Uninterpreted Function Abstraction and Refinement for Word-level Model Checking

Yen-Sheng Ho[1], Alan Mishchenko[1], Robert Brayton[1]
[1]Department of EECS, University of California, Berkeley, CA, USA
{ysho, alanmi, brayton}@eecs.berkeley.edu

***Abstract*—Methods for word-level model checking based on purely bit-level techniques have difficulties with heavy arithmetic logic. Word-level and SMT approaches often are limited by relying on (incomplete) bounded model checking. UFAR, a hybrid word- and bit-level approach, addresses these issues, taking advantage of modern bit-level sequential techniques while heavy arithmetic logic is addressed by word-level abstraction and the use of uninterpreted function (UF) constraints. The methods and efficiency improvements developed for UFAR enabled it to prove 2422 of a set of 2492 industrial sequential model checking problems within a 1-hour limit, while a bit-level model checker *super_prove* completed only 2115 of these within the same limit.**

I. Introduction

Model checking (MC) [20], [5], [22] on a Register-Transfer-Level (RTL) word-level netlist is a necessary verification task for applications involving sequential synthesis. In this, an RTL netlist is synthesized into another through retiming, clock-gating, pipelining etc., and MC is required for proving the correctness of the result. These problems are challenging if hard arithmetic operators such as multipliers, adders, and variable shifters are involved, and correspondences between flip flops are not known.

Previous methods in this domain fall into roughly three categories. One directly "bit-blasts" the problem and then uses bit-level sequential verification engines such as IC3/PDR [7], [16], interpolation [18], or BDDs [13]. Another translates the problems into SMT formulas (if possible) and then directly employs SMT solvers such as Boolector [11], Z3 [15], or CVC4 [3]. A third [2], [1], [8] applies *term-level abstraction*, replacing arithmetic operators with uninterpreted functions (UF), and then solving with SMT solvers. However, bit-level techniques are problematic when verifying circuits with heavy arithmetic logic. Most SMT-based approaches rely on (incomplete) bounded model checking (BMC) [6] or induction [21] and may not be applicable.

UFAR (Uninterpreted Function Abstraction and Refinement), is a hybrid word- and bit-level solver, which moderates the above issues. It takes advantage of modern sequential techniques such as PDR and BMC at the bit-level, while heavy word-level logic is tackled by abstraction and the use of uninterpreted function (UF) constraints.

Such techniques are not new, even at the word level. Conventional UF abstraction [2], [1], [8] methods implicitly enforce *all* possible UF constraints among the same functions. This becomes inefficient when the number of similar functions is large. Keys to UFAR's efficiency are how simulations and minimized counterexamples are used to refine abstractions, how constraints are added and removed lazily, which pairs of operators are constrained, and how UF constraints are applied between operators of the same type but with different bit widths. All this requires efficiently iterating between word-level Verilog and AIG representations as refinements are done. These techniques enable UFAR to prove problems containing hundreds of heavy word-level operators.

We prove that UFAR is a sound and complete framework for word-level counterexample guided abstraction and refinement (CEGAR) [14]. It starts with the extreme abstraction with all "problematic" word-level operators (e.g., multipliers, adders, etc) removed (i.e. operator outputs are replaced by unconstrained pseudo primary inputs). This is then bit-blasted and given to a sound and complete bit-level model checker. If a counterexample is returned, UFAR first simulates it on the original netlist to check if it is *real*. If so, UFAR terminates and reports it. Otherwise, the *spurious* counterexample is used to refine the current abstraction. Refinement is done in this context by 1) adding UF constraints between some pairs of chosen compatible operators, and 2) restoring one or more of the removed operators.

We experiment on 2492 industrial benchmarks for sequential RTL (word-level) model checking and show how different refinement methods and heuristics are complementary, each solving more problems in less time, and leading to a final algorithm which solves all but 70 of the benchmarks within a one hour time limit. We show detailed results on 19 examples having ranges of 4-475 multipliers, 21092-302277 AIG nodes, and 358-4785 flip-flops.

This paper first presents background material and formal settings in Section II. The UFAR algorithm is presented in Section III. Several optimization techniques for the algorithm are given in Section IV. Section V gives some details about the UFAR framework, including word-level representation and bit-blasting this into an AIG. Experimental results on an extensive set of industrial problems are presented in Section VI, comparing the effectiveness of the two optimizations and the overall UFAR algorithm. Some conclusions and future work are discussed in Section VII.

II. Bit-Vectors and UF Constraints

In the context of Verilog and its bit-vector operators, we need to be precise about applying UF constraints between pairs of operators. A UF constraint states that for two same-type functions, if their inputs are equal then their outputs are equal. Unfortunately, this is not at all straight-forward when bit-vector operators are involved. Incorrect application of UF constraints can lead to an unsound procedure on the one hand or to a too restrictive application on the other. In this section, we discuss bit-vector operators, define what it means to be the same function, state when and how to make UF constraints valid between two same-type operators, and prove the soundness of the derived methods.

A. The MC problem

We assume that the input RTL design is in *structural Verilog*. In structural Verilog, there are bit-vector (BV) signals including primary inputs (PIs), primary outputs (POs), flip flops (FFs), and internal signals. Flip flops have reset values as initial states. A bit-vector signal s can be either *signed* or *unsigned*, denoted by $signed(s)$. The *bit-width* of s is denoted by $bw(s)$. A design is modeled as a finite state machine (FSM).

