Automatic Abstraction and Verification of Verilog Models

Zaher S. Andraus and Karem A. Sakallah

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122

{zandrawi,karem}@eecs.umich.edu

ABSTRACT

Abstraction plays a critical role in verifying complex systems. A number of languages have been proposed to model hardware systems by, primarily, abstracting away their wide datapaths while keeping the low-level details of their control logic. This leads to a significant reduction in the size of the state space and makes it possible to verify intricate control interactions formally. These languages, however, require that the abstraction be done manually, a tedious and error-prone process. In this paper we describe Vapor, a tool that automatically abstracts behavioral RTL Verilog to the CLU language used by the UCLID system. Vapor performs a sound abstraction with emphasis on minimizing false errors. Our method is fast, systematic, and complements UCLID by serving as a back-end for dealing with UCLID counterexamples. Preliminary results show the feasibility of automatic abstraction and its utility in formal verification.

Categories and Subject Descriptors

B.6.3 [Hardware]: Logic Design - *Verification*

General Terms

Verification

Keywords

Register Transfer Level (RTL), Verilog, Abstraction, Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions (CLU), UCLID.

1 INTRODUCTION

The state explosion problem is still a major hurdle in verifying systems of today's scale and complexity. Researchers nowadays try to increase tools' scalability by integrating *abstraction* paradigms in various layers of verification systems. In addition to well established research studies for abstraction (e.g. [6]), it has been successfully used in practical systems, such as the Microsoft SLAM project [1] and the Synopsys RFN Tool [14], and research in this domain is still actively on-going. In particular, *datapath* abstraction [7, 9] was found to be a scalable approach for verifying hardware units, where abstracting the data path is relatively straightforward. Consequently, the process of verifying a microprocessor's implementation against its specification has gained

practicality. The UCLID tool [9] allows such an approach, whereby designers make assumptions on data path units, abstract them to uninterpreted entities, and prove properties on the rest of the circuit, mainly the control part.

Although UCLID is a completely automatic tool, designers have to manually abstract the design and express it in the UCLID language. We make the pragmatic assumption that designers would be unwilling to manually abstract the design for verification purposes, since this necessitates laborious analysis of the RTL, as well as incremental updates to the UCLID model once the RTL is updated, leading to a cumbersome verification iteration. In addition, this might introduce 'modeling' bugs that are due to human errors when modeling for UCLID, while hiding 'real' bugs in the original RTL.

We endeavor to perform such an abstraction automatically from design descriptions that are more familiar to designers such as a micro-architecture description in an HDL like Verilog [12]. Vapor (which stands for Verilog Abstraction for Processor Verification), performs a sound abstraction to UCLID, while minimizing the effect of false negatives that are inherent in any abstraction process.

The work of Hojati and Brayton [7] is the most relevant to our context. In this work, RTL Verilog is translated to an ICS (Integer Combinational Sequential Concurrency) model, which describes hardware systems in a high level of abstraction using integers, interpreted and uninterpreted functions. Unlike UCLID, the use of uninterpreted functions is limited to arithmetic manipulation, excludes bit-vector manipulation, and is not particularly tailored to datapath abstraction.

Our choice of UCLID as the abstraction target is due to UCLID's automatic and efficient decision procedure [4], which grants it superiority over theorem provers, and other decision procedures such as SVC [2]. The rest of this paper is organized in five sections. Section 2 provides the necessary UCLID and Verilog notions needed in the rest of the paper. The abstraction procedure is detailed in Section 3. False negatives and their processing is described in Section 4. The implementation of Vapor and its empirical evaluation are discussed in Section 5 and the paper concludes in Section 6.

2 PRELIMINARIES

2.1 UCLID Basics

The logic of equality with uninterpreted functions (LEUF) [5] enables the construction of abstract hardware models that are suitable for formal verification. The CLU logic [3] extends LEUF with counter arithmetic and Lambda expressions and forms the basis of the UCLID verification system

[9]. UCLID accepts a CLU model along with a safety property and generates a corresponding propositional formula that is unsatisfiable if and only if the property holds. Systems such as UCLID have been shown to provide acceptable expressiveness to model and verify safety properties of modern out-of-order advanced microprocessors [9, 10].

