Efficient Abstraction and Refinement for Word-level Model Checking

Yen-Sheng Ho

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-198
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-198.html

December 8, 2017

Efficient Abstraction and Refinement for Word-level Model Checking

By

Yen-Sheng Ho

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Robert K. Brayton, Chair
Professor Sanjit A. Seshia
Professor Alper Atamtürk

Fall 2017

Efficient Abstraction and Refinement for Word-level Model Checking

Yen-Sheng Ho

Abstract

Efficient Abstraction and Refinement for Word-level Model Checking

by

Yen-Sheng Ho

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Robert K. Brayton, Chair

Model Checking (MC) on a word-level circuit has important applications in the IC design industry, where MC is used to prove that a word-level circuit always satisfies a set of given properties. MC is challenging at the word level, when complex arithmetic operators like multipliers are involved. Abstraction and refinement are commonly used to address challenging MC problems. If an abstraction is proved, so is the original problem. Otherwise, spurious counterexamples are analyzed to refine abstractions. Although many abstraction refinement algorithms for word-level MC have been developed, few take full advantage of state-of-the-art bit-level MC algorithms, like Property Directed Reachability (PDR), which is considered the most efficient method for deriving unbounded proofs. Therefore, this thesis presents several techniques that enable efficient word-level MC by performing abstraction refinement at the word-level while verifying abstractions at the bit-level.

To compute good abstractions and refinements at the word-level, novel refinement strategies were proposed to take advantage of both structural and proof-based analysis. The proposed strategies are shown to achieve a good balance between the sizes of the abstractions and the number of refinement iterations needed for convergence.

To achieve efficient integration of abstraction refinement and bit-level MC algorithms, a bit-level algorithm, PDRA, was created, that minimally modifies the original PDR algorithm to perform on-the-fly abstraction refinement. Inspired by this, a word-level algorithm, PDR-WLA, was developed that efficiently integrates bit-level PDR implementations with word-level abstraction refinement. An important feature is the re-use of reachability information learned in previous refinement iterations.

Motivated by real industrial benchmarks characterized by having many related arithmetic operators, a word-level MC algorithm, UFAR, was proposed that uses uninterpreted functions (UF) constraints as a method of refinement. A UF constraint, between a pair of word-level operators, requires that if their inputs are equal then their outputs are equal. To enhance the applicability of UF constraints, a procedure for normalizing operators was devised. This allows UF constraints to be applied to a pair of same-type operators with

different operator sizes and signedness. UFAR explicitly encodes UF constraints into word-level circuits. This allows any bit-level or word-level MC algorithm to be used, including both PDRA and PDR-WLA.

All these developments were implemented in a publically available model checking system, ABC. Experiments were done which show that UFAR successfully solves most cases in a large set of challenging benchmarks provided by an industrial collaborator.

To My Family.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my deepest gratitude to my advisor, Professor Robert K. Brayton, for his endless patience of guiding me through my PhD years. Bob was always there and available. My favorite moments are those one-on-one meetings with Bob at his place. There were depressing times, but I gained energy and hope from Bob every time we met. Bob listened and understood my problems, giving his invaluable advice and warm encouragement. I always feel strongly supported by Bob. Without his support and guidance, I would not have accomplished this dissertation.

I would like to thank Dr. Alan Mishchenko for many stimulating discussions and critical feedback on my research. I was fortunate to have him as my unofficial co-advisor. I also learned a lot from his experience in developing ABC. Alan's enthusiasm for research and programming always inspired me to become a better researcher and programmer.

I would like to thank Professor Sanjit A. Seshia and Professor Alper Atamtürk for their constructive feedback on this dissertation. I would also like to thank Professor Edward A. Lee for serving on my qualifying examination committee. I would like to give special thanks to Professor Seshia for his guidance and mentoring during my first semester at Berkeley. I am grateful to Shirley Salanio for her help and support in graduate matters.

I would like to thank Pankaj Chauhan and Pritam Roy for mentoring me during my internships at Calypto. They had good influences in this thesis. I would also like to thank the whole SLEC team for making my internships at Calypto enjoyable and fulfilling.

I would like to thank Dr. Niklas Eén, Sayak Ray, Baruch Sterin, Jiang Long, and Yu-Yun Dai for many fruitful conversations and all the wonderful times we spent together in Bob's group. I am also thankful to Niklas for his mentoring in my early PhD years.

I would like to thank Wei Yang Tan, Antonio Iannopollo, Hokeun Kim, Tianshi Wang, Garvit Juniwal, Nishant Totla, Chung-Wei Lin, Pierluigi Nuzzo, Wenchao Li, Shromona Ghosh, and Baihong Jin for the intriguing conversations and all the fun times in the DOP center.

I would like to thank Semiconductor Research Corporation, National Science Foundation, National Security Agency, Altera, Atrenta, Cadence, Calypto, IBM, Intel, Mentor Graphics, Microsemi, Synopsys, and Verific for their financial supports and sponsorship.

Finally, I would like to thank my parents, my sister, and my brother for their unconditional love and strong support through all the years.

Chapter 1

Introduction

The development of integrated circuit (IC) design has been one of the main driving forces in technology. Bugs or errors in an IC can cause serious problems to a company or even human lives. Therefore, it is important to verify a circuit, checking if it is designed correctly according to its specifications. This process is called *verification*.

Verification can be either *simulation*-based or *formal*-based. In simulation-based techniques, an input vector is simulated on a circuit and the output results are checked against the specification. If the results are not consistent, then a bug is found. Otherwise, the circuit works properly under this input vector. However, as the complexity of an IC grows, it is impossible to simulate every possible input vector. If a simulation-based verification cannot cover all possible situations, then it cannot be concluded that a circuit is free of bugs. On the other hand, formal-based techniques automatically consider all possible inputs with the use of symbolic inputs and mathematical models of a circuit. This methodology is also known as *model checking* (MC), where a circuit is checked if a property holds under all possible situations. While MC is more powerful than simulation, it is less scalable on complicated designs. Improving the scalability of MC has been an active research area ever since its inception.

In the following sections, we introduce background material that is useful for understanding the problems, challenges, motivations, and contributions of this dissertation. Section 1.1 describes word-level circuits, which are the typical level of descriptions used for IC designs. These descriptions are the inputs to word-level model checking problems. Model checking word-level problems is the major focus of this thesis. The problem of model checking and state-of-the-art algorithms are presented in Section 1.2 and Section 1.3. Abstraction and refinement are discussed in Section 1.4. Challenges in word-level MC and motivations of this dissertation are given in Section 1.5. This chapter is then concluded in Section 1.6 by outlining the main contributions. The organization of this thesis in given Section 1.7.

1.1 Word-level Circuits

Practical circuits are usually specified at the *word level*, where bits are grouped as *words* or *bit-vectors*. A word-level circuit can be modeled at the *Register-Transfer Level (RTL)* in a hardware description language like *Verilog*.

Definition 1.1. A *word-level signal*, or *bit-vector*, of size m is a finite function whose domain is $\{x|x \in \mathbb{N}, 0 \leq x < m\}$ and the co-domain is $\{0, 1\}$. A word-level signal b with size m is denoted as b^m. For example, a word-level signal b^3 can have $b^3(0) = 1$, $b^3(1) = 0$, and $b^3(2) = 1$.

A word-level circuit is composed of two main parts, registers and combinational logic. Registers synchronize the behaviors of a circuit according to *clocks*. Registers are the only elements in a circuit that have memory, storing logic values computed in the previous clock cycle. The combinational logic in a circuit determines the output values at the current cycle as well as new register values for the next clock cycle.

1.1.1 Registers

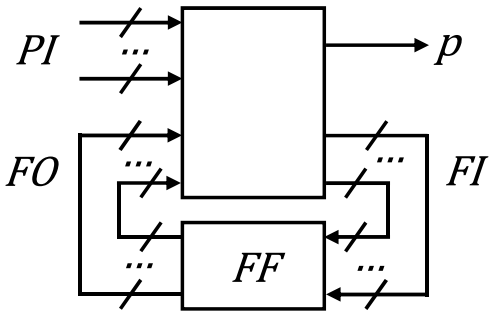
A word-level circuit is assumed to have a single clock and a single type of register: a D flip-flop. If a circuit has multiple clocks or different types of registers, it can be normalized to a single clock through a technique called *phase abstraction* [BK05]. With this normalization, a circuit has a single universal clock that ticks periodically, generating *cycles*. All registers are initialized with some given values, and are updated to new values at each clock cycle based on their inputs. In particular, every register is a D flip-flop that contains three components:

1. *The initial state*. This is the initial value for the register, which can be either a constant or a non-deterministic value.

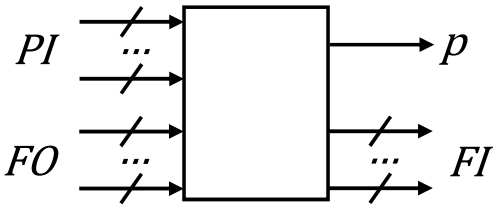
2. *The current state*. This is the output of this register, which is the value available at the current cycle. It is also called *the flop output (FO)*.

3. *The next state*. This is the input of this register, which is a function of the inputs and the current states in this circuit. The register will be updated to this value at the beginning of the next cycle. It is also called *the flop input (FI)*.

For the rest of this thesis, we will use the terms *flip-flops* or simply *flops* to refer to registers in a circuit.



(a) A word-level circuit with feedback loops (cycles) at flip-flops (FF).



(b) The DAG transformed from the cyclic circuit in (a) with FFs modeled as flop outputs (FO) and flop inputs (FI).

Figure 1.1: An example showing a word-level circuit with loops at flip-flops modeled as a directed acyclic graph (DAG).

1.1.2 Word-level Circuits as Directed Acyclic Graphs

A word-level circuit often contains feedback loops broken by flip-flops (FF), because next states are usually functions of current states. Given the simplified register model described in Section 1.1.1, a word-level circuit can be modeled as a directed acyclic graph (DAG) by breaking the feedback loops at the flip-flops, illustrated in Figure 1.1. At each clock cycle, the values of flop inputs (FI), or next states, and primary outputs (PO) are functions of flop outputs (FO), or current states, and primary inputs (PI). Internal nodes in a circuit are word-level operators such as such as $\{+, -, *, /, \%, \ll, \gg, \lll, \ggg\}$ as well as the usual bit-level operators such as $\{\neg, \wedge, \vee\}$. A word-level operator takes input signals (operands) and produces a result in its output signal according to its functionality.

Example 1.1. Figure 1.2 shows an example of a word-level circuit described in Verilog and its visualization as a DAG. The nodes shown in Figure 1.2b are PIs = {`a`}, POs = {`out`}, FOs = {`ff`}, FIs = {`ff_in`}, constants {`0`, `1`}, and word-level operators {$\ll$, +, `[ ]`}.

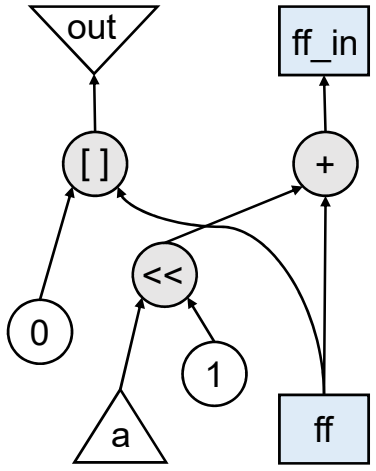
1.1.3 Bit-blasting

A word-level circuit can be *bit-blasted* into an equivalent bit-level circuit consisting of only logic gates such as $\{\neg, \wedge, \vee\}$. The process is called *bit-blasting*.

Example 1.2. Figure 1.3 shows a bit-level circuit bit-blasted from the word-level circuit shown in Figure 1.2. The bit-level circuit is an And-Invertor Graph (AIG) with AND gates ($\wedge$) and invertors (dashed arrows).

```
module main (a, clk, out);
input  a;
input  clk;
output out;

wire     [2:0] ff_in;
reg      [2:0] ff;
initial        ff = 3'b000;

assign ff_in = ff + (a << 1);
always @(posedge clk) begin
   ff <= ff_in;
end

assign out   = ff[0];

endmodule
```

(a) The Verilog snippet of a word-level circuit.



(b) The DAG visualization of the circuit described in (a).

Figure 1.2: An example showing a word-level circuit described in Verilog and its visualization as a DAG.

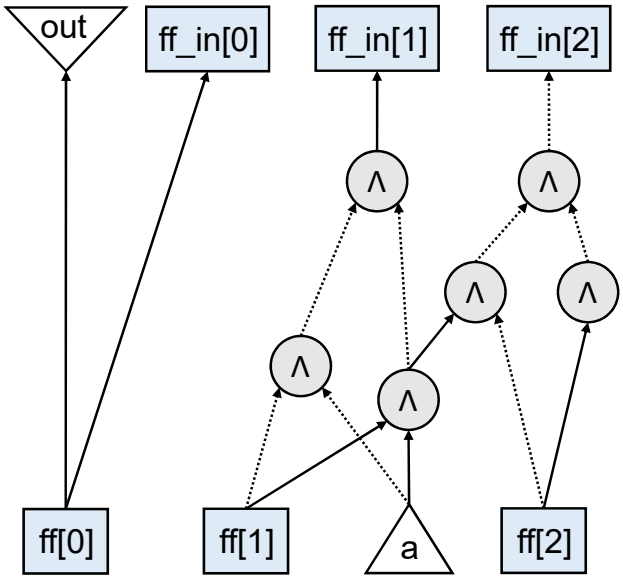


Figure 1.3: A bit-level circuit bit-blasted from the word-level circuit in Figure 1.2. Symbol ∧ denotes a logic AND gate; dash arrows represent inverters.

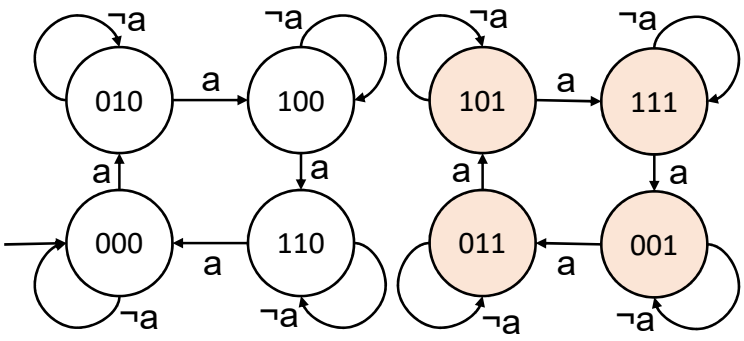


Figure 1.4: The state transition graph of the circuit in Figure 1.2.

1.1.4 Word-level Circuits as Finite State Machines

In a word-level circuit, the state space is finite because there are a finite number of finite-sized flip-flops. Moreover, under the simplified register model, the behavior of a circuit at each cycle is well defined by its combinational logic. Therefore, a word-level circuit can be modeled also as a finite state machine (FSM).

Definition 1.2. An *FSM* is a tuple $M = (I, O, S, Init, T)$ where I is the set of PIs, O is the set of POs, S is the set of FFs, $Init$ is the set of initial states, and T is the set of (deterministic) transition relations where $T \subseteq I \times S \times S$. If $(i, s, s') \in T$, there exists a transition from s to s' under input i.

Example 1.3. The word-level circuit shown in Figure 1.2 can be modeled as an FSM. Its state transition graph is shown in Figure 1.4. The shaded states are the ones that make the PO $out = 1$.

1.2 The Model Checking Problem

The inputs to the model checking (MC) problem are an FSM, M, and a property specified in Linear Temporal Logic (LTL) [Pnu77]. The LTL property is assumed to be transformed into a *safety* property, p, through a technique for example proposed by Claessen et al. [CES13]. In terms of LTL, the MC problem is to check if the following formula holds.

$$M \models \mathbf{G}p.$$

If the above formula is true, then the model M satisfies the property p *globally*.

An MC algorithm should either report a *counterexample* (CEX) that falsifies the property or produce an *inductive invariant* proving that the property always holds.

Definition 1.3. A *counterexample (CEX)* is a trace consisting of a sequence of PI assignments driving the design from an initial state into a state falsifying the property.

Definition 1.4. An *inductive invariant* (Inv) proving a property P is a set of states that a) contains the initial states ($Init$), b) does not contain states falsifying the property, and c) contains all states 1-step reachable from the states contained in this invariant. Formally, it is a predicate function satisfying the properties below.

1. $Init(s) \implies Inv(s)$
2. $Inv(s) \wedge T(i, s, s') \implies Inv(s')$
3. $Inv(s) \implies P(s)$

Example 1.4. For the circuit shown in Figure 1.2, if we let the PO represent the violation of the property, then the problem has a unique inductive invariant, $Inv = \neg$`ff[0]`, which proves the property. The invariant contains all four non-shaded states in Figure 1.4.

1.3 Algorithms for Model Checking Problems

Fundamental algorithms for model checking have been developed over the past few decades. Symbolic model checking with binary decision diagrams (BDD) was the first symbolic technique for MC problems [Bry92, BCM+92]. Later, Boolean satisfiability (SAT) solvers achieved major breakthroughs with conflict-driven clause learning [SS96] and two-literal watching [MMZ+01]. The first SAT-based model checking algorithm was bounded model checking (BMC) proposed by Biere et al. [BCCZ99]. k-induction was proposed to enhance BMC with unbounded proofs [SSS00]. Interpolation-based model checking was the first algorithm that explicitly computes over-approximations of reachable states to prove a property [McM03]. IC3 was proposed in 2011 [Bra11], later improved as Property Directed Reachability (PDR) [EMB11], has been considered the best performing algorithm in proving properties. In this dissertation, we focus on three algorithms that work best for word-level model checking: BMC, k-induction, and PDR.

Given an FSM $M = (I, O, S, Init, T)$ and a property p, BMC and k-induction both require *unrolling* of a circuit up to a certain depth k and then checking the satisfiability of the propositional formulas below.

$$Init(s_0) \wedge \bigwedge_{t=0}^{k-1} T(i_t, s_t, s_{t+1}) \wedge \bigvee_{t=0}^{k} \neg p(s_t) \tag{1.1}$$

$$\bigwedge_{t=0}^{k} T(i_t, s_t, s_{t+1}) \wedge \bigwedge_{t=0}^{k} p(s_t) \wedge \neg p(s_{k+1}) \wedge \bigwedge_{0 \le i < j \le k+1} s_i \neq s_j \tag{1.2}$$

BMC checks Formula (1.1): if it is satisfiable (SAT), then BMC finds a CEX that falsifies the property within k cycles. Otherwise, the formula is unsatisfiable (UNSAT), meaning there is no CEX within k cycles. BMC then either terminates with this bounded proof, or increases the depth k for the next check. k-induction checks an additional formula (1.2): if it is UNSAT, then it is proved that there exists no trace with the first k states satisfying the property and the $k+1$-th state falsifying the property. If Formula (1.1) is also UNSAT, the property holds globally. Formula (1.1) and (1.2) can be viewed as the *base case* and *inductive step* in mathematical induction, respectively.

In contrast, PDR does not require unrolling of a circuit. Instead of checking a large and difficult formula like (1.1) or (1.2), PDR incrementally learns reachability information in a circuit using many small formulas like (1.3). More details of PDR will be presented in Chapter 3.

$$R_j(s) \wedge \neg c(s) \wedge T(i, s, s') \wedge c(s') \tag{1.3}$$

1.4 Counterexample-guided Abstraction Refinement (CEGAR)

In practice, directly solving a MC problem may not scale well due to the high complexity of the problem. *Abstraction* is often used to simplify the original problem. The idea is to create a new abstraction circuit such that if it can be proved that there is no CEX for the abstraction, then there is no CEX for the original circuit also. On the other hand, if a CEX is found in the abstraction, the CEX can be either *real* or *spurious* (Definition 1.5).

Definition 1.5. Given an original circuit M and an abstraction circuit A of M, a CEX of A is *real* if it can falsify the property on M. Otherwise, it is *spurious*.

A spurious CEX does not tell whether the property is proved or falsified in the original circuit. However, it provides information on how to *refine* the current abstraction to make the property more provable. An ideal refined abstraction is still an abstraction of the original model, but refutes the existence of the previous spurious CEX. The process of creating a new abstraction to block a spurious CEX is called *refinement*.

Counterexample-guided abstraction refinement (CEGAR) is a framework that combines the ideas of abstraction and refinement for MC problems [CGJ+00]. An overview of CEGAR is shown in Algorithm 1.1. CEGAR starts by creating an initial abstraction from the original

Algorithm 1.1 Counterexample-guided Abstraction Refinement (CEGAR)

Input: M ▷ M: the input circuit
Output: status ∈ { SAT, UNSAT }
1: $A \leftarrow$ CreateAbstraction(M)
2: **while true do**
3: cex $\leftarrow$ ModelChecking(A)
4: **if** cex $\neq \emptyset$ **then**
5: **if** IsRealCEX(M, cex) **then**
6: **return** SAT
7: **else**
8: $A \leftarrow$ Refine(M, A, cex)
9: **else** ▷ No CEX exists in M.
10: **return** UNSAT

circuit M (line 1). Next, an abstraction-refinement loop is entered (line 2) where each iteration begins by verifying the current abstraction with model checking (line 3). If an MC solver concludes that there is no CEX for the problem, then an inductive invariant (line 9) has been found. The property is proved, and CEGAR returns UNSAT (line 10). Otherwise, a CEX to the abstraction, *cex*, exists (line 4) and is then checked against the original circuit M to see if it is *real* (line 5). If yes, the property is falsified and CEGAR returns SAT (line 6); otherwise *cex* is analyzed to refine the current abstraction (line 8). A new abstraction is then created and a new iteration begins.

1.5 Challenges and Motivations

Many CEGAR-based algorithms for *word-level* MC have been proposed [AS04, JKSC05, BKO+07, ALS08, BBSO10, BBS11, LS14]. However, none take full advantage of recent developments at the bit level, whereas PDR has been improved constantly [HBS13, IG15, GR16]. For example, some word-level techniques rely on BMC and k-induction with satisfiability modulo theories (SMT) solvers [AS04, BKO+07, ALS08, BSST09, BBSO10, BBS11]. As discussed in Section 1.3, BMC and k-induction require unrolling of a circuit to a certain depth k. This becomes inefficient if deep unrolling is required to derive a unbounded proof. On the other hand, PDR does not require unrolling and has been shown to be more efficient in deriving unbounded proofs [Bra11]. In VCEGAR [JKSC05], abstractions are verified with BDD-based algorithms, which do not scale to large problems due to possible memory explosion. Lee and Sakallah developed a CEGAR-based algorithm based on their SMT-based PDR [LS14]. However this was not shown to be competitive with bit-level PDR algorithms.

Another way of solving word-level MC problems is to bit-blast a circuit and to solve

the resulting bit-level circuit with state-of-the-art bit-level model checkers like ABC [BM10]. However, direct bit-blasting loses word-level information, such as word boundaries and operator semantics that can be useful in solving a problem.

To address those challenges, this dissertation proposes a new CEGAR-based paradigm: performing abstraction refinement at the word level while proving bit-blasted abstractions with state-of-the-art bit-level MC algorithms. This paradigm offers two main benefits:

1. It takes full advantage of recent developments in bit-level MC algorithms. In particular, all the recent improvements in bit-level PDR algorithms as well as the winners in the Hardware Model Checking Competition [BvDH17] can be integrated directly under this paradigm.

2. It takes advantage of word-level information by performing abstraction and refinement at the word level. This greatly improves the approach of direct bit-blasting without sophisticated use of word-level information.

1.6 Contributions

In this dissertation, we propose several techniques that enable efficient abstraction and refinement for word-level MC problems. We focus on the proposed CEGAR-based paradigm of computing abstraction and refinement at the word-level and verifying abstractions at the bit-level.

The success of a CEGAR-based algorithm mainly relies on a) the quality of abstractions created in each iteration, and b) the number of iterations for the CEGAR flow to terminate. The key challenge is to have a good refinement procedure given a spurious CEX. Therefore we propose new word-level refinement strategies that take advantage of both structural and proof-based analysis [HCR+16, HMB17]. The proposed strategies achieve a good balance between the sizes of abstractions and the number of iterations, leading to an efficient CEGAR flow compared to previous methods.

With good refinement strategies, the next challenge is to integrate MC algorithms into the CEGAR flow in an efficient way. For example, a straightforward integration of PDR and CEGAR is to use a fresh PDR engine to verify the current abstraction (line 3 in Algorithm 1.1), which is inefficient because reachability information learned in the current iteration would be lost in the next iteration. Therefore we first propose a bit-level algorithm, PDRA, which enhances PDR with an embedded localization abstraction [HMBE17]. In PDRA, we show that CEGAR can be built into the PDR algorithm with only slight modifications. The result is a PDR engine which is minimally modified to perform on-the-fly abstraction, where reach-

ability information is preserved across iterations. Inspired by this, we propose PDR-WLA, a word-level algorithm that efficiently integrates bit-level PDR with word-level abstraction and refinement [HMB17]. PDR-WLA re-uses reachability information learned in previous iterations explicitly to achieve an efficient integration.

The next challenge was a special class of word-level MC benchmarks, which was provided to us by industry. The benchmarks are characterized by containing hundreds of multipliers and adders where some of those operators may be related. For example, a pair of multipliers may have identical inputs only at certain clock cycles. Moreover, the related multipliers may have different sizes and signedness due to heavy synthesis and optimations done at the Verilog level. Given the above characteristics, simple localization abstraction does not work because localization cannot capture these relationships between multipliers. Therefore we propose UFAR, a word-level CEGAR-based framework, which takes advantage of the theory of uninterpreted functions (UF), by using UF constraints as a method of refinement [HCR+16]. A UF constraint, between a pair of multipliers, states that if their inputs are equal then their outputs must be equal. This is shown to be effective for a pair of *related* multipliers. However, a UF constraint is not necessarily valid between two multipliers with different sizes and/or signedness, even if they are related, because the two multipliers may implement different *functions*. Therefore in UFAR, we propose a way to *normalize* multipliers so that a UF constraint is applicable to *any* pair of multipliers in a design. The UFAR framework integrates all the proposed techniques presented in this thesis, leading to significant improvements in solving such challenging sets of industrial benchmarks.

1.7 Organization

This dissertation starts with new proposed refinement strategies in Chapter 2. The algorithm of PDRA, which enhances bit-level PDR with localization abstraction, is presented in Chapter 3. The algorithm of PDR-WLA, which efficiently integrates word-level abstraction with bit-level PDR, is discussed in Chapter 4. The framework of UFAR, which features using uninterpreted function constraints as a way of refinement, is presented in Chapter 5. This dissertation is concluded in Chapter 6.

Chapter 2

Refinement Strategies for Word-level Abstraction

This chapter presents novel refinement strategies that take advantage of both structural and proof-based analysis in order to compute good abstraction refinement at the word-level.

2.1 Introduction

Localization abstraction [WJK+01] has been shown effective in simplifying the original problem for model checkers. Given a word-level circuit and a set of target signals (e.g., outputs of arithmetic operators), an abstraction is created by replacing the target signals with free (unconstrained) variables called *pseudo PIs* (PPIs).

While the abstraction scheme is straightforward, it is challenging to make it efficient in the CEGAR flow, shown in Algorithm 1.1. There are two main factors to be considered: a) the quality of the current abstraction in each iteration and b) the number of iterations taken for CEGAR to conclude the result. If we get an abstraction that is unnecessarily complex (one that is very close to the original circuit), then the model checker could get stuck at proving this instance (line 3 in Algorithm 1.1). Therefore, abstractions should be refined gradually so that complexities are built up over several iterations. However, the refinement should not be too incremental, meaning there should be only slight differences between the refinement and the current abstraction, otherwise it could lead to an unnecessarily large number of iterations for CEGAR to terminate. Thus, it is important to have good refinement strategies that strike a good balance between the two trade-offs (refinement too much vs. too little), leading to an efficient CEGAR-based algorithm.

In this chapter, two novel refinement strategies are proposed that are able to solve more of the difficult cases than previous approaches, when evaluated on a set of 195 industrial Verilog benchmarks. The first strategy is called Proof-based Refinement (PBR). We propose a way to encode assumption variables into a circuit so that PBR can take advantage of assumption interfaces, which are available in modern SAT solvers. This provides a good and efficient way of estimating the minimum UNSAT subset (MUS) of assumptions. The second strategy is called the Maximum Fan-out Free Cone (MFFC) refinement. The idea is that using a simple structural analysis of a circuit (MFFC), unnecessary iterations can be avoided by refining additional relevant signals, which improves performances.

This chapter starts with background material in Section 2.2. An important previous approach, Simulation-based Refinement (SBR), is presented in Section 2.3. Our first proposed strategy, PBR, is given in Section 2.4. The refinement strategy of MFFC is presented in Section 2.5. Comparison of different refinement strategies is discussed with examples in Section 2.6. Related work is given in Section 2.7. Experimental results are discussed in Section 2.8. Conclusions are presented in Section 2.9.

2.2 Preliminaries

2.2.1 Word-level Localization Abstraction

In localization abstraction [WJK+01], given a word-level circuit and a set of target signals (e.g., outputs of arithmetic operators), an abstraction is created by replacing the target signals with free variables called *pseudo PIs* (PPIs). Localization is not necessarily restricted to flip flops; *any* signal can be abstracted, similar to GLA [MEB+13]. In this thesis, *RPIs* are used to denote the *real* PIs in the original circuit and *PPIs* are used for newly created PIs in localization abstraction.

Algorithm 2.1 presents the procedure of creating a localization abstraction (W_A) from the original circuit (W_M) and identifies a set of signals to be abstracted ($\mathcal{B}$). The procedure uses a signal map (U) that maps a signal in the original circuit (W_M) to a signal in the abstraction (W_A) (line 3). The procedure iterates through all the signals in W_M in a topological order (from [PIs, FOs] to [POs, FIs]) (line 4). For each signal v, if v is in the abstraction set ($\mathcal{B}$), then a new PI is created in the new abstraction (W_A), and the created PI is mapped to v (line 6). Otherwise, a copy of v is created in W_A, and the copy is mapped to v (line 8). Then the inputs of $U[v]$ in W_A are properly attached by iterating the inputs of v in W_M and using the signal map U (line 10). For example, if a signal v has inputs x and y in the original circuit (W_M), then the inputs of its counterpart $U[v]$ in the abstraction (W_A) need to be attached to the counterparts of x and y, which are $U[x]$ and $U[y]$. Finally, the procedure

returns the newly created abstraction W_A.

Algorithm 2.1 Word-level Localization Abstraction

```
1: procedure CreateAbstraction(W_M, B)
2:     W_A ← ∅
3:     U ← ∅
4:     for v in TopologicalSort(W_M) do
5:         if v ∈ B then
6:             U[v] ← CreatePI(W_A)
7:         else
8:             U[v] ← CopySignal(W_A, v)
9:             for x in GetInputs(v) do
10:                AttachInput(U[v], U[x])
11:    return W_A
```

Example 2.1. Consider the circuits in Figure 2.1. The PO, *out*, in Figure 2.1a is constant-0, since both $2 \times x \equiv x + x$ and $2 \times y \equiv y + y$ are true. Figure 2.1b is the result of abstracting the original circuit with the set of abstraction signals $\mathcal{B} = \{n_4, n_5, n_6, n_7, n_8, n_9\}$. The 6 signals are replaced with 6 PPIs $\{a, b, c, d, e, f\}$. There are 8 PIs in the abstraction, including RPIs $= \{x, y\}$ and PPIs $= \{a, b, c, d, e, f\}$. Note that while the example is combinational for illustration purposes, the abstraction scheme applies generally to circuits with FFs.

2.2.2 Counterexamples

A counterexample (CEX) of length k is a sequence of concrete assignments of PIs (including RPIs and PPIs) that drive the input model from its initial states into a state falsifying the property. An example of a CEX trace is shown in (2.1), where s_j denotes the value of the j-th state and i_j denotes the PI assignment at the j-th cycle.

$$s_0 \xrightarrow{i_0} s_1 \xrightarrow{i_1} s_2 \xrightarrow{i_2} \ldots \xrightarrow{i_{k-1}} s_k \tag{2.1}$$

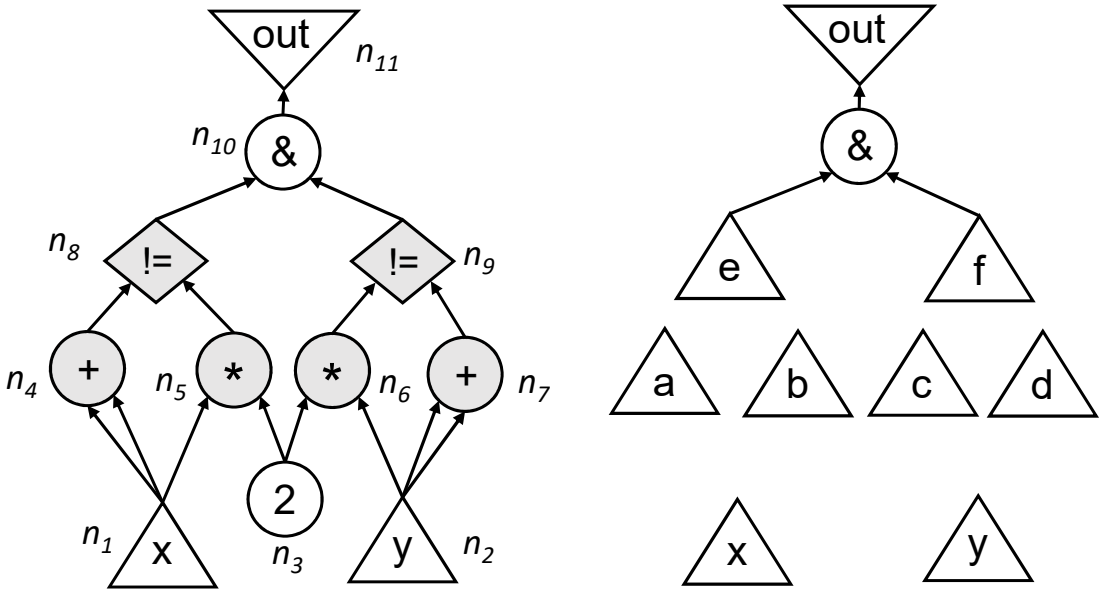
A CEX *cex* can thus be described using PI values.

$$cex = (i_0, i_1, \ldots, i_{k-1}) \tag{2.2}$$

The assignment function of a CEX can then be defined below.