Definition 1. A design in structural Verilog is a tuple $M = (I, O, S, S_0, T)$ where I is the set of inputs, O is the set of outputs, S is the set of state variables, S_0 is the set of initial states, and T is the set of (deterministic) transition relations where $T \subseteq I \times S \times S$. If $(i, s, s') \in T$, then there exists a transition from s to s' under i.

The input format is assumed to be *mitered* as a single FSM and a single output, out, representing the property to be checked. If the problem is to prove equivalence between two designs, a miter is created by merging all PIs and merging corresponding mapped FFs (if any). The output out is a Boolean signal, which is the OR of the pairwise XORs of the corresponding outputs of the two designs. Thus it is 1 if the two designs are different. Similarly for property checking, the output is a monitor which signals 1 if the property fails. In terms of linear temporal logic (LTL), the MC problem is formulated as $M \models \mathbf{G}\neg out$, meaning the miter M should never excite the signal out if the property holds.

B. Basics of word-level operators

We focus on abstracting problematic *word-level operators* in a design. The subset of operators considered are all word-level *binary* operators, such as $+, -, *, /, \%, <<, >>, <<<, >>>$. In Verilog, an operator is instantiated by a *structural statement* which only states the function type of the operator and the connection between signals[1]. An operator is modeled as a labeled node with a single output, up to two inputs, and its label of function type.

[1]Without loss of generality, we assume that each statement contains only 1 binary operator. Statements like `x = (a+b)*c` can always be rewritten to `y = a+b` and `x = y*c`.

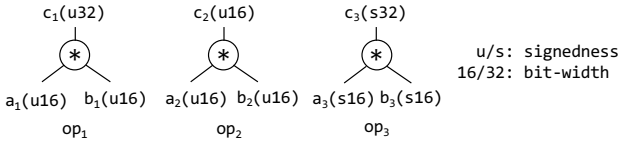


Fig. 1: Three multipliers with different functions.

Definition 2. An operator *op* is a tuple $op = (o, i_1, i_2, t)$ where o is the output signal, i_1 and i_2 are the input signals, and t is the label of function type.

For example, the Verilog statement, `c = a * b`, is modeled as $op = (c, a, b, *)$. Note that the inputs are *ordered* as specified in the Verilog statement. Note also that $*$ is a "function-type" and not a function, since the actual function that would be instantiated would depend on the properties of the signals to which its inputs and output are connected. The necessity of this important distinction will be clarified in the next section.

C. Functions of word-level operators

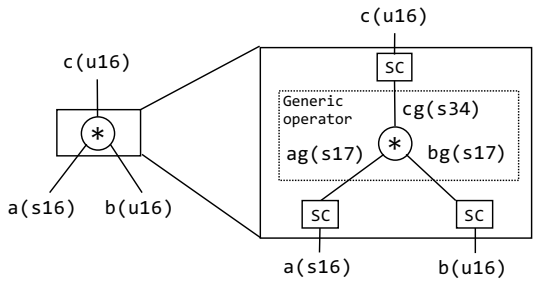
In Verilog, the actual *function* associated with an operator is determined by the bit-widths and signedness of its inputs, output, and function-type. Operators with the same function type do not necessarily have the same function; a function-type represents a set of functions. For example, the three multipliers in Figure 1 all represent different functions. Operators op_1 and op_3 are different since op_1 is *unsigned* multiplication while op_3 is *signed* multiplication. Operator op_2 is different because its output is only 16 bits.

To be precise in what follows, we need to explicitly model what a Verilog front end does when it bit-blasts a Verilog RTL design into a bit-level circuit. For this we need *generic operators* and *signal convertors*.

Definition 3 (Generic operator)**.** A generic operator is a bit-vector operator that agrees with the *integer function* of its function-type. That is, the bit-vector output, when evaluated as an integer, is consistent with the result using the integer function. It has the following properties.

- All of its inputs and output are signed.
- It is a pure arithmetic function parametrized with specified widths for its inputs and output.
- When a generic operation is implemented, its input widths should be compatible with the widths of the signals connected to them.
- The output width should be exactly large enough so as to not impose any restriction on the operation (such as truncation due to overflow).

In order to create signals used or produced by an instantiated generic function, they must be converted from unsigned to signed signals or vice versa. They also need to be converted by truncation, sign extension, or zero extension. We model this by emulating what Verilog does in its assignment operator (=) and concatenation operator ({}), called signal convertors.



(a) The relationship between a multiplier and its generic version. SC denotes signal converters.

```
wire signed [15:0] a;                    wire signed [15:0] a;
wire        [15:0] b;          Expose    wire        [15:0] b;
wire        [15:0] c = a * b;    ⟹      wire        [15:0] c;
                                         wire signed [16:0] ag = {1'b0, a};
                                         wire signed [16:0] bg = {1'b0, b};
                                         wire signed [33:0] cg = ag * bg;
                                         assign             c = cg;
```

(b) A piece of Verilog code for exposing the generic operator.

Fig. 2: An example showing how generic operators are modeled and exposed.

The benefits of explicitly exposing generic operators include 1) it agrees with the arithmetic of not only *bit-vectors* but *integers*, and 2) it unifies unsigned and signed operators. For example, the original bit-vector multiplier, `c = a * b` in Figure 2, does *not* agree with integer arithmetic, meaning that if signals (`a`, `b`, `c`) are evaluated as integers (A, B, C), then they do *not* necessarily satisfy the relation $C = A \times B$ (here $\times$ is integer multiplication). To expose the generic multiplier in Figure 2, first we observe that the original one is unsigned[2] multiplication, so the original inputs are converted to signed generic inputs with leading zeros inserted. Then a generic output of 34 bits is created to prevent overflow. Finally the generic output is converted to the unsigned original one with some upper bits truncated. The generic multiplier, `cg = ag * bg`, agrees with the integer arithmetic by construction. This way, the original multiplier is represented by its generic version and some signal convertors on the inputs and output.

