using a *blackbox simulator* of the hardware components.

The term *blackbox* simulator means the simulator can be used to find the next state and outputs of the system given its current state and input values, but it is not possible to "look inside" the simulator and get a full-definition of the system's behavior.[1] Simulators are often constructed during SoC design for validation purposes, e.g., simulation-based testing of firmware. In principle, it may be possible to extract an abstraction of the SoC through automated analysis of the simulator, however, in practice, due to the scale and complexity of the codebase it is not possible to do so. Our work constructs an abstraction of the system in this scenario.[2]

To validate the abstraction and ensure that the hardware implementation conforms to the abstraction, we automatically generate a "golden model" from the abstraction. A set of temporal refinement relations are model checked to ensure that the behavior of the implementation matches the behavior of the golden model. If the refinement relations are proven, we have a guarantee that the abstraction is a correct over-approximation of the hardware components 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.

Figure 1 is an overview of the methodology. The blue-boxes show the components that are provided by the verification engineer. We assume that the register-transfer level (RTL) model and a simulator are already available; these are gray. Automatically generated artifacts are green and off-the-shelf tools are red. The synthesis algorithm is in yellow.

C. 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 parameterized synthesis framework that allows scalable synthesis of the complex functionality in modern SoCs and a language for template-based synthesis that is tailored to modeling hardware components. We show how correctness of the synthesized abstraction can be verified.

Finally, we present a case study applying the methodology to the verification of a simple SoC design consisting of the 8051 microcontroller and two cryptographic accelerator cores. We discuss construction of the instruction-level abstraction and describe the bugs found during verification. The synthesis framework and experimental artifacts are available online [5].

[1]The *blackbox* simulator is akin the I/O oracle in [11].

[2]The hardware (RTL) implementation can also be used for simulation if a dedicated simulator is not available.

ISBN: 978-0-9835678-5-1.

II. Definitions and Formal Model

We model the hardware implementation as a deterministic finite state transition system. Let $\mathbb{B} = \{0, 1\}$ be the Boolean domain. The state space is defined by the union of two sets of Boolean variables encoding the states: $X = X_F \cup X_M$. $X_F = \{x_1, x_2, \ldots, x_m\}$ represent the firmware-accessible states. $X_M = \{u_1, u_2, \ldots, u_n\}$ is the microarchitectural state, and is not visible to the firmware. For example, in a microprocessor core, X_F will contain the architectural registers and program counter while X_M may contain the pipeline registers and reorder buffer. The transition system is then defined as the tuple $M = (X, I, Init, T)$. I is the set of external inputs to the transition system. $Init$ is a predicate over X and defines the initial states of the transition system. $T(I, X, Y)$ defines the transition relation where Y is the set of next-state variables.

The instruction-level abstraction is also modeled as a deterministic finite state transition system. The state space of the abstraction is defined over the set of variables X_A. All firmware-accessible states are included in X_A so $X_F \subseteq X_A$. The transition system is then defined by the tuple $M_A = (X_A, I_A, Init_A, T_A)$ where I_A, $Init_A$ and $T_A(I_A, X_A, Y_A)$ are analogously defined. We also define the blackbox simulation function $eval : I \times X_A \mapsto Y_A$. $eval(\mathtt{I}, \mathtt{X_A}) = \mathtt{Y_A}$ iff $T_A(\mathtt{I}, \mathtt{X_A}, \mathtt{Y_A})$ is true. Here $\mathtt{I}$, $\mathtt{X_A}$ and $\mathtt{Y_A}$ are specific values of I, X_A and Y_A respectively.

III. Synthesizing Instruction-Level Abstractions

The synthesis problem is to construct the finite state transition system $M_A = (X_A, I, Init_A, T_A)$ using the blackbox simulation function $eval$. One potential solution to this problem is to use results on learning finite state automata [2, 18]. Unfortunately the running time of these algorithms grows as a polynomial function of the number of states in the system. Since a typical hardware component has 2^m states with m state variables and m is in the range of hundreds, thousands or even more, these algorithms are not practical.