Definition 2.1. Given a CEX $cex = (i_0, i_1, \ldots, i_{k-1})$, the *assignment function* β maps PIs (i) and a time stamp (t) to a concrete assignment. Formally,

$$\beta(i, t) = i_t. \tag{2.3}$$



(a) The original circuit with four arithmetic operators $\{n_4, n_5, n_6, n_7\}$, where x and y are primary inputs, 2 is a constant, `!=` is the complement of a comparator, & is a bit-wise AND, and *out* is the negation of the property.

(b) An abstraction created from the original circuit in (a) and the abstraction set $\mathcal{B} = \{n_4, n_5, n_6, n_7, n_8, n_9\}$. The 6 signals in $\mathcal{B}$ are replaced with 6 pseudo primary inputs (PPI), $\{a, b, c, d, e, f\}$.

Figure 2.1: A combinational circuit illustrating word-level abstraction. $out \equiv 0$, UNSAT, since $2 \times x \equiv x + x$, which forces *out* to be constant 0.

Given a property p and an FSM M, an assignment function β of a length-k CEX to the pair (M, p) would make the formula (2.4) satisfiable (SAT), meaning that there exists a trace from s_0 to s_k under PI assignments i_0 to i_{k-1} such that some state s_t falsifies the property.

$$Init(s_0) \wedge \bigwedge_{t=0}^{k-1} T(i_t, s_t, s_{t+1}) \wedge \bigvee_{t=0}^{k} \neg p(s_t) \wedge \bigwedge_{t=0}^{k-1} (i_t = \beta(i,t)) \tag{2.4}$$

Given a property p and two FSMs, M and A, where A is an abstraction of M, a *spurious* CEX and its assignment function β satisfy the following properties.

1. The formula below is SAT, meaning that the CEX falsifies the property in the abstraction A.

$$Init_A(s_0) \wedge \bigwedge_{t=0}^{k-1} T_A(i_t, s_t, s_{t+1}) \wedge \bigvee_{t=0}^{k} \neg p(s_t) \wedge \bigwedge_{t=0}^{k-1} (i_t = \beta(i,t)) \tag{2.5}$$

2. The formula below is UNSAT, meaning that the CEX cannot falsify the property in the original model M. Note that the domain of β includes both RPIs and PPIs, so the PIs in M (RPIs) are correctly mapped to their corresponding concrete values in the CEX.

$$Init_M(s_0) \wedge \bigwedge_{t=0}^{k-1} T_M(i_t, s_t, s_{t+1}) \wedge \bigvee_{t=0}^{k} \neg p(s_t) \wedge \bigwedge_{t=0}^{k-1} (i_t = \beta(i,t)) \tag{2.6}$$

2.2.3 Word-level CEGAR

Algorithm 2.2 shows a word-level extension to Algorithm 1.1 (CEGAR). The algorithm starts by abstracting *all* signals in the set $\mathcal{S}$ (e.g., outputs of all specified arithmetic operators). Next, an abstraction-refinement loop is entered where each iteration begins by creating a word-level abstraction based on the current set $\mathcal{B}$, the set of signals to be abstracted away. Procedure CreateAbstraction is presented in Algorithm 2.1. The abstraction is then bit-blasted and solved by a bit-level MC solver (e.g., PDR). If the solver returns UNSAT, the property is proved. Otherwise a CEX to the abstraction, *cex*, exists and is then *simulated* on the original circuit (W_M) to check if it is *real*. If yes, the property is falsified and *cex* is returned; otherwise *cex* is analyzed to derive a set of signals ($\Delta\mathcal{B}$) that, if un-abstracted, can block *cex*. A new abstraction, with the set $\Delta\mathcal{B}$ un-abstracted, is then created and a new iteration begins.

Algorithm 2.2 Word-level CEGAR

Input: W_M ▷ W_M: the word-level input circuit
Input: $\mathcal{S}$ ▷ $\mathcal{S}$: the initial set of targeted signals
Output: status ∈ { SAT, UNSAT }
1: $Iterations \leftarrow 1$
2: $\mathcal{B} \leftarrow \mathcal{S}$ ▷ $\mathcal{B}$: the set of abstracted signals
3: **while true do**
4: $W_A \leftarrow$ CreateAbstraction(W_M, $\mathcal{B}$)
5: $G_A \leftarrow$ BitBlast(W_A)
6: cex $\leftarrow$ ModelChecking(G_A)
7: **if** cex $\neq \emptyset$ **then**
8: **if** IsRealCEX(W_M, cex) **then**
9: **return** SAT
10: **else**
11: $\Delta\mathcal{B} \leftarrow$ Refine(W_M, G_A, $\mathcal{B}$, cex)
12: $\mathcal{B} \leftarrow \mathcal{B} \backslash \Delta\mathcal{B}$
13: $Iterations \leftarrow Iterations + 1$
14: **else**
15: **return** UNSAT

2.2.4 The Refinement Problem

Given a spurious CEX, *cex*, the goal of refinement is to identify a subset of signals $\Delta\mathcal{B}$ in the current set of abstraction signals $\mathcal{B}$, such that if $\Delta\mathcal{B}$ is removed from $\mathcal{B}$, then *cex* is *blocked* in the next iteration. We say that $\Delta\mathcal{B}$ is *un-abstracted*. This procedure corresponds to lines 11 and 12 in Algorithm 2.2.

A spurious CEX with assignment function β is said to be *blocked* in the refined abstraction N if the formula below is UNSAT, meaning that the same CEX cannot falsify the property in N. In practice, the formula can be checked without SAT solving by simulating the CEX in N and checking that the output $\neg p$ is constant-0 for the first k cycles.

$$Init_N(s_0) \wedge \bigwedge_{t=0}^{k-1} T_N(i_t, s_t, s_{t+1}) \wedge \bigvee_{t=0}^{k} \neg p(s_t) \wedge \bigwedge_{t=0}^{k-1} (i_t = \beta(i,t)) \tag{2.7}$$

Example 2.2. Consider the abstraction circuit shown in Figure 2.1b. Suppose the spurious CEX to this abstraction is found with PI assignments

$$(x, y, a, b, c, d, e, f) = (0, 0, 0, 0, 0, 0, 1, 1).$$

Suppose after some CEX analysis, a refined abstraction is created with PPIs = $\{a, b, e\}$ being un-abstracted, as shown in Figure 2.2. The spurious CEX is blocked in the refined circuit

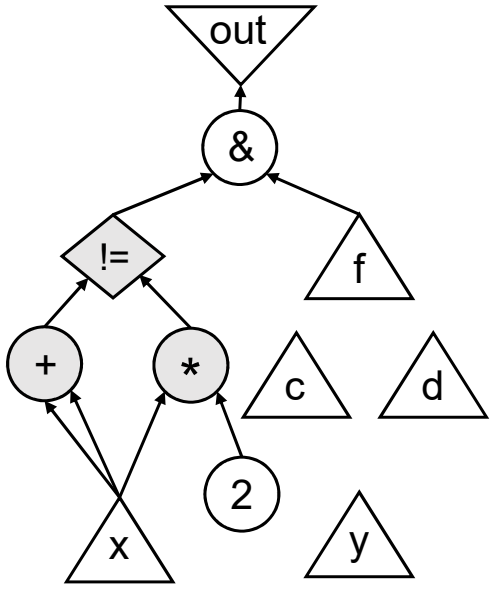


Figure 2.2: An example of refining the circuit in Figure 2.1b with $\Delta\mathcal{B} = \{n_4, n_5, n_8\}$, a subset of the set of the current abstraction signals $\mathcal{B} = \{n_4, n_5, n_6, n_7, n_8, n_9\}$. The refined circuit is created from the original circuit using the updated set $\mathcal{B} \setminus \Delta\mathcal{B} = \{n_6, n_7, n_9\}$, corresponding to PPIs $= \{c, d, f\}$.

because if we simulate the PI assignments below in the refined abstraction, the PO (*out*) value would be constant-0.

$$(x, y, c, d, f) = (0, 0, 0, 0, 1)$$

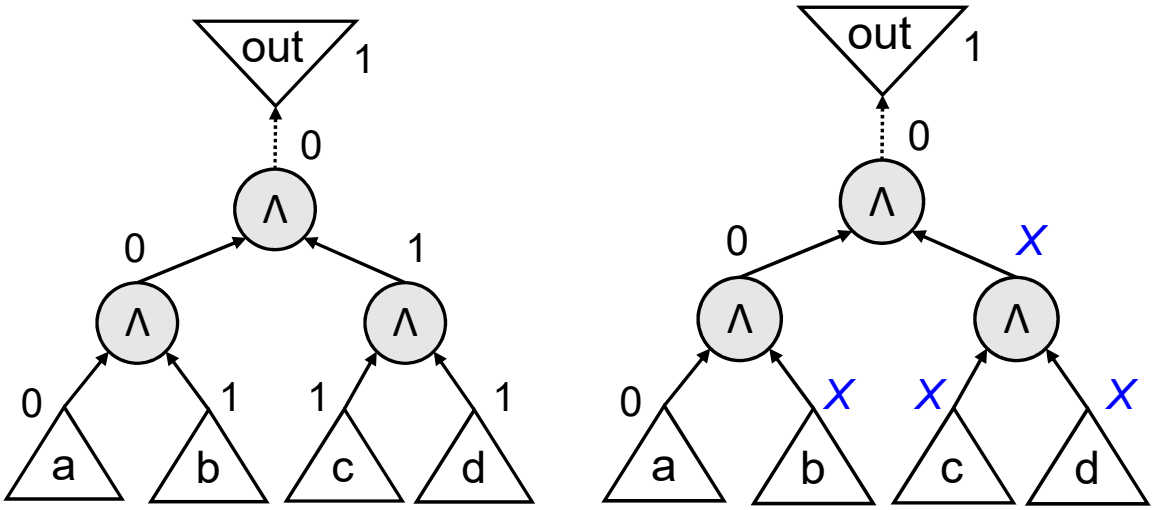
2.2.5 Ternary Simulation and X-value Counterexamples (XCEX)

A CEX can be *minimized* or *generalized* in the sense that some PIs can be assigned X values (unknown logic values), but the resulting X-value CEX (XCEX) can still falsify the property using *ternary simulation*.

In ternary simulation, Boolean logic with binary values $\{0, 1\}$ is extended to ternary logic with three values $\{0, 1, X\}$. The semantics of basic logic gates like AND ($\wedge$) and NOT ($\neg$) are given below.

$A \wedge B$		A		
		0	1	X
B	0	0	0	0
	1	0	1	X
	X	0	X	X

A	$\neg A$
0	1
1	0
X	X



(a) The original CEX with concrete-value PI assignments $(a, b, c, d) = (0, 1, 1, 1)$. The values of each gate derived using simulation are shown.

(b) An XCEX derived from the CEX in (a). PI assignments now contain X values with $(a, b, c, d) = (0, X, X, X)$. The values of each gate derived using ternary simulation are shown. The XCEX still falsifies the property (making $out = 1$).

Figure 2.3: An example showing a CEX can be minimized or generalized into an X-value CEX (XCEX). Symbol $\wedge$ denotes logic AND; dashed arrows represent logic NOT.

An XCEX is a generalized representation of a CEX because it represents a set of CEXes that can falsify the property. In later discussions, one of the refinement strategies is based on the idea of blocking a XCEX instead of a CEX, avoiding unnecessary CEGAR iterations.

Example 2.3. Figure 2.3 shows an example of how a CEX can be minimized into an XCEX. The original CEX has concrete values for the PIs $(a, b, c, d) = (0, 1, 1, 1)$. Three of the PIs $\{b, c, d\}$ can be assgined X values and the resulting XCEX $(a, b, c, d) = (0, X, X, X)$ still falsifies the property using ternary simulation. The XCEX now represents a set of 8 CEXes, including the original CEX:

$$\{(0,0,0,0), (0,0,0,1), (0,0,1,0), (0,0,1,1), (0,1,0,0), (0,1,0,1), (0,1,1,0), (0,1,1,1)\}.$$

2.2.6 Assumption Interfaces in SAT Solvers

Modern SAT solvers have assumption interfaces pioneered by Eén and Sörensson [ES03a, ES03b]. The interface is available through the following method.

$$\textbf{\textit{SolveSAT}}(\textit{assumptions}) \tag{2.8}$$

The assumptions are a set of unit clauses added to a SAT solver, which can be denoted as $assumptions = a_0 \wedge a_1 \wedge \ldots \wedge a_{n-1}$. The method (2.8) returns SAT if the SAT solver finds a satisfying assignment to the original problem under the given assumptions. Otherwise, the problem is UNSAT if the assumptions are given. The SAT solver would also return a subset of the assumptions $\{a_0, a_1, \ldots, a_{n-1}\}$ that is sufficient to make the problem UNSAT. This capability of SAT solvers has been shown useful and effective in MC algorithms like abstraction and PDR [EMA10, EMB11], since it provides a cheap and efficient way of estimating the minimum UNSAT subset (MUS) of the given assumptions.

Example 2.4. Consider the SAT instance below.

$$P = (a \vee c) \wedge (\neg b \vee c) \wedge (\neg a \vee b \vee c) \tag{2.9}$$

If we call ***SolveSAT***$(a \wedge \neg b \wedge \neg c)$, the SAT solver would return UNSAT since

$$a \wedge \neg b \wedge \neg c \wedge P \tag{2.10}$$

is UNSAT. A possible subset of assumptions $\{a, \neg b, \neg c\}$ that is sufficient for the UNSAT result is $\{\neg c\}$, since

$$\neg c \wedge P \tag{2.11}$$

is UNSAT. This means that we only need to assume $\neg c$ to make Query (2.9) UNSAT.

2.3 Simulation-based Refinement (SBR)

A simple refinement strategy is to simulate the given sprious CEX *cex* on the original circuit (W_M) and compare the PPI values (in *cex*) with their counterparts in W_M. If the values of a signal s do not match, then s is a refinement *candidate*, i.e. a candidate for un-abstraction. If *all* such candidates are un-abstracted, the property must hold; thus *cex* is blocked. However, this approach often results in too many candidates being un-abstracted, and thus is not a good strategy.

Example 2.5. Consider the circuits shown in Figure 2.1. Suppose a spurious CEX to the abstraction (Fig. 2.1b) is given below.

$$(x, y, a, b, c, d, e, f) = (0, 0, 1, 2, 1, 2, 1, 1).$$

(Recall that 2 here is the constant 2). If the CEX is simulated on the original circuit (Fig. 2.1a), then the values of the counterparts of PPIs are

$$(n_4, n_5, n_6, n_7, n_8, n_9) = (0, 0, 0, 0, 0, 0).$$

In this case, all the values of PPIs do not match with their counterparts, so all PPIs are refinement candidates. Un-abstracting all PPIs $\{a, b, c, d, e, f\}$ results in a refinement that can block the CEX, but it refines more PPIs than necessary. As we shall see later, an optimal refinement is to un-abstract PPIs $\{a, b, e\}$.

A more advanced way is to use a X-value CEX, discussed in Section 2.2.5 where some PIs are assigned to X (unknown logic value), while the XCEX still falsifies the property using ternary simulation.

An XCEX can be derived from a CEX in two ways:

1. For each PI value in the CEX, replace it with X, and then use ternary simulation to check if the resulting XCEX still falsifies the property. If so, then keep X for that PI. Otherwise, keep its original concrete value.

2. A more efficient way is to use *Priority-based Refinement* proposed by Mishchenko et al. [MEB+13]. The procedure traverses a circuit twice. The first traversal is from PIs to POs, assigning a priority for each node in a topological order. The second traversal goes from POs to PIs in a reverse topological order. Nodes traversed are based on their priorities and CEX values. Finally, a *justifying subset* (JS) of PPIs is returned. PPIs in the JS are the only PPIs that need to keep their concrete values. Other PPIs can be assigned X.

The concrete-value PPIs are also called *care-set* PPIs, since if their values are restricted to ones given in the CEX while other PPIs are assigned X, the property would still fail using ternary simulation. This provides a set of good candidates for refinement. If *all* PPIs in the care set are un-abstracted, then *cex* is very likely to be blocked. The refinement strategy of using *Priority-based Refinement* to identify a care set of PPIs to un-abstract, is called Simulation-based Refinement (SBR), because the resulting XCEX needs to falsify the property using ternary simulation.

SBR has the benefit of refining a more focused subset of PPIs than the one used in Example 2.5.

Example 2.6. Consider the same CEX and circuits shown in Example 2.5. The XCEX returned by Priority-based Refinement is

$$(x, y, a, b, c, d, e, f) = (X, X, X, X, X, X, 1, 1).$$

Therefore, the PPIs to un-abstract using SBR would be the care-set PPIs, $\{e, f\}$, instead of all PPIs as described in Example 2.5.

However, there are limitations in SBR. It is possible that each care-set PPI is fed by a tree, without overlaps with the trees of other care-set PPIs. Then, even if all the care-set

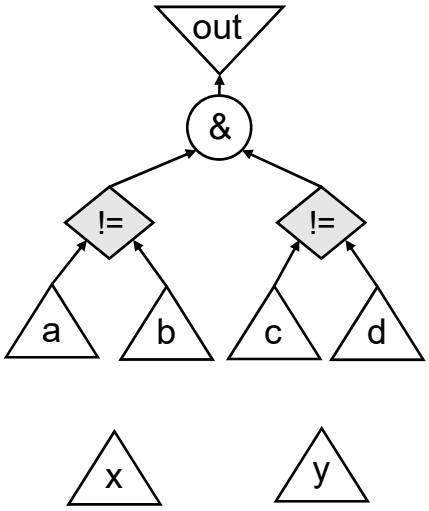


Figure 2.4: An example of refining the circuit in Figure 2.1b with $\Delta\mathcal{B} = \{n_8, n_9\}$ (PPIs $\{e, f\}$).

PPIs are un-abstracted, this will not provide enough constraints, and therefore the CEX is not blocked.

Example 2.7. Continuing Example 2.6, the refined circuit is shown in Figure 2.4. PPIs e and f are fed by trees that do not overlap. Their immediate inputs (a, b) and (c, d) are still PPIs, so the outputs of the refined e and f are still unconstrained. The CEX in Example 2.5 is not blocked in this refined circuit.

$$(x, y, a, b, c, d) = (0, 0, 1, 2, 1, 2)$$

Example 2.8. Consider the circuits shown in Figure 2.5. Suppose a spurious CEX to the abstraction (Fig. 2.5b) is given below.

$$(x, y, a, b, c, d, e, f) = (0, 0, 1, 2, 1, 2, 1, 1).$$

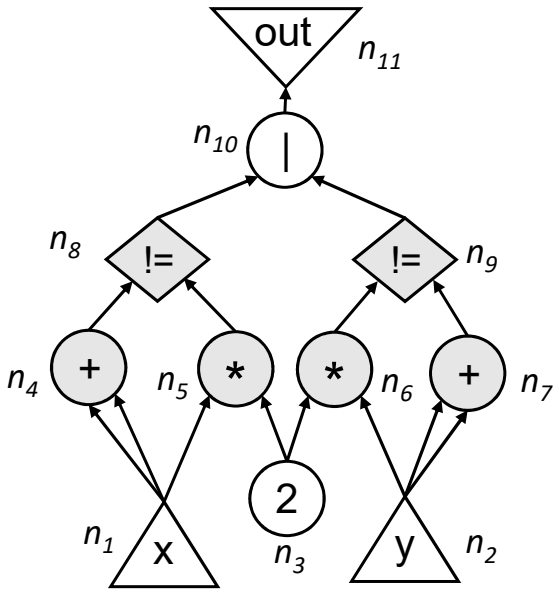
A minimized XCEX can be

$$(x, y, a, b, c, d, e, f) = (X, X, X, X, X, X, 1, X).$$

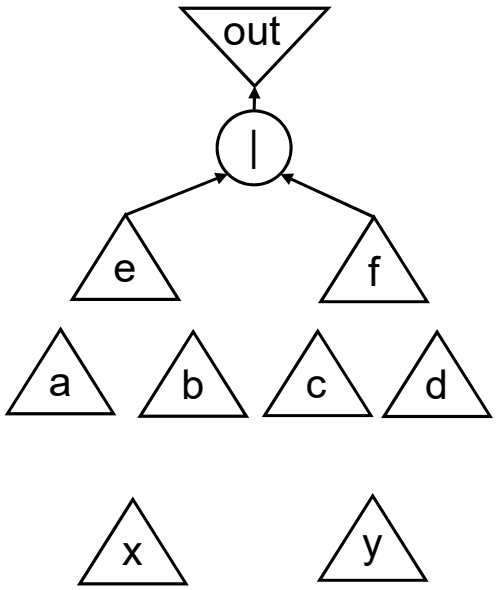
The only care-set signal is PPI $e = 1$, since it is sufficient to make $out = 1$ using ternary simulation. In this case, SBR un-abstracts PPI e and the refined abstraction is shown in Figure 2.6. Unfortunately, the same CEX is not blocked in this refinement, since the same PI assignments still falsify the property ($f = 1$ makes $out = 1$).

$$(x, y, a, b, c, d, f) = (0, 0, 1, 2, 1, 2, 1).$$

This shows another limitation of SBR.



(a) The original circuit with four arithmetic operators $\{n_4, n_5, n_6, n_7\}$, where x and y are PIs, 2 is a constant, `!=` is the complement of a comparator, | is a bit-wise OR, and *out* is the negation of the property.



(b) An abstraction created from the original circuit in (a) and the abstraction set $\mathcal{B} = \{n_4, n_5, n_6, n_7, n_8, n_9\}$. The 6 signals in $\mathcal{B}$ are replaced with 6 pseudo primary inputs (PPI), $\{a, b, c, d, e, f\}$.

Figure 2.5: A word-level abstraction example similar to the one in Figure 2.1; the node n_{10} is changed to an OR gate. $out \equiv 0$, UNSAT, since $2 \times x \equiv x + x$, which forces *out* to be constant 0.

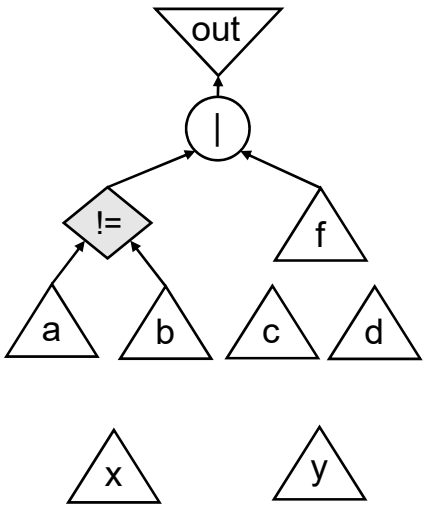


Figure 2.6: An example of refining the circuit in Figure 2.5b with $\Delta\mathcal{B} = \{n_8\}$ (PPI e).

2.4 Proof-based Refinement (PBR)

To address the limitations of SBR, a more effective refinement strategy, called Proof-based Refinement (PBR), is proposed. The main idea is that if a CEX *cex* is spurious and the original circuit (M) is simulated with *cex*, the property holds in all time frames. This implies that the BMC Formula (2.12) below is UNSAT, where i_t is the PI i at time t, s_t is the state variable at time t, k is the depth of *cex*, $\beta(\cdot)$ denotes the assignment function of *cex* (Definition 2.1), and *out* is the output signal ($out = 1$ means the property fails).

$$\begin{aligned} Init_M(s_0) \wedge \bigwedge_{t=0}^{k-1} T_M(i_t, s_t, s_{t+1}) \wedge \bigvee_{t=0}^{k} out(s_t) \\ \wedge \bigwedge_{t=0}^{k} (i_t = \beta(i,t)) \end{aligned} \tag{2.12}$$

Next, multiplexers are introduced to select between the concrete version and the abstracted version of a signal. If assumptions are made such that all concrete versions are selected initially, then the resulting BMC formula is still UNSAT and a modern SAT solver, such as MiniSat [ES03a], returns a final conflict clause. This contains a subset of the assumptions sufficient for UNSAT (Section 2.2.6). This is an efficient variation of finding an *unsat core*, and the subset returned is a candidate for $\Delta\mathcal{B}$.

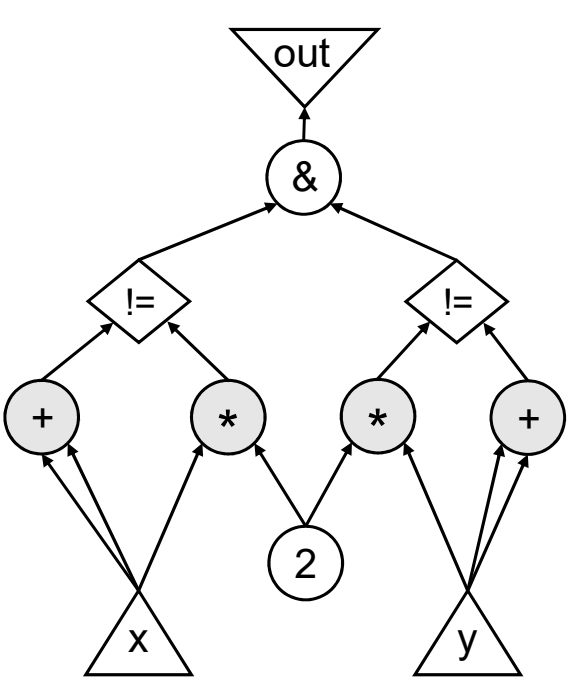
The procedure operates in four steps:

1. Starting with the original circuit (W_M), for each signal s in $\mathcal{B}$, introduce two new PIs, *sel* and *ppi*, where *sel* is a Boolean signal and *ppi* is a bit-vector signal *consistent*[1] with the signal s. Replace s with $s' = ITE(sel, s, ppi)$ where ITE is the *if-then-else* operator. Depending on the value of *sel*, either the concrete signal (s) or the abstracted one (*ppi*) becomes the new signal s'.

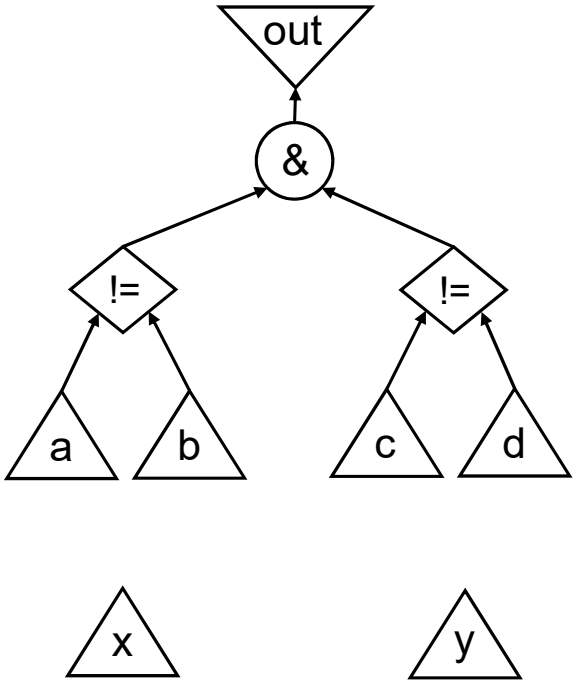
2. Denote the circuit created in Step 1 as N and unroll it with the values of *cex* plugged in, and keep *sel* and *ppi* as the remaining PIs. The *cex* values plugged in are initial states and PIs at each time frame.

3. Solve the BMC query (2.13) below, which is guaranteed to be UNSAT. Note that $\beta(\cdot)$ is the assignment function of *cex*, rpi_t is the real PIs (RPI) at time t, X_t is the set of *sel* inputs at time t, and x_{tn} is the *sel* input for the n-th replaced signal at time t. By propagating $x_{tn} = 1$ for all t and n, Query (2.13) is reduced to (2.12) by construction ($sel = 1$ means that the concrete version is chosen).

[1] Signals are consistent if they have the same widths and signedness.



(a) The original circuit with four arithmetic operators, where x and y are primary inputs, 2 is a constant, ! = is the complement of a comparator, & is a bit-wise AND, and *out* is the negation of the property.



(b) An abstraction derived from the original by replacing the 4 arithmetic operators with 4 new primary inputs, a, b, c, and d.

Figure 2.7: A combinational circuit illustrating word-level abstraction. $out \equiv 0$, UNSAT, since $2 \times x \equiv x + x$, which forces *out* to be constant 0.

$$Init_N(s_0) \wedge \bigwedge_{t=0}^{k-1} T_N(i_t, s_t, s_{t+1}) \wedge \bigvee_{t=0}^{k} out(s_t) \\ \wedge \bigwedge_{t=0}^{k} (rpi_t = \beta(rpi, t)) \wedge \bigwedge_{t=0}^{k} \bigwedge_{n=1}^{|X_t|} x_{tn} \tag{2.13}$$

4. Derive a subset ΔX of X using the assumption interface of a modern SAT solver, and determine $\Delta\mathcal{B}$ from ΔX. In our implementation, there is only one free variable x_n associated with the replaced signal, i.e. $x_n \equiv x_{0n} \equiv x_{1n} \equiv \ldots \equiv x_{kn}$ for $1 \leq n \leq |\mathcal{B}|$. This way, we have $|\mathcal{B}|$ assumptions (instead of $(k+1)|\mathcal{B}|$) and the returned ΔX is exactly our candidate for $\Delta\mathcal{B}$.

Example 2.9. Consider the circuits in Figure 2.7. Suppose a CEX to the abstraction

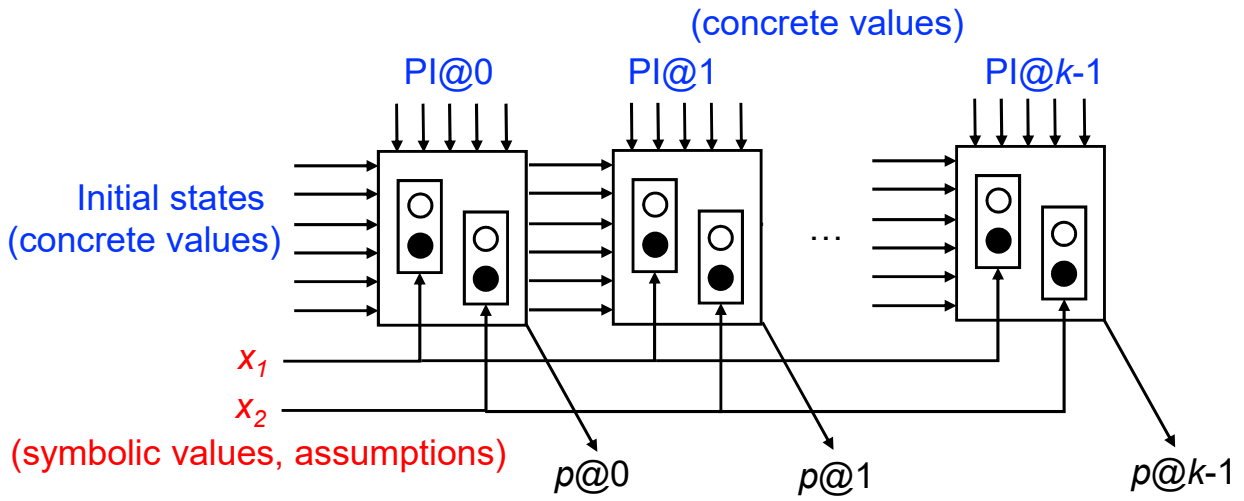


Figure 2.8: An example of unrolling a circuit in PBR. ITE operators (multiplexers) are introduced to select the concrete signals (white circles) and the abstracted ones (black circles). If all concrete signals are chosen, then the unrolling becomes the same as the k-unrolling of the original circuit, where the property holds under the spurious CEX.

(Fig. 2.7b) is obtained, where the assignments of PIs and PPIs are

$$(x, y, a, b, c, d) = (0, 0, 0, 1, 0, 1).$$

Circuit (N), derived by introducing ITEs for each PPI, is shown in Figure 2.9. If all *sel* PIs $\{s_1, s_2, s_3, s_4\}$ are 1, then the circuit is reduced to the original. Next, PI values ($x = 0$, $y = 0$) are plugged in, and PPIs $\{a, b, c, d\}$ are left unconstrained. The SAT solver is called to determine if *out* can be 1. The result must be UNSAT with the assumptions of the *sel* PIs being all 1. In this case, the subset returned would be either $\{s_1, s_2\}$ or $\{s_3, s_4\}$, which is the *minimum* set needed. This example demonstrates that PBR can pinpoint a precise set for refinement while a simulation-based approach only gives a rough approximation.

Example 2.10. Consider the unrolled circuit shown in Figure 2.8. The concrete values (constants) used in the unrolling are the initial states and PI assignments from the given CEX. The symbolic values, or non-constant PIs, are the *sel* and *ppi* PIs introduced with ITE operators (multiplexers). In this example, there are two signals in the current abstraction ($|\mathcal{B}| = 2$). Therefore, two assumptions $\{x_1, x_2\}$ can be made such that both original signals (white circles) are selected and the resulting query (2.13) is UNSAT. The SAT solver then reports a subset of $\{x_1, x_2\}$ that is sufficient to block the CEX in the next iteration.