With generic operators, all same-type generic operators (e.g., multipliers) are considered to have the same function since they all agree with their integer functions (e.g. integer multiplication). This is important for uninterpreted function abstraction since uninterpreted function constraints are valid only for same-function classes.

D. Uninterpreted function constraints

The theory of uninterpreted functions (UF) states that given any *function* F with its input X, and any two instances of the same function, (x_1, f_1) and (x_2, f_2), then the Property (1) holds, stating that if the inputs are equal then the two outputs must be equal.

$$(x_1 = x_2) \Rightarrow (f_1 = f_2) \tag{1}$$

[2]In Verilog, an operation is unsigned if at least one input is unsigned.

This is called a UF constraint which is simply a relation implied by any pair of the same two functions.

For Verilog, we need to be more precise about "same function" and "equal inputs". By f and g being the same function we mean that f and g are instantiations of the same generic function-type. By two signals being equal, we will mean that they are signed and bit-wise equal *after* extension. Then Property (1) holds with these modifications. Thus,
a UF constraint is valid between any pair of same function-type generic operators (even if they have different bit widths).

Definition 4. Two signals, s_1 and s_2, are said to be *equal in Verilog* if the corresponding statement, `s1 == s2`, is evaluated to 1 in Verilog.

The precise Verilog semantics for comparing two signals is as follows. It does either zero- or sign-extension for the signal with the smaller bit-width depending on their signedness. If both signals are signed, then it does sign-extension. Otherwise, zero-extension is applied. Two signals are equal if they are bit-wise equal after extension.

Definition 5. For two same function-type generic operators, $op_1 = (o_1, i_{11}, i_{12}, t)$ and $op_2 = (o_2, i_{21}, i_{22}, t)$, the UF constraint, denoted as c, is either Constraint (2) or (3).

- If t is asymmetric:

$$c = (i_{11}{=}{=}i_{21}) \wedge (i_{12}{=}{=}i_{22}) \Rightarrow (o_1{=}{=}o_2) \tag{2}$$

- If t is symmetric:

$$c = \Big((i_{11}{=}{=}i_{21}) \wedge (i_{12}{=}{=}i_{22}) \Rightarrow (o_1{=}{=}o_2)\Big) \bigvee \Big((i_{11}{=}{=}i_{22}) \wedge (i_{12}{=}{=}i_{21}) \Rightarrow (o_1{=}{=}o_2)\Big) \tag{3}$$

We only apply UF constraints between generic instances of same function-type operators. The constraints are created as signals first and then treated as invariant constraints to the model checking problem (see Section III-C). Thus, abstractions are created by 1) using UF constraints and 2) replacing their outputs by new primary inputs (the generic operators are "black-boxed").

Definition 6. A generic instance is said to be *black-boxed* if its output is replaced by a fresh primary input consistent with the generic's output.

Thus the new primary input is signed and has the same width as the instance output being replaced. Note that a UF constraint may be added even though the two operators involved are both white-boxed. This can still be effective as it provides a relation between operators which may not be easy to derive using bit-level operations.

III. UFAR

In this section, the abstraction-refinement algorithm, UFAR, for solving word level model checking problems is described.

Algorithm 1 UFAR

Input: M ▷ M: the input miter
Input: $\mathcal{S}$ ▷ $\mathcal{S}$: the set of problematic operators
Output: status $\in$ { SAT, UNSAT }
1: $\mathcal{B} \leftarrow \mathcal{S}$ ▷ $\mathcal{B}$: the set of black-box operators
2: $\mathcal{P} \leftarrow \emptyset$ ▷ $\mathcal{P}$: the set of UF constraints
3: $M \leftarrow$ EXPOSINGFUNCTIONS(M, $\mathcal{S}$)
4: **while true do**
5: $A \leftarrow$ CREATEABSTRACTION(M, $\mathcal{P}$, $\mathcal{B}$)
6: status, cex $\leftarrow$ MODELCHECKING(A)
7: **if** status = SAT **then**
8: **if** ISREALCEX(M, cex) **then**
9: **return** SAT
10: **else**
11: $\Delta\mathcal{P} \leftarrow$ REFINEUFPAIRS(A, $\mathcal{S}$, cex)
12: **if** $\Delta\mathcal{P} \neq \emptyset$ **then**
13: $\mathcal{P} \leftarrow \mathcal{P} \cup \Delta\mathcal{P}$
14: **continue**
15: **else**
16: $\Delta\mathcal{B} \leftarrow$ REFINEBLACK(M, $\mathcal{P}$, $\mathcal{B}$, cex)
17: $\mathcal{B} \leftarrow \mathcal{B} \backslash \Delta\mathcal{B}$
18: **else**
19: **return** UNSAT

A. The algorithm

Algorithm 1 provides a high level view of UFAR. It takes two inputs; one is a miter M in word-level structural Verilog and the other is $\mathcal{S}$, the set of *problematic* operators that we want to abstract (multipliers in most cases). UFAR will return SAT if a true counterexample is found; otherwise, it concludes that $M \models \mathbf{G}\neg out$ and returns UNSAT. We will prove that UFAR is a sound and complete algorithm in Section III-G.

