Learning Conditional Abstractions

Bryan A. Brady[1]
IBM
Poughkeepsie, NY 12601

Randal E. Bryant
Carnegie Mellon University
randy.bryant@cs.cmu.edu

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

***Abstract*—Abstraction is central to formal verification. In term-level abstraction, the design is abstracted using a fragment of first-order logic with background theories, such as the theory of uninterpreted functions with equality. The main challenge in using term-level abstraction is determining what components to abstract and under what conditions. In this paper, we present an automatic technique to conditionally abstract register transfer level (RTL) hardware designs to the term level. Our approach is a layered approach that combines random simulation and machine learning inside a counter-example guided abstraction refinement (CEGAR) loop. First, random simulation is used to determine modules that are candidates for abstraction. Next, machine learning is used on the resulting simulation traces to generate candidate conditions under which those modules can be abstracted. Finally, a verifier is invoked. If spurious counter-examples arise, we refine the abstraction by performing a further iteration of random simulation and machine learning. We present an experimental evaluation on processor designs.**

I. Introduction

Designs are usually specified at the register-transfer-level (RTL). For formal verification, however, RTL can be a rather low level of abstraction where data are represented as bits and bit vectors, and operations on 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 leads to state-space explosion, necessitating additional abstraction.

Term-level modeling can make formal verification of data-intensive properties tractable by abstracting away details of data representations and operations, viewing data as symbolic *terms* and operations as *uninterpreted* functions. Term-level abstraction has been found to be especially useful in microprocessor design verification [14], [18], [20], [21]. The precise functionality 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 [5] are then 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 bit-vector-level (word-level) RTL. Two recent efforts have sought to automate the generation of term-level models. Andraus and Sakallah [4] were the first to address the problem, proposing a counterexample-guided abstraction refinement (CEGAR) approach. While the CEGAR technique works in some cases, it can require very many iterations of abstraction-refinement in other situations. Brady et al. [8] proposed ATLAS, an approach that combines random simulation with static analysis to compute *interpretation conditions* — conditions under which a functional block is replaced with an uninterpreted function. ATLAS avoids the need for several abstraction-refinement iterations by computing conservative interpretation conditions using static analysis. However, in some cases, these conditions are so large as to negate the advantages of term-level verification over word-level methods.

In this paper, we present CAL, a new technique for *automatically generating a term-level verification model* from a word-level description. The main focus of this work is function abstraction. Similar to ATLAS, CAL conditionally abstracts functional blocks in the original design with uninterpreted functions. In contrast with previous work, CAL uses a novel layered approach based on a combination of *random simulation*, *machine learning*, and *counterexample-guided abstraction-refinement*. In the first stage, we exploit the module structure specified by the designer using random simulation to identify functional blocks corresponding to module instantiations that are suitable for abstraction with uninterpreted functions. Replacing functional blocks with uninterpreted functions is always sound, that is, the correctness of the resulting abstract design implies the correctness of the original design. However, this abstraction loses information and can lead to spurious counterexamples. In the second stage, we use machine learning inside a CEGAR loop to rule out such spurious counterexamples. First, a verifier is invoked on the unconditionally abstracted verification model. If spurious counterexamples arise, machine learning is used to compute conditions under which abstraction can be performed without loss of precision; i.e., if the resulting term-level design is in correct, then so is the original word-level design. This process is repeated until we arrive with a term-level model that is valid or a legitimate counterexample is found. Fig. 1 illustrates the CAL approach.

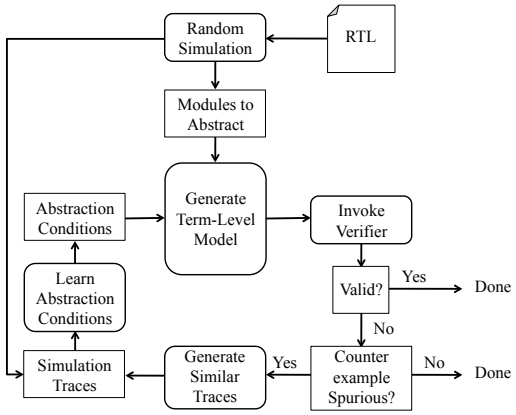

Fig. 1. **The CAL approach** A CEGAR-based approach, CAL identifies candidate abstractions with random simulation and uses machine learning to refine the abstraction if necessary.

[1]This work was conducted while the author was affiliated with the University of California, Berkeley.

We present experimental evidence that our approach is efficient and that the resulting term-level models are easier to verify. Moreover, we show that the abstraction conditions that we learn are as good or better than the previous best-known conditions.

The rest of this paper is organized as follows. We discuss some background material and related work in Section II. In Section III, we present the formal model for our work as well as some relevant ideas borrowed from our previous work on ATLAS [8]. Our new approach, CAL, is described in Section IV. Case studies are discussed in detail in Section V. We conclude in Section VI.

II. Background and Related Work

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

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 [8]:

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 is, they must evaluate to the same values when applied to 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 (applied selectively) 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 [14] or lambda expressions [12]. We do not address automatic memory abstraction in this paper.