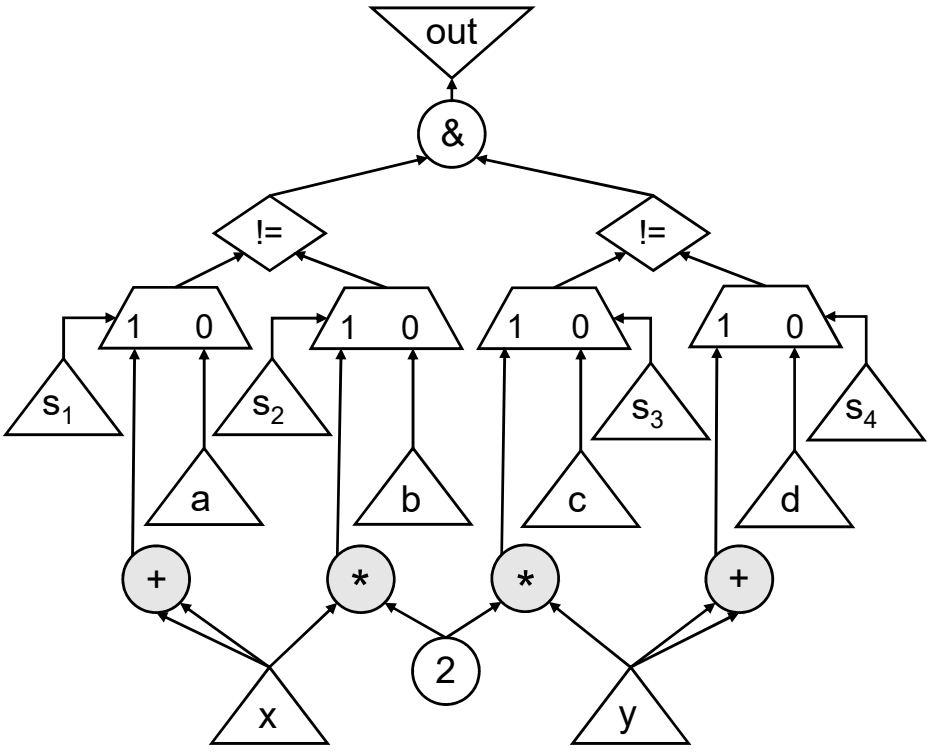


Figure 2.9: Example for proof-based refinement, where x and y are original PIs, a-d are pseudo PIs, s_1-s_4 are *sel* PIs. This is created from the current abstraction shown in Figure 2.7b. If the assignments of x and y in *cex* are plugged in, and assumptions are made that s_1-s_4 are all 1, then *out* is constant-0 (UNSAT).

2.4.1 Variants of Proof-based Refinement

Two additional proof-based refinement strategies, PBR-A and PBR-B, are presented compared with SBR (Sec. 2.3) and PBR (Sec. 2.4).

Given a spurious CEX, *cex*, there are at least two more ways to formulate an UNSAT query that can be used for proof-based refinements. $\beta(\cdot)$ is the assignment function of *cex*.

PBR-A. This considers Formula (2.14) below. The idea is that if the values in *cex* are plugged into the *abstraction* T_A, then *out* must be 1 at some time frame t. Therefore, the formula asserting that *out* is 0 for all time frames, with *cex* plugged in, must be UNSAT. One can then compute the subset of PPIs sufficient for UNSAT, deriving a refinement. Note that PBR-A does not use the information of the original circuit and can be considered as a proof-based version of SBR.

$$\begin{aligned} Init_A(s_0) \wedge \bigwedge_{t=0}^{k-1} T_A(i_t, s_t, s_{t+1}) \wedge \bigwedge_{t=0}^{k} \neg out(s_t) \\ \wedge \bigwedge_{t=0}^{k} (i_t = \beta(i,t)) \end{aligned} \tag{2.14}$$

PBR-B. This uses Formula (2.15) below. Let rpi_t and ppi_t be the original PIs and the PPIs at time t, respectively. Similar to PBR (Formula 2.13), it takes the original circuit into account by introducing MUXes selecting between PPIs and the original signals, creating a circuit N. The only difference with PBR is that PBR-B also plugs in the assignments of the PPIs in *cex* into the formula. Otherwise it proceeds like PBR using the assumption interface to derive a candidate for $\Delta\mathcal{B}$.

$$\begin{aligned} Init_N(s_0) \wedge \bigwedge_{t=0}^{k-1} T_N(i_t, s_t, s_{t+1}) \wedge \bigvee_{t=0}^{k} out(s_t) \\ \wedge \bigwedge_{t=0}^{k} (rpi_t = \beta(rpi, t) \wedge ppi_t = \beta(ppi, t)) \wedge \bigwedge_{t=0}^{k} \bigwedge_{n=1}^{|X_t|} x_{tn} \end{aligned} \tag{2.15}$$

2.5 Maximum Fan-out Free Cone (MFFC) Refinement

We observed that in many cases, the signals in the fanin cones of those candidate signals would appear in the next iteration of refinement, implying that an additional *structural analysis* can improve the speed of convergence further.

The main idea is to use the *maximum fanout free cones* (MFFC) of those candidate signals. The MFFC of a signal s is a subset of its fanin cone, where each path from a signal in the MFFC to the POs passes through s, i.e. the MFFC of a signal contains all the logic used exclusively by the signal. If a signal is abstracted, its MFFC would be abstracted. However, if a signal is un-abstracted, its MFFC is better un-abstracted also; otherwise, additional iterations may be needed.

In our experience, un-abstracting all candidate signals as well as all those in their MFFCs often converges faster, i.e. reaching a final abstracion after fewer iterations. We note that this MFFC strategy is complementary to any of the refinement strategies like SBR and PBR. SBR and PBR find a set of candidate signals first, and then MFFCs of those signals can be added to the refinement set $\Delta\mathcal{B}$.

Example 2.11. Consider the original circuit shown in Figure 2.1a and a given spurious CEX:

$$(x, y, a, b, c, d, e, f) = (0, 0, 1, 2, 1, 2, 1, 1).$$

Using SBR, the care-set PPIs returned are $\{e, f\}$ as shown in Example 2.6. However, as discussed in Example 2.7, the CEX is not blocked in the refined circuit shown in Figure 2.4. This problem can be addressed by using MFFC refinement. In this example, PPIs $\{a, b\}$ and $\{c, d\}$ are in the MFFCs of PPI e and PPI f, respectively. Therefore, instead of refining PPIs $\{e, f\}$, we can refine the PPIs in their MFFCs also. The resulting PPIs to un-abstract are then $\{a, b, c, d, e, f\}$. The refined circuit is thus the original circuit and does block the CEX. In this case, MFFC saves one unnecessary iteration in CEGAR.

2.6 Comparison of Refinement Strategies

While SBR is good enough in many applications [MEB+13, FYH16], frequently it does not find a minimal set to un-abstract.

Example 2.12. Consider the original circuit and its abstraction in Figure 2.7. Suppose a CEX to the abstraction is found (Fig. 2.7b), where the assignments of PIs and PPIs are

$$(x, y, a, b, c, d) = (0, 0, 0, 1, 0, 1).$$

For this example, the care set C returned by counterexample minimization would be all PPIs, $C = \{a, b, c, d\}$. If any PPI is assigned an X, the PO would become X as well; thus all PPIs must be in the care set. However, it is clear that the set is not *minimum* because only $\{a, b\}$ or $\{c, d\}$ needs to be un-abstracted to block the CEX. In fact, if we un-abstract $\{a, b\}$ (or $\{c, d\}$), the property can be proved with the refined abstraction.

The four refinement strategies (SBR, PBR, PBR-A, PBR-B) are compared using the two examples below.

Example 2.13. Consider the circuits in Figure 2.7. Suppose a CEX is obtained with the assignments of PIs and PPIs as

$$(x, y, a, b, c, d) = (0, 0, 0, 1, 0, 1).$$

SBR and PBR-A would refine all PPIs $\{a, b, c, d\}$. PBR-B and PBR would refine only either $\{a, b\}$ or $\{c, d\}$ to obtain a final abstraction. This shows that PBR can get a smaller final abstraction by refining fewer PPIs compared to using SBR and PBR-A.

Example 2.14. Consider slightly different circuits from those in Figure 2.7: the AND gates (&) are now replaced by OR gates (|) in both the original circuit and its abstraction. Suppose a CEX is obtained with the assignments of PIs and PPIs as

$$(x, y, a, b, c, d) = (0, 0, 0, 1, 0, 0).$$

SBR, PBR-A, and PBR-B all would refine $\{a, b\}$, which is not a final abstraction, requiring another iteration. PBR would refine all PPIs $\{a, b, c, d\}$, which is a final abstraction. This shows that PBR is able to converge with less iterations than the other three. The insight is that PBR refutes *all* spurious CEXes under the same assignments of original PIs in *cex*, while the others only refute CEXes with the same values of *both* PIs and PPIs.

2.7 Related Work

In Gate-Level Abstraction (GLA), Mishchenko et al. proposed Priority-based abstraction refinement [MEB+13], which works very well in their bit-level CEGAR algorithm. Our formulated Simulation-based Refinement (SBR) in Section 2.3 demonstrates that their bit-level procedure can be nicely extended to work at the word level. SBR analyzes the given spurious CEX in the *abstraction* without using information from the original circuit. As a result, SBR finds very different sets of refinement candidates from PBR, which is discussed in Section 2.6.

In ATLAS and CAL, the authors proposed ways of refinement in their word-level *Term-level Abstraction* [BBSO10, BBS11], which features uninterpreted function (UF) abstraction. The idea is to synthesize control signals such that if a control signal is True, then a

fully-interpreted function is used; otherwise, an uninterpreted function (UF) is used. The refinement strategies discussed in this chapter are simple localization abstraction, where abstraction is used to replace signals with PPIs while refinement is used to replace PPIs with their original signals. We propose an approach to UF abstraction refinement in Chapter 5.

The closest work to ours is REVEAL [ALS06], a word-level CEGAR-based solver where several proposed refinement strategies are used.

1. Their first strategy is *localization*, where a cone-of-influence (COI) analysis is used to remove irrelevant assignments in a CEX. This is handled automatically in SBR discussed in this chapter, since assignments not in the COI of the property would be assigned X values.

2. Their second strategy is *generalization*. In their formulation, a generalized CEX must fail the property in the current abstraction. This again is very similar to SBR in the sense that the generalization does not consider any information from the original circuit; it only tries to enlarge a CEX in the current abstraction, which has inherent limitations, as discussed in Section 2.6.

3. Their third strategy is *Minimal Unsatisfiable Subset (MUS) extraction*. The idea is similar to PBR-A with Query (2.14). Since the query is UNSAT, an MUS of constraints can be extracted as another generalized CEX.

4. In their last strategy, they combine the original model and the generalized CEX derived from the previous three strategies to construct an UNSAT query similar to Query (2.15) in PBR-B. Since the query is UNSAT, an MUS of the constraints of the original model can be extracted. Those constraints from the original model are then refined in the next abstraction.

On the other hand, this chapter presents MFFC refinement, which is entirely new. The PBR formulation proposed in Section 2.4 takes advantage of smart circuit transformations and assumption interfaces in SAT solvers, which is also new. The assumption interfaces provide good estimates of MUS with only little overhead [ES03a].

2.8 Experimental Results

Experiments were done to evaluate the refinement strategies discussed in this chapter. We implemented Algorithm 2.2 in the public verification tool, ABC [BM10]. Our tool can parse word-level Verilog and transform the resulting design into a bit-level circuit by bit-blasting.

The benchmarks used for evaluation were a set of 195 industrial Verilog RTL designs.

Table 2.1: Detailed experimental results for the first 45 out of the 89 word-level test-cases that can be solved by at least one of the six refinement strategies (the last 44 are shown in the next table). $|S|$ and $|B|$ are sizes of the set of the initial targeted signals ($\mathcal{S}$) and the set of signals to be abstracted away for each iteration ($\mathcal{B}$) in Algorithm 2.2.

ID	|S|	CPU Time (seconds)						Iterations						|B| in the last iteration					
		SBR (S1)	PBRB (S2)	PBR (S3)	SBR-MFFC (S4)	PBRB-MFFC (S5)	PBR-MFFC (S6)	S1	S2	S3	S4	S5	S6	S1	S2	S3	S4	S5	S6
1	101	7.27	7.68	8.43	**7.17**	8.05	9.22	2	2	2	2	2	2	66	83	86	48	82	86
2	101	10.89	51.39	13.04	**10.35**	44.56	12.78	2	3	2	2	3	2	68	76	80	51	75	79
3	101	1.32	1.36	1.39	1.32	**1.3**	1.36	2	2	2	2	2	2	68	99	91	54	99	91
4	101	**1.26**	1.28	1.46	1.28	1.4	1.51	2	2	2	2	2	2	67	95	91	53	95	91
5	100	105.55		227.46	**86.75**	418.88	215.82	5	45	24	3	43	22	30	39	29	26	27	21
6	101	**2023.24**	2284.77	2223.96	2165.26	2280.97	2056.82	14	4	3	3	3	3	0	0	0	0	0	0
7	102	3496.41	3541.65	3444.83		**3440.69**	3495.45	18	4	3	7	3	3	18	18	18	18	18	18
8	101	2393.77	2480.27	**2217.38**	2450.32	2497.89	2428.66	14	4	4	2	2	2	0	0	0	0	0	0
9	101	1929.49	1859.44	1845.07	**1673.05**	1801.61	1905	15	5	3	2	3	2	0	0	0	0	0	0
10	101	1974.66	2054.12	2103.06	1922.37	1865.55	**1811.23**	11	3	2	2	2	2	18	18	18	18	18	18
11	100	153.6	**129.69**	139.08	159.19	130.97	137.44	2	3	3	2	3	3	88	98	97	88	98	97
12	100	3457.89		**2612.71**	3481.93		2678.15	4	9	5	4	8	5	79	83	75	79	86	75
13	100	1788.64		**978.63**	1797.17		982.13	5	11	5	5	11	5	78	86	79	78	86	79
14	100	650.29	343.78	**265.14**	659.66	355.25	279.36	2	5	3	2	5	3	87	93	94	87	93	94
15	100			2842.95			**2810.39**	4	7	5	4	7	5	82	82	78	82	82	78
16	100	717.44	612.59	375.79	700.35	617.71	**368.44**	2	4	3	2	4	3	87	92	90	87	92	90
17	100	3295.02		**2341.31**	3287.12		2373.59	4	8	5	4	8	5	78	84	78	78	84	78
18	100	2678.2		2597.72	2725.71		**2580.75**	5	5	7	5	5	7	78	87	78	78	87	78
19	100	1171.53	2628.9	1521.6	**1104.93**	3008.29	1510.93	4	12	6	4	12	6	78	83	78	78	83	78
20	100			**2785.81**			2848.13	4	7	5	4	7	5	80	84	79	80	84	79
21	100	2999.94		**2171.42**	3035.92		2259.94	5	8	5	5	7	5	79	87	78	79	88	78
22	100	354.39	356.33	**261.62**	354.87	363.59	262.72	2	3	3	2	3	3	90	98	97	90	98	97
23	100			2329.17			**2319.51**	4	6	5	4	6	5	79	89	78	79	89	78
24	100	1928.19	3253.66	1663.75	**1272.04**	3514.02	2530.2	4	13	6	4	13	6	78	83	78	78	83	78
25	100			**2210.4**			2214.2	5	8	6	5	8	6	77	88	78	75	88	78
26	101					277.15	**8.85**	6	20	10	4	16	8	26	26	25	11	19	30
27	101					281.96	**10.83**	7	22	10	4	17	8	16	26	25	11	19	30
28	101					342.93	**13.4**	7	20	10	4	16	8	16	28	28	11	21	30
29	101					159.03	**12.91**	7	22	11	4	16	8	23	27	27	15	21	30
30	101					398.53	**10.27**	6	20	10	4	17	8	28	25	25	11	19	30
31	101					369.41	**13.65**	7	20	10	4	18	9	17	26	25	11	19	30
32	100	132.37	171.24	134.94	**124.51**	137.14	180.88	12	19	9	3	9	9	57	70	54	56	51	36
33	107			**2808.62**		3552.65		8	17	13	6	17	13	63	73	41	45	53	36
34	102		3151.35	**906.52**				9	17	8	6	14	9	63	64	64	35	48	41
35	102	2250.69	1338.1	782.35	1522.3	992.8	**759.08**	13	20	14	8	19	15	32	69	69	19	52	54
36	130	**1028.51**	1244.23	1333.5	1938.27	1234.09	1291.82	7	39	25	3	36	24	68	75	51	57	53	51
37	130	736.04	**705.99**	841.08	1083.67	818.27	812.73	7	42	23	3	30	23	70	61	71	71	59	71
38	82				**631.87**			15	6	3	3	3	2	0	0	0	12	0	0
39	82				**397.73**			15	7	4	3	3	2	0	0	0	12	0	0
40	58	172.2	482.55	422.09	203.36	**171.93**	178.38	14	22	5	3	5	2	12	10	13	13	0	13
41	58	156.1	369.84	322.78	173.97	168.64	**136.12**	17	17	6	3	4	2	13	12	12	0	0	0
42	61	487.89	500.58	439.97	147.94	220.88	**142.59**	15	22	5	3	4	2	13	11	14	13	0	13
43	64	407.25	310.82	283.87	119.19	121.55	**106.63**	18	18	6	3	4	2	15	17	17	0	0	0
44	72	340.63	336.1	**294.28**	312.5	295.37	327.78	15	19	7	6	12	5	34	33	31	30	28	32
45	108				**602.13**			3	5	8	3	5	6	70	100	95	38	100	98

Table 2.2: Detailed experimental results for the last 44 out of 89 word-level test-cases that can be solved by at least one of the six refinement strategies. $|S|$ and $|B|$ are sizes of the set of the initial targeted signals ($\mathcal{S}$) and the set of signals to be abstracted away for each iteration ($\mathcal{B}$) in Algorithm 2.2.

ID	|S|	CPU Time (seconds)						Iterations						|B| in the last iteration					
		SBR (S1)	PBRB (S2)	PBR (S3)	SBR-MFFC (S4)	PBRB-MFFC (S5)	PBR-MFFC (S6)	S1	S2	S3	S4	S5	S6	S1	S2	S3	S4	S5	S6
46	107				**1341.35**			3	6	5	3	5	5	65	102	100	29	98	97
47	72	**77.84**	81.63	100.14	123.25	118.77	107.21	21	33	13	12	15	9	19	21	16	18	18	15
48	72	100.21	89.08	95.8	93.24	136.8	**69.02**	19	27	12	12	15	8	14	17	14	17	17	13
49	72	83.26	157.86	**75.05**	124.16	101.62	104.38	19	30	8	9	13	9	14	17	14	16	16	13
50	72	83.36	89.43	124.3	81.39	97.02	**71.44**	19	33	12	11	14	10	17	19	16	18	15	19
51	43	102.5	80.64	79.62	33.15	32.98	**32.03**	18	18	6	3	4	2	0	0	0	0	0	0
52	57	1339.87	2933.42	1538.02	**1103.25**	1120.24	1156.2	12	25	4	3	4	2	12	10	13	13	0	13
53	43	106.13	87.98	82.72	**33.84**	37.1	37.63	18	18	6	3	4	2	0	0	0	0	0	0
54	58	1032.16	3128.21	3373.36	**1022.92**	1127.69	1121.2	12	25	5	3	4	2	12	10	13	13	13	13
55	123	2053.23		2886.73	425.61		**425.4**	10	48	32	5	45	33	26	45	36	22	37	29
56	109	883.17	1617.54	1207.3	**141.44**	462.93	520.23	10	37	27	5	32	28	25	52	49	21	45	36
57	110	567.06	1596.7	1419.74	**240.54**	305.12	2628.35	9	38	30	5	33	28	26	52	39	21	46	30
58	112	613.87	1262.28	2557.01	**188.9**	755.17	249.38	9	38	26	5	34	23	27	53	40	21	45	44
59	129	1439.7			1295.29	**655.41**	1377.74	9	55	31	4	43	31	33	44	36	28	38	38
60	118	2160.43	1848.54	1484.31	**602.48**		3232.79	9	36	23	5	33	23	28	50	43	17	42	32
61	111	842.97	1307.2	1627.09	**122.79**	507.26	865.06	9	37	28	4	32	28	27	55	39	22	48	30
62	103	2125.66			**629.68**	1157.13	1255.68	6	23	16	3	18	12	69	76	76	76	77	77
63	100			3215.1	**699.37**	2887.95	3057.55	14	40	34	3	34	33	29	10	12	13	21	4
64	108	362.94	**266.24**	342.67	345.52	307.01	374.27	14	45	25	7	28	17	0	3	0	0	3	0
65	94	55.97	73.65	43.75	**41.53**	58.2	53.43	13	25	9	5	11	6	25	33	28	15	25	13
66	106	**1276.9**			3246.63	3207.94	3504.72	5	18	10	5	17	9	63	67	71	43	47	50
67	140	**1525.91**	1741.04	1679.15	1708.12	1596.07	1885.51	2	4	5	2	4	5	92	98	92	78	98	78
68	100	228.75	228.39	**215.45**	233.06	227.47	218.5	1	1	1	1	1	1	100	100	100	100	100	100
69	140	**1545.84**	1601.52	1748.93	1763.27	1617.1	1920.46	2	4	5	2	4	5	92	98	92	78	98	78
70	100	**220.95**	223.3	233.52	251.58	239.88	236.43	1	1	1	1	1	1	100	100	100	100	100	100
71	104	128.75	291.43	162.47	**39.12**	247.9	159.88	11	23	13	4	19	15	51	50	59	57	59	38
72	140	**1410.71**	1553.44	1532.42	1600.27	1612.86	1687.66	2	4	5	2	4	5	92	98	92	78	98	78
73	100	221.16	**210.56**	220.15	246.97	248.86	252.64	1	1	1	1	1	1	100	100	100	100	100	100
74	105	228.36	512.63	307.79	**70.4**	217.74	203.87	12	27	17	4	14	12	53	49	47	49	51	37
75	100	163.2	177.38	183.87	229.68	160.88	**87.24**	6	19	10	4	11	8	57	74	68	56	72	72
76	140	**1487.07**	1668.42	1541.78	1609.97	1689.67	1842.31	2	4	5	2	4	5	92	98	92	78	98	78
77	100	64.41	118.3	48.64	51.37	115.19	**47.4**	5	10	4	3	5	4	77	86	81	75	86	81
78	100	**299.21**	349.83	340.55	442.9	453.5	313.88	9	15	11	4	11	9	64	68	60	49	69	62
79	101	80.39	199.21	178.62	**36.98**	147.7	162.67	5	11	10	3	8	9	80	86	82	70	86	74
80	105	37.52	155.76	76.7	**25.28**	79.3	82.02	4	13	8	3	8	7	89	88	87	77	92	78
81	100	248.21	228.73	245.43	254.9	**201.2**	254.17	1	1	1	1	1	1	100	100	100	100	100	100
82	140	**1496.98**	1755.59	1656.71	1754.7	1734.71	1726.49	2	4	5	2	4	5	92	98	92	78	98	78
83	113	57.26	186.61	122.53	**28.93**	111.34	114.2	6	18	10	3	11	10	68	84	78	75	90	51
84	100	146.29	187.66	144.14	124.05	279.21	**112.22**	4	7	3	2	3	3	84	87	86	72	90	78
85	101	54.62	110.39	139.6	**42.06**	86.14	103.02	4	10	10	3	8	9	82	87	81	72	88	75
86	100	238.95	227.19	231.08	227.94	225.7	**224.31**	1	1	1	1	1	1	100	100	100	100	100	100
87	100	219.98	**212.96**	213.5	213.89	214.92	214.34	1	1	1	1	1	1	100	100	100	100	100	100
88	140	**1341.78**	1514.02	1655.51	1599.15	1580	1894.38	2	4	5	2	4	5	92	98	92	78	98	78
89	100	160.54	197.15	**148.7**	211.29	252.71	150.99	7	18	9	5	11	9	68	74	76	54	77	76

Table 2.3: Summary of Table 2.1 and Table 2.2 in terms of the number of test cases solved.

SBR	PBRB	PBR	SBR-MFFC	PBRB-MFFC	PBR-MFFC
72	63	76	76	75	**83**

Large arithmetic operators and multiplexers were the signals targeted for possible abstraction (set $\mathcal{S}$). A workstation with Intel Xeon E5504 CPUs clocked at 2.0 GHz with 24 GB of RAM was used. A time-out of 3600 seconds was used on all experiments.

Table 2.1 and Table 2.2 show the 89 (of the 195 designs), which can be solved by at least one of the six settings: 1) SBR, 2) PBR-B, 3) PBR, and 1)-3) with MFFC. All were proved UNSAT. The tables give an idea of details such as CPU time, expected ranges of iterations needed, and the sizes of $\mathcal{B}$ (signals to be abstracted way) in the final abstractions. All test cases are UNSAT.

The results show that there is no one refinement strategy that is always better than the others in terms of CPU time. However, the proposed PBR-MFFC (S6) performs the best in terms of the number of cases solved within the 3600-sec timeout, which is given in Table 2.3.

Other observations from Table 2.1 and Table 2.2 are given below.

1. **SBR (S1) vs. PBR (S3)**. PBR uses more iterations and derives smaller final abstractions (large $|\mathcal{B}|$) in most cases, implying that PBR leads to more fine-grained and focused refinements in most cases.

2. **PBR-B (S2) vs. PBR (S3)**. In most cases, PBR uses less iterations to find a final abstraction, while PBR-B takes more iterations, which can be avoided by a proper analysis (see Example 2.14). PBR-B can derive a small final abstraction (large $|\mathcal{B}|$), but large numbers of iterations can cause poor performance. Note: comparison with PBR-A was not done due to its similarity to SBR.

3. **Without MFFC (S1, S2, S3) vs. with MFFC (S4, S5, S6)**. MFFC can be crucial in preventing unnecessary refinement iterations, which is illustrated in Example 2.11. This is critical in cases like ID 26 - 31, where PBR-MFFC easily proves the property with less iterations than PBR does.

2.9 Conclusion

This chapter describes several refinement strategies. SBR generalizes a CEX into an XCEX that represents a set of CEXes needs to be blocked. One weakness of SBR is that the

CEX (or XCEX) is not guaranteed to be blocked in the refined circuit, since SBR does not use any information from the original circuit. PBR was developed to address the challenges in SBR. The idea is to encode assumptions into a circuit with *ITE* operators (or multiplexers). PBR takes advantage of both the information in the original circuit and the assumption interfaces in SAT solvers, which result in an efficient procedure with the guarantee that the current CEX would be blocked after refinement. Moreover, MFFC refinement was developed to exploit the circuit structure that can save unnecessary iterations in CEGAR. The strategies discussed in this chapter were implemented in the public model checker ABC and evaluated on a set of industrial benchmarks. The experimental results show that PBR with MFFC solved the most cases compared with other settings.

Chapter 3

Enhancing PDR with Localization Abstraction

With good refinement strategies presented in Chapter 2, our next challenge is to integrate MC algorithms into the CEGAR flow in an efficient way. This chapter presents an extensive summary of the PDR algorithm and goes on to present a version with abstraction, called PDRA. This is a bit-level MC algorithm that is minimally modified from the original PDR to perform on-the-fly abstraction and refinement. A word-level version of this, called PDR-WLA, is presented in Chapter 4, which will be a significant part of UFAR, a word level model checker, presented in Chapter 5.

3.1 Introduction

Property Directed Reachability (PDR) is an elegant and powerful engine pioneered in 2010 by Aaron Bradley under the name of IC3 [Bra11] and improved by ongoing research [EMB11, BIMM12, HBS13, IG15, GR16]. The engine continues to receive attention because of its ability to solve hard model checking problems, both satisfiable and unsatisfiable. The inductive invariants computed as a by-product of solving unclassifiable verification instances with PDR, are useful as certificates of correctness of unsatisfiability and as a means for design analysis. For example, the support of an invariant indicates what parts of the design are needed to prove the property.

Localization abstraction [WJK+01, EMA10, MEB+13] is a method aimed at reducing the complexity of a verification instance by removing some logic. The remaining part of the instance is called an abstraction. An abstraction typically contains the property output

of the original instance along with logic nodes and flip-flops deemed necessary to prove the property. The connections to the removed logic are called pseudo primary inputs (PPIs) and are treated as free variables, which increases the behavior, i.e. the abstractions had more satisfying assignments. As a result, if the abstraction is proved UNSAT, the verification problem is solved. If a counterexample (CEX) is discovered, abstraction refinement is used to add new logic to rule out the cex, before a new proof is attempted. A taxonomy of abstraction methods can be found in [MEB+12].

The contribution of this chapter is integrating PDR with an adaptive localization abstraction. As a result, the PDR engine is minimally modified to perform on-the-fly abstraction while solving a verification instance. The modified PDR engine is capable of solving more problem instances than the original PDR engine. Moreover, inductive invariants computed by the modified PDR are on average about 20% smaller than those computed by the original PDR. A smaller invariant is more representative of the verification problem and more suitable for design analysis. Moreover, the PDR engine is complementary to other PDR improvements like [HBS13] and can be easily integrated.

The chapter is organized as follows. Section 3.2 contains relevant background. Section 3.3 contains an overview of the original PDR algorithm. Section 3.4 describes modifications, to the original algorithm, needed to integrate it with abstraction. Section 3.5 compares the approach presented in this paper with previous work. Section 3.6 shows experimental results. Section 3.7 concludes the chapter.

3.2 Background

It is assumed that the verification problem is presented to a model-checking engine as a sequential logic circuit with an all-0 initial state having a single property output. If the property holds, the output of the logic circuit evaluates to 0 for any state reachable from the initial state. If the property fails, the engine returns a CEX, which is a sequence of inputs taking the design from the initial state into a state where the output evaluates to 1. If the initial state is not constant-0, the sequential circuit can be equivalently transformed to ensure that the initial state is 0. Similarly, if there are more outputs than one, the problem can be transformed by ORing individual outputs together.

3.3 Property Directed Reachability (PDR)

In this section, we review the basics of the algorithm of Property Directed Reachability (PDR), which is presented in Algorithm 3.1.

Algorithm 3.1 PDR

```
Input: G                                                    ▷ G: the input circuit
Output: result ∈ { SAT, UNSAT }
 1: Ω ← {Init}                                              ▷ Ω: the PDR trace
 2: k ← 0                                                   ▷ k: the PDR depth
 3: while true do
 4:     c ← GetBadCube(G, k)
 5:     if c = ∅ then
 6:         k ← k + 1
 7:         Ω ← Ω ∪ {⊤}                                     ▷ Open a new frame
 8:         invariant, Ω ← PropagateBlockedCubes(G, Ω)
 9:         if invariant ≠ ∅ then
10:             return UNSAT
11:     else
12:         Ω, cex ← RecBlockCube(G, Ω, c, k)
13:         if cex ≠ ∅ then
14:             return SAT
15:
16: procedure RecBlockCube(G, Ω, c, k)
17:     Q ← ∅                                  ▷ the priority queue of proof obligations
18:     AddPOB(Q, c, k)
19:     while Q ≠ ∅ do
20:         s ← PopMin(Q)
21:         if Frame(s)= 0 then
22:             return GetCEX()
23:         result, z ← CheckCube(s)
24:         if result = UNSAT then
25:             z ← Generalize(z)
26:             AddBlockedCube(Ω, z)
27:         else                                           ▷ result is SAT
28:             AddPOB(Q, z)
29:             AddPOB(Q, s)
30:     return ∅
```

3.3.1 The PDR Trace

PDR performs an incremental computation creating sets of CNF clauses. Each set is associated with a timeframe, which over-approximates the states that are reachable in that timeframe. Such a list of sets of clauses is called a *PDR trace*: $\Omega = (R_0, R_1, \ldots, R_N)$. Every R_j is a set of clauses that over-approximates the set of states reachable from the initial states within j steps. The clauses in a PDR trace are called *reachability clauses*. Each R_j is also called the j-th *frame* in the trace.

Definition 3.1. Given an FSM, $M = (I, O, S, Init, T)$, and a property P, a *PDR trace* is a sequence of predicate functions, $\Omega = (R_0, R_1, \ldots, R_N)$, such that

1. $R_0(s) = Init(s)$
2. $R_j(s) \implies R_{j+1}(s)$ for $0 \leq j < N$.
3. $R_j(s) \wedge T(i, s, s') \implies R_{j+1}(s')$ for $0 \leq j < N$.
4. $R_j(s) \implies P(s)$ for $0 \leq j < N$.

We note that $R_N(s)$ does not necessarily imply $P(s)$, i.e. $R_N(s)$ can contain bad states. This is consistent with the presentation in [EMB11] but different than the original one in IC3 [Bra11].

3.3.2 Overview of PDR

Algorithm 3.1 shows a high-level view of the PDR algorithm in [EMB11]. The PDR trace (Ω) is initialized to contain a single element, the initial states. Before the main while loop starts (line 3), a new frame is opened (line 1 and 7) and bad states (states that violate the property) of this timeframe are enumerated (line 4). A bad state, m, is derived by checking the following SAT query:

$$R_k(s) \wedge \neg P(s) \tag{3.1}$$

For each bad state, PDR checks whether it overlaps with the initial states, and if so, the verification problem is satisfiable (SAT) and PDR terminates. If this bad state does not overlap with the initial states, *ternary simulation* is performed to expand a state minterm (m) into a state cube (c), such that all the states belonging to this cube (including the original minterm) make the property fail.

