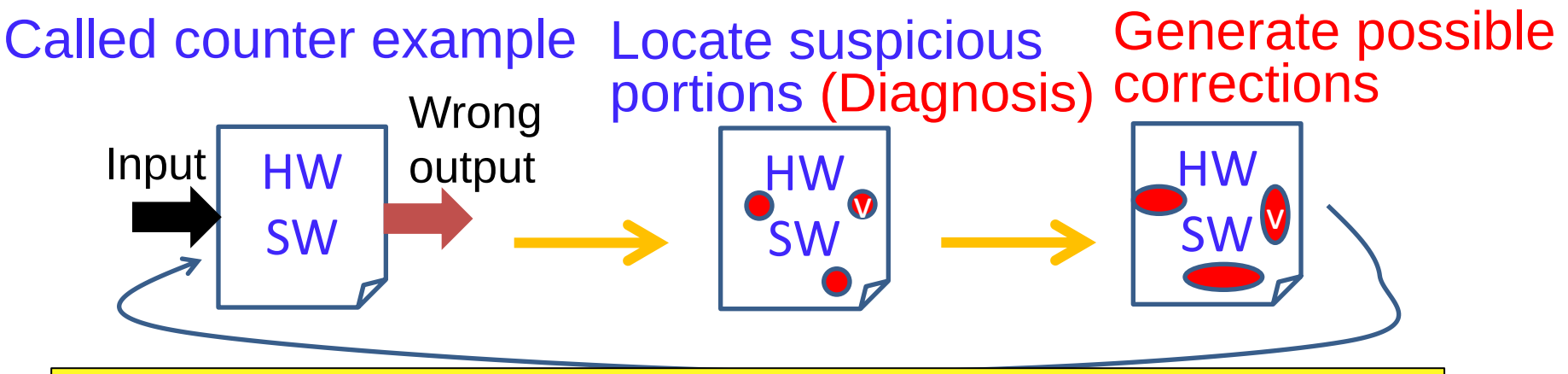
坂井グループ:
形式的検証とテスト段階の修復

論理設計デバッグ時の自動修正技術

藤田昌宏

Debugging process

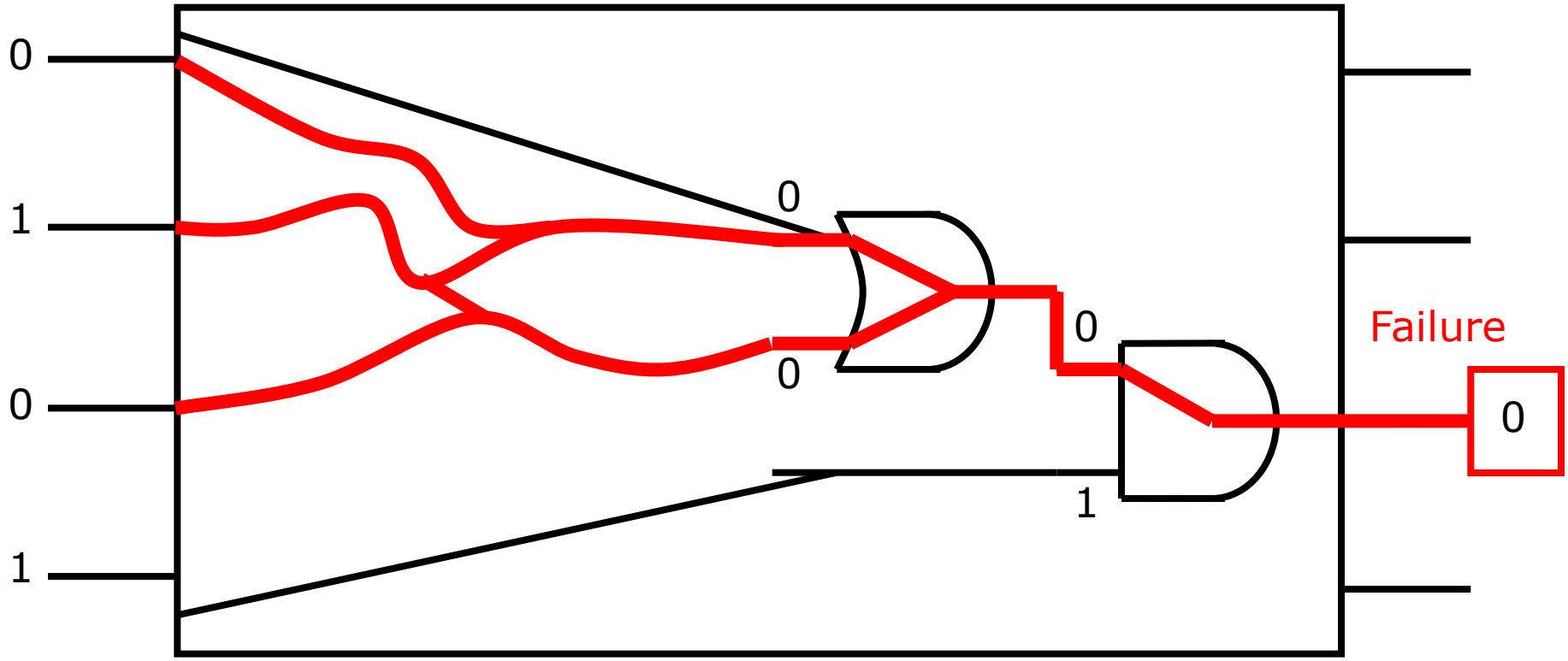
- Given counter examples (test cases that do not produce correct outputs), do the followings:
  - Locate suspicious portions of HW/SW (Diagnose)
  - Come up with possible corrections on them
  - Make sure the corrected HW/SW is OK (start verification process again). If not, repeat the debugging process



