Efficient Uninterpreted Function Abstraction and Refinement for Word-level Model Checking

Yen-Sheng Ho, Pankaj Chauhan, Pritam Roy,
Alan Mishchenko, Robert Brayton

Berkeley
UNIVERSITY OF CALIFORNIA

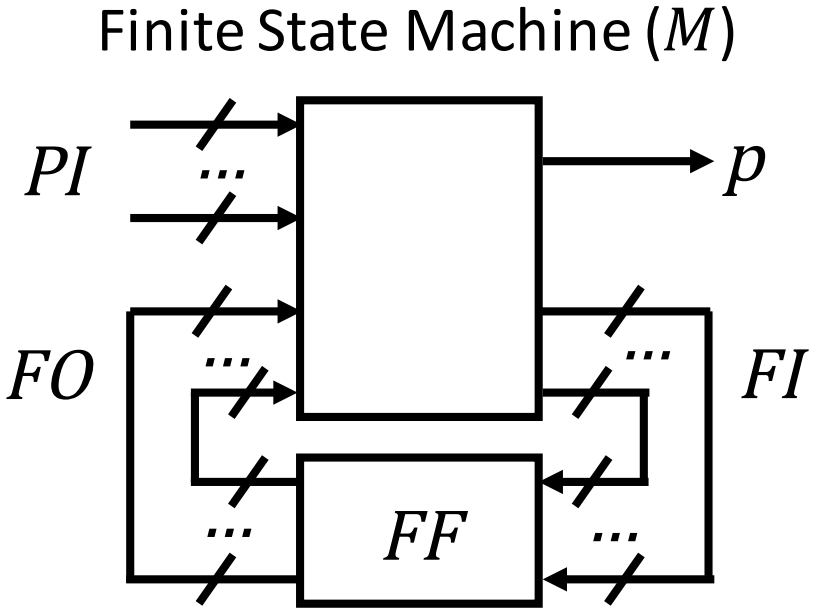
Mentor Graphics®

Word-level model checking

Given a word-level circuit (in structural Verilog),

Prove the property always holds.

