Managing Complexity in Design Debugging with Sequential Abstraction and Refinement

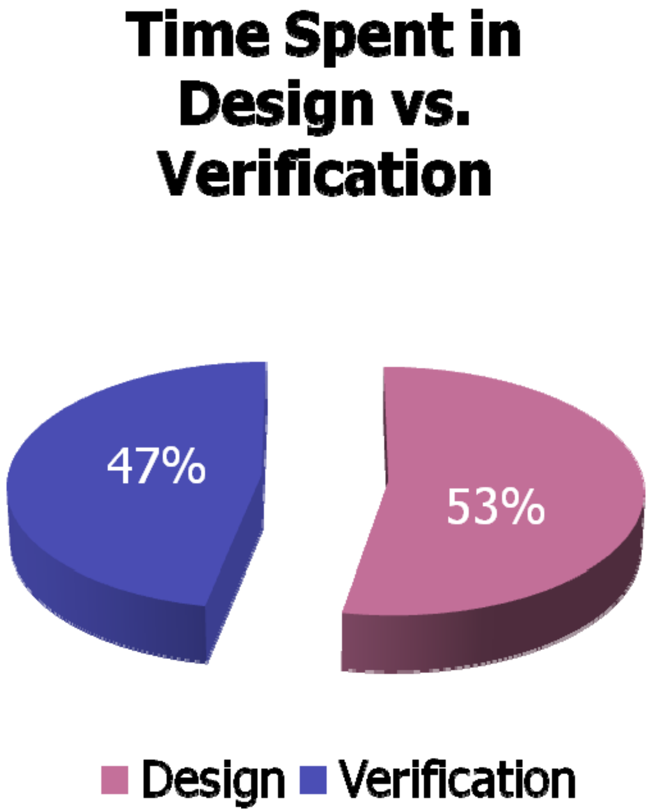
Brian Keng and Andreas Veneris

University of Toronto

ASP-DAC 2011

Motivation

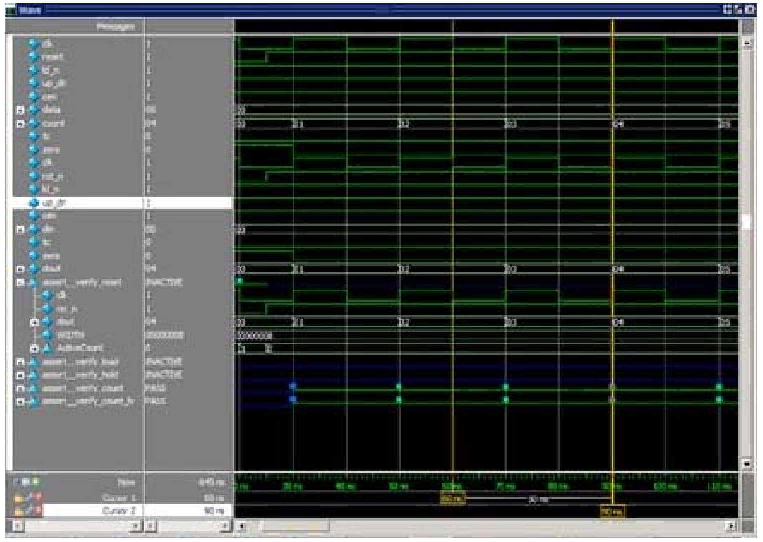
- Functional Verification and Debug are *major* problems
  - Exponentially more costly to find bugs in silicon
  - Functional errors responsible for over 60% of re-spins
  - Trend: Two verification engineers per single designer!
- What's the biggest bottleneck?
  - **Debug**: Takes up to 60% of total verification time



The Debugging Bottleneck

- Functional Debug
  - Localize errors detected during verification
- Bottleneck:
  - Manual process
  - Designs are getting bigger and more complex
  - Consumes 5-7 months of design time per cycle
- How do we address it?
  - **Automation!**

```
...
always @ (*) begin
  if(reset)
    rd6[0:31] <= 32'b0;
  else if(read_active_6)
    rd6[0:31] <= do_6[0:31];
  else if(rck_6)
    // bug orig: {32{1'b1}};
    rd6[0:31] <= 32'b1 ;
end
...
```



Automating Debug

- Automated Debugging
  - Automatically locate places (i.e. *suspects*) in RTL that could fix failure
- Algorithms
  - Simulation-based, BDD-based, SAT-based etc.

$$\text{Complexity} = (\text{design size} * \#\ \text{cycles})^{\#\ \text{errors}}$$

- How can these factors be managed for:
  - Larger circuits?
  - Longer traces?
  - Multiple Errors?