CLU supports two basic data types, TRUTH and TERM. It also supports two function types: FUNC which maps a list of TERMs to a TERM, and PRED which maps a list of TERMs to TRUTH. These types are combined using operators from the following set:

- Boolean connectives for TRUTH constants and variables.

- Equality $(=)$ and ordering $(<, >)$ relations which operate on TERMs and return TRUTH.

- Interpreted functions **succ** and **pred** which take a TERM and return, respectively, its successor and predecessor. These functions allow modeling counters and represent a limited form of integer arithmetic.

- The ITE (if-then-else) operator which selects between two TERMs based on a Boolean condition.

- Uninterpreted PRED symbols or Lambda expressions that take TERM arguments and return a TRUTH value.

- Uninterpreted FUNC symbols or Lambda expressions that take TERM arguments and return a TERM.

Modeling hardware systems using TERMs, uninterpreted functions and predicates (UFs and UPs) is the basic mechanism for abstracting unimportant details during verification. Correctness is assured in the sense that if the abstract model satisfies a given property, then so does the original concrete model. Violation of the property by the abstract model, however, may not imply its violation by the concrete model but rather that the abstract model is too coarse. The occurrence of such false negatives is inherent in the abstraction process and cannot be completely eliminated. The abstraction process, thus, becomes a trade-off between hiding as much detail as possible from the concrete model (for scalable verification) while insuring that the incidence of false negatives is sufficiently small (for meaningful verification.)

A basic mechanism for reducing the occurrence of false negatives in logics that use uninterpreted functions is to insure *functional consistency*. Symbolically, an *n*-argument UF or UP g must satisfy

$$(x_1 = y_1) \wedge \dots \wedge (x_n = y_n) \rightarrow g(x_1, \dots, x_n) = g(y_1, \dots, y_n)$$

Such constraints are automatically accounted for by UCLID during the process of converting the CLU formula to propositional form. Readers are referred to [11] for a more detailed discussion of UCLID and its usage scenarios.

2.2 Verilog Basics

Unlike CLU, the Verilog language [12] lacks formal semantics. Verilog evolved as a simulation and, later, synthesis language. Thus, its use in formal verification is, at best, problematic and our abstraction methodology must be pragmatic in the sense of "doing what's intended" without making any formal claims. Furthermore, we believe that the benefits of driving formal verification tools directly from a Verilog description outweigh Verilog's shortcomings as a formal language.

We assume that the hardware description being abstracted is written in the so-called synthesizable subset of Verilog. For abstraction purposes, it is sufficient to view a design description in Verilog in terms of its data types and operators. The basic data type in Verilog is the bit. Verilog also offers two composite data types, bit vectors and memories, that can be viewed, respectively, as one- and two-dimensional arrays of bits. Bit vectors and memories can also be referred to as "words" and "word arrays." Bit vectors allow read and write access to the entire vector (viewed as an atomic object) as well as to individual bits or contiguous bit fields. Furthermore, access to parts of a bit vector can be explicit (by specifying a range of bits) or implicit (using the concatenation operator.) In contrast, memories cannot be accessed atomically; only single words can be read from or written to a memory. Finally, bits and bit vectors can be declared as either wires or registers to model, respectively, combinational or sequential behavior; memories can only be declared as registers. Verilog fragments showing example declarations of bits and bit vectors, as well as implicit and explicit access to bit fields, are shown in Figure 1(a, b, c).