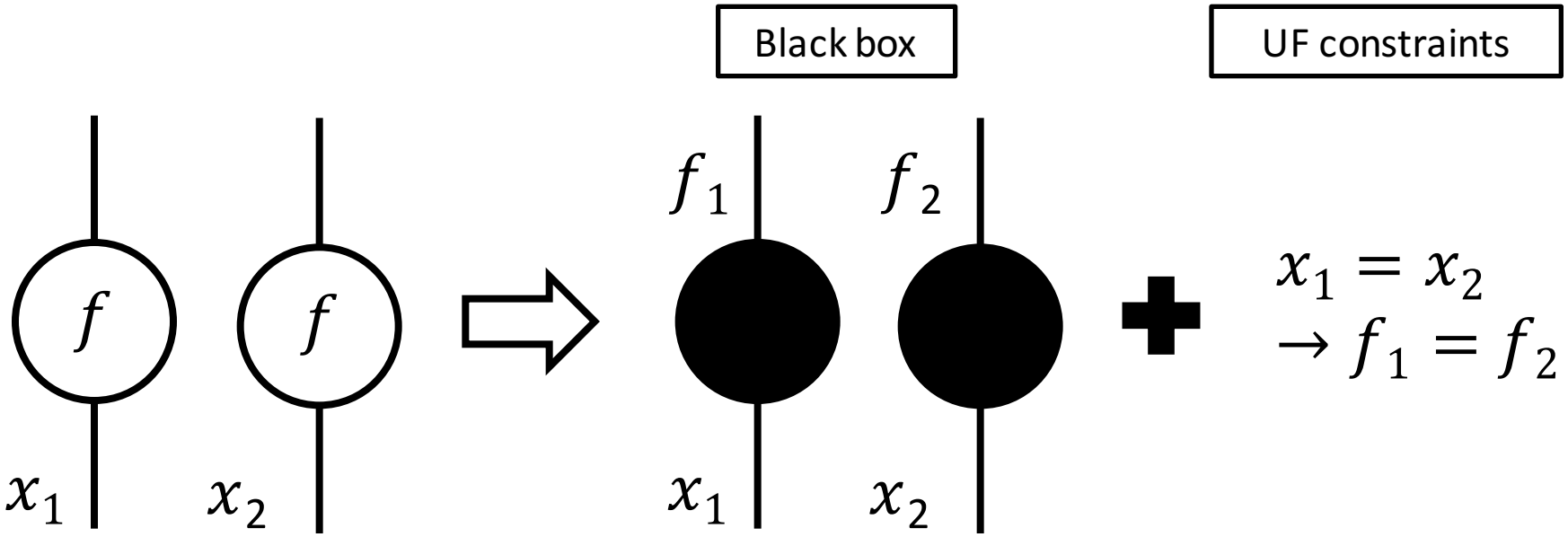
$$M \models \mathbf{G}\, p$$



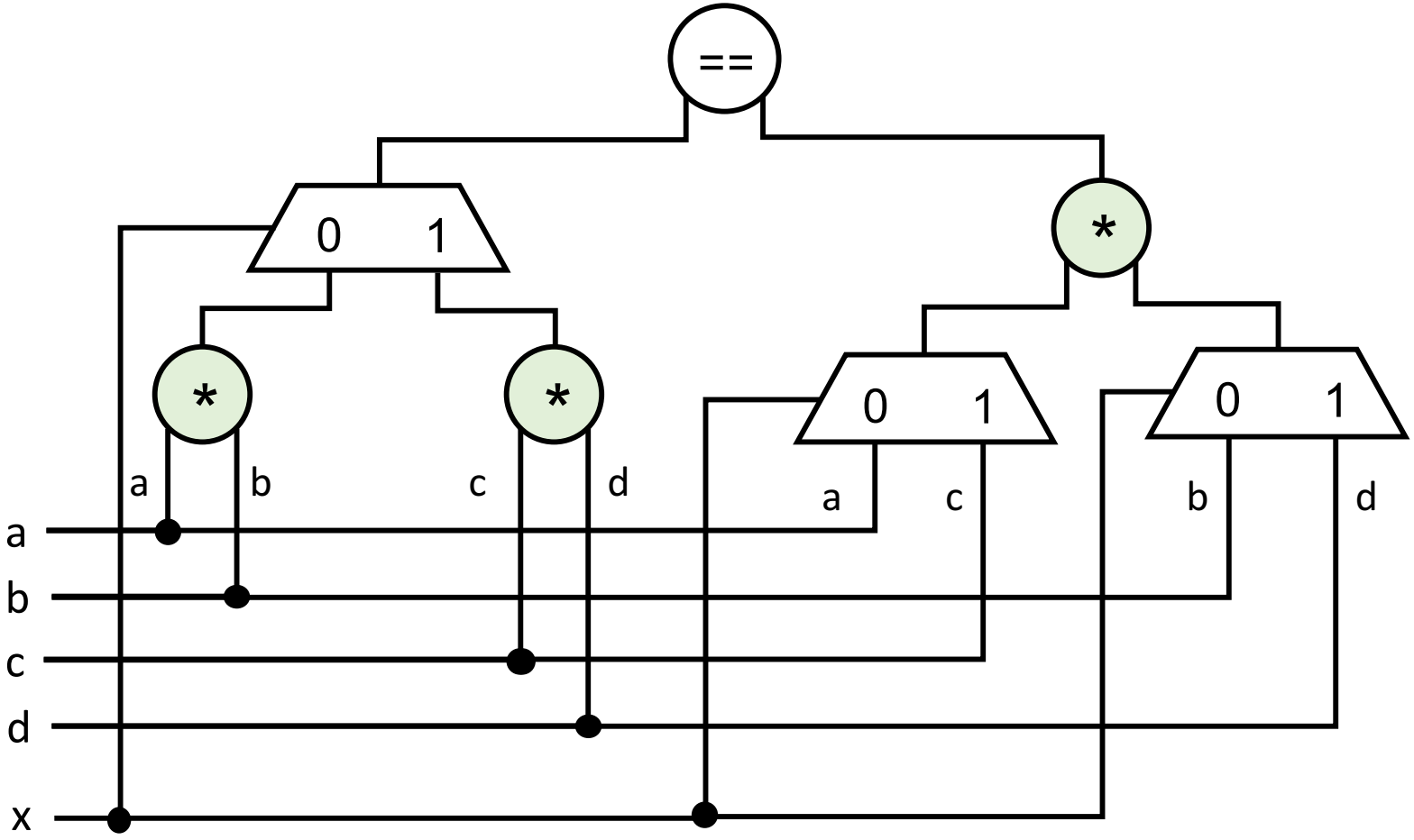
Uninterpreted functions

Uninterpreted function (UF) constraints

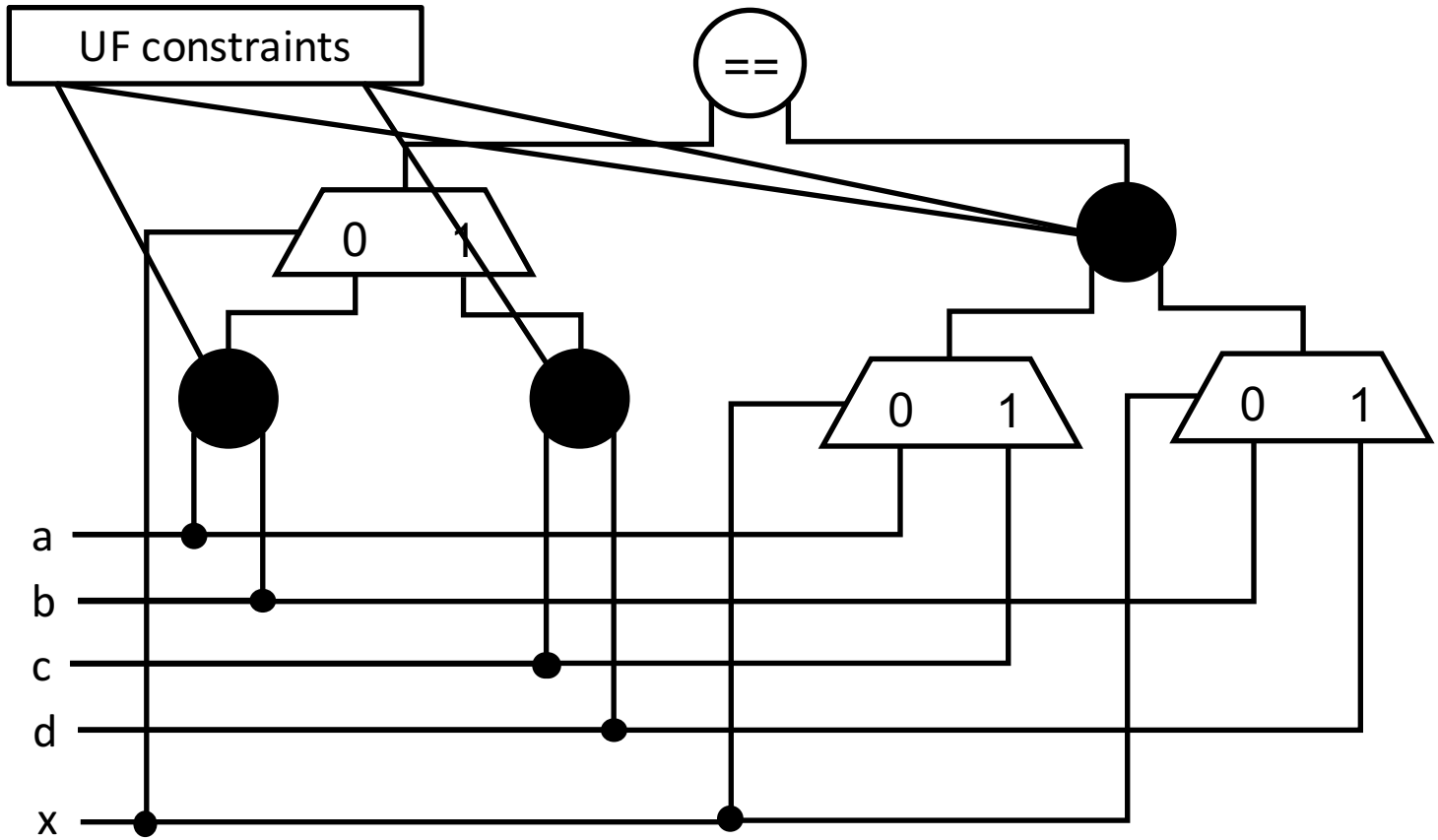
$$x_1 = x_2 \rightarrow f(x_1) = f(x_2)$$



A toy example



A toy example



Challenges

The number of similar functions can be large.