Previous Work and Contributions

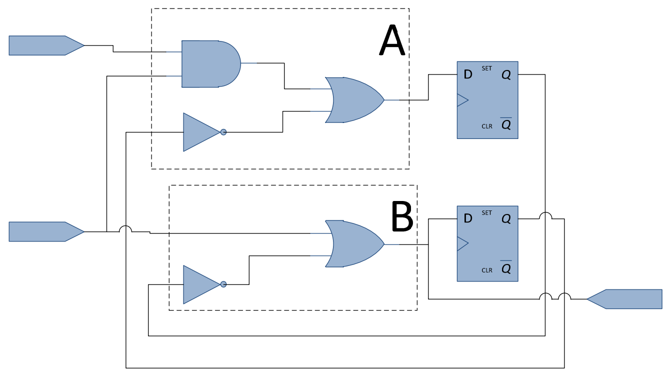
	Previous Work **[Safarpour et al., TCAD09]**	**Contributions**
Abstraction	• Simulated values (Neither over/under-approximation)	• Simulated values to generate an under-approximate model
Refinement	• Solutions for module refinement	• UNSAT cores for time + module refinement
Solutions	• Over-approximation of solutions	• Exact solutions
Error Complexity	• Requires increased error complexity	• No increased error complexity

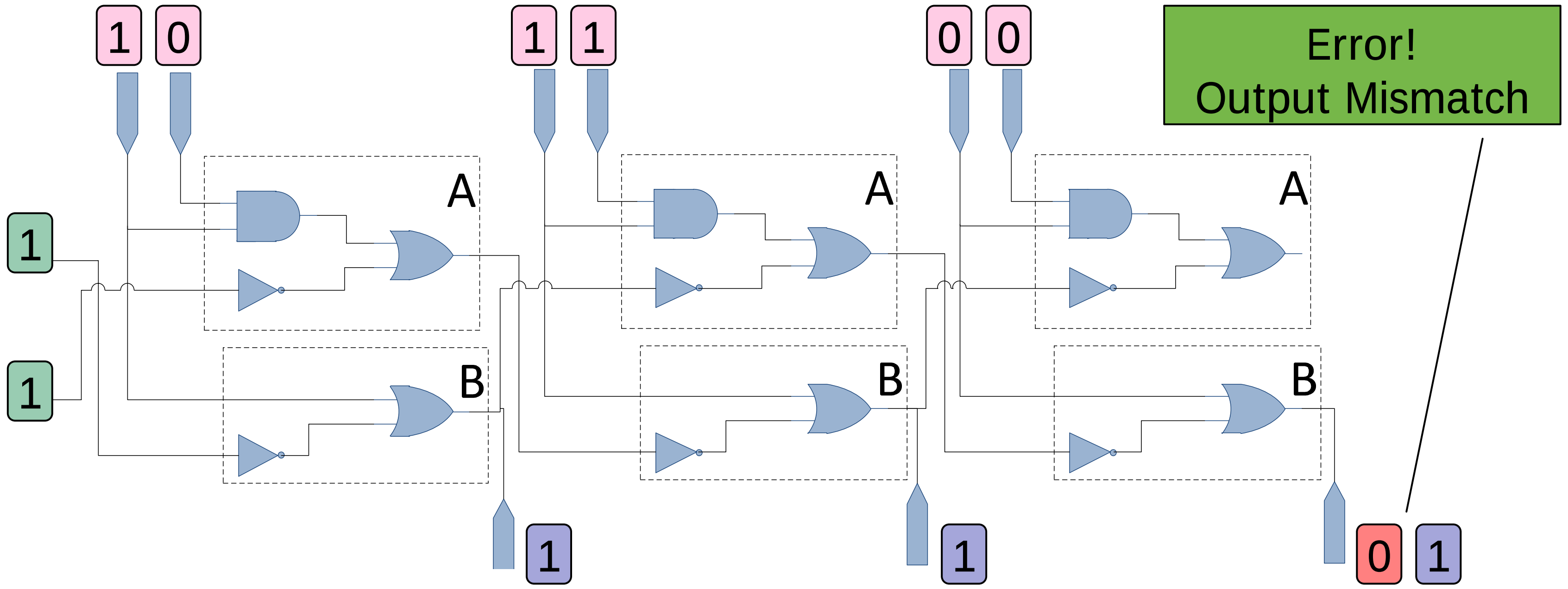
Outline

- **Background**
  - Automated Debugging
  - SAT-based Debugging
  - UNSAT Cores
- Sequential Abstraction and Refinement
- Experiments
- Conclusion

Automated Debugging

- Erroneous circuit
- Error Trace
  - Initial State
  - Primary Inputs
  - Expected Values



