Soft-Error Analysis

Patrick Klampfl

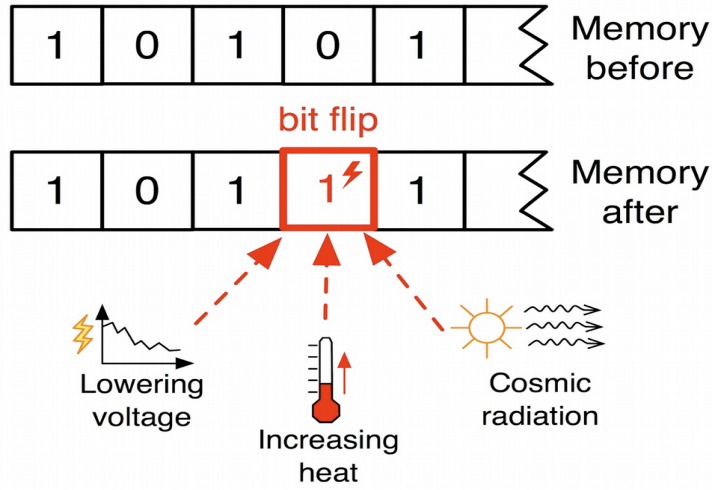
February 1st, 2017

Overview

- Introduction:
  - Soft-Errors
  - Protection Logic
  - Classification of components
- Techniques:
  - simulation vs. formal vs. **semi-formal**
- Algorithms (application of the techniques for classification)
- Benchmark Results

Soft Errors / Faults

- Sequential Boolean Circuits: inputs, AND gates, latches, outputs
- Components (latches, AND gates) can suffer from faults
  - flip truth value
- Single fault assumption

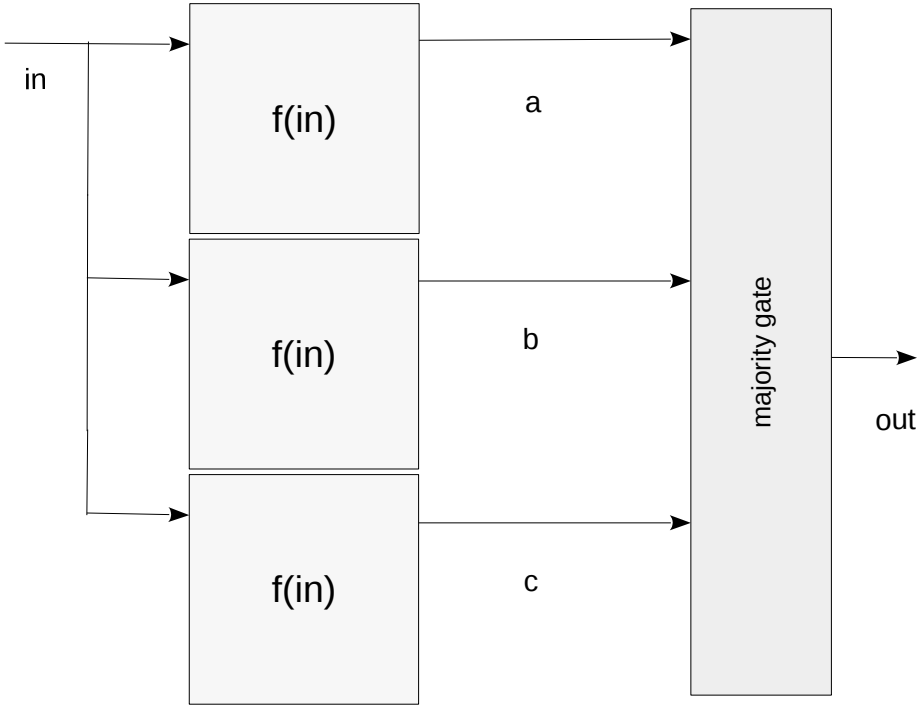


Protection Logic

- Special part of a circuit for reliability
- Detect faults or correct faults
- Examples: Triple Modular Redundancy (TMR), parity-checks, ...