3 VERILOG-TO-UCLID ABSTRACTION

As a first-order approximation, the abstraction of a Verilog description to UCLID can be thought of as a syntactic mapping between related variable types in the two languages. For instance, single- and multi-bit signals in Verilog can be mapped, respectively, to TRUTH and TERM variables in UCLID. These mappings, in turn, induce corresponding mappings between Verilog operators and UCLID logical connectives, UFs, and UPs. Such an approach basically assumes that multi-bit signals and the function units that operate on them should be automatically abstracted. This, however, may not be the case, and may lead to the unintended abstraction of critical control signals that are grouped in Verilog as multi-bit vectors, making the abstract UCLID model too coarse to be usable in verification. In addition, multi-bit signals typically consist of bit fields that are individually accessed for reading and/or writing. Correct abstraction in such cases must account for the relation among the bit fields and between each bit field and its parent vector. Finally, abstraction of certain Verilog operators may lead to the generation of spurious errors since functional abstraction guarantees consistency under equality but is oblivious to properties such as associativity and commutativity; for example abstracting integer addition with the UF $\text{add}(x, y)$ will insure functional consistency but will not treat $\text{add}(x, y)$ as identical to $\text{add}(y, x)$ as required by commutativity of addition.

The above observations suggest that an abstraction algorithm must not only examine the declared signal types in Verilog but also the way such signals are "used" in the body of the Verilog description. In addition, the abstraction process must be complemented with a mechanism that detects false errors when they arise. In the rest of this section, we describe how our tool, Vapor, abstracts various Verilog constructs to corresponding ones in UCLID. The treatment of false errors is described in Section 4.

```
reg [16:0] word;   // 17-bit register
wire [7:0] w_low;  // 8-bit bus
wire [7:0] w_high; // 8-bit bus
wire [16:0] out;   // 17-bit bus
wire parity;       // single-bit wire
wire clk;          // clock
reg mode;          // single flip-flop
```

(a) Signal declarations in Verilog

```
always @(posedge clk)
    if (mode == 1'b1)
        word[10:3] <= 8'b11001110;
    else
        word<={parity,{w_high,~w_low}};
assign out = word;
```
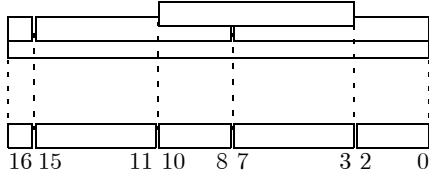
(b) Verilog fragment showing explicit as well as
implicit access to bit fields of 'word'

```
always @(posedge clk)
    if (mode == 1'b1)
        word[10:3] <= 8'b11001110;
    else begin
        word[16] <= parity;
        word[15:8] <= w_high;
        word[7:0] <= ~w_low;
    end;
assign out = word;
```

(c) Equivalent Verilog fragment where all implicit
accesses to bit fields of 'word' are made explicit

(d) Partition induced by the bit fields of 'word'

```
word_10_3 = concat_3_5(word_P_10_8,word_P_7_3)
word_P_10_8 = extract_7_3(word_10_3)
word_P_7_3 = extract_4_5(word_10_3)
```

(e) Uninterpreted functions that act as axioms relating bit field
word[10:3] to its corresponding blocks in the partition

```
1   CONST
2       INITS : TERM;
3       concat_5_3 : FUNC[2];
4       extract_7_3 : FUNC[1];(* [7:5] *)
5       extract_4_5 : FUNC[1];(* [4:0] *)
6       bitw_not_8 : FUNC[1];
7       . . .
8   VAR
9       mode_0_0 : TRUTH;        (* mode *)
10      word_16_0 : TERM;        (* word[16:0] *)
11      word_16_16 : TRUTH;      (* word[16] *)
12      word_10_3 : TERM;        (* word[10:3] *)
13      word_7_0 : TERM;         (* word[7:0] *)
14      w_low_7_0 : TERM;        (* w_low[7:0] *)
15      word_P_2_0 : TERM;       (* word_P[2:0] *)
16      word_P_7_3_n : TERM;     (* word_P_n[7:3] *)
17      word_P_10_8_n : TERM;    (* word_P_n[10:8] *)
18      const53 : TERM;          (* 8'b11001110 *)
19      . . .
20  DEFINE
21      word_P_7_3_n := case
22          mode_0_0: extract_4_5(const53);
23          default: ...
24      esac;
25      . . .
26  ASSIGN
27      init[word_7_0] := INITS;  (* init val *)
28      next[word_10_3] := case
29          mode_0_0 : const53;
30          default: . . .;
31      esac;
32      next[word_7_0] := case
33          mode_0_0: concat_5_3(word_P_7_3_n ,
34                              word_P_2_0);
35          default: bitw_not_8(w_low_7_0);
36      esac;
37      . . .
```

