ATLAS: Automatic Term-Level Abstraction of RTL Designs

Bryan A. Brady
UC Berkeley
bbrady@eecs.berkeley.edu

Randal E. Bryant
CMU
randy.bryant@cs.cmu.edu

Sanjit A. Seshia
UC Berkeley
sseshia@eecs.berkeley.edu

John W. O'Leary
Intel
john.w.oleary@intel.com

***Abstract*—Abstraction plays a central role in formal verification. Term-level abstraction is a technique for abstracting word-level designs in a formal logic, wherein data is modeled with abstract terms, functional blocks with uninterpreted functions, and memories with a suitable theory of memories. A major challenge for any abstraction technique is to determine what components can be safely abstracted. We present an automatic technique for term-level abstraction of hardware designs, in the context of equivalence and refinement checking problems. Our approach is hybrid, involving a combination of random simulation and static analysis. We use random simulation to identify functional blocks that are suitable for abstraction with uninterpreted functions. Static analysis is then used to compute conditions under which such function abstraction is performed. The generated term-level abstractions are verified using techniques based on Boolean satisfiability (SAT) and satisfiability modulo theories (SMT). We demonstrate our approach for verifying processor designs, interface logic, and low-power designs. We present experimental evidence that our approach is efficient and that the resulting term-level models are easier to verify even when the abstracted designs generate larger SAT problems.**

I. Introduction

Register-transfer-level (RTL) descriptions are often the most authoritative models of a system, so it is essential for formal verification tools to operate on RTL. In practice, however, RTL is written at a very low level of abstraction: data are represented as bits and bit vectors, and operations on the data are accomplished by bit-level manipulation. In verification tasks that involve proving strongly data-dependent properties such as equivalence or refinement checking, bit-level RTL quickly causes state-space explosion, and additional abstraction is required.

Term-level modeling seeks to make formal verification of data-intensive properties tractable by abstracting away details of data representations and operations, viewing data as symbolic *terms*. Term-level abstraction has been found to be especially useful in microprocessor design verification, using techniques such as term-level bounded model checking, correspondence checking, refinement verification, and predicate abstraction [12], [15], [17], [18]. The precise functionality of operations of units such as instruction decoders and the ALU are abstracted away using *uninterpreted functions*, and decidable fragments of first-order logic are employed in modeling memories, queues, counters, and other common constructs. Efficient satisfiability modulo theories (SMT) solvers for fragments of first-order logic [5] are used as the computational engines for term-level verifiers.

A major obstacle for term-level verification is the need to generate term-level models from word-level RTL. On the one hand, constructing these models by hand is a tedious process prone to errors, hence automation is essential. On the other hand, automatically abstracting all bit-vector signals to terms and all operators to uninterpreted functions results in too coarse an abstraction, in which properties of bit-wise and finite-precision arithmetic operators are obscured, leading to a huge number of spurious counterexamples. While such spurious counterexamples can in many cases be eliminated by selectively abstracting only parts of the design to the term level, manual abstraction requires detailed knowledge of the RTL design and the property to be verified it is difficult for a human to decide what functional blocks or operators to abstract in order to obtain efficiency gains and also avoid spurious counterexamples.

In this paper, we present ATLAS, an approach for *automatically generating a term-level verification model* from a word-level description such as Verilog RTL. In particular, we focus on *function abstraction*: ATLAS conditionally abstracts functional blocks in the original design with uninterpreted functions. To perform such abstraction, ATLAS employs *a combination of random simulation and static analysis*, exploiting the module structure specified by the designer. Random simulation is used to identify functional blocks corresponding to module instantiations that are suitable for abstraction with uninterpreted functions. It is always sound to replace a functional block with an uninterpreted function, in the sense that the correctness of the resulting design implies that of the original. However, because such abstraction loses information, it is necessary to rule out spurious counterexamples. For this purpose, ATLAS aims to use static analysis to compute conditions under which such function abstraction can be performed without loss of precision; i.e., if the resulting term-level design is incorrect, then so is the original word-level design. However, as we show in this paper, even checking that a candidate function abstraction is precise is co-NP-hard. Therefore, we provide heuristics to statically compute conditions which in many cases are sufficiently precise that spurious counterexamples are avoided. The generated term-level abstractions are verified using techniques based on Boolean satisfiability (SAT) and satisfiability modulo theories (SMT). We demonstrate our approach for verifying processor designs, interface logic, and low-power designs. We present experimental evidence that our approach is efficient and that the resulting term-level models are easier to verify *even when* the abstracted designs generate larger SAT problems.

The rest of this paper is organized as follows. We present background material on term-level abstraction and discuss related work in Section II. Formal notation is introduced in Section III, along with a running example. The ATLAS approach is described in Section IV. Experimental results are presented in Section V and we conclude in Section VI.

II. Background and Related Work

Background material on term-level abstraction is presented in Sec. II-A and related work in Sec. II-B.

A. Term-Level Abstraction

Informally, a (word-level) design is said to be abstracted to the *term-level* if one or more of the following three abstraction techniques is employed:

1. *Function Abstraction:* In function abstraction, bit-vector operators and modules computing bit-vector values are treated as "black-box," *uninterpreted* functions constrained only by functional consistency: that they must evaluate to the same values on the same arguments. It is possible for the inputs and outputs of uninterpreted functions to be bit vectors or to be abstract terms (say, interpreted over $\mathbb{Z}$). Function abstraction is the focus of this paper, and we limit ourselves to uninterpreted functions that map bit vectors to bit vectors.
2. *Data Abstraction:* Bit-vector expressions are modeled as abstract terms that are interpreted over a suitable domain (typically a subset of $\mathbb{Z}$). Data abstraction is effective when it is possible to reason over the domain of abstract terms far more efficiently than it is to do so over the original bit-vector domain, through use of small-domain or bit-width reduction techniques. Data abstraction is not the focus of this paper.
3. *Memory Abstraction:* In memory abstraction, memories and data structures are modeled in a suitable theory of arrays or memories, such as by the use of special **read** and **write** functions [12] or lambda expressions [10]. We do not address automatic memory abstraction in this paper.

We illustrate the concept of function abstraction using a toy ALU design. Consider the simplified ALU shown in Figure 1(a). Here a 20-bit instruction is split into a 4-bit opcode and a 16-bit data field. If the opcode indicates that the instruction is a jump, the data field indicates a target address for the jump and is simply passed through the ALU unchanged. Otherwise, the ALU computes the square of its input 16-bit data field and generates as output the resulting 16-bit result.

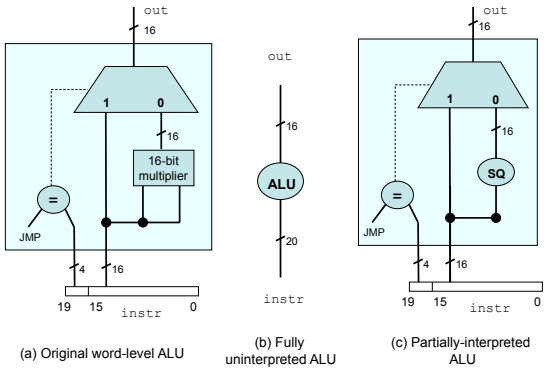

Fig. 1. **Three versions of an ALU design.** Boolean signals are shown as dashed lines and bit-vector signals as solid black lines.

Using very coarse-grained term-level abstraction, one could abstract the entire ALU module with a single uninterpreted function (UF), as shown in Figure 1(b). However, we lose the precise mapping from `instr` to `out`.

Such a coarse abstraction is quite easy to perform automatically. However, this abstraction loses information about the behavior of the ALU on jump instructions and can easily result in spurious counterexamples. In Section III-B, we will describe a larger equivalence checking problem within which such an abstraction is too coarse to be useful.