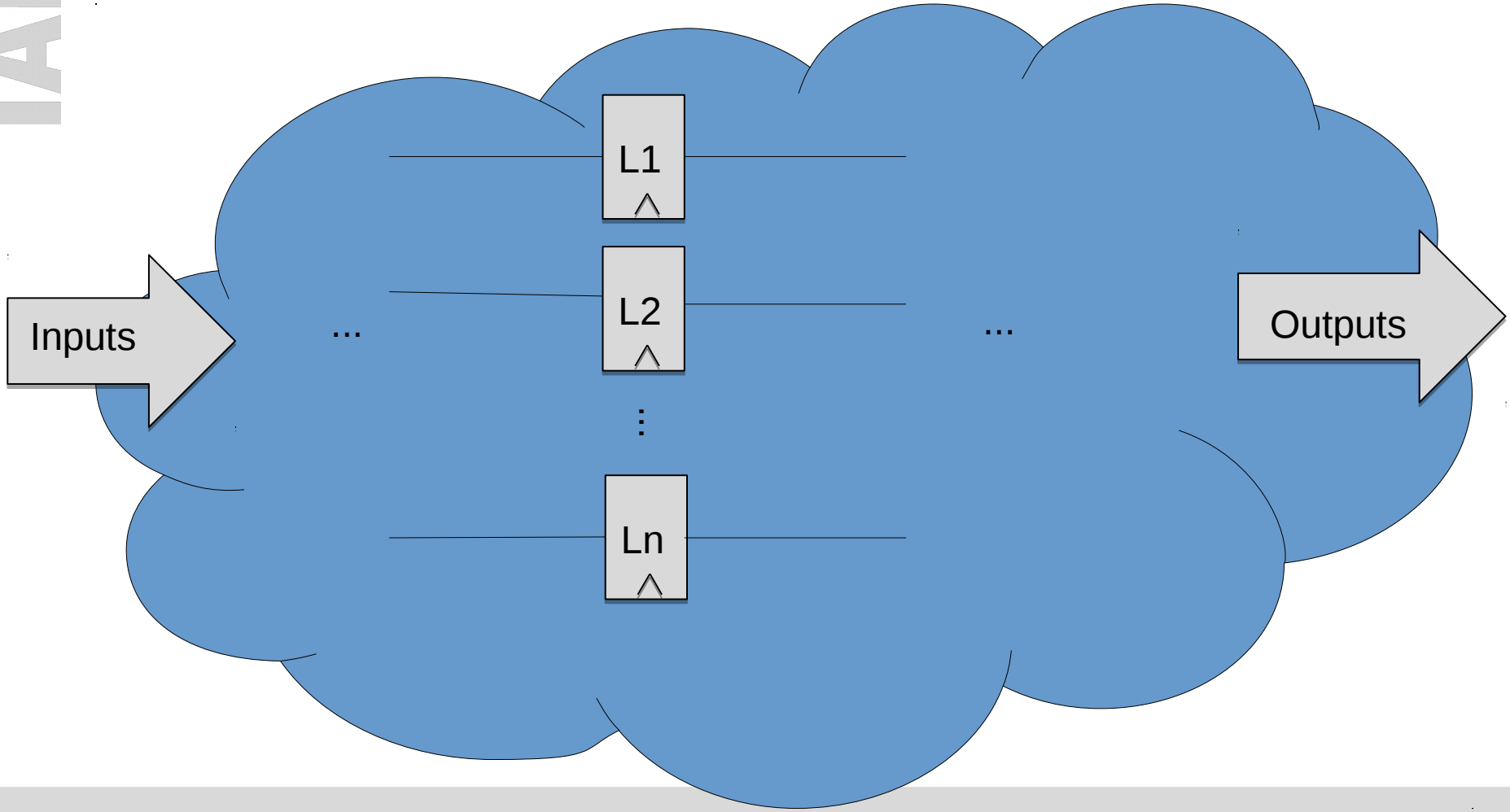
How to **correct** errors: TMR Example

- Example: triple modular redundancy



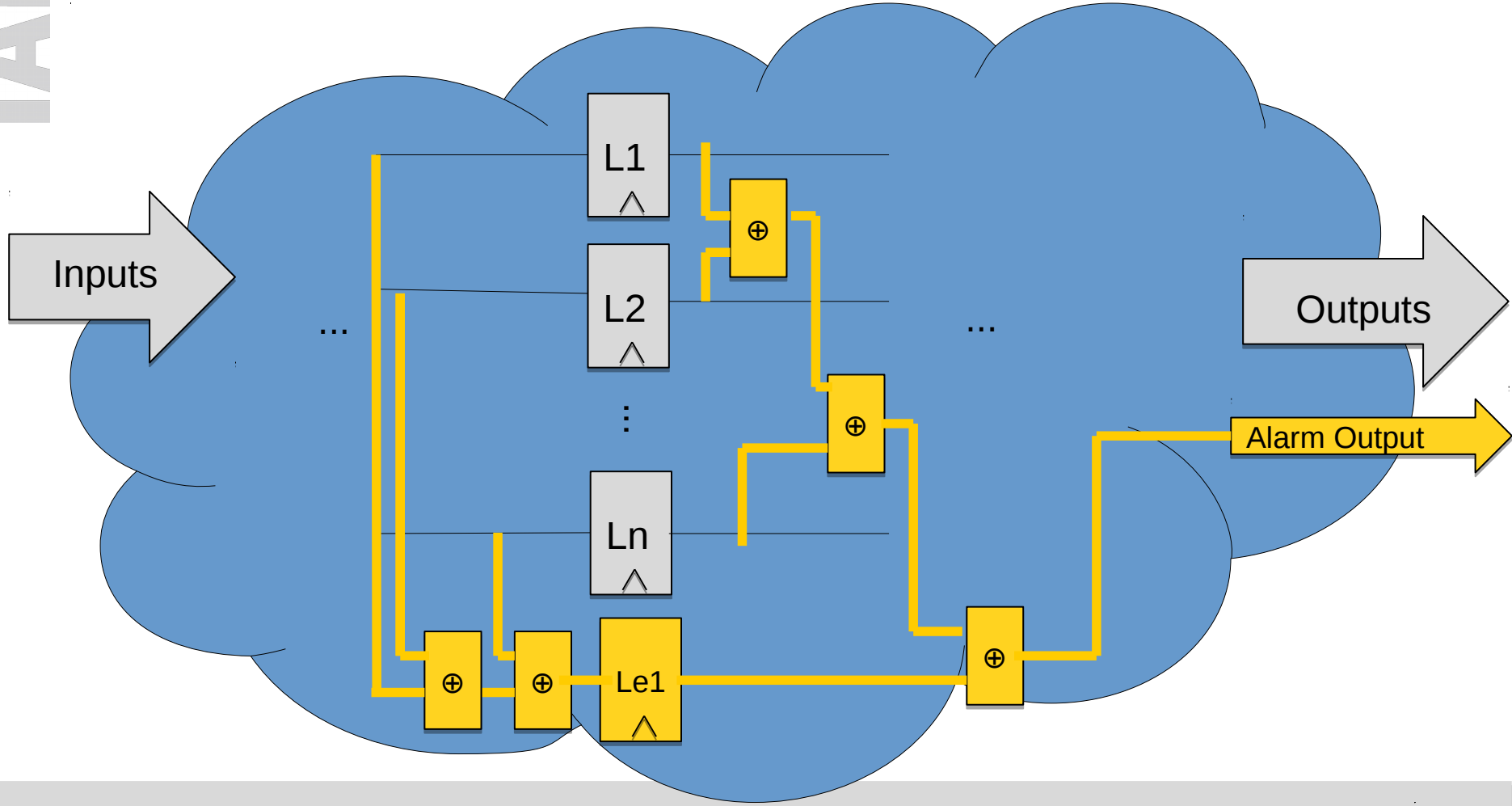
- Con: > 200% overhead in power consumption and area!

How to **detect** errors?



How to detect errors:

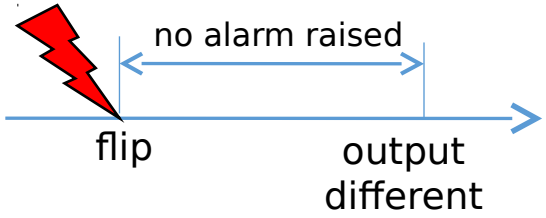
- → add **redundancy.**



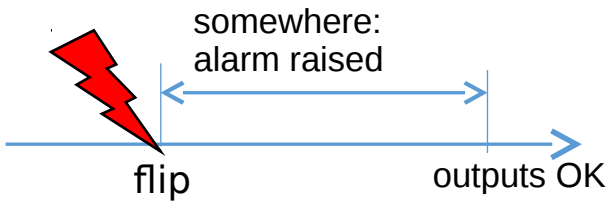
Errors in protection logic

- vulnerable latches
  - Faults that corrupt primary outputs

    → soft error escaped

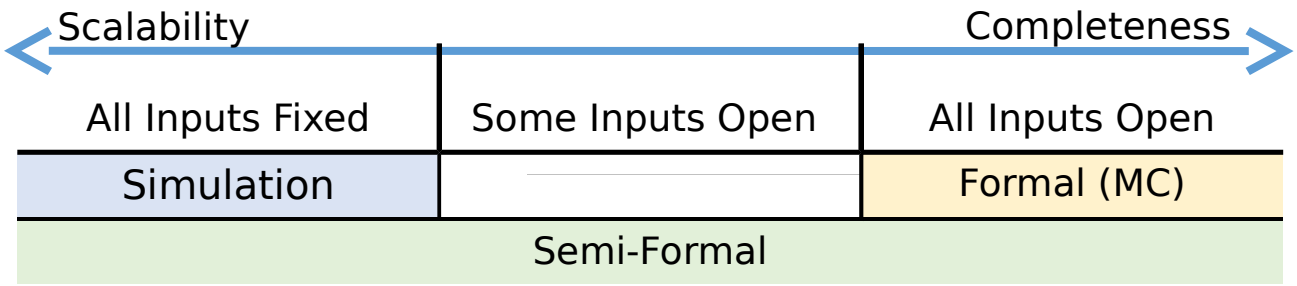


- false positives
  - Unnecessarily reported faults

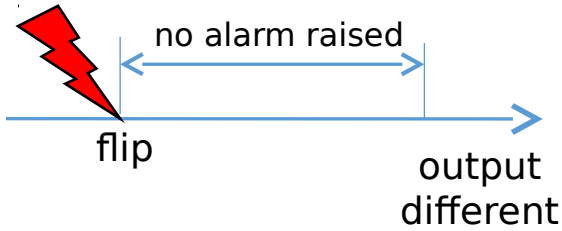


Techniques for Component Classification

- Formal methods
  - Complete
  - Slow, bad scalability
- Semi-Formal methods
  - Flexible
- Simulation
  - Not Complete
  - Fast, good scalability