(f) UCLID fragment corresponding to the update of
bit field word[7:0]
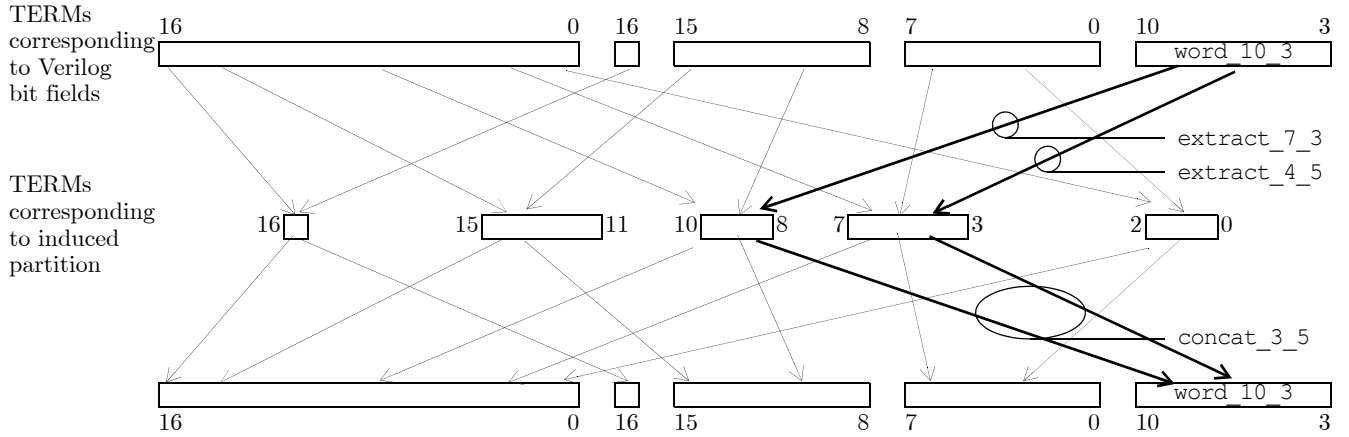
(g) Uninterpreted extraction and concatenation functions needed to insure consistency between 'word' and its bit fields.
The highlighted arrows show the relation between 'word_10_3' and its corresponding blocks in the partition of 'word'.

Figure 1. Abstraction Example

Table 1. Basic Abstraction of Verilog Variables

Variable Type	Verilog	UCLID
Single-bit	wire parity;	parity : TRUTH;
1-D bit vector	reg [31:0] PC;	PC : TERM;
2-D bit vector	reg [63:0] RF [31:0];	RF : FUNC[1];

3.1 Abstraction of Verilog Variables

Table 1 depicts the basic template for abstracting Verilog variables to corresponding UCLID variables. Based on their "bit structure" Verilog variables are classified into three main types.

- Single-bit variables which are 2-valued and naturally modeled as UCLID TRUTH variables.
- Multi-bit words which are viewed as unsigned integers and translated into corresponding UCLID TERM variables.
- Word arrays which typically denote memories or register files and are conveniently represented by UCLID UF variables.

Except for the abstraction of bit vectors, these mappings are straightforward. Bit vectors require additional machinery to insure that their abstraction is consistent. Specifically, given a Verilog bit vector X, we must not only create a UCLID TERM to represent X but also create additional TERMs to represent each of its individually-accessed bit fields. Furthermore, we must introduce a set of uninterpreted functions that relate these TERMs to each other. Otherwise, UCLID treats these TERMs as completely independent, potentially leading to the generation of numerous false errors.