There are two internal state sets in UFAR. The first is $\mathcal{B}$, the set of *black* operators that will be black-boxed in the abstraction. The second is $\mathcal{P}$, the set of operator pairs whose UF constraints will be added to the abstraction. Initially $\mathcal{B}$ = $\mathcal{S}$, thereby black-boxing all problematic operators, and $\mathcal{P}$ = $\emptyset$.

Algorithm 1 begins with the procedure of exposing generic operators (see Section III-B). It then operates in an abstraction-refinement loop (lines 4–19). Each iteration begins by creating an abstraction based on the current states of the algorithm, which will be discussed in Section III-C. The abstraction is then bit-blasted and solved by state-of-the-art bit-level engines concurrently (see Section III-D). If the solver returns UNSAT, the property is proven and UFAR terminates (line 19). Otherwise a counterexample to the abstraction (cex) exists. If cex is also a counterexample to the original miter, then the property is falsified and UFAR terminates (lines 8–9). Otherwise cex is spurious and UFAR analyzes it to refine the abstraction (lines 11–17).

Refinement is achieved in two phases. UFAR first tries to find new UF pairs that will block cex (see Section III-E). If such are found, UFAR adds them to $\mathcal{P}$ and starts a new iteration (lines 12–14). Otherwise, the second phase is started, where cex is analyzed to determine a set of critical operators ($\Delta\mathcal{B}$) that can block cex (see Section III-F). For the next iteration, UFAR will remove operators in $\Delta\mathcal{B}$ from $\mathcal{B}$ (lines 16–17) and hence these will be *white-boxed*.

B. Exposing generic operators

To expose the generic version of an operator, we modify the Verilog by inserting signed- or zero-extended signal convertors to ensure that it becomes signed and that the bit-width of its output is large enough. The procedure for each operator $op = (o, i_1, i_2, t)$ in the problematic set $\mathcal{S}$ is summarized below.

1) If one of the inputs is *unsigned*, then create zero-extension-by-1 signed signal convertors for both inputs. Denote two generic inputs as a_1 and a_2.
2) Create the generic operator $op_2 = (o_2, a_1, a_2, t)$ where o_2 is signed and has a large enough bit-width.
3) Replace the original output o with the statement $\circ$ = $\circ 2$. Note that this step creates the generic operator op_2, eliminating the original one op.

C. Creating abstractions

An abstraction (A) is created from the original miter (M), using $\mathcal{P}$ and $\mathcal{B}$, the two current states of Algorithm 1. CREATEABSTRACTION operates in two steps:

1) For each pair $p = (op_1, op_2)$ in $\mathcal{P}$, construct a Boolean signal c as defined in UF Constraints (2) or (3). Signal c = 1 implies that a UF constraint is active in M between op_1 and op_2. Signal c is then treated as an invariant constraint.
2) For each operator $op = (o, i_1, i_2, t)$ in $\mathcal{B}$, replace its output o with a fresh primary input ppi with the same signedness and bit-width, i.e. black-box it.

Note that an operator can be in a pair of $\mathcal{P}$ but not $\mathcal{B}$. For example, one benchmark contained a group of 3 multipliers where 2 UF constraints were used between them, but only one of the 3 was needed to be white-boxed for the final proof. Note also that $\mathcal{P}$ and $\mathcal{B}$ are monotone.

We claim that the model A is an abstraction of M.

Lemma 1. *Let N denote the model created after Step 1 (adding UF constraints) in* CREATEABSTRACTION. *N and M satisfy: ($\neg out$ denotes the property)*

$$N \models \boldsymbol{G}\neg out \Leftrightarrow M \models \boldsymbol{G}\neg out.$$

Proof. Consider any constraint signal c. We have $M \models \mathbf{G}c$ since the model M satisfies any valid UF constraint. Thus,

$$\begin{aligned} M \models \mathbf{G}\neg out &\Leftrightarrow M \models \mathbf{G}\neg out \wedge \mathbf{G}c \\ &\Leftrightarrow M, \mathbf{G}c \models \mathbf{G}\neg out \Leftrightarrow N \models \mathbf{G}\neg out \end{aligned}$$

□

Theorem 1. *The model A created by* CREATEABSTRACTION *is an abstraction of the miter M.*

Proof. From Lemma 1, N generated by Step 1 is equisatisfiable to the miter M. In Step 2, it creates the model A by replacing some internal signals in N with fresh primary inputs, which is a known procedure for producing an abstraction. □

D. Model checking using concurrency

To verify the current abstraction at the bit level, we could use a single engine like PDR since it is efficient, sound, and complete. Also, this procedure should be parallelized to take advantage of different engines. Running a BMC engine in parallel with PDR usually finds counterexamples to the current abstraction more efficiently and thus very effective in improving the algorithm. Also, various versions (based on different implementations and parameters) of PDR and BMC complement each other.