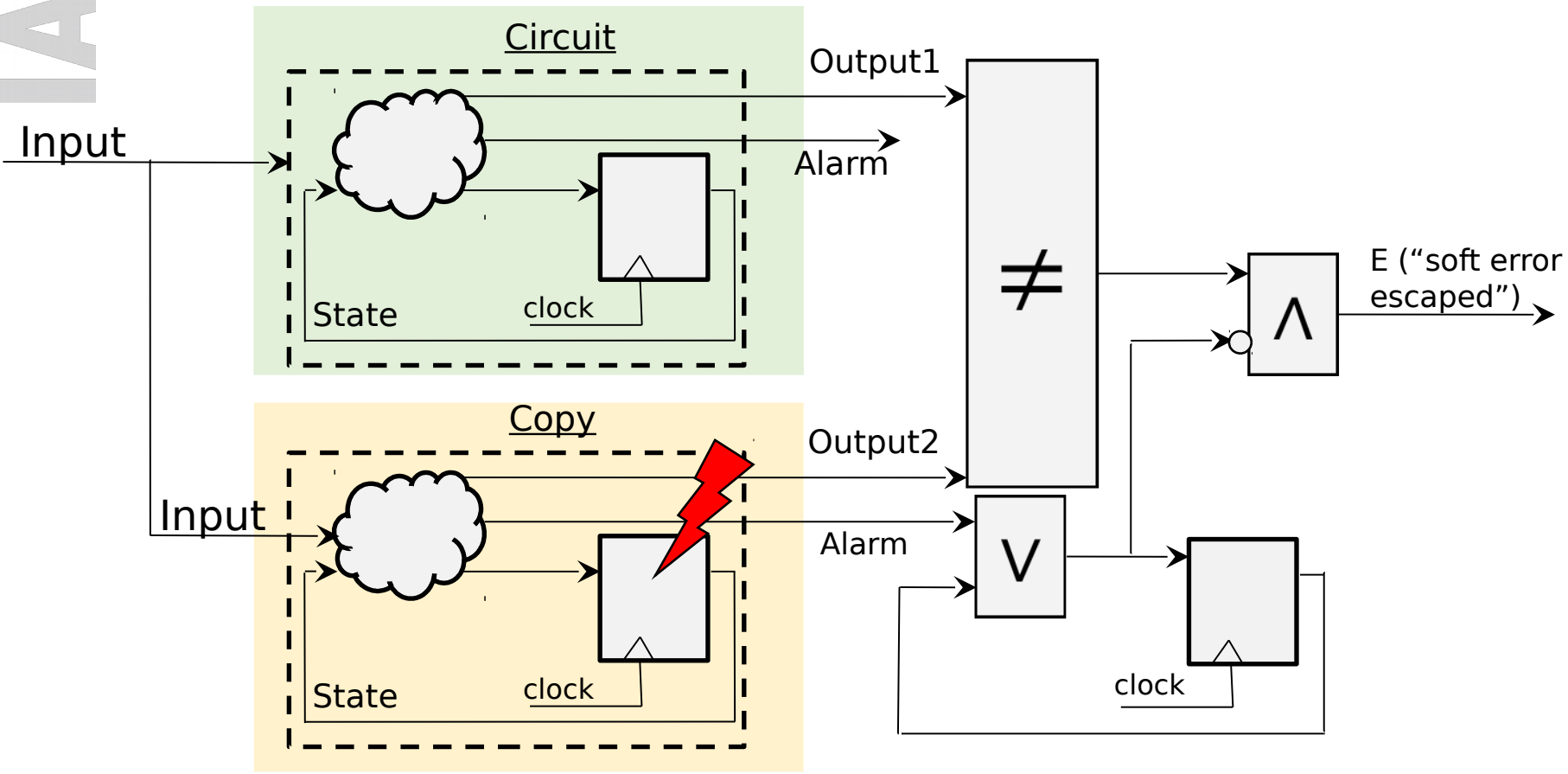


Detecting **vulnerable** latches



- Formal method: (Bounded) Model Checking
- Simulation
- Semi-formal methods

Formal: Model Checking Approach



Simulation (1/2)

- Test case provides input values for the circuit:

```
01101000110
10101100111
...
```

(1) Execute **correct** simulation

(2) Execute **faulty** simulations

- Induce faults
- Compare output values with correct simulation

```
for all latches l in L:
  for all time steps i=1 to len(t):
    // check if soft error injected at
    // latch l in step i of test case t
    // escapes (now or in future steps)
```

Simulation (2/2)

- **Not complete!** (only concrete test cases)
- But: might already find some **vulnerable** latches / **false positives**
- Fast for a **single** test case
- Not suited for **verification**

Semi-Formal approach

- Sequential Equivalence Checking
- SAT-based: symbolically encode the circuit as a formula
- test case-based: Input values from test cases, can be concrete or open

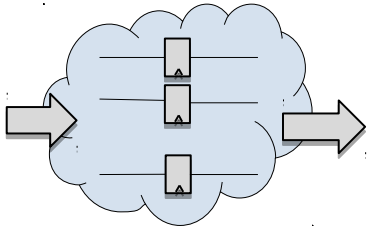
```
011??01000110
101??01100111
...
```

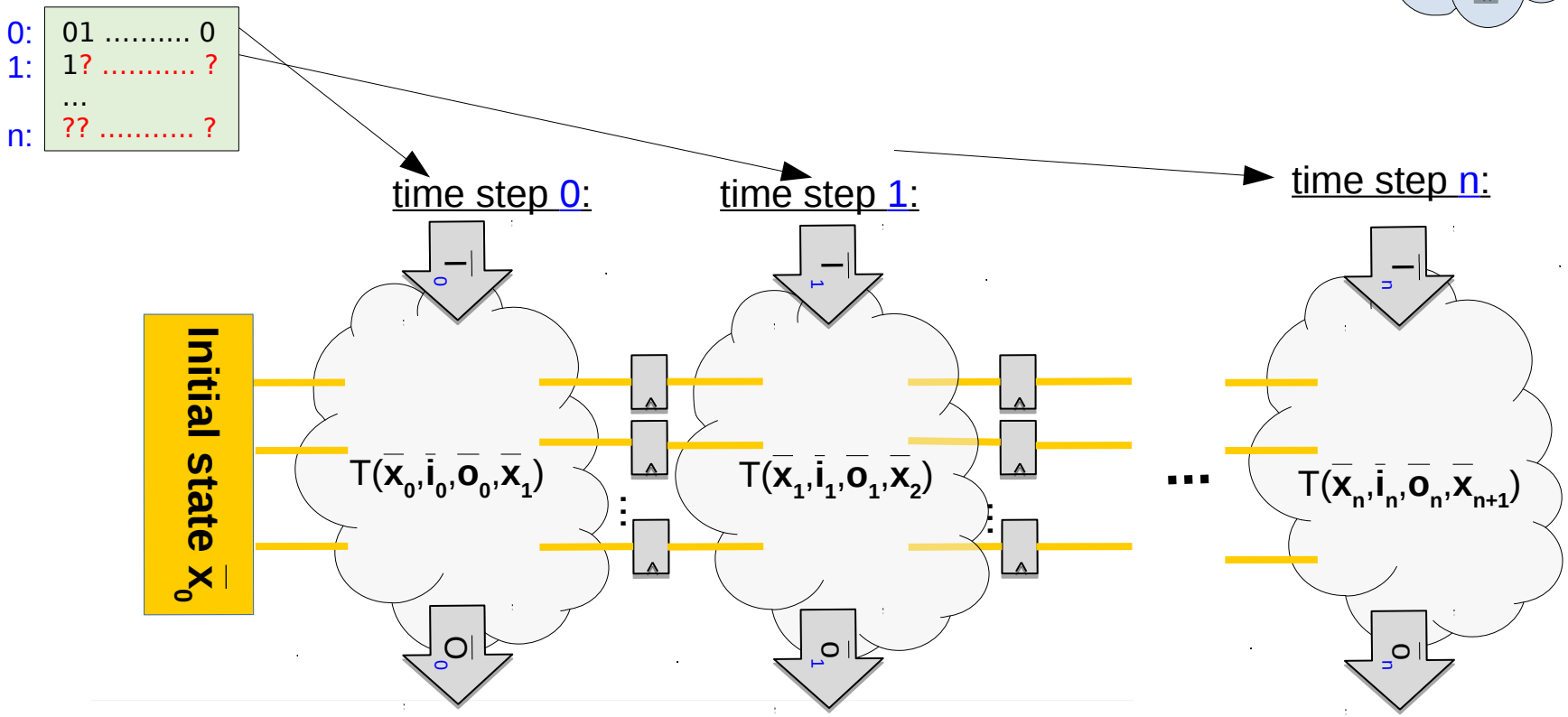
- point in time AND component to flip symbolic

```
for all latches l in L:
  for all time steps i=1 to len(t):
     Flip latch
      //check ...
```

