General and efficient SAT-based ATPG framework for multiple various faults its application to logic synthesis

Masahiro Fujita

University of Tokyo

(joint research with Alan and Bob of UCB)

Here only considers combinational circuits

Long history with Ed

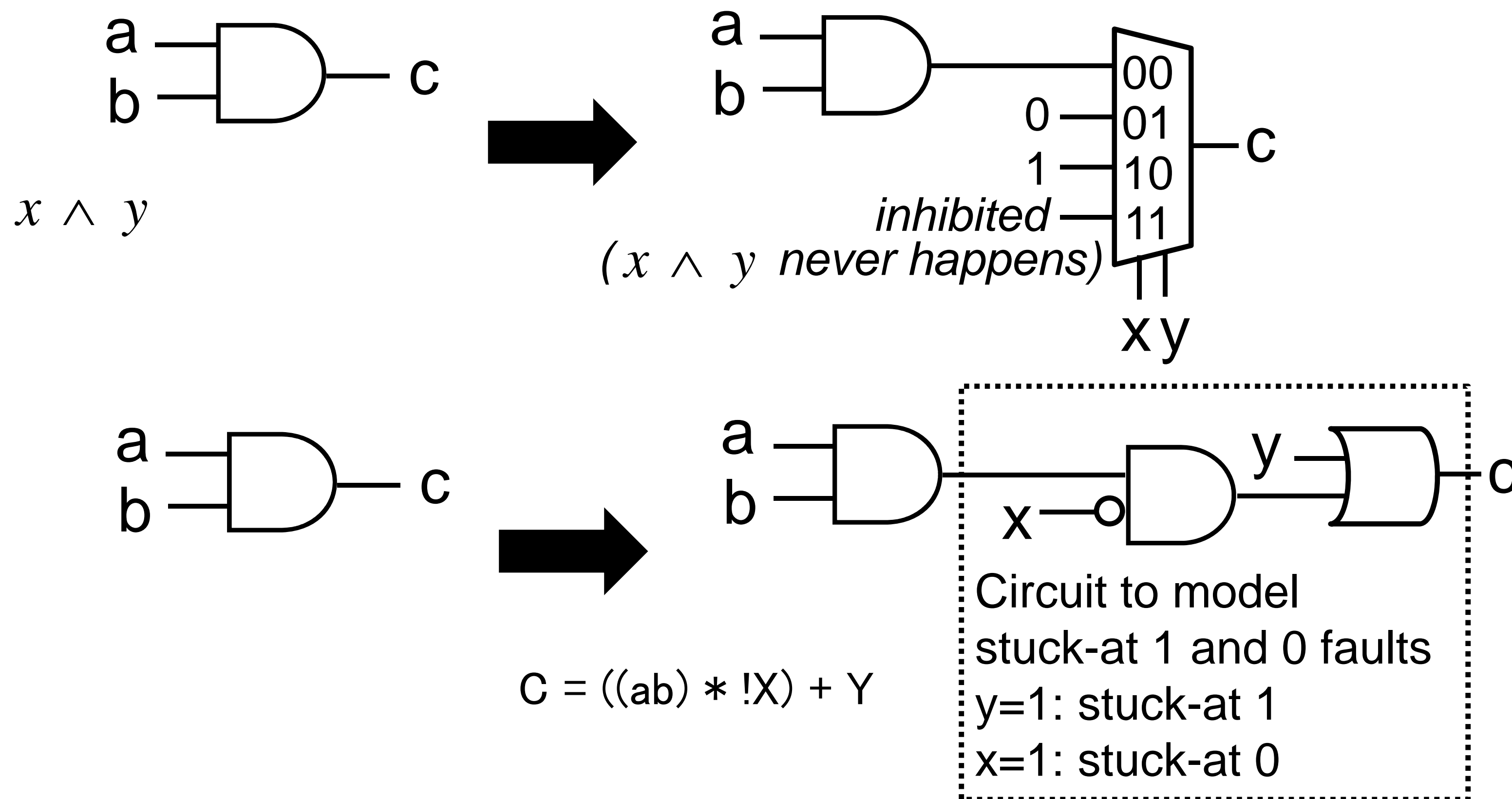
- First met Ed at a conference (CHDL) in May, 1983
  - Presented a paper on sort of model checking hardware with Prolog implementation
  - Following Japanese fifth generation computer project
- Ed approached me and said, "We are working on somehow similar problem, but our approach is better"
- Since then, we have been collaborating
  - Have several jointly authored papers
- I am not a student, pos-doc, long-time visitor. I am just a frequent short-time visitor