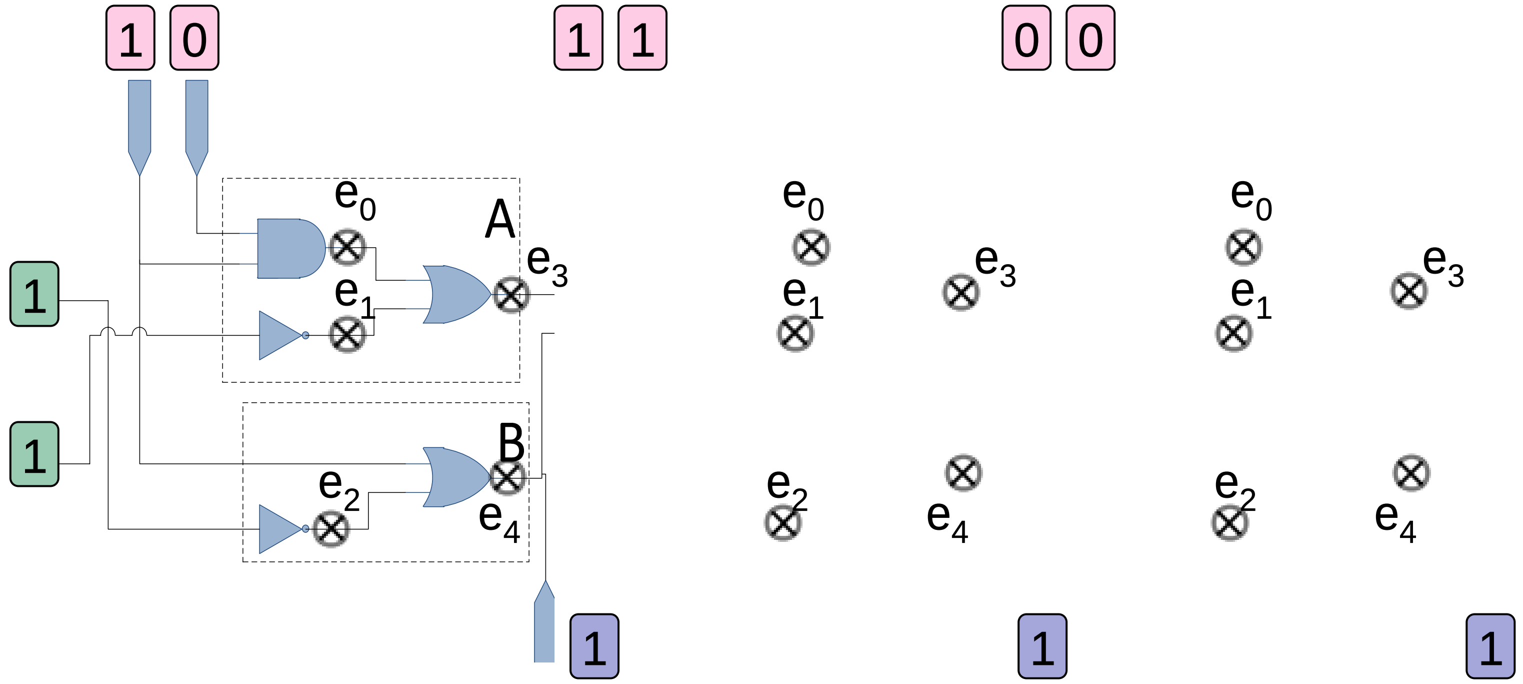


SAT-based Debugging

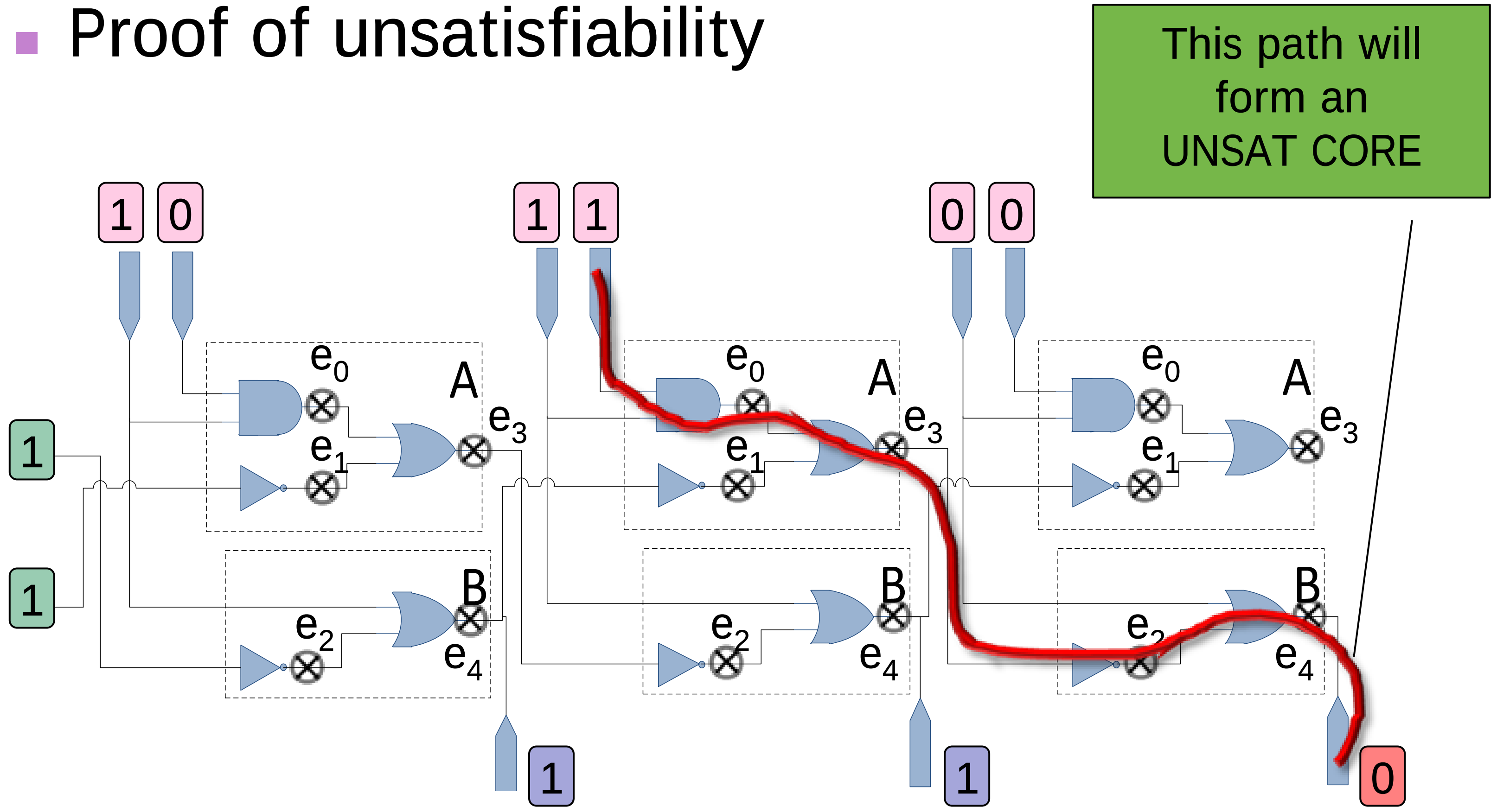
[Smith, et. al TCAD '05]

