Logic Synthesis and Verification

Jie-Hong Roland Jiang
江介宏

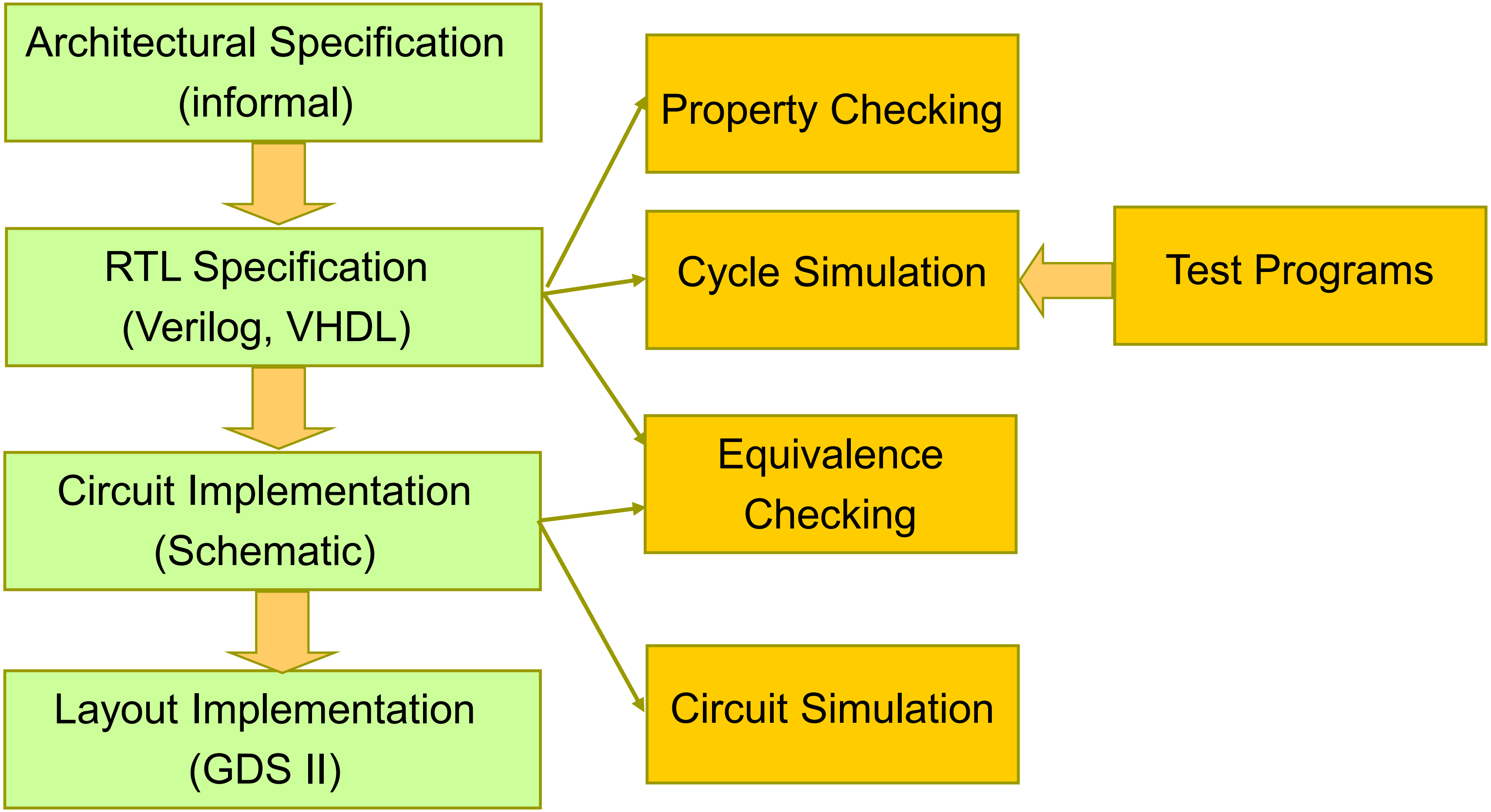
Department of Electrical Engineering
National Taiwan University