Without loss of generality, assume that X is a vector of n bits such that $X[n-1]$ is the most significant bit. It is convenient to view X as the interval $[n-1:0]$. Assume further that the set of individually-accessed bit fields of X is denoted by X^F. Thus, X^F is a set of possibly overlapping subintervals of $[n-1:0]$. Finally, let $\pi(X^F)$ denote the coarsest partition of $[n-1:0]$ induced by X^F. For example, if X is $[15:0]$, and $X^F = \{[15:0], [15:8], [7:0], [10:3]\}$, then $\pi(X^F) = \{[15:11], [10:8], [7:3], [2:0]\}$.

Consistency can now be established by introducing TERMs for each of the bit fields in X^F and $\pi(X^F)$ and a corresponding set of complementary uninterpreted *extraction* and *concatenation* functions that relate these TERMs. These functions are designed to insure that whenever a bit field in X^F is changed, appropriate updates are made to all the other bit fields that overlap it. While such functions can be given arbitrary names (subject to giving different functions distinct names), for documentation purposes and to facilitate debugging, they are given names that indicate their intended purpose. Thus, extraction functions are named $extract\_m\_w(X)$ to indicate the extraction of w bits from bit vector X starting at bit position m[1]. Similarly, concatenation functions are named $concat\_w_1\_\ldots\_w_k(X_1, \ldots, X_k)$ to indicate the concatenation of k bit vectors $X_1, \ldots, X_k$ whose bit widths are $w_1, \ldots, w_k$. A similar naming conven-

tion is adopted for TERM and TRUTH variables; e.g., the Verilog bit vector $X[a{:}b]$ is declared as the TERM X_a_b.

These notions are illustrated in Figure 1 which depicts (in part c) a Verilog fragment and (in parts e and f) the corresponding UCLID abstraction. Consider, in particular, how the bit vector word[7:0] gets updated. From the Verilog fragment, it is clear that portions of word[7:0] are assigned to in both branches of the if statement. Specifically, when mode is equal to 1, the five most significant bits of word[7:0] (i.e. word[7:3]) may change because of the assignment to word[10:3]. And when mode is equal to 0, word[7:0] is assigned the value of ~w_low. These updates are facilitated by introducing the following UCLID TERMs and associated uninterpreted functions:

- mode_0_0, word_10_3, and word_7_0 to denote the Verilog variable mode, and the individually-accessed bit fields word[10:3] and word[7:0]
- word_P_2_0 and word_P_7_3 to denote the bit fields of word in the induced partition; word_P_7_3_n is a temporary TERM that denotes the next value of word_P_7_3
- the UF extract_4_5( ) which relates word_7_3 to word_10_3; word_7_3 is derived from word_10_3 by extracting 5 bits starting from the fourth most significant bit position
- the UF concat_5_3( ) which reconstructs word_7_0 from word_P_7_3_n and word_P_2_0
- the UF bitw_not_8( ) which represents bitwise negation applied on w_low_7_0.

The update of word[7:0] is now achieved as follows:

1. word[7:0] is initialized to some arbitrary symbolic constant (line 27)
2. when mode is equal to 1, word[10:3] is assigned an uninterpreted constant value (lines 28 and 29)
3. the next value of word[7:0] is set to bitw_not_8(w_low) if mode is equal to 0 (line 35) or, if mode is equal to 1, to the concatenation of the new value of its 5 most significant bits and the old value of its 3 least significant bits (lines 33 and 34).

The general scheme described above can be simplified in certain situations and such simplifications can lead to significantly more efficient translations from Verilog to UCLID. For example, if the individually-accessed bit fields of a Verilog bit vector are mutually disjoint, it is not necessary to introduce additional TERMs for the partition blocks. Extraction may also be simplified when applied on constants. These optimizations reduce the size of the propositional formula generated by UCLID since UCLID encodes TERMs using a bit string whose length is a function of the total number of TERMs and UFs applications being processed. Furthermore, we found that such an optimization eliminates many unnecessary false errors by avoiding the need for using extraction UFs.