We tackle the problem using two insights. The first is that it is reasonable to expect the verification engineer to identify the state variables of the ILA: X_A. We then build on recent progress in syntax-guided synthesis [1] to synthesize the transition relation T_A from a template. The challenge here is that the transition relation for a hardware component is likely too complex to synthesize directly. For instance, consider the transition relation for the ILA of a microprocessor. The inputs to the relation will be all state the processor can access: all data registers, all memory, all external I/O ports, the program counter, flag register, etc. The relation captures the functionality of each opcode by performing a "case-split" for each opcode, which can take hundreds of different values. Synthesis algorithms are currently limited to templates with a few tens to hundreds of synthesis elements [1, 8, 11]. Since the opcode can take hundreds of different values, synthesizing the complete transition relation appears to be out of reach.

Our solution is to simplify the synthesis problem by eliminating the "case-split" structure. The synthesis framework starts with a template, *i.e.,* a specification containing "holes" [1, 19] and a *synthesis parameter*. Synthesis is done for each value of the parameter and the complete transition relation combines the individually synthesized elements.

A. Synthesis Problem Formulation

To synthesize the ILA, the verification engineer constructs a *template* transition relation $\mathcal{T}_A(S, I, X_A, Y_A)$. S is the set of *synthesis variables* which have to be assigned appropriately to make the template $\mathcal{T}_A$ equivalent to T_A. The synthesis problem is parameterized over the parameter p_i where $i = 1, 2, \ldots N$. p_i is a family of predicates defined on X_A such that $p_1 \vee p_2 \vee \cdots \vee p_N = 1$ and $(i \neq j) \implies \neg(p_i \wedge p_j)$. For each i, the synthesis algorithm uses the function $eval : I \times X_A \mapsto Y_A$ and attempts to find an assignment $\mathtt{S_i}$ to S such that $\forall I, X_A : p_i \implies \big(\mathcal{T}_A(\mathtt{S_i}, I, X_A, Y_A) \iff T_A(I, X_A, Y_A)\big)$. The conjunction of these relations yields the ILA T_A.

A formal definition of the parameterized synthesis problem is as follows. Find $\mathtt{S_1} \ldots \mathtt{S_N}$ such that for all I, X_A, Y_A:

$$\Big(\bigwedge_{i=1}^{N} \big(p_i \implies \mathcal{T}_A(\mathtt{S_i}, I, X_A, Y_A)\big)\Big) \iff T_A(I, X_A, Y_A)$$

Consider the example of synthesizing an ILA for a microprocessor. The template transition relation expresses the different ways in which architectural state can be updated by each instruction. The synthesis parameter is the currently executing opcode, therefore, the predicate p_i would be defined over the ROM values pointed to by the current program counter. The predicate p_0 would be true when the opcode 0 is being executed, predicate p_1 would be true for opcode 1 and so on. Synthesis is done for each opcode and the conjunction $(p_0 \implies \mathcal{T}_A(\mathtt{S_0}, I, X_A, Y_A) \wedge \cdots \wedge (p_N \implies \mathcal{T}_A(\mathtt{S_N}, I, X_A, Y_A))$ defines the operation of the microprocessor under every possible opcode. This is the complete ILA.

B. Template Language Definition

The transition relation is defined using the template language shown in Figure 3. To easily model hardware behavior, a number of primitives to manipulate Boolean and bitvector values are included in the language. It also models memory-like structures (RAM/ROM/register files) and uninterpreted functions from bitvectors to bitvectors.

The template consists of a list of statements. Each statement is either an assignment or an **output** statement. An **output** statement indicates this identifier is one of the outputs of the transition relation. The constructs **bool**, **bv**, **mem** and **func** create Boolean, bitvector, memory and function variables of the appropriate sizes, respectively. A memory variable is specified with two parameters: the bitvector size of the address and the bitvector size of the data cells. A function variable is a map from a bitvector of size `in_width` to a bitvector of size `out_width`. ***bvop*** and ***boolop*** represent all usual bitvector and boolean operators: and, or, not, addition, subtraction etc.