Fall 2024

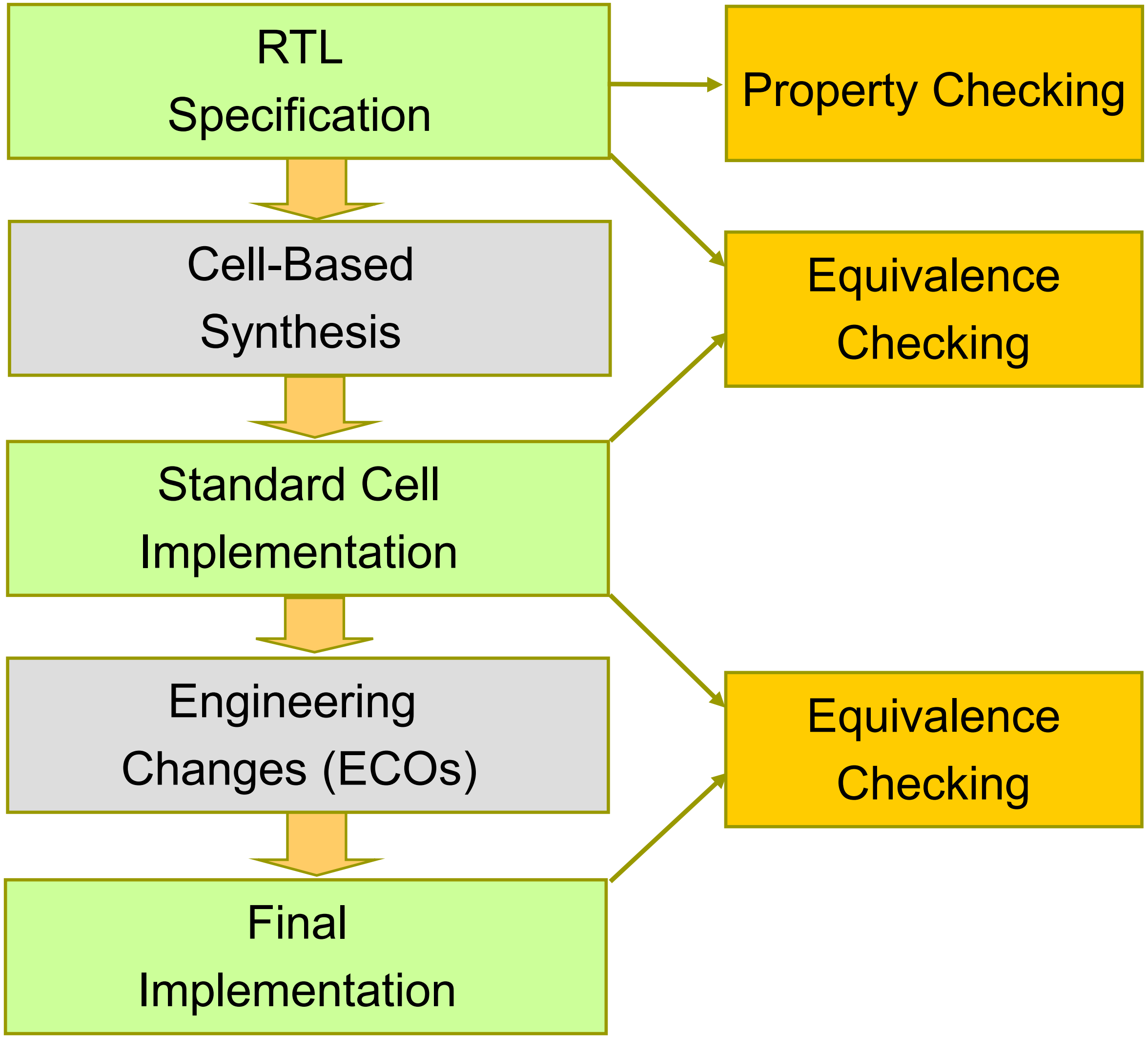
Equivalence and Property Checking

part of the following slides are by courtesy of Andreas Kuehlmann

Equivalence Checking in Microprocessor Design


Architectural Specification (informal)
RTL Specification (Verilog, VHDL)
Circuit Implementation (Schematic)
Layout Implementation (GDS II)
Property Checking
Cycle Simulation
Test Programs
Equivalence Checking
Circuit Simulation

Equivalence Checking in ASIC Design



RTL Specification

Cell-Based Synthesis

Standard Cell Implementation

Engineering Changes (ECOs)

Final Implementation

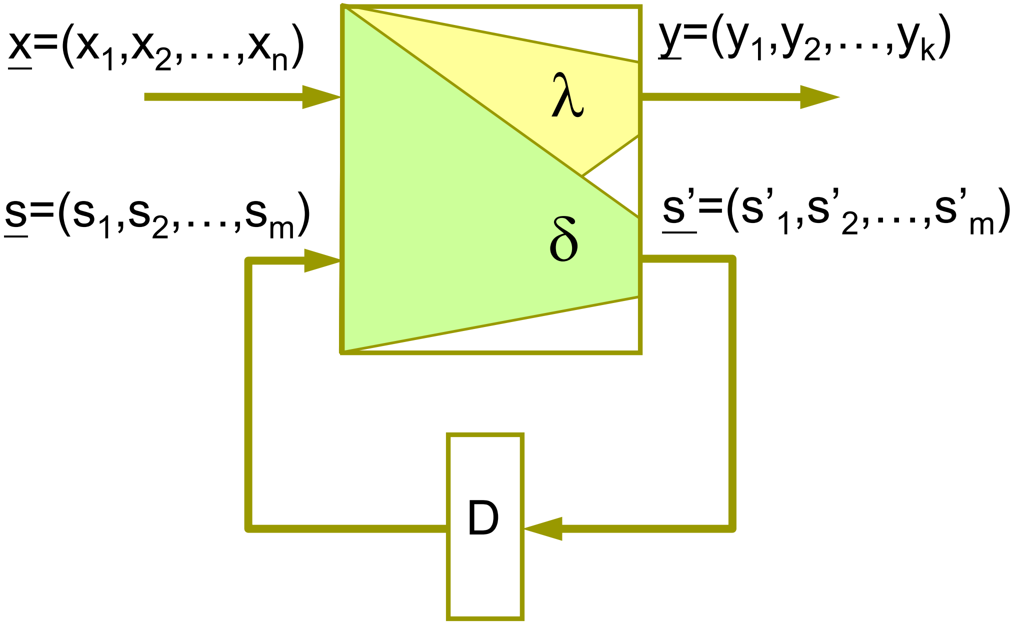
Property Checking

Equivalence Checking

Equivalence Checking

Finite State Machine Model

- $M(X,Y,S,S^0,\delta,\lambda)$:
  - X: Inputs
  - Y: Outputs
  - S: States
  - S^0: Initial State(s)
  - δ: $X \times S \rightarrow S$
    (next-state function)
  - λ: $X \times S \rightarrow Y$
    (output function)