Convert circuit to CNF formula (for SAT-solver)

- Circuit can be converted to a transition relation formula $T(\bar{x},\bar{i},\bar{o},\bar{x'})$
  - talks about **inputs, outputs**, **current**- and **next state**
- Execution of circuit: $T(\bar{x}_0,\bar{i}_0,\bar{o}_0,\bar{x}_1) \wedge T(\bar{x}_1,\bar{i}_1,\bar{o}_1,\bar{x}_2) \wedge \ldots \wedge T(\bar{x}_n,\bar{i}_n,\bar{o}_n,\bar{x}_{n+1})$





Semi-Formal Algorithm

- Sequential Equivalence Checking: compare **fault-free** and **faulty** circuit
- SAT-based: symbolically encode the execution of both circuits as a formula:
  - fault-free: $T(\bar{x}_0, \bar{i}_0, \bar{o}_0, \bar{x}_1) \wedge T(\bar{x}_1, \bar{i}_1, \bar{o}_1, \bar{x}_2) \wedge \ldots \wedge T(\bar{x}_n, \bar{i}_n, \bar{o}_n, \bar{x}_{n+1})$
  - faulty: $T(f_0, \bar{c}, \bar{x'}_0, \bar{i}_0, \bar{o'}_0, \bar{x'}_1) \wedge T(f_1, \bar{c}, \bar{x'}_1, \bar{i}_1, \bar{o'}_1, \bar{x'}_2) \wedge \ldots \wedge T(f_n, \bar{c}, \bar{x'}_n, \bar{i}_n, \bar{o'}_n, \bar{x'}_{n+1})$
  - $\bar{c}$ defines **which** latch should be flipped, $\bar{f}$ defines the point in time **when** it should be flipped
- test case-based: Input values $\bar{i}$ from test cases, can be concrete or open

```
011??01000110
101??01100111
...
```

SAT-query to find vulnerable components

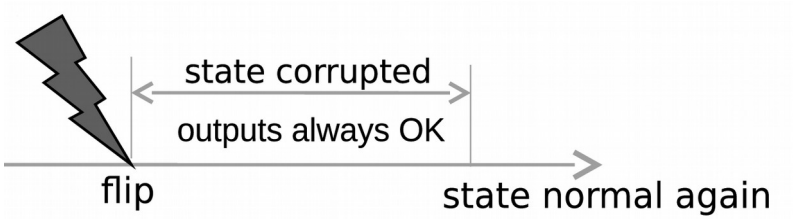
```
for all time steps i=1 to len(t):
    “Dear SAT solver, can you produce a wrong output at step i without
     alarm raised by flipping any latch in some earlier step?”
```

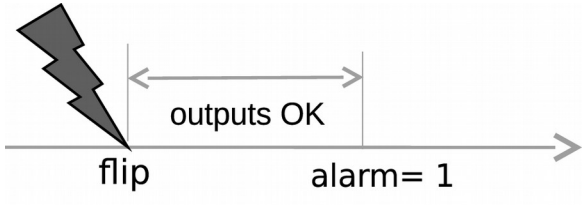
- If SATISFIABLE:

assignment: ¬f0, **f1**, ¬f2, .. , ¬fn, ¬c1, ¬c2, **c3**, .., ¬cm

  - c3: Latch 3 is flipped
  - f1: it is flipped in time step 1
  - Output wrong in step i (at query time)

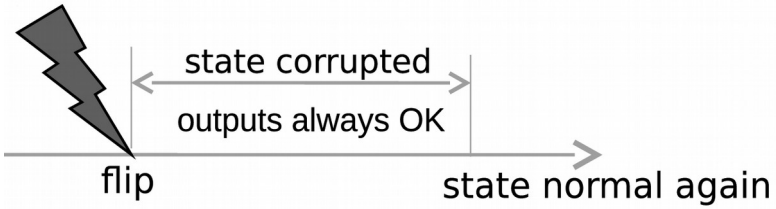
Detecting **definitely protected** latches

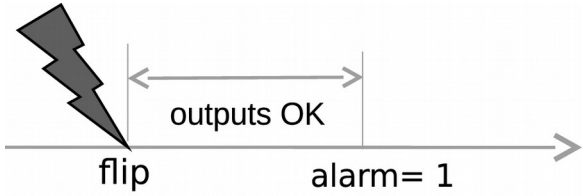




- must hold in **any** state!
- Definitely protected → not vulnerable
- Not definitely protected → ??? (maybe protected, maybe vulnerable..)

Detecting **definitely protected** latches





- Check property using a SAT-solver

- **k-step protection** of a latch**:**

  A fault in **any** state (an over-approximation of all states) ...

  a) … either gets masked out within at most **k** time steps

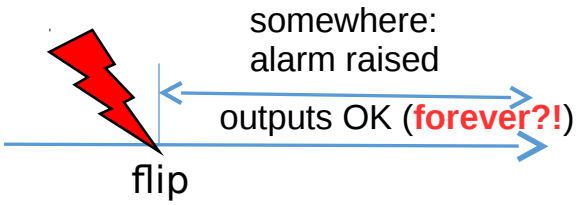
  (and does not affect any outputs)

  b) … or an alarm is raised within at most **k** time steps

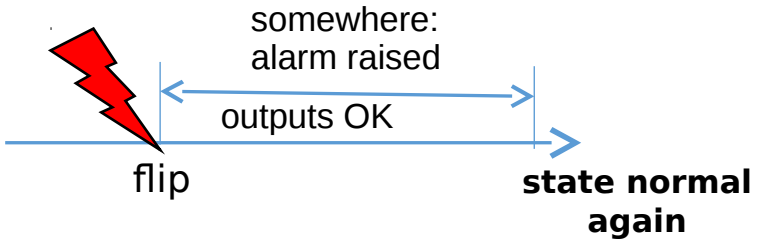
  (and does not affect any outputs before)