Synthesis is supported using three constructs. The **choice** construct takes a list of expressions as its argument and specifies that the synthesized ILA must replace the choice construct with one of the argument expressions. For example,

⟨*template*⟩ ::= ⟨*stmt*⟩ ; ⟨*template*⟩
  | ⟨*empty*⟩

⟨*stmt*⟩ ::= ⟨*id*⟩ ← ⟨*exp*⟩
  | **output** ⟨*id*⟩

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

⟨*bv-exp*⟩ ::= ⟨*id*⟩ | **bv** `width` | **bvcnst** `value width`
  | ***bvop*** ⟨*bv-exp*⟩ ...
  | **if** ⟨*bool-exp*⟩ **then** ⟨*bv-exp*⟩ **else** ⟨*bv-exp*⟩
  | **readmem** ⟨*mem-exp*⟩ ⟨*bv-exp*⟩
  | **apply** ⟨*func-exp*⟩ ⟨*bv-exp*⟩
  | **choice** ⟨*id*⟩ [⟨*bv-exp*⟩ ⟨*bv-exp*⟩ ...]
  | **read-slice-choice** ⟨*id*⟩ ⟨*bv-exp*⟩ `length`
  | **bv-in-range** ⟨*bv-exp*⟩ ⟨*bv-exp*⟩

⟨*bool-exp*⟩ ::= ⟨*id*⟩ | **bool** | **true** | **false**
  | ***boolop*** ⟨*bool-exp*⟩ ...
  | ⟨*bv-exp*⟩ == ⟨*bv-exp*⟩ | ⟨*bv-exp*⟩ ≠ ⟨*bv-exp*⟩
  | **if** ⟨*bool-exp*⟩ **then** ⟨*bool-exp*⟩ **else** ⟨*bool-exp*⟩
  | **choice** ⟨*id*⟩ [⟨*bool-exp*⟩ ⟨*bool-exp*⟩ ...]

⟨*mem-exp*⟩ ::= ⟨*id*⟩
  | **mem** `addr_width data_width`
  | **write-mem** ⟨*mem-exp*⟩ ⟨*bv-exp*⟩ ⟨*bv-exp*⟩
  | **if** ⟨*bool-exp*⟩ **then** ⟨*mem-exp*⟩ **else** ⟨*mem-exp*⟩
  | **choice** ⟨*id*⟩ [⟨*mem-exp*⟩ ⟨*mem-exp*⟩ ...]

⟨*func-exp*⟩ ::= ⟨*id*⟩
  | **func** ⟨*id*⟩ `in_width out_width`

Fig. 3: Template Language Grammar

suppose we want to model an 8-bit ALU that performs addition, subtraction and the increment operations. This is written as:

$\texttt{ALU}_{\texttt{INC}} \leftarrow \texttt{SRC}_1$ + **bvcnst** 1 8
$\texttt{ALU}_{\texttt{ADD}} \leftarrow \texttt{SRC}_1$ + $\texttt{SRC}_2$
$\texttt{ALU}_{\texttt{SUB}} \leftarrow \texttt{SRC}_1$ - $\texttt{SRC}_2$
$\texttt{ALU}_{\texttt{RESULT}} \leftarrow$ **choice** $\texttt{ALU}_{\texttt{OP}}$ [$\texttt{ALU}_{\texttt{INC}}$ $\texttt{ALU}_{\texttt{ADD}}$ $\texttt{ALU}_{\texttt{SUB}}$]

The **read-slice-choice** has bitvector b and width k as arguments and synthesizes an expression that extracts bits i to $i + k - 1$ of b for some index i. In other words, it provides a convenient way to operate on slices of bitvectors without specifying the indices of the slice. For example, if one of the bits in the PSW register is the carry flag, we can write this as follows: $\texttt{CY} \leftarrow$ **read-slice-choice** $\texttt{CY}_{\texttt{F}}$ PSW 1.

The final synthesis operator is **bv-in-range** which synthesizes a bitvector value within the specified range. Somewhat surprisingly, we found this minimal set of synthesis operators to be sufficiently expressive for our case study.[3] It is very easy to add more synthesis primitives.

C. Synthesis Algorithm

The synthesis procedure first "compiles" the template into a satisfiability modulo theory (SMT) formula which uses the theories of bitvectors, arrays and uninterpreted functions with equality. Most elements in the template language can be mapped to SMT using a straightforward recursive algorithm. The synthesis primitives: **choice**, **read-slice-choice** and **bv-in-range** require special treatment. These primitives introduce new synthesis variables, whose values have to be inferred by the synthesis procedure. As an example consider the **choice** primitive. The statement **choice** `id` [$\texttt{c}_1$ $\texttt{c}_2$ ... $\texttt{c}_\texttt{k}$] is converted to the SMT formula $\text{ITE}(\texttt{id}_1, \texttt{c}_1, \text{ITE}(\texttt{id}_2, \texttt{c}_2, \text{ITE}(\texttt{id}_3, \texttt{c}_3, \ \ldots\ , \texttt{c}_\texttt{k})))$ where $\texttt{id}_1$ ... $\texttt{id}_{\texttt{k}-1}$ are Boolean synthesis variables.