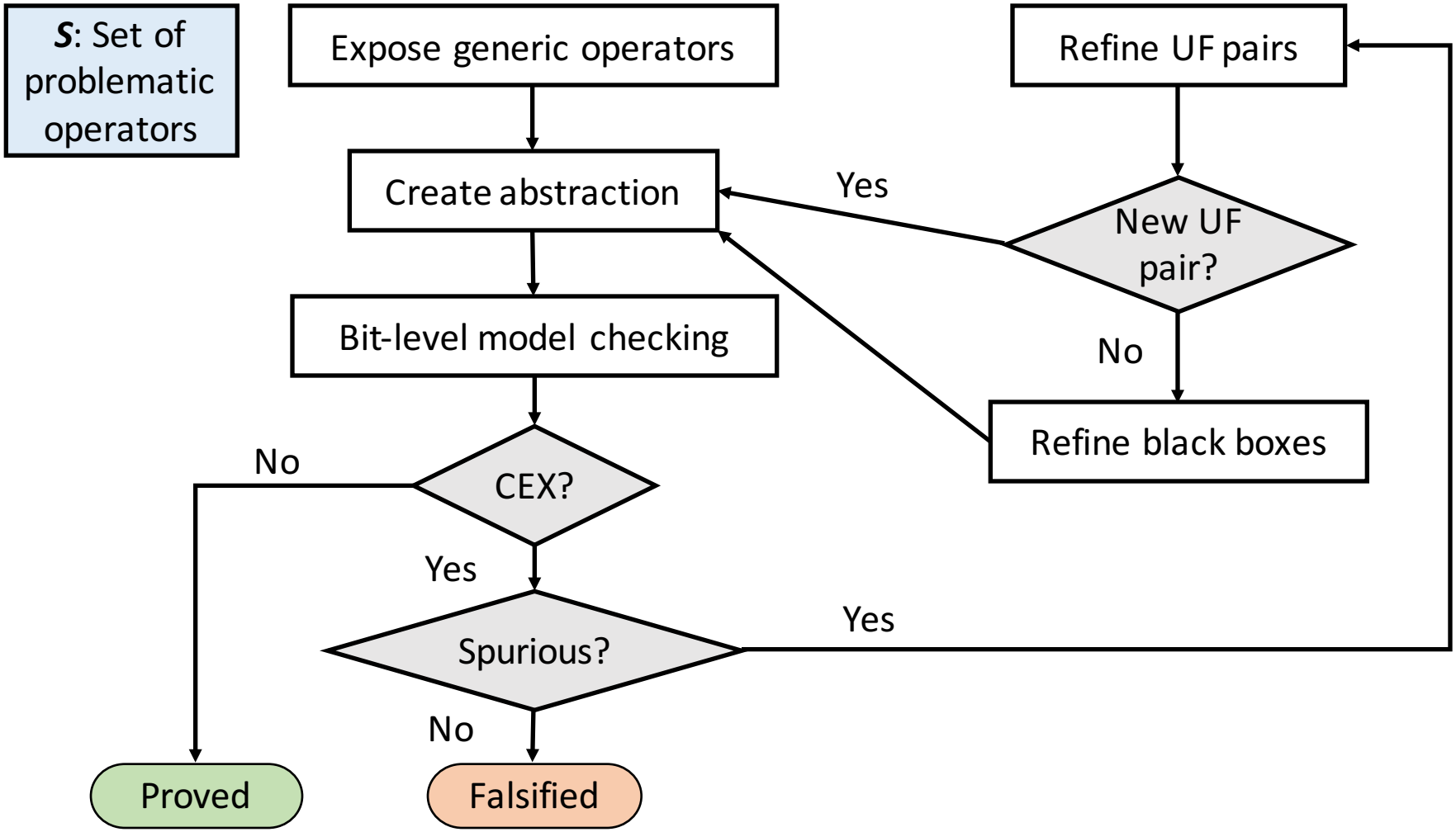
Spurious counterexamples will appear and should be handled carefully.

Uninterpreted Function Abstraction and Refinement (UFAR)

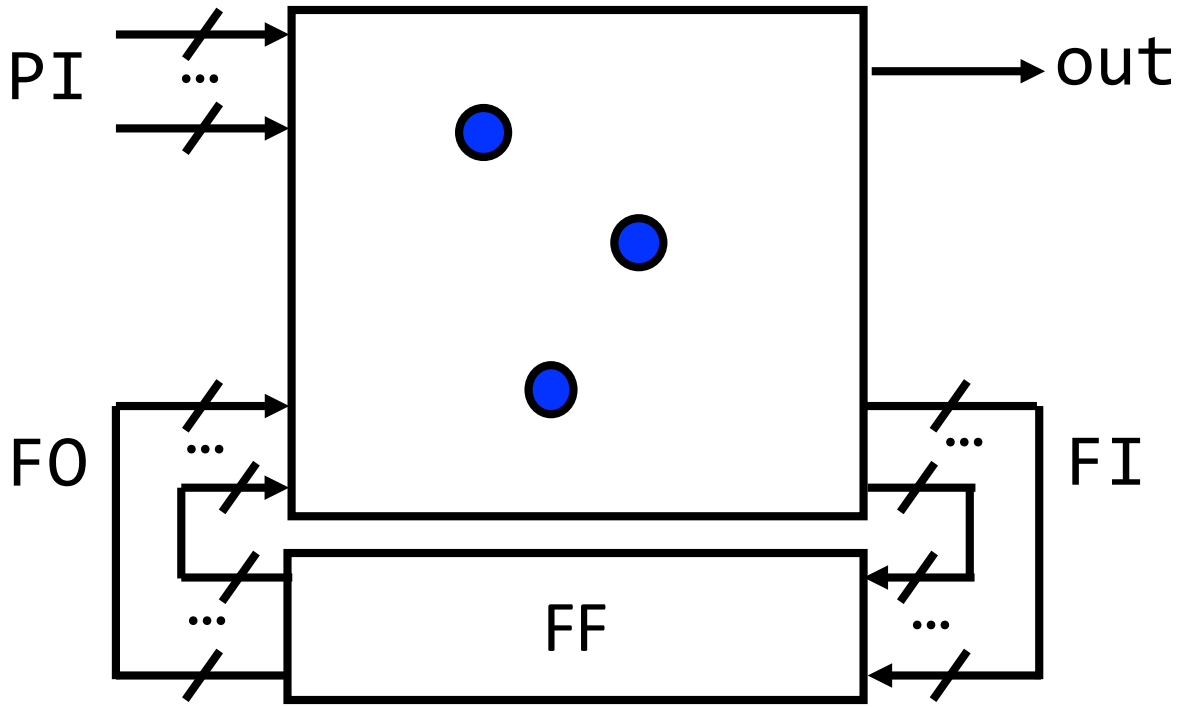
A CEGAR-based algorithm that

- Applies UF constraints lazily
- Uses proof-based approaches to refine black boxes
- Can be combined with any off-the-shelf model checker
- Performs well on a large set of industrial benchmarks