Let's discuss diagnosis (existing techniques) first

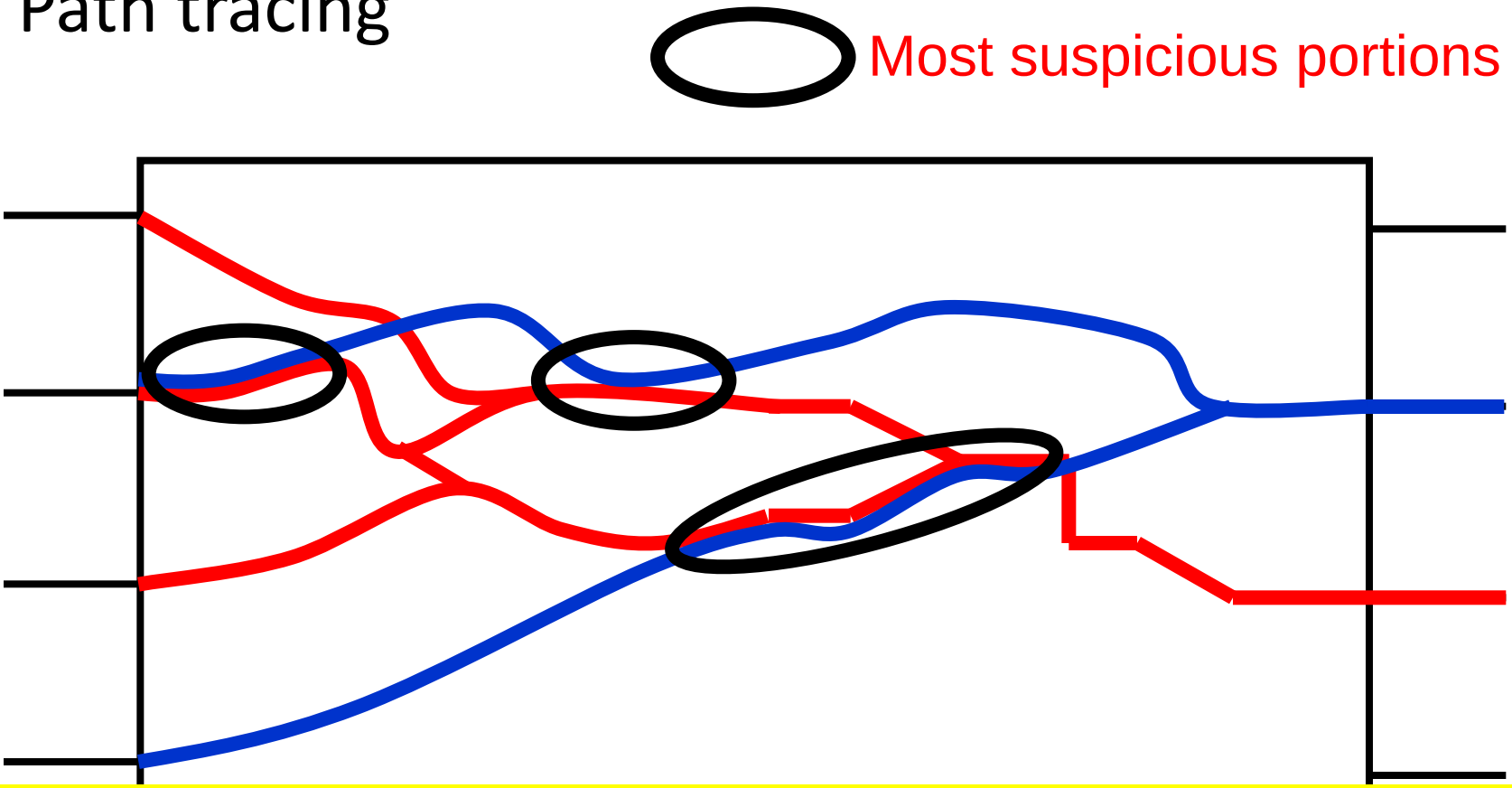
Basic Simulation-based Diagnosis (1)

Path tracing



Basic Simulation-based Diagnosis (2)

Path tracing

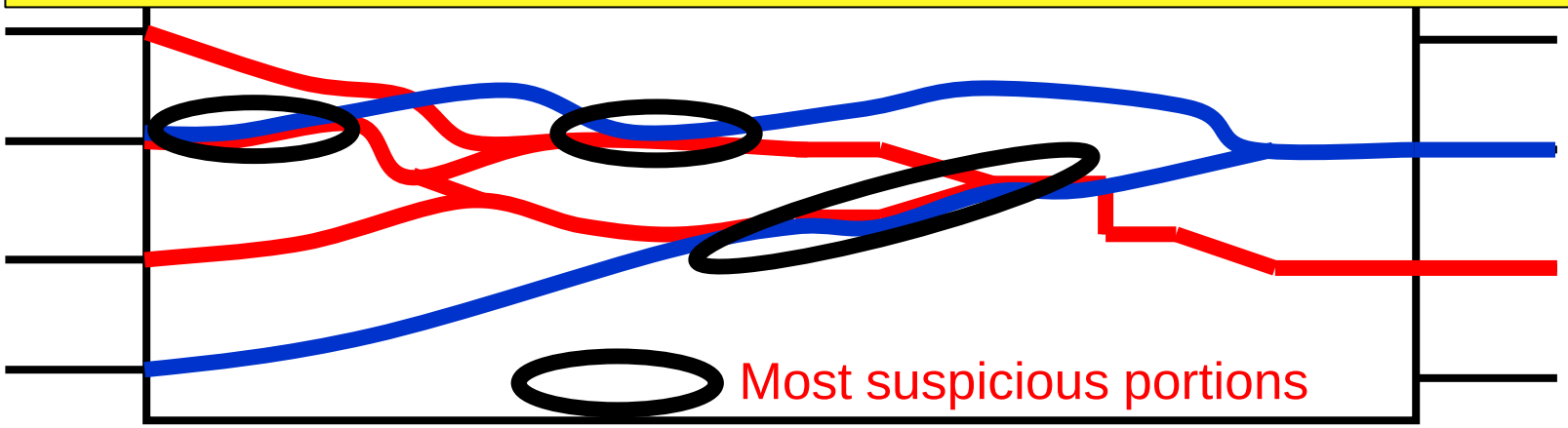


This can be easily formulated as SAT problem

What can we do with SAT-based debugging method ?