Algorithm 1 Synthesis Algorithm

Function: *synthesize*.
Inputs: $\mathcal{T}_A(S, I, X_A, Y_A)$, p_i and $eval$.
Output: $\texttt{S}_\texttt{i}$

1: $j \leftarrow 1$
2: $\mathcal{T}_1 = \mathcal{T}_A(S_1, I, X_A, Y_{A1})$
3: $\mathcal{T}_2 = \mathcal{T}_A(S_2, I, X_A, Y_{A2})$
4: $F^1 = p_i \wedge \mathcal{T}_1 \wedge \mathcal{T}_2$
5: **while** $sat[F^j \wedge (Y_{A1} \neq Y_{A2})]$ **do**
6: $\quad(\texttt{I}^\texttt{j}, \texttt{X}^\texttt{j}) \leftarrow \text{SATASSIGNMENT}_{I,X_A}(F^j)$
7: $\quad\texttt{Y}^\texttt{j} \leftarrow eval(\texttt{I}^\texttt{j}, \texttt{X}^\texttt{j})$
8: $\quad\text{T}_1^j \leftarrow \text{SUBSTITUTE}(\mathcal{T}_1, I = \texttt{I}^\texttt{j}, X_A = \texttt{X}^\texttt{j}, Y_{A1} = \texttt{Y}^\texttt{j})$
9: $\quad\text{T}_2^j \leftarrow \text{SUBSTITUTE}(\mathcal{T}_2, I = \texttt{I}^\texttt{j}, X_A = \texttt{X}^\texttt{j}, Y_{A2} = \texttt{Y}^\texttt{j})$
10: $\quad F^{j+1} \leftarrow F^j \wedge \text{T}_1^j \wedge \text{T}_2^j$
11: $\quad j \leftarrow j + 1$
12: **end while**
13: $\texttt{S}_\texttt{i} \leftarrow \text{SATASSIGNMENT}_{S_1}(F^j)$

The translation procedure yields the SMT formula $\mathcal{T}_A(S, I, X_A, Y_A)$ which is then synthesized using Algorithm 1. The key idea is to repeatedly find *distinguishing inputs* [11] while ensuring the simulation input/output values observed thus far are satisfied. A distinguishing input for $\texttt{S}_1$ and $\texttt{S}_2$ is an assignment to I and X_A such that the $\mathcal{T}_A$ transitions to a different states with $\texttt{S}_1$ and $\texttt{S}_2$. The distinguishing input is found in line 6. Next we use $eval$ to find the correct next-state $\texttt{Y}^\texttt{i}$ and assert that the next distinguishing input must satisfy this transition (line 10). 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 13.

1) Template Bugs: We say the template $\mathcal{T}_A$ and parameters p_i *can express* the relation T_A if for all i: $p_i \implies T_A(I, X_A, Y_A)$ is a member of the set of relations defined by $\{s \in \mathbb{B}^{|S|} \mid \mathcal{T}_A(s, I, X_A, Y_A)\}$. If $p_i \implies T_A(I, X_A, Y_A)$ does not belong to this family of relations, we say that the $\mathcal{T}_A$ and p_i *cannot express* T_A.

We refer to the scenario when $\mathcal{T}_A$ and p_i *cannot express* T_A as a template bug. A template bug may result in the call to the SMT solver in line 13 to be unsatisfiable. When this happens, our synthesis framework prints out the unsat core of F^j. In our experience, examining the simulation inputs and outputs present in the unsat core is sufficient to identify the bug. The algorithm may also return an incorrect transition relation and this will be discovered either when verifying the ILA (see §IV) or when verifying system-level properties using the ILA.

[3]This finding is consistent with SKETCH where the supported "holes" are of only three types: index expressions, lookup tables and bitmasks.