The UFAR algorithm

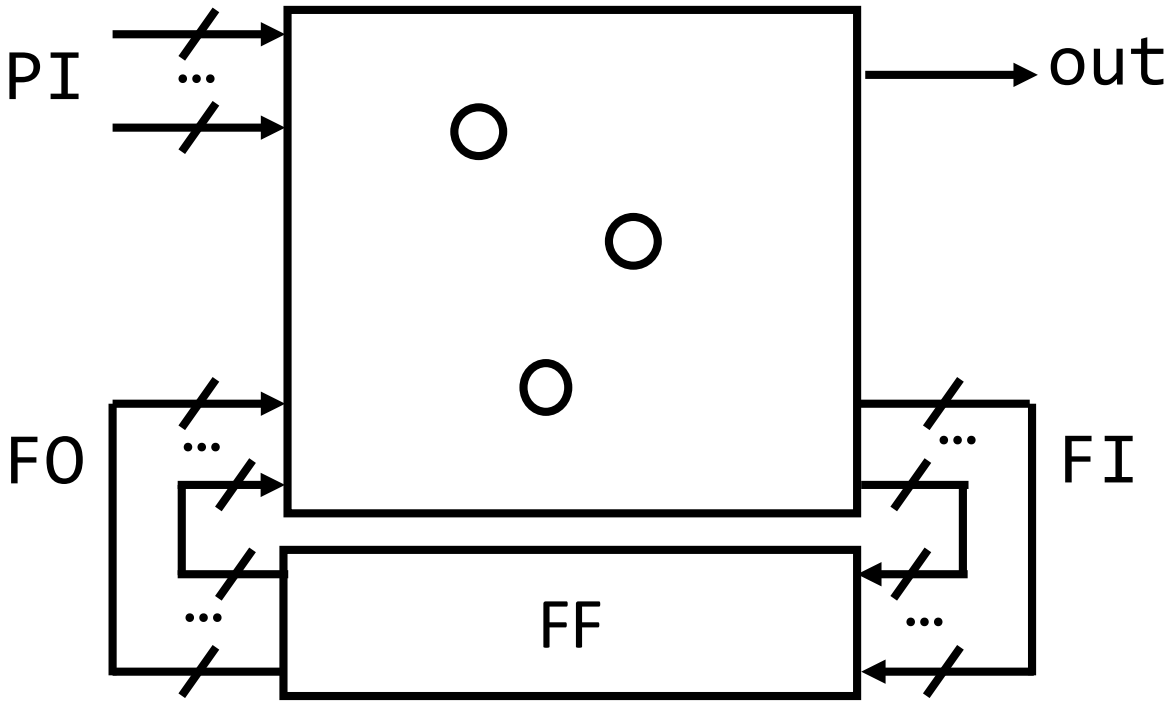


The UFAR algorithm



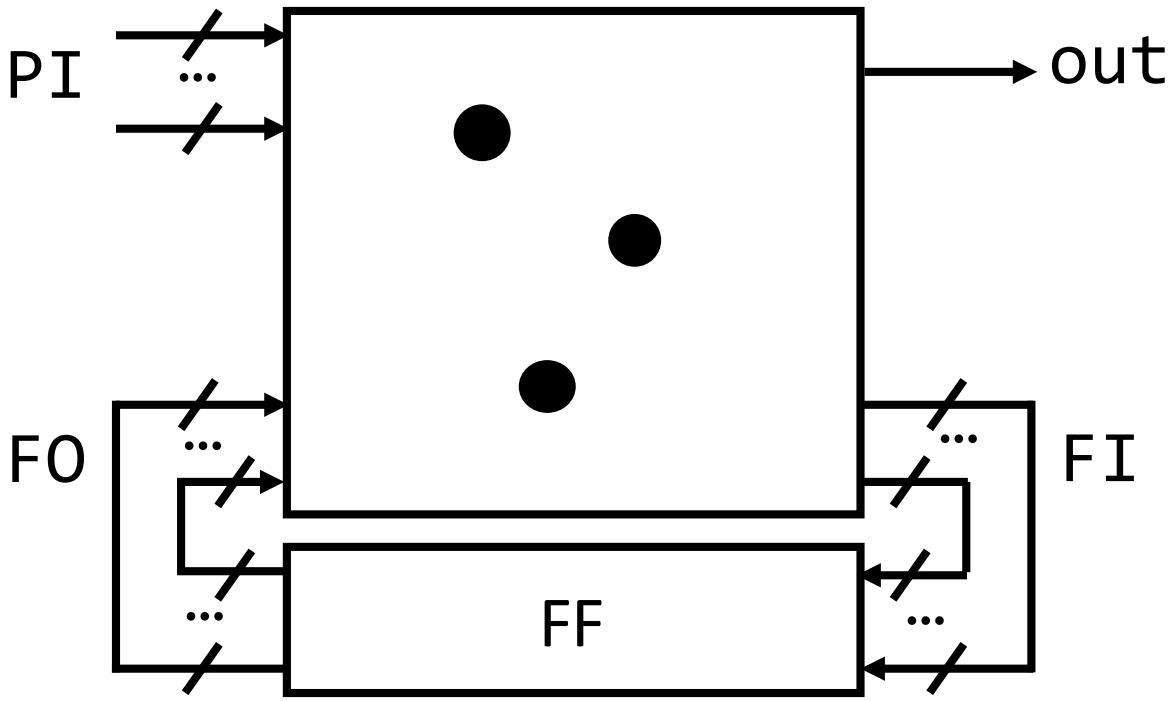
The original word-level miter

The UFAR algorithm



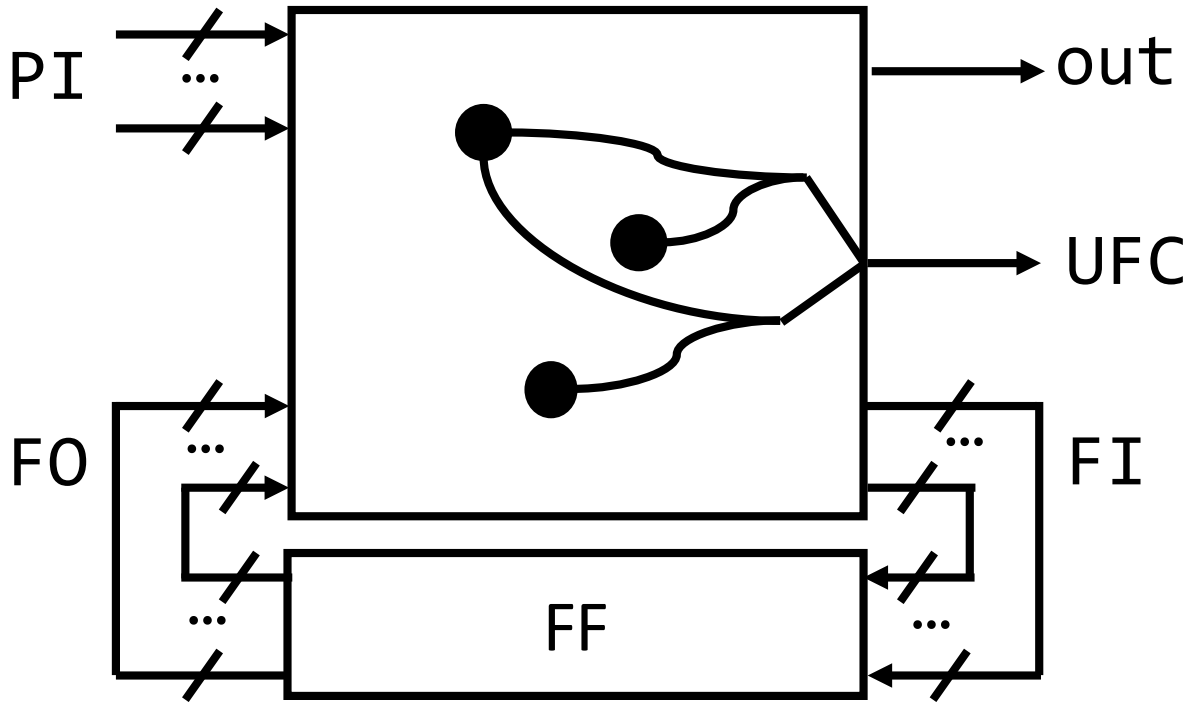
The miter after exposing generic operators