In the process of obtaining the coarsest refinement over a set of bit vectors, some of the blocks in the resulting partition may end up being single bits. These single-bit fields can be modeled as TERMs and used in extraction and concatenation as described above. This, however, might allow them to get more than 2 different symbolic values. In such cases, we use UPs, instead of UFs, as extraction functions. When

---

[1] Without loss of generality, bit vectors are assumed to be numbered such that bit 0 is in the least significant position.

the block (TRUTH variable) needs to be concatenated, it has to be "type cast" to TERM, using an appropriate ITE expression.

3.2 Abstraction of Verilog Constants

Constants in Verilog are treated as unsigned integers. Typically, small constants are used in arithmetic expressions such as "PC <= PC + 32'd4". Large constants, on the other hand, are frequently employed for bit masking as in "var <= var ^ 8'b01101010". Vapor distinguishes between small and large Verilog constants based on a user-specified threshold. It then abstracts large constants to CONST TERMs in UCLID. Such an abstraction disregards the numerical value of these constants and treats them as a collection of independent unordered integers. Small constants are not abstracted. Rather, they are modeled as UCLID variables using the interpreted functions **succ** and **pred**. For example, 32'd4 is declared in UCLID's VAR and DEFINE sections as "const4:TERM", and "const4:=succ^4(const0)", respectively, where const0 is declared as a CONST TERM representing the integer 0. Treating small constants in this fashion guarantees that their ordering is preserved.

3.3 Abstraction of Verilog Operators

In our translation, the variable types in UCLID induce operator abstractions: TRUTH variables are manipulated via Boolean connectives; TERM variables via UFs. Equality for TERMs is implemented using '=' in UCLID, while for TRUTH variables equality is modeled using an XNOR relation. Arithmetic (and bitwise) operators correspond to UFs[2] except when small constants are involved, where we use the **succ** and **pred** interpreted functions. Comparisons (less than and greater than) are modeled as UPs. A facility to black box certain modules and treat them as atomic is also provided, and mainly used to model memories and FIFOs, etc.

It is worth mentioning that the use of UCLID's **succ**/**pred** to represent constant separation imposes the semantics of unbounded addition/subtraction in Verilog, and disallows arithmetic overflow. This can be easily guaranteed by rewriting the Verilog code if necessary.

4 FALSE NEGATIVES

As mentioned earlier, one potential source of false negatives is UCLID's obliviousness to associativity and commutativity of integer arithmetic. Less obvious, perhaps, is that false negatives can be caused by the concatenation and subfield extraction UFs introduced above. These UFs are also oblivious to the fact that the bit vectors they operate on encode integers and can only guarantee functional consistency. Suppose that X and Y are two bit vectors such that $Y = X + 8$. Clearly, integer arithmetic guarantees that $Y[2:0] = X[2:0]$. This fact, however, cannot be discerned by the concatenation or extraction UFs, and might lead to false errors.

A possible way of tackling these problems is using UCLID's quantification [9]. Unfortunately, UCLID does not have a *complete* translation to quantified formulas, since this fragment of the logic is undecidable. In addition, the over-

---

[2] The UF/UP naming convention is similar to the one used for the concatenation UFs.

head of handling a large number of quantified properties a priori is costly and not needed in most cases.

Instead, we handle counterexamples on demand as they occur during the verification process. Thus far, designers had to manually analyze UCLID's counterexamples. This is impractical since these examples are presented in terms of the abstract model, and consist of interpretations to the terms and applications of UFs and UPs. These interpretations are consistent, as guaranteed by the correctness of UCLID's decision procedure, but are not necessarily meaningful in terms of the original design, and the exact values cannot be validated by means of simulation. Instead, a *satisfiability* check is necessary to validate that the interpretations are consistent with the original Verilog semantics.