B. Function Abstraction

The concept of function abstraction is illustrated using a toy ALU design [8]. Consider the simplified ALU shown in Figure 2(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 16-bit input and generates as output the resulting 16-bit result.

Using very coarse-grained term-level abstraction, one could abstract the entire ALU module with a single uninterpreted function (UF), as shown in Figure 2(b). However, we lose the precise mapping from `instr` to `out`.

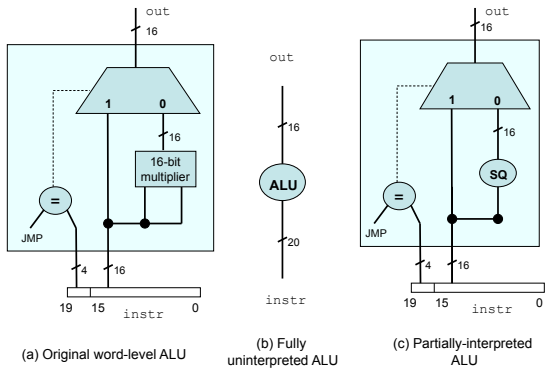

Fig. 2. **Three versions of an ALU design.** Boolean signals are shown as dashed lines and bit-vector signals as solid black lines [8].

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 2(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 large burden on the designer. It is this burden that we seek to mitigate using the approach presented in this paper.

C. 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 [12]. Vapor uses a counterexample-guided abstraction-refinement (CEGAR) approach [4]. Vapor has since been 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, and then iteratively rule out spurious counterexamples. While the CEGAR approach has shown much promise [3], in many cases, however, several abstraction-refinement iterations are needed to infer fairly straightforward properties of data, thus imposing a significant overhead [8]. While the approach presented in this paper is also counterexample-guided, we require very few refinement iterations in practice.

A more recent approach to automatic abstraction is ATLAS [8]. ATLAS exploits the module structure specified by the designer and uses random simulation to determine module instantiations that are candidates for function abstraction. ATLAS uses static analysis to heuristically compute *interpretation conditions* that specify when a functional block must be represented precisely.

While this works in many cases, for some benchmarks the interpretation conditions can grow extremely large, leading to poor performance [8]. Our approach, CAL, addresses this limitation by using a dynamic approach based on machine learning. As is the case with ATLAS, the CAL approach can be combined with bit-width reduction techniques (e.g. [7], [19]) to perform combined function and data abstraction.

To our knowledge, Clarke, Gupta et al. [15], [17] were the first to use machine learning to compute abstractions for model checking. Our work is similar in spirit to theirs. One difference is that we generate abstract, term-level models for SMT-based verification, whereas their work focuses on bit-level model checking and localization abstraction. Consequently, the learned concept is different: CAL learns Boolean interpretation conditions whereas their technique learns sets of variables to make visible. Additionally, our use of machine learning is more direct — e.g., while Clarke et al. [15] also use decision tree learning, they only indirectly use the learned decision tree (all variables branched upon in the tree are made visible), whereas we use the Boolean function corresponding to the entire tree as the learned interpretation condition.

III. Preliminaries

We adopt the formal model used in [8]. In Sec. III-A, we present only the elements of this formal model necessary for the rest of the paper. An illustrative example is given in Sec. III-B.

A. Basic Definitions

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

Input and output signals are assumed combinational, without loss of generality. 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. Similarly, a *sequential assignment* is a rule of the form $v := u$, where u is a signal. Signals v, u and expressions e are of three types: bit-vector, Boolean, or memory. For brevity, we omit the detailed syntax (see [8] for this), and present only the notation used in the paper. In word-level netlists, a memory is modeled as a flat array of bit-vector signals.

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*.

A *term-level netlist* is a generalization of a word-level netlist where bit-vector and Boolean expressions can include applications of uninterpreted functions and predicates, written $UF(v_1, \ldots, v_k)$ and $UP(v_1, \ldots, v_k)$ for $k \geq 0$, and memory operations can be modeled in a suitable theory of arrays/memories using the usual **read** and **write** functions or lambda expressions [12].

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) $\mathcal{T}_2$.

We follow the approach taken by ATLAS and generate the term-level abstraction by computing an *interpretation condition* — a condition under which we will retain the precise fblock in the term-level model (i.e, we replace the fblock by an uninterpreted function under the negation of the interpretation condition). The idea of conditional function abstraction is illustrated in Figure 3. The original word-level circuit is shown in Fig. 3(a) and the conditionally abstracted version with interpretation condition c is shown in Fig. 3(b).

In Section IV, we present our CAL approach to automatically generate term-level abstract models. In Section V, we show that using CAL 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 [8]. Two variants of the same circuit, denoted Design A and Design B, are to be checked for output equivalence.