E. Refining UF pairs

This is the first phase of refinement. Given a (spurious) counterexample cex to the abstraction, we want to find new UF pairs $\Delta\mathcal{P}$ among operators in $\mathcal{S}$ that can block cex during the next iteration. REFINEUFPAIRS operates in two steps:

1) Simulate cex on the abstraction A to derive an assignment function $\alpha : S \times \mathbb{N} \to \mathbb{Z}$ that maps every signal in A at each time frame to a concrete value.
2) Identify pairs that violate UF constraints and add them to $\Delta\mathcal{P}$. For each time frame t and every pair of operators $(op_1, op_2) : op_1, op_2 \in \mathcal{S}, op_1 \neq op_2$, if the values of the inputs are equal but the outputs are different (Formula 4), then add (op_1, op_2) to $\Delta\mathcal{P}$. Note that we consider both input orders for symmetric operators although this is not shown in Formula 4 for simplicity.

$$(\alpha(i_{11}, t) = \alpha(i_{21}, t) \wedge \alpha(i_{12}, t) = \alpha(i_{22}, t)) \bigwedge \alpha(o_1, t) \neq \alpha(o_2, t) \tag{4}$$

Next, we discuss an upper bound for the size of $\mathcal{P}$.

Theorem 2. *The size of $\mathcal{P}$ in Algorithm 1 is bounded by $|\mathcal{S}|(|\mathcal{S}| - 1)$.*

Proof. Consider the worst case where the operators in $\mathcal{S}$ are all symmetric, then there are $\binom{|\mathcal{S}|}{2}$ pairs of operators with 2 possible permutations of binary inputs. Hence the number of pairs in the algorithm cannot exceed $|\mathcal{S}|(|\mathcal{S}| - 1)$. □

F. Refining black operators

In the second phase of refinement, we want to identify a subset of operators $\Delta\mathcal{B}$ in $\mathcal{B}$ such that if $\Delta\mathcal{B}$ is removed from $\mathcal{B}$, cex will be blocked for the next iteration. We call the procedure of removing elements from $\mathcal{B}$ *white-boxing* and the operators in $\mathcal{S} \setminus \mathcal{B}$ *white boxes*.

A straightforward way of identifying $\Delta\mathcal{B}$ is to simulate cex on the abstraction A and collect those operators in $\mathcal{B}$ that have input-output values inconsistent with their white-box values. However, this approach often finds an overly large $\Delta\mathcal{B}$, resulting in an unnecessarily large abstraction in the next round. Hence, we propose a *proof-based* approach that often obtains a much smaller $\Delta\mathcal{B}$.

The main idea is that if cex is spurious, then the BMC Formula (5) is UNSAT. Here the function $\beta(i, t)$ denotes the assignment of signal i at time t derived from cex being simulated on the original miter M, k is the depth of cex, and out is the miter signal.

$$I_M(\beta(s,0)) \wedge \bigwedge_{t=0}^{k-1} T_M(\beta(i,t), \beta(s,t), \beta(s,t+1)) \wedge \bigvee_{t=0}^{k} out(\beta(i,t), \beta(s,t)) \tag{5}$$

Next, multiplexers are introduced to select between the concrete version (white-box) and the abstracted version (black-box) of an operator. If assumptions are made such that all the concrete ones are selected initially, then the resulting BMC formula would still be UNSAT and a modern SAT solver like MiniSat [17] will return a subset of the assumptions that is sufficient for UNSAT. This is a variation of finding an *unsat core* and the subset returned is our candidate for $\Delta\mathcal{B}$.

The procedure REFINEBLACK operates in five steps.

1) For each pair in $\mathcal{P}$, construct a UF constraint signal and treat it as an invariant constraint on M.
2) For each operator $op = (o, i_1, i_2, t)$ in $\mathcal{B}$, introduce two fresh primary inputs, sel and ppi, where sel is a Boolean signal and ppi a bit-vector signal which is consistent with the output o_{gen} of the associated generic operator. Replace o_{gen} with $o'_{gen} = ITE(sel, o_{gen}, ppi)$ where ITE is the *if-then-else* operator. Depending on the value of sel, either the concrete operator (o_{gen}) or the abstracted one (ppi) flows to the new output o'_{gen}.
3) Denote the model created in Step 2 by N and unroll it with the values of cex plugged in, and keep sel and ppi as the remaining primary inputs. The cex values plugged in are initial states and PIs at each time frame, denoted by $\gamma(s, 0)$ and $\gamma(i, t)$ respectively.
4) Solve the BMC query (6), which is guaranteed to be UNSAT. Note that γ is the assignment function of cex, X_t is the set of sel input signals at time t, PPI_t is the set of ppi input signals at time t, and x_{tn} is the sel signal for the n-th operator at time t. By propagating $x_{tn} = 1$ for all t and n, the query (6) is reduced to (5) by construction ($sel = 1$ means that the concrete version is chosen).

$$I_N(\gamma(s,0)) \wedge \bigwedge_{t=0}^{k-1} T_N(\gamma(i,t), X_t, PPI_t, s_t, s_{t+1}) \wedge \bigvee_{t=0}^{k} out(\gamma(i,t), X_t, PPI_t, s_t) \wedge \bigwedge_{t=0}^{k} \bigwedge_{n=0}^{|X_t|} x_{tn} \tag{6}$$

5) Derive a subset ΔX of X using the assumption interface of a modern SAT solver, and determine $\Delta\mathcal{B}$ from ΔX.

Theorem 3. *The set $\Delta\mathcal{B}$ found by* REFINEBLACK *is not empty ($|\Delta\mathcal{B}| > 0$).*

Proof. $|\Delta\mathcal{B}| = 0$ would mean that the formula (6) is SAT, which contradicts with the fact that cex is spurious. □

G. Analysis of the algorithm

Theorem 4. *Algorithm 1 is sound and complete.*

Proof. (sketch) Algorithm 1 is sound because it returns UNSAT only if the model A satisfies $\mathbf{G}\neg out$, which implies $M \models \mathbf{G}\neg out$ from Theorem 1. As for the completeness, the algorithm returns SAT only if a counterexample is real (line 8–9). Convergence follows because for each iteration (line 4–19), the following statements are true.