2) Simulator Bugs: Since $eval$ models a simulator and real-world simulators may contain bugs, it is possible that $eval$ is not equivalent to the idealized transition relation T_A, *i.e.*, $eval(\mathtt{I}, \mathtt{X_A}) = \mathtt{Y_A} \iff T_A(\mathtt{I}, \mathtt{X_A}, \mathtt{Y_A})$ does not hold. This will also either cause an unsatisfiable result or an incorrect transition relation. The former can be debugged using the unsat core of F^j while the latter will be detected during verification.

3) Correctness of Algorithm 1: In the absence of template bugs and if $eval$ is equivalent to T_A, we have the following result about Algorithm 1.

(Theorem) If $\mathcal{T}_A$ and p_i can express T_A and $eval(\mathtt{I}, \mathtt{X_A}) = \mathtt{Y_A} \iff T_A(\mathtt{I}, \mathtt{X_A}, \mathtt{Y_A})$ then $\left(\wedge_{i=1}^{N} (p_i \implies \mathcal{T}_A(\mathtt{S_i}, I, X_A, Y_A))\right) \iff T_A(I, X_A, Y_A)$.

IV. Verifying the Instruction-Level Abstraction

Once we have ILA, the next step is to verify that it correctly abstracts the hardware implementation. Our first attempt at this might be to verify properties of the form $\mathbf{G}(x_a = x_f)$ where $x_a \in X_A$ and $x_f \in X_F$ are elements of the firmware-accessible states present in both the abstraction and the implementation. Unfortunately, this property is likely to be false for most internal state in hardware designs. Consider an accelerator design. The ILA may only model a high-level state machine of the accelerator which "executes" an entire operation in one transition but the implementation will step through many intermediate states to accomplish the equivalent. The property is likely invalid during these intermediate states.

A. Verifying Abstraction Correctness

When considering the internal state of the hardware components, we verify the ILA by defining *refinement relations* as proposed by McMillan [14]: $\mathbf{G}(cond \implies x_a = x_f)$. The predicate $cond$ specifies when the equivalence between state in the ILA and the corresponding state in the implementation holds. For example, in a pipelined microprocessor, we might expect that when an instruction commits, the architectural state of the implementation matches the ILA. Defining the refinement relations as above allows compositional verification [12]. Consider the property $\neg(\phi\ \mathbf{U}\ (cond \wedge (x_a \neq x_f)))$ where ϕ states that all refinement relations hold until time $t-1$. This is equivalent to the above property, but we can abstract away irrelevant parts of ϕ when proving equivalence of x_a and x_f.

For state variables that are outputs of hardware components being modeled, we expect that ILA outputs always match the implementation. In this case, the property is $\mathbf{G}(x_a = x_f)$.

1) Discussion of Verification Issues: One part of our case study is a pipelined microcontroller with limited speculative execution. Our refinement relations are of the form $\mathbf{G}(inst\_finished \implies (x_a = x_f))$, *i.e.*, the state of the ILA and implementation must match when each instruction commits. The other part of the case study involves the verification of two cryptographic accelerators. Here the refinement relations are: $\mathbf{G}(hlsm\_state\_changed \implies x_a = x_f)$.

If we had to verify a superscalar processor, the ILA would execute multiple instructions in each transition. The exact number of instructions to be executed with each transition is an output of the implementation and an input to the abstraction. The property would state that after these many instructions are executed, the states of the ILA and implementation match.

B. Verification Correctness

If we prove the refinement relations for all outputs of the ILA and implementation: $\mathbf{G}(x_a = x_f)$, 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.

In practice, proving the property $\mathbf{G}(x_a = x_f)$ for all external outputs may not be scalable, so we will have to adopt McMillan's compositional approach. We prove refinement relations of the form $\neg(\phi\ \mathbf{U}\ (cond \wedge x_a^i \neq x_f^i))$ for internal state and use these to prove the equivalence of the outputs.

If these compositional refinement relations are proven for all firmware-visible state in the ILA and implementation, then we know that all firmware-visible state updates are equivalent between the ILA and the implementation. Further, we know that transitions of the high-level of state machines in the ILA are equivalent to those in the implementation. These properties guarantee that firmware/hardware interactions in the ILA are equivalent to the implementation, capturing the requirements mentioned in Section I-A.

V. Evaluation

This section describes the evaluation methodology, the example SoC used as a case study, and then presents the synthesis and verification results.