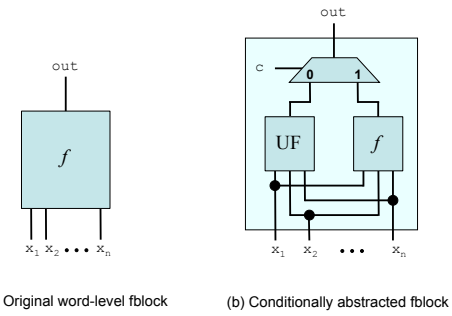

Fig. 3. **Conditional abstraction** (a) Original word-level fblock f. (b) Conditionally abstracted version of f with interpretation condition c

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. The ALU is similar to the ALU design shown in Figure 2(a) – both ALUs pass the target address through unaltered when the instruction is a jump. 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`$[15:0]$.

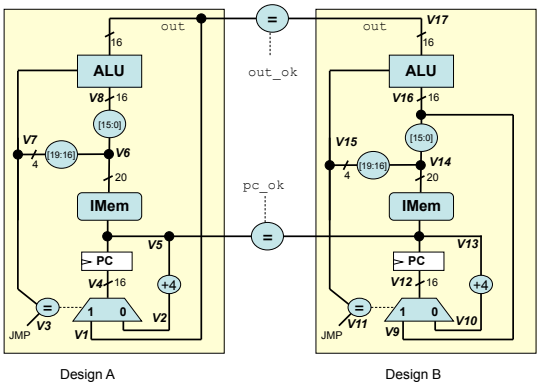

Fig. 4. **Equivalence checking of two versions of a portion of a processor design.** Boolean signals are shown as dashed lines and bit-vector signals as solid lines [8].

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 2(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 2(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 2(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 CAL Approach

The main contribution of this paper is presented in this section. The goal of this step is to compute conditions under which it is precise to abstract using a machine-learning-based CEGAR loop.

A. Identifying Candidate fblocks

The first step in CAL is the same as in ATLAS: to use syntactic matching and random simulation to identify a set of fblocks that are candidates for replacement with uninterpreted functions. We review this procedure in this section since it is crucial to understand the rest of the CAL procedure.

The first step in identifying candidates for abstraction is to identify *replicated fblocks*. A replicated fblock is an fblock in $\mathcal{D}_1$ that has an isomorphic counterpart in $\mathcal{D}_2$. A formal definition can be found in [8]. In equivalence and refinement checking problems, identifying replicated fblocks is typically a matter of finding instances of the same RTL module present in both designs.

The fblock identification process generates a collection of sets of fblocks $\mathcal{FS} = \{\mathcal{F}_1, \mathcal{F}_2, ..., \mathcal{F}_k, \}$. Each set $\mathcal{F}_j$ contains replicated fblocks that are isomorphic to each other. $\mathcal{F}_j$ can be viewed as an equivalence class of the fblocks it contains. In later steps when function abstractions are computed, it is important to note that the same function abstraction is used for each fblock in $\mathcal{F}_j$.

The next step in the abstraction candidate identification process is to determine which equivalence classes $\mathcal{F} \in \mathcal{FS}$ will be considered for abstraction. This is accomplished using random simulation.

Fix an equivalence class $\mathcal{F}$. Let its cardinality be l. Let $f_i \in \mathcal{F}$ be an arbitrary fblock with 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 $\chi_j = (v_{1j}, v_{2j}, \ldots, v_{lj})$, for each $j = 1, 2, \ldots, m$, as a tuple of *isomorphic output signals*.

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

For each fblock $f_i \in \mathcal{F}$, $i = 1, 2, \ldots, l$, we replace the assignment to the output signal v_{ij} with the random assignment $v_{ij} \leftarrow RF_{\chi_j}(u_{i1}, \ldots, u_{in})$. 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. Note that all other fblocks

not in $\mathcal{F}$ are interpreted precisely. This process is repeated for $\mathcal{F}$ using T different random functions RF_{χ_j}.

If the fraction of failing verification runs is greater than a threshold τ, then we drop the equivalence class $\mathcal{F}$ from further consideration. Otherwise, we retain $\mathcal{F}$ for further analysis, as described in the following section. It is important to note that we have not yet decided to replace fblocks in $\mathcal{F}$ with uninterpreted functions – this will be determined later using the counterexample-guided loop. We denote the set of equivalence classes that are to be considered for abstraction as $\mathcal{FS}_{\mathcal{A}}$.

B. Top-Level CAL *Procedure*

The top-level CAL procedure, VERIFYABS, is shown in Algorithm 1. VERIFYABS takes two arguments, the design $\mathcal{D}$ being verified (which includes both designs – e.g., it is the miter for equivalence checking) and the set of equivalence classes being abstracted $\mathcal{FS}_{\mathcal{A}}$. Initially, the interpretation conditions $\mathsf{c}_i \in \mathcal{IC}$ are set to **false** meaning that we start by unconditionally abstracting the fblocks in $\mathcal{D}$. The procedure CONDABS creates the abstracted term-level design $\mathcal{T}$ from the word-level design $\mathcal{D}$, the set of equivalence classes to be abstracted $\mathcal{FS}_{\mathcal{A}}$, and the set of interpretation conditions $\mathcal{IC}$. Next, we invoke a term-level verifier on $\mathcal{T}$. If VERIFY $(\mathcal{T})$ returns "Valid", we report that result and terminate. If a counterexample arises, we evaluate the counterexample on the word-level design. If the counterexample is non-spurious, we report the counterexample and terminate, otherwise we store the counterexample in $\mathcal{CE}$ and invoke the abstraction condition learning procedure, LEARNABSCONDS $(\mathcal{D}, \mathcal{FS}_{\mathcal{A}}, \mathcal{CE})$.