The UFAR algorithm



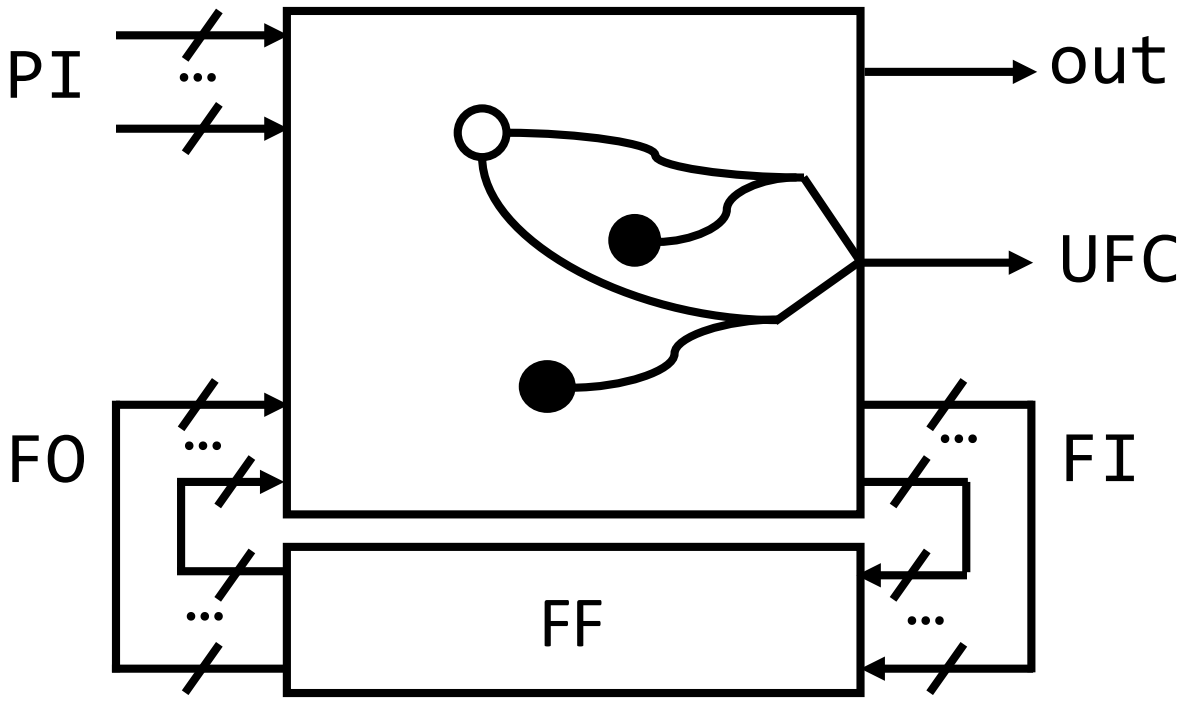
The first abstraction: 3 black boxes, 0 UF constraints

The UFAR algorithm



The second abstraction: 3 black boxes, 2 UF constraints

The UFAR algorithm

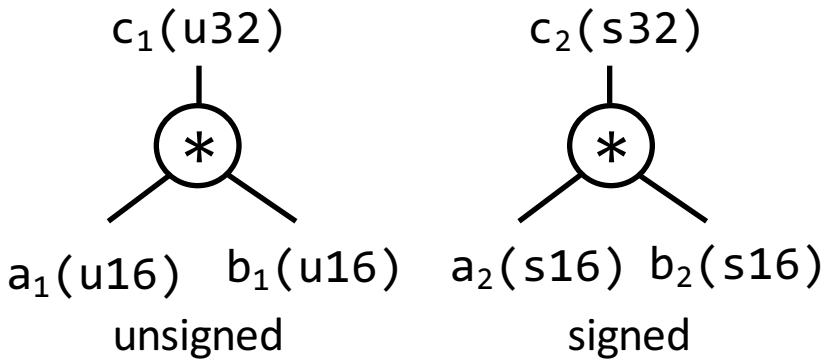


The final abstraction: 2 black boxes, 2 UF constraints

Exposing generic operators

Motivation

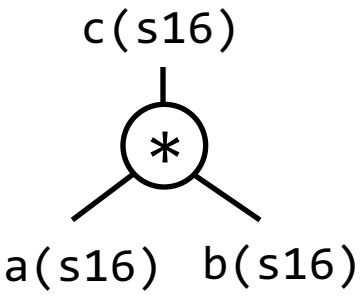
- Want to apply UF for all same-type operators
- Same-type operators do not imply same functions in Verilog
- UF constraints are only valid for same functions



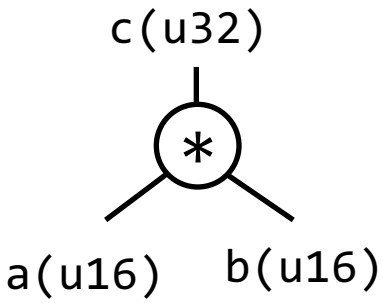
Exposing generic operators

Generic operator

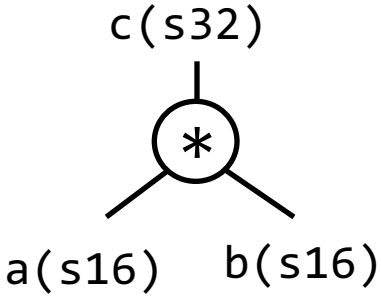
A **signed** bit-vector operator that agrees with the **integer function** of its function-type.



Not generic



Not generic



Generic

Exposing generic operators

Procedure (for multipliers and adders)

- If one of the inputs is *unsigned*, then create zero-padding-by-1 *signed* signal convertors for both inputs.
- Create a generic operator whose output is large enough to prevent overflow.
- Replace the original output with the new output

Exposing generic operators

```
wire signed [4:0] a;
wire        [4:0] b;
wire        [4:0] c = a * b;
```

Expose

```
wire signed [4:0] a;
wire        [4:0] b;
wire        [4:0] c;
wire signed [5:0]  a2 = {1'b0, a};
wire signed [5:0]  b2 = {1'b0, b};
wire signed [11:0] c2 = a2 * b2;
assign             c = c2;
```

Generic operator

Creating abstractions

An abstraction is created from the original circuit using

- ***B***: the set of black-box operators
- ***P***: the set of operator pairs for UF constraints

Procedure

1. For each pair in ***P***, construct a UF constraint.
2. For each operator in ***B***, replace its output with a fresh primary input

Refining UF pairs (*P*)

Goal

Given a spurious counterexample (CEX),
Identify and add UF constraints to block this CEX

Procedure

1. Simulate the spurious CEX on the abstraction
2. Identify same-function pairs that violate UF constraints (the values of the inputs are equal but the outputs are different) and add them to the current set of UF pairs

Refining black operators (*B*)