A. Evaluation Methodology

We implemented the template-based synthesis framework as a Python library using the Z3 SMT solver [4]. Besides synthesis of the ILA, the library also provides a set of functions for generating behavioral Verilog corresponding to the "golden model". This is used to verify that the ILA matches the implementation. We have made the synthesis framework, template abstractions, synthesized ILA, Verilog netlists and other experimental artifacts available online [5].

We used a slightly-modified version of the open source Yosys [24] tool to synthesize netlists from behavioral Verilog. We used ABC [22] for property verification. Experiments were run on Intel(R) Xeon(R) E5645 and E3-1230 CPUs. The E5645 has 12 cores and 128 GB of RAM, while the E3-1230 has 6 cores and 32 GB of RAM. All experiments were run on Ubuntu Linux v12.04.

1) Example SoC Structure: A block diagram of the example SoC is shown in Figure 4. It consists of the 8051 microcontroller and two cryptographic accelerators. The register-transfer level (RTL) Verilog implementation of the 8051 was taken from OpenCores.org [23]. We used *i8051sim* from UC Riverside as a blackbox instruction-level simulator of the 8051 [13]. One accelerator implements encryption/decryption using the Advanced Encryption Standard (AES) [7]. This is from OpenCores.org [10]. The second accelerator [20]

implements the SHA-1 cryptographic hash function [6]. We wrote interface modules that "expose" the AES and SHA-1 accelerators to the 8051 using a memory-mapped I/O interface.

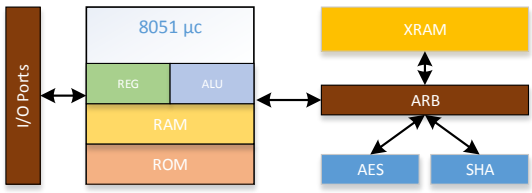

Fig. 4: Example SoC Block Diagram

2) Firmware Programming Interface: The firmware running on the 8051 initiates operation of the accelerators by writing the addresses of the data to be encrypted/decrypted/hashed to memory-mapped registers within the accelerators. Operation is started by writing to the start register which is also memory-mapped. Once the operation is started, 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. The processor determines completion by polling a memory-mapped status register.

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

B. Verifying the Example SoC

We performed the verification in a modular manner by constructing two ILAs: one for the 8051 microcontroller and another for the arbiter, XRAM, AES and SHA modules. The insight here is that the 8051 communicates with the accelerators and XRAM by reading/writing to XRAM addresses. So from the perspective of the 8051, it is sufficient to show that all instructions that modify the internal state of the 8051 are executed correctly and instructions which read/write XRAM produce the correct results at the external memory interface. What happens after these instructions "leave" the external memory interface - whether they modify the XRAM or start AES encryption, or return the current state of the SHA accelerator - need not be considered in this model. For the accelerators and the XRAM, we construct a separate ILA and the only instructions we need to consider here are reads and writes to XRAM addresses. In this ILA, we verify that these operations produce the expected results.

1) Synthesizing the 8051 ILA: We constructed a template ILA of the 8051 which is parameterized over the opcode and models all 256 opcodes of the microcontroller and other elements of architectural state including the internal RAM which contains the register banks, the accumulator and other registers. 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.* [8] report that ISA documents only partially define some instructions and leave some state undefined. They 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 all of these pitfalls.

Model	**LoC**	**Size**
Template ILA	$\approx$ 650	30 KB
C++ instruction-level simulator	$\approx$ 3000	106 KB
Behavioral Verilog implementation	$\approx$ 9600	360 KB

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

As an 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 significantly smaller than both the simulator and the RTL. Table II shows the execution time for synthesis of each element of architectural state. We report the average and maximum values over all 256 opcodes. Except for the internal RAM, all other elements are synthesized with a few seconds.

2) Verifying the 8051 ILA: We first attempted to verify the 8051 by generating a large monolithic golden model 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 golden 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 golden model to verify properties of the form $\mathbf{G}(inst\_finished \implies x_a = x_f)$. For the external outputs of the processor, e.g., the external ram address and data outputs, the properties were of the form $\mathbf{G}(output\_valid \implies x_a = x_f)$. Verification was 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.