We say that VERIFYABS is *sound* if it reports "Valid" only if $\mathcal{D}$ is correct. It is *complete* if it reports a true counterexample when $\mathcal{D}$ is incorrect. We have the following guarantee for the procedure VERIFYABS:

Theorem 1: If VERIFYABS terminates, it is sound and complete.

Proof: Any term-level abstraction is a sound abstraction of the original design, since any partially-interpreted function (for any interpretation condition) is a sound abstraction of the fblock it replaces. Thus VERIFYABS is sound. Moreover, VERIFYABS terminates with a counterexample only if it deems the counterexample to be non-spurious, by simulating it on the concrete design $\mathcal{D}$. Therefore VERIFYABS is complete. ■

In order to guarantee termination of VERIFYABS, we must impose certain constraints on the learning algorithm LEARNABSCONDS. This is formalized in the theorem below.

Theorem 2: Suppose that the learning algorithm LEARNABSCONDS satisfies the following properties:

(i) If c_i denotes the interpretation condition for an fblock learned in iteration i of the VERIFYABS loop, then $\mathsf{c}_i \implies \mathsf{c}_{i+1}$ and $\mathsf{c}_i \neq \mathsf{c}_{i+1}$;
(ii) The trivial interpretation condition **true** belongs to the hypothesis space of LEARNABSCONDS, and
(iii) The hypothesis space of LEARNABSCONDS is finite.

Then, VERIFYABS will terminate and return either Valid or a non-spurious counterexample.

Proof: Consider an arbitrary fblock that is a candidate for function abstraction. Let the sequence of interpretation conditions generated in successive iterations of the VERIFYABS loop be $\mathsf{c}_0 = \mathbf{false}, \mathsf{c}_1, \mathsf{c}_2, \ldots$. By condition (i), $\mathsf{c}_0 \implies \mathsf{c}_1 \implies \mathsf{c}_2 \implies \ldots$ where $\mathsf{c}_i \neq \mathsf{c}_{i+1}$. Since no two elements of the sequence are equal, and the hypothesis space is finite, no element of the sequence can repeat. Thus, the sequence (for any fblock) forms a finite chain of implications. Moreover, since **true** belongs to the hypothesis space, in the extreme case, VERIFYABS can generate in its final iteration the term-level design $\mathcal{T}$ identical to the original design $\mathcal{D}$, which will yield termination with either Valid or a non-spurious counterexample. ■

In practice, the conditions (i)-(iii) stated above can be implemented on top of any learning procedure. The most straightforward way is to set an upper bound on the number of iterations that LEARNABSCONDS can be invoked, after which the interpretation condition is set to **true**. Another option is to set c_{i+1} to $\mathsf{c}_i \vee d_{i+1}$ where d_{i+1} is the condition learned in the $i+1$th iteration. Yet another option is to keep a log of the interpretation conditions generated, and if an interpretation condition is generated for a second time, the abstraction procedure is terminated by setting the interpretation condition to **true**. Many other heuristics are possible; we leave an exploration of these to future work.

Algorithm 1 Procedure VERIFYABS $(\mathcal{D}, \mathcal{FS}_{\mathcal{A}})$: Top-level CAL verification procedure.

```
1:  // Input: Combined word-level design (miter)
    D := ⟨I, O, {N_i | i = 1, ..., N}⟩
2:  // Input: Equivalence classes of fblocks
    FS_A := {F_j | j = 1, ..., k}
3:  // Output: Verification result
    Result ∈ {Valid, CounterExample}
4:  Set c_i = false for all c_i ∈ IC.
5:  while true do
6:    T = CONDABS (D, FS_A, IC)
7:    Result = VERIFY (T)
8:    if Result = Valid then
9:      Return Valid.
10:   else
11:     Store counterexample in CE.
12:     if CE is spurious then
13:       IC ← LEARNABSCONDS (D, FS_A, CE)
14:     else
15:       Return CounterExample.
16:     end if
17:   end if
18: end while
```