False Positives

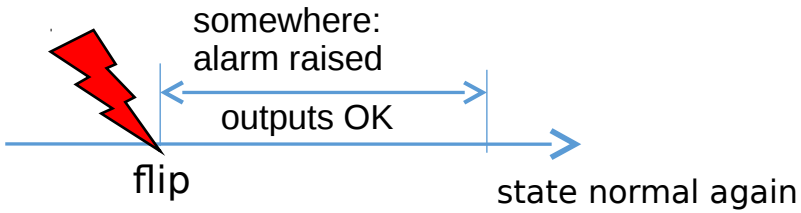
- Problem: impossible to compare outputs **forever**
  - undecidable



- Solution: search for situations where the state is repaired within finite time
  - Incomplete: might not find all false positives



Detecting **false-positives**



- Formal method: (Bounded) Model Checking
- Simulation
- **Semi-formal methods**

SAT query to find false positives

- Find situations where
  - bit flip raises an alarm, does not corrupt outputs, and state recovers from bitflip

```
for all time steps i=1 to len(test-case):
    “Dear SAT solver, can you find a false positive by flipping
      any latch in any previous time step?”
```

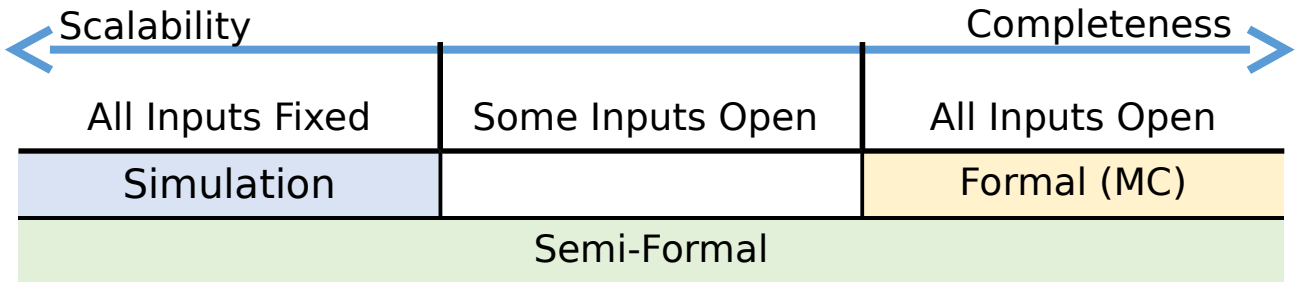
- If SATISFIABLE: false positive found

assignment: ¬f0, **f1**, ¬f2, .. , ¬fn, ¬c1, ¬c2, **c3**, .., ¬cm

  - c3: Latch 3 has to be flipped
  - f1: it has to be flipped in time step 1
  - state ok again in step i (at query time)

Summary

- Properties to check
  - Search for **vulnerable** components
  - Prove that component is **definitely protected**
  - Find situations for **false positives**
- Techniques for checking:

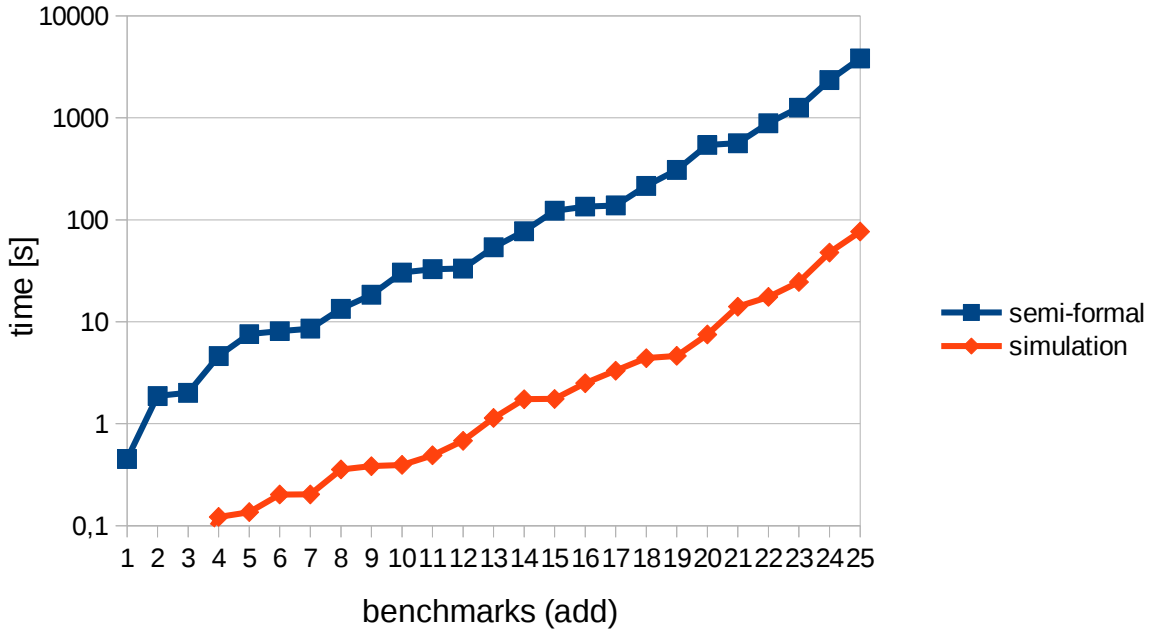


Results – simulation vs semi-formal (1/2)

- Simulation vs semi-formal (**point in time** and **component** to flip are symbolic variables)
- 3 **concrete** test cases, each 15 time steps long

```
011010001110
101011010111
...
```

- Search for **vulnerable** latches

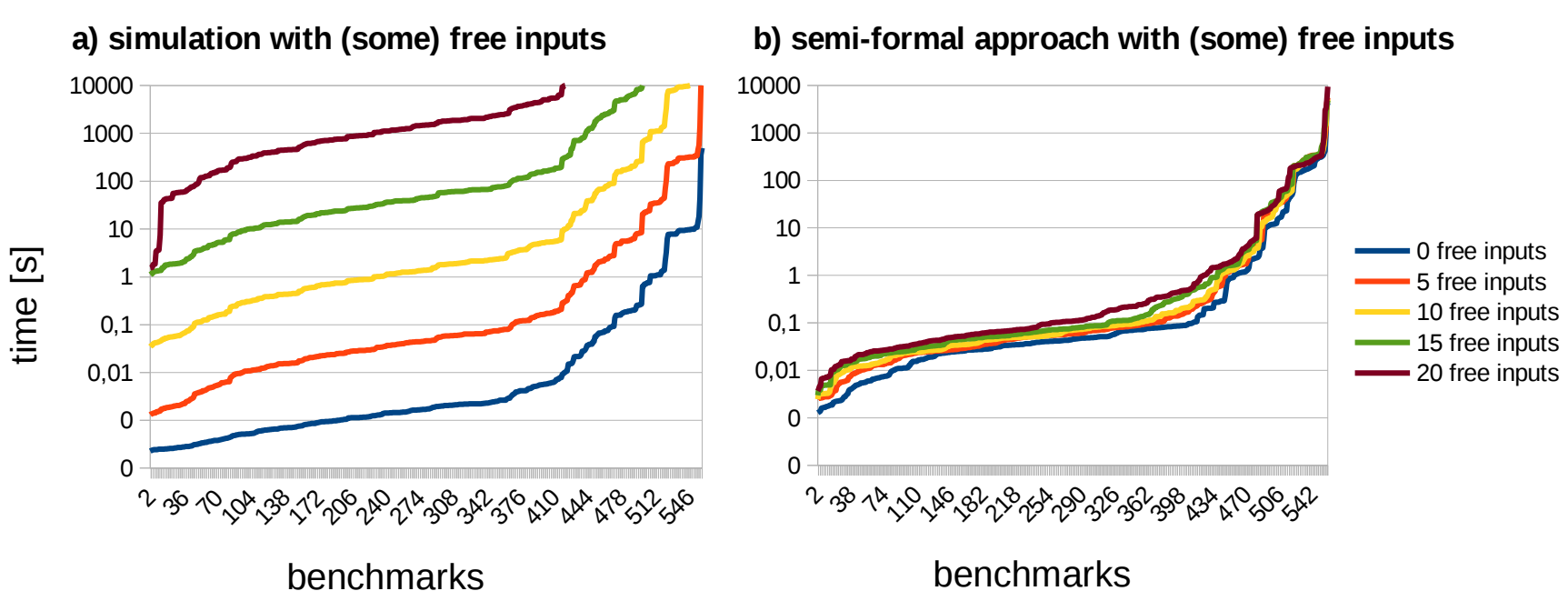


Results – simulation vs semi-formal (2/2)