Vapor retrieves the semantics of Verilog operators, as well as values of constants, and combines them with the current interpretations of UFs and UPs. To illustrate using the above example, UCLID produces the counterexample `extract_2_3`$(X) \neq$ `extract_2_3`$(\text{succ}^8(X))$. Vapor translates this to $\{X[2:0] \neq Y[2:0]\} \wedge \{Y = X + 8\}$ and passes it on to a theorem prover which finds that the negation of this formula is provable, concluding that the formula is not satisfiable. This helps the designer identify UCLID's counterexample as spurious instead of treating it as a real bug.

5 EXPERIMENTAL RESULTS

Vapor was implemented in C++ for Linux, and integrated with UCLID and Verilog Icarus Compiler [17]. The Verilog subset supported by Vapor is constrained by:

1. Synthesizable behavioral Verilog that is compliant with IEEE Verilog 1364-1995 Standard [13].
2. Sequential logic is synchronized using a single clock edge.
3. Bit- and part-selection use only constant indices, unless it is a 2-D memory array.

In this version of Vapor, we work with ACL2 theorem prover [8]. In case the counter-example includes an UF that did not originate from a Verilog operator (e.g. a user defined UF), we use ACL2 *stub functions* to model it, which allows reasoning with the presence of unknown semantics.

Our first set of examples is taken from the VIS benchmarks [19]. Using Vapor, we abstracted to UCLID and verified a set of control-dominated circuits:.

Table 2: Results for VIS benchmarks

Circuit Name	Verilog Lines	Property/Result
ITC99-b01	91	controller property passed
ITC99-b12	494	controller property failed as specified in the benchmark
ITC99-b13	274	Absence of reset, passed

In the ITC99-b12 UCLID generated a false negative due to the absence of integer semantics in the concatenation UFs. Specifically, it determined that {1'b1,1'b1,1'b1,1'b1}' $\neq$ 4'dF. ACL2 showed this to be false, and the UCLID model was updated (a counter was re-implemented) to eliminate the false error, and eventually lead to a true counterexample.

In [10] Mneimneh et al. present a hybrid verification method for microprocessors, whereby a *checker* processor verifies correctness of a *core* processor. The checker processor

has to be bug-free, and thus has to be formally verified. In order to perform that, a set of properties has to be verified, which exercise the possible scenarios of the checker. For example, starting from a 'valid' state (regular mode, all previous core errors were recovered), the checker will remain in the same mode if the result given by the core is identical to the specifications. To perform this task, we used Vapor to abstract the Verilog description of the checker. We wrote a Verilog specification module that was automatically abstracted as well. The 75 Verilog signals included 438 bit- and part-selections, were modeled by corresponding TERMs, and induced 227 partition TERMs. Applying optimizations, as explained earlier, reduced the model to 12 concatenation/ extraction UFs, yielding a CNF model of 3828 and 10630 variables and clauses respectively, which is a 10X reduction relatively to the unoptimized model. The following errors were found in the Verilog code:

1. A coding error caused the Ra and Rb register indices from the IR to swap. Vapor successfully revealed the error.

2. The HLT (halting) op-code was coded differently in the specification and implementation, and our system issued a counterexample in this case as well, showing the error.

3. The implementation module had a discrepancy relatively to the specification module: The code checks for the *CMPULT* (compare-unsigned less-than) opcode, and assigns the commit register with 64'd1 if Ra<Rb; The specification code, in contrast, follows the mnemonics of the CMPULT instruction as specified by the Alpha Spec [15], and assigns the result register with the value *{63'd0, Ra<Rb}*. Due to the loss of semantics of the concatenation UF in UCLID, a false counterexample was generated, albeit successfully identified by ACL2 to be spurious.