Suppose that reasoning about the correctness of the larger circuit containing this ALU design only requires one to precisely model the difference in how the jump and squaring instructions are handled. In this case, it would be preferable to use a partially-interpreted ALU model as depicted in Figure 1(c). In this model, the control logic distinguishing the handling of jump and non-jump instructions is precisely modeled, but the datapath is abstracted using the uninterpreted function SQ. However, creating this fine-grained abstraction by hand is difficult in general and places a larger burden on the designer. It is this burden that we seek to mitigate using our ATLAS approach.

B. Related Work

The first automatic term-level abstraction tool was Vapor [4], which aimed at generating term-level models from Verilog. The underlying logic for term-level modeling in Vapor is CLU, which originally formed the basis for the UCLID system [10]. Vapor uses a counterexample-guided abstraction-refinement (CEGAR) approach [4]. Vapor has been since subsumed by the Reveal system [2], [3] which differs mainly in the refinement strategies in the CEGAR loop.

Both Vapor and Reveal start by completely abstracting a Verilog description to the UCLID language by modeling all bit-vector signals as abstract terms and all operators as uninterpreted functions. Next, verification is attempted on the abstracted design. If the verification succeeds, the tool terminates. However, if the verification fails, it checks whether the counterexample is spurious using a bit-vector decision procedure. If the counterexample is spurious, a set of bit-vector facts are derived, heuristically reduced, and used on the next iteration of term-level verification. If the counterexample is real, the system terminates, having found a real bug.

The CEGAR approach has shown promise [3]. In many cases, however, several abstraction-refinement iterations are needed to infer fairly straightforward properties of data, thus imposing a significant overhead. For instance, in one example, a chip multiprocessor router [20], the header field of a packet must be extracted and compared several times to determine whether the packet is correctly forwarded. If any one of these extractions is not modeled precisely at the word-level, a spurious counterexample results. The translation is complicated by the need to instantiate relations between individually accessed bit fields of a word modeled as a term using special uninterpreted functions to represent concatenation and extraction operations.

Our paper is the first to combine random simulation with static analysis to perform *automatic conditional function abstraction*. Potentially, if the statically-computed conditions generated by ATLAS make the problem size too large, one can fall back to a CEGAR approach.

We also note that the ATLAS approach presented herein could in principle be combined with bitwidth reduction techniques (e.g. [6], [16]) to perform combined function and data abstraction.

III. Definitions and Overview

A. Formal Definitions

We model a design at the word level as a *word-level netlist* $\mathcal{N} = (\mathcal{I}, \mathcal{O}, \mathcal{S}, \mathcal{C}, \mathit{Init}, \mathcal{A})$ where

- $\mathcal{I}$ is a finite set of input signals;
- $\mathcal{O}$ is a finite set of output signals;
- $\mathcal{S}$ is a finite set of intermediate sequential (state-holding) signals;
- $\mathcal{C}$ is a finite set of intermediate combinational (stateless) signals;
- Init is a set of initial states, i.e., initial valuations to elements of $\mathcal{S}$, and
- $\mathcal{A}$ is a finite set of assignments to outputs and to sequential and combinational intermediate signals. An assignment is an expression that defines how a signal is computed and updated. We elaborate below on the form of assignments.

First, note that input and output signals are assumed combinational, without loss of generality. Moreover, although the signals in the designs we consider can all be modeled as bit vectors of varying sizes, it is useful to distinguish Boolean signals for the control logic from bit-vector valued signals modeling the datapath. In word-level designs, a memory is modeled as a flat array of bit-vector signals.

A *combinational assignment* is a rule of the form $v \leftarrow e$, where v is a signal in the disjoint union $\mathcal{C} \uplus \mathcal{O}$ and e is an expression that is a function of $\mathcal{C} \uplus \mathcal{S} \uplus \mathcal{I}$. Combinational loops are disallowed. We differentiate between combinational assignments based on the type of the right-hand side expression and write them as follows:

$$v \leftarrow \mathit{bv} \;\mid\; b \leftarrow \mathit{bool}$$

Here *bv* and *bool* represent bit-vector and Boolean expressions in a word-level design, as listed in the grammar in Fig. 2.

$$\begin{array}{ll} \mathit{bv} ::= & c \mid v \mid \mathit{ITE}(b,\, v_1,\, v_2) \mid \mathsf{bvop}(v_1, \ldots, v_k) \quad (k \geq 1) \\ \mathit{bool} ::= & \mathbf{true} \mid \mathbf{false} \mid b \mid \neg b \mid b_1 \vee b_2 \\ & \mid b_1 \wedge b_2 \mid v_1 = v_2 \mid \mathsf{bvrel}(v_1, \ldots, v_k) \quad (k \geq 1) \end{array}$$

Fig. 2. **Syntax for Bit-Vector and Boolean Expressions.** c and v denote a bit-vector constant and variable respectively, and b is a Boolean variable. bvop denotes any arithmetic operator mapping bit vectors to bit vectors, while bvrel is a relational operator other than equality mapping bit vectors to a Boolean value.

A *sequential assignment* is a rule of the form $v := e$, where v is a signal in $\mathcal{S}$ and e is an expression that is a function of $\mathcal{C} \uplus \mathcal{S} \uplus \mathcal{I}$. Again, we differentiate between sequential assignments based on type, and write them as follows (where v, u are any bit-vector signals and b, b_a are any Boolean signals):

$$v := u \;\mid\; b := b_a$$

Note that we assume that the right-hand side of a sequential assignment is a signal; this loses no expressiveness since we can always introduce a new name to represent any expression.

A *word-level design* $\mathcal{D}$ is a tuple $\langle \mathcal{I}, \mathcal{O}, \{\mathcal{N}_i \mid i = 1, \ldots, N\}\rangle$, where $\mathcal{I}$ and $\mathcal{O}$ denotes the set of input and output signals of the design, and the design is partitioned into a collection of N word-level netlists. A *well-formed* design is one where (i) every output of a netlist is either an output of the design or an input to some netlist (including itself) – i.e., there are no dangling outputs; and (ii) every input of a netlist is either an input to the design or exactly one output of some netlist. We refer to the netlists $\mathcal{N}_i$ as *functional blocks*, or *fblocks*.

We revise the expression syntax in order to model designs at the term level, with the revisions shown in Fig. 3. Since data abstraction is not addressed in this paper, we exclude abstract term-level expressions from the syntax. We include memories in the syntax since we employ memory abstraction in this paper for some verification problems. The interpreted memory functions **read** and **write** are used for brevity; we can use any specific memory modeling technique including the use of lambda expressions [10].

$$\begin{array}{ll} \mathit{bv} ::= & \mathbf{read}(M, v) \mid UF(v_1, \ldots, v_k) \quad (k \geq 0) \\ \mathit{bool} ::= & UP(v_1, \ldots, v_k) \quad (k \geq 0) \\ \mathit{mem} ::= & A \mid M \mid \mathbf{write}(M, v_1, v_2) \end{array}$$

Fig. 3. **Syntax for Term-Level Expressions.** We only show additions to the expression syntax of Fig. 2. UF and UP denote an uninterpreted function and predicate symbol respectively. A and M denote constant and variable memories. The second argument to **read** and **write** denote addresses and the third argument to **write** denotes the data value to be written.

A *term-level netlist* is a generalization of a word-level netlist where expressions can be both from the syntax shown in Figure 2 and that in Fig. 3. Additionally, a term-level netlist can have sequential and combinational assignments to memory variables, of the form below (where M, M_1 are any memory signals and *mem* is any memory expression):

$$M := \mathit{mem}$$

A term-level netlist that has at least one expression of the form $UF(v_1, \ldots, v_k)$ or $UP(v_1, \ldots, v_k)$ is referred to as a *strict term-level netlist*.

A *term-level design* $\mathcal{T}$ is a tuple $(\mathcal{I}, \mathcal{O}, \{\mathcal{N}_i \mid i = 1, \ldots, N\})$, where each fblock $\mathcal{N}_i$ is a term-level netlist.