- It shows a set of internal signals:
  - By modifying those portions of circuits using all of primary inputs, entire design may become correct
    - Or may not, since we do not know complete specification
    - Only counter examples are given

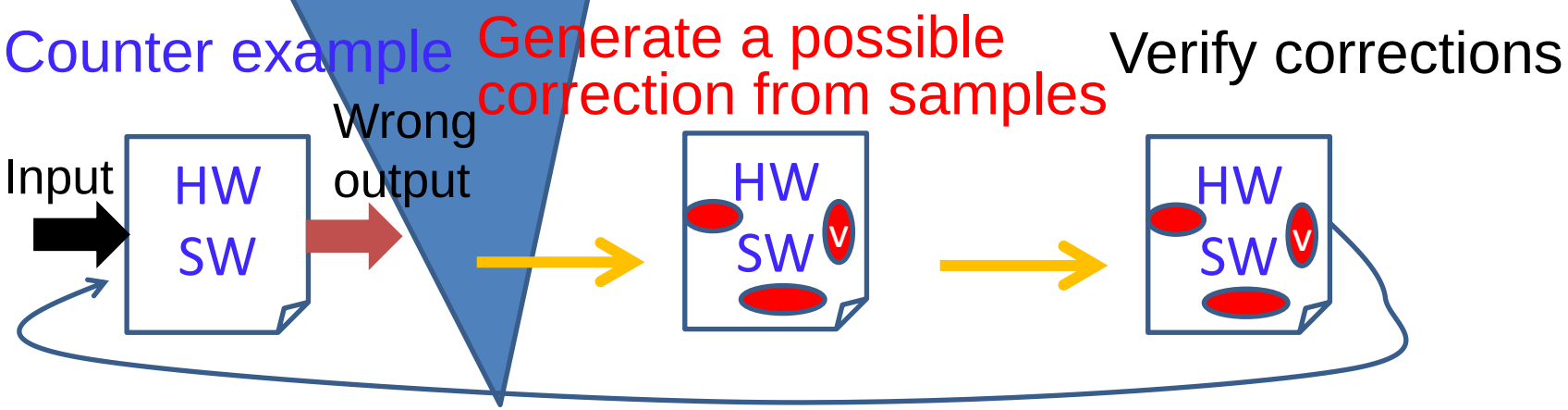
It only suggests where to modify, not the way to modify and also can be incomplete



Debug: Sampling based synthesis with refinement

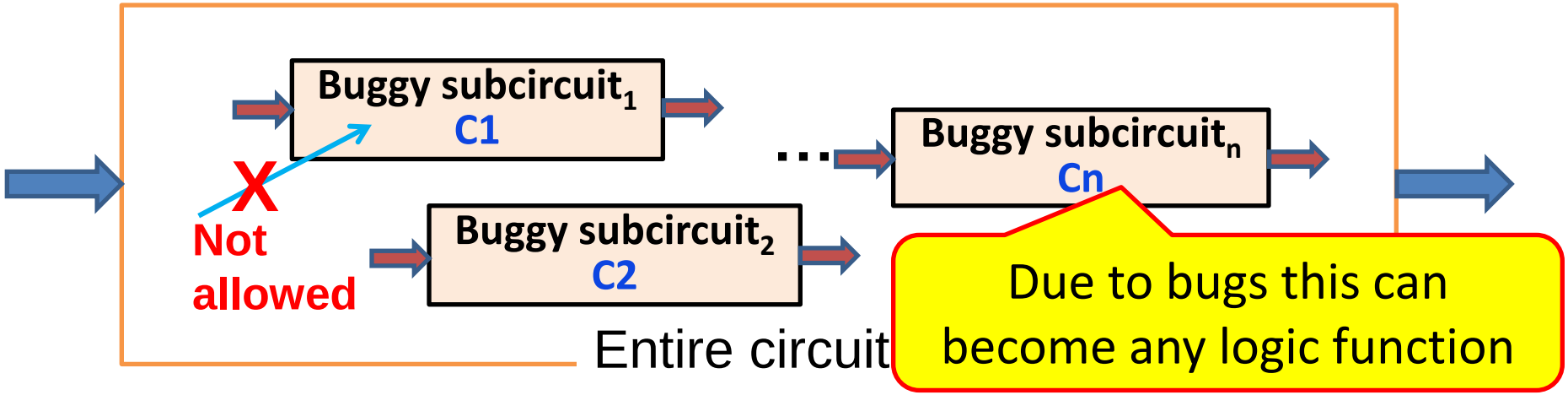
- Formal verifier (SAT solvers) can generate counter examples, if specification is given formally
- Hopefully, numbers of iteration are small

- First fix suspicious portions (discovered by SAT-based)



Correction method

- Bugs exist only inside suspicious subcircuits
- They cannot refer to signals outside of the subcircuits
- Multiple bugs are targeted



- Buggy subcircuits should be modified in such a way that entire circuit becomes correct
- We do not know those functions, but we know the inputs
  - Use Look Up Table (LUT) for each of subcircuit

Problem formulation

- Look up Table (LUT) can represent all logic functions with the set of inputs
  - Introducing programmable variables
- Then the problem becomes QBF (Quantified Boolean Formula)

$$\exists ProgVar.\ \forall Input.\ f(ProgVar, Input) = TargetNet\_SPEC(Input)$$

With appropriate programming of LUT, the circuit is eq[illegible] the spec

Solve the problem with repeated application of SAT solvers

(actually Incremental SAT solvers)

The problem can be efficiently solved by "incremental" SAT solvers

Performance in practice:

10,000 gates, 100 multiple errors need only a couple of minutes and tens of iterations !