- 3 test cases, each 15 time steps long, containing some **open** input values

```
011??01000110
10101??110111
...
```

- Search for **vulnerable** latches
- **>5** open inputs: **semi-formal faster**, **< 5**: simulation faster, **~5**: about the same



Conclusion

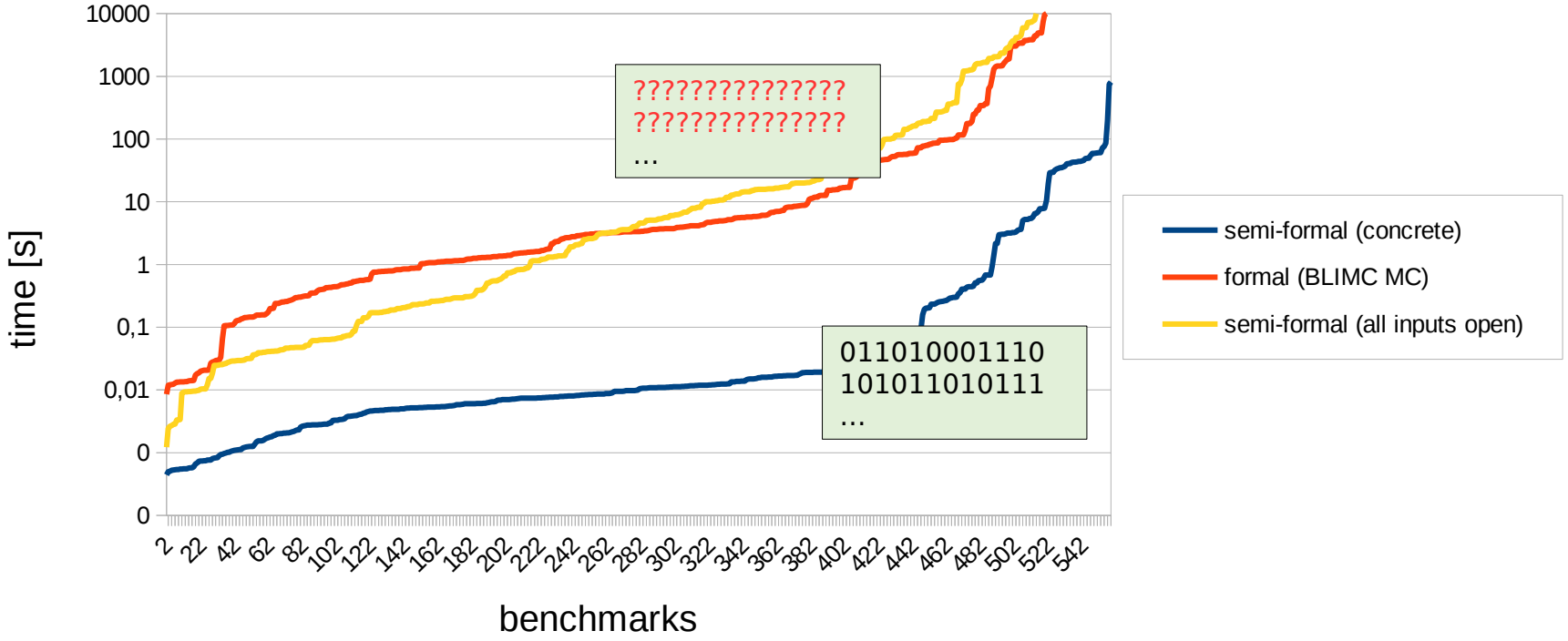
- Open inputs: Semi-formal algorithms scale significantly better than simulation
- Concrete Inputs: Simulation is fastest
- Reducing input space: better than MC

```
011??01000110
10101??110111
...
```

Thank you for your attention!
Questions?

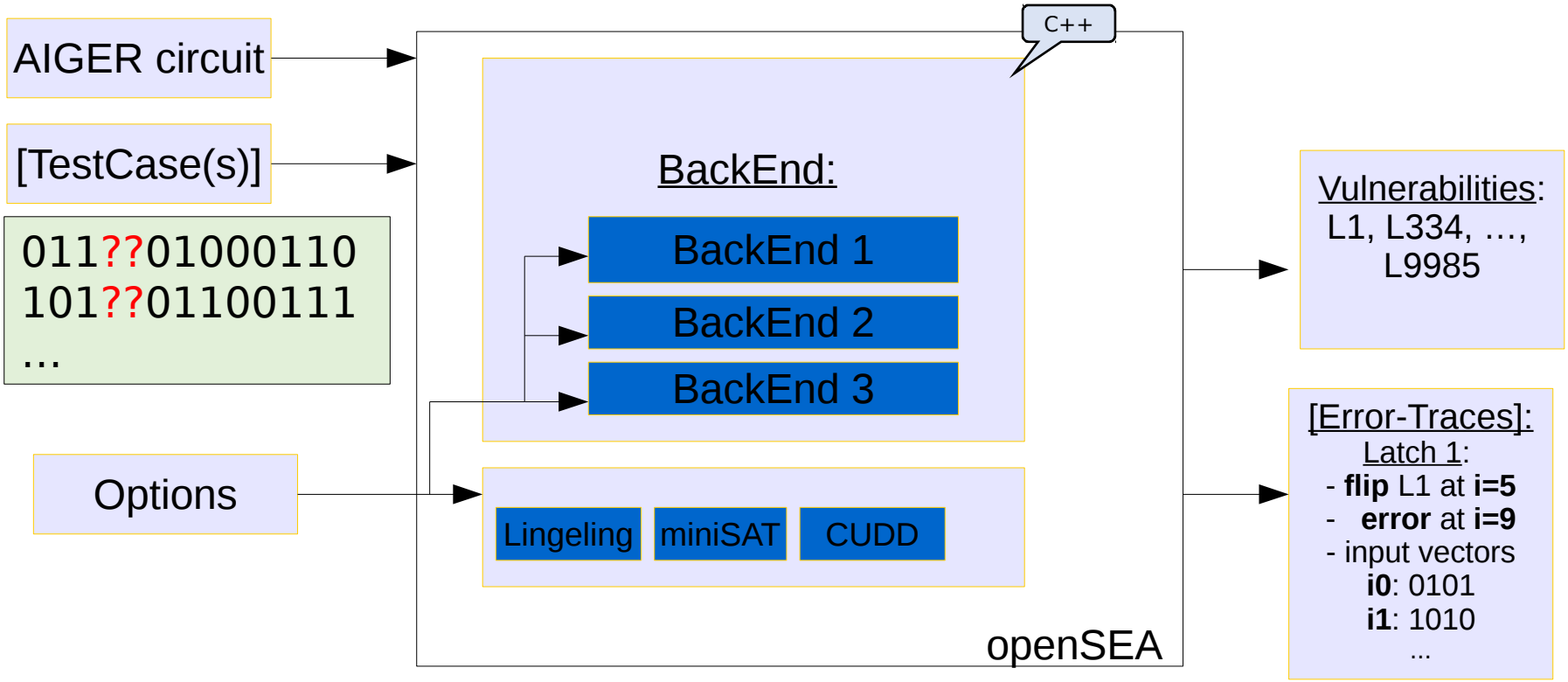
formal vs semi-formal method

- **All inputs open** (MC and semi-formal) vs. **all inputs concrete** (semi-formal)
- Search for **vulnerable** latches
- Semi-formal method and BMC about the same performance with all inputs open
- Semi-formal can be faster by setting (some) inputs to concrete values