Given a word-level design $\mathcal{D} = (\mathcal{I}, \mathcal{O}, \{\mathcal{N}_i \mid i = 1, \ldots, N\})$, we say that $\mathcal{T}$ is a *term-level abstraction* of $\mathcal{D}$ if $\mathcal{T}$ is obtained from $\mathcal{D}$ by replacing some word-level fblocks $\mathcal{N}_i$ by strict term-level fblocks $\mathcal{N}_i'$.

The verification problems of interest in this paper are *equivalence checking* and *refinement checking*.

Given two word-level designs $\mathcal{D}_1$ and $\mathcal{D}_2$, the *word-level equivalence* (*word-level refinement*) checking problem is to verify that $\mathcal{D}_1$ is *sequentially equivalent* to (*refines*) $\mathcal{D}_2$.

The definition is similarly extended to a pair of term-level designs $\mathcal{T}_1$ and $\mathcal{T}_2$. We also consider bounded equivalence checking problems, where the designs are to be proved equivalent for a bounded number of cycles from the initial state.

The *term-level abstraction problem* we consider in this paper is as follows.

> Given a pair of word-level designs $\mathcal{D}_1$ and $\mathcal{D}_2$, abstract them to term-level designs $\mathcal{T}_1$ and $\mathcal{T}_2$, such that $\mathcal{D}_1$ is equivalent to (refines) $\mathcal{D}_2$ if and only if $\mathcal{T}_1$ is equivalent to (refines) $\mathcal{T}_2$.

In this paper, we demonstrate that such term-level abstraction can be performed efficiently and automatically, and it can scale up verification by orders of magnitude.

B. Illustrative Example

Figure 4 depicts the equivalence checking problem that we will use as a running example in this section. Two variants of the same circuit, denoted Design A and Design B, are to be checked for output equivalence.

Consider Design A. This design models a fragment of a processor datapath. PC models the program counter register, which is an index into the instruction memory denoted as IMem. The instruction is a 20-bit word denoted `instr`, and is an input to the ALU design shown earlier in Figure 1(a). The top four bits of `instr` are the operation code. If the instruction is a jump instruction (`instr`$[19:16]$ equals JMP), then the PC is set equal to the ALU output `out`; otherwise, it is incremented by 4.

Design B is virtually identical to Design A, except in how the PC is updated. For this version, if `instr`$[19:16]$ equals JMP, the PC is directly set to be the jump address `instr`$[15:0]$.

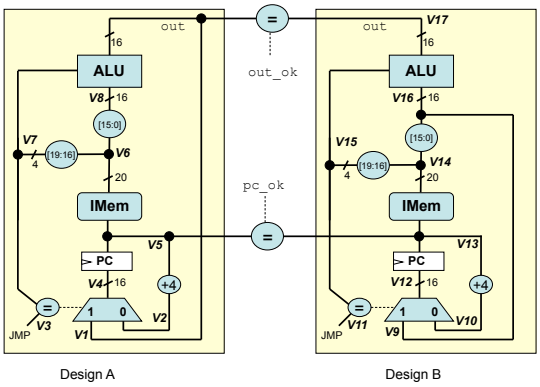

Fig. 4. **Equivalence checking of two versions of a portion of a processor design.** Boolean signals are shown as dashed lines and bit-vector signals as solid lines.

Note that we model the instruction memory as a read-only memory using an uninterpreted function IMem. The same uninterpreted function is used for both Design A and Design B. We also assume that Designs A and B start out with identical values in their PC registers.

The two designs are equivalent iff their outputs are equal at every cycle, meaning that the Boolean assertion `out_ok` $\wedge$ `pc_ok` is always **true**.

It is easy to see that this is the case, because from Figure 1(a) we know that `A.out` always equals `A.instr`$[15:0]$ when `A.instr`$[19:16]$ equals JMP. The question is whether we can infer this without the full word-level representation of the ALU.

Consider what happens if we use the abstraction of Figure 1(b). In this case, we lose the relationship between `A.out` and `A.instr`$[19:16]$. Thus, the verifier comes back to us with a spurious counterexample, where in cycle 1 a jump instruction is read, with the jump target in Design A different from that in Design B, and hence `A.PC` differs from `B.PC` in cycle 2.

However, if we instead used the partial term-level abstraction of Figure 1(c) then we can see that the proof goes through, because the ALU is precisely modeled under the condition that `A.instr`$[19:16]$ equals JMP, which is all that is necessary.

The challenge is to be able to generate this partial term-level abstraction automatically. We describe our approach to solving this problem below.

IV. The ATLAS Approach

Our term-level abstraction approach, ATLAS, operates in three stages:

1. *Perform Random Simulation:* For each fblock in the design, this step determines whether or not to mutate the module. We replace each candidate fblock with a random, functionally-consistent fblock, where each output of the fblock is a random function of its inputs. For every such replacement, we perform verification by random simulation. If replacing an fblock by a random function causes the verification to fail in more than a specific fraction of random simulations, we do not consider it further for function abstraction. Those fblocks that survive are analyzed in the next step.
2. *Perform Static Analysis:* We perform static dataflow analysis on every fblock that survives Step 1 above. For each output signal of every such fblock, we compute conditions under which that output signal is modeled as an uninterpreted function of the input signals to that fblock.
3. *Generate Term-Level Abstraction:* Finally, we generate the term-level netlist from the word-level netlist by replacing each fblock with a term-level fblock where some outputs are modeled as a partially-interpreted function of the inputs. The conditions under which the output is partially interpreted are the ones computed in Step 2 above.

We then use SAT and SMT based methods to verify equivalence (refinement) on the resulting term-level designs.

In this paper, we limit ourselves to abstracting fblocks as *combinational* uninterpreted functions of their inputs. It is possible to also abstract fblocks as uninterpreted functions of a bounded history of inputs; however, we leave an exploration of this direction to future work.

Each step in ATLAS is described in detail below. We also discuss optimizations to the basic approach outlined above, such as skipping conditional abstraction when unconditional abstraction is also precise.

A. Identifying candidate fblocks

Word-level designs $\mathcal{D}_1$ and $\mathcal{D}_2$ are derived from RTL designs in languages such as Verilog and VHDL. In such languages, modules defined by the designer provide natural boundaries for function abstraction.

Consider the flat word-level netlist obtained from an RTL design after performing all module instantiations. Every module instance corresponds to a functional block, or fblock, of the flat netlist. However, only some of these fblocks are of interest for function abstraction.

The first important notion in this regard is that of *isomorphic fblocks*. Two fblocks $\mathcal{N}_1 = (\mathcal{I}_1, \mathcal{O}_1, \mathcal{S}_1, \mathcal{C}_1, \mathit{Init}_1, \mathcal{A}_1)$ and $\mathcal{N}_2 = (\mathcal{I}_2, \mathcal{O}_2, \mathcal{S}_2, \mathcal{C}_2, \mathit{Init}_2, \mathcal{A}_2)$ are said to be *isomorphic* if there is a bijective function φ such that $\varphi(\mathcal{I}_1, \mathcal{O}_1, \mathcal{S}_1, \mathcal{C}_1) = (\mathcal{I}_2, \mathcal{O}_2, \mathcal{S}_2, \mathcal{C}_2)$ and if we substitute every signal s in Init_1 and $\mathcal{A}_1$ by $\varphi(s)$ we obtain Init_2 and $\mathcal{A}_2$.

Thus, of all the fblocks that are candidates for function abstraction, we only consider those fblocks in $\mathcal{D}_1$ that have an isomorphic counterpart in $\mathcal{D}_2$ (and vice-versa). Such an fblock is termed as a *replicated fblock*.

For example, each ALU in Fig. 4 is a replicated fblock.

In equivalence or refinement checking, replicated fblocks are easy to identify as instances of the same RTL module that appear in both designs, and this is how we identify them in this work. Note, however, that it is also possible for fblocks that are not instances of the same module to be isomorphic.