Set of generated counter examples (x_1,x_2,…,x_n) :

Complete test vectors (100% verification) assuming bugs are only inside subcircuits

$\ldots \wedge f(Y, x_{n-1}) = Spec(x_{n-1})$ => SAT, solution is (y_n,x_n)

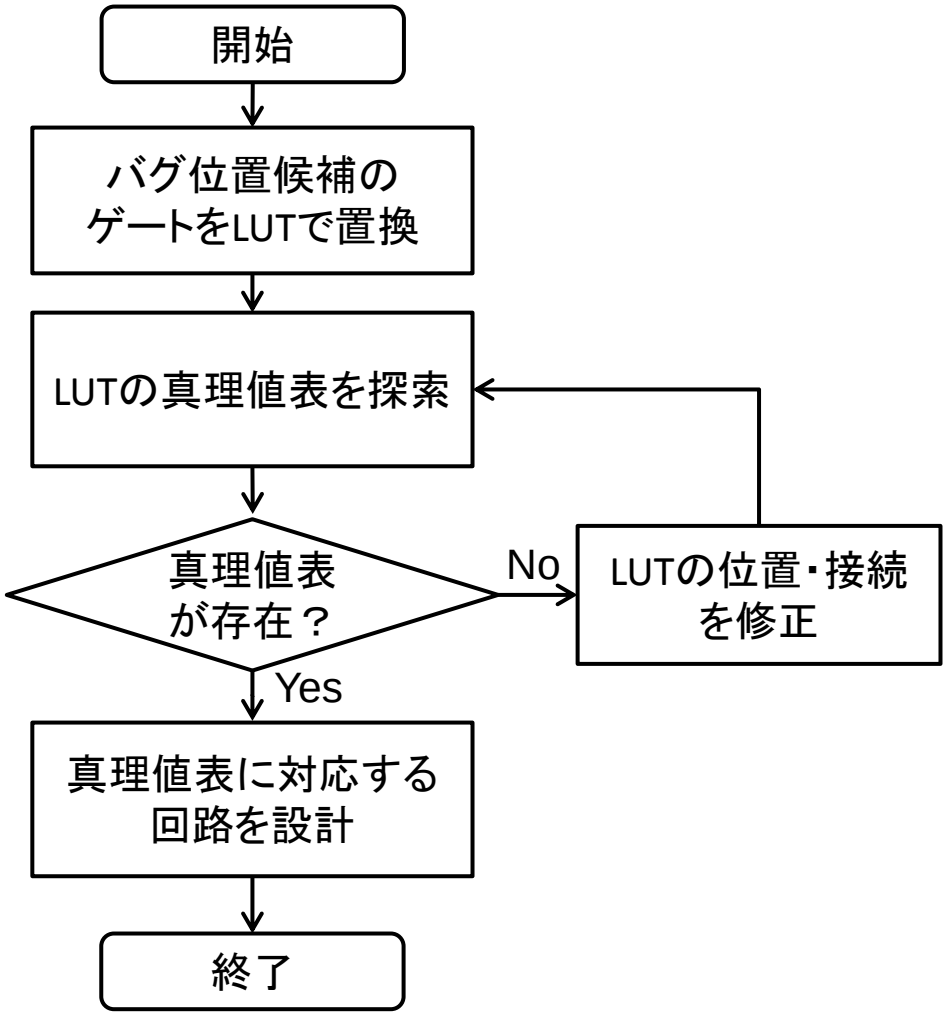
$$\exists Y.X.f(Y,X) \neq Spec(X) \wedge f(Y,x_1) = Spec(x_1) \wedge f(Y,x_2) = Spec(x_2)$$

$$\ldots \wedge f(Y,x_{n-1}) = Spec(x_{n-1}) \wedge f(Y,x_n) = Spec(x_n)$$ => UNSAT

- There are no more counter examples, as long as it works correctly under inputs (x_1, x_2, …, x_n)

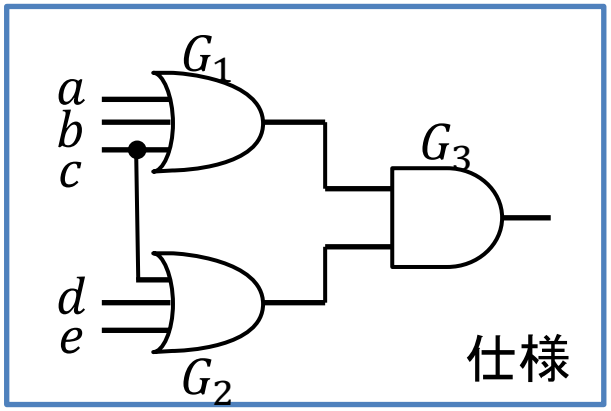
バグ修正のフロー

- バグ位置の候補をLUTで置換
- 設計を仕様と等価にするLUTの真理値表を探索
- 真理値表が求まった場合:真理値表に従って設計を修正する
- デバッグ可能な場合には、短時間にいつも解を見つけられる
- 真理値表が見つからない場合:LUTの配置や変数の接続を修正する必要がある