The obtained expanded cube composed of bad states (c), together with the frame it is derived from (k), is called a proof obligation (POB). This is because, to prove the property, we need to show that none of the states contained in this cube c are reachable from the initial states. A sufficient condition to check if POB (c, k) is reachable from the initial states is if the SAT query below is UNSAT.

$$R_{k-1}(s) \wedge \neg c(s) \wedge T(i, s, s') \wedge c(s') \tag{3.2}$$

If the query is UNSAT, then there is no R_{k-1}-state that can reach a state in cube c within a one-step state transition. This proves that all states in cube c are not reachable within k steps from the initial states. Therefore PDR can strengthen the trace Ω by blocking this cube:

$$R_j \leftarrow R_j \wedge \neg c, \quad 1 \leq j \leq k. \tag{3.3}$$

On the other hand, if Query (3.2) is SAT, then an R_{k-1}-cube d is extracted from the SAT solver and generalized by ternary simulation. Cube d at frame $k-1$ must be shown unreachable to prove the property. Therefore a new POB (d, $k-1$) is generated. The POBs are ordered in each timeframe by the time they are generated, which can be implemented with a priority queue (Q).

In procedure RecBlockCube of Algorithm 3.1, PDR retrieves POBs from the queue, one at a time, and checks if they can be blocked. Given a POB (c, t), this means checking the SAT query below.

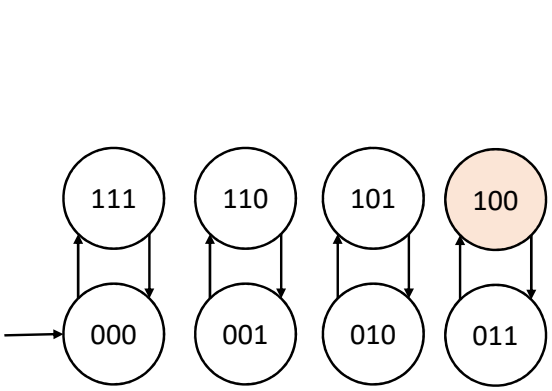
$$R_{t-1}(s) \wedge \neg c(s) \wedge T(i, s, s') \wedge c(s') \tag{3.4}$$

If the query is SAT, the POB is not blocked. There is a previous state, from which at least one state in the POB can be reached. This state is checked for being an initial state and, if not, a new POB is generated and queued.

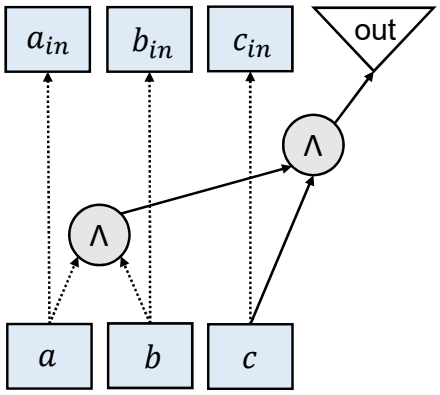
On the other hand, if query (3.4) is UNSAT, this POB is blocked: all the previous states that reach the POB are ruled out by the reachable-state over-approximation computed so far. Cube c is then generalized into a clause, which is added to the reachable state over-approximation under construction.

Finally, when PDR has finished blocking all bad states in a given timeframe, and the queue of proof-obligations is empty, PDR attempts to move the clauses forward, that is, to prove that the clauses holding in a given timeframe, also hold in the next timeframe. If, in any timeframe, all the computed clauses are moved, these clauses form a property-directed inductive invariant.

A property-directed inductive invariant (Inv) is a Boolean function defined over the flip-flop output variables, which is characterized as follows: (a) it contains the initial state; (b)



(a) The state transition graph of the FSM. The colored state is the bad state $(c \wedge \neg b \wedge \neg a)$.



(b) An and-inverter graph (AIG) of the FSM. The property fails if $out = 1$.

Figure 3.1: A simple finite state machine (FSM).

it does not contain bad states; and (c) for each state contained in the invariant, the next states reachable from it are contained in the invariant. Formally, it satisfies the properties below.

(a) $Init(s) \implies Inv(s)$

(b) $Inv(s) \wedge T(i, s, s') \implies Inv(s')$

(c) $Inv(s) \implies P(s)$

When such an inductive invariant is found, the property is proved because there does not exist a sequence of reachable states, originating in an initial state, leading to a bad state.

Example 3.1. Consider the finite state machine (FSM) shown in Figure 3.1. The state variables are $\{a, b, c\}$. The initial state is $\neg c \wedge \neg b \wedge \neg a$ and the bad state is $c \wedge \neg b \wedge \neg a$. PDR starts with $k = 0$ and $R_0 = Init(s) = \neg c \wedge \neg b \wedge \neg a$. There is no bad state at frame 0, so k increases to 1 and a new frame $R_1 = 1$ is added to Ω. Then PDR gets a bad state in R_1: $(c \wedge \neg b \wedge \neg a)$ (or `100`) , and checks if the bad state is reachable from R_0 (Query 3.4):

$$\neg c \wedge \neg b \wedge \neg a \wedge (\neg c \vee b \vee a) \wedge T(a, b, c, a', b', c') \wedge c' \wedge \neg b' \wedge \neg a'$$

The result is UNSAT because state `000` cannot reach `100`, as shown in Figure 3.1a. The minterm can be generalized into the cube $(c \wedge \neg a)$, meaning that state `000` cannot reach both `100` and `110`. R_1 is then strengthened by this cube and updated as $R_1 = \neg(c \wedge \neg a) = \neg c \vee a$. After the strengthening, there are no more bad states in R_1, so k increases to 2 and a new frame $R_2 = 1$ is added. PDR again gets a bad state in R_2: $(c \wedge \neg b \wedge \neg a)$, and checks if the

bad state is reachable from R_1 (Query 3.4):

$$(\neg c \vee a) \wedge (\neg c \vee b \vee a) \wedge T(a, b, c, a', b', c') \wedge c' \wedge \neg b' \wedge \neg a'$$

The result is SAT and a predecessor in R_1 is returned: $(\neg c \wedge b \wedge a)$. A POB, $(\neg c \wedge b \wedge a, 1)$, is generated and PDR handles this POB by checking if $(\neg c \wedge b \wedge a, 1)$ is reachable from R_0:

$$\neg c \wedge \neg b \wedge \neg a \wedge (c \vee \neg b \vee \neg a) \wedge T(a, b, c, a', b', c') \wedge \neg c' \wedge b' \wedge a'$$

The result is UNSAT because state `000` cannot reach `011`. The minterm is then generalized into the cube $(\neg c \wedge a)$ because state `000` cannot reach both `011` and `001`. PDR then tries to push this cube from R_1 to R_2 by checking

$$(\neg c \vee a) \wedge (c \vee \neg a) \wedge T(a, b, c, a', b', c') \wedge \neg c' \wedge a'$$

The result is UNSAT because the four states $\{\texttt{111}, \texttt{000}, \texttt{101}, \texttt{010}\}$ cannot reach any of the $\{\texttt{011}, \texttt{001}\}$. Therefore both R_1 and R_2 are strengthened by this cube and updated as $R_1 = (\neg c \vee a) \wedge (c \vee \neg a)$ and $R_2 = c \vee \neg a$. PDR then tries to push the other cube $(c \wedge \neg a)$ from R_1 to R_2 by checking

$$(\neg c \vee a) \wedge (c \vee \neg a) \wedge T(a, b, c, a', b', c') \wedge c' \wedge \neg a'$$

The result is also UNSAT because the four states in R_1 $\{\texttt{111}, \texttt{000}, \texttt{101}, \texttt{010}\}$ cannot reach any of the $\{\texttt{100}, \texttt{110}\}$. As a result, R_2 is strengthened by this cube and updated as $R_2 = (\neg c \vee a) \wedge (c \vee \neg a)$. PDR then finds out that $R_1 \equiv R_2$ and concludes that the problem is UNSAT with a property-directed inductive invariant $R_1 = R_2 = (\neg c \vee a) \wedge (c \vee \neg a)$.

3.4 The Algorithm: PDRA

The performance of PDR is hampered when it takes a long time to converge on an inductive invariant. There can be several reasons for this:

1. the reachable state space may be irregular making it hard to separate reachable states from bad states by using a two-level representation such as a set of clauses;

2. it may be possible to express the inductive invariant in the two-level form but PDR fails to find it because the state space exploration is unfocused.

It may be hard to mitigate the first limitation of PDR without developing a brand-new engine, which computes an over-approximation in a non-clausal form. In this chapter, we address the second limitation by making state-space exploration more focused. To this end, localization abstraction is added to the PDR engine, making the set of flop variables

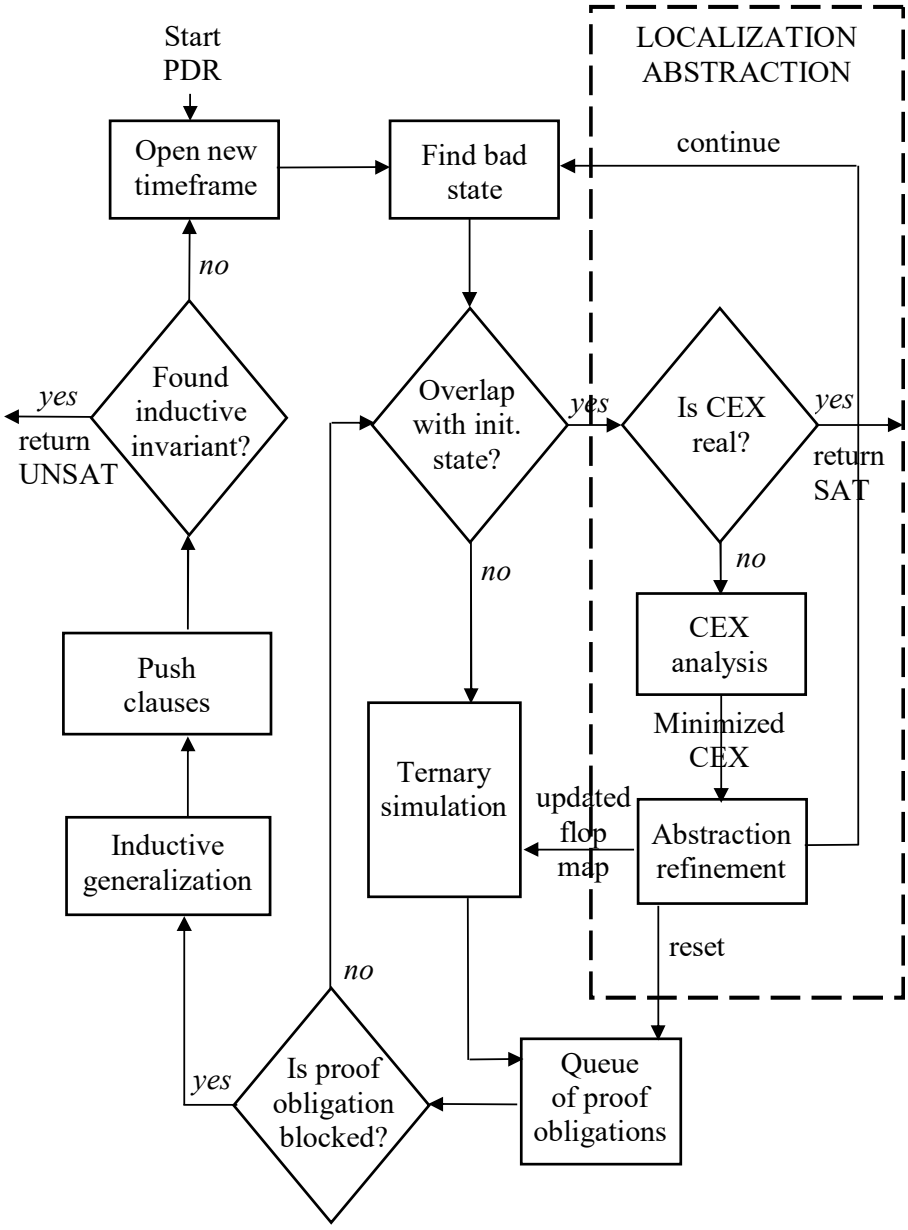


Figure 3.2: Overview of the PDRA algorithm.

Algorithm 3.2 PDR with Abstraction (PDRA)

Input: G ▷ G: the input circuit
Output: result ∈ { SAT, UNSAT }

```
1: Iterations ← 1
2: Ω ← {Init}                                   ▷ Ω: the PDR trace
3: k ← 0                                        ▷ k: the PDR depth
4: V ← CreateFlopMap()                          ▷ V: the flop map
5: while true do
6:     c ← GetBadCubeAbs(G, k, V)
7:     if c = ∅ then
8:         CleanFlopMap(V)
9:         k ← k + 1
10:        Ω ← Ω ∪ {⊤}                          ▷ Open a new frame
11:        invariant, Ω ← PropagateBlockedCubes(G, Ω)
12:        if invariant ≠ ∅ then
13:            return UNSAT
14:    else
15:        Ω, cex ← RecBlockCubeAbs(G, Ω, c, k, V)
16:        if cex ≠ ∅ then
17:            V′ ← AnalyzeCEX(G, V, cex)
18:            if V = V′ then
19:                return SAT
20:            else
21:                V ← V′
22:                Iterations ← Iterations + 1
23:
24: procedure RecBlockCubeAbs(G, Ω, c, k, V)
25:    Q ← ∅                                    ▷ the priority queue of proof obligations
26:    AddPOB(Q, c, k)
27:    while Q ≠ ∅ do
28:        s ← PopMin(Q)
29:        if Frame(s)= 0 then
30:            return GetCEX()
31:        result, z ← CheckCubeAbs(s, V)
32:        if result = UNSAT then
33:            z ← Generalize(z)
34:            AddBlockedCube(Ω, z)
35:        else                                 ▷ result is SAT
36:            AddPOB(Q, z)
37:            AddPOB(Q, s)
38:    return ∅
```

participating in the clauses grow in a more predicable manner, compared to the original engine. As a result, the state-space exploration becomes more focused and more likely to converge to an inductive invariant. The modified engine is called PDR with Abstraction (PDRA).

The modifications needed to go from PDR to PDRA are shown, in the block diagram in Figure 3.2, as boxes inside the dashed rectangle. The changes comprise counter-example (CEX) analysis and CEX-based abstraction refinement, affecting the PDR engine components as described below. PDRA is also presented in Algorithm 3.2.

PDRA maintains an additional data-structure called flop map (V), remembering what subset of flip-flops are used in the abstraction. A flip-flop is used in the abstraction if there is a clause containing a literal of the corresponding flop variable in any timeframe. Otherwise, a flop is not used. The flop map is empty at the beginning (line 4). It is incrementally updated by the abstraction refinement when enumerating bad states. The set of flops included in the flop map does not grow monotonically from frame to frame because the clauses containing certain flop variables may be subsumed later by stronger clauses, not containing these variables. As a result, some flop variables present in the flop map at an earlier timeframe may disappear in the later timeframes.

PDRA uses the flop map during *ternary simulation*. In particular, ternary simulation is used when PDR finds a bad cube at the current frame (Query 3.2) or a predecessor of a POB (Query 3.4). Procedures GetBadCubeAbs (line 6) and CheckCubeAbs (line 31) take the flop map as an input, so they can perform ternary simulation accordingly. In PDR, ternary simulation converts a bad-state minterm into a bad-state cube while removing as many flop variables as possible in a given order. If a flop variable cannot be removed, it is added to the POB and may later appear in the generalized clause when the POB is blocked. As a result, even if a flop variable is not used in any of the clauses so far, the original PDR adds it whenever needed. In contrast, PDRA treats flops not used in the abstraction as pseudo-primary inputs (PPIs). This allows the derived clauses to continue depending only on the flops used in the abstraction at the risk of running into a spurious CEX.

This is why, when a CEX is detected by PDRA (line 16), a dedicated CEX analysis is performed (line 17), as described in Section 2.3 [MEB+13] (Section 3.3 "Priority based abstraction refinement"). The analysis results in a set of PPIs needed for making the CEX fail the property output. If the set of PPIs is empty, then the CEX is real and PDRA returns SAT because no PPI can be added to block the CEX (line 19). Otherwise, the computed PPIs correspond to flops absent in the current abstraction. The next-state functions of these flops are added to the abstraction to rule out the given spurious CEX (line 21). Other spurious CEXes may be generated and ruled out in a similar manner.

At some point (when enough next-state logic functions have been added to the current abstraction) PDRA finishes the current timeframe without spurious CEXes (line 7). Then an

additional cleanup step is done where PDRA checks if the flops added by refinement appear in the generated clauses (line 8). Frequently, some flops do not appear in these clauses and can be removed from the flop map before PDRA opens the next timeframe. The CEX-based refinement is the same as the refinement step in GLA [MEB+13], while the cleanup step is analogous to the proof-based cleanup in GLA.

In summary, PDRA maintains a data structure called flop map to remember what flops are used in the abstraction. The flop map is empty at the beginning and grows from one frame to another. When a new timeframe is opened, PDRA tries to maintain the set of used flops unchanged compared to the previous timeframe. To this end, additional flops required by ternary simulation are treated as PPIs. Once a spurious CEX is found, refinement is performed, the queue of POBs is emptied, and the enumeration of bad states continues, as shown by the block contained within the dotted line in Figure 3.2. If a real CEX or an inductive invariant is discovered, PDRA terminates.

The modifications described in this section can be implemented on top of an available PDR engine, such as the one in ABC [BM10]. The implementation requires adding approximately 80 lines of C language code, not counting the CEX analysis code, which is reused from [MEB+13].

3.5 Comparison with Previous Work

The proposed method comes close to some previous work [BIMM12, VGS12, LS14, FYH16]. In particular, [BIMM12] integrates PDR and localization abstraction at a high level, by making these two engines exchange information. Flop variables participating in bounded PDR runs are scored and used to guide the abstraction. This is different from our approach, which essentially consists of building a minimalistic localization abstraction engine within the PDR engine.

The first fully integrated approach combining PDR with localization abstraction was presented in [VGS12]. However, the abstraction used there is "variable timeframe", as defined in [MEB+12], that is, in each timeframe, the abstraction states what flop outputs should be used to express clauses in the given timeframe. Our method is based on a simpler "fixed timeframe" abstraction used in [MEB+13].

The work of [LS14] combines PDR with abstraction by targeting datapath flip-flops to be abstracted. In contrast, our approach does not have information to distinguish control logic and datapath. PDRA tries to abstract any flops not used in a precise over-approximation of the reachable state space. Our approaches to datapath abstraction will be presented in the next chapter.

Table 3.1: Comparing different flavors of PDR in terms of the number of solved cases and runtime on 77 industrial examples (implementations with abstraction, pdr -t, treb -abs, and pdr -nct, are compared against the baselines, pdr, treb, and pdr -nc).

Test	AND	FF	pdr	pdr -t	treb	treb-abs	pdr -nc	pdr -nct
Ex01	509	142				33.26		
Ex02	509	142				57.80		626.73
Ex03	2602	330	23.47	18.77	31.33	43.94	16.88	39.60
Ex04	2602	330	26.64	24.26	38.75	54.07	26.59	46.50
Ex05	1135	242				317.04		
Ex06	2602	330	29.25	21.49	15.62	45.30	26.42	42.70
Ex07	2602	330	30.08	22.17	22.51	51.75	23.65	50.42
Ex08	1135	242				42.86		
Ex09	19886	782		47.05		148.37		56.14
Ex10	19387	771	38.58	13.71	18.70	24.76	15.10	15.88
Ex11	15555	607				103.44	546.90	
Ex12	15555	607				101.85	544.19	
Ex13	21772	782		308.74	544.75	143.55	138.14	183.21
Ex14	21302	771	116.40	14.56	26.12	30.49	20.93	23.12
Ex15	15555	607				105.72	549.58	
Ex16	21772	782		304.82	556.40	147.49	155.16	182.67
Ex17	5777	726				728.77	141.88	82.70
Ex18	479	89	0.59	5.31	0.15	6.09	1.01	1.21
Ex19	20068	3785	9.18	54.57	38.59	81.98	7.22	20.27
Ex20	20066	3785	19.53	10.46	28.02	21.71	10.50	6.94
Ex21	20047	3785		11.05		38.42		12.46
Ex22	20098	3795		658.28		840.66		311.08
Ex23	9985	2654		640.58				169.56
Ex24	2122	353	10.85	13.31	20.51	22.63	16.99	18.71
Ex25	5043	869	11.53	15.54	28.69	38.89	24.76	40.91
Ex26	7408	965	41.18	560.69	80.60	885.90	26.54	
Ex27	18347	1207	142.47	154.26	243.24	515.71	155.65	167.72
Ex28	1755	384		18.66	74.78	46.32	16.12	16.54
Ex29	1746	383		3.72		16.97		23.31
Ex30	11945	781	14.63	13.42	24.01	26.05	16.51	17.69
Ex31	4452	731		50.33		29.82	167.96	34.09
Ex32	1979	368	89.62	79.61	40.55	63.09	38.84	97.01
Ex33	1917	360	58.79	66.04	38.47	56.96	36.20	56.58
Ex34	1840	348	54.29	51.00	64.73	40.53	30.73	55.92
Ex35	1762	335	20.74	29.36	39.28	46.66	24.54	22.53
Ex36	1697	327	17.53	32.17	28.92	29.61	44.57	18.20
Ex37	2675	178	380.46	284.26				
Ex38	2360	178	600.22		279.76	289.69	321.80	275.02
Ex39	1973	146	70.55	61.12	51.19	110.57	123.74	148.24

Table continues on the right hand side

Test	AND	FF	pdr	pdr -t	treb	treb-abs	pdr -nc	pdr -nct
Ex40	36851	2434				348.63		316.15
Ex41	36851	2434				92.10		
Ex42	9895	2249		37.53	14.25	4.56	37.73	8.38
Ex43	9897	2249				6.37	322.77	382.68
Ex44	36851	2434				353.64		314.82
Ex45	9460	1564	28.40	8.14	70.10	52.34	32.46	16.31
Ex46	531	131	2.55	4.34	4.57	6.00	4.93	6.61
Ex47	920	231	9.38	8.63	15.79	20.06	12.21	8.90
Ex48	952	249	24.80	34.62	120.18	36.98	19.84	22.09
Ex49	2052	413			52.44		36.15	74.71
Ex50	1072	253	24.83	38.27	67.43	43.76	21.77	21.98
Ex51	952	249	28.73	22.72	98.06	27.39	14.10	24.99
Ex52	930	241	9.1	17.62	27.44	19.70	18.48	11.84
Ex53	890	229	27.71	19.70	31.32	22.88	15.51	16.64
Ex54	920	231	9.91	8.79	15.63	20.19	11.94	8.85
Ex55	934	239	11.12	18.47	20.36	17.36	15.97	15.90
Ex56	952	249	35.61	27.54	33.61	27.27	19.31	17.22
Ex57	1948	397				297.88	44.77	72.25
Ex58	872	221	16.83	13.34	39.24	13.47	12.24	12.58
Ex59	966	237	30.29	18.57	33.86	45.23	27.67	18.70
Ex60	952	249	21.55	20.16	90.53	25.97	20.96	17.26
Ex61	1050	183	0.46	1.98	4.48	19.74	0.77	0.40
Ex62	1533	252			26.02	32.38	7.28	6.47
Ex63	3632	521	103.22	166.92	180.20	358.97	308.01	287.2
Ex64	1600	309	5.00	1.62	10.92	3.53	4.61	1.74
Ex65	1189	227	80.72	104.11	55.66	84.38	20.07	27.4
Ex66	9422	1324	108.60	165.03	116.82	267.66	148.11	158.88
Ex67	6199	972	873.41	461.90			271.85	200.09
Ex68	1233	171		480.73			798.93	
Ex69	16745	3113		284.45		308.88	281.91	406.61
Ex70	16700	3107	101.77			422.50	117.75	157.22
Ex71	16701	3107		502.31		104.58	45.32	70.61
Ex72	16701	3107	221.54		135.05	140.39	22.70	149.08
Ex73	12049	2389			151.12	239.02	20.85	244.87
Ex74	541	76	4.73	13.93	1.04	51.50	7.44	17.04
Ex75	528	76	10.05	8.69	1.13	48.31	7.17	13.51
Ex76	1228	208		688.81			257.81	419.00
Ex77	1177	195	2.75	1.69	84.52	2.12	8.64	1.76
Solved			47	58	51	71	64	67
Time, %			1.000	1.047	1.400	1.812	0.973	1.038

Another integration of PDR with localization abstraction is described in [FYH16]. It uses gate-level abstraction while our approach is flop-level. The difference between the two is discussed in [MEB+13]. It is also important to note that our implementation is simpler. Given a clear understanding, our abstraction can be developed on top of a working PDR engine in a matter of hours.

3.6 Experimental Results

PDRA is part of two public verification tools: ABC [BM10] (command pdr -t) and ABC-ZZ [Een] (command treb -abs). The baseline of pdr and treb is reviewed in Section 3.3 and

originally described in [EMB11].

PDRA has been tested on HWMCC 2014 benchmarks [BH14] with inconclusive results because most of the testcases require preprocessing for PDR to be effective. Moreover, often a test case is solved by one flavor of PDR and not by others, making it hard to compare, except by the sheer number of cases solved.

Table 3.1 lists the runtimes, in seconds, taken by different PDR flavors to solve 77 unsatisfiable industrial verification instances of unknown origin. Empty entries indicate that the instance is not solved on a Linux workstation in 900 seconds. Table 3.1 shows several versions of PDR along with their corresponding abstracting versions (pdr, pdr -t), (treb, treb -abs), and (pdr -nc, pdr -nct). The last, pdr -nc, is a version of IC3 with improved generalization [HBS13]. As claimed, all three versions were modified fairly easily using the ideas outlined in this chapter.

The last row of Table 3.1 shows that the PDRs with abstraction solve more test cases than the PDRs without abstraction. The final row shows geometric averages of runtime for 41 out of the 77 test cases solved by all six flavors of PDR. The runtime overhead for PDRA is negligible, except for treb -abs, which takes 20% more time compared to its baseline, treb.

Table 3.2 compares different flavors of PDR on the 41 commonly solved test cases in terms of the the number of timeframes needed to converge to an invariant (Column "Frames"), and its clause count (Column "Size") and flop count (Column "Supp"). Table 3.2 demonstrates that when PDRA is used, the number of timeframes increases by about 10% on average, while the number of clauses and flops is reduced by 15-20% on average.

3.7 Conclusion

The chapter describes a practical variation of the known model checking algorithm PDR/IC3. The idea is to add localization abstraction to the baseline algorithm to reduce the set of flop output variables used in the over-approximation. The modified engine performs better in terms of the number of cases solved with a slightly increased runtime. Furthermore, it reduces the size of the inductive invariants, making them more suitable for design analysis and debugging.

Future work will include

- Using structural reverse engineering to detect control flops and target abstraction to include the remaining flops that likely belong to a datapath.

Table 3.2: Comparing different flavors of PDR in terms of the frame count and the invariant size on 41 industrial examples (implementations with abstraction, pdr -t, treb -abs, and pdr -nct, are compared against the baselines, pdr, treb, and pdr -nc).

			pdr			pdr -t			treb			treb -abs			pdr -nc			pdr -nct		
Test	AND	FF	Frame	Size	Supp	Frame	Size	Supp	Frame	Size	Supp	Frame	Size	Supp	Frame	Size	Supp	Frame	Size	Supp
Ex03	2602	330	9	4222	228	15	3929	178	11	3906	196	12	3914	179	9	3998	208	9	4099	178
Ex04	2602	330	15	4108	228	16	4069	186	11	3926	203	14	4047	193	15	4112	226	16	4097	206
Ex06	2602	330	11	4197	209	14	4073	178	15	3858	184	10	3879	193	11	4127	206	16	4096	186
Ex07	2602	330	12	4285	256	13	4110	178	10	3845	194	14	3945	198	15	4151	236	18	4120	196
Ex10	19387	771	4	3104	379	3	2626	99	3	2577	103	3	2562	96	4	2824	177	3	2627	99
Ex14	21302	771	4	3504	442	4	2636	99	3	2587	108	3	2563	96	4	2798	135	3	2640	99
Ex18	479	89	14	74	65	20	306	61	11	53	48	25	140	58	15	139	66	17	110	46
Ex19	20068	3785	33	661	256	62	1194	227	35	359	192	65	276	161	30	356	267	57	348	215
Ex20	20066	3785	60	1285	382	107	523	42	59	486	98	63	212	43	74	693	121	75	228	40
Ex24	2122	353	11	2134	242	13	1932	191	7	2014	237	9	1782	167	10	2104	230	12	1972	166
Ex25	5043	869	4	4123	60	7	4139	68	4	4136	58	8	4130	67	4	4132	59	7	4182	110
Ex27	18347	1207	17	2403	1077	17	2457	1077	17	2217	1077	17	1378	1078	17	2410	1077	17	2432	1077
Ex30	11945	781	8	634	247	9	603	210	8	582	240	8	563	187	9	601	243	9	602	222
Ex32	1979	368	24	3398	339	50	2466	340	19	1981	339	26	1594	339	22	1942	338	44	2447	340
Ex33	1917	360	21	3174	333	29	2732	331	21	2184	331	24	1553	328	21	1955	336	22	2389	333
Ex34	1840	348	44	1930	320	36	2462	317	26	2103	315	20	1744	315	36	1375	320	40	1919	320
Ex35	1762	335	22	1619	310	30	1737	305	42	1409	307	24	1519	302	22	1700	305	20	1571	305
Ex36	1697	327	22	1271	298	26	1635	298	18	1257	292	20	1456	296	25	1984	299	28	1105	298
Ex39	1973	146	8	4534	137	8	4096	126	8	3746	132	8	3859	127	9	3898	137	8	4045	135
Ex45	9460	1564	59	1022	246	58	659	208	67	909	209	62	743	208	80	1121	265	58	865	207
Ex46	531	131	8	1056	120	9	1168	121	8	1057	119	9	950	116	8	1186	119	9	1085	118
Ex47	920	231	14	1637	174	14	1464	169	14	1664	186	20	1595	167	17	1237	179	19	1179	169
Ex48	952	249	15	2364	233	20	3074	235	15	4933	235	14	2975	228	17	1833	233	19	2090	237
Ex50	1072	253	19	2208	236	17	2949	230	16	2532	232	16	2510	229	15	1835	239	18	1785	237
Ex51	952	249	16	2853	234	21	2485	231	15	3900	234	13	1781	229	21	1418	235	19	2427	237
Ex52	930	241	15	1596	192	21	1972	193	16	3516	216	15	2037	196	16	1749	206	17	1222	191
Ex53	890	229	20	2831	212	16	2459	212	15	2452	207	20	1455	205	17	1812	214	16	1759	208
Ex54	920	231	14	1637	174	14	1464	169	14	1664	186	20	1595	167	17	1237	179	19	1179	169
Ex55	934	239	18	1597	186	15	2294	192	17	2781	206	18	1336	183	23	1846	195	17	1953	182
Ex56	952	249	21	3015	233	21	2369	228	21	2907	234	16	1853	231	21	1446	235	17	2154	229
Ex58	872	221	15	2326	195	19	1789	170	16	2702	198	15	1402	152	21	1428	193	17	1341	175
Ex59	966	237	19	2489	220	17	2580	217	15	2521	219	18	2665	219	20	1975	223	18	1968	219
Ex60	952	249	15	2653	233	17	2629	233	17	4232	232	19	1752	227	17	2285	234	17	1859	224
Ex61	1050	183	8	124	84	10	109	67	13	123	81	14	362	82	11	101	76	10	84	53
Ex63	3632	521	95	1441	513	116	1460	512	122	1222	509	115	1241	510	177	1741	514	152	1670	513
Ex64	1600	309	81	303	138	15	321	136	81	353	138	12	244	130	65	306	140	19	302	139
Ex65	1189	227	9	2831	216	10	2974	217	10	2602	216	9	1801	217	9	1422	216	9	1626	216
Ex66	9422	1324	18	1440	1323	25	1455	1323	16	1405	1323	18	1414	1324	19	1400	1323	22	1394	1323
Ex74	541	76	43	23	15	179	10	17	39	70	18	416	12	18	54	63	21	160	67	24
Ex75	528	76	53	17	15	109	19	17	39	83	19	369	12	17	61	65	21	142	67	22
Ex77	1177	195	21	482	139	16	304	66	28	4088	138	13	258	64	28	926	178	15	313	68
Geo			1.000	1.000	1.000	1.161	0.976	0.829	0.968	1.089	0.886	1.098	0.798	0.816	1.085	0.926	0.948	1.130	0.871	0.843

- Exploring different abstraction refinement strategies, which might be better at ruling out counterexamples.
- Developing an application-specific SAT solver to speed up PDR/IC3 with and without abstraction.

Chapter 4

Property Directed Reachability with Word-Level Abstraction

PDRA presented in Chapter 3 integrates bit-level PDR with bit-level localization abstraction. This chapter presents PDR-WLA, a word-level algorithm that efficiently integrates bit-level PDR with word-level localization abstraction by re-using reachability information learned in previous refinement iterations.

4.1 Introduction

Unbounded model checking (UMC) on a Register-Transfer-Level (RTL) circuit is hard but has important applications in the IC design industry:

1. **Sequential equivalence checking (SEC)**. An RTL circuit is sequentially synthesized by retiming, clock-gating, pipelining etc., and UMC is required for proving the correctness of the synthesis.

2. **Property checking**. UMC is used to prove that a circuit always satisfies a set of given properties.

UMC is challenging at the bit level, and even more so at the word level, where complex arithmetic operators, such as multipliers, adders, and variable shifters, are involved.

IC3 [Bra11] or *Property Directed Reachability* (PDR) [EMB11] is considered the best algorithm for bit-level UMC. Abstraction has been a key development and is widely used. Different methods of abstraction include the following. *Word-level abstraction* [JKSC05, ALS08,

BBSO10, BBS11, LS14, HCR+16] can be effective by abstracting away heavy arithmetic logic. *Localization abstraction* [WJK+01] is a method where gates or signals are replaced by new unconstrained primary inputs. *Counterexample guided abstraction and refinement* (CEGAR) [CGJ+00] is a framework for iterating abstraction and refinement, where refinement is based on the analysis of *spurious* counterexamples.