- $\mathcal{B}$ and $\mathcal{P}$ are monotone. Either $\mathcal{P}$ becomes strictly bigger (line 12–13) or $\mathcal{B}$ becomes strictly smaller (Theorem 3).
- $|\mathcal{P}|$ is upper bounded by $|\mathcal{S}|(|\mathcal{S}|-1)$ (Theorem 2) and $|\mathcal{B}|$ is lower bounded by 0 (empty set of black boxes).

Therefore the iteration must terminate implying that a definitive answer must have been found. □

IV. Optimization

In this section, we introduce two optimizations (counterexample minimization, and random simulation), each of which improves the basic version of UFAR, Algorithm 1.

A. Minimizing counterexamples

A counterexample can be minimized [19] in the sense that some inputs can be assigned as X (don't care), but the counterexample is still valid after ternary simulation. This way, the number of concrete assignments is minimized.

The main advantage of using minimized counterexamples is that Procedure RefineUFPairs in Algorithm 1 can return potentially fewer, but higher-quality pairs of constraints. This is done by modifying the condition (Formula 4) for identifying and adding a UF constraint, where we check if the inputs are equal and the outputs are different under concrete assignments. With minimized counterexamples, Xs might appear on the outputs of black-box operators (unconstrained pseudo primary inputs). We strengthen the condition by considering only *incompatible* outputs with X assignments. Two assignments are said to be *incompatible* if they have opposite values at some bit position, and *compatible* if they do not. For example, the assignments `XX01` and `X000` are incompatible while `10XX` and `100X` are compatible. With this strengthening, pairs that satisfy Formula 4 under concrete assignments might violate the new condition since their outputs become compatible after the minimization. For example, consider two operators with concrete assignments (o, in_1, in_2), (`0011`, `01`, `10`) and (`0101`, `01`, `10`), which satisfies Formula 4. After the minimization, if the assignments become (`0XX1`, `01`, `10`) and (`XXX1`, `01`, `10`), then the pair will not be added as UF constraints since it violates the strengthened condition with compatible outputs. Thus, it is likely that fewer constraints are added. Also, the constraints we drop are lower-quality in the sense that if they are added, then UFAR will still get similar counterexamples.

B. Performing random simulation

UFAR in Algorithm 1 only finds and applies UF constraints when a counterexample (CEX) is found. However, the CEX returned by a verification engine may not be unique. If UFAR were to get a different CEX, then it might find and apply a different set of UF constraints. This inherent randomness of counterexamples could cause UFAR to take a path where more white boxes are needed for a proof. Thus, random simulation is applied on the original miter to find candidates for "good" UF constraints. The idea is that if a UF constraint is useful for the final proof, then the corresponding pair of operators must be related in some way. This means that for some execution traces they would have identical input assignments.

The procedure of random simulation operates in 2 steps.

1) Determine the parameters: the number of patterns and the number of time frames. Run random simulation on the original miter.
2) For each time frame and for each pair of same function-type generic operators, count the number of times identical input patterns occur.

A threshold is then set for determining what are good candidates of UF constraints (a pair is considered good if its count is above the threshold). A threshold should be chosen carefully since there is a trade-off between the number and the quality of constraints; a lower threshold increases the chances of getting higher-quality UF constraints (in the sense that it is more difficult to find them with counterexamples), but a lower threshold also leads to a larger number of constraints.

V. The UFAR framework

UFAR involves an iteration of abstraction and refinement between two types of representations,

1) AIGs (bit-level circuit), and
2) an internal netlist format called WLC (word-level circuit), a new development in ABC [10] to represent word-level designs.

This capability includes 1) a very fast Verilog based bit-blaster, using Verilog semantics of the WLC box operators, to translate into an AIG, and 2) a duplication-based method to create different WLC netlists at the word level. These developments are critical in making UFAR efficient, to the extent that UFAR run-time is dominated by the SAT solving in the bit-level model checker.

A. Bit-blasting WLC with Verilog semantics

The framework starts with reading in a structural Verilog miter representing the model checking problem. This is translated into a WLC netlist (WLCm) using ABC's structural Verilog parser. Next, the generic operators of all designated "problematic" operators are exposed by creating a new WLC netlist, denoted as WLCg. More details of creating a new WLC netlist are described in the next subsection. It is important to note that WLCg needs to be created only once during the entire flow and represents the fully concretized problem. This is bit-blasted into an AIG, denoted by AIGg to be used later.

The next step is to create a WLC netlist, WLCa, for the current abstraction using WLCg and the state sets $\mathcal{P}$ and $\mathcal{B}$. WLCa is bit-blasted into an AIG, denoted as AIGa. During this, Verilog semantics are used to faithfully interpret the box operators of WLC netlists.

Typically the model checker, applied to AIGa, returns a counterexample which is simulated on AIGg to see if it is spurious. If so, the counterexample is first minimized, using AIGg as reference. This is analyzed to decide the state changes to $\mathcal{P}$ and $\mathcal{B}$, which will be used to block this counterexample. These are implemented by creating a new WLCa from WLCg and the current state sets. Then the next iteration proceeds.