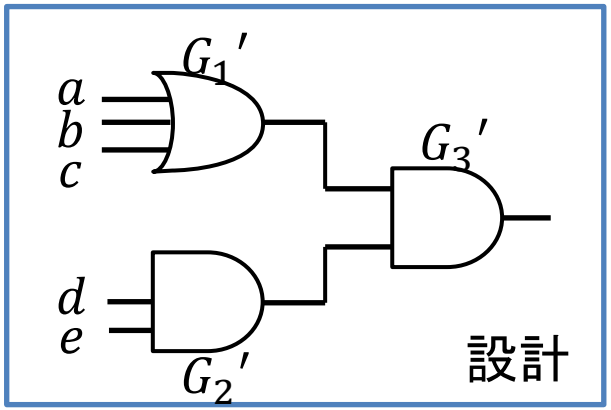


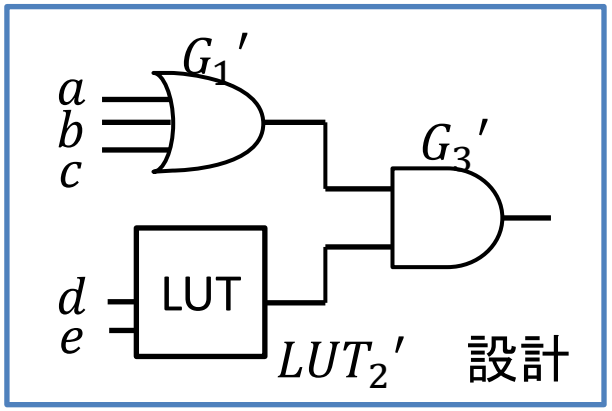
LUTを用いたバグ修正の問題点

- 変数の不足によるバグを修正できない:変数を追加する
- 回路中の変数は多数ある
  :効率よく探索する手法が必要



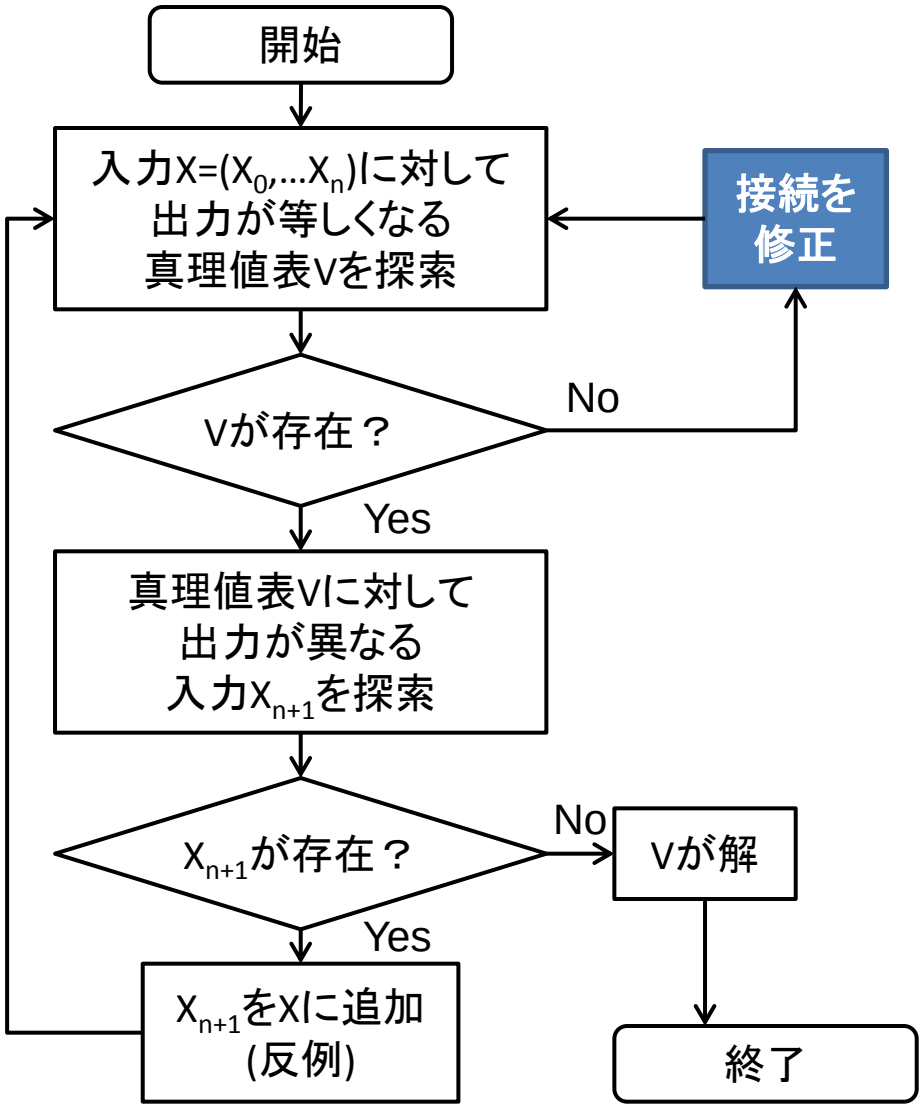
LUTが変数cを入力に
持っていないため、
設計を仕様と等価に
することができない