We propose *PDR-WLA*, an efficient CEGAR-based word-level localization algorithm integrated with PDR. Given a word-level design, PDR-WLA starts with the extreme abstraction with all *hard signals* (e.g., outputs of multipliers, adders, etc.) abstracted (i.e. replaced by new primary inputs). Next, the resulting word-level abstraction is bit-blasted and given to a modified PDR algorithm. If a counterexample (CEX) is found, PDR-WLA simulates it on the original design to check if it is *real*. If so, PDR-WLA reports it and terminates; otherwise, the CEX is *spurious* and is used to refine the current abstraction. Then a new iteration begins with the refined abstraction.

The main contributions embodied in PDR-WLA are that it

- integrates word-level abstraction with PDR efficiently,
- uses a new refinement strategy that takes advantage of *structural* and *proof-based* analysis of spurious counterexamples, and
- re-uses *reachability information* (reachability clauses) derived in previous iterations.

PDR-WLA is implemented and available in the public verification tool ABC [BM10] (command *%pdra*). It was evaluated on a set of 195 industrial Verilog RTL benchmarks. PDR-WLA is capable of solving 18 hard problems not solved by PDR. The results also show that 1) reusing previously derived reachability clauses improves performance significantly and 2) the new refinement strategy is the most effective compared to several others proposed and tested.

This chapter starts with background material in Section 4.2. PDR-WLA is presented in Section 4.3. The proposed refinement strategies are given in Section 4.4. Related work is discussed in Section 4.5. Experiments are presented in Section 4.6. Conclusion and future work are discussed in Section 4.7.

4.2 Preliminaries

4.2.1 The UMC problem

The input is a word-level circuit given in *structural Verilog* containing 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. Reset values are either constants or free variables (unknown value X). A design is modeled as a finite state machine (FSM).

Definition 4.1. An *FSM* is a tuple $M = (I, O, S, Init, T)$ where I is the set of PIs, O is the set of POs, S is the set of FFs, $Init$ is the set of initial states, and T is the set of (deterministic) transition relations where $T \subseteq I \times S \times S$. If $(i, s, s') \in T$, there exists a transition from s to s' under i.

The input word-level circuit is assumed to contain a single FSM and a single output, *out*, representing a property to be checked. If the problem is to prove equivalence between two designs, it is assumed that a *miter* circuit, M, has been created by merging all PIs and merging FFs if their correspondences are known. The miter's output, *out*, is a Boolean signal, which is the OR of the pairwise XORs of the corresponding outputs of the two designs. Thus $out = 1$ if the two designs are different. Similarly for property checking, *out* is the output of a monitor, and $out = 1$ if the property fails. In terms of linear temporal logic (LTL), the UMC problem is formulated as $M \models \mathbf{G}\neg out$, i.e. *out* is never 1 if the property holds.

A UMC solver either reports a *counterexample* (CEX) that falsifies the property or produces an *inductive invariant* proving that the property holds globally.

Definition 4.2. A *counterexample (CEX)* is a sequence of PI assignments driving the design from an initial state into a state falsifying the property.

Definition 4.3. An *inductive invariant* (Inv) proving a property $P(s)$ is a predicate function satisfying the properties below.

1. $Init(s) \implies Inv(s)$
2. $Inv(s) \wedge T(i, s, s') \implies Inv(s')$
3. $Inv(s) \implies P(s)$

4.2.2 Property Directed Reachability

A detailed review of PDR is presented in Section 3.3 following the ideas in [EMB11]. Algorithm 4.1 outlines a high-level view of PDR. It maintains a list of sets of clauses, called the *PDR trace*: $\Omega = (R_0, R_1, \ldots, R_N)$. Every R_j is a set of clauses that over-approximates the set of states reachable from the initial states within j steps. These clauses in a PDR trace are called *reachability clauses*.

Definition 4.4. Given an FSM, $M = (I, O, S, Init, T)$, and a property P, a *PDR trace* is a sequence of predicate functions, $\Omega = (R_0, R_1, \ldots, R_N)$, such that

1. $R_0(s) = Init(s)$
2. $R_j(s) \implies R_{j+1}(s)$ for $0 \leq j < N$.
3. $R_j(s) \wedge T(i, s, s') \implies R_{j+1}(s')$ for $0 \leq j < N$.
4. $R_j(s) \implies P(s)$ for $0 \leq j < N$. [1]

Algorithm 4.1 PDR

Input: G_M ▷ G_M: the bit-level input circuit
Output: status ∈ { SAT, UNSAT }
1: $\Omega \leftarrow \{Init\}$ ▷ Ω: the PDR trace
2: $k \leftarrow 0$ ▷ k: the PDR depth
3: **while true do**
4: Ω, cex $\leftarrow$ RecBlockCube(G_M, Ω, k)
5: **if** cex $\neq \emptyset$ **then**
6: **return** SAT ▷ Found a real CEX
7: $k \leftarrow k + 1$
8: $\Omega \leftarrow \Omega \cup \{\top\}$ ▷ Open a new frame
9: $\Omega \leftarrow$ PropagateBlockedCubes(G_M, Ω)
10: **if** Ω contains a fixed point **then**
11: **return** UNSAT

PDR starts with the trace Ω with only one element $R_0 = Init$. It then tries to strengthen the trace by recursively blocking bad cubes (a cube of states that can make the property fail) (line 4). If a bad cube intersects with the initial states, then a CEX is returned. Otherwise, the last element R_k of the trace now satisfies the property P. PDR then adds a new element $\top$ (empty set of clauses) to Ω, and tries to propagate clauses (using induction) from R_1 to R_k (line 9). If a fixed point ($R_j = R_{j+1}$) is found, the problem is declared UNSAT and

[1] $R_N(s)$ does not necessarily imply $P(s)$, i.e. $R_N(s)$ can contain bad states. Recursive blocking (line 4) tries to remove bad states from $R_N(s)$.

the inductive invariant (R_j) is returned. Other details of procedures RecBlockCube and PropagateBlockedCubes can be found in Section 3.3 and [EMB11].

4.2.3 Word-level Abstraction

Localization abstraction [WJK+01] is used in this chapter. Given a word-level circuit and a set of target signals (e.g., outputs of arithmetic operators), an abstraction is created by replacing the target signals with free variables called *pseudo PIs* (PPIs). Localization is not necessarily restricted to flip flops; *any* signal can be abstracted, similar to GLA [MEB+13]. More details of word-level localization abstraction are presented in Section 2.2.1.

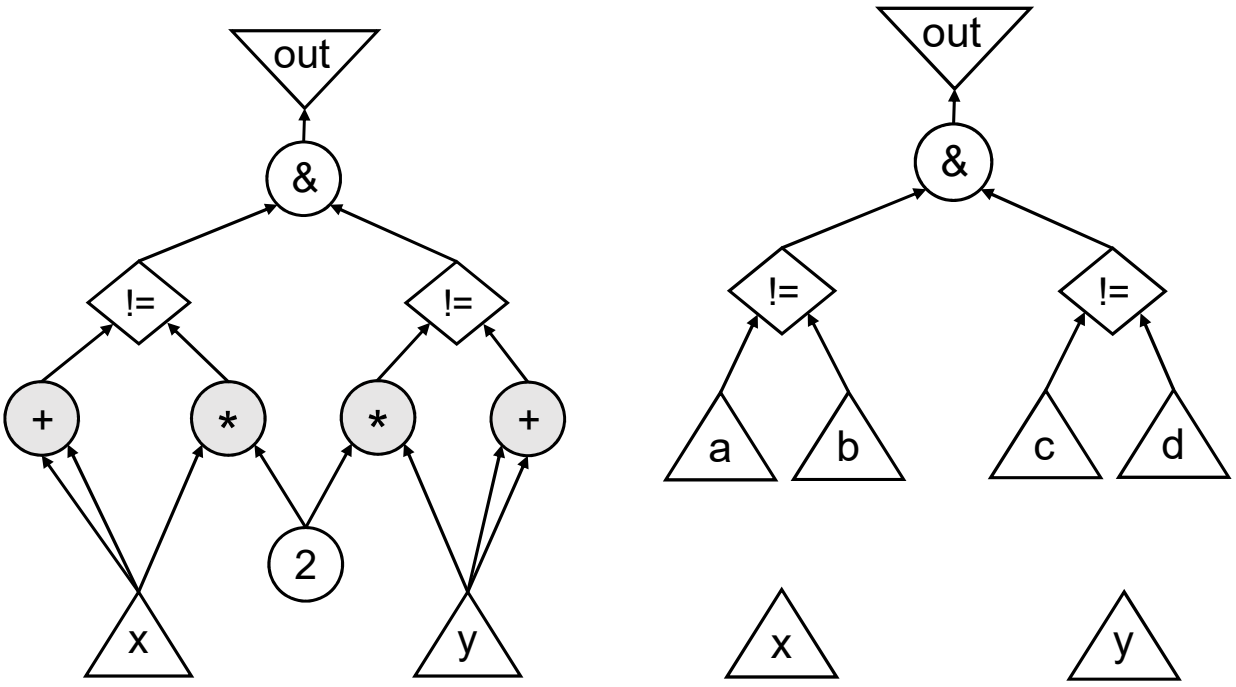
Example 4.1. Consider the circuits in Figure 4.1. The PO, *out*, in Figure 4.1a is constant-0, since both $2 \times x \equiv x + x$ and $2 \times y \equiv y + y$ are true. Figure 4.1b is the result of abstracting all 4 arithmetic operators by replacing their outputs with PPIs. Note that while the example is combinational for illustration purposes, the abstraction scheme applies generally to sequential circuits and UMC problems.

Definition 4.5. Given an original circuit M and an abstraction A of M, a CEX of A is *real* if it can falsify the property on M (make $out = 1$). Otherwise, it is *spurious*.

4.2.4 Simple CEGAR (S-CEGAR)

Algorithm 4.2 (S-CEGAR) is an example of a simple integration of CEGAR and PDR at the word level. The algorithm starts by abstracting *all* signals in the set $\mathcal{S}$ (e.g., outputs of all specified arithmetic operators). Next, an abstraction-refinement loop is entered where each iteration begins by creating a word-level abstraction based on the current set $\mathcal{B}$, the set of signals to be abstracted away. The abstraction is then bit-blasted and solved by a bit-level PDR. If the solver returns UNSAT, the property is proved. Otherwise a CEX to the abstraction, *cex*, exists and is then *simulated* on the original circuit (W_M) to check if it is *real*. If yes, the property is falsified and *cex* is returned; otherwise *cex* is analyzed to derive a set of signals ($\Delta\mathcal{B}$) that, if un-abstracted, can block *cex*. A new abstraction, with $\Delta\mathcal{B}$ un-abstracted, is then created and a new iteration begins.

In each iteration of S-CEGAR, a new PDR solver is used and reachability clauses are recomputed from scratch. This is inefficient when the algorithm needs many iterations to find a *final* abstraction, i.e. one that proves the property.



(a) The original circuit with four arithmetic operators, where x and y are primary inputs, 2 is a constant, ! = is the complement of a comparator, & is a bit-wise AND, and *out* is the negation of the property.

(b) An abstraction derived from the original by replacing the 4 arithmetic operators with 4 new primary inputs, a, b, c, and d.

Figure 4.1: A combinational circuit illustrating word-level abstraction. $out \equiv 0$, UNSAT, since $2 \times x \equiv x + x$, which forces *out* to be constant 0.

Algorithm 4.2 Simple CEGAR (S-CEGAR)

Input: W_M ▷ W_M: the word-level input circuit
Input: $\mathcal{S}$ ▷ $\mathcal{S}$: the initial set of targeted signals
Output: status ∈ { SAT, UNSAT }
1: $Iterations \leftarrow 1$
2: $\mathcal{B} \leftarrow \mathcal{S}$ ▷ $\mathcal{B}$: the set of abstracted signals
3: **while true do**
4: $W_A \leftarrow$ CreateAbstraction(W_M, $\mathcal{B}$)
5: $G_A \leftarrow$ BitBlast(W_A)
6: cex $\leftarrow$ PDR(G_A)
7: **if** cex $\neq \emptyset$ **then**
8: **if** IsRealCEX(W_M, cex) **then**
9: **return** SAT
10: **else**
11: $\Delta\mathcal{B} \leftarrow$ Refine(W_M, G_A, $\mathcal{B}$, cex)
12: $\mathcal{B} \leftarrow \mathcal{B} \backslash \Delta\mathcal{B}$
13: $Iterations \leftarrow Iterations + 1$
14: **else**
15: **return** UNSAT

4.3 PDR with Word-Level Abstraction

4.3.1 The Algorithm

PDR-WLA uses an important insight; *PDR traces* can be re-used between iterations if abstractions are *monotone*. The idea is similar to previous work of PDR with abstraction [VGS12, FYH16], extending it to the word level.

Similar to PDR, PDR-WLA starts with the trace Ω containing only $R_0 = Init$. One difference is that PDR-WLA works on an abstraction instead of the original circuit. Similar to S-CEGAR, it begins by abstracting all targeted signals $\mathcal{S}$, resulting in a word-level abstraction (W_A), which is then bit-blasted into a circuit (G_A). As with PDR, PDR-WLA tries to recursively block bad cubes at depth k with the abstract model G_A and the trace Ω. If a bad cube intersects with the initial states, then a CEX, *cex*, is returned and checked on the original circuit (W_M). If *cex* is also a CEX on W_M, the property is falsified; otherwise *cex* is used to compute a subset ($\Delta\mathcal{B}$) of $\mathcal{B}$ to refine the current abstraction ($\Delta\mathcal{B}$ will be un-abstracted). Note that a *nonempty* $\Delta\mathcal{B}$ exists because *cex* can always be blocked by un-abstracting some signals. Set $\mathcal{B}$ is updated by removing $\Delta\mathcal{B}$. A new abstraction is derived for the next iteration of recursive blocking. If PDR-WLA successfully blocks bad cubes at the current depth k, then it increments the depth by one and adds a new element ($\top$) to Ω.

It then tries to propagate the clauses in Ω using induction. If a fixed point is found, then the property holds; otherwise, blocking bad cubes at the new depth will be tried (line 10).

Note that PDR-WLA can be viewed as a PDR algorithm with on-the-fly word-level abstraction. The same trace Ω is re-used throughout the computation, even though the current abstraction is continuously refined. Thus, important reachability information derived in previous iterations is re-used, resulting in a significant speedup over S-CEGAR.

Algorithm 4.3 PDR with Word-Level Abstraction (PDR-WLA)

Input: W_M ▷ W_M: the word-level input circuit
Input: $\mathcal{S}$ ▷ $\mathcal{S}$: the initial set of targeted signals
Output: status ∈ { SAT, UNSAT }
1: $Iterations \leftarrow 1$
2: $\Omega \leftarrow \{Init\}$ ▷ Ω: the PDR trace
3: $k \leftarrow 0$ ▷ k: the PDR depth
4: $\mathcal{B} \leftarrow \mathcal{S}$ ▷ $\mathcal{B}$: the set of abstracted signals
5: $W_A \leftarrow$ CreateAbstraction(W_M, $\mathcal{B}$)
6: ▷ W_A: the word-level abstraction
7: $G_A \leftarrow$ BitBlast(W_A)
8: **while true do**
9: **while true do**
10: Ω, cex $\leftarrow$ RecBlockCube(G_A, Ω, k)
11: **if** cex $\neq \emptyset$ **then**
12: **if** IsRealCEX(W_M, cex) **then**
13: **return** SAT
14: **else**
15: $\Delta\mathcal{B} \leftarrow$ Refine(G_A, $\mathcal{B}$, cex)
16: $\mathcal{B} \leftarrow \mathcal{B} \backslash \Delta\mathcal{B}$ ▷ Un-abstract some signals
17: $W_A \leftarrow$ CreateAbstraction(W_M, $\mathcal{B}$)
18: $G_A \leftarrow$ BitBlast(W_A)
19: $Iterations \leftarrow Iterations + 1$
20: **else**
21: **break**
22: $k \leftarrow k + 1$
23: $\Omega \leftarrow \Omega \cup \{\top\}$ ▷ Open a new frame
24: $\Omega \leftarrow$ PropagateBlockedCubes(G_A, Ω)
25: **if** Ω contains a fixed point **then**
26: **return** UNSAT

4.3.2 Analysis of PDR-WLA

PDR-WLA represents a general framework for word-level abstraction. It is complementary to other abstraction techniques. The only requirement for soundness is that the derived sequence of abstractions (line 17) is *monotone*:

Definition 4.6. Let $\{A_j\}$ be a sequence of abstractions, let $\{T_j\}$ be their transition relations, and let $\{Init_j\}$ be their initial states. $\{A_j\}$ is *monotone* if $T_{j+1}(i,s,s') \implies T_j(i,s,s')$ and $Init_{j+1}(s) \implies Init_j(s)$.

Theorem 4.1. *Let M and A be FSMs where $T_M \implies T_A$ and $Init_M \implies Init_A$. Given a property P, if $\Omega = (R_0, R_1, \ldots, R_N)$ is a PDR trace of A with P, then $\Omega' = (Init_M, R_1, \ldots, R_N)$ is a PDR trace of M with P.*

Proof. Since Ω is a PDR trace of A with P, we have

$$\begin{aligned} R_j(s) &\implies R_{j+1}(s) && \text{for } 0 \le j < N \\ R_j(s) \wedge T_A(i,s,s') &\implies R_{j+1}(s') && \text{for } 0 \le j < N \\ R_j(s) &\implies P(s) && \text{for } 0 \le j < N \end{aligned}$$

Note that Ω' is the same as Ω, except that R_0 is replaced by $Init_M$. Since $Init_M \implies R_0$ and $T_M \implies T_A$, we have

$$\begin{aligned} Init_M(s) &\implies R_1(s) \\ Init_M(s) \wedge T_M(i,s,s') &\implies R_1(s') \\ R_j(s) \wedge T_M(i,s,s') &\implies R_{j+1}(s') \text{ for } 1 \le j < N \\ Init_M(s) &\implies P(s) \end{aligned}$$

Therefore by Definition 4.4, Ω' is a PDR trace of M with P. □

Theorem 4.2. *Algorithm 4.3 is sound and complete.*

Proof. **Soundness.** It is sound to start a new iteration with the previous trace (line 10) because each iteration makes the current abstraction tighter by removing signals from $\mathcal{B}$. Note that R_0 is the initial states of the original circuit (W_M) and is shared by all abstractions. Similarly any state variable in clauses from a previous abstraction must remain in the next abstraction because abstractions are monotone. Thus, a trace can be safely copied over to the next abstraction (Theorem 4.1). Finally, Algorithm 4.3 is sound because it returns UNSAT only if it finds an inductive invariant proving the property.

Completeness. The algorithm returns SAT only if a CEX is real. Convergence follows because, in each iteration, the size of $\mathcal{B}$ decreases by at least one (otherwise the CEX must be real). The number of iterations is bounded by $|\mathcal{S}|$. □

4.4 Refinement

Given a spurious CEX, *cex*, the goal of refinement is to identify a subset of signals $\Delta\mathcal{B}$ in $\mathcal{B}$, such that if $\Delta\mathcal{B}$ is removed from $\mathcal{B}$, then *cex* is blocked in the next iteration. We say that $\Delta\mathcal{B}$ is *un-abstracted*.

In Chapter 2, several refinement strategies are presented. Simulation-based Refinement (SBR) generalizes a CEX into an XCEX (Section 2.2.5) that represents a set of CEXes needs to be blocked. The remaining concrete-value PPIs in an XCEX are our candidate subset $\Delta\mathcal{B}_{SBR}$. However, SBR does not guarantee that the CEX (or XCEX) would be blocked in the refined circuit, since SBR does not use any information from the original circuit. In contrast, Proof-based Refinement (PBR) explicitly encodes assumptions into a circuit with *ITE* operators (or multiplexers), so it can take advantage of both the information in the original circuit and the assumption interfaces in SAT solvers, which result in an efficient procedure with the guarantee that the current CEX would be blocked after refinement. The returned subset of assumptions are our candidate subset $\Delta\mathcal{B}_{PBR}$. Maximum Fan-out Free Cone (MFFC) refinement exploits the underlying circuit structure that can save unnecessary iterations. The main idea is that if a signal is un-abstracted, its MFFC is better un-abstracted also. Therefore we propose the following refinement strategy for PDR-WLA:

1. Compute $\Delta\mathcal{B}_{PBR}$, a set of candidate signals, using PBR.
2. Compute $\Delta\mathcal{B}_{MFFC}$, the set of signals in the intersections of the MFFCs of $\Delta\mathcal{B}_{PBR}$ and $\mathcal{B}$.
3. Derive set $\Delta\mathcal{B}$: $\Delta\mathcal{B} = \Delta\mathcal{B}_{PBR} \cup \Delta\mathcal{B}_{MFFC}$.

4.5 Related Work

4.5.1 Word-level Abstraction and Model Checking

Most previous work is *bounded* in that it requires unrolling a circuit to a certain depth k, and then they use SMT solvers [JKSC05, ALS08, BBSO10, BBS11, KP]. These methods rely on Bounded Model Checking (BMC) [BCCZ99] and/or k-induction [SSS00]. This becomes inefficient when deep unrolling is needed. In practice, BMC- and induction- based approaches are efficient in finding CEXes, but often incapable of producing an inductive invariant, which is required for UMC problems. PDR-WLA addresses unbounded problems and does not require unrolling.

Welp and Kuehlmann proposed a generalization of PDR to the theory of quantifier free formulas over bit-vectors (QF_BV) [WK13, WK14]. Hybrid simulation and mixed types of atomic reasoning units are used for inductive and CEX generalization. However, they do not re-use PDR traces nor do they perform word-level abstractions.

The closest work to PDR-WLA is AVERROES [LS14], a word-level algorithm integrating CEGAR and PDR. It abstracts wide data-paths into uninterpreted predicates, constants, terms, and functions, and solves the abstraction with an SMT-based PDR (where SMT solvers are used instead of SAT). The main differences between PDR-WLA and AVERROES are

- PDR-WLA re-uses PDR traces derived in previous iterations; AVERROES does not.
- PDR-WLA uses PBR and MFFC as the main refinement strategy; AVERROES uses strategies similar to SBR, PBR-A, and PBR-B.

UFAR [HCR+16] is a word-level algorithm that combines CEGAR and bit-level model checking. It abstracts arithmetic operators with black boxes as well as uninterpreted function constraints, and solves the abstraction with a portfolio of tools, including BMC and PDR. However, UFAR does not reuse PDR traces nor does it perform MFFC refinement.

4.5.2 PDR with Abstraction

Vizel et al. proposed L-IC3 [VGS12], a bit-level IC3 with localization abstraction, where state variables are the targeted signals and different abstractions are used in different time frames. Fan et al. showed that gate-level abstraction (GLA) [MEB+13] can be integrated with PDR [FYH16]. However, both approaches consider only bit-level problems. At the word-level, abstracting only state variables may result in aggressive refinement where the entire logic cone of a flip flop would be refined, limiting scalability. On the other hand, GLA cannot be applied directly to the word level. In particular, it mainly uses SBR without considering MFFC, which could be ineffective as discussed in Section 2.6.

In contrast, PDR-WLA considers not only flip flops, but any type of signals, resulting in a finer-grained abstraction and refinement. Also, it uses specific procedures, PBR and MFFC, to find a final abstraction faster than the bit-level GLA.

4.6 Experimental Results

Experiments were done to evaluate PDR-WLA using different settings. PDR-WLA is part of the public verification tool, ABC [BM10] (command *%pdra*), which can parse word-level Verilog and transform the resulting design into a bit-level circuit by bit-blasting. For comparison, S-CEGAR (Section 4.2.4, Algorithm 4.2) was implemented in ABC (command *%abs*).

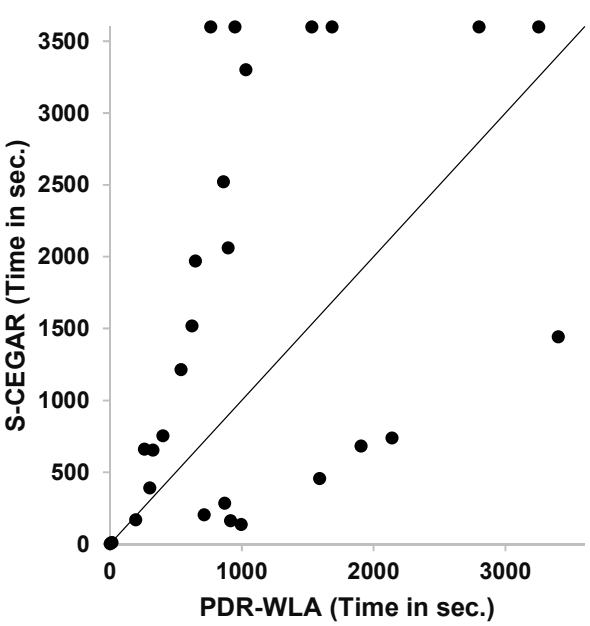
The benchmarks used for evaluating PDR-WLA were a set of 195 industrial Verilog RTL designs. Large arithmetic operators and multiplexers were the signals targeted for possible abstraction (set $\mathcal{S}$). A workstation with Intel Xeon E5504 CPUs clocked at 2.0 GHz with 24 GB of RAM was used. A time-out of 3600 seconds was used on all experiments.

First, we compare PDR-WLA to the original PDR [EMB11], in which the input Verilog circuit is immediately bit-blasted. Given a 1-hour time-out, PDR-WLA solves 22 fewer test-cases than PDR (89 vs. 111), but PDR-WLA manages to solve 18 hard cases not solved by PDR. It is our experience that abstraction doen not always work. It is likely that many of these 22 cases cannot be abstracted well, so trying such is a waste of time. However, The two methods are nicely complementary; together they can solve 129 out of 195 benchmarks. Thus PDR-WLA complements PDR and would work well in a portfolio-based word-level model checker like [HCR+16].

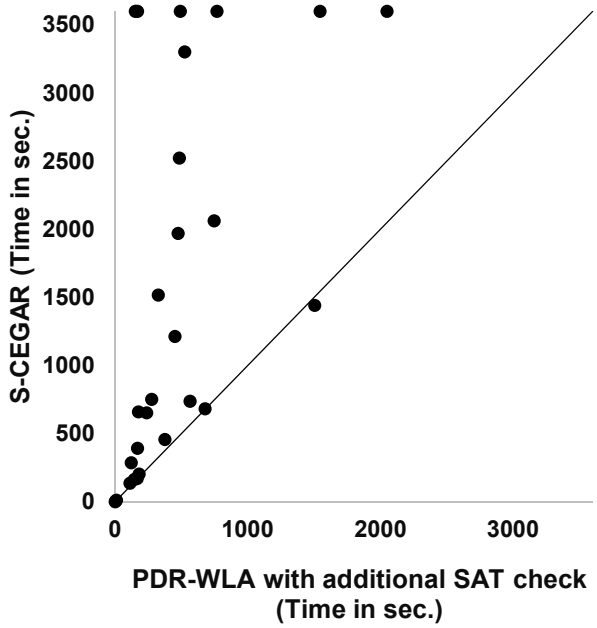
To demonstrate the importance of re-using PDR traces in PDR-WLA, it was compared with S-CEGAR, which uses a fresh PDR solver in each iteration and does not preserve the reachability clauses across PDR runs. The results are shown in Figure 4.2, where the x and y axes represent the solving times of PDR-WLA and S-CEGAR, respectively. In Figure 4.2a, PDR-WLA outperforms S-CEGAR in all but eight cases. An investigation revealed that in some problems, after several iterations of refinement, an abstraction can become *combinationally UNSAT*, implying that the circuit output can be proved UNSAT with all FFs un-initialized. In those cases, PDR-WLA would work hard to get a non-trivial inductive invariant while S-CEGAR proves that the problem is UNSAT after just one SAT call. To address this problem, PDR-WLA was enhanced to always check if the problem is combinationally UNSAT when an iteration begins. The results are shown in Figure 4.2b, where PDR-WLA dominates S-CEGAR in all but one case.

20 out of the 195 designs were chosen for Table 4.1 to give an idea of details such as expected ranges of iterations needed, clauses in PDR traces re-used, and the sizes of $\mathcal{B}$ (signals to be abstracted way) in the final abstractions. All are UNSAT; each is characterized by the number of *hard signals*.

Definition 4.7. A *hard signal* is the output of



(a) Running with the default settings, PDR-WLA outperforms S-CEGAR in many cases but not all of them.



(b) With appropriate additional SAT checking, PDR-WLA is able to outperform S-CEGAR in all but one case.

Figure 4.2: Comparison of PDR-WLA (*%pdra*) and S-CEGAR (*%abs*). This shows the effectiveness of re-using PDR traces. Note that PDR-WLA and S-CEGAR would be the same if no PDR traces can be re-used. Therefore, only 29 cases with non-zero re-used PDR traces are shown.

Table 4.1: Detailed experimental results for 20 unsatisfiable word-level test-cases. #HardSignals is the number of hard signals (Definition 4.7). $|S|$ and $|B|$ are sizes of the set of the initial targeted signals ($\mathcal{S}$) and the set of signals to be abstracted away for each iteration ($\mathcal{B}$) in Algorithm 4.3. #ReusedClauses is the number of clauses in PDR traces re-used by PDR-WLA. The number is 0 if all refinements occur at $k = 0$. The details of SBR, MFFC, PBR, and PBR-B can be found in Chapter 2.

ID	#Hard Signals	|S|	CPU Time (seconds)						Iterations					#ReusedClauses				|B| in the last iteration				
			pdr	%abs (S1) SBR +MFFC	%pdra (S2) SBR +MFFC	%pdra (S3) PBR-B +MFFC	%pdra (S4) PBR	%pdra (S5) PBR +MFFC	S1	S2	S3	S4	S5	S2	S3	S4	S5	S1	S2	S3	S4	S5
1	1252	100	479.32	170.50	196.46	369.95	**145.23**	164.67	2	2	3	4	4	11	110	181	181	88	88	98	92	92
2	1437	100	1759.97		3253.29	956.76	931.43	**914.51**	4	4	11	4	4	7438	8129	1493	1493	79	79	81	81	81
3	1437	100	1201.74	653.70	326.80	308.24	**306.83**	335.50	2	2	3	3	3	17	100	155	155	87	87	97	94	94
4	1437	100	1800.60		1529.84	1299.36	**583.27**	597.26	4	5	11	5	5	4981	11061	1732	1732	82	79	82	77	77
5	1437	100	931.73	753.11	401.78	272.46	**169.87**	170.91	2	2	3	3	3	19	84	114	114	87	87	96	90	90
6	1437	100	2531.39		2799.57	1128.25	**672.48**	686.62	4	4	11	6	6	3661	6804	2694	2694	78	78	81	78	78
7	1437	100	1383.61	2521.83	862.04	925.89	**410.96**	415.29	5	4	11	6	6	2080	7241	3317	3317	78	78	85	79	79
8	1252	100	925.41	1213.75	538.48	472.20	**225.96**	227.98	4	4	11	6	6	2518	10122	3889	3889	78	78	83	79	79
9	1437	100	1984.69		949.32	2573.81	387.51	**366.87**	4	4	8	5	5	2304	9868	2198	2198	80	79	82	77	77
10	1437	100	850.77	391.41	302.57	766.21	242.32	**225.09**	2	2	5	5	5	113	625	693	693	90	90	95	94	94
11	1437	100	1151.91	2060.92	896.89	958.62	372.40	**349.45**	4	4	10	5	5	2456	7017	1776	1776	78	78	80	79	79
12	133	101	13.61			675.78		**10.38**	4	4	17	17	10	0	2486	0	0	11	11	21	27	30
13	133	101	15.00			624.42		**8.99**	4	4	16	19	10	0	1713	0	0	15	15	21	26	30
14	94	75			763.06	197.19	295.68	**112.98**	6	6	8	11	6	135	228	551	139	3	3	2	21	3
15	95	75			1685.29	745.23	**259.12**	816.54	7	7	8	11	6	147	115	475	151	3	3	1	21	3
16	82	82		545.37	507.48			**417.23**	3	3	4	12	2	0	0	0	0	12	12	0	0	33
17	72	72	353.69	124.98	128.85	132.70	**77.52**	113.61	9	9	14	18	9	0	0	0	0	16	16	16	14	17
18	58	58	1684.21	1343.36	1237.67	1270.25		**861.53**	3	3	4	9	2	0	0	0	0	13	13	13	13	13
19	2150	103	1731.26	**731.82**	732.24	1544.19		789.06	3	3	18	18	12	0	0	0	0	76	76	77	74	77
20	1132	100	**414.30**	739.13	2138.99	3045.19	2191.30	1296.62	3	3	35	40	33	481	5510	10307	4520	13	13	17	15	9

1. an adder, subtractor with width of at least 8, or

2. a multiplier, divider, modulus with width of at least 4, or

3. a multiplexer with width of at least 8.

The initial set of targeted signals ($\mathcal{S}$) is chosen from hard signals with an upper bound of 50 for each of the three categories (e.g., there can be at most 50 adders in $\mathcal{S}$). For each test-case, we show the runtime of six solvers: a) one PDR, b) one S-CEGAR (*%abs*), and c) four PDR-WLA versions (*%pdra*) with different refinement strategies (Chapter 2).

Observations from Table 4.1 are given below.

1. **PDR-WLA vs. PDR**. PDR-WLA generally is more efficient when proving hard problems for which small abstractions can be derived. On the other hand, if a problem cannot be abstracted well (e.g., case 20), PDR performs better.