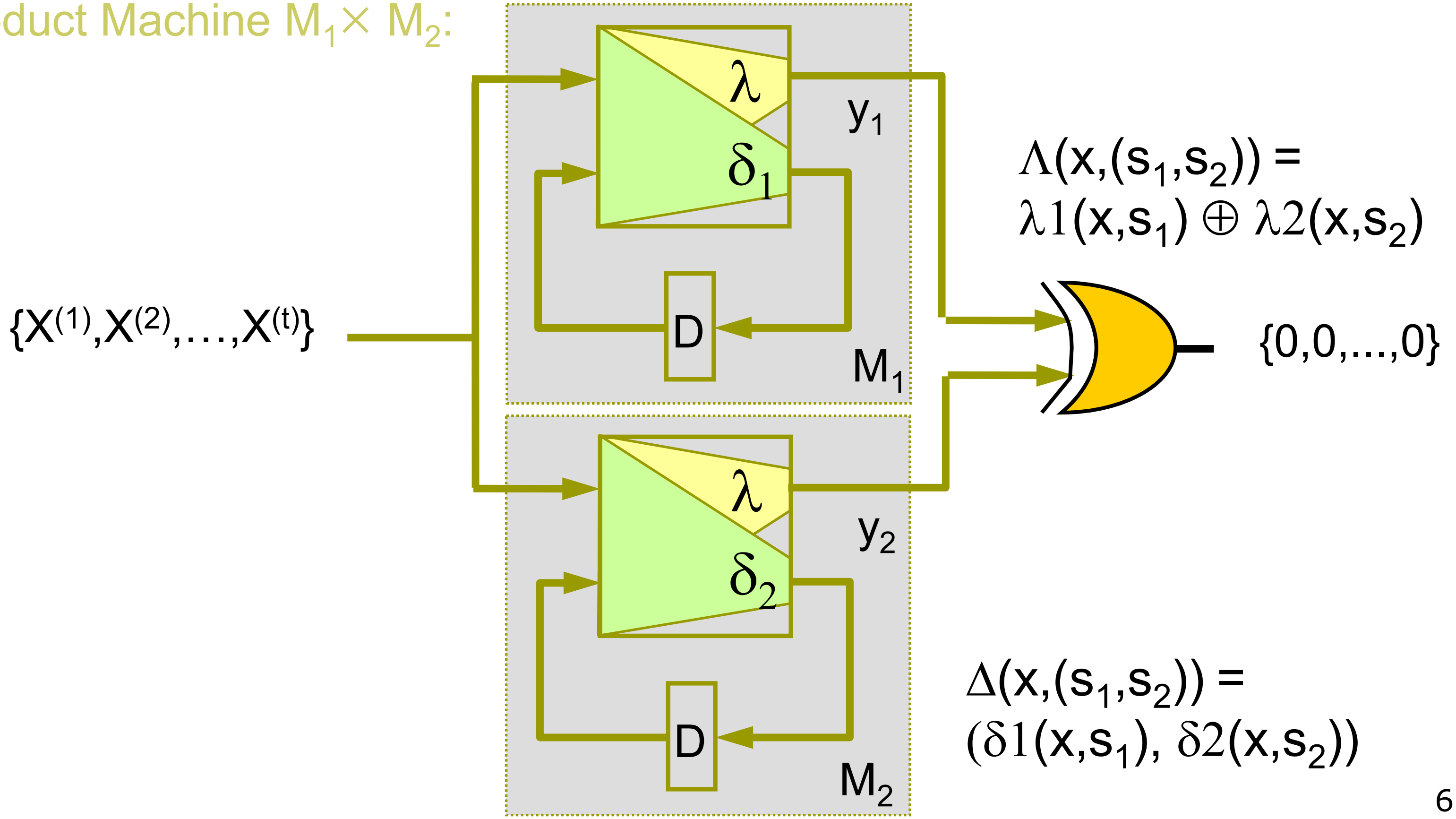

$\underline{x}=(x_1,x_2,\ldots,x_n)$
$\underline{y}=(y_1,y_2,\ldots,y_k)$
λ
$\underline{s}=(s_1,s_2,\ldots,s_m)$
δ
$\underline{s}'=(s'_1,s'_2,\ldots,s'_m)$
D

where a uppercase letter, e.g., X, denote the set of assignments to the variable set denoted in a lowercase letter, e.g., $\underline{x}$

Sequential Equivalence Checking

- Definition: Two FSMs M_1 and M_2 are functionally equivalent iff the product machine $M_1 \times M_2$ produces a constant 0 sequence for all valid input sequences $\{X^{(1)},\ldots,X^{(t)}\}$

Product Machine $M_1 \times M_2$:



$\Lambda(x,(s_1,s_2)) = \lambda1(x,s_1) \oplus \lambda2(x,s_2)$

$\Delta(x,(s_1,s_2)) = (\delta1(x,s_1), \delta2(x,s_2))$

General Approach to SEC

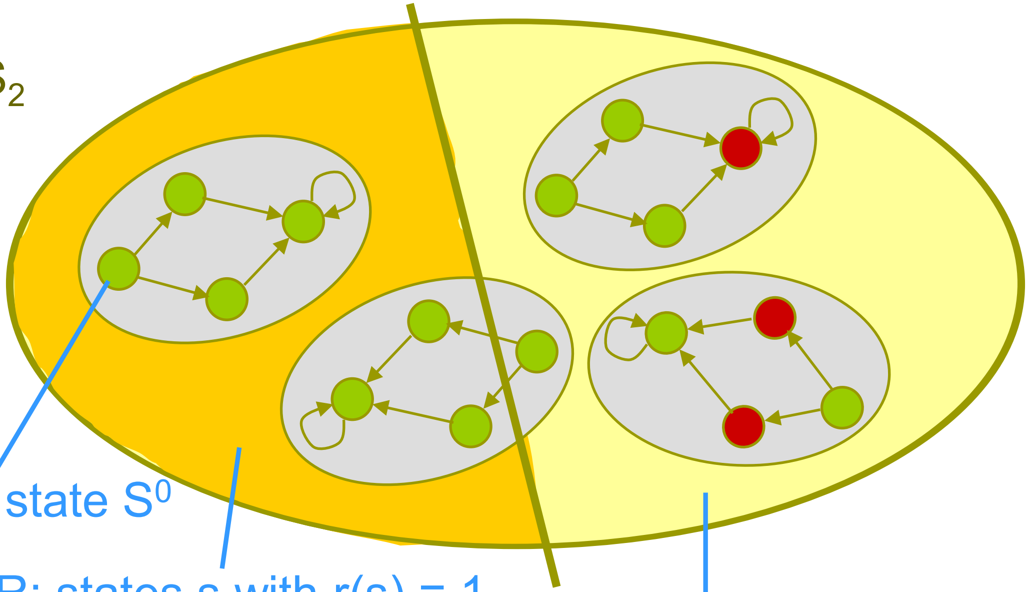
Product state space $S = S_1 \times S_2$

🔴 bad states

i.e. $\exists x.\Lambda(x,s) \neq 0$

🟢 good states

i.e. $\forall x.\Lambda(x,s) = 0$



initial state S^0

R: states s with r(s) = 1

R’= [S]\R: states s with r(s) = 0

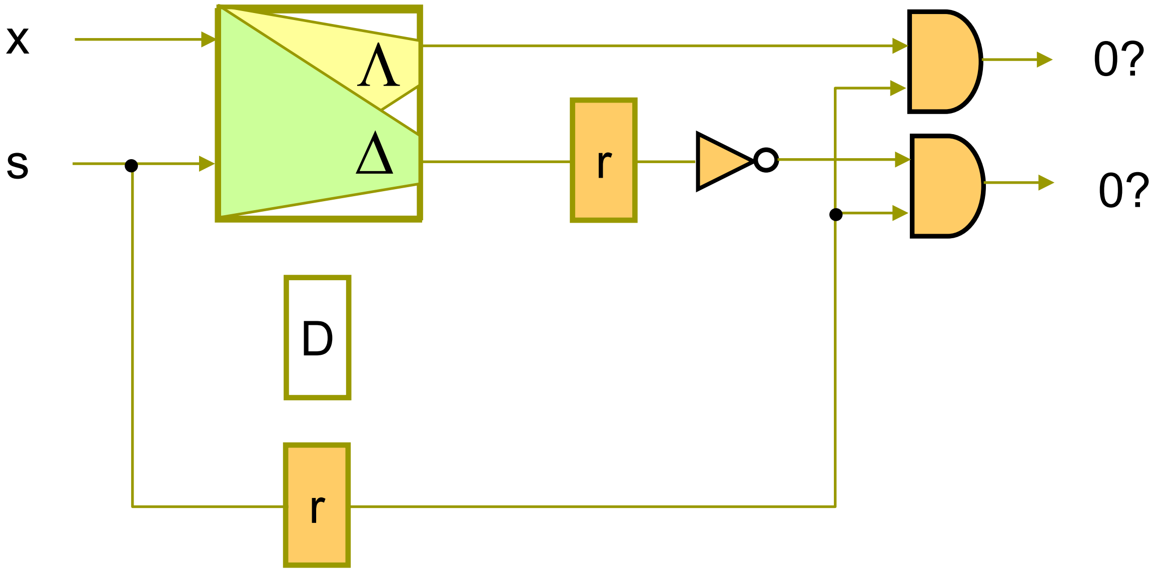
Inductive proof of equivalence:

Find subset $R \subseteq S$ with characteristic function $r: S \rightarrow \{0,1\}$ such that:

1. $r(s^0) = 1$ (initial state is in R)
2. $(r(s) = 1) \Rightarrow r(\Delta(x,s)) = 1$ (all R states cannot go to R’ states)
3. $(r(s) = 1) \Rightarrow \Lambda(x,s) = 0$ (all R states are good states)

Sequential Equivalence Checking

- Proving sequential equivalence under state set R
  1. Check (by SAT) that initial state S^0 is contained in R, i.e. $r(s^0) = 1$
  2. Check (by SAT) that
     - states in R are good states: $\forall x.\ r(s) \Rightarrow \neg\Lambda(x,s)$, i.e., $r(s) \wedge \Lambda(x,s)$ unsatisfiable
     - all states from R lead only to states in R: $\forall x.\ r(s) \Rightarrow r(\Delta(x,s))$, i.e., $r(s) \wedge \neg r(\Delta(x,s))$ unsatisfiable



Soundness and Completeness

- With a candidate state set R we can
  - prove equivalence
    - that means the method is “sound”
    - we will not produce “false positives”
  - but not disprove equivalence
    - that means the method is “incomplete”
    - we may produce “false negatives”

Inductive State Set Derivation

- Reachability analysis:
  - state traversal until no more states can be explored
    - forward vs. backward
    - explicit vs. implicit (symbolic)
- Relying on the design methodology to provide R:
  - equivalent state encoding in both machines
  - synthesis tool provides hint for R from sequential optimization
    - manual register correspondence
    - automatic register correspondence
- Combination of them

Combinational EC

- Industrial equivalence checkers almost exclusively use a combinational EC paradigm
  - sequential EC is too complex, can only be applied to design with a few hundred state bits
  - combinational methods scale linearly with the design size for a given fixed size and “functional complexity” of the individual cones

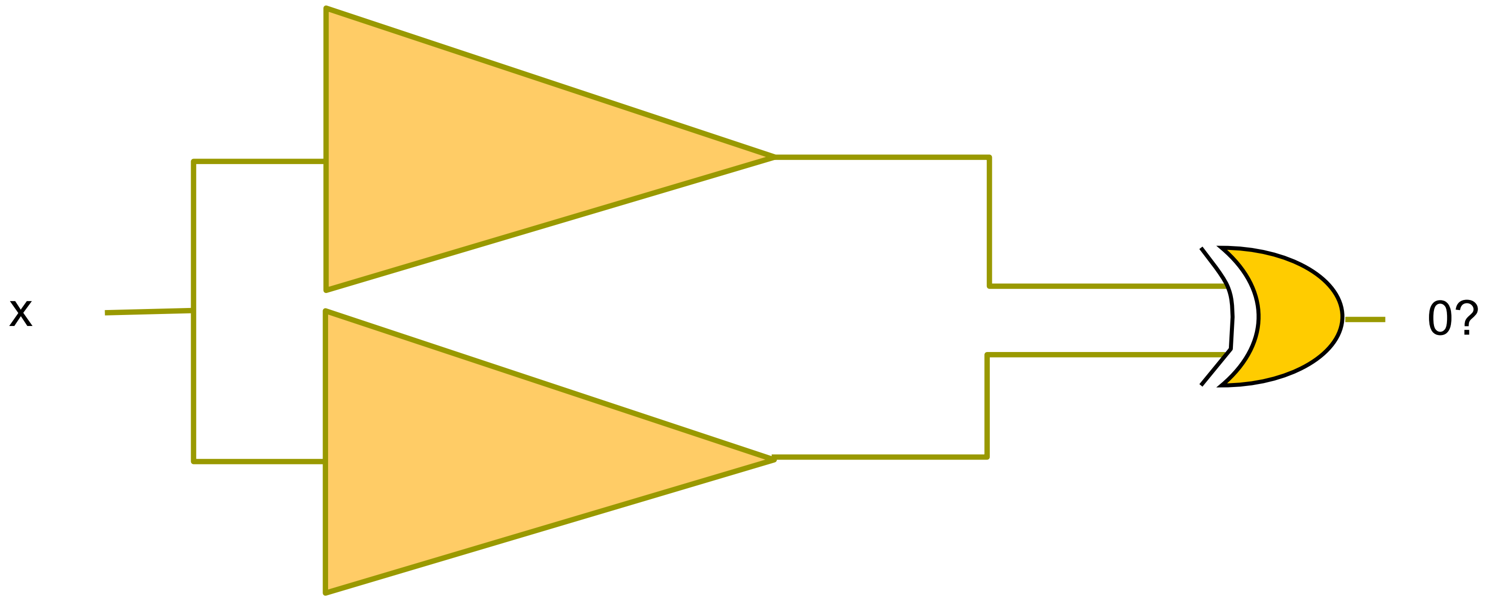
- Still, pure BDDs and plain SAT solver cannot handle all cones
  - BDDs can be built for about 80% of the cones of high-speed designs
  - less for complex ASICs
  - plain SAT blows up on a “miter” structure

- Contemporary method highly exploit structural similarity of designs to be compared