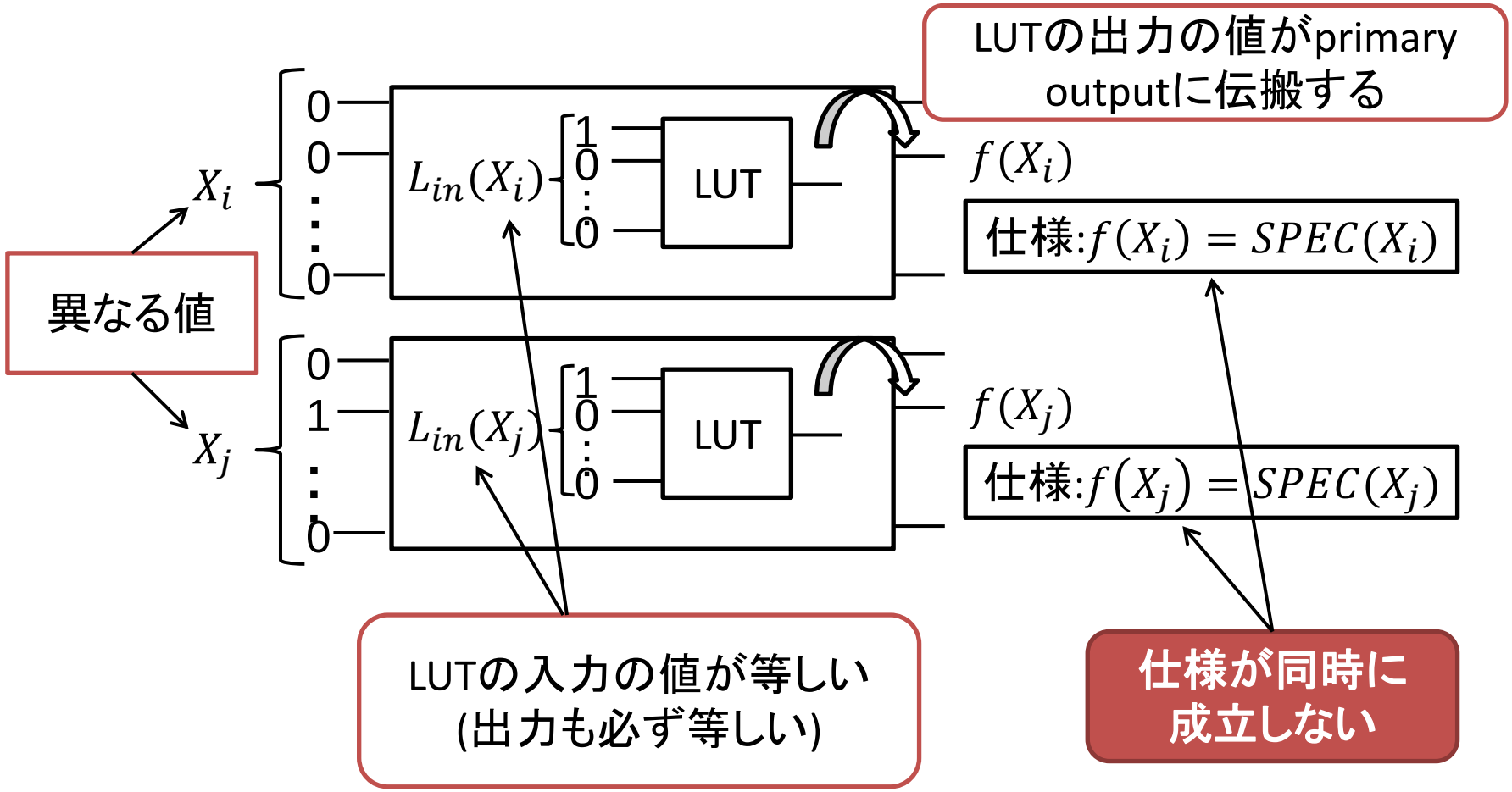
変数値の絞り込み

- LUTに追加する変数を絞り込む
  - 仮定:
    設計中のLUTは1つ
    (複数のLUTがあり、
    いかなるLUTの出力値も
    他のLUTの入力値に伝搬
    しない場合に拡張可能)
  - バグを修正できない場合:
    入力パタンの集合X
    (反例の集合)に
    次のスライドで示す
    条件を満たすものが
    含まれる