Procedure CONDABS $(\mathcal{D}, \mathcal{FS}_{\mathcal{A}}, \mathcal{IC})$ is responsible for creating a term-level design $\mathcal{T}$ from the original word-level design $\mathcal{D}$, the set of equivalence classes to be abstracted $\mathcal{FS}_{\mathcal{A}}$, and the set of interpretation conditions $\mathcal{IC}$. CONDABS operates by iterating through the equivalence classes in $\mathcal{FS}_{\mathcal{A}}$. A fresh uninterpreted function symbol UF_j is created for each tuple of isomorphic output signals χ_j associated with equivalence class $\mathcal{F}_i \in \mathcal{FS}_{\mathcal{A}}$. Each output signal $v_{ij} \in \chi_j$ is conditionally abstracted with UF_j as illustrated in Fig. 3. More formally, if $\mathsf{c}_{v_{ij}} \in \mathcal{IC}$ denotes the interpretation condition associated with v_{ij}, then we replace the assignment $v_{ij} \leftarrow e$ in fblock f_i with the assignment $v_{ij} \leftarrow$

ITE($\mathsf{c}_{v_{ij}}, e, UF_j(i_1, ..., i_k)$), where *ITE* denotes the if-then-else operator. See [8] for a more detailed description.

C. Learning Conditional Abstractions

Spurious counterexamples arise due to imprecision introduced during abstraction. More specifically, when a spurious counterexample arises, it means that at least one fblock $f_i \in \mathcal{F}$ (where $\mathcal{F} \in \mathcal{FS}_{\mathcal{A}}$) is being abstracted when it needs to be modeled precisely. In the context of our abstraction procedure VERIFYABS, if VERIFY $(\mathcal{T})$ returns a spurious counterexample $\mathcal{CE}$, then we must invoke the procedure LEARNABSCONDS $(\mathcal{D}, \mathcal{FS}_{\mathcal{A}}, \mathcal{CE})$.

The LEARNABSCONDS procedure invokes a *decision tree learning* algorithm on traces generated by replacing fblocks $f_i \in \mathcal{F}$ by a tuple of random functions RF_{χ_j}. Traces are classified as being "bad" or "good" depending on whether the replacement with a random function results in a property violation or not. The learning algorithm generates a classifier in the form of a decision tree to separate the good traces from the bad ones. The classifier is essentially a Boolean function over signals in the original word-level design. More information about decision tree learning can be found in Mitchell's textbook [22].

There are three main steps in the LEARNABSCONDS procedure:

1. Generate good and bad traces for the learning procedure;
2. Determine meaningful features that will help decision tree learning procedures compute high quality decision trees, and
3. Invoke a decision tree learning algorithm with the above features and traces.

The data input to the decision tree software is a set of tuples where one of the tuple elements is the target attribute and the remaining elements are features. In our context, a target attribute α is either Good or Bad. Our goal is to select features such that we can classify the set of all tuples where $\alpha = \mathsf{Bad}$ based on the rules provided by the decision tree learner. Since we use an off-the-shelf decision tree learning tool, we omit a description of how this works. However, it is very important to provide the decision tree learning with quality input data and features, otherwise, the rules generated will not be of use. The data generation procedure is described in Sec. IV-D and feature selection is described in Sec. IV-E.

D. Generating Data

In order to produce a meaningful decision tree, we must provide the decision tree learner with both good and bad traces. We use random simulation to generate *witnesses* and *counterexamples* and describe these procedures in detail below.

1) Generating Witnesses: Good traces, or *witnesses*, are generated using a modified version of the random simulation procedure described in Sec. IV-A. Instead of simulating the abstract design when only a single fblock has been replaced with a random function, we replace all fblocks with their respective random functions *at the same time* and perform verification via simulation. Replacing *all* the fblocks to be abstracted with the respective random function ensures diversity in the set of traces fed to the decision tree learner.

After replacing each fblock to be abstracted with the corresponding random functions, we perform simulation by verification, repeating the process for N different random functions for each fblock. N is chosen heuristically similar to T in Sec. IV-A (we discuss typical values for N in Sec. V-D). The initial state of design $\mathcal{D}$ is set randomly before each run of simulation. This usually results in simulation runs that pass, and hence in good traces — recall that at this stage we only consider fblocks that produce failing runs in a small fraction of simulation runs. Now, instead of only logging the result of the simulation, we log the value of every signal in the design for every cycle of each passing simulation. It is up to the feature selection step, described in Sec. IV-E, to decide what signals are important.

2) Generating Similar Counterexamples: Whenever LEARNABSCONDS is called, there is a spurious counterexample stored in $\mathcal{CE}$. We generate many counterexamples similar to $\mathcal{CE}$ using random simulation in a manner similar to that used while identifying abstraction candidates. If more than one equivalence class of fblocks has been abstracted, the counterexample $\mathcal{CE}$ can be the result of abstracting any individual equivalence class, or a combination of them.

Consider the situation where $\mathcal{CE}$ is the result of only abstracting a single equivalence class. In this situation, we replace each fblock in that class with a random function in the word-level design, just as we did when identifying abstraction candidates in Sec. IV-A. Next, verification via simulation is performed, and this process is iterated for N different random functions, for heuristically-chosen N. A main point of difference between generating similar counterexamples and generating witnesses is that in generating similar counterexamples, we set the initial state of design $\mathcal{D}$ to be *consistent with the initial state in* $\mathcal{CE}$, whereas we randomly set the initial state of design $\mathcal{D}$ when generating witnesses. We log the values of every signal in the design for each failing simulation run. It is possible that none of the simulation runs fail, because the counterexample could be the result of abstracting a different equivalence class. We repeat this process for each fblock that is being abstracted.