OpenSEA

- Input: arbitrary circuit with protection logic (alarm output)
- Output: List of definitely vulnerable latches



Simulation (3/3)

Simulation-based solution: Details

```
// Step 1: fault-free simulation
for all test cases t in T:
  s := initial state
  for all time steps i=1 to len(t):
    correctState[t][i] := s
    s,o := sim1step(s,t[i])
    correctOutput[t][i] := o
```

```
// Step 2: simulation with faults:
for all latches l in L:
  for all test cases t in T:
    for all time steps i=1 to len(t):
      s := correctState[i]
      s := s with L flipped
      for all time steps j=i to len(t):
        if(s == correctState[t][j])
          continue with next step i
        s,o,c := simu1step(s,t[i])
        if(alarm)
          continue with next step i
        if(o != correctOutput[t][j])
          print “Latch L is vulnerable”
          continue with next latch l
```

Semi-Formal approach

- SAT-based: symbolically encode the circuit as a formula
- test case-based: Input values from test cases, can be concrete or open

```
011??01000110
101??01100111
...
```

- 2 variants:
  - A) point in time to flip symbolic:

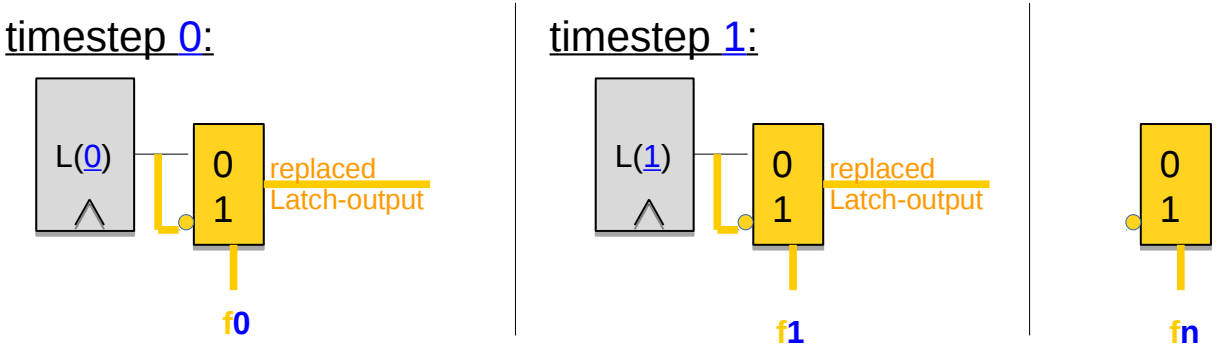
```
for all test cases t in T:
  for all latches l in L:
    for all time steps i=1 to len(t):
      // check ...
```

  - B) point in time AND component to flip symbolic

```
for all test cases t in T:
  for all latches l in L:
    for all time steps i=1 to len(t):
      // check ...
```

A) point in time to flip symbolic:

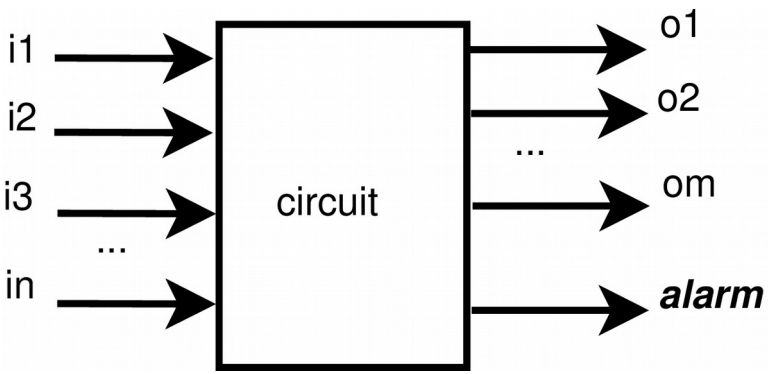
```
for all test cases t in T:
  for all latches l in L:
    for all time steps i=1 to len(t):
      // check ...
```

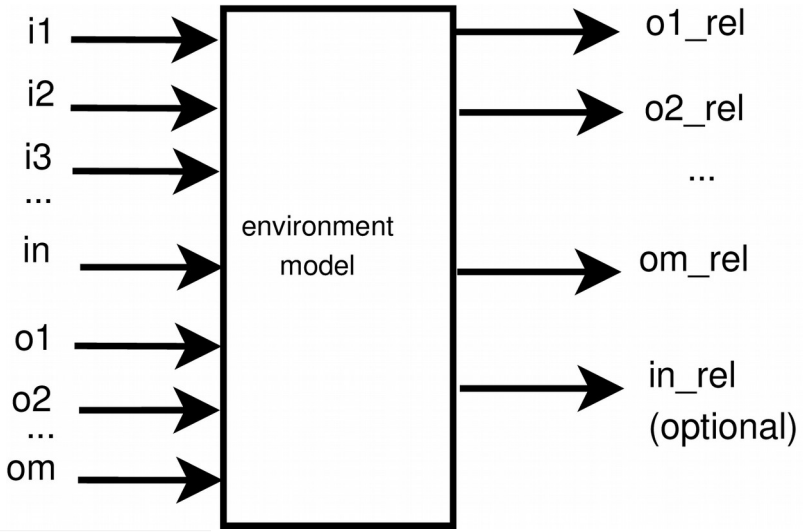


- If SATISFIABLE: assignment: ¬f0, **f1**, ¬f2, .. , ¬fn
  - Flip L in step 1 to produce corrupt outputs