Given that we identify replicated fblocks as instances of the same module, the question then becomes one of selecting RTL modules whose instances generate candidates fblocks for abstraction. Currently, we make this selection based on heuristic rules, such as the size of the module in terms of the number of input, output and internal signals, or the presence of operators such as multiplication or XOR that are hard for formal verification engines (such as SAT solvers) to reason efficiently about. Note however, that this is purely an optimization step. One can identify any set of replicated fblocks as candidates for function abstraction.

To summarize, given designs $\mathcal{D}_1$ and $\mathcal{D}_2$ that are to be checked for equivalence or refinement, we can generate the set containing all replicated fblocks in those designs. This set can be partitioned into a collection of sets of fblocks $\mathcal{FS} = \{\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k\}$. Each set $\mathcal{F}_j$ comprises replicated fblocks that are isomorphic to each other. We term each $\mathcal{F}_j$ as an *equivalence class of the fblocks it contains*. Our ATLAS approach uses the same function abstraction for every fblock in $\mathcal{F}_j$. For example, in the design of Fig. 4, `A.ALU` and `B.ALU` are isomorphic to each other and together constitute one set $\mathcal{F}_j$. ATLAS will compute the same function abstraction, shown in Fig. 1(c), for both fblocks.

In the following sections, we describe how ATLAS analyzes the sets in $\mathcal{FS}$ in two phases. In the first phase, *random simulation* is used to prune out replicated fblocks that most likely cannot be abstracted with uninterpreted functions. Every fblock that survives the first phase is then *statically analyzed* in the second phase in order to compute conditions under which that fblock can be abstracted with an uninterpreted function. The resulting conditions are used to generate a term-level netlist for further formal verification.

B. Random simulation

Given an equivalence class of functional blocks $\mathcal{F}$, we use random simulation to determine whether the fblocks it contains are considered for abstraction with an uninterpreted function.

We begin by introducing some notation.

Let the cardinality of $\mathcal{F}$ be l. Let each fblock $f_i \in \mathcal{F}$ have m bit-vector output signals $\langle v_{i1}, \ldots, v_{im} \rangle$, and n input signals $\langle u_{i1}, \ldots, u_{in} \rangle$. Then, we term the tuple of corresponding output signals $\chi = (v_{1j}, v_{2j}, \ldots, v_{lj})$, for each $j = 1, 2, \ldots, m$, as a tuple of *isomorphic output signals*.

Given a tuple of isomorphic output signals $\chi = (v_{1j}, v_{2j}, \ldots, v_{lj})$, we create a *random function* RF_χ unique to χ that has n inputs (corresponding to input signals $\langle v_{i1}, \ldots, v_{in} \rangle$, for fblock f_i).

For each fblock f_i, $i = 1, 2, \ldots, l$, we replace the assignment to the output signal v_{ij} with the random assignment $v_{ij} \leftarrow RF_\chi(u_{i1}, \ldots, u_{in})$. This substitution is performed for all output signals $j = 1, 2, \ldots, m$.

The resulting designs $\mathcal{D}_1$ and $\mathcal{D}_2$ are then verified through simulation. This process is repeated for T different random functions RF_χ.

If the fraction of failing verification runs is greater than a threshold τ, then we drop the equivalence class $\mathcal{F}$ from further consideration. (The values of T and τ we used in experiments are given in Sec. V-B.) Otherwise, we retain $\mathcal{F}$ for static analysis, as described in the following section.

C. Static analysis

The goal of static analysis is to compute conditions under which fblocks can be abstracted with uninterpreted functions (UFs) without loss of *precision* – i.e., without generation of spurious counterexamples. ATLAS performs this analysis by attempting to compute the opposite condition, under which the fblocks are not abstracted with UFs.

More specifically, for each tuple of isomorphic output signals χ of each equivalence class $\mathcal{F}$, we compute a Boolean condition under which the elements of χ *should not be abstracted* as uninterpreted functions of the inputs to their respective fblocks. We term these conditons as *interpretation conditions*, with the connotation that the fblocks are precisely interpreted iff these conditions are $\mathbf{true}$.

Clearly, $\mathbf{true}$ is a valid interpretation condition, but it is a trivial one and not very useful. It turns out that even *checking whether a given interpretation condition is precise is co-NP-hard*. We prove this by formalizing the problem as below:

> INTCONDCHK: Given word-level designs $\mathcal{D}_1$ and $\mathcal{D}_2$, let f_1 and f_2 be fblocks in $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively, where f_1 and f_2 are isomorphic. Let c be a Boolean condition such that $c \not\equiv \mathbf{true}$. Let designs $\mathcal{T}_1$ and $\mathcal{T}_2$ result from conditionally abstracting f_1 and f_2 with an uninterpreted function UF only when condition c is $\mathbf{false}$.
> Then, the INTCONDCHK problem is to decide whether, given $\langle \mathcal{D}_1, \mathcal{D}_2, f_1, f_2, c \rangle$, $\mathcal{D}_1$ is equivalent to $\mathcal{D}_2$ iff $\mathcal{T}_1$ is equivalent to $\mathcal{T}_2$.

Theorem 1: Problem INTCONDCHK is co-NP-hard.

Proof: The proof is by reduction from UNSAT – the Boolean unsatisfiability problem. We map an arbitrary Boolean formula f to a tuple $\langle \mathcal{D}_1, \mathcal{D}_2, f_1, f_2, c \rangle$, so that f is unsatisfiable if and only if $\mathcal{D}_1$ is equivalent to $\mathcal{D}_2$ iff $\mathcal{T}_1$ is equivalent to $\mathcal{T}_2$.

Consider the word-level circuit in Fig. 5, where the $\mathcal{D}_1$ is the circuit rooted at the left-hand input of the equality node, and $\mathcal{D}_2$ is the circuit rooted at the right-hand input. Clearly, $\mathcal{D}_1$ is equivalent to $\mathcal{D}_2$. Let $c = \mathbf{false}$, in other words, we want to know if unconditional abstraction is precise. Consider the multiplier blocks in $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively. Since these blocks are isomorphic, we can consider replacing them with the same uninterpreted function. Note that, unless $f(x_1, x_2, \ldots, x_n)$ is equivalent to $\mathbf{false}$, this abstraction can result in spurious counterexamples, since it is possible that $UF(2, 5) \neq UF(1, 10)$, whereas MULT$(2, 5)$ = MULT$(1, 10)$ always. In other words, we answer 'yes' to this instance of INTCONDCHK iff $f(x_1, x_2, \ldots, x_n) \equiv \mathbf{false}$, implying that INTCONDCHK is co-NP-hard.

■

Algorithm 1 Procedure CONDITIONALFUNCABSTRACTION($\mathcal{D}$, $\mathcal{FS}$): abstracting fblocks with uninterpreted functions, either wholly or partially.