Testing manufacturing faults

- Make sure that manufactured chips behaves as described in the design descriptions
- Introduce fault models for efficient processing
  - Ways for HW to fail can be pre-determined
- Suppose there are m possibly faulty locations and there are p ways of faults for each location
  - Single fault assumption: total number of fault combinations is $m*p$
  - Multiple fault assumption: total number of fault combinations is p^m-1
- Generation of complete test vectors for multiple faults was (is) believed to be very difficult

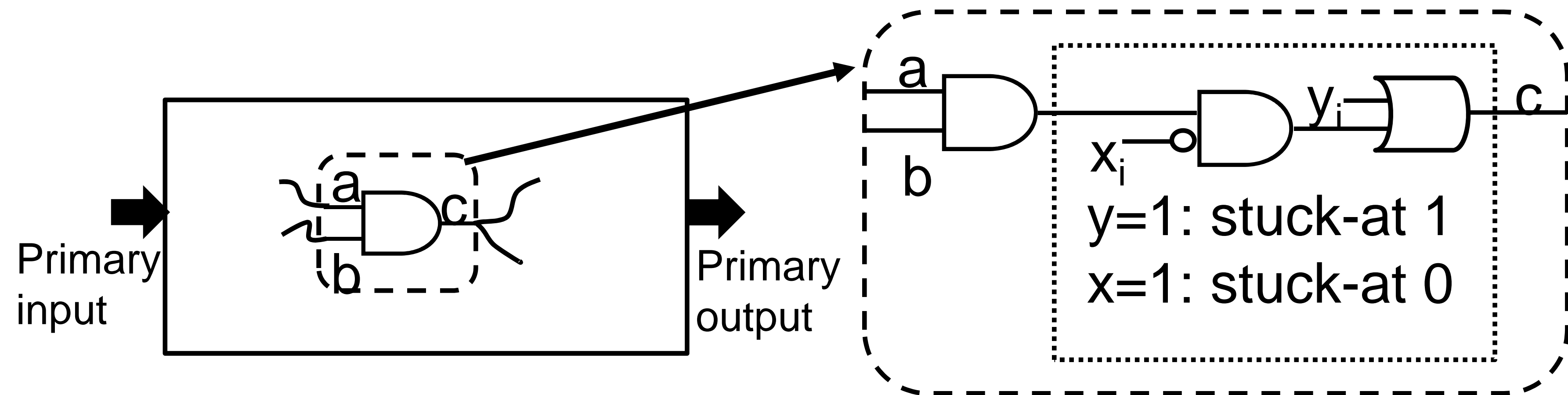
How to represent faults "implicitly as part of SAT problem ?

- Introduce circuits and variables that represent faults into each possibly faulty location
- For stuck-at fault:



Multiple faults

- For each possibly faulty location, insert the circuit to represent stuck-at faults
- If all of x_i and y_i are 0, no fault in the circuit
  - Can represent all fault combinations: $3^m - 1$



- Circuit transformation can be defined in the same way: AND -> (AND, OR, NAND, NOR, EOR, …)
  - Exponentially many transformations are considered

ATPG with SAT

X: Faults

In: Inputs

- Under some inputs, some faults can be detected

$$\exists In.X.Faulty(In, X) \neq NoFault(In)$$

Circuits with faults

Circuit with no fault

=> SAT solution is (in_1, x_1)

Fault x_1 can be detected by input in_1

- There are many techniques based on circuit analysis for much more efficient SAT-based ATPG
  - But here we use this very simple one…

How to eliminate already detected faults

- Under some inputs, some faults can be detected

$$\exists In.X.Faulty(In, X) \neq NoFault(In)$$

=> SAT solution is (in_1, x_1)