B. Creating abstractions WLCa

In the iteration in the previous section, the next abstraction is constructed as a WLC netlist using inputs $\mathcal{P}$ and $\mathcal{B}$ and WLCg. This is achieved by constructing one intermediate netlist (WLCp) and the final netlist (WLCa). To activate the UF constraints in $\mathcal{P}$, WLCp is created by duplicating WLCg but attaching the UF constraints in $\mathcal{P}$ to the appropriate signals. The boxes listed in $\mathcal{B}$ need to be made black, so the outputs of each such box need to be replaced by new PIs. WLCa is built by duplicating WLCp but with the outputs of the boxes in $\mathcal{B}$ replaced by the new PIs.

VI. Experimental Results

In this section, we present the experimental results of our implementation of UFAR with different optimization methods enabled. The implementation is based on ABC [10] using its latest improvements to Verilog parsing and bit-blasting.

We ran UFAR on a set of 2492 industrial word-level Verilog designs that were synthesized by an industrial tool to be cycle-accurate with the original circuit. Multipliers are the targeted problematic operators for UFAR to abstract. All experiments were performed on a workstation of Intel Xeon E5504 CPUs clocked at 2.0 GHz with 24 GB of RAM.

Comparing our results against publicly available verification tools is difficult. To our knowledge, no tools exist that can parse such designs directly without requiring a major modification. Also, there is no standard format for sequential word level circuits, as there is for the combinational case with SMT-LIB [4]. Therefore we compared results of running a) `super_prove` [9] on bit-blasted designs against b) three UFAR versions with different optimization settings.

For `super_prove`, we simply bit-blasted an input miter and immediately called `super_prove` to solve it. For UFAR we used three versions in this comparison:

- `opt1` means the basic version.
- `opt2` means `opt1` plus counterexample minimization.
- `opt3` means `opt2` plus random simulation.

For all UFAR versions, four bit-level verification engines were run in parallel, 3 variants of PDR and one BMC implementation. BMC is much more efficient at finding counterexamples, while the 3 versions of PDR in combination are efficient at proving a problem UNSAT.

We present the results in Figure 3, where the horizontal axis represents wall-clock time and the vertical axis represents the cumulative number of solved instances. A time-out of 1 hour was enforced for each example. The result of `super_prove` is not shown in Figure 3 because its number of solved instances is 2115, well below the bottom scale of 2330.

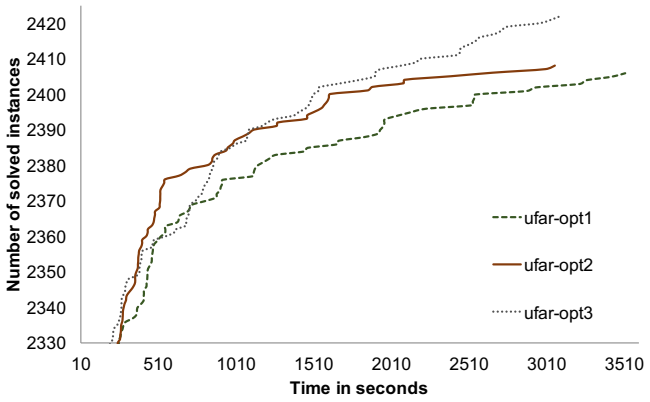


Fig. 3: Comparison of UFAR variants.

super_prove	ufar-opt1	ufar-opt2	ufar-opt3
2115	2398	2408	2422

TABLE I: The numbers of solved instances using different settings. 70 instances remain unsolved.

The `opt2` version is slightly better than `opt1` because the counterexample minimization prevents UFAR from applying too many constraints. The `opt3` version works best because the random simulation finds important UF constraints that can be missed by counterexamples. All solved instances are *unsat*.

Table I shows the numbers of instances finally solved by all versions within the 1-hour time-out. The three versions of UFAR outperform `super_prove`, which is often ineffective in solving problems with many arithmetic operators.

We selected 19 out of 2492 designs to present more detailed results in Table II. The selection is somewhat arbitrary but it does represent designs that are dissimilar and gives an idea of expected ranges of iterations needed, UF constraints used, and white box operators in the final abstractions. We observe the following from the table.

1) UFAR proves most cases with a relatively small number of white-box multipliers.
2) The number and quality of UF constraints are two important factors of performance. If the number is large, then UFAR generally needs more time to run, which is why counterexample-based constraint reduction is important. If the quality is good, then UFAR may prove a problem with fewer white boxes (or none). This supports the using of the random simulation to find good constraints.
3) It takes a nontrivial number of refinements for UFAR to converge, implying that UFAR builds up abstractions gradually. A major challenge is to figure out how to strike a good balance between the number and quality of UF constraints and the number of white boxes needed.
4) The main effect of counterexample minimization seems to be to make the overall algorithm more efficient but solves only 2 additional benchmarks.
5) Random simulation made UFAR faster and helped solve