1: // Input: Combined netlist (miter) $\mathcal{D} := \langle \mathcal{I}, \mathcal{O}, \{\mathcal{N}_i \mid i = 1, \ldots, N\}\rangle$
2: // Input: Equivalence classes of fblocks $\mathcal{FS} := \{\mathcal{F}_j \mid j = 1, \ldots, k\}$,
3: // Output: Rewritten netlist (miter) $\mathcal{D}' := \langle \mathcal{I}, \mathcal{O}, \{\mathcal{N}'_i \mid i = 1, \ldots, N\}\rangle$
4: **for** each $\mathcal{F}_j \in \mathcal{FS}$ **do**
5: **for** each tuple of isomorphic output signals $\chi_j = (v_1, v_2, \ldots, v_{l_j})$, where $l_j = |\mathcal{F}_j|$, $v_i \in f_i$ for fblock $f_i \in \mathcal{F}_j$, $i = 1, 2, \ldots, l_j$ **do**
6: **for** each output signal $v_i \in \chi_j$ **do**
7: Compute equivalence class $\mathcal{E}(v_i)$ of v_i by repeatedly applying rules in Table I to all assignments in $\mathcal{D}$ except for those inside the fblock f_i
8: If $\mathcal{E}(v_i)$ contains a signal u s.t. u is assigned a bit-vector constant or is the input to or output of a bit-vector operator, mark χ_j.
9: **end for**
10: If χ_j is unmarked { $\mathcal{F}_j \leftarrow$ ABSTRACTWITHUF($\mathcal{F}_j$,χ_j) }
11: **end for**
12: **end for**
13: For all $\mathcal{F}_j \in \mathcal{FS}$, if all isomorphic output signal tuples χ_j are unmarked, delete $\mathcal{F}_j$ from $\mathcal{FS}$.
14: // Now compute conditions for partial abstraction with a UF
15: **for** each remaining $\mathcal{F}_j \in \mathcal{FS}$ **do**
16: **for** each tuple of isomorphic output signals $\chi_j = (v_1, v_2, \ldots, v_{l_j})$, where $v_i \in f_i$ for fblock $f_i \in \mathcal{F}_j$, $i = 1, 2, \ldots, l_j$ **do**
17: Compute interpretation conditions c_{v_i} for all i by repeatedly applying rules in Table II to the netlist obtained by deleting all signals (and corresponding assignments) inside fblocks in $\mathcal{F}_j$. The rules are applied until the conditions do not change or up to a specified bounded number of iterations, whichever is smaller.
18: Compute $\mathsf{oc}_j := \bigvee_{i=1}^{l_j} \mathsf{c}_{v_i}$.
19: Perform partial function abstraction of $\mathcal{F}_j$ with oc_j: $\mathcal{F}_j \leftarrow$ CONDITIONALABSTRACTWITHUF($\mathcal{F}_j$,χ_j,oc_j)
20: **end for**
21: **end for**

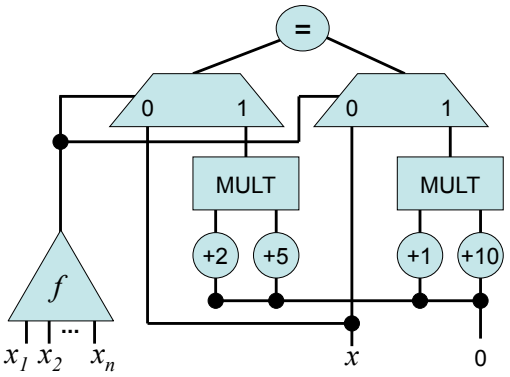

Fig. 5. **Circuit for showing NP-hardness of** INTCONDCHK**.** f is any arbitrary Boolean function of $x_1, x_2, \ldots, x_n$.

Given this hardness result, ATLAS uses the following three-step procedure for verification by term-level abstraction:

1. Unconditionally abstract all isomorphic fblocks with the same unintepreted function, for all equivalence classes of fblocks. Verify the resulting term-level designs. If the term-level verifier returns "VERIFIED", then return that result and terminate. However, if we get a counterexample, evaluate the counterexample on the word-level design to check if it is spurious. If non-spurious, return the counterexample, else go to Step 2.
2. Call Procedure CONDITIONALFUNCABSTRACTION to conditionally abstract to the term-level. Again, verify the resulting term-level designs, performing exactly the same checks as in Step 1 above: If the term-level verifier returns "VERIFIED", we return that result; otherwise, we return the counterexample only if it is non-spurious, going to Step 3 if it is spurious.
3. Invoke a word-level verifier on the original word-level designs.

The following theorem about ATLAS follows easily.

Theorem 2: ATLAS is sound and complete.

Proof: (sketch) Soundness follows from the fact that ATLAS only attempts to verify over-approximate abstractions of the original designs. Completeness follows because ATLAS only outputs a counterexample if it is evaluated to be a counterexample on the original word-level design. ■

Algorithm 1 summarizes our static abstraction procedure. Procedure CONDITIONALFUNCABSTRACTION takes two inputs. The first is the netlist $\mathcal{D}$ obtained by combining $\mathcal{D}_1$ and $\mathcal{D}_2$ to do equivalence or refinement checking. For equivalence checking, this is the standard "miter" circuit. For refinement checking, $\mathcal{D}$ is obtained by connecting inputs to $\mathcal{D}_1$ and $\mathcal{D}_2$ for use in symbolic simulation (e.g., a "flush" input to the pipeline for Burch-Dill style processor verification), as well as logic to compare the outputs of $\mathcal{D}_1$ and $\mathcal{D}_2$. The second input to CONDITIONALFUNCABSTRACTION is the set of all equivalence classes of fblocks $\mathcal{FS}$. Given these inputs, CONDITIONALFUNCABSTRACTION generates a rewritten netlist as output where some outputs of fblocks are conditionally rewritten as outputs of uninterpreted functions.

Algorithm 1 operates in two phases. In the first phase (lines 4-12), we identify outputs of fblocks that can be *unconditionally* abstracted with an uninterpreted function. This is performed by first computing, for every bit-vector output signal v in $\mathcal{FS}$,

the equivalence class of signals $\mathcal{E}(v)$ that its value flows to or which it is compared to. Table I lists the rules for computing $\mathcal{E}(v)$. Suppose there is no signal in $\mathcal{E}(v)$ that is assigned or compared to a bit-vector constant, or is the input or output of a bit-vector arithmetic or relational operator other than equality. This implies that the value of v does not flow to any bit-vector operation, arithmetic or relational, and is never compared with a specific bit-vector constant. In such a scenario, it is possible to always abstract v as the output of an uninterpreted function. Procedure ABSTRACTWITHUF listed as Algorithm 2 performs such a full function abstraction.

The second phase of Algorithm 1 comprises lines 13-21. We first remove from consideration fblocks all of whose (bit-vector) outputs have been abstracted using uninterpreted functions in phase 1 (line 13). Then, in each remaining fblock f_j, we compute an interpretation condition c_v for every bit-vector signal $v \in f_j$. Table II lists all rules for iteratively computing c_v. Most of the rules are intuitive, so we describe them only briefly. Consider rules 1,4, and 6: all of these involve a bit-vector operator or constant. Therefore, any signal involved in such an assignment is assigned an interpretation condition of $\mathbf{true}$. For equality comparisons or combinational assignments (rules 2 and 3), both sides of the comparison or assignment must have the same interpretation condition. For a multiplexor assignment (rule 5), the condition under which an input of the mux flows to its output is incorporated into the interpretation conditions. Rule 7, for a sequential assignment, makes use of special prev and next operators. The prev operator indicates that the condition is to be evaluated in the preceding cycle, whereas the next operator indicates that it must be evaluated in the following cycle. During symbolic simulation for term-level equivalence or refinement checking, these operators are translated to point to the conditions in the appropriate cycles. If we only symbolically simulate for a bounded number of cycles, it is sufficient to evaluate these rules a bounded number of times. Rules 8 and 9 deal with memory reads and writes. Finally, rules 10 and 11 handle the case where we have some fblock outputs replaced with uninterpreted functions, but not others; in this case, the interpretation conditions remain unchanged because we do not maintain a precise connection between inputs and outputs of uninterpreted functions.

Once the interpretation conditions are computed (by performing a bounded number of iterations of the rules in Table II), we use them to perform conditional function abstraction. Lines 20-21 of Algorithm 1 indicate the process: we first compute the disjunction of all conditions computed for output signals in an isomorphic tuple χ_j, and then use this disjunction oc_j within Procedure CONDITIONALABSTRACTWITHUF to compute the new output assignment for each element of χ_j as an conditional (ITE) expression.

The new design $\mathcal{D}'$ resulting from substitutions performed in Procedures ABSTRACTWITHUF and CONDITIONALABSTRACTWITHUF is the output of Procedure CONDITIONALFUNCABSTRACTION.

D. Illustrative Example

We illustrate the operation of our approach on the equivalence checking problem in Fig. 4. Note that all signals in this design have been given names from v_1 (A.out) to v_{17} (B.out).

Assume that the ALU modules have passed the first two steps in ATLAS: identifying replicated fblocks A.ALU and B.ALU and performing random simulation.