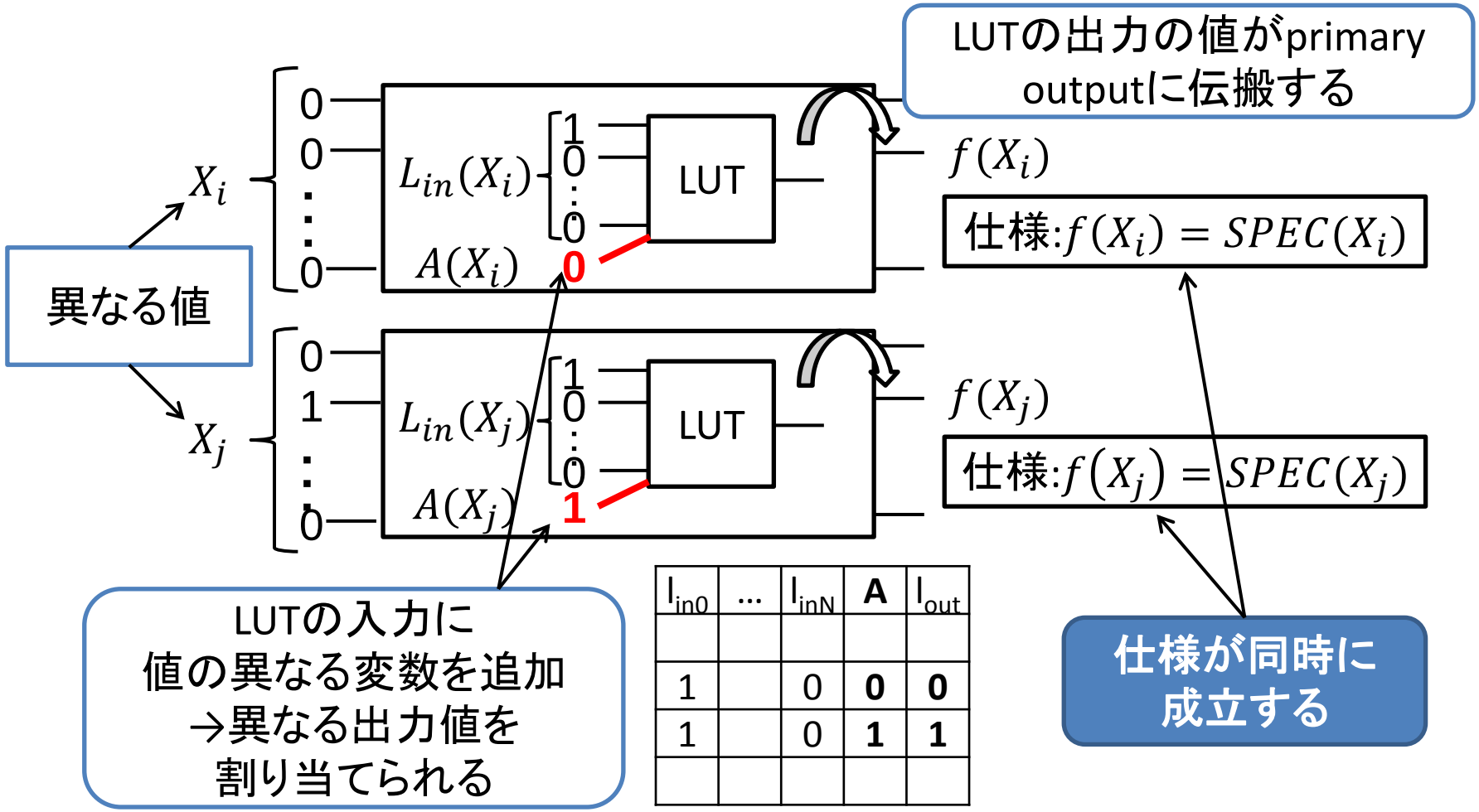
変数値の絞り込み: 修正不可能な場合

- 入力値X_i,X_j が以下の条件を満たすとき、いかなるLUTの真理値表の割り当てによっても修正不可能



変数値の絞り込み: 追加される変数の必要条件

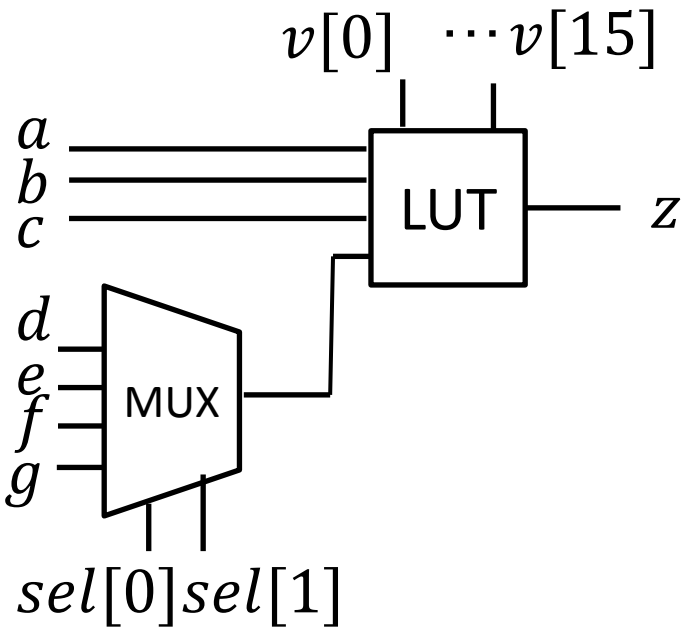
- バグを修正するために追加すべき変数:
  入力X_i,X_j に対して以下の条件を満たす必要がある



I_{in0}	…	I_{inN}	A	I_{out}
1		0	**0**	**0**
1		0	**1**	**1**

マルチプレクサを利用した探索の効率化

- マルチプレクサを利用して複数の変数を追加
- 複数の変数について同時に探索可能
- LUTに比べて変数が少なく、効率が良い



論理設計のデバッグ

- 論理回路の1つのゲートをLUTで置き換え
  - 3入力のゲートを置き換え
- LUTの1つの入力を取り除く
- 変数の追加によって設計を仕様と等価にできるかを調べる
- 例題: ISCAS85 ベンチマーク
- 実験環境
  - OS: Linux CPU: Intel Xeon 2.66GHz Memory: 20GB
  - SATソルバ: PicoSAT
  - Verilog→CNFの変換: ABC, AIGER

論理設計のデバッグの結果

- すべての設計が仕様と等価に
- 設計中の変数のうち10%~40%が候補に
- MUXにより時間が80%以上短縮

PI: 回路の入力から追加
Filtering: 変数の絞り込み
Filtering+MUX:
マルチプレクサを利用

回路	変数	手法	成功/回路数	候補数	割合	繰り返し	時間[sec]
c499	243	PI	0/5	N/A	N/A	18.8	46.4
		Filtering	5/5	88.6	36%	14.0	48.3
		Filtering+MUX	5/5	88.6	36%	16.4	2.3
c880	443	PI	1/5	61.0	14%	55.0	80.8
		Filtering	5/5	54.2	12%	10.6	57.0
		Filtering+MUX	5/5	54.2	12%	11.6	2.6
c1355	587	PI	0/5	N/A	N/A	12.0	60.6
		Filtering	5/5	155.8	27%	13.0	227.7
		Filtering+MUX	5/5	155.8	27%	19.6	3.3

論理設計のデバッグの結果

回路	変数	手法	成功/回路数	候補数	割合	繰り返し	時間[sec]
c1908	911	PI	2/5	34.0	4%	119.6	69.2
		Filtering	5/5	194.2	21%	14.0	284.5
		Filter+MUX	5/5	194.2	21%	17.0	3.9
c2670	1194	PI	0/5	N/A	N/A	6.8	708.1
		Filtering	5/5	142.2	12%	12.8	83.2
		Filter+MUX	5/5	142.2	12%	22.4	4.7
c3540	1670	PI	0/5	N/A	N/A	45.2	154.9
		Filtering	5/5	503.8	30%	11.4	915.9
		Filter+MUX	5/5	503.8	30%	20.0	7.8
c5315	2476	PI	0/5	N/A	N/A	30.8	915.5
		Filtering	5/5	324.6	13%	9.8	268.1
		Filter+MUX	5/5	324.6	13%	13.6	8.8
c7552	3604	PI	0/5	N/A	N/A	5.4	1484.3
		Filtering	5/5	1016.0	28%	9.8	3990.1
		Filter+MUX	5/5	1016.0	28%	16.6	15.9