Goal

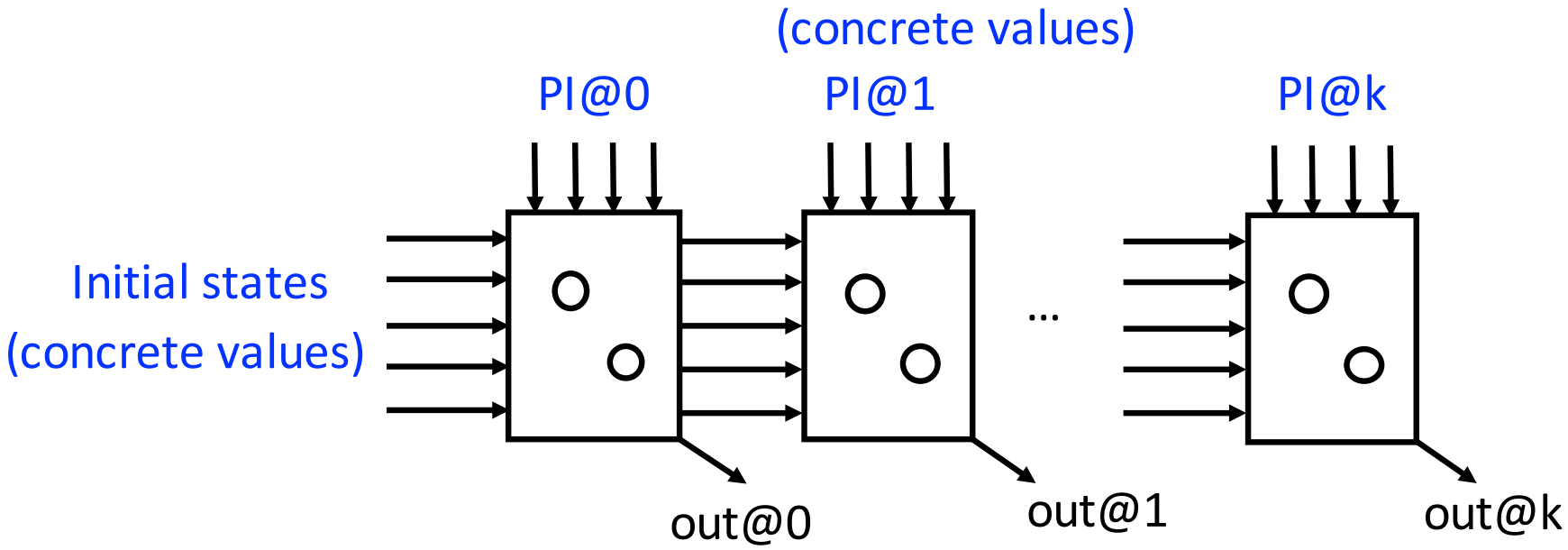
Given a spurious CEX,
Identify and white-box operators to block this CEX.

Procedure

Proof-based analysis to determine what black operators are the responsible for this CEX

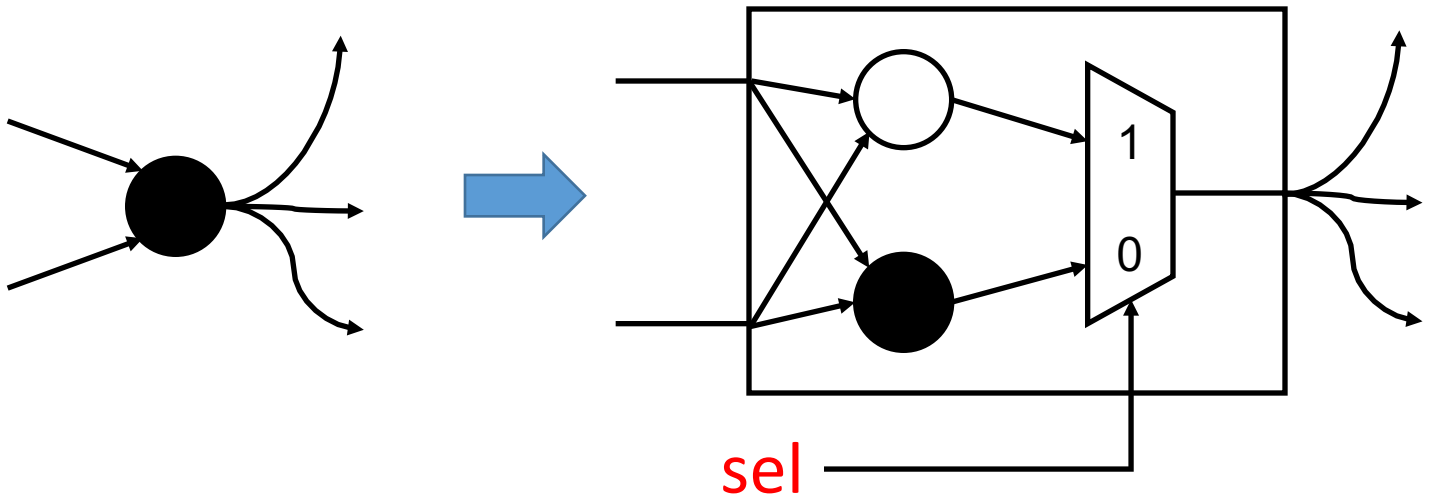
Refining black operators (*B*)

Since the CEX is spurious, we can formulate a satisfiable query which is UNSAT by construction.



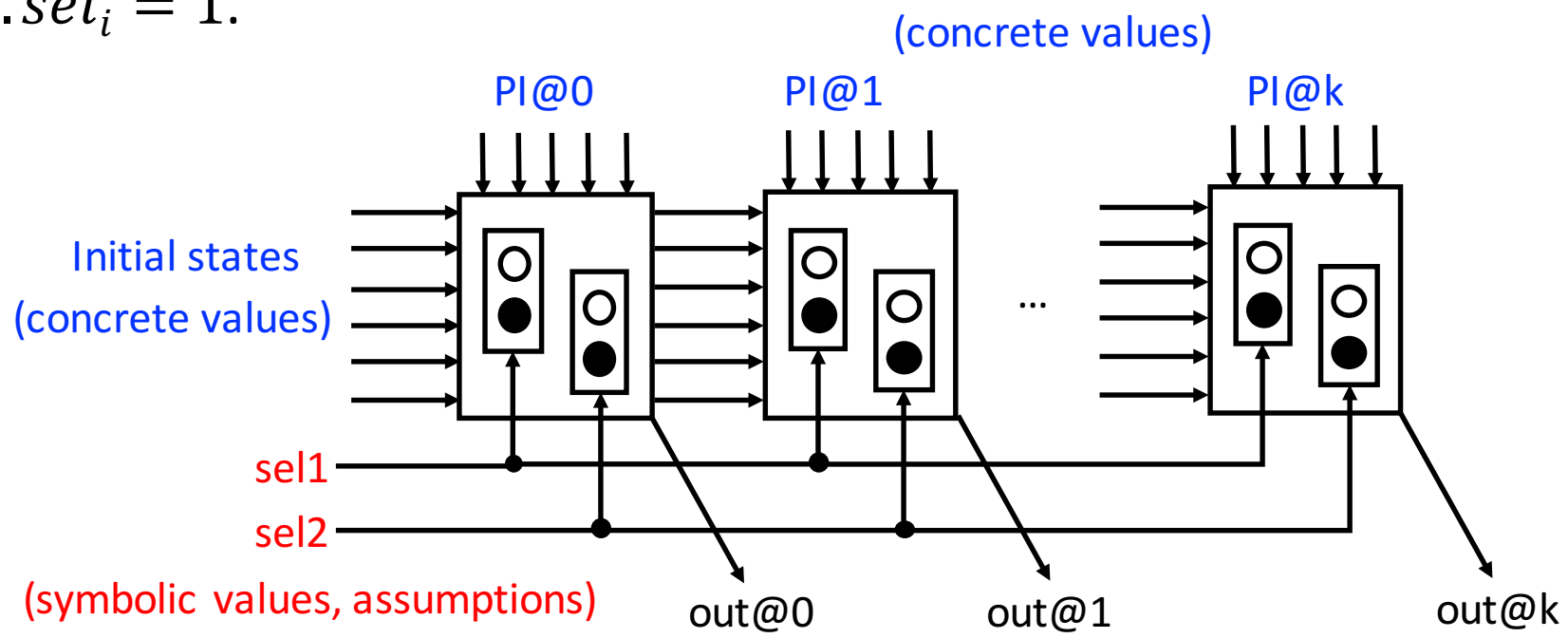
Refining black operators (*B*)