If replacing individual equivalence classes with random functions does not result in any failing simulation run, we must take into account combinations of equivalence classes. In this case, we try pairs of equivalence classes, then triples, and so on. Clearly, there is a potential exponential blowup here; however, this has not occurred in our experiments. In fact, considering a single equivalence class at a time sufficed for all examples considered in this work. We leave the exploration of heuristics that determine how to choose interpretation conditions for combinations of fblocks for future work.

As noted above, the witness generation and the counterexample generation procedures can both generate good and bad traces. Denote the set of all bad traces by Bad and the good traces as $Good$. We label each trace in Bad with the Bad attribute and each trace in $Good$ with the Good attribute.

E. Choosing Features

The quality of the decision tree generated is highly dependent on the features used to generate the decision tree. We use two heuristics to identify features:

1. Include input signals to the fblock being abstracted.
2. Include signals encoding the "unit-of-work" being processed by the design, such as the instruction being executed.

Input signals. Suppose we wish to determine when fblock f must be interpreted. It is very likely that whether or not f must be interpreted is dependent on the inputs to f. So, if f has input signals $(i_1, i_2, ..., i_n)$ it is almost always the case that we would include the input arguments as features to the decision tree learner.

Unit-of-work signals. There are cases when the input arguments alone are not enough to generate a quality decision tree. In these cases, human insight can be provided by defining the unit-of-work being performed by the design. For example, in a microprocessor design, a unit-of-work is an instruction. Similarly, in a network-on-a-chip (NoC), the unit-of-work is a packet, where the relevant signals could include the source address, destination address, or possibly the type of packet being sent across the network. Once a unit-of-work signal is identified at one part of the design, automatic dataflow analysis can identify all signals derived from it and label these also as features for the learning algorithm. For instance, in the case of a pipelined processor, the registers storing instructions in *each stage* of the pipeline are relevant signals to treat as features.

In rare cases, the above heuristics are not enough to generate quality decision trees; we discuss these scenarios and give additional features in Sec. V.

V. Case Studies

We performed two case studies to evaluate CAL. Both of these case studies have also been verified using ATLAS. Additionally, each case study requires a non-trivial interpretation condition (i.e., an interpretation condition different from **false**). The first case study involves verifying the example shown in Fig. 4. In the second case study, we verify, via correspondence checking, two versions of the Y86 microprocessor.

All experiments were run on a MacBook Pro with a 2.4 GHz Intel Core 2 Duo processor with 4GB RAM. The term-level verification engine used for the experiments was the UCLID verification system [1], [9] with Minisat2 [16] and Boolector [11] as the SAT and SMT backends, respectively. Random simulation was performed using Icarus Verilog [25]. The decision tree learner we used in the experiments is C5.0 [23]. We compared our term-level abstraction-based approach with the state-of-the-art bit-level equivalence checker, ABC [6], [10]. The benchmarks used in these experiments as well as the results can be found at [24].

A. The Illustrative Example

In this experiment, we perform equivalence checking between Design A and B shown in Fig. 4. First, we initialize the designs to the same initial state and inject an arbitrary instruction. Then we check whether the designs are in the same state. The precise property that we wish to prove is that the ALU and PC outputs are the same for design A and B. Let $\texttt{out}_\texttt{A}$ and $\texttt{out}_\texttt{B}$ denote the ALU outputs and $\texttt{pc}_\texttt{A}$ and $\texttt{pc}_\texttt{B}$ denote the PC outputs for designs A and B, respectively. The property we prove is: $\texttt{out}_\texttt{A} = \texttt{out}_\texttt{B} \wedge \texttt{pc}_\texttt{A} = \texttt{pc}_\texttt{B}$. Aside from the top-level modules, the design consists of only two modules, the instruction memory (IMEM) and the ALU. We do not consider the instruction memory for abstraction because we do not address automatic memory abstraction. The ALU passes the random simulation stage, so it is an abstraction candidate.

The features we use in this case are arguments to the ALU; the instruction and the data arguments. The interpretation condition learned from the trace data is $\texttt{op} = \mathsf{JMP}$ where op is the top 4 bits of the instruction. As shown in Table I, the runtime for CAL is comparable with that of ABC.

Interpretation Condition	Runtime (sec)		
		UCLID	
	ABC	SAT	SMT
true	0.02	28.51	27.01
$\texttt{op} = \mathsf{JMP}$	—	**0.31**	**0.01**

TABLE I
Performance comparison Runtime comparison between ABC and UCLID for the processor fragment shown in 4. The runtime associated with the model abstracted with CAL is shown in **bold**.