Assignment Type	Update Rule
Bitvector arithmetic operator: $v \leftarrow \mathsf{bvop}(v_1, \ldots, v_k)$	Merge $\mathcal{E}(v), \mathcal{E}(v_1), \ldots, \mathcal{E}(v_k)$
Relational operator: $\mathsf{bvrel}(v_1, \ldots, v_k)$	Merge $\mathcal{E}(v_1), \ldots, \mathcal{E}(v_k)$
Equality: $v_1 = v_2$	Merge $\mathcal{E}(v_1), \mathcal{E}(v_2)$
Sequential/combinational assignment: $v \leftarrow u$ $v := u$	Merge $\mathcal{E}(v), \mathcal{E}(u)$
Multiplexor assignment: $v \leftarrow \mathit{ITE}(b, v_1, v_2)$	Merge $\mathcal{E}(v), \mathcal{E}(v_1), \mathcal{E}(v_2)$
Memory operations: $v \leftarrow \mathbf{read}(M, u)$ $M := \mathbf{write}(M, u, v)$	Merge $\mathcal{E}(v)$, $\mathcal{E}(M)$ Merge $\mathcal{E}(M), \mathcal{E}(v)$

TABLE I
Rules for merging equivalence classes. We extend the $\mathcal{E}$ notation to memories also to track dependencies through memory reads and writes.

Algorithm 2 Procedure ABSTRACTWITHUF($\mathcal{F}$, χ): wholly abstract outputs in χ with uninterpreted functions.

1: // Input: Equivalence class of functional blocks, $\mathcal{F} = \{f_1, f_2, \ldots, f_l\}$
2: // Input: Tuple of isomorphic output signals of f_i's, $\chi = (v_1, v_2, \ldots, v_l)$
3: // Output: Updated functional blocks $\mathcal{F}'$.
4: Create a fresh uninterpreted function symbol UF_χ.
5: **for** each output signal $v_i \in \chi$ **do**
6: Let $(i_1, \ldots, i_{k_i})$ denote the input symbols to fblock f_i.
7: Replace the assignment $v_i \leftarrow e$ in f_i with the assignment $v_i \leftarrow UF_\chi(i_1, \ldots, i_{k_i})$.
8: Transitively delete all assignments $u \leftarrow e$ or $u := e$ in f_i where signal u does not appear on the right-hand side of any assignment in f_i.
9: Denote the resulting fblock by f_i'.
10: **end for**
11: Return the updated equivalence class of fblocks $\mathcal{F}' = \{f_1', f_2', \ldots, f_l'\}$.

We describe how procedure CONDITIONALFUNCABSTRACTION operates on this example. The first phase of CONDITIONALFUNCABSTRACTION computes equivalence classes of the output signals v_1 and v_{17} of the two ALUs. We observe that

$$\begin{aligned}\mathcal{E}(v_1) &= \mathcal{E}(v_{17}) \\ &= \{v_1, v_2, v_4, v_5, v_{13}, v_{10}, v_9, v_{12}, v_{16}, v_{14}, v_{15}, v_{11}, v_{17}\}\end{aligned}$$

Clearly, since some of the above signals are outputs or inputs of bit-vector arithmetic operators such as $+$ and bit-extraction, we cannot abstract the two ALUs unconditionally with an uninterpreted function.

Therefore, CONDITIONALFUNCABSTRACTION performs the second phase: computing interpretation conditions for the signals in Designs A and B.

As stated in the caption of Table II, all conditions are initialized to $\mathbf{false}$.

Next, consider all signals that are inputs or outputs of bit-vector

Rule No.	English Description	Form of Assignments	Rules for Updating Interpretation Condition
1.	Bit-vector constant	$v \leftarrow c$	$\mathsf{c}'_v := \mathbf{true}$
2.	Combinational copy	$v \leftarrow u$	$\mathsf{c}'_v := \mathsf{c}_v \vee \mathsf{c}_u$ $\mathsf{c}'_u := \mathsf{c}_v \vee \mathsf{c}_u$
3.	Equality comparison	$b \leftarrow v = u$	$\mathsf{c}'_v := \mathsf{c}_v \vee \mathsf{c}_u$ $\mathsf{c}'_u := \mathsf{c}_v \vee \mathsf{c}_u$
4.	Bit-vector relational operator	$b \leftarrow \mathsf{bvrel}(v_1, v_2, \ldots, v_k)$	$\mathsf{c}'_{v_i} := \mathbf{true}$ $\forall i = 1, 2, \ldots, k$
5.	Multiplexor assignment	$v \leftarrow \mathit{ITE}(b,\ v_1,\ v_2)$	$\mathsf{c}'_v := \mathsf{c}_v \vee (b \wedge \mathsf{c}_{v_1} \vee \neg b \wedge \mathsf{c}_{v_2})$ $\mathsf{c}'_{v_1} := \mathsf{c}_{v_1} \vee (b \wedge \mathsf{c}_v)$ $\mathsf{c}'_{v_2} := \mathsf{c}_{v_2} \vee (\neg b \wedge \mathsf{c}_v)$
6.	Bit-vector operator	$v \leftarrow \mathsf{bvop}(v_1, v_2, \ldots, v_k)$	$\mathsf{c}'_v := \mathbf{true}$ $\mathsf{c}'_{v_i} := \mathbf{true}\ \forall i = 1, 2, \ldots, k$
7.	Sequential assignment	$v := u$	$\mathsf{c}'_v := \mathsf{c}_v \vee \mathsf{prev}(\mathsf{c}_u)$ $\mathsf{c}'_u := \mathsf{c}_u \vee \mathsf{next}(\mathsf{c}_v)$
8.	Memory read	$v \leftarrow \mathbf{read}(M, u)$	$\mathsf{c}'_v := \mathsf{c}_v \vee \mathsf{c}_M$ $\mathsf{c}'_M := \mathsf{c}_v \vee \mathsf{c}_M$
9.	Memory write	$M := \mathbf{write}(M, v_a, v_d)$	$\mathsf{c}'_M := \mathsf{c}_M \vee \mathsf{prev}(\mathsf{c}_{v_d})$ $\mathsf{c}'_{v_d} := \mathsf{next}(\mathsf{c}_M) \vee \mathsf{c}_{v_d}$
10.	Uninterpreted function	$v \leftarrow UF(v_1, \ldots, v_k)$	$\mathsf{c}'_v := \mathsf{c}_v$, $\mathsf{c}'_{v_i} := \mathsf{c}_{v_i}\ \forall i = 1, \ldots, k$
11.	Uninterpreted predicate	$b \leftarrow UP(v_1, \ldots, v_k)$	$\mathsf{c}'_{v_i} := \mathsf{c}_{v_i}\ \forall i = 1, \ldots, k$

TABLE II

Rules for computing the *interpretation condition* c_v for every bit-vector (or memory) signal v (or M) in a set of signals V. Every condition c_v initially starts out as **false**. c'_x denotes the updated value of c_x for a bit-vector or memory signal x.

Algorithm 3 Procedure CONDITIONALABSTRACTWITHUF($\mathcal{F}$, χ, oc): conditionally abstract outputs in χ with uninterpreted functions using condition oc.