The bug-free version of the checker eventually passed the test. The abstraction and verification process took 83 seconds on a Pentium-III 1GHz machine equipped with 3 GB of RAM, and running Linux Redhat 9. We used UCLID v. 1.0 [18] with ACL2 v. 2.7 [16].

Unlike in [10], where opcodes are manually verified one by one, our system verifies all 14 implemented opcodes simultaneously. Moreover, the checker includes two 256 32-bit word memory arrays which are modeled symbolically by 2 UFs. The advantage of Vapor compared to bit-level tools becomes more important with the presence of big memory arrays.

6   CONCLUSIONS

The motivation behind Vapor emerges from the need for automatic abstraction from the RTL, in an era of increasing complexity of hardware systems. While verification of datapath elements was extensively studied in the past, control and mixed (data/control) property verification has not been systematically approached. In this paper, we presented a systematic approach for safe abstraction from Verilog, and easy counterexample analysis. We are enhancing Vapor to allow utilization of UCLID capabilities in propositional encoding, as well as automatic refinement from the RTL.

7   ACKNOWLEDGEMENTS

REFERENCES

[1] Thomas Ball and Sriram K. Rajamani, "The Slam Project: Debugging System Software via Static Analysis". POPL 2002, January 2002, pages 1-3.

[2] C. Barrett, D. Dill, and J. Levitt, "Validity checking for combinations of theories with equality". In FMCAD '96, LNCS 1166 pages 187-201.

[3] Randal E. Bryant, Shuvendu K. Lahiri, Sanjit A. Seshia, "Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions". In Proc. CAV, July 2002.

[4] R. E. Bryant, S. German, and M. N. Velev, "Exploiting positive equality in a logic of equality with uninterpreted functions". ACM Transactions on Computational Logic, 2(1):93-134, January 2001.

[5] J. R. Burch and D. L. Dill, "Automatic Verification of Pipelined Microprocessor Control". CAV '94, D. L. Dill, ed., LNCS 818, Springer-Verlag, June 1994, pp. 68-80.

[6] Edmund M. Clarke, Orna Grumberg and David E. Long, "Model Checking and Abstraction". ACM-TOPLAS, Vol. 16, No. 5, 1512 -, September 1994.

[7] R. Hojati and R. K. Brayton, "Automatic Datapath Abstraction of Hardware Systems". Proc. Conf. Computer-Aided Verification, Liege, Belgium, June 1995.

[8] Matt Kaufmann and J Moore, "An Industrial Strength Theorem Prover for a Logic Based on Common Lisp". IEEE Transactions on Software Engineering 23(4), April 1997, pp. 203-213

[9] Shevendue K. Lahiri, Sanjit A. Seshia, Randal E. Bryant, "Modeling and Verification of Out-of-Order Microprocessors in UCLID". FMCAD 2002

[10] Maher Mneimneh, Fadi Aloul, Chris Weaver, Saugata Chatterjee, Karem Sakallah andTodd Austin, "Scalable Hybrid Verification of Complex Microprocessors". Proc. 38th DAC, pages 41-46, July 2001.

[11] Sanjit A. Seshia, Shuvendue K. Lahiri, Randal E. Bryant. "A User's Guide to UCLID version 0.1".

[12] Donald E. Thomas and Philip R. Moorby, "The Verilog Hardware Description Language". Kluwer Academic Publishers, Nowell, Massachusetts, 1991.

[13] "IEEE Std 1364-1995, IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language" IEEE, Incs.

[14] Dong Wang, Pei-Hsin Ho, Jiang Long, James Kukula, Yunshan Zhu, Tony Ma, Robert Damiano, "Formal property verification by abstraction refinement with formal, simulation and hybrid engines", In Proceedings of the DAC, pages 25-40, 2001.

[15] Digital Equipment corporation, "The Alpha Architecture Handbook". 1992.

[16] www.cs.utexas.edu/users/moore/acl2/

[17] www.icarus.com/eda/verilog/

[18] www-2.cs.cmu.edu/~uclid

[19] vlsi.colorado.edu/~vis