B. The Y86 Processor

In this experiment, we verify two versions of the well-known Y86 processor model introduced by Bryant and O'Hallaron [13]. The Y86 processor is a pipelined CISC microprocessor styled after the Intel IA32 instruction set. While the Y86 is relatively small for a processor, it contains several realistic features, such as a dual read, dual write register file, separate data and instruction memories, branch prediction, hazard resolution, and an ALU that supports bit-vector arithmetic and logical instructions. Note that we have extended the ALU to include multiplication in order to create a harder verification problem. Of the several variants of the Y86 processor we focus on two that have different versions of branch prediction logic: NT and BTFNT. These versions are the only versions where the ALU cannot be fully abstracted (i.e., partial abstraction is required). In NT branches are predicted as not taken, whereas in BTFNT branches backwards in the address space are predicted as taken, while branches forward in the address space are predicted as not taken. NT and BTFNT were the designs that the ATLAS approach had the most difficulty abstracting [8]. The property we wish to prove on the Y86 variants is Burch-Dill style correspondence-checking [14].

Both NT and BTFNT versions have the same module hierarchy and differ only in the logic pertaining to branch prediction. The following modules are candidates for abstraction: register file (RF), condition code (CC), branch function (BCH), arithmetic-logic unit (ALU), instruction memory (IMEM), and data memory (DMEM). The RF module is ruled out as a candidate for abstraction during the random simulation stage due to a large number of failures during verification via simulation. This occurs because an uninterpreted function is unable to accurately model a mutable memory. We do not consider IMEM and DMEM for automatic abstraction because they are memories and we do not address automatic memory abstraction in this work. Instead, we manually model IMEM and DMEM with completely uninterpreted functions. The CC and BCH modules are also removed from consideration due to the relatively simple logic contained within them. Abstracting these modules is unlikely to yield substantial

verification gains and may even hurt performance due to the overhead associated with uninterpreted functions. This leaves us with the ALU module.

1) Decision tree feature selection: In the case of both BTFNT and NT using only the arguments of the abstracted ALU is not sufficient to generate a useful decision tree. The ALU takes three arguments, the op-code op and two data arguments a and b. Closer inspection of the data provided to the decision tree learner reveals a problem. In almost every cycle of both good and bad traces, the ALU op is equal to ALUADD and the b argument is equal to 0.

In this situation, the arguments to the ALU are not good features by themselves (i.e., there is not enough diversity within the traces to learn a useful classifier). Conceptually, the unit-of-work that we are performing in a pipelined processor is a sequence of instructions, specifically the instructions that are currently in the pipeline. The most relevant instruction is the instruction currently in the execute stage (i.e., the stage containing the ALU). When $Instr_E$, op, a, and b are used as features, the resulting decision tree yields the interpretation condition: $\mathsf{c}_{E,b} := Instr_E = \mathsf{JXX} \wedge b = 0$.

The main reason we need a partial abstraction is so that the target address of a jump instruction can pass through the ALU unaltered. Thus, this is the best interpretation condition we can hope for. In fact, in previous attempts to manually abstract the ALU in the BTFNT version, we used: $\mathsf{c}_{Hand} := op = \mathsf{ALUADD} \wedge b = 0$.

When we compare the runtimes for verification of the Y86-BTFNT processor, we see that verifying BTFNT with the interpretation condition $\mathsf{c}_{E,b}$ outperforms the unabstracted version and the previously best known abstraction condition (c_{Hand}). Table II compares the UCLID runtimes for the Y86 BTFNT model with the different versions of the abstracted ALU.

Interpretation Condition	Runtime (sec)		
	ABC	UCLID	
		SAT	SMT
true	> 1200	> 1200	> 1200
c_{Hand}	—	133.03	105.34
$\mathsf{c}_{E,b}$	—	**101.10**	**65.52**

TABLE II
Performance comparison Runtime comparison between ABC and UCLID for Y86-BTFNT for different interpretation conditions. The runtime associated with the model abstracted with CAL is shown in **bold**.

2) Abstraction-refinement: The NT version of the Y86 processor requires an additional level of abstraction refinement. In general, requiring multiple iterations of abstraction refinement is not interesting by itself. However, it is interesting to see how the interpretation conditions change using this machine learning-based approach.

Attempting unconditional abstraction of the ALU in the NT version results in a spurious counterexample. The interpretation condition learned from the traces generated in this step is $\mathsf{c} := a = 0$. It is interesting that the same interpretation condition is generated regardless of whether we consider all of the instructions as features, or only the instruction in the same stage as the ALU. Not surprisingly, the second attempt at verification using the interpretation condition c results in another spurious counterexample. In this case, the interpretation condition learned is $\mathsf{c}_E := Instr_E = \mathsf{ALUADD}$, which states that we must interpret anytime an addition operation is present in the ALU. Similarly with the first iteration, the interpretation condition learned is the same regardless of whether we use all of the instructions as features, or only the instruction in the execute stage. Verification is successful when c_E is used as the interpretation condition.