1: // Input: Equivalence class of functional blocks, $\mathcal{F} = \{f_1, f_2, \ldots, f_l\}$
2: // Input: Tuple of isomorphic output signals of f_i's, $\chi = (v_1, v_2, \ldots, v_l)$
3: // Input: Boolean condition: $\mathsf{oc} := \bigvee_{i=1}^{l} \mathsf{c}_{v_i}$.
4: // Output: Updated functional blocks $\mathcal{F}'$.
5: Create a fresh uninterpreted function symbol UF_χ.
6: **for** each output signal $v_i \in \chi$ **do**
7: Let $(i_1, \ldots, i_{k_i})$ denote the input symbols to fblock f_i.
8: Replace the assignment $v_i \leftarrow e$ in f_i with the assignment $v_i \leftarrow \mathit{ITE}(\mathsf{oc},\ e,\ UF_\chi(i_1, \ldots, i_{k_i}))$.
9: Denote the resulting fblock by f'_i.
10: **end for**
11: Return the updated equivalence class of fblocks $\mathcal{F}' = \{f'_1, f'_2, \ldots, f'_l\}$.

operators, or compared with a bit-vector constant (such as JMP). We apply Rules 1 and 6 to these signals, to get:

$$\mathsf{c}_{v_2} = \mathsf{c}_{v_5} = \mathsf{c}_{v_8} = \mathsf{c}_{v_6} = \mathsf{c}_{v_7} = \mathbf{true} \text{ and}$$
$$\mathsf{c}_{v_{13}} = \mathsf{c}_{v_{10}} = \mathsf{c}_{v_{14}} = \mathsf{c}_{v_1} = \mathsf{c}_{v_{15}} = \mathbf{true}$$

Since we have the assignments $v_5 := v_4$ and $v_{13} := v_{12}$, we can apply Rule 7 to obtain

$$\mathsf{c}_{v_4} = \mathsf{next}(\mathsf{c}_{v_5}) = \mathbf{true} \text{ and } \mathsf{c}_{v_{12}} = \mathsf{next}(\mathsf{c}_{v_{13}}) = \mathbf{true}$$

Now, using Rule 5 for the multiplexor in Design A, we obtain

$$\mathsf{c}_{v_1} = \{(v_7 = \mathsf{JMP}) \wedge \mathsf{c}_{v_4}\} = (\mathtt{A.instr}[19:16] = \mathsf{JMP})$$

Finally, using Rule 3 for the equality corresponding to out_ok, we conclude that $\mathsf{c}_{v_{17}} = \mathsf{c}_{v_1} = (\mathtt{A.instr}[19:16] = \mathsf{JMP})$.

The computation of interpretation conditions terminates here. We can thus compute a partial abstraction of the ALUs using a fresh uninterpreted function symbol UF by employing the new assignments below:

$$v_1 \leftarrow \mathit{ITE}(\mathtt{A.instr}[19:16] = \mathsf{JMP},\ ALU(v_8),\ UF(v_8))$$
$$v_{17} \leftarrow \mathit{ITE}(\mathtt{A.instr}[19:16] = \mathsf{JMP},\ ALU(v_{16}),\ UF(v_{16}))$$

Note that ALU above refers to the original ALU as shown in Fig. 1(a). The right-hand side expressions in the new assignments shown above are instances of the partially-abstracted ALU shown in Fig. 1(c).

In summary, for our running example, ATLAS correctly computes the conditions under which the ALU can be abstracted with an uninterpreted function, as we discussed in Sec. III.

V. EXPERIMENTAL RESULTS

A. Benchmarks

In addition to the running example, we performed experiments on four benchmarks: a simple pipelined processor [7], the packet disassembler from the USB 2.0 function core [19], a power-gated calculator design [21], and the Y86 processor designs [9]. We describe the additional benchmarks here.

Pipelined Datapath. (PIPE) The simple pipelined processor consists of 3-stages: fetch, execute, and writeback. It supports 7 arithmetic instructions and has a 32x32-bit, dual-read, single-write register file. The design we use here differs from the one in the UCLID manual [7] only in that it does not use memory abstraction for the register file. We verify that the pipelined processor refines (i.e., is simulated by) a single-cycle, sequential version of the processor. The shared state variables that are checked for equality are the program counter (PC) and register file (RF). Excluding the top-level processor modules, there are 3 candidate modules for abstraction: PC update, RF, and arithmetic-logic unit (ALU). The PC update and ALU modules both pass the random simulation stage of ATLAS, however, we only abstract the ALU due to the

small size of the PC update module. The RF module does not pass random simulation and, hence, we don't abstract it. By replacing the RF with a combinational random function, we lose the ability to store values, which causes random simulation to fail.

USB controller. (USB) For the USB design [19], we created a refinement of the original packet disassembler. In the refined version, we removed the notion of a TOKEN packet and updated the state machine and other relevant logic accordingly. We performed bounded equivalence checking on the original and refined packet disassemblers by injecting tokens on each cycle. The property checked was that the disassembler state, error condition state, and cyclic redundancy check (CRC) error signals were the same for each cycle. The two candidate modules for abstraction were the 16- and 5-bit CRC modules. Both passed random simulation, which is expected because neither influences the state machine control, however, only the 16-bit CRC module was abstracted because the 5-bit CRC module is not in the cone-of-influence of the property being checked.

Calculator. (CALC) This design has 4 input and output ports and accepts 4 instructions: add, subtract, shift left, and shift right. Each port can have up to 4 outstanding instructions. A two-bit tag is used to keep track of outstanding instructions. For this experiment, we created a power-gated version of an existing calculator design, in a manner similar to that in [14]. In the power-gated version, the adder-subtractor unit (ASU) is powered down (by fencing the outputs) whenever there are no add or subtract instructions in the add/subtract queue. We performed equivalence checking on the outputs of the two versions to make sure that the correct results come out in the same order, with the proper tags, and on the correct ports. For this design, there are only 2 modules which passed random simulation: the ASU and the shifter. There are many modules which didn't pass random simulation. An example is the priority module. The priority module takes the incoming commands and adds them to the appropriate queues and dispatches commands to the appropriate unit (ASU or shifter). The priority module has state holding elements which combinational functions can not properly represent.

Y86 processor. (Y86) The Y86 processor is a pipelined CISC microprocessor design described by Bryant and O'Hallaron [9], [11]. Sequential and several pipelined implementations of the Y86 design are available from the textbook website [11]; the netlists we used are the versions after memory abstraction was performed. Similar to PIPE, we check that the pipelined implementations refine the sequential version. Only the ALU module survived the random simulation phase.

B. Results

The hypothesis we test with our experiments is that performing automatic term-level abstraction before verification can yield substantial speedups over verifying the original word-level design. While we would have liked to compare with the Reveal or Vapor systems, they are not publicly available. Our own experience with performing fine-grained term-level abstraction as with Reveal/Vapor is that there are far too many spurious counterexamples generated to yield any improvements, especially given the recent advances in bit-vector SMT solvers.

Our experiments were performed by first extracting ATLAS netlist representations from the Verilog RTL. Random simulations were performed using the Icarus Verilog simulator [22]. We used $T = 1000$ random functions for each equivalence class of fblocks, selecting a class for function abstraction if at most $\tau = 50$ (5%) simulations failed. ATLAS translates both word-level and term-level netlists into UCLID format [1], before using UCLID's symbolic simulation engine to perform bounded equivalence checking or refinement (correspondence) checking of processor designs. Experiments were run on a Linux workstation with 64-bit 3.0 GHz Xeon processors and 2 GB of RAM.[1]

Some characteristics of the benchmarks are given in the first six columns of Table III. The size of the designs is described in terms of the numbers of latches as well as the number of signals in the word-level netlist (based on ATLAS' representation). In general, random simulation was very effective at pruning out fblocks that cannot be replaced with uninterpreted functions. For the PIPE, USB, and Y86 designs, only two fblocks survived the results of random simulation, both being instantiations of the same Verilog module (one in each circuit in the equivalence/refinement check). For the CALC, the ADD/SUB as well as the Shifter fblocks could be abstracted, again symmetrically on each side of the miter.

Once candidate fblocks are identified for abstraction, ATLAS generates word-level and term-level UCLID models using the approach outlined in Sec. IV-C. UCLID is used to perform symbolic simulation. For refinement checking of processor designs, the number of cycles of symbolic simulation is defined by the Burch-Dill approach [12] and based on the pipeline depth. For equivalence checking tasks, we performed symbolic simulation for various numbers of cycles. Both verification tasks, at the end, generate a decision problem in a combination of logical theories. For word-level models, this problem is in the theory of finite-precision bit-vector arithmetic, possibly including the theory of arrays if memory abstraction is performed (as for Y86 benchmarks). For term-level models, the decision problem is in the combination of bit-vector arithmetic, uninterpreted functions, and arrays. We experimented with several SMT solvers for this combination of theories, including Boolector, MathSAT, and Yices, three of the top solvers in the 2008 and 2009 SMTCOMP competition [13]. We present our results for Boolector [8], the SMT solver that performs best for the word-level designs.