					sp	ufar-opt1		ufar-opt2		ufar-opt3	
Design	#Mults	#AIGs	#FFs	Result	Time	Time	$i_p/i_w/n_p/n_w$	Time	$i_p/i_w/n_p/n_w$	Time	$i_p/i_w/n_p/n_w$
1	60	187303	2608	unsat	306.71	173.54	1/0/364/0	1127.1	3/0/12/0	3.39	1/0/874/0
2	11	72003	358	unsat			2/1/24/9	2661.2	3/1/30/9	1209.7	1/1/6/9
3	16	115888	550	unsat		20.75	2/0/37/0		3/1/17/10	18.69	1/0/12/0
4	12	49948	819	unsat		4.01	4/0/26/0	4.47	4/0/18/0	1.33	1/0/4/0
5	102	104272	1524	unsat		60.65	3/0/641/0	65.56	6/0/82/0	98.18	2/0/1252/0
6	144	161721	2456	unsat		1225.8	4/0/2366/0	2085.1	12/0/57/0	843.09	1/0/352/0
7	8	192143	5825	unsat		725.36	1/0/14/0	488.99	2/0/4/0	833.42	2/0/11/0
8	14	21092	400	unsat		474.51	3/1/30/10	316.95	3/1/28/8	272.7	2/1/33/8
9	32	46239	972	unsat	8.93		7/3/315/5		3/2/16/4	2428.6	3/3/276/9
10	43	302277	4065	unsat		157.14	3/0/597/0	106.83	4/0/40/0	66.38	1/0/447/0
11	4	25355	1552	unsat	156.16	1.49	1/0/2/0	1.10	1/0/2/0	0.99	1/0/2/0
12	15	49718	1707	unsat		25.33	3/1/40/7	48.46	8/1/34/7	24.93	3/1/52/7
13	21	65892	1997	unsat		40.18	2/1/154/7	50.42	5/1/40/9	102.24	6/1/90/7
14	223	292183	2649	unsat		2548.9	41/8/2398/50		34/9/674/55	1670.8	7/5/1401/37
15	63	91259	875	unsat		1967.4	10/4/915/29	517.86	10/4/142/37	2457.5	3/5/374/35
16	15	184859	4785	unsat		2939.8	1/1/68/4	535.18	2/1/7/3	2169.9	1/1/73/3
17	216	128137	1661	unsat		2231.2	4/0/1107/0		16/0/1943/0	401.76	1/0/394/0
18	253	199466	3751	unsat			118/0/23825/0	6.15	5/2/78/5	686.49	82/2/8638/3
19	475	274801	4204	unsat		470.71	5/0/10556/0	150.18	8/0/172/0	158.99	1/0/268/0

TABLE II: Detailed results of 19 unsat designs. The #Mults/#AIGs/#FFs means the number of multipliers/bit-level AIG nodes/bit-level flip flops. The $i_p/i_w/n_p/n_w$ means the number of iterations of applying new UF constraints/iterations of applying new white boxes/total UF constraints/total white boxes.

4 more benchmarks.

VII. Conclusion and Future Work

UFAR is an algorithm that abstracts (black-boxes) all problematic operators up front and refines them by applying UF constraints and/or white-boxing. We presented two optimization techniques for UFAR. We demonstrated UFAR's scalability on a large set of industrial problems.

For future work, we would like to understand a few of the anomalies in Table II (e.g., Designs 15 and 18) where an optimization caused quite a large slow-down in the solving. We also want to experiment on the 70 remaining unsolved benchmarks to find additional techniques to solve more problems. We plan to extend UFAR to use UF constraints across time frames and to perform refinement more gradually. For example, instead of white-boxing an entire operator, we might *grey-box* it. The under- and over-approximation method [12] can also be modified to fit into our work. An under-approximation is created by restricting bit-widths of primary inputs while an over-approximation can be built by abstracting problematic operators. Last, we plan to integrate modern SMT solvers to investigate possible advantages in this setting.

VIII. Acknowledgements

This work was supported in part by SRC contract 2265.001 as well as NSA under the TRUST project. We also thank industrial sponsors of BVSRC: Altera, Atrenta, Cadence, Calypto, IBM, Intel, Mentor Graphics, Microsemi, Synopsys, and Verific. for their continued support.

References

[1] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. Reveal: A formal verification tool for verilog designs. In *Proc. of LPAR '08*.

[2] Z. S. Andraus and K. A. Sakallah. Automatic abstraction and verification of verilog models. In *Proc. of DAC'04*.

[3] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Proc. of CAV'11*.

[4] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[5] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen. Scalable sequential equivalence checking across arbitrary design transformations. In *Proc. of ICCD'06*.

[6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Proc. of TACAS'99*.

[7] A. R. Bradley. Sat-based model checking without unrolling. In *Proc. of VMCAI'11*.

[8] B. A. Brady, R. E. Bryant, S. A. Seshia, and J. W. O'Leary. ATLAS: automatic term-level abstraction of RTL designs. In *Proc. of MEMOCODE'10*.

[9] R. Brayton, N. Een, and A. Mishchenko. Using speculation for sequential equivalence checking. In *Proc. of IWLS'12*.

[10] R. Brayton and A. Mishchenko. Abc: An academic industrial-strength verification tool. In *Proc. of CAV'10*.

[11] R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Proc. of TACAS'09*.

[12] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *Proc. of TACAS'07*.

[13] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. of LICS'90*.

[14] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of CAV'00*.

[15] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proc. of TACAS'08*.

[16] N. Eén, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *Proc. of FMCAD'11*.

[17] N. Eén and N. Sörensson. An extensible sat-solver. In *Proc. of SAT'03*.

[18] K. L. McMillan. Interpolation and sat-based model checking. In *Proc. of CAV'03*.

[19] A. Mishchenko, N. Eén, and R. Brayton. A toolbox for counter-example analysis and optimization. In *Proc. of IWLS'13*.

[20] C. Pixley. A theory and implementation of sequential hardware equivalence. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 2006.

[21] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *Proc. of FMCAD'00*.

[22] C. A. van Eijk. Sequential equivalence checking based on structural similarities. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 2006.