- 1) Unroll
- 2) Error models (e.g. muxes)
- 3) Constrain initial state, inputs, expected outputs
- 4) Constrain number of errors (error cardinality, N)

SAT when N=1
$\{e_0=1, e_2=1, e_3=1, e_4=1\}$



- UNSAT Cores
  - Subset of clauses that are unsatisfiable
  - Proof of unsatisfiability

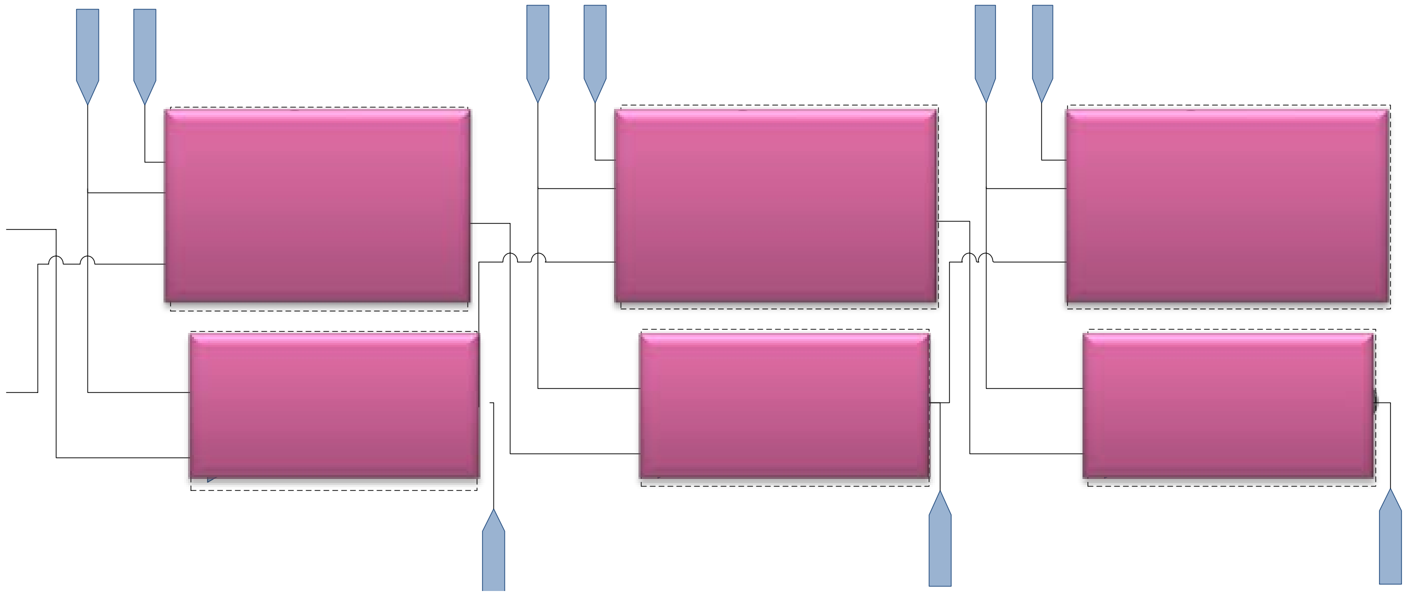


Outline

- Background
- **Sequential Abstraction and Refinement**
  - Overall Algorithm
  - Abstraction
  - Module Refinement
  - Sequential Refinement
  - Comparison to Previous Work
- Experiments
- Conclusion

Overall Algorithm

1. Generate initial abstract model
2. Solve abstract model
3. Analyze UNSAT core:
    1. Exit if UNSAT core has no abstract clauses
    2. Refine using UNSAT core, repeat step 2