The experimental results are presented in the last 9 columns of Table III. Consider the last two columns of the table. Here we present two ratios: "SMT" indicates the speedup of running Boolector on ATLAS output versus the original design. We observe that we get a speedup on all benchmarks, ranging from a factor of 2 to 92. However, when the running time for random simulations and static analysis is factored in ("Total"), we observe that ATLAS does worse on the USB design, and has a somewhat smaller speedup on the other designs. The main reason is the time spent in random simulation. We believe there is scope for optimizing the performance of the random simulator, as well as amortizing simulation time across different formal verification runs.

We also experimented with a purely SAT-based approach. Here the word-level problems are bit-blasted to a SAT problem. For term-level problems, UCLID's internal elimination of uninterpreted functions (using the "Ackermann method") is first invoked, and then the resulting word-level problem is bit-blasted to SAT.

[1] A more detailed description of experimental data and benchmarks are available at http://uclid.eecs.berkeley.edu/atlas/.

Name	*Benchmark Characteristics*					*Performance Comparison*									
	L	N_{orig}	N_{fb}	N_{abs}	Abs	N	Word-Level (sec.)		ATLAS & Term-Level (sec.)					Speedup	
							SMT	Total	Iter	RSim	Static	SMT	Total	Total	SMT
PIPE	2233	1233	2	42	ALU (2)	9	171.09	171.22	0	3.71	0.20	1.86	5.77	29.7	92.0
USB	134	892	2	252	CRC16 (2)	15	0.29	0.38	0	3.52	0.06	0.11	3.69	0.1	2.6
						25	0.53	0.62	0	5.63	0.06	0.20	5.89	0.1	2.7
CALC	5539	2913	4	54	Add/Sub (2), Shifter (2)	15	11.82	12.64	0	8.43	0.88	2.40	11.71	1.1	4.9
						25	133.72	134.76	0	14.14	1.16	23.86	39.16	3.4	5.6
Y86-BTFNT	567	936	2	36	ALU (2)	13	1077.72	1077.84	1	5.20	0.09	1385.34	1390.63	0.7	0.7
Y86-FULL	567	961	2	36	ALU (2)	13	2166.66	2166.78	0	4.44	0.09	56.30	60.83	35.6	38.5
Y86-LF	567	931	2	36	ALU (2)	13	728.05	728.17	0	4.18	0.09	42.11	46.38	15.7	17.3
Y86-NT	567	928	2	36	ALU (2)	13	1736.66	1736.77	1	4.37	0.08	1350.95	1355.40	1.28	1.28
Y86-STD	567	923	2	36	ALU (2)	13	1239.00	1239.12	0	5.22	0.08	54.19	59.49	20.8	22.9

TABLE III

Performance Comparison and Benchmark Characteristics. Column headings are as follows: L: Number of latches in original word-level netlist; N_{orig}: Number of signals in word-level netlist; N_{fb}: Number of fblocks selected by random simulation; N_{abs}: Total number of signals in the selected fblocks, Abs are the names of the RTL module abstracted (with number of instances); "Word-level" indicates columns for verification of original word-level model; "Term-level" indicates columns for verification of ATLAS-generated term-level model; N: Number of steps of symbolic simulation for equivalence/refinement checking; SMT indicates the time taken by the Boolector SMT solver; Iter is the number of iterations of interpretation condition computation; RSim is the runtime for random simulation; Static is the time for ATLAS' static analysis; Total indicates the total verifier time (for the word-level model, this includes SMT time, for the term-level model, this includes RSim, Static, and SMT times); "Speedup": the speedup of the term-level verification over the word-level verification tasks, for both SMT time and Total time.

We experimented with several SAT engines, including MiniSat, PicoSat, and Precosat. Table IV reports the SAT problem sizes and run-times for a selected subset of generated SAT problems. The run-time of the best SAT solver is reported for each run. For the PIPE and USB examples, term-level abstraction by ATLAS performs significantly better even when the SAT problem size is much bigger. This indicates the benefit of abstracting modules such as CRC16 which can have operators such as XORs that are hard for SAT engines. For the CALC example, the SAT solvers perform better on the original word-level model, which is understandable, since reasoning about ADD/SUB/SHIFT is not particularly hard for SAT engines, and the impact of SAT problem size is observed.

Name	N	Abs?	SAT Size		Run-time
			#Vars	#Clauses	(sec.)
PIPE	9	No	41911	122203	>3600
		Yes	45644	133084	29.86
USB	25	No	17667	51916	>3600
		Yes	159509	475057	68.74
CALC	25	No	351892	1039501	823.71
		Yes	753164	2234485	1771.66

TABLE IV

Performance Comparison of SAT-based Verification. N is the number of cycles symbolically simulated. "Abs?" indicates whether term-level abstraction via ATLAS was used or not.

VI. Conclusion

We presented an automatic approach to perform term-level abstraction of RTL designs. Our results indicate that verification time can be greatly reduced in many cases. For future work, we plan to expand the approach to "sequential" uninterpreted functions, as well as combining it with data abstraction.

Acknowledgments. This research was supported in part by SRC contracts 1355.001 and 2045.001 and by an Alfred P. Sloan Research Fellowship.

References

[1] UCLID Verification System. Available at http://uclid.eecs.berkeley.edu.

[2] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In *Proceedings of ASP-DAC*, pages 19–24, 2006.

[3] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. CEGAR-based formal hardware verification: A case study. Technical Report CSE-TR-531-07, University of Michigan, May 2007.

[4] Z. S. Andraus and K. A. Sakallah. Automatic abstraction and verification of Verilog models. In *Proceedings of the 41st Design Automation Conference (DAC)*, pages 218–223, 2004.

[5] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.

[6] P. Bjesse. A practical approach to word level model checking of industrial netlists. In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 446–458, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] B. A. Brady, S. A. Seshia, S. K. Lahiri, and R. E. Bryant. *A User's Guide to UCLID Version 3.0*, October 2008.

[8] R. D. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *In* Proc. of TACAS, March 2009.

[9] R. E. Bryant. Term-level verification of a pipelined CISC microprocessor. Technical Report CMU-CS-05-195, Computer Science Department, Carnegie Mellon University, 2005.

[10] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 78–92, July 2002.

[11] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice-Hall, 2002. Website: http://csapp.cs.cmu.edu.

[12] J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In D. L. Dill, editor, *Computer-Aided Verification (CAV '94)*, LNCS 818, pages 68–80. Springer-Verlag, June 1994.

[13] S. Competition. http://www.smtcomp.org/.

[14] C. Eisner, A. Nahir, and K. Yorav. Functional verification of power gated designs by compositional reasoning. In *CAV*, pages 433–445, 2008.

[15] W. A. Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

[16] P. Johannesen. BOOSTER: Speeding up RTL property checking of digital designs through word-level abstraction. In *Computer Aided Verification*, 2001.

[17] S. K. Lahiri and R. E. Bryant. Deductive verification of advanced out-of-order microprocessors. In *Proc. 15th International Conference on Computer-Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 341–354, 2003.

[18] P. Manolios and S. K. Srinivasan. Refinement maps for efficient verification of processor models. In *Design, Automation, and Test in Europe (DATE)*, pages 1304–1309, 2005.

[19] Opencores.org. Usb controller. http://www.opencores.org/project,usb.

[20] L.-S. Peh. *Flow Control and Micro-Architectural Mechanisms for Extending the Performance of Interconnection Networks*. PhD thesis, Stanford University, August 2001.

[21] B. Wile, J. Goss, and W. Roesner. *Comprehensive Functional Verification: The Complete Industry Cycle*. Morgan Kaufmann, 2005.

[22] S. Williams. Icarus verilog. http://www.icarus.com/eda/verilog/.