2. **S-CEGAR vs. PDR-WLA**. An important factor in the comparison, is the number of *re-used clauses* of all previous PDR traces. If the number is high, a high speedup in PDR-WLA is usually observed. Case 20 is an exception to this, where the re-use number is non-trivial but PDR-WLA is still slower. The reason is that the design

becomes combinationally UNSAT after 3 iterations. This problem can be fixed by additional SAT calls as shown in Fig. 4.2. Note that there can be 0 re-used clauses (e.g., cases 16-19), since all refinements occur at $k = 0$ and no bad states are blocked at $k = 1$. If the trace Ω contains only $R_0 = Init$, no clause can be re-used in the next iteration.

3. **SBR (S2) vs. PBR (S5)**. PBR uses more iterations and derives smaller final abstractions (large $|\mathcal{B}|$) in most cases, implying that PBR leads to more fine-grained and focused refinements.

4. **PBR-B (S3) vs. PBR (S5)**. PBR uses less iterations to find a final abstraction, while PBR-B takes more iterations, which can be avoided by a proper analysis (see Example 2.14). PBR-B can derive a small final abstraction, but large numbers of iterations can cause poor performance. Note: comparison with PBR-A was not done due to its similarity to SBR.

5. **Without MFFC (S4) vs. with MFFC (S5)**. MFFC can be crucial in preventing unnecessary refinement iterations. This is critical in cases 12, 13, 16, 18, and 19.

4.7 Conclusion

PDR-WLA efficiently integrates PDR with word-level abstraction. It re-uses PDR traces, or reachability clauses, derived in previous iterations of refinement. An effective refinement strategy, PBR with MFFC, was developed which was shown capable of deriving small final abstractions using fewer iterations. PDR-WLA was implemented in the public verification system ABC and evaluated on industrial benchmarks. PDR-WLA solves more hard problems and offers speedups, compared to PDR and S-CEGAR.

Future work.

- Integrate BMC into Algorithm 4.3. The idea is that BMC can help PDR-WLA find spurious CEXes faster. Early prototypes suggest speedups in some benchmarks.

- Develop a good way to *shrink* abstractions. A shrinking procedure can be useful as shown in GLA [MEB+13]. One of the main challenges is that PDR traces cannot be re-used if abstractions are no longer monotone.

- Enhance the refinement strategies with *constraints*. For example, uninterpreted function constraints are known to be effective for SEC problems; partial interpretation constraints can also be useful. The challenge is to derive and apply constraints efficiently and automatically.

Chapter 5

Uninterpreted Function Abstraction and Refinement

Motivated by industrial benchmarks characterized by many related arithmetic operators, this chapter presents UFAR, a word-level MC algorithm that uses uninterpreted functions (UF) constraints as a method of refinement. The explicit application of UF constraints also enables the UFAR framework to integrate any bit-level or word-level MC algorithm, including both PDRA in Chapter 3 and PDR-WLA in Chapter 4.

5.1 Introduction

Model checking (MC) 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 can be classified as follows. One directly "bit-blasts" the problem and then solves with bit-level techniques such as IC3/PDR [Bra11, EMB11], interpolation [McM03], or BDDs [BCM+92]. Another [KP] translates the problem into SMT formulas (if possible) and then directly employs SMT solvers such as Boolector [BB09], or Z3 [DMB08]. A third [JKSC05] applies *predicate abstraction* [GS97]. *Term-level abstraction* [Hun89, BD94, BLS02, LSB02, LB03, AS04, ALS08, BBSO10, BBS11] replaces arithmetic operators with uninterpreted functions (UF), and then solves with SMT solvers.

However, bit-level techniques are problematic when verifying circuits with heavy arithmetic logic. Techniques adapted from software verification are often not effective for hardware equivalence checking. Most SMT-based approaches rely on (incomplete) bounded model checking (BMC) [BCCZ99] or induction [SSS00] 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 [AS04, ALS08, BBSO10, BBS11] 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) [CGJ+00]. 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 experimented 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 67 of the benchmarks within a one hour time limit for each example. To illustrate the results on a variety of examples with different ranges, we show detailed results on 100 examples having ranges of 4-475 multipliers, 6442-306429 AIG nodes, and 86-5627 flip-flops.

This chapter first presents background material and formal settings in Section 5.2. The UFAR algorithm is presented in Section 5.3. Several improvement techniques for the algorithm are given in Section 5.4. Section 5.5 gives some details about the UFAR framework, including word-level representation and bit-blasting this into an AIG. Related work is discussed in Section 5.6. Experimental results on an extensive set of industrial problems are presented in Section 5.7, comparing the effectiveness of the two improvements and the overall

UFAR algorithm. Conclusions are discussed in Section 5.8.

5.2 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.

5.2.1 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 design is modeled as a finite state machine (FSM).

Definition 5.1. 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 to UFAR, M, is assumed to be *mitered* as a single FSM and with 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.

5.2.2 Word-level Signals (Bit-Vectors)

In Verilog, a word-level signal (or bit-vector) is characterized by its bit-width and signedness.

Definition 5.2. A *word-level signal* in Verilog is denoted as s^w where s is the symbol of the signal and $w \in \mathbb{Z}$ represents both the bit-width and signedness of the signal. The bit-width of the signal is $|w|$. The signal is *signed* if $w < 0$; *unsigned* otherwise. A word-level signal s^w is also a finite function whose domain is $\{x | x \in \mathbb{N}, 0 \leq x < |w|\}$ and the co-domain is $\{0, 1\}$, mapping each bit position to its value.

Example 5.1. Consider the Verilog signal below.

```
wire signed [3:0] b;
```

The signal is denoted as b^{-4} from Definition 5.2.

A word-level signal can be converted into either a natural number or an integer using two's complement.

Definition 5.3. Given a word-level signal s^w, the word-to-natural-number function, $N(\cdot)$, maps this signal to a natural number. Let each bit in the signal be $s_i = s^w(i)$, where s_0 is the least significant bit (LSB) and $s_{|w|-1}$ is the most significant bit (MSB). Function $N(\cdot)$ is defined as,

$$N(s^w) = \sum_{i=0}^{|w|-1} s_i 2^i. \tag{5.1}$$

Definition 5.4. Given a word-level signal s^w, the word-to-integer function, $Z(\cdot)$, maps this signal to an integer. Let each bit in the signal be $s_i = s^w(i)$. Function $Z(\cdot)$ is defined as,

$$Z(s^w) = -s_{|w|-1} 2^{|w|-1} + \sum_{i=0}^{|w|-2} s_i 2^i. \tag{5.2}$$

Example 5.2. Given a word-level signal $b^{-4} = \texttt{1101}$. The natural number value of b^{-4} is

$$N(b^{-4}) = 2^3 + 2^2 + 0 + 2^0 = 13.$$

The integer value of b^{-4} is given by

$$Z(b^{-4}) = -2^3 + 2^2 + 0 + 2^0 = -3.$$

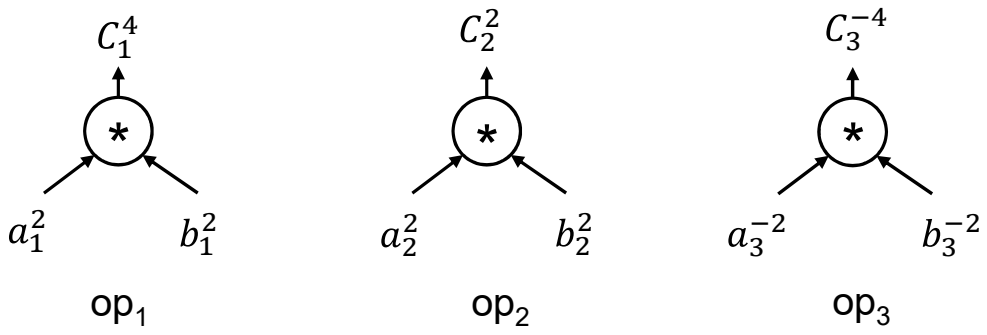


Figure 5.1: Three multipliers with different functions.

5.2.3 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 $\{+, -, *, /, \%, \ll, \gg\}$. In Verilog, an operator is instantiated by a *structural statement* which only states the function type of the operator and the connection between signals. 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`. An operator is modeled as a labeled node with a single output, up to two inputs, and its label of function type.

Definition 5.5. An operator *op* is a tuple $op = (o^k, i_1^l, i_2^m, t)$ where o^k is the output signal, i_1^l and i_2^m are the input signals, and t is the label of function type.

Example 5.3. Consider the following Verilog snippet describing a multiplier.

```
1: wire signed [1:0] a;
2: wire signed [1:0] b;
3: wire        [2:0] c;
4: assign            c = a * b;
```

The multiplier in line 4 is denoted as $op = (c^3, a^{-2}, b^{-2}, *)$. 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.

5.2.4 Functions of Word-level Operators

In Verilog, the actual *function* associated with an arithmetic operator is determined by the bit-widths and signedness of its inputs, output, and function-type. Given an arithmetic operator $op = (o^k, i_1^l, i_2^m, t)$, the function of op depends on (k, l, m, t). Operators with the same function type do not necessarily have the same function; a function-type represents a set of functions.

Example 5.4. Consider the three multipliers shown in Figure 5.1. They 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 2 bits. The functions of the three multipliers are given below using the standard integer multiplication ($\times$).

- op_1: It is an unsigned multiplication since both inputs are unsigned. The output bit-width 4 is large enough to prevent overflow. Therefore the operation is formulated using the integer multiplication with signals interpreted as natural numbers (function N).

$$N(c_1^4) = N(a_1^2) \times N(b_1^2). \tag{5.3}$$

- op_2: It is an unsigned multiplication similar to op_1, but the output bit-width is only 2, which requires an additional modulo operator (mod 2^2).

$$N(c_2^2) = N(a_2^2) \times N(b_2^2) \mod 2^2 \tag{5.4}$$

- op_3: It is a signed multiplication since both inputs are signed. The output bit-width 4 is large enough to prevent overflow. Therefore the operation is formulated using the integer multiplication with signals interpreted as integers (function Z).

$$Z(c_3^{-4}) = Z(a_3^{-2}) \times Z(b_3^{-2}). \tag{5.5}$$

5.2.5 Generic Operators

Example 5.4 shows that a multiplication symbol ($*$) in Verilog can implement many variants of multiplication functions based on the characteristics of the operator. This makes it more difficult to apply uninterpreted function (UF) abstractions. However, it can be observed that those multipliers share one thing in common: standard integer multiplication ($\times$). This observation inspires the idea of *generic operators*, which are ideal Verilog operators implementing their *standard integer functions*. For example, a generic multiplier

Table 5.1: The standard integer functions (SIF) for Verilog operators.

Verilog Operator	Standard Integer Function
$(c^k, a^l, b^m, +)$	$SIF_+(a^l, b^m) = Z(a^l) + Z(b^m)$
$(c^k, a^l, b^m, -)$	$SIF_-(a^l, b^m) = Z(a^l) - Z(b^m)$
$(c^k, a^l, b^m, *)$	$SIF_*(a^l, b^m) = Z(a^l) \times Z(b^m)$
$(c^k, a^l, b^m, /)$	$SIF_/(a^l, b^m) = Z(a^l) \ /^* \ Z(b^m)$
$(c^k, a^l, b^m, \%)$	$SIF_\%(a^l, b^m) = Z(a^l) - (Z(a^l) \ /^* \ Z(b^m)) \times Z(b^m)$
$(c^k, a^l, b^m, \ll)$	$SIF_\ll(a^l, b^m) = Z(a^l) \times 2^{N(b^m)}$
$(c^k, a^l, b^m, \gg)$	$SIF_\gg(a^l, b^m) = N(a^l) \ /^* \ 2^{N(b^m)}$

* This division function takes an integer x and an integer $y \neq 0$, and returns the integer part of x divided by y (truncated integer division). The result for $y = 0$ is an unknown value (X).

$op = (c^k, a^l, b^m, *)$, always implements standard integer multiplication without using modulo operations:

$$Z(c^k) = Z(a^l) \times Z(b^m) \tag{5.6}$$

Definition 5.6 (Standard Integer Function)**.** Given a word-level operator $op = (c^k, a^l, b^m, t)$ where $t \in \{+, -, *, /, \%, \ll, \gg\}$, the *standard integer function* (SIF) of op takes input signals a^l and b^m, evaluates them as integers, and then generates an integer output based on regular integer operations. SIFs for different function-types are defined in Table 5.1.

Definition 5.7 (Generic Operator)**.** A *generic operator* is a Verilog operator that implements its standard integer function (Definition 5.6). Formally, given a Verilog operator $op = (c^k, a^l, b^m, t)$, let its SIF be SIF_t. Operator op is a *generic operator* if

$$\forall a, b, c. \quad Z(c^k) = SIF_t(a^l, b^m). \tag{5.7}$$

Example 5.5. Consider the three multipliers shown in Figure 5.1. Since their function types are all $t = *$, we have the multiplication SIF: $SIF_*(a^l, b^m) = Z(a^l) \times Z(b^m)$. We can examine if the three multipliers are generic operators:

- op_1: It is not a generic operator, since there exists a counterexample: $(c^4, a^2, b^2) = (1000, 10, 10)$ such that

$$Z(c^4) = -8 \neq Z(a^2) \times Z(b^2) = -2 \times -2 = 4.$$

- op_2: It is not a generic operator, since there exists a counterexample: $(c^2, a^2, b^2) =$

(00, 10, 10) such that

$$Z(c^2) = 0 \neq Z(a^2) \times Z(b^2) = -2 \times -2 = 4.$$

- op_3: It is a generic operator, since it is a signed multiplication and its output would not overflow.
$$\forall a, b, c. \quad Z(c^{-4}) = Z(a^{-2}) \times Z(b^{-2}).$$

With generic operators, all same-type generic operators (e.g., multipliers) are considered to have the same function since they all implement the same SIFs. This is important for uninterpreted function (UF) abstraction since uninterpreted function constraints are valid only for same-function classes.

Example 5.6. Given two generic multipliers $op_1 = (c_1^{k_1}, a_1^{l_1}, b_1^{m_1}, *)$ and $op_2 = (c_2^{k_2}, a_2^{l_2}, b_2^{m_2}, *)$, since they implement the same SIF, the following implication holds, which is important in UF abstraction.

$$\left(Z(a_1^{l_1}) = Z(a_2^{l_2}) \wedge Z(b_1^{m_1}) = Z(b_2^{m_2})\right) \rightarrow \left(Z(c_1^{k_1}) = Z(c_2^{k_2})\right).$$

Non-generic operators can be converted to generic ones by transforming their input and output signals. Signals used or produced by a generic operator must be converted from unsigned to signed signals or vice versa. They also need to be converted by truncation, sign extension, or zero extension. This is modeled by introducing *signal converting functions*, which emulate what Verilog does in its assignment operator (=) and concatenation operator ({}).

Definition 5.8 (Signal Converting Function)**.** A *signal converting function*, denoted as $c^k = SC(a^l, Sign, k)$, takes an input signal a^l, a Boolean value $Sign$, and an output bit-width and signedness integer $k \in \mathbb{Z}$. The signal converting function generates its output c^k in the following ways.

- $|k| \leq |l|$ (truncation):
$$c^k(i) = a^l(i) \text{ for } 0 \leq i < |k| \tag{5.8}$$

- $|k| > |l|$ and $Sign = 1$ (sign extension):
$$\begin{cases} c^k(i) = a^l(i) & \text{for } 0 \leq i < |l| \\ c^k(i) = a^l(|l| - 1) & \text{for } |l| \leq i < |k| \end{cases} \tag{5.9}$$

- $|k| > |l|$ and $Sign = 0$ (zero extension):

$$\begin{cases} c^k(i) = a^l(i) & \text{for } 0 \le i < |l| \\ c^k(i) = 0 & \text{for } |l| \le i < |k| \end{cases} \tag{5.10}$$

Example 5.7. Given a signal $a^4 = \texttt{1000}$, the output of $SC(a^4, 1, -6)$ is

$$c^{-6} = \texttt{111000}.$$

In general, Verilog does not show the implicit signal conversion mechanism it uses to convert numbers. It is not needed since intermediate conversion results are not referred to. However, we will need to "expose" this conversion in order to apply UFs widely. Other benefits of explicitly exposing generic operators include 1) The generic operator agrees with the arithmetic of not only *bit-vectors* but *integers*, and 2) it unifies unsigned and signed operators. A procedure of "exposing" the generic operator within a non-generic one will be presented in Section 5.3.2.

Example 5.8. Consider the non-generic multiplier $op = (c^{16}, a^{-16}, b^{16}, *)$ shown in Figure 5.2a. This is an unsigned multiplication with the output truncated. The function of op is

$$N(c^{16}) = N(a^{-16}) \times N(b^{16}) \mod 2^{16}$$

In order to use a generic operator to represent op, we first convert both inputs (a^{-16}, b^{16}) to signed signals with leading zeros inserted (ag^{-17}, bg^{-17}) using signal converting functions and observe that

$$N(a^{-16}) = Z(SC(a^{-16}, 0, -17)) = Z(ag^{-17})$$
$$N(b^{16}) = Z(SC(b^{16}, 0, -17)) = Z(bg^{-17})$$

This leads to

$$N(a^{-16}) \times N(b^{16}) = Z(ag^{-17}) \times Z(bg^{-17})$$

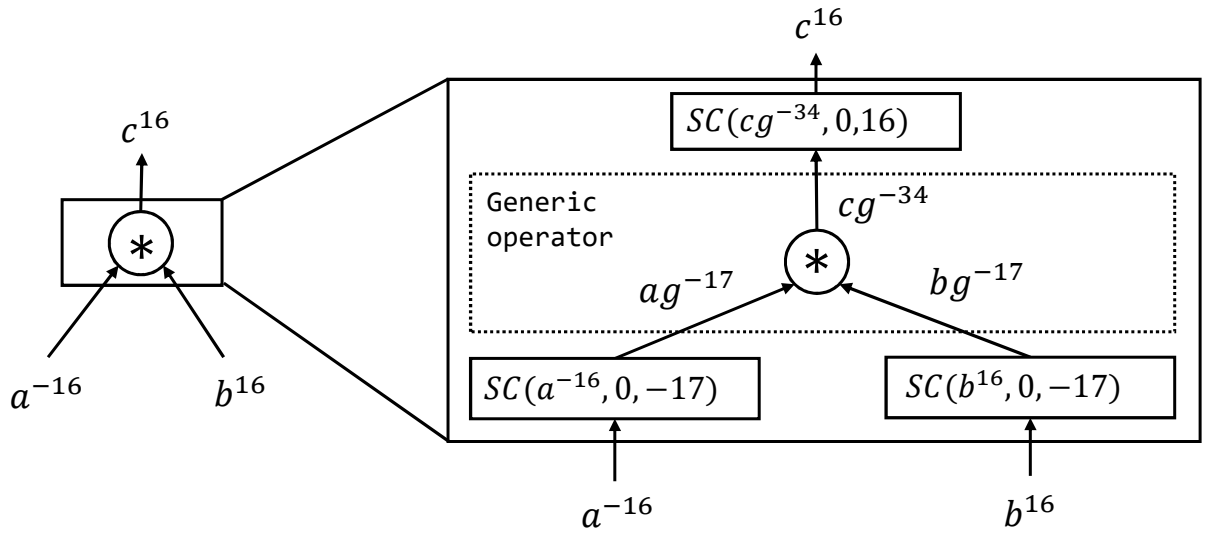
With the converted inputs, a generic multiplier can be created: $gop = (cg^{-34}, ag^{-17}, bg^{-17}, *)$ such that

$$N(a^{-16}) \times N(b^{16}) = Z(ag^{-17}) \times Z(bg^{-17}) = Z(cg^{-34})$$

Finally, we show that the original output c^{16} can be converted from cg^{-34} with another signal converting function (note that $Z(cg^{-34}) = N(cg^{-34})$ since the MSB of cg^{-34} is 0):

$$\begin{aligned} N(c^{16}) &= N(a^{-16}) \times N(b^{16}) \mod 2^{16} = Z(cg^{-34}) \mod 2^{16} = N(cg^{-34}) \mod 2^{16} \\ &= N(SC(cg^{-34}, 0, 16)) \end{aligned}$$

This example shows that a non-generic multiplier can be represented by a generic multiplier with proper conversions of input and output signals, as shown in Figure 5.2.



(a) The relationship between a non-generic multiplier and its generic version. Signal converting function SC is given in Definition 5.8.

```
wire signed [15:0] a;
wire        [15:0] b;
wire        [15:0] c = a * b;
```