State	AVG/MAX Time (s)	State	AVG/MAX Time (s)
ACC	4.3/8.5	B	3.6/5.1
DPH	2.7/5.0	DPL	2.6/4.4
IRAM	1245.7/14043.6	P0	1.8/2.7
P1	2.4/3.8	P2	2.2/3.5
P3	2.7/4.6	PC	6.3/141.2
PSW	7.3/15.9	SP	2.8/5.0
XRAM/addr	0.4/0.4	XRAM/dataout	0.3/0.4

TABLE II: Synthesis execution time for 8051 ILA.

ISBN: 978-0-9835678-5-1.

To improve scalability, we generated a set of "per-instruction" golden 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: $\neg(\phi\ \mathbf{U}(inst\_finished \wedge opcode = o_i \wedge x_a \neq x_f))$. 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 golden model 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 III: Results with per-instruction golden model.

Results for these verification experiments are shown in Table III. 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 algorithm [3] with a time limit of 1950 seconds. Before running `pdr`, we preprocessed the netlists using the gate-level abstraction [15] technique with a time limit of 450 seconds.

We believe all instructions and all architectural states can be proven to match the ILA with some verification effort. We will have to apply the appropriate abstractions and possibly specify a few intermediate lemmas. Due to limited time we were unable to perform these proofs for all cases, so we report the partial results shown above. Yet, the current results do substantiate our claim that the ILA can be proven correct.

3) Bugs Found During 8051 Verification: In the simulator, we found 5 bugs in total. Bugs in `CJNE`, `DA` 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. These bugs were found during synthesis.

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 benefits of our methodology: 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. This is an example of the methodology detecting and preventing unintended leakage of information through undefined state.

4) Synthesizing XRAM+AES+SHA ILA: The template ILA for the cryptographic accelerators models the high-level state machines (HLSM) for each accelerator. The synthesis parameter is the current state of the HLSM of the two accelerators. The template also models reads/write operations from the processor which read/write the external RAM or internal registers in the accelerators. The AES and SHA functions were modeled using uninterpreted functions.

Model	**LoC**	**Size**
Template ILA	≈ 500	26 KB
Python HLSM simulator	≈ 400	14 KB
Behavioral Verilog implementation	≈ 2800	87 KB

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

The sizes of the model are shown in Table IV. Table V shows the time to synthesize each element of the abstraction's state space. Synthesis can be completed in about an hour.

5) Verifying the XRAM+AES+SHA ILA: As before, we generated a Verilog golden model for the XRAM+AES+SHA ILA. We reduced the size of the XRAM in the ILA and the implementation to just one byte because we were not looking to prove correctness of reads and writes to the XRAM. We then attempted to prove a set of properties of the form $\mathbf{G}(hlsm\_state\_change \implies (x_a = x_f))$. 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.

VI. Related Work

Syntax-Guided Synthesis: Our work builds on recent progress in syntax-guided synthesis which is surveyed in [1]. The synthesis primitives we introduce are similar to the idea of "holes" and the `??` operator proposed in SKETCH [19].

State	AVG/MAX Time (s)	State	AVG/MAX Time (s)
AES:Addr	0.5/1.0	AES:BytesProcessed	0.6/1.5
AES:Ctr	0.6/1.6	AES:EncData	0.4/0.4
AES:Key0	0.7/1.7	AES:Key1	0.6/1.5
AES:Len	0.4/0.9	AES:ReadData	0.4/0.5
AES:State	0.8/2.0	Dataout	91.9/345.2
SHA:BytesProcessed	0.3/0.5	SHA:Digest	0.3/0.3
SHA:Len	0.4/0.4	SHA:RDAddr	0.4/0.4
SHA:Readdata	81.9/588.3	SHA:State	0.3/0.4
SHA:WRAddr	0.4/0.5	XRAM	22.4/58.1

TABLE V: Synthesis execution time for XRAM+AES+SHA ILA.

The synthesis algorithm is based on oracle-guided synthesis from [11]. Our contribution is in the application of synthesis to constructing abstractions for verification and the parameterized formulation which makes synthesizing the ILA tractable.

Synthesizing Abstractions: Godefroid *et al*. [8] synthesize a symbolic model for a subset of the ALU instructions in an x86-core using input/output samples. They cannot verify the correctness of the synthesized model, so it may or may not correspond to the implementation. As such, it is insufficient for our scenario where we wish to use the model for system-level verification with strong guarantees of correctness. Furthermore, our synthesis framework can be used to model general hardware components while they focus on a specific part of the microprocessor: the ALU result and flag outputs.