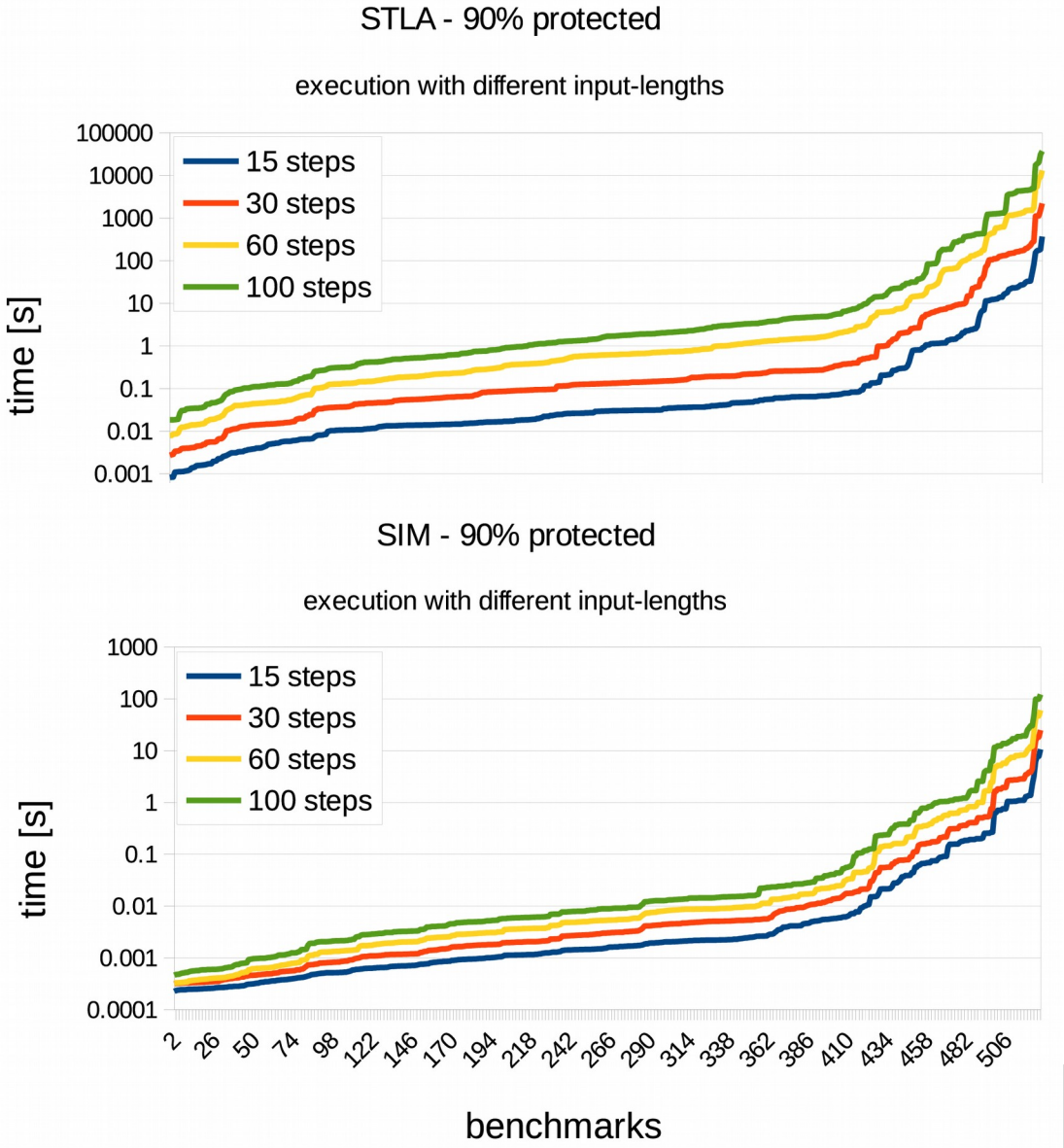
Environment Models

- Output values might be irrelevant
  - e.g. if data on bus is not ready
- Some input combinations might not be allowed
  - SAT-solver choices for input values can be restricted



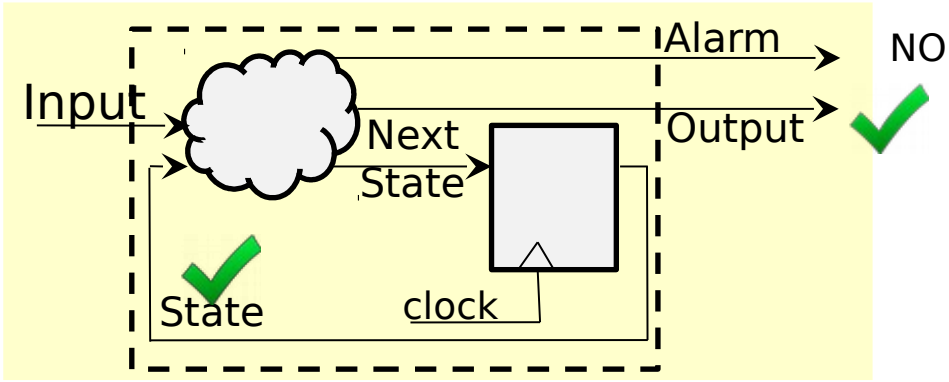


Results – Length of Test Cases



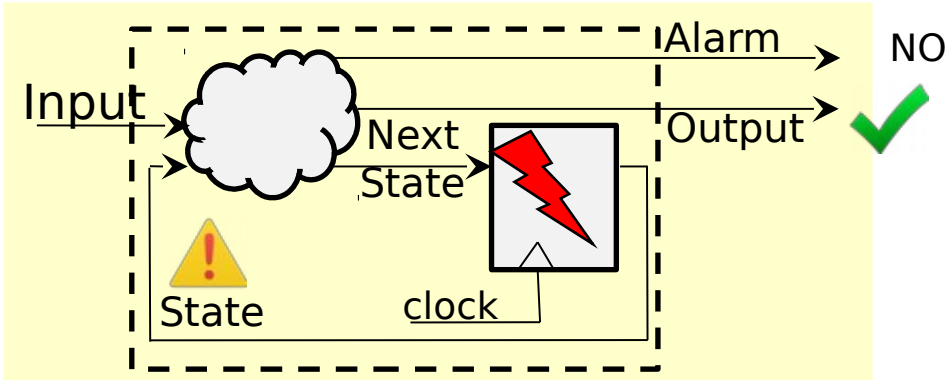
Vulnerable Latches:

- Given: circuit

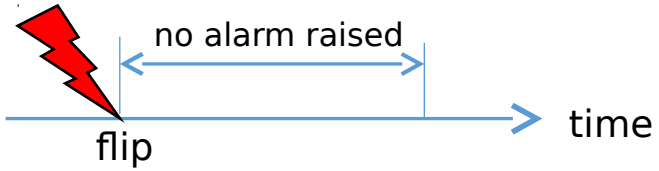


Vulnerable Latches:

- Given: circuit

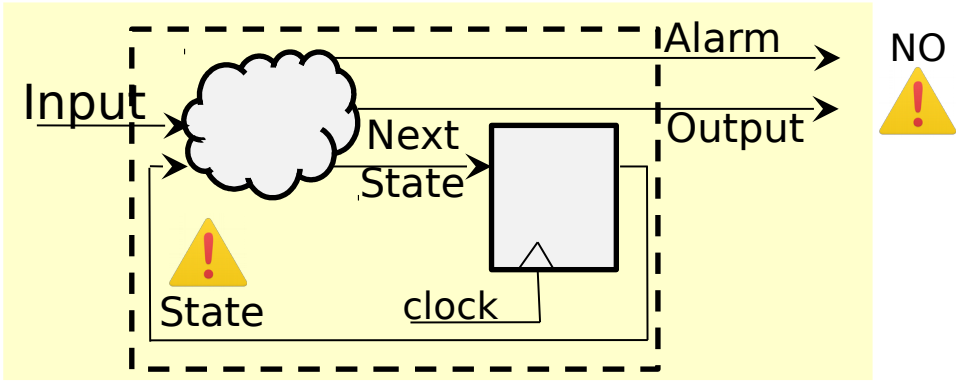


- Find situation..
  - where Latch can be flipped ...



Vulnerable Latches:

- Given: circuit



- Find situation..
  - where Latch can be flipped ...
  - such that outputs are wrong
  - without raising an alarm before