Combinational EC

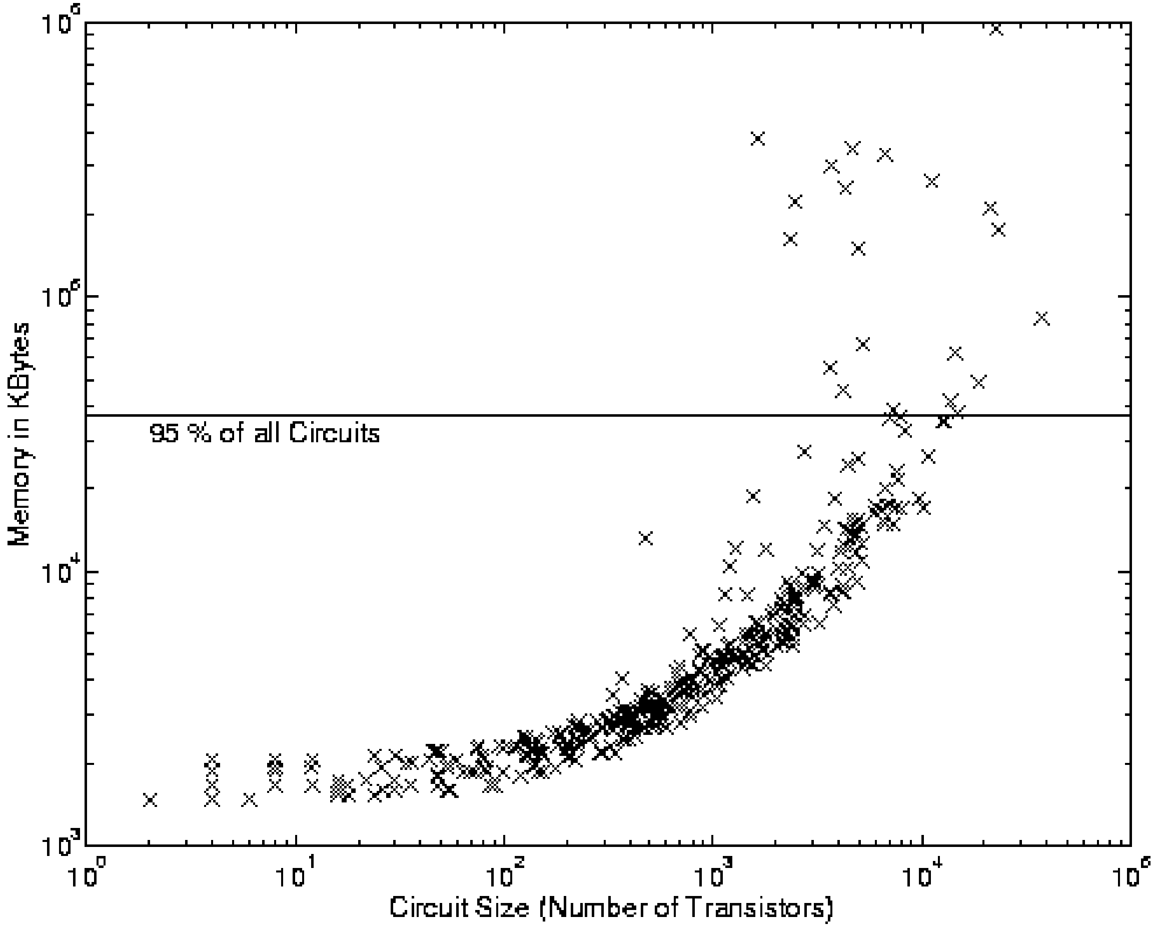
- Basic methods:
  - random simulation, good for finding mis-compares
  - BDD-based with modifications
  - structural SAT-based with modifications