Expose

```
wire signed [15:0] a;
wire        [15:0] 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.

Figure 5.2: An example showing how generic operators are modeled and exposed.

5.2.6 Uninterpreted Function (UF) Constraints

In the theory of uninterpreted functions (UF), the congruence axiom states that given any n-ary function f, the Property (5.11) holds, stating that if the inputs are equal then the two outputs must be equal.

$$\forall x_1, \ldots, x_n, y_1, \ldots, y_n. \bigwedge_{i=1}^{n} x_i = y_i \Rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \tag{5.11}$$

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 both generic operators of the same function-type. By two signals (a^l, b^m) being equal, we will mean that they are equal when evaluated as integers $(Z(a^l) = Z(b^m))$. For example, signals $a^{-2} = \texttt{10}$ and $b^{-4} = \texttt{1110}$ are equal because $Z(a^{-2}) = -2 = Z(b^{-4})$. Then Property (5.11) holds with these modifications. Therefore,

> *a UF constraint is valid between any pair of same function-type generic operators (even if they have different bit widths).*

The notion of two signals being equal when evaluated as integers can be defined using the comparison operator (`==`) in Verilog.

Definition 5.9 (Verilog Equality)**.** Two signals, s_1^l and s_2^m, 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. Therefore, given two same-signed signals s_1^l and s_2^m, if `s1 == s2`, then $Z(s_1^l) = Z(s_2^m)$.

Definition 5.10. Given two same function-type generic operators, $op_1 = (o_1^{k_1}, i_{11}^{l_1}, i_{12}^{m_1}, t)$ and $op_2 = (o_2^{k_2}, i_{21}^{l_2}, i_{22}^{m_2}, t)$, the UF constraint, denoted as c, is either Constraint (5.12) or (5.13) using the Verilog equality.

- If t is asymmetric:

$$c \texttt{ = } (i_{11}^{l_1} \texttt{==} i_{21}^{l_2}) \wedge (i_{12}^{m_1} \texttt{==} i_{22}^{m_2}) \Rightarrow (o_1^{k_1} \texttt{==} o_2^{k_2}) \tag{5.12}$$

- If t is symmetric:

$$c = \left((i_{11}^{l_1} == i_{21}^{l_2}) \wedge (i_{12}^{m_1} == i_{22}^{m_2}) \Rightarrow (o_1^{k_1} == o_2^{k_2})\right) \bigwedge \left((i_{11}^{l_1} == i_{22}^{m_2}) \wedge (i_{12}^{m_1} == i_{21}^{l_2}) \Rightarrow (o_1^{k_1} == o_2^{k_2})\right) \tag{5.13}$$

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 5.3.3). 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 5.11. 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.

5.3 UFAR

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

5.3.1 The Algorithm

Algorithm 5.1 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 5.3.7.

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 5.1 UFAR

Input: M ▷ M: the input miter
Input: $\mathcal{S}$ ▷ $\mathcal{S}$: the set of problematic operators
Output: status ∈ { 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

Algorithm 5.1 begins with the procedure of exposing generic operators (see Section 5.3.2). 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 5.3.3. The abstraction is then bit-blasted and solved by state-of-the-art bit-level engines concurrently (see Section 5.3.4). 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 5.3.5). 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 5.3.6). 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*.

5.3.2 Exposing Generic Operators

In this subsection, we propose a procedure that transforms any adder or multiplier by exposing their generic versions. Recall from Definition 5.7 that a generic adder or multiplier (c^k, a^l, b^m, t) must implement their SIFs.

$$Z(c^k) = SIF_t(a^l, b^m) = \begin{cases} Z(a^l) + Z(b^m), & t = + \\ Z(a^l) \times Z(b^m), & t = * \end{cases} \tag{5.14}$$

We start with sufficient conditions for generic adders and multipliers.

Theorem 5.1. *An adder $(c^k, a^l, b^m, +)$ is generic if both conditions below are true, meaning that all input and output signals are signed and that the output bit-width prevents overflow.*

$$\begin{cases} k < 0 \wedge l < 0 \wedge m < 0 \\ |k| = \max(|l|, |m|) + 1 \end{cases} \tag{5.15}$$

Theorem 5.2. *A multiplier $(c^k, a^l, b^m, *)$ is generic if both conditions below are true, meaning that all input and output signals are signed and that the output bit-width prevents overflow.*

$$\begin{cases} k < 0 \wedge l < 0 \wedge m < 0 \\ |k| = |l| + |m| \end{cases} \tag{5.16}$$

Proof. According to the Verilog standard [IEE06], if an adder or a multiplier is signed and the output bit-width is large enough to prevent overflow, then the operator would have the function given in Equation (5.14), which is the definition of being generic. □

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

1. If one of the inputs is *unsigned* ($l > 0 \vee m > 0$), then create zero-extension-by-1 signed signal converting functions for both inputs, which is formulated in (5.17). Otherwise, both inputs are signed and remain unmodified.

$$\begin{cases} a_1^{-|l|-1} & = SC(a^l, 0, -|l| - 1) \\ a_2^{-|m|-1} & = SC(b^m, 0, -|m| - 1) \end{cases} \tag{5.17}$$

2. Create the generic operator $op_2 = (o_2^{k_2}, b_1^{l_2}, b_2^{m_2}, t)$, where $b_1^{l_2}$ and $b_2^{m_2}$ are the new inputs from the last step, and $o_2^{k_2}$ is signed and has a large enough bit-width, as defined in Equations (5.15) and (5.16).

3. Create another signal converting function given in (5.18) that replaces the original output o^k.

$$o^k = SC(o_2^{k_2}, k < 0, k) \tag{5.18}$$

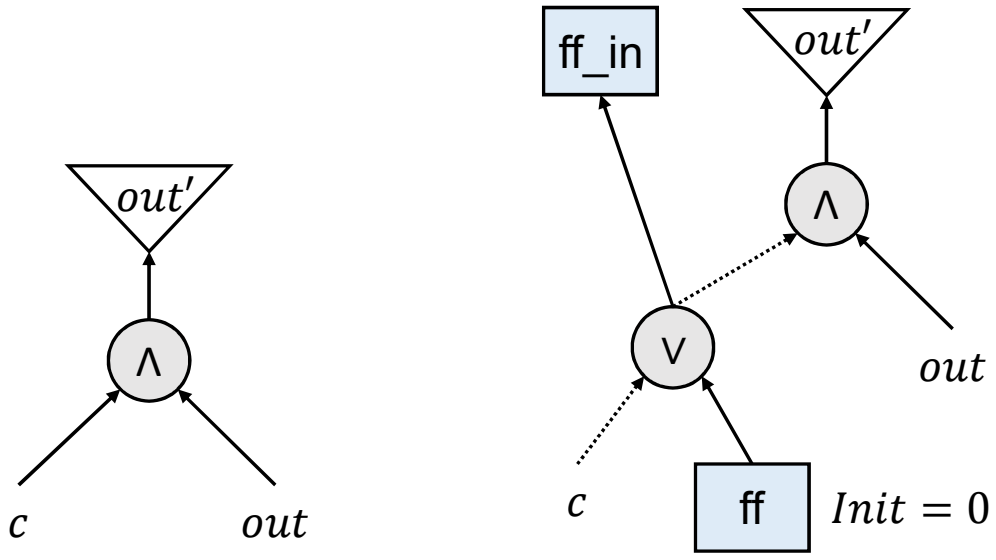
   Note that after this step, the generic operator op_2 is created, and the original operator op is eliminated.

Example 5.8 and Figure 5.2 show how the procedure works. For the non-generic operator $(c^{16}, a^{-16}, b^{16}, *)$, one of the input (b^{16}) is unsigned, so we create signal converting functions for both inputs and get new inputs ag^{-17} and bg^{-17}. Next, the generic operator $(cg^{-34}, ag^{-17}, bg^{-17})$ is created. Finally, the original output is replaced by $SC(cg^{-34}, 0, 16)$.

5.3.3 Creating Abstractions

An abstraction (A) is created from the original circuit (M), using $\mathcal{P}$ and $\mathcal{B}$, the two current states of Algorithm 5.1. 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 (5.12) or (5.13). Signal $c = 1$ implies that a UF constraint is active in



(a) A transformation that works for combinational circuits. A CEX in this circuit makes both $c = 1$ and $out = 1$, meaning that the property is violated when the constraint holds.

(b) A transformation that works for sequential circuits. A CEX in this circuit makes $c = 1$, ff $= 0$, and $out = 1$, meaning that the property is violated when the constraint holds at the current cycle, and the constraint was not violated in previous cycles.

Figure 5.3: An example showing how a UF constraint (signal c) is encoded as an invariant constraint in a circuit. Signal out is the original output where $out = 1$ means the property is violated. Signal out' is the new output with the UF constraint encoded. Dashed arrows denote negations.

M between op_1 and op_2. Signal c is then treated as an invariant constraint, which is illustrated in Figure 5.3. In Figure 5.3b, a new flip-flop (ff) is introduced to remember if the constraint c is violated in previous cycles. If at certain cycle $c = 0$, then it makes ff $= 1$ for all the following cycles. A valid CEX is thus a trace where the property is violated at the current cycle while the constraint holds for all the cycles so far.

2. For each operator $op = (o^k, i_1^l, i_2^m, t)$ in $\mathcal{B}$, replace its output o^k with a fresh primary input ppi^k with the same signedness and bit-width (same k), i.e. black-box it.

Example 5.9. The abstraction in Figure 5.4b is created from the original circuit in Figure 5.4a with the UF pair set $\mathcal{P} = \{(op_1, op_3)\}$ and the abstraction set $\mathcal{B} = \{op_1, op_2, op_3\}$. The outputs of the three multipliers $(c_1^{-4}, c_2^{-4}, c_3^{-4})$ are replaced by the three PPIs $(ppi_1^{-4}, ppi_2^{-4}, ppi_3^{-4})$. A UF constraint for the multiplier pair (op_1, op_3) is created using the original multiplier inputs $\{a_1^{-2}, b_1^{-2}, a_3^{-2}, b_3^{-2}\}$ and the abstracted multiplier outputs $\{ppi_1^{-4}, ppi_3^{-4}\}$, which is formulated

below (following Definition 5.10).

$$\text{UF Constraint} = \left(a_1^{-2}{==}a_3^{-2} \wedge b_1^{-2}{==}b_3^{-2} \Rightarrow ppi_1^{-4}{==}ppi_3^{-4}\right)$$

Note that an operator can be in a pair of $\mathcal{P}$ but not $\mathcal{B}$. Note also that $\mathcal{P}$ and $\mathcal{B}$ are monotone because $\mathcal{P}$ can only increase in size (line 13) and $\mathcal{B}$ can only become smaller (line 17) in each iteration.

Example 5.10. The original circuit in Figure 5.5a is similar to the one in Figure 5.4a; an additional right-side formula is added. The right-side formula is UNSAT because if all the multiplier inputs $\{a_1^{-2}, b_1^{-2}, a_2^{-2}, b_2^{-2}\}$ are not 0, then the multiplier outputs $\{c_3^{-4}, c_4^{-4}\}$ are not 0 either. Therefore this problem is UNSAT. Moreover, this canbe proved using the abstraction shown in Figure 5.5b, which is created with $\mathcal{P} = \{(op_1, op_3), (op_2, op_3)\}$ and $\mathcal{B} = \{op_1, op_2\}$. Note that op_3 is in pairs in $\mathcal{P}$ but not in $\mathcal{B}$.

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

Lemma 5.1. *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,

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

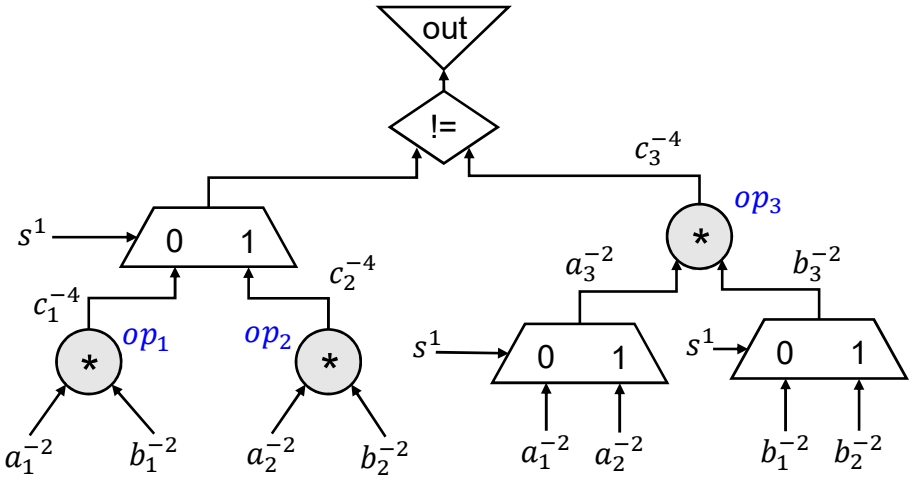
□

Theorem 5.3. *The model A created by* CreateAbstraction *is an abstraction of the miter M.*

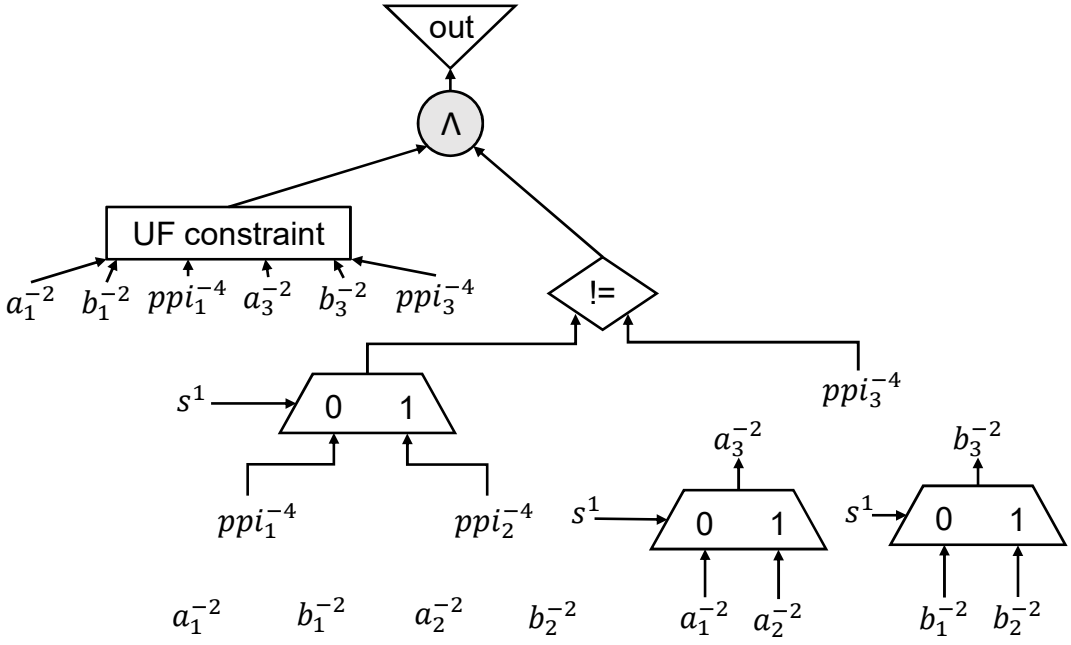
Proof. From Lemma 5.1, N generated by Step 1 is equisatisfiable to the miter M. In Step 2, it creates the model A by replacing some internal signals in N with fresh primary inputs, which is a known procedure for producing an abstraction. □

5.3.4 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

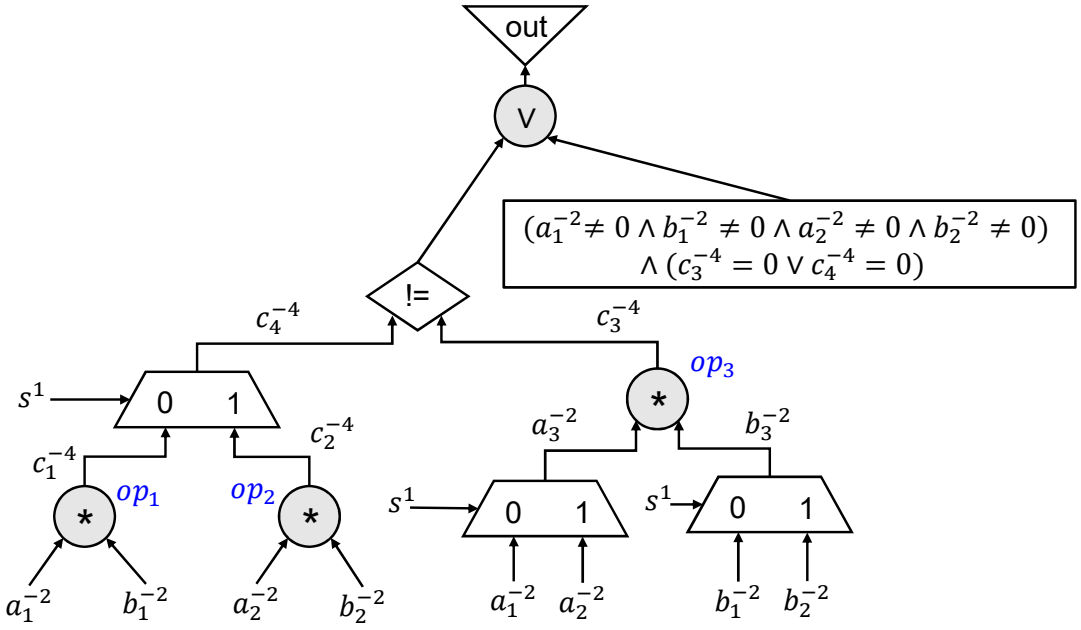


(a) The original circuit with three multipliers. In this circuit, *out* $\equiv 0$, UNSAT, since if $s^1 = 0$, then the left side and the right side are both $a_1^{-2} * b_1^{-2}$; otherwise, both sides are $a_2^{-2} * b_2^{-2}$.

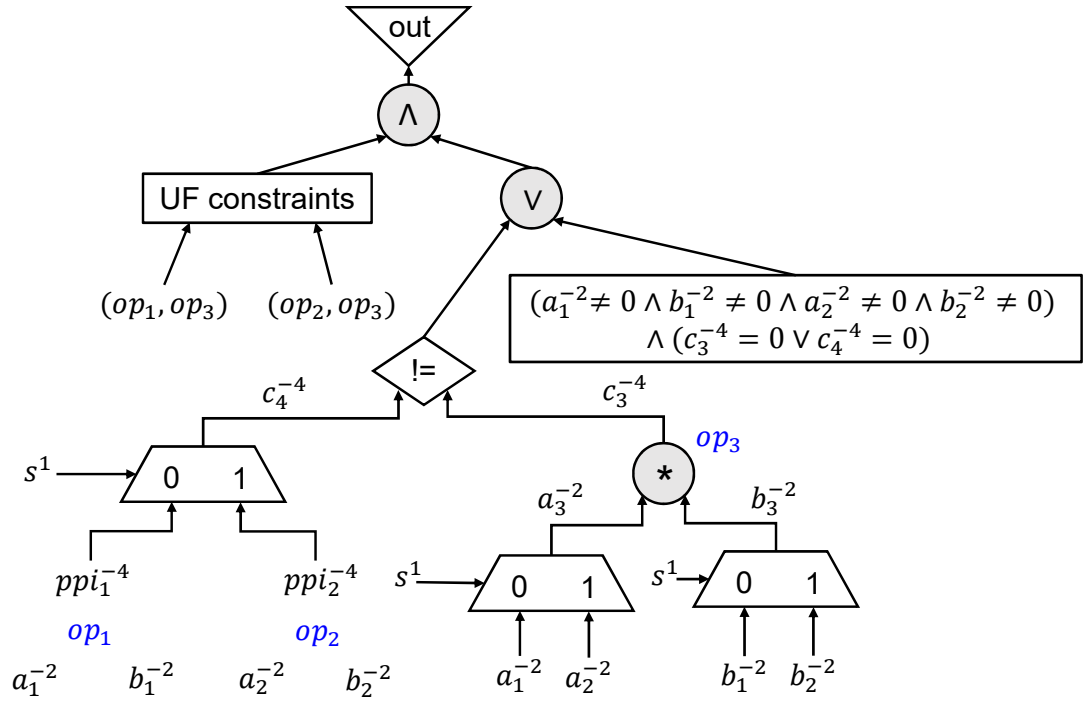


(b) An abstraction created from the original circuit in (a), the UF pair set $\mathcal{P} = \{(op_1, op_3)\}$, and the abstraction set $\mathcal{B} = \{op_1, op_2, op_3\}$. The three PPIs replace the original multiplier outputs. A UF constraint is created between the black-boxed op_1 and op_3.

Figure 5.4: A combinational circuit illustrating word-level UF abstraction.



(a) A circuit similar to the one in Figure 5.4a with an additional property.



(b) An abstraction created from the original circuit in (a), the UF pair set $\mathcal{P} = \{(op_1, op_3), (op_2, op_3)\}$, and the abstraction set $\mathcal{B} = \{op_1, op_2\}$.

Figure 5.5: An example showing UF constraints are useful when applied to real multipliers.

take advantage of different engines. Running a BMC engine in parallel with PDR usually finds counterexamples to the current abstraction more efficiently and thus is very effective in improving the algorithm. Also, various versions (based on different implementations and parameters) of PDR and BMC complement each other.

On the other hand, the current abstraction can be verified also at the word level. For example, our approach, PDR-WLA, presented in Chapter 4 has been shown very effective in handling a set of industrial benchmarks. The results will be presented in Section 5.7.

5.3.5 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 $\alpha : S \times \mathbb{N} \rightarrow \mathbb{Z}$ 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 5.19), then add (op_1, op_2) to $\Delta\mathcal{P}$. Note that both input orders are considered for symmetric operators, although for simplicity this is not shown in Formula 5.19 .

$$\begin{aligned}\big(\alpha(i_{11}^{l_1}, t) = \alpha(i_{21}^{l_2}, t) \wedge \alpha(i_{12}^{m_1}, t) = \alpha(i_{22}^{m_2}, t)\big) \bigwedge \\ \alpha(o_1^{k_1}, t) \neq \alpha(o_2^{k_2}, t)\end{aligned} \tag{5.19}$$

Example 5.11. Consider the abstraction circuit shown in Figure 5.4b. Suppose a spurious CEX is given below.

$$\begin{aligned}(s^1, a_1^{-2}, b_1^{-2}, a_2^{-2}, b_2^{-2}, ppi_1^{-4}, ppi_2^{-4}, ppi_3^{-4}) = \\ (1, 00, 00, 00, 00, 0000, 1111, 0000).\end{aligned}$$

The CEX is simulated on the abstraction to derive the values of inputs and outputs for the three multipliers.

$$\begin{aligned}op_1 : (ppi_1^{-4}, a_1^{-2}, b_1^{-2}) = (0000, 00, 00) \\ op_2 : (ppi_2^{-4}, a_2^{-2}, b_2^{-2}) = (1111, 00, 00) \\ op_3 : (ppi_3^{-4}, a_3^{-2}, b_3^{-2}) = (0000, 00, 00)\end{aligned}$$

Observe that there are two pairs satisfying Formula 5.19: (op_1, op_2) and (op_2, op_3). Therefore, both pairs are added to $\Delta\mathcal{P}$ and the UF pair set is updated as

$$\mathcal{P} = \{(op_1, op_3), (op_1, op_2), (op_2, op_3)\}.$$

The size of $\mathcal{P}$ can only grow monotonically with an upper bound as:

Theorem 5.4. *The size of $\mathcal{P}$ in Algorithm 5.1 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 $|\mathcal{S}|(|\mathcal{S}|-1)$. □

5.3.6 Refining Black Operators

In the second phase of refinement, the goal is 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 has been found often to create overly large $\Delta\mathcal{B}$, resulting in unnecessarily large abstractions in the next round. Hence, a *proof-based* approach is used that often obtains a much smaller $\Delta\mathcal{B}$. This approach is similar to PBR discussed in Chapter 2.

The main idea is that if *cex* is spurious, then the BMC Formula (5.20) is UNSAT. Here the functions $\beta(i,t)$ and $\beta(s,t)$ denote the assignment of input i or state s 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.

$$\begin{aligned} I_M(\beta(s,0)) \wedge \bigwedge_{t=0}^{k-1} T_M(\beta(i,t),\beta(s,t),\beta(s,t+1)) \\ \wedge \bigvee_{t=0}^{k} out(\beta(i,t),\beta(s,t)) \end{aligned} \tag{5.20}$$

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 [ES03a] will return a subset of the assumptions that is sufficient for UNSAT. This is a variation of finding an estimate of the minimum UNSAT subset, as discussed in Section 2.2.6. The returned subset can be further minimized with extra SAT calls. However, in our experience, such minimization does not lead to overall improved performances because of the overhead. Therefore, the subset returned by a SAT solver 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 $op = (o^k, i_1^l, i_2^m, t)$ in $\mathcal{B}$, introduce two fresh primary inputs, *sel* and *ppi*, where *sel* is a Boolean signal and ppi^k a bit-vector signal which is consistent with the output o^k. Replace o^k with $o_2^k = ITE(sel, o^k, ppi^k)$ where ITE is the *if-then-else* operator. Depending on the value of *sel*, either the concrete operator (o^k) or the abstracted one (ppi^k) flows to the new output o_2^k.

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 (5.21), 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 (5.21) is reduced to (5.20) by construction ($sel = 1$ means that the concrete version is chosen).

$$\begin{aligned} I_N(\gamma(s,0)) \wedge & \bigwedge_{t=0}^{k-1} T_N(\gamma(i,t), X_t, PPI_t, s_t, s_{t+1}) \\ \wedge & \bigvee_{t=0}^{k} out(\gamma(i,t), X_t, PPI_t, s_t) \\ \wedge & \bigwedge_{t=0}^{k} \bigwedge_{n=0}^{|X_t|} x_{tn} \end{aligned} \tag{5.21}$$

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

Example 5.12. Consider the abstraction circuit shown in Figure 5.4b. Suppose we get a spurious CEX and try to refine this abstraction by white-boxing some of the three black-boxed multipliers. To take advantage of the assumption interfaces in modern SAT solvers, an auxiliary circuit is created using the proposed procedure, as shown in Figure 5.6. The first step is to formulate UF constraints for the current set $\mathcal{P}$. This step is important because the number of white boxes needed can be smaller than without using those constraints. Next, three ITE operators are introduced for each multiplier in the current set $\mathcal{B}$ to select between concrete multipliers (c_i^{-4}) and abstracted multipliers (ppi_i^{-4}) using select signals (x_i^1). Then,

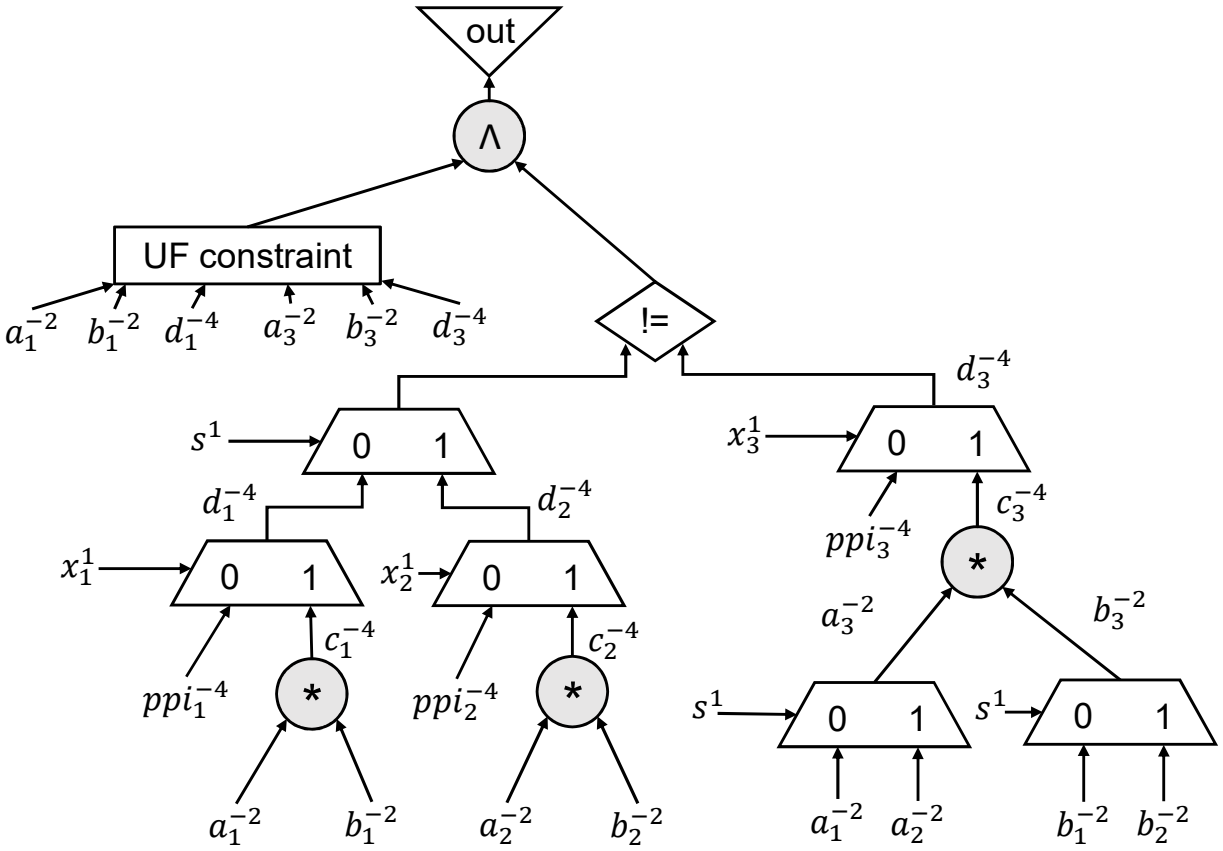


Figure 5.6: An auxiliary circuit created in the proposed proof-based procedure for refining black operators. The original and the current abstraction circuits are shown in Figure 5.4.

if the concrete values are used in the CEX for the real PIs ($\{a_1^{-2}, b_1^{-2}, a_2^{-2}, b_2^{-2}\}$) and the SAT solver is called with $out = 1$ and with assumptions:

$$\textbf{\textit{SolveSAT}}(x_1 \wedge x_2 \wedge x_3),$$

then it must return UNSAT. This is because by assuming the $x_1 = x_2 = x_3 = 1$, the circuit is reduced to the original (Figure 5.4a) with UF constraints added; a spurious CEX will make $out = 0$ by definition. The SAT solver will return a subset of the assumptions for $\{x_1, x_2, x_3\}$ that is sufficient to make $out = 0$, which is our candidate to white-box.

Theorem 5.5. *The set $\Delta\mathcal{B}$ found by* RefineBlack *is not empty (i.e.$|\Delta\mathcal{B}| > 0$).*

Proof. $|\Delta\mathcal{B}| = 0$ would mean that the formula (5.21) is SAT, which contradicts that *cex* is spurious. □

5.3.7 Analysis of UFAR

Theorem 5.6. *Algorithm 5.1 is sound and complete.*

Proof. (sketch) Algorithm 5.1 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 5.3. 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.

- Either $\mathcal{P}$ becomes strictly bigger (line 12–13) or $\mathcal{B}$ becomes strictly smaller (Theorem 5.5).
- $|\mathcal{P}|$ is upper bounded by $|\mathcal{S}|(|\mathcal{S}|-1)$ (Theorem 5.4) 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. □

5.4 Improvement Techniques

In this section, we introduce two improvement techniques (counterexample minimization, and random simulation), each of which improves the basic version of UFAR, Algorithm 5.1.

5.4.1 Minimizing Counterexamples

A counterexample can be minimized into an XCEX (Section 2.2.5) using *Priority-based Refinement* proposed by Mishchenko et al. [MEB+13]. Some inputs in a counterexample can be assigned as X (unknown value), but the counterexample still violates the property after ternary simulation. The XCEX returned by Priority-based Refinement is minimal in the sense that if a concrete assignment is replaced by X, then the counterexample no longer makes the property fail. Therefore, we say that the number of concrete assignments is minimized. More details are presented in Section 2.3.

The main advantage of using minimized counterexamples is that Procedure RefineUFPairs in Algorithm 5.1 can return potentially fewer, but higher-quality pairs of constraints. This is done by modifying the condition (Formula 5.19) 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* otherwise. For example, the assignments `XX01` and `X000` are incompatible while `10XX` and `100X` are compatible. With this strengthening, pairs that satisfy Formula 5.19 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 5.19. 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.

Example 5.13. Consider the original circuit (M) shown in Figure 5.4a. Suppose an abstraction is created from M with $\mathcal{P} = \emptyset$ and $\mathcal{B} = \{op_1, op_2, op_3\}$. Suppose a spurious CEX is given below.

$$(s^1, a_1^{-2}, b_1^{-2}, a_2^{-2}, b_2^{-2}, ppi_1^{-4}, ppi_2^{-4}, ppi_3^{-4}) = \\ (\texttt{1}, \texttt{00}, \texttt{00}, \texttt{00}, \texttt{00}, \texttt{0000}, \texttt{1000}, \texttt{1010}).$$

Without using CEX minimization, there are three multiplier pairs with idential inputs but different outputs: $\{(op_1, op_2), (op_1, op_3), (op_2, op_3)\}$. However, since $s^1 =$ `1` (op_2 is selected instead of op_1), adding pairs (op_1, op_2) and (op_1, op_3) does not help block counterexamples in the next iteration. This issue can be addressed by using a minimized CEX:

$$(s^1, a_1^{-2}, b_1^{-2}, a_2^{-2}, b_2^{-2}, ppi_1^{-4}, ppi_2^{-4}, ppi_3^{-4}) = \\ (\texttt{1}, \texttt{XX}, \texttt{XX}, \texttt{XX}, \texttt{XX}, \texttt{XXXX}, \texttt{XX0X}, \texttt{XX1X}).$$

From the minimized CEX, there is only one pair with compatible inputs and incompatible outputs, which is (op_2, op_3). In this example, using minimized counterexamples reduces the number of refined UF constraints by two. This heuristic is important especially for test cases with many candidate operators and pairs.

5.4.2 Performing Random Simulation

UFAR in Algorithm 5.1 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. A similar idea of using random simulation to identify candidates for UF abstraction was proposed by Brady et al. [BBSO10].

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. In our experience, a threshold of 10% seems to work well in practice.

Example 5.14. Consider the original circuit shown in Figure 5.4a. If we run random simulation on the circuit, the resulting input patterns for the pairs of the three multipliers can be

- (op_1, op_2): Their inputs are identical for about 0% of the time, since $\{a_1^{-2}, b_1^{-2}, a_2^{-2}, b_2^{-2}\}$ are four independent PIs.

- (op_1, op_3): Their inputs are identical for about 50% of the time, since (a_1^{-2}, b_1^{-2}) and (a_3^{-2}, b_3^{-2}) are identical if $s^1 = \mathtt{0}$.

- (op_2, op_3): Their inputs are identical for about 50% of the time, since (a_2^{-2}, b_2^{-2}) and (a_3^{-2}, b_3^{-2}) are identical if $s^1 = \mathtt{1}$.

If the threshold is set to be 10%, then the pairs (op_1, op_3) and (op_2, op_3) would be added to the UF pair set $\mathcal{P}$, and the resulting abstraction can be proved UNSAT. In this example, we use random simulation to identify good candidates of UF constraints without relying on counterexamples found in the CEGAR flow.

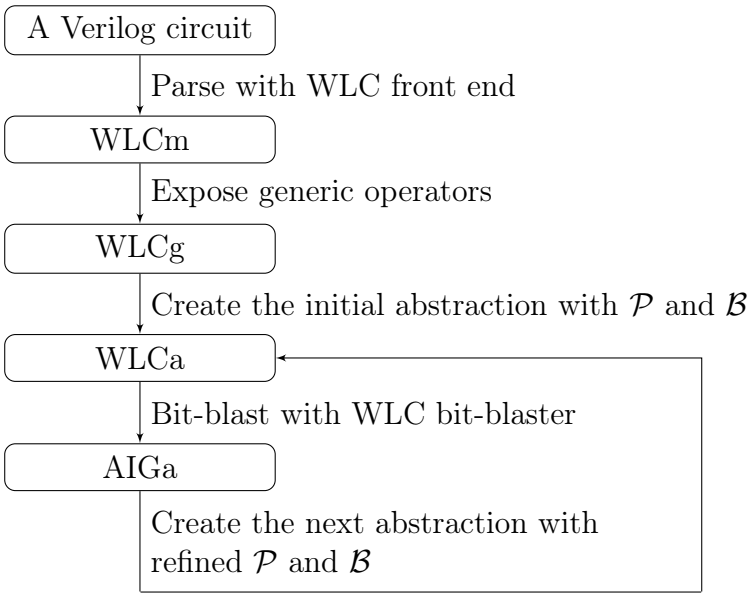


Figure 5.7: The flow of the UFAR framework

5.5 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 [BM10] 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.

5.5.1 Bit-blasting WLC with Verilog Semantics

As shown in Figure 5.7, 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 (see Section 5.2) 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 [IEE06] 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.

5.5.2 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.

5.6 Related Work

In this section, UFAR is compared with other word-level approaches based on *term-level abstraction* (TLA) [Hun89, BD94, BLS02, LSB02, LB03, AS04, ALS08, BBSO10, BBS11]. Term-level abstraction employs three abstraction techniques: 1) function abstraction, 2) data abstraction, and 3) memory abstraction. In this section, we focus on function abstraction, which features UF abstraction. In term-level abstraction, UF abstraction is performed by replacing a concrete function with a UF symbol. Same function instances share the same UF symbol and are constrained by the same functional consistency, meaning identical inputs implies identical values for the symbol. An abstraction created with term-level abstraction is then given to a dedicated word-level solver that implements specialized procedures to handle those UF symbols [Ack54, BGV99]. In contrast, UFAR performs UF abstraction with explicit

application of UF constraints guided by counterexamples and random simulation. This is greatly enhanced by our ability to efficiently create word level netlists like WLCa and bit blasted versions like AIGa in an iterative loop. The main benefit of using UF constraints is that abstractions can be verified by *any* model checker, bounded or unbounded, thereby taking full advantage of all the recent developments at the bit-level and the word-level.

Other differences between UFAR and TLA-based approaches are given below.

1. UFAR addresses an important challenge when performing UF abstraction: the applicability of UF constraints. As shown in Example 5.4, multipliers can have different functions, which makes UF inapplicable if the functions are not identical. UFAR tackles this problem by exposing generic operators within regular operators, as presented in Section 5.3.2. UF constraints then become applicable to any pair of same-type operators. In contrast, none of the TLA-based approaches can deal with this problem. Without generic operators, UF abstractions can be performed only under strict conditions: 1) instantiations of the same module, 2) replicated functional blocks [BBS11], or 3) operators with exactly matched bit-widths and signedness, which limits the capability of UFs to abstract a circuit. However, TLA-based approaches could be extended to benefit from UFAR if generic operators are exposed and normalized to the one with the maximum size.

2. TLA-based approaches all rely on SMT solvers to handle formulas with UF abstractions, which limits the possibility of integrating them with recent bit-level developments like IC3/PDR and its improvements. TLA-based approaches rely on Bounded Model Checking (BMC) [BCCZ99] and/or k-induction [SSS00]. This not only limits its use but also becomes inefficient when deep unrolling is needed. In practice, BMC- and induction- based approaches are efficient in finding CEXes, but often incapable of producing an inductive invariant, which is required for UMC problems. In contrast, UFAR explicitly formulates UF constraints when creating an abstraction circuit, which can be the target for proof by any UMC techniques like IC3/PDR and `super_prove` [BEM12]. Moreover, a straightforward application of UF constraints can have the problem of explosion in the number of operator pairs that are constrained. This problem is mitigated by UFAR with counterexample guided refinement and random simulation.

3. TLA-based approaches use UF symbols instead of UF constraints, which limits their capability to take full advantage of UF abstractions. For instance, in Example 5.10, the problem can be proved by UFAR with only two multipliers abstracted. The key is to apply UF constraints even to a pair of multipliers where one is abstracted and the other concrete. In contrast in the example, TLA-based approaches cannot abstract any multiplier using UF symbols, since there always exists a spurious CEX when replacing any subset of the three multipliers with UF symbols. In particular in the example, UF constraints must be applied to both pairs (op_1, op_3) and (op_2, op_3), and if TLA-

based approaches replace all three multipliers with the same UF symbol, then spurious CEXes exist because of the right-hand-side formula demanding certain properties of multiplication. UFAR can apply UF constraints to any combination of abstracted and concrete operators: 1) (abstracted, abstracted), 2) (abstracted, concrete), and 3) (concrete, concrete). Therefore, UFAR offers more flexibility in UF abstractions.

5.7 Experimental Results

In this section, we present experimental results of our implementation of UFAR with different improvement techniques enabled. We also integrated the the method PDR-WLA (Chapter 4) into the UFAR framework. The implementation is based on ABC [BM10] 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 and optimized by an industrial tool to be cycle-accurate with the original circuit. Multipliers are the main targeted problematic operators for UFAR to abstract. Thus, the initial abstraction set $\mathcal{S}$ contains all the exposed generic multipliers in a circuit. 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 other publicly available verification tools is difficult. To our knowledge, no tools exist that can parse such designs directly without requiring a major modification. For example, Ebmc [KP] cannot handle parameterized modules or functions/tasks in Verilog; VCEGAR [JKSC05] has a more limited front-end than the one in Ebmc; UCLID [ucl, BLS02, LS04] does not have a general Verilog front end either. Also, there is no standard format for sequential word level circuits, as there is for the combinational case with SMT-LIB [BST10]. Therefore we compared results of running

1. `super_prove` (SP) [BEM12]: `super_prove` is a portfolio-based bit-level model checker that won the last 5 Hardware Model Checking Competitions (HWMCC), the latest being 2017 [BvDH17]. An input word-level circuit is parsed and bit-blasted into And-Invertor Graphs (AIGs) by ABC, and then solved by `super_prove`, and

2. five UFAR versions with different settings.

   - ufar-S1 (S1): It is the baseline version implementing Algorithm 5.1.

   - ufar-S2 (S2): It features the technique of using minimized counterexamples (Section 5.4.1).

   - ufar-S3 (S3): It features the technique of using random simulation (Section 5.4.2).

The parameters for random simulation are set to be (1) 64 random patterns, (2) 100 time frames, (3) 10% threshold.

- ufar-S4 (S4): It features both techniques in ufar-S2 and ufar-S3.
- ufar-S5 (S5): It features both techniques and the PDR-WLA word-level model checker presented in Chapter 4.

Abstractions in UFAR are verified using various verification engines that were run in parallel. All settings (S1-S5) bit-blast current abstraction circuits and verify them with the following four bit-level engines.

(a) Command "`bmc3`" in ABC. A BMC engine that is very efficient at finding counterexamples.

(b) Command "`pdr`" in ABC. A PDR engine that is good at proving a UMC problem with an inductive invariant.

(c) Command "`,treb`" in ABC-ZZ [Een]. A PDR engine developed by Een et al. in [EMB11].

(d) Command "`,treb -abs`" in ABC-ZZ. An implementation of PDR with internal abstraction developed by Niklas Een. We present a similar algorithm in Chapter 3.

Setting S5 also runs the word-level PDR-WLA engine in parallel with the four bit-level ones.

We present the results in Figure 5.8, 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. All solved instances are UNSAT. The result of `super_prove` is not shown in Figure 5.8 because its number of solved instances is 2087, well below the bottom scale of 2330. Setting S2 performs worse than the baseline S1, which shows one weakness of using minimized counterexamples: while this technique can reduce the number of UF constraints added (Section 5.4.1), it can also miss important constraints, which leads to more iterations of refinement and results in worse performance. Setting S3 is slightly better than the baseline S1, which demonstrates that random simulation does find important UF constraints that can be missed by counterexamples, thereby improving performance. Setting S4 outperforms the three previous settings, which shows that the combination of random simulation and minimized counterexamples achieves a good balance of what UF constraints should be added to abstractions. Setting S5 works best with the additional PDR-WLA solver, which demonstrates that UFAR can easily benefit from any improvement in UMC algorithms at both bit-level and word-level.