For every black operator in abstraction,
Introduce a MUX selecting between the white and the black.



Refining black operators (B)

We can formulate a new satisfiable query which is UNSAT if $\forall i. sel_i = 1$.



A subset of assumptions will be returned by the SAT solver which is sufficient to block the CEX.

Correctness

UFAR is sound

- An over-approximation is used for each iteration

UFAR is complete

- Every counterexample is simulated on the original circuit

The number of iterations in UFAR is bounded

- $\boldsymbol{B}$ and $\boldsymbol{P}$ are monotone. Either $\boldsymbol{P}$ becomes strictly bigger or $\boldsymbol{B}$ becomes strictly smaller.
- $|\boldsymbol{P}|$ is upper bounded by $|\boldsymbol{S}|(|\boldsymbol{S}| - 1)$
- $|\boldsymbol{B}|$ is lower bounded by 0 (empty set of black boxes).

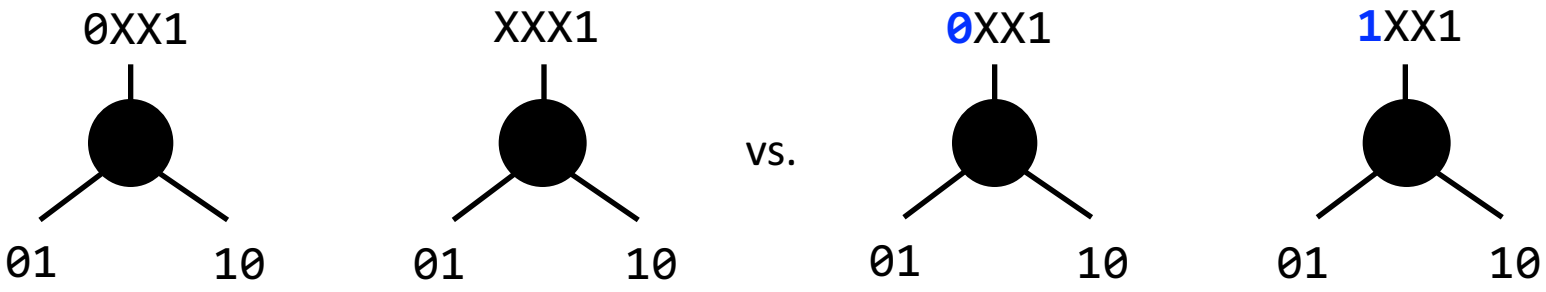
Counterexample minimization

Issue

Ineffective UF pairs may be added

Procedure

- Run `cexmin` [Mishchenko13] to determine don't cares (Xs) for PIs in CEX
- Add only relevant UF pairs



Cannot block minimized CEX

Relevant UF pair

Random simulation

Idea

Relevant UF pairs could be discovered by **random simulation** and should be added before white-boxing.

Procedure

- Run random simulation and count the number of times identical input patterns occur for each pair of operators.
- A UF pair is added if its count is above some threshold.

Related work

Direct bit-blasting with bit-level model checking

Enhanced bounded model checking [cprover.org/ebmc]

- BMC and k-induction with SAT/SMT solvers

Predicate abstraction (VCEGAR) [JKSC'05]

- Predicate abstraction over state variables
- Abstract states and abstract transitions

Term-level abstraction [AS'04][ALS'08][BBSO'10][BBS'11]

- Traditional uninterpreted function abstraction
- BMC and k-induction with SMT solvers

Experimental results

The algorithm is implemented in ABC [BM'10] (available soon).

Bit-level model checkers include 3 PDRs and 1 BMC.

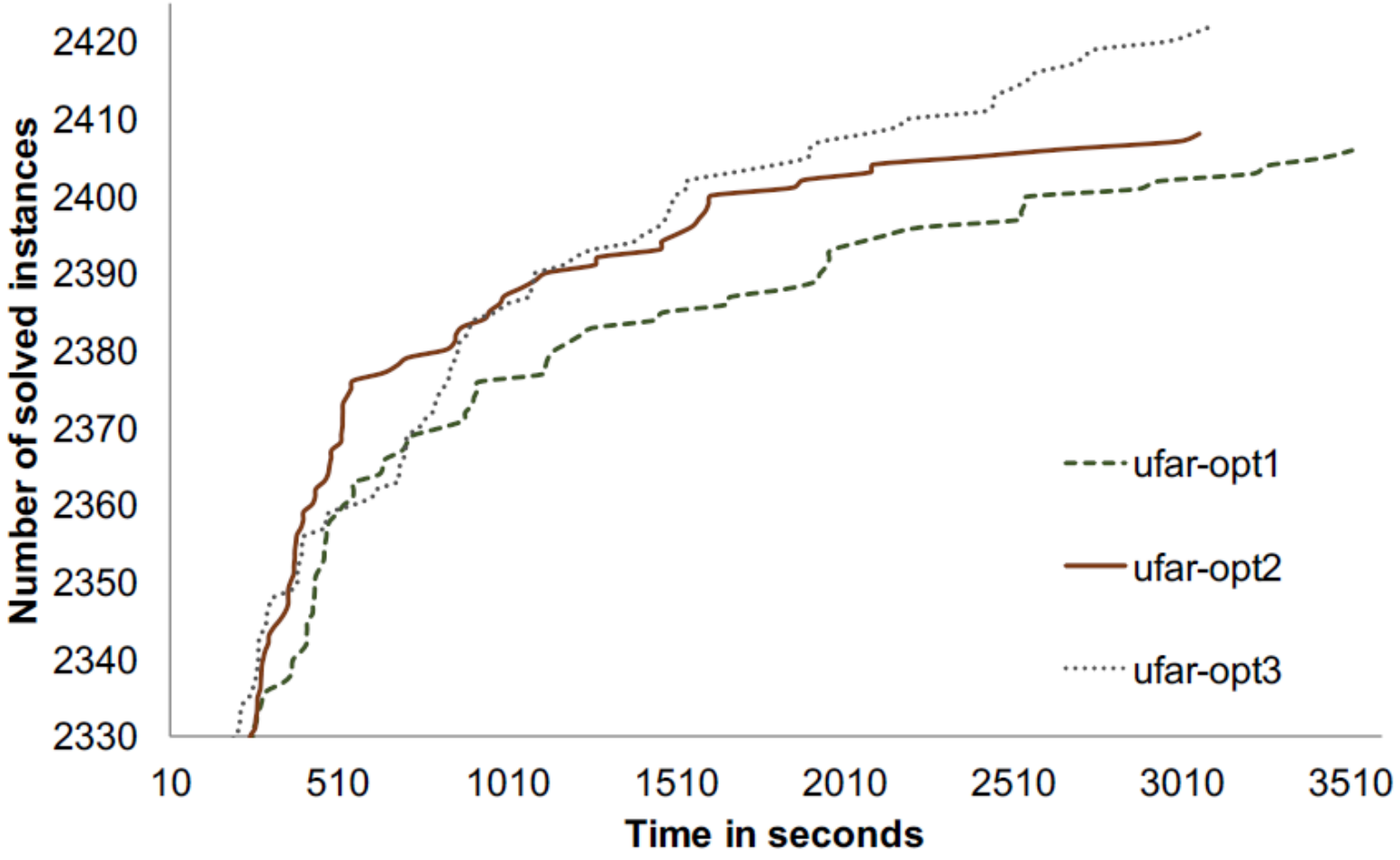
Benchmarks are 2492 industrial Verilog designs