Miter structure



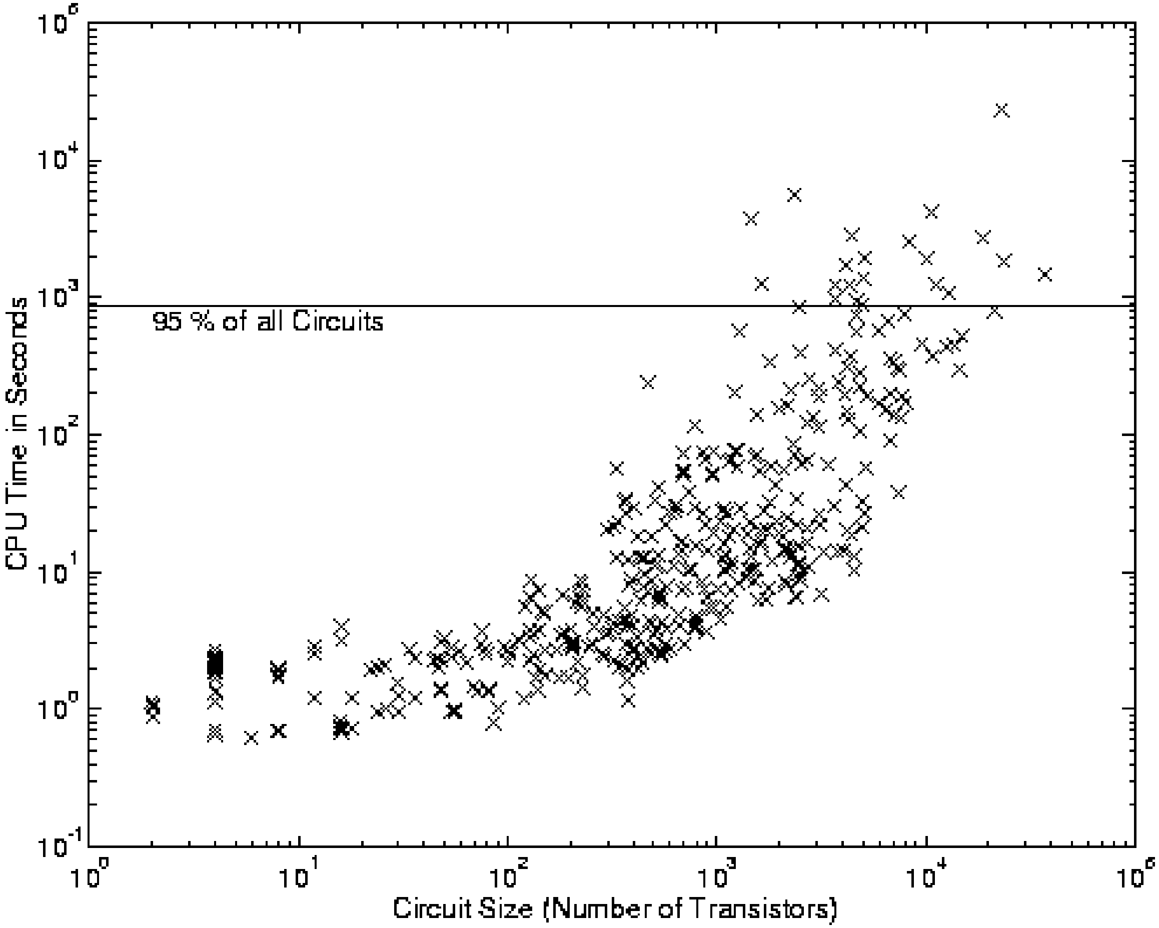
Combinational EC

- Memory statistics of BDD-based EC on a PowerPC processor design



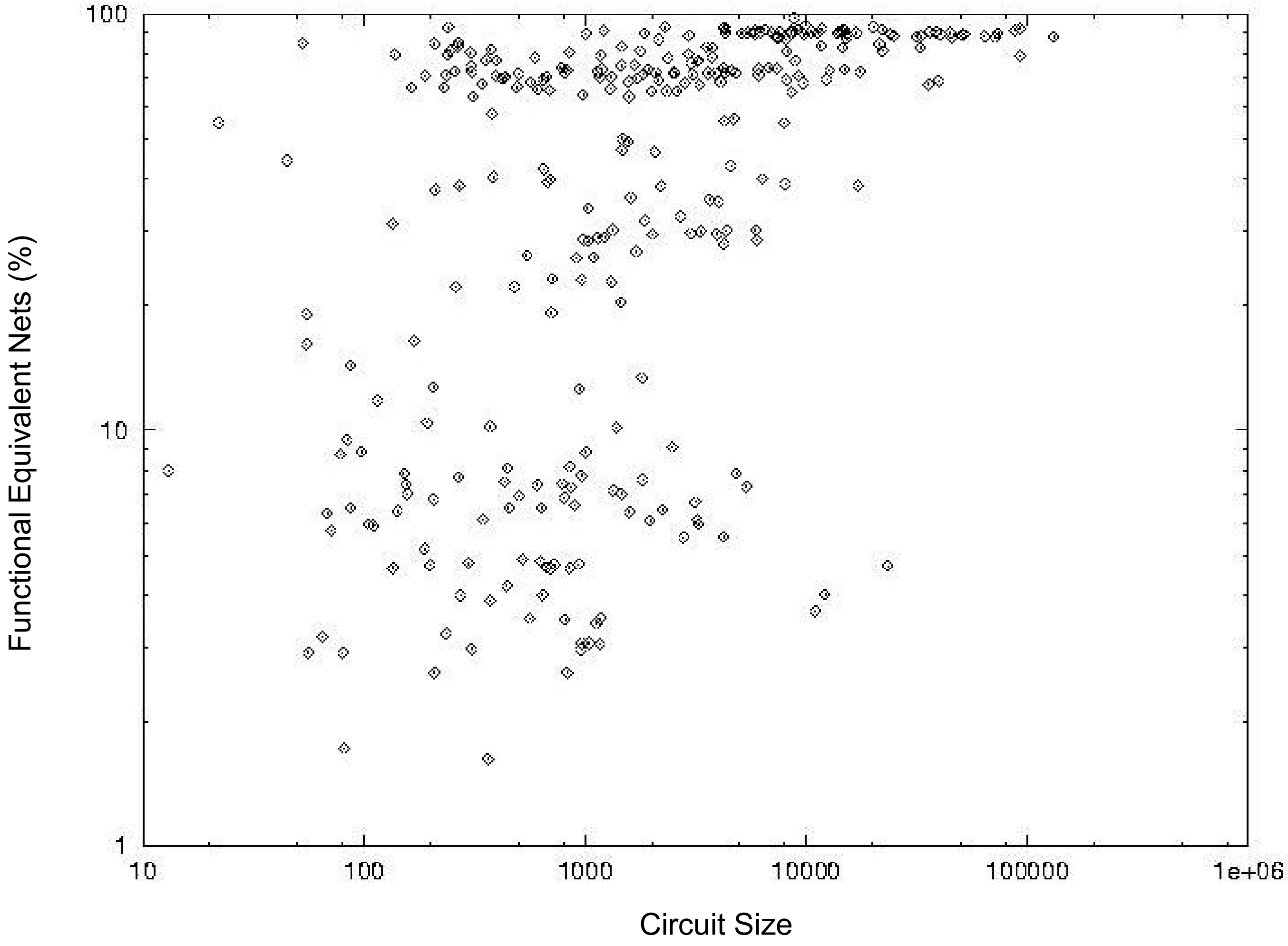
Combinational EC

- Runtime statistics of BDD-based EC on a PowerPC processor design



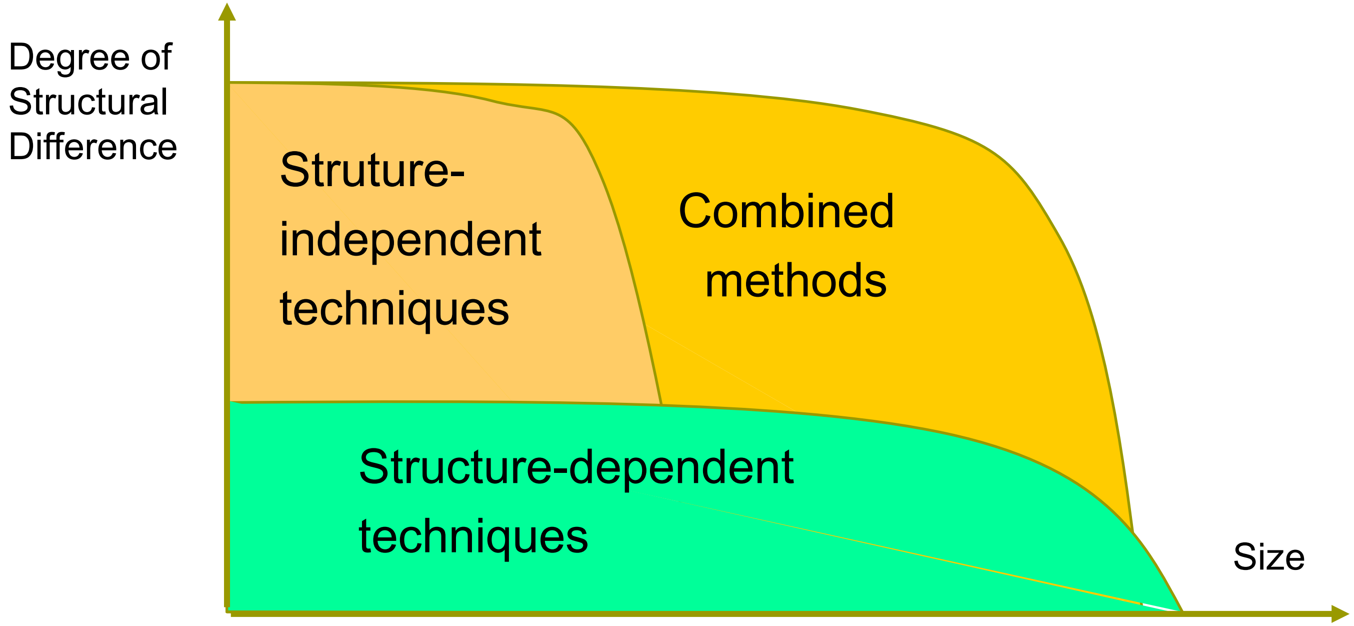
Combinational EC

- Evidence of vast existence of structure similarities



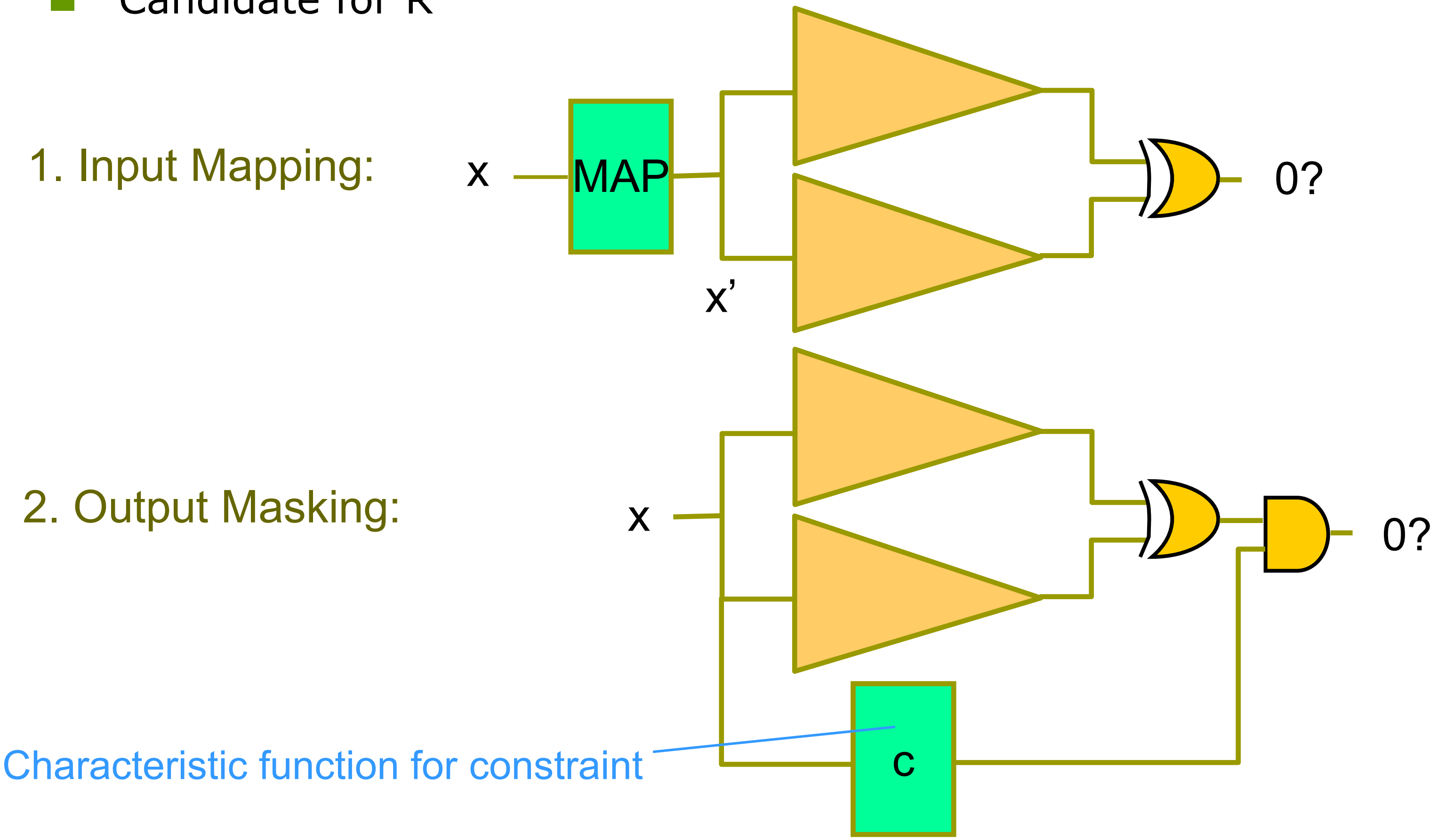
Structure and Verification