A performance comparison for the NT variant of the Y86 processor is shown in Table III. Unlike the BTFNT case, the abstraction condition we learn for the NT model is not quite as precise as the previously best known interpretation condition, and the performance isn't as good. However, the runtimes for conditional abstraction, including the time spent in abstraction-refinement, are smaller than that of verifying the original word-level circuit. That is, the runtime when the interpretation condition is c_E is accounting for two runs of UCLID that produce a counterexample and an additional run when the property is proven Valid. Note that the most precise abstraction condition is the same for both BTFNT and NT. The best performance on the NT version is obtained when the interpretation condition $\mathsf{c}_{BTFNT} := Instr_E = \mathsf{JXX} \wedge b = 0$ is used.

Condition	Runtime (sec)		
	ABC	UCLID	
		SAT	SMT
true	> 1200	> 1200	> 1200
c_{Hand}	—	154.95	89.02
c_E	—	**191.34**	**187.64**
c_{BTFNT}	—	94.00	52.76

TABLE III
Performance comparison Runtime comparison between ABC and UCLID for Y86-BTFNT for different interpretation conditions. The runtime associated with the model abstracted with CAL is shown in **bold**.

The reason the interpretation condition for BTFNT differs from that of NT is because the root cause of the counterexamples are different. The counterexample generated for the BTFNT model arises because the branch target that would pass through the ALU unaltered, gets mangled when the ALU is abstracted. The counterexample generated for the NT model arises because the abstracted ALU incorrectly squashes a properly predicted branch.

C. Comparison with ATLAS

ATLAS and CAL compute the same interpretation conditions for the processor fragment described in Sec. V-A. Thus, the only interesting comparison with regard to the interpretation conditions is for the Y86 design.

ATLAS is able to verify both BTFNT and NT Y86 versions with one caveat—the multiplication operator was removed from the ALU to create a more tractable verification problem. When multiplication is present inside the ALU, the ATLAS approach can not verify BTFNT or NT in under 1200 seconds. In the case where the multiplication operator is removed, the interpretation conditions generated by ATLAS simplify to **true**. In this case, ATLAS actually takes longer to verify BTFNT, with the abstracted version taking 1390 seconds and the word-level version taking only 1077 seconds. This behavior highlights the main drawback of ATLAS. The static analysis procedure blindly takes into account the structure of the design, giving equal importance to every

signal. This was the inspiration behind using machine learning to compute interpretation conditions. Not only is CAL able to verify the BTFNT and NT Y86 versions when multiplication *is* included in the ALU, but it does so with an order of magnitude speedup over the unabstracted version.

D. Remarks

The runtimes listed in Tables I, II, and III focus only on the time taken by ABC and UCLID. The remaining runtime taken by the other components of the CAL procedure is, in comparison, negligible. First, the runtime of the decision tree learner is less than 0.1 seconds in every case. Second, the simulation time is quite small. For instance, simulating 1000 correspondence checking runs for the Y86 model takes less than 5 seconds. However, we are unable to verify the original word-level Y86 designs within 1200 seconds, so the CAL runtime is negligible. Similarly, the runtime of generating an AIG for input to ABC is less than 1 second.

The number of good and bad traces required to produce a quality decision tree for the processor fragment example in Sec. V-A is 5 (10 total). For the Y86 examples, the number of good and bad traces was 50 (100 total). Thus, in every example, it takes only a fraction of a second to generate enough data for the machine learning algorithm to be able to produce useful results.

VI. Conclusion

In this paper, we present CAL, an automatic abstraction procedure based on a combination of random simulation and machine learning. We evaluate the effectiveness and efficiency of our approach on equivalence and refinement checking problems in the context of pipelined processors. We have shown that we are able to automatically learn conditional abstractions that lead to better verification performance. Additionally, we learned abstraction conditions that were better than the previously best known abstraction conditions for two variants of the Y86 microprocessor design.

Acknowledgements. This research was supported in part by SRC contract 2045.001 and 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] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/~alanmi/abc.

[7] 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.

[8] B. A. Brady, R. E. Bryant, S. A. Seshia, and J. W. O'Leary. ATLAS: automatic term-level abstraction of RTL designs. In *Proceedings of the Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, July 2010.

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

[10] R. K. Brayton and A. Mishchenko. ABC: An academic industrial-strength verification tool. In *CAV*, pages 24–40, 2010.

[11] R. D. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *In* Proc. of TACAS, pages 174–177, March 2009.

[12] 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.

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

[14] 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.

[15] E. M. Clarke, A. Gupta, J. H. Kukula, and O. Strichman. SAT based abstraction-refinement using ilp and machine learning techniques. In *Proc. Computer-Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 265–279, 2002.

[16] N. Eén and N. Sörensson. The MiniSAT Page. http://minisat.se.

[17] A. Gupta and E. M. Clarke. Reconsidering CEGAR: Learning good abstractions without refinement. In *Proc. International Conference on Computer Design (ICCD)*, pages 591–598, 2005.

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

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

[20] 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.

[21] 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.

[22] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[23] R. Quinlan. Rulequest research. http://www.rulequest.com.

[24] The CAL Approach. CAL: Conditional Abstraction through Learning. http://uclid.eecs.berkeley.edu/cal.

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