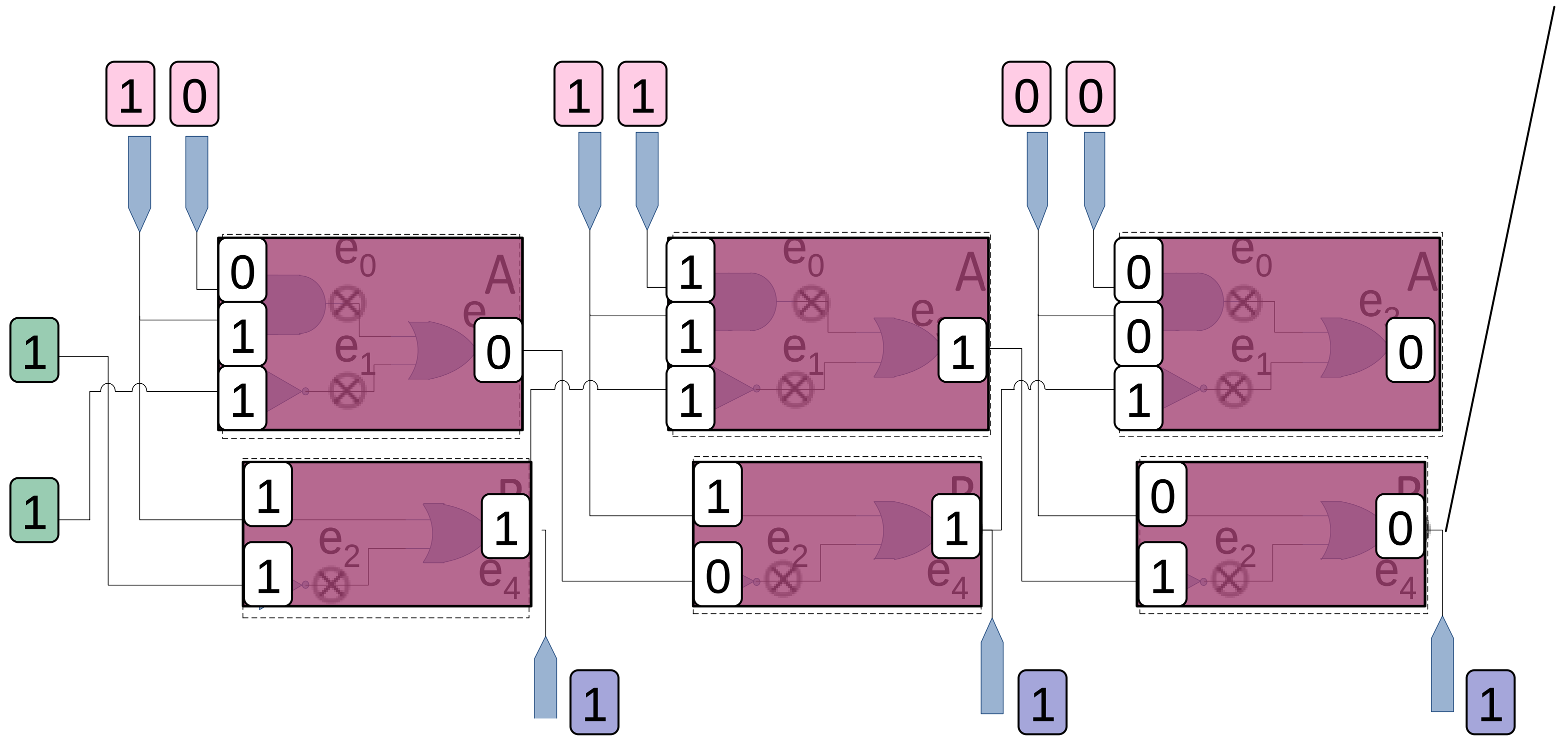
Abstraction

- Abstraction:
  - Replace module constraints in SAT instance with their simulated input/output values
    - Reduce size of SAT instance (design size)
    - Smaller run-time/memory
  - Abstract instance finds a subset of the suspects of the original SAT instance (Under-approximation)
    - Property holds even after refinement
    - No need to find previous found solutions
    - Incremental solving

Abstraction Example

- Replace module constraints with simulated input/output values

Trivially UNSAT



Module Refinement

- Refinement
  - Use UNSAT core to determine which modules to refine
  - In next iteration, do not replace module constraints with simulated values
  - Allows for refinement with the same error cardinality
- Exit condition:
  - When UNSAT core does not contain any abstract input/output values
  - Complete set of solutions without refining entire problem