Any solution for X corresponds to a detectable fault

- Faults that cannot be detected by in_1

$$\exists X.Faulty(in_1, X) = NoFault(in_1)$$

Under these faults, circuit behave correctly

- When generating the next test vector, add the above constraints
  - Then we are targeting only remaining faults !
  - Should continue until the resulting SAT becomes UNSAT

The problem is essentially an incremental SAT

- $(in_1, in_2, \ldots, in_n)$ are complete test vectors for multiple stuck-at faults

$\exists In.X.Faulty(In, X) \neq NoFault(In)$ => SAT, solution is (in_1,x_1)

$\exists In.X.Faulty(In, X) \neq NoFault(In) \wedge Faulty(in_1, X) = NoFault(in_1)$

=> SAT, solution is (in_2,x_2)

$\exists In.X.Faulty(In, X) \neq NoFault(In) \wedge Faulty(in_1, X) = NoFault(in_1)$

$\wedge Fautly(in_2, X) = NoFault(in_2)$ => SAT, solution is (in_3,x_3)

...

$\exists In.X.Faulty(In, X) \neq NoFault(In) \wedge Faulty(in_1, X) = NoFault(in_1)$

$\wedge Fautly(in_2, X) = NoFault(in_2) \wedge \ldots \wedge Fautly(in_{n-1}, X) = NoFault(in_{n-1})$

=> SAT, solution is (in_n,x_n)

$\exists In.X.Faulty(In, X) \neq NoFault(In) \wedge Faulty(in_1, X) = NoFault(in_1)$

$\wedge Fautly(in_2, X) = NoFault(in_2) \wedge \ldots \wedge Fautly(in_{n-1}, X) = NoFault(in_{n-1})$

$\wedge Fautly(in_n, X) = NoFault(in_n)$ => UNSAT

Recent findings

- Numbers of complete test pattern for single and multiple faults are not much different
  - Need a little bit more test patterns for multiple faults
- ATPG (automatic test pattern generation) is not so much inefficient
  - Entire process of ATPG for multiple faults can be formulated as single incremental SAT problem
- Test patterns generated guarantee 100% correctness
  - Very small numbers of test patterns are sufficient for typical fault models (always less than 5,000 !?)
  - Why ? The ways for HW to fail is prefixed (but exponentially many ways)

Formal analysis with $(in_1, in_2, \ldots, in_n)$

- If the circuit is correct with $(in_1, in_2, \ldots, in_n)$, it is guaranteed to be correct for all input patters
- Why ?
  - The ways for circuits to be buggy/faulty are controlled by *X* variables (parameter variables)
  - Circuits cannot change themselves freely
  - Instead must follow the possible values of *X*
  - *This dramatically reduced the ways to fail*
  - But multiple bugs are take care
  - *The ways to fail are exponentially many*

A little bit surprise

- If we start ATPG for multiple faults with the sets of test vectors for single faults, we do not need many more test vectors !

Name	Test SSA	Tests Multiple SA (reading tests ssa)					Additional Tests
		Vars	Clauses	Conflicts	Tests	Time (s)	
s1423	25	36689	57739	1519	25	0.09	0
s1196	117	150946	246265	798	116	0.3	−1
s1238	130	186570	389819	2882	130	1.53	0
s1488	108	171621	270965	368	107	0.35	−1
s1494	110	173752	280337	187	107	0.32	−3
s5378	102	428024	729438	10954	102	3.56	0
s38417	120	3724712	5492811	159859	130	154.03	10
s35932	30	1473112	2175030	30896	44	99.46	14

ATPG with SAT for logic synthesis

X: Circuit transformation (Faults)

In: Inputs

- Under some inputs, some faults can be detected

$$\exists In.X.Faulty(In, X) \neq NewSpec(In)$$

Circuit with transformation

New spec to be satisfied

=> SAT solution is (in_1, x_1)

Under input in_1, transformation x_1 behaves differently from spec

- Then how can we come up with transformations by which we can realize the spec ?
  - Key observation: Redundant faults

By the way

- International Test Conference (ITC) has been organized for more than 30 years
- It has been dealing with “hardware” testing in general
- But like to extend the scope to include “software” testing as well
- Please consider submitting papers to ITC 2015, which will be Disneyland (Los Angeles) Hotel in September