Four configurations

1. super_prove (HWMCC winner) [BEM'12]
2. opt1 -- UFAR
3. opt2 -- UFAR with CEX minimization
4. opt3 -- UFAR with CEX minimization and random simulation

	super_prove	opt1	opt2	opt3
#Solved	2115	2398	2408	2422

Experimental results



Experimental results

UFAR with CEX minimization and random simulation						
Design	Time	#iter_UFC	#iter_WB	#UFConstr	#WBox	#Mults
1	3.3996	1	0	874	0	60
2	1209.7	1	1	6	9	11
3	18.696	1	0	12	0	16
4	1.3337	1	0	4	0	12
5	98.188	2	0	1252	0	102
6	843.09	1	0	352	0	144
8	272.7	2	1	33	8	14
9	2428.6	3	3	276	9	32
10	66.381	1	0	447	0	43
14	1670.8	7	5	1401	37	223
15	2457.5	3	5	374	35	63
16	2169.9	1	1	73	3	15
17	401.76	1	0	394	0	216
18	686.49	82	2	8638	3	253
19	158.99	1	0	268	0	475

Experimental results

UFAR with CEX minimization and random simulation						
Design	Time	#iter_UFC	#iter_WB	#UFConstr	#WBox	#Mults
1	3.3996	1	0	874	0	60
2	1209.7	1	1	6	9	11
3	18.696	1	0	12	0	16
4	1.3337	1	0	4	0	12
5	98.188	2	0	1252	0	102
6	843.09	1	0	352	0	144
8	272.7	2	1	33	8	14
9	2428.6	3	3	276	9	32
10	66.381	1	0	447	0	43
14	1670.8	7	5	1401	37	223
15	2457.5	3	5	374	35	63
16	2169.9	1	1	73	3	15
17	401.76	1	0	394	0	216
18	686.49	82	2	8638	3	253
19	158.99	1	0	268	0	475

Experimental results

UFAR with CEX minimization and random simulation						
Design	Time	#iter_UFC	#iter_WB	#UFConstr	#WBox	#Mults
1	3.3996	1	0	874	0	60
2	1209.7	1	1	6	9	11
3	18.696	1	0	12	0	16
4	1.3337	1	0	4	0	12
5	98.188	2	0	1252	0	102
6	843.09	1	0	352	0	144
8	272.7	2	1	33	8	14
9	2428.6	3	3	276	9	32
10	66.381	1	0	447	0	43
14	1670.8	7	5	1401	37	223
15	2457.5	3	5	374	35	63
16	2169.9	1	1	73	3	15
17	401.76	1	0	394	0	216
18	686.49	82	2	8638	3	253
19	158.99	1	0	268	0	475

Experimental results

		ufar		ufar + cexmin		
Design	#Mults	Time	#UFConstr	Time	#UFConstr	#UFC Reduction
1	**60**	**173.54**	364	1127.1	12	**30.3**
2	**11**	---	24	**2661.2**	30	**0.8**
3	**16**	**20.759**	37	---	17	**2.2**
4	**12**	**4.0193**	26	4.4759	18	**1.4**
5	**102**	**60.659**	641	65.564	82	**7.8**
6	**144**	**1225.8**	2366	2085.1	57	**41.5**
8	**14**	474.51	30	**316.95**	28	**1.1**
9	**32**	**---**	315	---	16	**19.7**
10	**43**	157.14	597	**106.83**	40	**14.9**
14	**223**	**2548.9**	2398	---	674	**3.6**
15	**63**	1967.4	915	**517.86**	142	**6.4**
16	**15**	2939.8	68	**535.18**	7	**9.7**
17	**216**	**2231.2**	1107	---	1943	**0.6**
18	**253**	---	23825	**6.1581**	78	**305.4**
19	**475**	470.71	10556	**150.18**	172	**61.4**

Experimental results

		ufar + cexmin			ufar + cexmin + rsim		
Design	#Mults	Time	#UFConstr	#Wbox	Time	#UFConstr	#Wbox
1	**60**	1127.1	**12**	**0**	**3.3996**	**874**	**0**
2	**11**	2661.2	**30**	**9**	**1209.7**	**6**	**9**
3	**16**	---	**17**	**10**	**18.696**	**12**	**0**
4	**12**	4.4759	**18**	**0**	**1.3337**	**4**	**0**
5	**102**	**65.564**	**82**	**0**	98.188	**1252**	**0**
6	**144**	2085.1	**57**	**0**	**843.09**	**352**	**0**
8	**14**	316.95	**28**	**8**	**272.7**	**33**	**8**
9	**32**	---	**16**	**4**	**2428.6**	**276**	**9**
10	**43**	106.83	**40**	**0**	**66.381**	**447**	**0**
14	**223**	---	**674**	**55**	**1670.8**	**1401**	**37**
15	**63**	**517.86**	**142**	**37**	2457.5	**374**	**35**
16	**15**	**535.18**	**7**	**3**	2169.9	**73**	**3**
17	**216**	---	**1943**	**0**	**401.76**	**394**	**0**
18	**253**	**6.1581**	**78**	**5**	686.49	**8638**	**3**
19	**475**	**150.18**	**172**	**0**	158.99	**268**	**0**

Summary

UFAR offers a new way of applying UF abstraction

UFAR addresses heavy arithmetic logic by abstracting all problematic operators up front and refining them with UF constraints and/or white boxes.

UFAR is scalable on a large set of industrial benchmarks.

Thank you!

Yen-Sheng Ho, Pankaj Chauhan, Pritam Roy,

Alan Mishchenko, Robert Brayton

Berkeley
UNIVERSITY OF CALIFORNIA

Mentor Graphics®