実設計のデバッグ

- 例題: コントローラ回路のRTL設計
  - 実際の設計で行われた変更をバグとする
- 合成された論理回路に対し手法を適用
- 順序回路を3サイクル分の組み合わせ回路に展開
- デバッグに成功
  - 提案手法により探索時間が短縮された

回路	セル数	変数数	手法	候補数	時間[sec]
Industrial	8289	3209	PI	395	>5h
			Filtering	100	972.3
			Filtering+MUX	100	172.5

RTL設計のデバッグ

- RTL設計に対する性能を評価
- 例題: ARM Cortex Microprocessor
- RTL設計にバグを挿入

バグの種類	仕様	バグ
論理の誤り	A = B & C	A = B | C
余剰な変数	A = B & C	A = B & C & D
変数の誤り	A = B & C	A = B & D
変数の不足	A = B & C	A = B

- Synopsys Inc. Design Compiler を用いて合成
- Synopsys Inc. Formality によりバグ位置の候補を特定

RTL設計のデバッグ

- 例題: ARM Cortex Microprocessor
  - FFを取り除く
  - FFの出力を回路のprimary input、FFへの入力を回路のprimary outputとして等価性検証
  - 入力: 895　出力: 923　セル数: 9,545　変数数: 10,500
- 実験環境
  - OS: Linux CPU: Intel Xeon 2.66GHz Memory: 20GB
  - SATソルバ: MiniSat
  - Verilog→CNFの変換: ABC, AIGER

RTL設計のデバッグの結果

- バグの種類によって異なる結果
  - 余剰な変数のバグにおいて多くの場合成功
  - その他のバグは3~4個の例題について成功

バグ	成功/回路数	変数の追加	候補数	割合	ループ	時間[sec]	
						成功時	失敗時
論理の誤り	3/10	0	N/A	N/A	10.4	29563.1	213.8
余剰な変数	9/10	0	N/A	N/A	14.1	38141.4	1550.2
変数の誤り	3/10	3	4287.7	40%	13.0	39192.1	173.2
変数の不足	4/10	4	5333.8	50%	39.0	38016.6	480.2

考察: デバッグが行えない場合(1)

- ゲートの順番によりデバッグ不可能な場合がある

仕様:
$x = a \mid b$
$y = c \,\&\, d$
$z = x \,\&\, y$

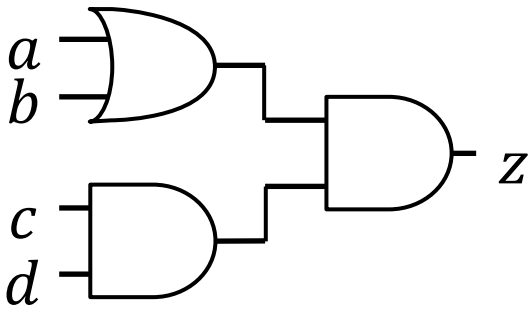
バグを挿入

バグを含む設計:
$x = a \,\&\, b$
$y = c \,\&\, d$
$z = x \,\&\, y$

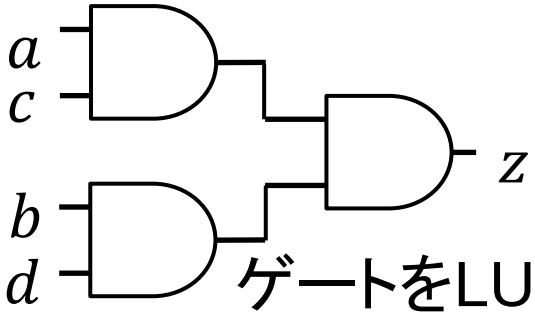
合成

合成

仕様を実現する論理回路



合成結果



ゲートをLUTで置き換えても
仕様を実現できない

考察: デバッグが行えない場合(2)

- 変数が存在しない場合がある

仕様:
$x = a \,\&\, b$
$y = c \,\&\, x$
$z = d \mid x$

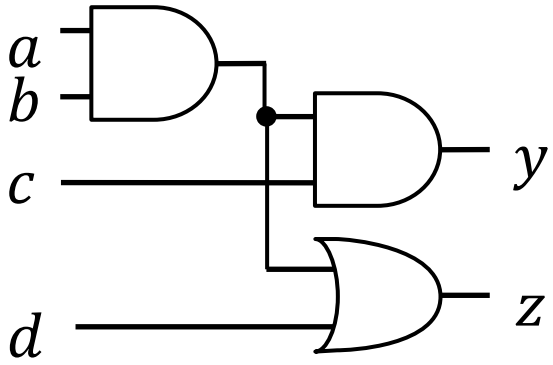
バグを挿入

バグを含む設計:
$x = a \,\&\, b$
$y = c \,\&\, x$
$z = d \mid c$

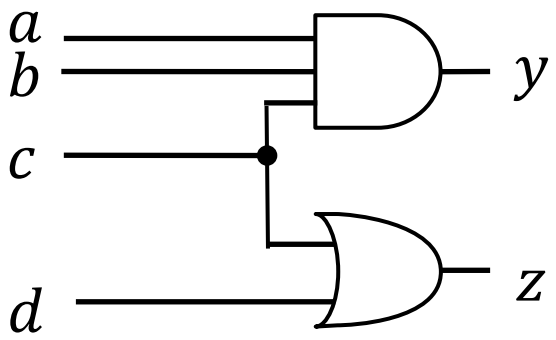
合成

合成

仕様を実現する論理回路



合成結果



追加すべき
変数が
存在しない

今後の課題

- 接続順の入れ替えとLUTの拡大
  - より多くのバグに対応
  - 計算量は増加する
- 複数の変数を追加
  - 3つ以上の反例から絞り込みを行う
- LUT以外の回路を用いた計算量の削減