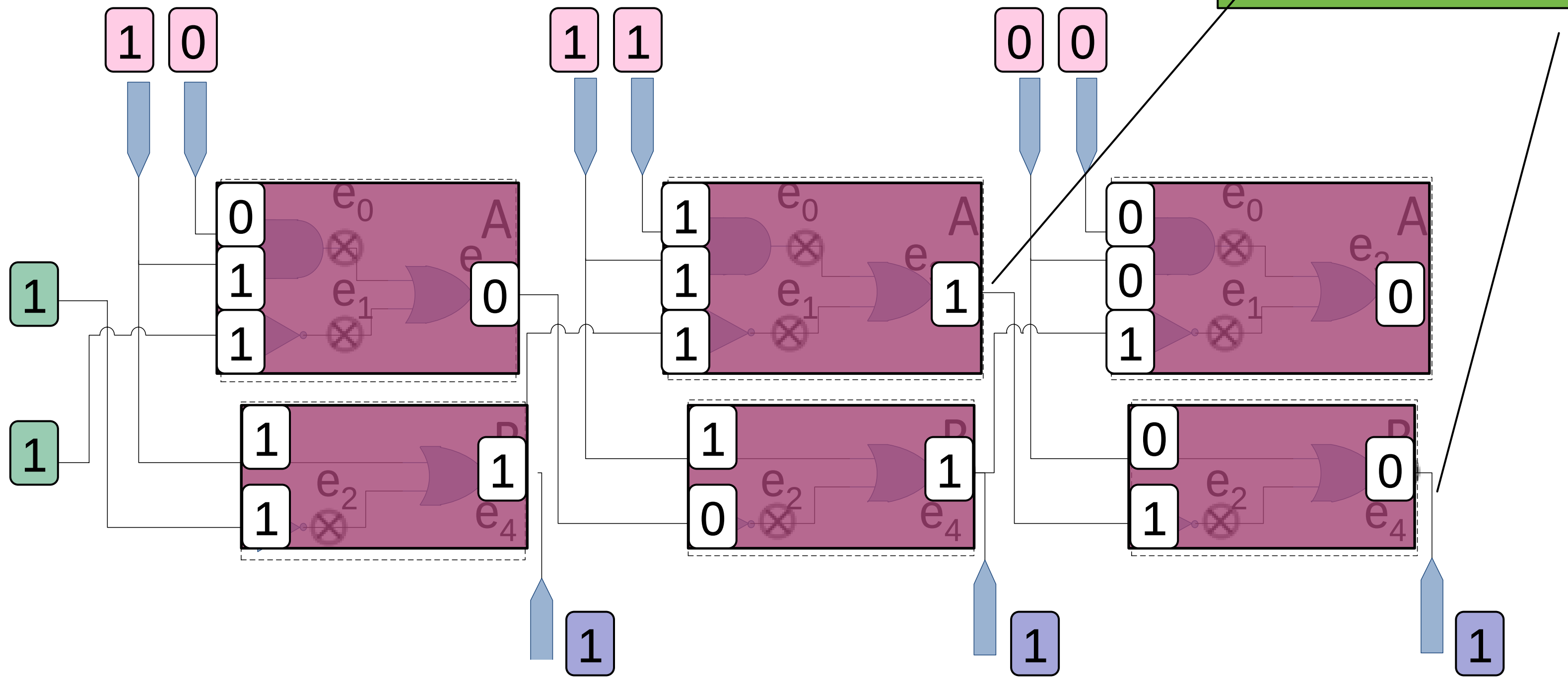
Module Refinement Example

SAT when
$\{e_0=1, e_2=1, e_3=1\}$

SAT when $e_4=1$

UNSAT Core

UNSAT Core



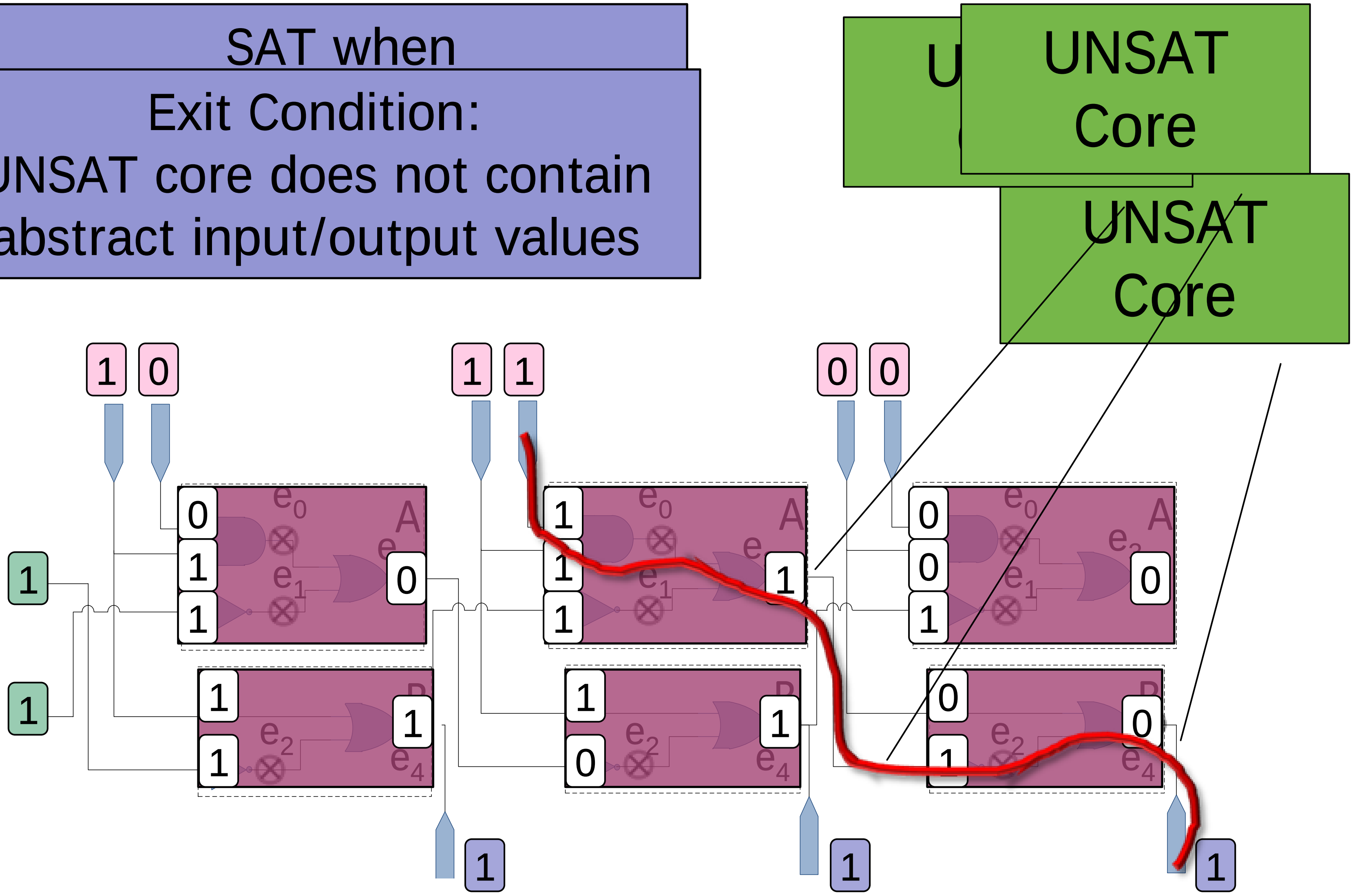
Sequential Refinement

- Sequential Refinement
  - Only refine modules in time-frames that are in UNSAT core
  - Allows fine-grain refinement across time
  - Smaller instances vs. many iterations
  - Use same exit condition as before
- Refine windows
  - Refine all modules around radius *r* involved with the UNSAT core

Sequential Refinement Example

SAT when

Exit Condition:
UNSAT core does not contain abstract input/output values



Comparison to Previous Work

	Previous Work [Safapour et al.]	Sequential Abstraction & Refinement
Abstraction	Neither	Under-approximation