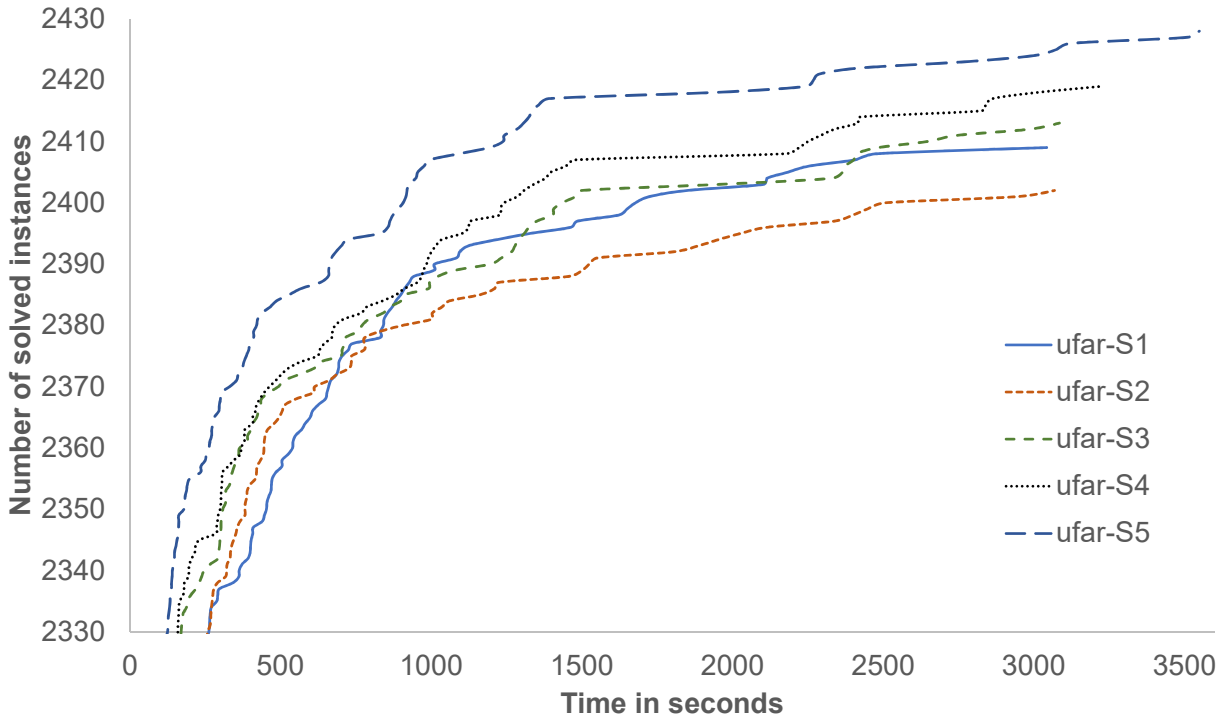


Figure 5.8: Comparison of five UFAR variants. The result of `super_prove` is not shown here because it only solves 2087 instances, well below the bottom scale of 2330.

Table 5.5 shows the numbers of instances finally solved by all versions within the 1-hour time-out. The five versions of UFAR outperform `super_prove`, which is often ineffective in solving problems with many arithmetic operators.

We selected 100 out of 2492 designs to present more detailed results in Table 5.2, Table 5.3, and Table 5.4. The selection is somewhat arbitrary but it does represent designs that are dissimilar and gives an idea of the expected ranges of iterations needed, the number of UF constraints used, and the number of white-box operators existing in the final abstractions. We observe the following from the four tables.

1. UFAR proves most cases with a relatively small number of white-box multipliers in the final abstraction. For example, Designs 78-100 in Table 5.3 contain more than 248 multipliers originally, but no more than 44 white boxes are needed for the final proofs.

2. The number and quality of UF constraints are two important factors correlated with the runtime performance. If the number is large, then UFAR generally needs more time to complete, which is why counterexample-based constraint reduction is important. If the quality of the reduction is good, then UFAR may prove a problem with fewer white boxes (or none). This supports the use of random simulation to find good constraints.

3. The technique of using minimized counterexamples does reduce the number of final UF constraints significantly, as shown in Table 5.2 and Table 5.3. While this technique does not improve the runtime performance of S2 compared with S1, it does improve the runtime performance of S4 over S3 by preventing the addition of too many UF constraints, as shown in Designs 79, 81, and 87 in Table 5.3.

4. The technique of using random simulation helps find important UF constraints or related operator pairs without needing information derived from counterexamples. With those constraints found by random simulation, the overall number of UF constraints and white boxes used can drop, resulting in better runtime performance, as shown in Designs 10, 68, 89 in columns of S1 and S3.

5. As shown in Table 5.4, it takes a nontrivial number of refinements for UFAR to converge, implying that UFAR builds up abstractions gradually. A major challenge is how to strike a better balance between the number and quality of UF constraints and the number of white boxes needed.

6. Table 5.5 shows the progress made in UFAR over its development cycle. It is important to comment that the increment improvements are 320, -7, 11, 6, and 8, additional problems solved may seem small, but the remaining problems are extremely hard, and the complexity of model checking dictates that we probably can never solve all of them.

Table 5.2: Detailed experimental results for the first 50 out of the 100 word-level UNSAT test-cases that can be solved by at least one of the six verification settings (the last 50 are shown in the next table). The #Mults/#ANDs/#FFs means the number of multipliers/bit-level AND nodes/bit-level flip flops. The numbers of UF constraints and white boxes used in the last iteration are also presented. Blanks in CPU Time represent time-outs (1 hour).

ID	#Mults	#ANDs	#FFs	CPU Time (seconds)						Number of UF Constraints					Number of White Boxes				
				SP	ufar-S1	ufar-S2	ufar-S3	ufar-S4	ufar-S5	S1	S2	S3	S4	S5	S1	S2	S3	S4	S5
1	4	33062	977	**2**		2449	2484	2424	23	10	8	10	8	0	0	1	1	1	1
2	4	33080	977	**1**		1163		1130	49	6	0	6	0	6	0	1	0	1	1
3	4	32968	975	**2**		3069		3012	76	7	6	7	6	0	0	1	0	1	1
4	60	187389	2608	568	505	552	299	**293**	297	0	0	874	874	874	0	0	0	0	0
5	92	239442	2608	11	**5**	**5**	563	526	529	0	0	2052	2052	2052	0	0	0	0	0
6	60	38900	472		**2102**					1300	63	2084	2084	2084	18	17	23	23	23
7	60	44989	634		**2479**		3085	2827	2999	2155	34	1420	1420	1420	14	13	18	18	18
8	60	44056	648						**3510**	1671	64	1546	1546	1546	21	18	21	21	21
9	60	52538	681		1651		1403	1379	**1307**	1595	34	2692	2725	2725	11	12	9	15	15
10	11	82237	128			2338	**1957**	3226	3119	24	42	12	38	38	9	9	9	9	9
11	16	148910	179		22		20	21	**19**	36	0	12	12	12	0	16	0	0	0
12	102	60052	508		87	**59**	69	106	101	998	84	1320	1313	1313	0	0	0	0	0
13	144	161864	2456		1463	2116	699	**645**	684	2234	91	352	352	352	0	0	0	0	0
14	20	78142	551	108	8	**7**	11	11	11	82	14	190	190	190	0	0	0	0	0
15	6	161192	5627	447	484	323	434	463	**234**	10	2	4	4	4	0	0	0	0	0
16	8	73021	2056		**599**	609	1351	1016	996	14	11	24	14	14	0	0	0	0	0
17	8	72903	2051		667	**661**	713	709	669	14	11	11	11	11	0	0	0	0	0
18	14	21156	400		453	417	123	**116**	120	30	28	25	25	25	10	8	8	8	8
19	14	21159	400		447	446	**295**	380	379	29	28	29	33	33	10	8	8	8	8
20	16	84465	3899		52	39	33	33	**23**	96	165	8	8	8	0	0	0	0	0
21	32	141251	3917		215	135	150	161	**134**	913	167	16	16	16	0	0	0	0	0
22	32	46246	972	**3**	2184		2331	2176	2269	271	28	240	224	224	9	2	9	7	7
23	40	108487	1003		54	**49**	59	51	**49**	326	405	510	344	344	20	22	22	22	22
24	24	69002	685		34	23	35	19	**18**	140	48	453	126	126	6	6	6	6	6
25	86	213795	2423		206	**135**	189	954	920	1423	210	2377	502	502	16	16	16	28	28
26	40	73448	688		56	71	**42**	50	49	626	259	312	248	248	22	22	22	23	23
27	24	69002	685		23	**21**	58	28	25	233	71	403	98	98	6	6	6	6	6
28	86	213967	2417		372	**113**	170	145	145	4552	146	777	331	331	40	16	29	16	16
29	40	108493	1003		34	**33**	39	36	35	340	147	172	106	106	10	10	10	10	10
30	24	69003	685		44	25	**19**	30	29	417	89	542	126	126	6	6	6	6	6
31	40	108493	1003		46	35	**32**	41	33	361	233	226	161	239	16	10	16	14	10
32	40	6771	134		118	**47**	60	63	59	656	273	750	100	100	22	22	16	22	22
33	40	108487	1003		**30**	50	41	**30**	43	414	522	192	359	389	22	22	16	19	22
34	40	106576	863		28	23	27	25	**20**	518	217	90	262	132	10	17	16	10	16
35	43	306429	4408		20	14	14	13	**11**	274	13	447	447	447	0	0	0	0	0
36	6	11597	99	**4**					18	0	0	3	3	2	6	6	6	6	6
37	6	11626	99	**5**					31	0	0	2	2	2	6	6	6	6	6
38	6	11379	99	**4**					32	0	0	2	2	2	6	6	6	6	6
39	6	11626	99	**5**					27	1	0	3	2	2	6	6	6	6	6
40	12	17589	243		218	217	246	218	**138**	5	0	5	0	0	11	7	11	7	7
41	15	16522	204		173	102	105	107	**90**	1	0	0	0	0	10	10	10	10	10
42	16	74935	3770		362	333	239	**232**	236	69	183	8	8	8	0	0	0	0	0
43	15	49769	1707		22		**14**	22	18	72	27	50	57	57	7	8	7	7	7
44	21	65955	1997		57		**40**	72	67	172	31	59	86	86	7	10	7	9	9
45	5	17951	265	596	55	44	57	53	**30**	2	7	2	2	2	0	0	0	0	0
46	4	16660	336	554	111	258	87	84	**83**	10	2	1	1	1	0	0	0	0	0
47	10	15485	204		**25**	26	26	**25**	26	0	0	0	0	0	10	10	10	10	10
48	42	56591	705		**87**	233	107	110	108	219	212	145	145	145	14	12	12	12	12
49	54	47963	383		289	422	389	134	**127**	542	244	267	262	262	24	22	25	29	29
50	56	54250	419			1889	1479	1479	**1391**	1096	222	268	268	268	25	32	31	31	31

Table 5.3: Detailed experimental results for the last 50 out of the 100 word-level UNSAT test-cases that can be solved by at least one of the six verification settings. The #Mults/#ANDs/#FFs means the number of multipliers/bit-level AND nodes/bit-level flip flops. The numbers of UF constraints and white boxes used in the last iteration are also presented. Blanks in CPU Time represent time-outs (1 hour).

ID	#Mults	#ANDs	#FFs	CPU Time (seconds)						Number of UF Constraints					Number of White Boxes				
				SP	ufar-S1	ufar-S2	ufar-S3	ufar-S4	ufar-S5	S1	S2	S3	S4	S5	S1	S2	S3	S4	S5
51	56	52535	371			2027	298	291	**265**	736	110	268	268	268	30	34	33	33	33
52	63	69913	470		**694**		2373	2294	2288	888	238	436	436	436	36	22	29	29	29
53	63	58991	461		1623	1002	913	968	**856**	725	278	375	375	375	17	20	35	35	35
54	56	48639	366			1525	704	673	**660**	651	162	268	268	268	33	32	30	30	30
55	56	53791	406		1126		**332**	345	907	802	120	348	348	332	26	32	28	28	34
56	63	52933	479		**842**		1435	1308	1336	435	149	346	346	346	21	24	17	17	17
57	56	60014	395		912		**340**	630	663	640	139	284	270	270	24	23	21	16	16
58	63	69913	470		**738**		2366	2339	1942	888	238	436	436	436	36	22	29	29	29
59	63	65458	436		**871**		1407	1397	1325	435	149	346	346	346	21	24	17	17	17
60	42	20109	334		225	115	115	**99**	102	292	89	145	145	145	14	11	9	9	9
61	223	237517	1130		2256		2648	2840	**1239**	3858	481	2123	1654	1280	41	63	54	49	42
62	63	64414	443		**855**		1325	1340	1243	435	149	346	346	346	21	24	17	17	17
63	56	59561	417		1486	1954	318	307	**302**	810	251	268	268	268	31	32	35	35	35
64	49	49557	403		262	74	**71**	72	72	478	94	204	204	204	11	11	9	9	9
65	56	52757	416		550	**352**	757	670	703	816	196	268	268	268	32	33	34	34	34
66	63	66328	464		1724	1062	1023	1036	**949**	725	278	375	375	375	17	20	35	35	35
67	63	69913	470		**726**		2403	2251	2246	888	238	436	436	436	36	22	29	29	29
68	56	47817	391			728	421	**382**	395	397	183	268	268	268	30	28	16	16	16
69	15	51293	1360		943	**218**	1266	622	632	57	5	73	73	73	4	4	3	3	3
70	68	37874	1174		11		**3**	4	**3**	375	119	30	30	30	0	34	0	0	0
71	216	35097	521		899		408	408	**373**	1023	1795	394	394	394	0	0	0	0	0
72	68	34275	663		7		**3**	**3**	**3**	381	110	30	30	30	0	34	0	0	0
73	12	10569	86		844	131	104	81	**80**	16	16	18	13	13	10	4	4	4	4
74	30	20790	296				901	912	**360**	110	144	81	33	53	22	16	16	8	6
75	6	6442	196		20	20	33	24	**17**	1	1	4	3	3	4	4	4	4	4
76	14	14056	316		71	**49**	130	63	84	49	12	6	7	6	5	5	6	4	6
77	7	62203	373		**10**	318	**10**	**10**	**10**	10	7	21	21	21	3	3	4	4	4
78	283	181158	3949						**3549**	35603	23800	26799	21279	27218	0	40	0	0	44
79	248	174145	3491		3832	**9**		974	977	16038	96	14403	9439	9439	7	8	0	5	5
80	253	177932	3979		144	**66**	129	130	100	2767	33	115	115	115	0	0	0	0	0
81	248	173543	3466			**8**		1286	891	15037	65	14644	9701	9397	0	3	0	6	7
82	467	255665	3691		654	383	167	165	**129**	18142	14326	249	249	249	0	0	0	0	0
83	283	181158	3949						**3075**	37975	23800	26154	27241	27236	0	40	0	0	44
84	259	181198	4095		35	**6**	11	9	13	7799	40	124	124	124	0	0	0	0	0
85	253	176313	3979		**23**	1211	**23**	**23**	**23**	2379	34	115	115	115	0	2	0	0	0
86	467	268906	4041		1330	1224	331	327	**200**	12461	24982	249	249	249	0	0	0	0	0
87	251	168447	3526		36	**6**		32	29	7091	28	7578	4946	4946	0	0	0	0	0
88	456	250828	3526		11	**3**	21	14	22	4527	73	215	215	215	0	0	0	0	0
89	250	190268	3740			**5**	715	682	658	19044	32	13774	10850	10850	0	1	1	1	1
90	250	191087	3749		58	**8**	1309	884	830	5834	51	14637	10853	10853	2	2	4	6	6
91	283	180535	3949		**267**	275	630	303	271	0	0	98	98	98	0	0	0	0	0
92	283	181158	3949						**2769**	35603	23836	26073	21268	26936	0	0	0	0	44
93	253	199513	3751		883	**10**	2746	1125	1093	56667	80	22295	11650	11650	4	4	3	9	9
94	255	173230	3508			**31**				11134	40	23828	33268	33268	0	0	0	0	0
95	454	247975	3526		105	**76**	123	118	119	906	12	214	214	214	0	0	0	0	0
96	250	190436	3740		704	**5**		2221	1358	17396	39	17035	31064	30829	2	1	0	2	1
97	249	167395	3508		139	**8**	354	88	71	13676	11	14296	21647	21647	0	0	0	0	0
98	413	253390	4294	765	125	**36**	55	49	51	10621	16	157	157	157	0	0	0	0	0
99	475	274840	4204		343	479	203	198	**191**	8740	19993	268	268	268	0	0	0	0	0
100	259	175172	3526		194	**30**				17286	58	9590	11685	11685	0	0	0	0	0

Table 5.4: Detailed experimental results for the 100 word-level UNSAT test-cases that can be solved by at least one of the six verification settings. The numbers of iterations of applying new UF constraints and iterations of applying new white boxes in UFAR are presented.

(a)

ID	#Iterations for UF Constraints					#Iterations for White Boxes				
	S1	S2	S3	S4	S5	S1	S2	S3	S4	S5
1	1	1	1	1	0	0	1	1	1	1
2	1	0	1	0	1	0	1	0	1	1
3	2	1	2	1	0	0	1	0	1	1
4	0	0	1	1	1	0	0	0	0	0
5	0	0	1	1	1	0	0	0	0	0
6	6	11	2	2	2	3	3	6	6	6
7	5	6	2	2	2	2	2	6	6	6
8	8	12	3	3	3	6	4	8	8	8
9	6	5	2	3	3	3	1	3	3	3
10	2	2	2	2	2	1	1	1	1	1
11	3	0	1	1	1	0	1	0	0	0
12	3	4	2	2	2	0	0	0	0	0
13	6	10	1	1	1	0	0	0	0	0
14	1	1	1	1	1	0	0	0	0	0
15	1	1	1	1	1	0	0	0	0	0
16	1	2	2	2	2	0	0	0	0	0
17	1	2	2	2	2	0	0	0	0	0
18	3	3	2	2	2	1	1	1	1	1
19	3	3	2	3	3	1	1	1	1	1
20	5	2	1	1	1	0	0	0	0	0
21	2	2	1	1	1	0	0	0	0	0
22	5	4	3	2	2	2	1	2	2	2
23	10	4	5	8	8	1	1	1	1	1
24	7	2	5	2	2	1	1	1	1	1
25	4	4	8	4	4	1	1	1	2	2
26	4	10	4	4	4	1	1	1	1	1
27	4	5	19	4	4	1	1	1	1	1
28	3	8	6	5	5	2	1	1	1	1
29	3	3	4	3	3	1	1	1	1	1
30	13	1	2	2	2	1	1	1	1	1
31	4	4	5	7	4	1	1	1	1	1
32	2	4	2	4	4	1	1	1	1	1
33	2	2	5	3	5	1	1	1	1	1
34	4	3	3	3	4	1	1	1	1	1
35	1	1	1	1	1	0	0	0	0	0
36	0	0	2	2	1	1	1	2	1	2
37	0	0	1	1	1	1	1	1	1	1
38	0	0	1	1	1	1	1	1	1	1
39	1	0	2	1	1	1	1	1	1	1
40	1	0	1	0	0	2	1	2	1	1
41	1	0	0	0	0	1	1	1	1	1
42	6	2	1	1	1	0	0	0	0	0
43	2	3	2	3	3	1	1	1	1	1
44	3	2	3	4	4	1	2	1	1	1
45	1	2	1	1	1	0	0	0	0	0
46	1	1	1	1	1	0	0	0	0	0
47	0	0	0	0	0	1	1	1	1	1
48	6	11	1	1	1	3	3	3	3	3
49	8	12	2	1	1	3	4	4	3	3
50	9	14	1	1	1	4	4	4	4	4

(b)

ID	#Iterations for UF Constraints					#Iterations for White Boxes				
	S1	S2	S3	S4	S5	S1	S2	S3	S4	S5
51	9	10	1	1	1	3	3	3	3	3
52	8	6	2	2	2	4	2	4	4	4
53	9	10	4	4	4	3	4	4	4	4
54	9	11	1	1	1	4	4	3	3	3
55	5	13	3	3	2	3	4	4	4	4
56	5	11	1	1	1	4	3	2	2	2
57	10	13	3	2	2	4	4	4	4	4
58	8	6	2	2	2	4	2	4	4	4
59	5	11	1	1	1	4	3	2	2	2
60	7	8	1	1	1	3	3	3	3	3
61	36	32	57	19	14	7	9	9	9	6
62	5	11	1	1	1	4	3	2	2	2
63	7	9	1	1	1	5	4	4	4	4
64	8	6	1	1	1	3	3	3	3	3
65	6	8	1	1	1	4	4	3	3	3
66	9	10	4	4	4	3	4	4	4	4
67	8	6	2	2	2	4	2	4	4	4
68	4	15	1	1	1	3	4	3	3	3
69	1	1	1	1	1	1	1	1	1	1
70	3	4	1	1	1	0	1	0	0	0
71	3	13	1	1	1	0	0	0	0	0
72	3	5	1	1	1	0	1	0	0	0
73	2	4	3	3	3	1	1	1	1	1
74	10	14	12	6	8	1	2	1	1	1
75	1	1	2	2	2	1	1	1	1	1
76	5	3	1	2	1	1	1	1	1	1
77	1	3	1	1	1	1	1	2	2	2
78	36	53	56	62	45	0	1	0	0	1
79	165	6	185	89	89	1	1	0	1	1
80	2	3	1	1	1	0	0	0	0	0
81	114	5	203	111	85	0	1	0	1	1
82	4	5	1	1	1	0	0	0	0	0
83	80	53	45	49	35	0	1	0	0	1
84	3	3	1	1	1	0	0	0	0	0
85	2	2	1	1	1	0	1	0	0	0
86	4	6	1	1	1	0	0	0	0	0
87	2	2	3	3	3	0	0	0	0	0
88	2	2	1	1	1	0	0	0	0	0
89	84	4	56	71	71	0	1	1	1	1
90	10	6	103	104	104	1	1	1	1	1
91	0	0	1	1	1	0	0	0	0	0
92	36	57	44	55	29	0	0	0	0	1
93	14	5	428	88	88	2	2	2	2	2
94	3	6	3	3	3	0	0	0	0	0
95	1	1	1	1	1	0	0	0	0	0
96	48	4	191	78	55	1	1	0	1	1
97	3	3	3	3	3	0	0	0	0	0
98	2	2	1	1	1	0	0	0	0	0
99	5	8	1	1	1	0	0	0	0	0
100	3	6	4	5	5	0	0	0	0	0

super_prove	ufar-S1	ufar-S2	ufar-S3	ufar-S4	ufar-S5
2087	2407	2400	2411	2417	2425

Table 5.5: The numbers of solved instances using different settings. 67 instances remain unsolved.

5.8 Conclusion

UFAR is an algorithm that abstracts *all* problematic operators as black boxes up front (black-boxing them), and refines them by applying UF constraints and/or converting them back into their original concrete representations (white-boxing them). The main benefit of explicitly applying UF constraints is that abstractions can be verified by *any* unbounded model checking solver, not limited to SMT-based model checkers as in the cases of most previous work. This enables UFAR to take full advantage of all the recent research developments at the bit-level and the word level, especially PDR and its improvements, PDRA and PDR-WLC. To maximize the applicability of UF constraints, in the first part of this chapter, formal definitions of generic operators were given and a procedure to expose generic operators within regular ones was proposed. This allowed UF constraints to be applied between all same-type generic operators. To improve the quality of UF constraints applied, two improvement techniques for UFAR were presented: counterexample minimization and random simulation. UFAR was implemented and integrated with state-of-the-art bit-level and word-level solvers. UFAR's scalability was demonstrated on a large set of industrial problems.

Chapter 6

Conclusions

6.1 Summary

To enable efficient abstraction and refinement for word-level MC problems, this thesis proposed, implemented, and experimented on a CEGAR-based paradigm of computing abstraction and refinement at the word-level and verifying abstractions at the bit-level.

To compute good abstraction and refinement at the word-level, refinement strategies PBR and MFFC were proposed for word-level localization abstraction in Chapter 2. PBR explicitly encodes assumptions into a circuit with *ITE* operators, so it can take advantage of both the information in the original circuit and assumption interfaces in SAT solvers. MFFC refinement exploits circuit structures that can save unnecessary refinement iterations. The experimental results showed that PBR with MFFC seemed to strike a good balance between the quality of current abstractions and the number of refinement iterations, solving the most cases compared with other settings.

To achieve efficient integration of word-level abstraction refinement and bit-level MC algorithms, we proposed algorithm PDR-WLA in Chapter 4 which efficiently integrates bit-level PDR with word-level refinement strategies proposed in Chapter 2. An important idea of PDR-WLA is to explicitly re-use PDR traces, or reachability clauses, derived in previous refinement iterations. Experiments demonstrated that PDR-WLA solved a larger number of hard problems while offering speedups. PDR-WLA was inspired by algorithm PDRA described in Chapter 3, which efficiently integrates bit-level PDR with bit-level localization abstraction. PDRA introduces a flop map to remember what flops are used in the abstraction. With sophisticated use of the map and only slight modifications of the original PDR, PDRA is able to perform on-the-fly abstraction and refinement while solving an MC problem. Experimental results showed that PDRA solved more hard test cases compared with

the original PDR.

Motivated by industrial benchmarks characterized by having many related arithmetic operators, a framework, UFAR, was proposed in Chapter 5 for word-level model checking. In addition to word-level localization, UF constraints are used to refine related operators by asserting that equal inputs must lead to equal outputs, UF constraints. To address the problem that related operators may not have the same functionality, we proposed to use generic operators that implement standard integer functions. Formal definitions of generic operators were given and a procedure to expose them within regular operators were proposed to increase the applicability of UF constraints. To address the problem of potential constraint explosion, we proposed a counterexample-guided approach and two improvement techniques using minimized counterexamples and random simulation, applying only important UF constraints to abstractions. UFAR explicitly formulates UF constraints in sequential circuits, which allows any bit-level or word-level MC solver to be integrated into the framework, including both PDRA and PDR-WLA. Experimental results showed that UFAR successfully solved most cases in the given industrial benchmarks.

6.2 Future Work

- Large memories in word-level circuits have been challenging for word-level MC. Memories are typically described as arrays of word-level registers in HDL. Our abstraction and refinement approaches proposed in this thesis do not perform memory abstraction at the word-level. Without proper abstraction of memories, memories are interpreted as independent registers, where useful high-level information like operations `read` and `write` is lost. Therefore, our approaches can be enhanced significantly in solving practical benchmarks from industry if they are integrated with a good memory abstraction technique. A possible direction is to integrate the technique proposed by Per Bjesse [Bje08], where original memories are replaced by abstracted ones with additional memory constraints explicitly formulated in a circuit. Abstractions created using this approach can also be solved by any bit-level or word-level MC solver, which fits perfectly into the paradigm focus of this thesis. Another possible directions is to learn from SMT-based memory abstraction [Ger11, SSM+12].

- We proposed an efficient integration of bit-level PDR and word-level abstraction refinement in Chapter 4. While PDR has been shown to be very effective in producing unbounded proofs, BMC is known to be more efficient in finding counterexamples, which is important in a CEGAR-based algorithm where many spurious counterexamples are expected. Therefore, a possible direction is to develop an efficient integration of BMC and word-level abstraction refinement. An idea is to extend Gate-level Abstraction (GLA) proposed by Mishchenko et al. [MEB+13] to the word-level. The extension can

use either SAT-based or SMT-based BMC engines. The refinement strategies proposed in this thesis can be directly applied. The PDR improvements proposed in this thesis also complement the BMC-based approaches, which only generate bounded proofs.

- Word-level localization and UF constraints were shown to be effective in refining current abstractions in this thesis. However, we observed that in certain benchmarks, the proposed refinement scheme failed to find good abstractions. A possible direction is to use other types of constraints, such as partial interpretations of arithmetic operators. For example, properties of a multiplier when given special input values like 1 and 0 can be used to refine an abstracted multiplier. In some cases, only partial interpretations are needed for a final proof without using fully-interpreted operators. One of the challenges is to automatically identify good constraints and prevent constraint explosion, which leads to improved overall performances. An idea is to integrate the work on conditional abstractions proposed by Brady et al. [BBS11], where conditions are learned automatically for function abstraction.

- SMT-based CEGAR algorithms for word-level MC can benefit from the ideas proposed in this thesis. For example, the idea of re-using reachability information can enhance other CEGAR-based algorithms with integrated PDR. In particular, AVERROES [LS14] is a word-level CEGAR algorithm with an integrated SMT-based PDR, which can be enhanced by re-using word-level PDR traces across refinement iterations. On the other hand, our approaches would benefit from SMT-based algorithms also. For example, the UFAR framework can be greatly enhanced by using efficient SMT-based MC solvers to verify current abstractions.

Bibliography

[Ack54] Wilhelm Ackermann. *Solvable Cases of the Decision Problem.* North Holland Pub. Co., 1954.

[ALS06] Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ASP-DAC '06, 2006.

[ALS08] Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah. Reveal: A formal verification tool for verilog designs. In *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2008.

[AS04] Zaher S. Andraus and Karem A. Sakallah. Automatic abstraction and verification of verilog models. In *Proceedings of the 41st Annual Design Automation Conference (DAC)*, 2004.

[BB09] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2009.

[BBS11] Bryan A. Brady, Randal E. Bryant, and Sanjit A. Seshia. Learning conditional abstractions. In *Proceedings of the 11th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2011.

[BBSO10] Bryan A. Brady, Randal E. Bryant, Sanjit A. Seshia, and John 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)*, 2010.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 1999.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS)*, 1992.

[BD94] Jerry R. Burch and David L. Dill. Automatic verification of pipelined micropro-

cessor control. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV)*, 1994.

[BEM12] Robert Brayton, Niklas Eén, and Alan Mishchenko. Using speculation for sequential equivalence checking. In *Proceedings of the 21st International Workshop on Logic and Synthesis (IWLS)*, 2012.

[BGV99] Randal E. Bryant, Steven German, and Miroslav N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV)*, 1999.

[BH14] Armin Biere and Keijo Heljanko. Hardware model checking competition, 2014. `http://fmv.jku.at/hwmcc14cav/`.

[BIMM12] Jason Baumgartner, Alexander Ivrii, Arie Matsliah, and Hari Mony. Ic3-guided abstraction. In *Proceedings of the 12th Formal Methods in Computer-Aided Design (FMCAD)*, 2012.

[Bje08] Per Bjesse. Word-level sequential memory abstraction for model checking. In *Proceedings of the 8th Formal Methods in Computer-Aided Design (FMCAD)*, 2008.

[BK05] Per Bjesse and James Kukula. Automatic generalized phase abstraction for formal verification. In *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design (ICCAD)*, 2005.

[BKO+07] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. Deciding bit-vector arithmetic with abstraction. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2007.

[BLS02] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In E. Brinksma and K. G. Larsen, editors, *CAV02*, 2002.

[BM10] Robert Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In *Proceedings of the 22Nd International Conference on Computer Aided Verification (CAV)*, 2010.

[Bra11] Aaron R. Bradley. Sat-based model checking without unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2011.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, September 1992.

[BSST09] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.

[BST10] Clark Barrett, Aaron Stump, and Cesare 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`.

[BvDH17] Armin Biere, Tom van Dijk, and Keijo Heljanko. Hardware model checking

competition, 2017. http://fmv.jku.at/hwmcc17/.

[CES13] Koen Claessen, Niklas Een, and Baruch Sterin. A circuit approach to ltl model checking. In *Proceedings of the 13th Formal Methods in Computer-Aided Design (FMCAD)*, 2013.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV)*, 2000.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[Een] Niklas Een. Abc-zz. https://bitbucket.org/niklaseen/abc-zz.

[EMA10] Niklas Een, Alan Mishchenko, and Nina Amla. A single-instance incremental sat formulation of proof- and counterexample-based abstraction. In *Proceedings of the 10th Formal Methods in Computer Aided Design (FMCAD)*, 2010.

[EMB11] Niklas Eén, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Proceedings of the 11th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2011.

[ES03a] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2003.

[ES03b] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543 – 560, 2003. BMC'2003, First International Workshop on Bounded Model Checking.

[FYH16] Kuan Fan, Ming-Jen Yang, and Chung-Yang Huang. Automatic abstraction refinement of TR for PDR. In *Proceedings of the 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2016.

[Ger11] Steven M. German. A theory of abstraction for arrays. In *Proceedings of the 11th Formal Methods in Computer-Aided Design (FMCAD)*, 2011.

[GR16] Alberto Griggio and Marco Roveri. Comparing different variants of the ic3 algorithm for hardware model checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 35(6):1026–1039, 2016.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV)*, 1997.

[HBS13] Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi. Better generalization in IC3. In *Proceedings of the 13th Formal Methods in Computer-Aided Design (FMCAD)*, 2013.

[HCR+16] Yen-Sheng Ho, Pankaj Chauhan, Pritam Roy, Alan Mishchenko, and Robert Brayton. Efficient uninterpreted function abstraction and refinement for word-level model checking. In *Proceedings of the 16th Formal Methods in Computer-Aided Design (FMCAD)*, 2016.

[HMB17] Yen-Sheng Ho, Alan Mishchenko, and Robert Brayton. Property directed reach-

ability with word-level abstraction. In *Proceedings of the 17th Formal Methods in Computer-Aided Design (FMCAD)*, 2017.

[HMBE17] Yen-Sheng Ho, Alan Mishchenko, Robert Brayton, and Niklas Een. Enhancing PDR/IC3 with localization abstraction. In *Proceedings of the 26th International Workshop on Logic and Synthesis (IWLS)*, 2017.

[Hun89] Warren A. Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, Dec 1989.

[IEE06] IEEE standard 1364-2005 for verilog hardware description language, 2006.

[IG15] Alexander Ivrii and Arie Gurfinkel. Pushing to the top. In *Proceedings of the 15th Formal Methods in Computer-Aided Design (FMCAD)*, 2015.

[JKSC05] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund Clarke. Word level predicate abstraction and refinement for verifying RTL verilog. In *Proceedings of the 42nd annual Design Automation Conference (DAC)*, 2005.

[KP] Daniel Kroening and Mitra Purandare. Ebmc: The enhanced bounded model checker. `www.cprover.org/ebmc`.

[LB03] Shuvendu K. Lahiri and Randal E. Bryant. Deductive verification of advanced out-of-order microprocessors. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV)*, 2003.

[LS04] Shuvendu K. Lahiri and Sanjit A. Seshia. The UCLID decision procedure. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)*, 2004.

[LS14] Suho Lee and Karem A. Sakallah. Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, 2014.

[LSB02] Shuvendu K. Lahiri, Sanjit A. Seshia, and Randal E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *Proceedings of the 4th Formal Methods in Computer-Aided Design (FMCAD)*, 2002.

[McM03] Kenneth L. McMillan. Interpolation and sat-based model checking. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV)*, 2003.

[MEB+12] Alan Mishchenko, Niklas Een, Robert Brayton, Jason Baumgartner, Hari Mony, and Pradeep Nalla. Variable time-frame abstraction. In *Proceedings of the 21st International Workshop on Logic and Synthesis (IWLS)*, 2012.

[MEB+13] Alan Mishchenko, Niklas Eén, Robert K. Brayton, Jason Baumgartner, Hari Mony, and Pradeep Kumar Nalla. GLA: Gate-level abstraction revisited. In *Proceedings of the 16th Design, Automation and Test in Europe (DATE)*, 2013.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference (DAC)*, 2001.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS)*, 1977.

[SS96] João P. Marques Silva and Karem A. Sakallah. Grasp —a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 1996.

[SSM+12] Rohit Sinha, Cynthia Sturton, Petros Maniatis, Sanjit A. Seshia, and David Wagner. Verification with small and short worlds. In *Proceedings of the 12th Formal Methods in Computer-Aided Design (FMCAD)*, 2012.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2000.

[ucl] UCLID. `uclid.eecs.berkeley.edu`.

[VGS12] Yakir Vizel, Orna Grumberg, and Sharon Shoham. Lazy abstraction and sat-based reachability in hardware model checking. In *Proceedings of the 12th Formal Methods in Computer-Aided Design (FMCAD)*, 2012.

[WJK+01] Dong Wang, Pei-Hsin Jiang, James Kukula, Yunshan Zhu, Tony Ma, and Robert Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *Proceedings of the 38th Annual Design Automation Conference (DAC)*, 2001.

[WK13] Tobias Welp and Andreas Kuehlmann. QF BV model checking with property directed reachability. In *Proceedings of the 16th Design, Automation and Test in Europe (DATE)*, 2013.

[WK14] Tobias Welp and Andreas Kuehlmann. Property directed reachability for qf_bv with mixed type atomic reasoning units. In *Proceedings of the 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014.