Verifying Abstraction Correctness: The *refinement relations* we use in proving that the abstraction and the implementation match are from [12, 14]. In [12], Jhala and McMillan show how refinement relations can be defined to prove the correctness of an out-of-order superscalar processor. 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: One approach to compositional SoC verification is by Xie *et al*. [25, 26]. They suggest manually constructing a "bridge" specification that along with a set of hardware properties can be used to verify software components that rely on these properties. Our methodology makes it easy to construct the equivalent of the bridge specifications. It has the added benefit of ensuring the abstraction is correct.

Horn *et al*. [9] suggests symbolic execution on a software model that contains both firmware and software models of hardware components. This approach is complementary to ours because it can used for early-design stage verification, when an RTL model may not be available. However, once the RTL model is constructed, there is no easy way of ensuring that the software model and the RTL are in agreement. This is the critical challenge addressed by our work.

VII. Conclusion

Modern SoCs consist of a number of programmable cores and many accelerators and peripheral devices which are controlled by firmware running on the cores. The functionality of the SoC is derived by this combination of firmware and hardware. Verifying such SoCs is challenging because formally verifying the complete SoC with firmware and hardware is not scalable, while verifying the two separately may miss bugs.

In this paper, we introduced a methodology for SoC verification that *synthesizes* an *instruction-level* abstraction (ILA) of the SoC. 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. A set of refinement relations are defined to prove that the behavior of the ILA matches the implementation. The other advantage is that instead of specifying the complete ILA, the verification has the much easier task of writing a template ILA which partially defines the operation of the hardware components, and the synthesis algorithm is able to synthesize 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 *Formal Methods in Computer-Aided Design*, 2013.

[2] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computing*, November 1987.

[3] A. R. Bradley. SAT-based Model Checking Without Unrolling. In *Verification, Model Checking, and Abstract Interpretation*, 2011.

[4] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[5] Experimental artifacts and synthesis framework source code. https://bitbucket.org/spramod/fmcad-15-soc-ila, 2015.

[6] NIST FIPS. 180-2: Secure Hash Standard (SHS). Technical report, National Institute of Standards and Technology, 2001.

[7] NIST FIPS. 197: Announcing the Advanced Encryption Standard (AES). Technical report, 2001.

[8] P. Godefroid and A. Taly. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *Programming Language Design and Implementation*, 2012.

[9] 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 *Formal Methods in Computer-Aided Design*, Oct 2013.

[10] H. Hsing. http://opencores.org/project,tiny_aes, 2014.

[11] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *International Conference on Software Engineering*, 2010.

[12] R. Jhala and K. L. Mcmillan. Microarchitecture verification by compositional model checking. In *Computer-Aided Verification*, 2001.

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

[14] K. L McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *Correct Hardware Design and Verification Methods*. Springer, 2001.

[15] Alan Mishchenko, Niklas Een, Robert Brayton, Jason Baumgartner, Hari Mony, and Pradeep Nalla. GLA: Gate-level Abstraction Revisited. In *Design, Automation and Test in Europe*, 2013.

[16] M. D. Nguyen, M. Wedler, D. Stoffel, and W. Kunz. Formal Hardware/Software Co-verification by Interval Property Checking with Abstraction. In *Design Automation Conference*, 2011.

[17] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. P. Pande, C. Grecu, and A. Ivanov. System-on-Chip: Reuse and Integration. *Proceedings of the IEEE*, 94(6), 2006.

[18] R. E. Schapire. *The Design and Analysis of Efficient Learning Algorithms*. MIT Press, 1992.

[19] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *Architectural Support for Programming Languages and Operating Systems*, 2006.

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

[21] P. Subramanyan and D. Arora. Formal Verification of Taint-Propagation Security Properties in a Commercial SoC Design. In *Design, Automation and Test in Europe*, 2014.

[22] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/~alanmi/abc/, 2014.

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

[24] Clifford Wolf. http://www.clifford.at/yosys/, 2015.

[25] F. Xie, X. Song, H. Chung, and Ranajoy N. Translation-based Co-verification. In *Formal Methods and Models for Co-Design*, 2005.

[26] F. Xie, G. Yang, and X. Song. Component-based Hardware/Software Co-verification for Building Trustworthy Embedded Systems. volume 80, May 2007.