Refinement	Module	Module/Time
Debugging Engine	Any	SAT-based
Exact Solutions	No (over-approximation)	Yes
Error Cardinality	Requires increase	No increase

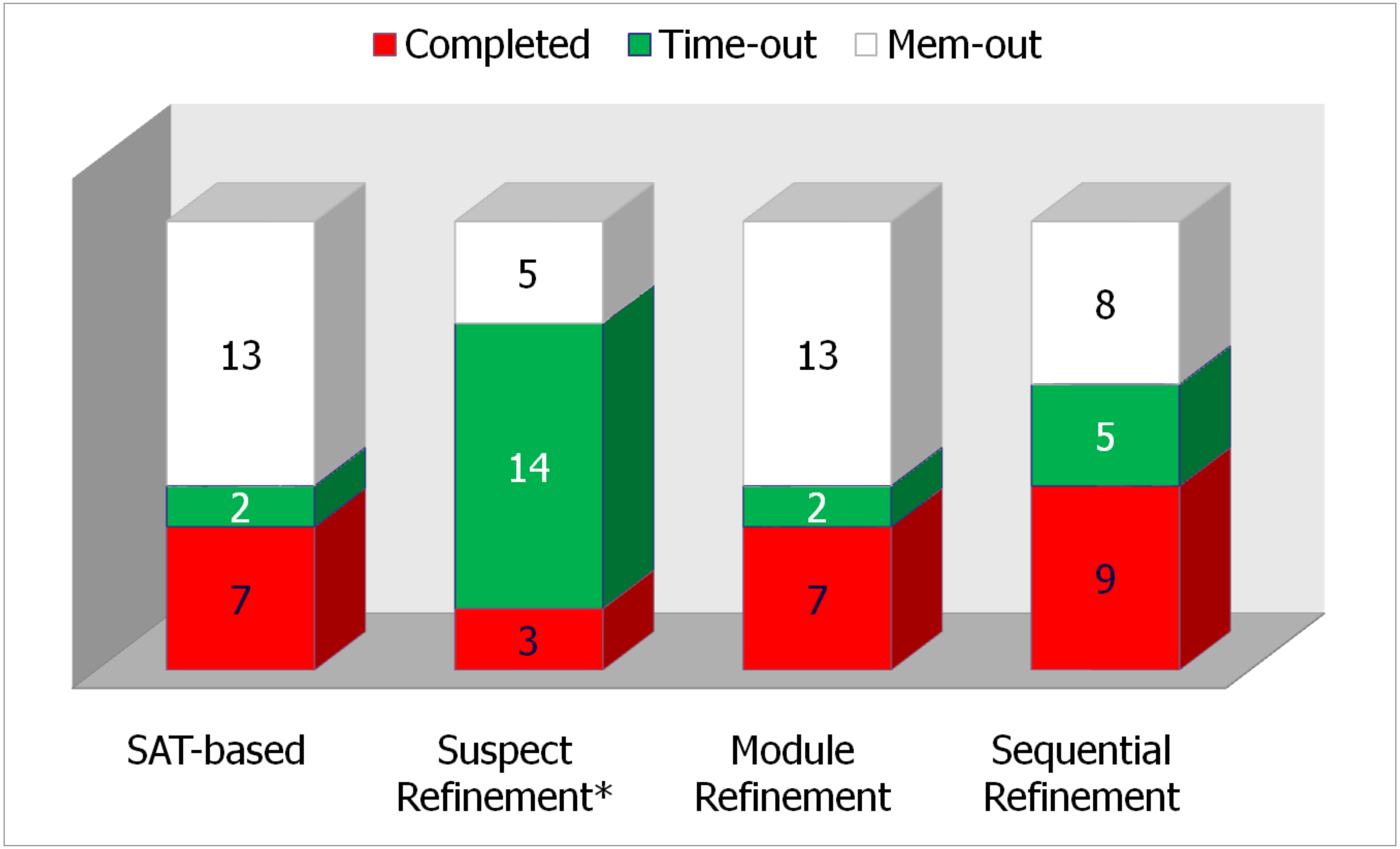
Outline

- Background
- Sequential Abstraction and Refinement
- **Experiments**
  - Experimental Setup
  - Solved Instances
  - Number of Solutions
  - Module vs. Sequential Refinement
- Conclusion

Experimental Setup

- Pentium Core 2, 2.66 Ghz, 8 GB ram
- 10 circuits from OpenCores.org and industrial partners
- Inserted in a typical RTL error
    - Wrong assignment, missing case statement, incorrect operator, etc.
- PicoSAT v913
- Timeout: 3600 seconds
- Sequential Refinement Window: r=20

Solved Instances



* Suspect Refinement [Safarpour et al.]

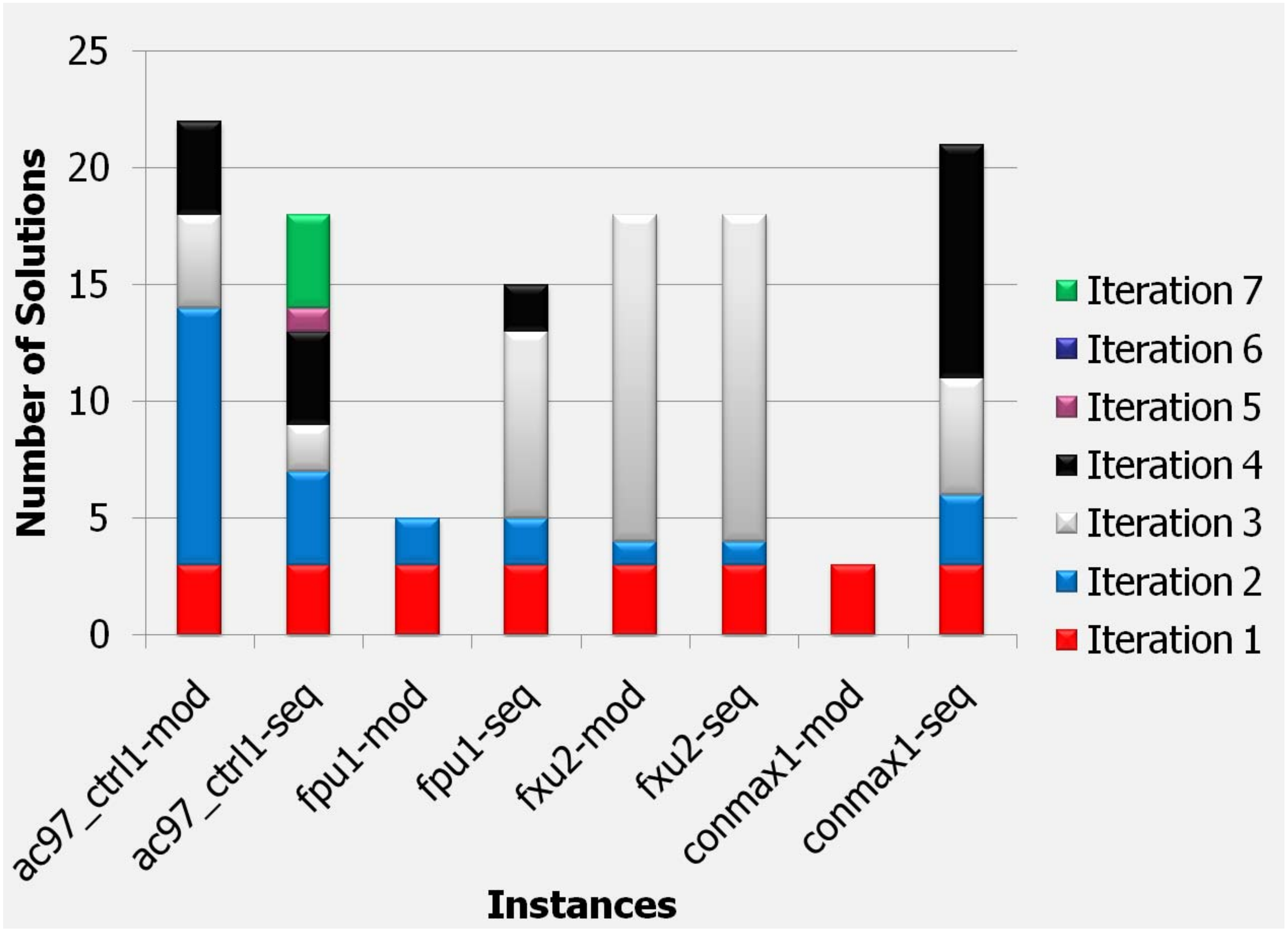
Number of Solutions

	SAT-based	Suspect * Refinement	Module Refinement	Sequential Refinement
conmax1	0	3	3	20
fdct1	0	2450	8	8
fpu1	0	879	5	15
fxu1	24	1313	24	24
s_comm1	0	213	17	17
vga1	0	11	0	14

- Sequential refinement returns solutions for all instances

* Suspect Refinement [Safarpour et al.]

Module vs. Sequential Refinement



Conclusion

- Sequential Abstraction and Refinement
  - Finds exact solutions
  - Under-approximate abstraction
  - UNSAT core based refinement
    - Module refinement
    - Sequential refinement
- Experiments
  - Returns solutions for 100% of instances compared to 41% without the technique