- Structure-independent techniques
  - Exhaustive simulation
  - Decision diagrams
- Structure-dependent techniques
  - Graph hashing
  - SAT based cutpoint identification


Degree of Structural Difference
Struture-independent techniques
Combined methods
Structure-dependent techniques
Size

Constrained EC

- Input constraints:
  - Non-occurring input values (don’t cares)
  - Unreachable states
  - Candidate for R



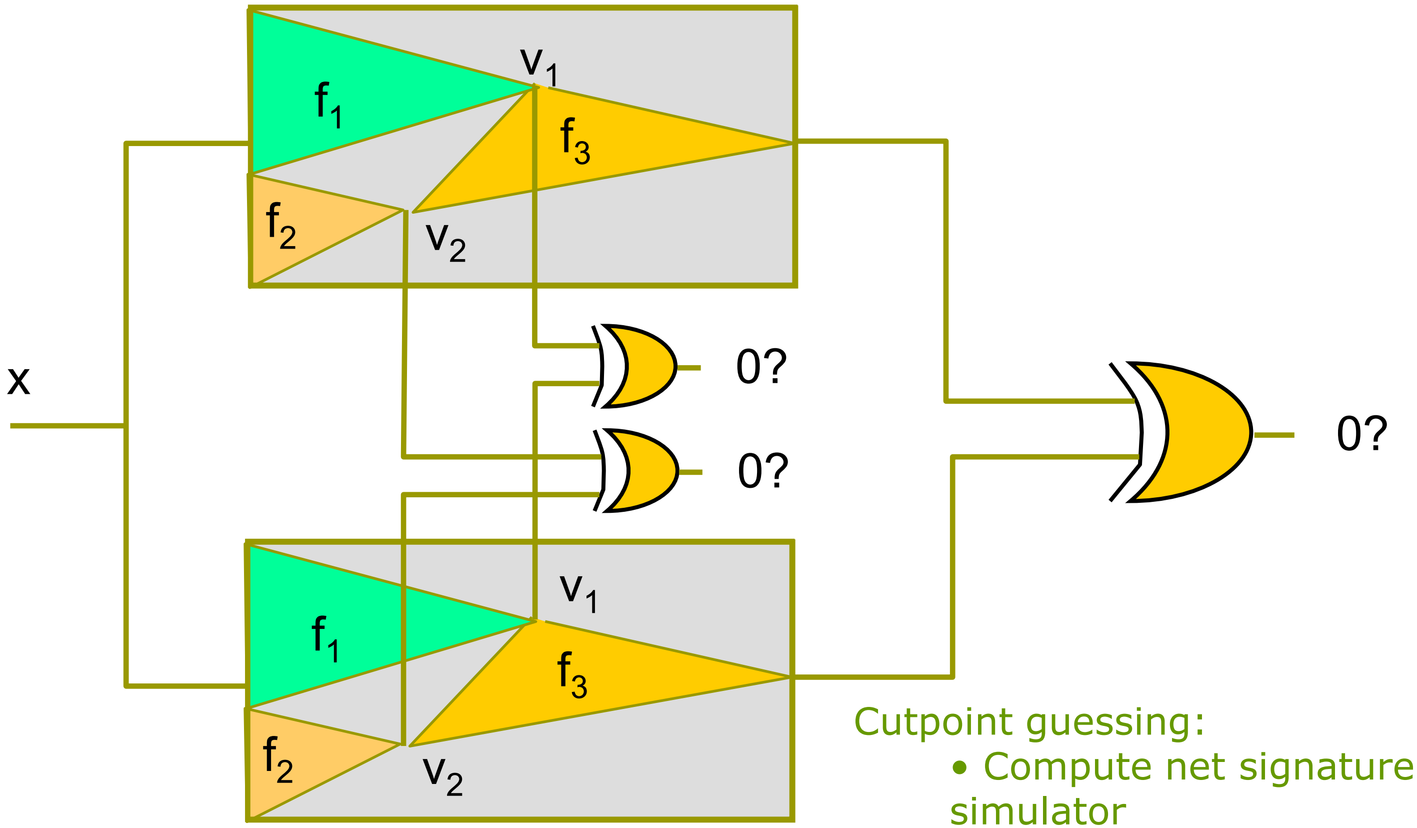
1. Input Mapping:

2. Output Masking:

Characteristic function for constraint

Cutpoint-Based EC

- Cutpoints are used to partition the miter

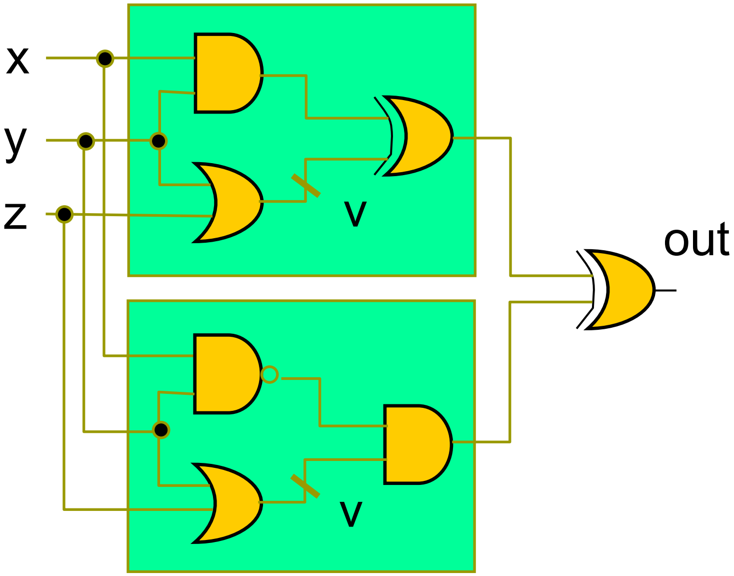


Cutpoint guessing:

- Compute net signature with random simulator
- Sort signatures + select cutpoints
- Verify and refine cutpoints iteratively
- Verify outputs

Cutpoint-Based EC

- False negatives
  - Outputs may miscompare for invalid cutpoint values



vz xy	00	10	11	01
00			1	
01			1	
11	1	1		1
10	1	1		1

vz xy	00	10	11	01
00				
01				
11	1	1		1
10	1	1		1

Constraint:

c = (v ≡ y+z)

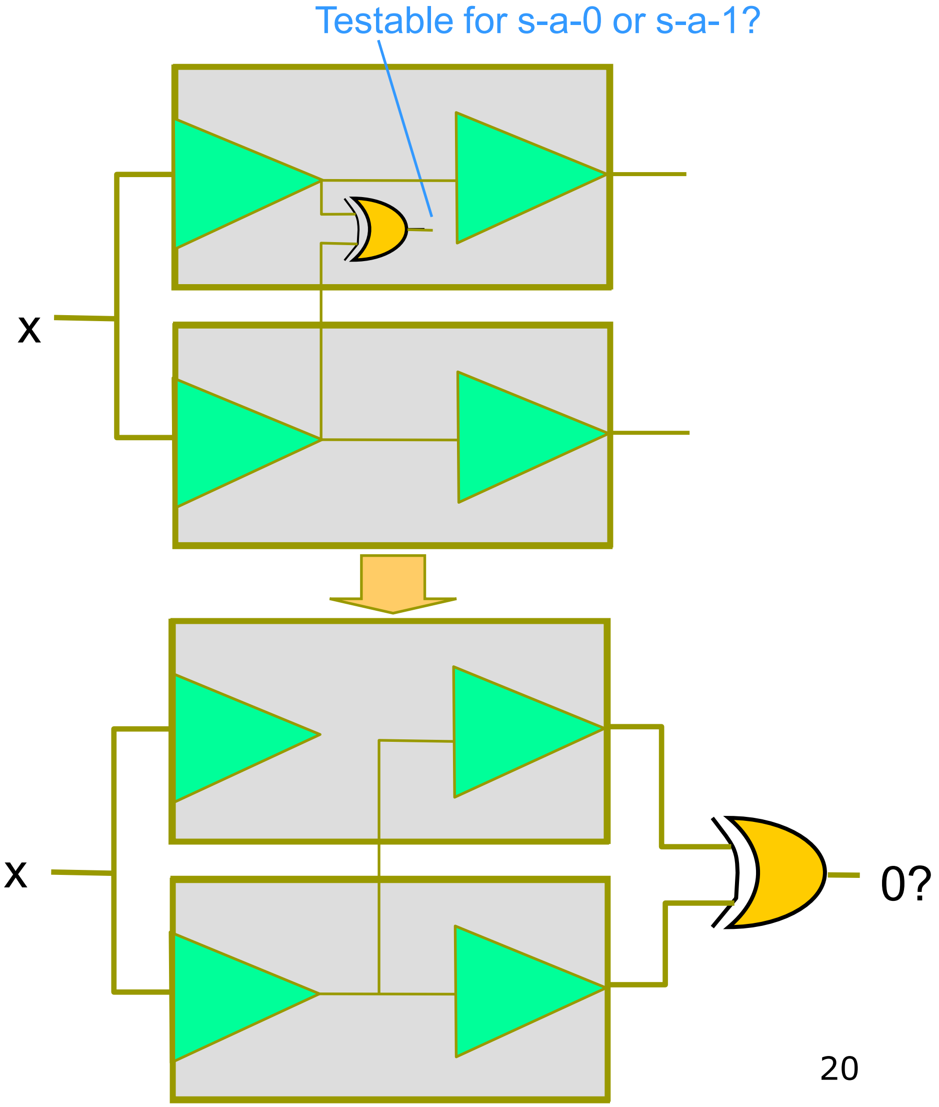
vz xy	00	10	11	01
00	1	1		
01				
11	1	1	1	1
10			1	1

What can we do about false negatives:

- constrain input space to c = (v ≡ y+z)
- if v in SUPPORT(out), then out = compose(out, v, f_v)

Cutpoint-Based EC

- Permissible cutpoints
  - Apply ATPG:
    - test for s-a-0 at output checks for permissible functions
    - test for s-a-1 at output checks for inverse permissible functions
  - Merge permissible cutpoints successively from inputs to outputs



Sequential EC

- If combinational verification paradigm fails (e.g. we have no name matching)
- Two options:
  - Run general sequential verification based on state traversal
    - Very expensive but most general
  - Try to match registers automatically
    - Structural register correspondence
    - Functional register correspondence

Register Correspondence

- Find registers in product machine that implement identical or complemented function
  - These are matching registers in the two FSMs under comparison
  - BUT: might be more, we may have redundant registers

- Definition: A register correspondence $RC \subseteq \underline{s} \times \underline{s}$ is an equivalence relation in the set of registers $\underline{s}$
  - Can be extended to also include complemented functions
  - A register correspondence can be used as a candidate for R:

$$r(s) = \prod_{\forall (s^i, s^j) \in RC} (s^i \equiv s^j) \qquad RC \subseteq \underline{s} \times \underline{s}$$

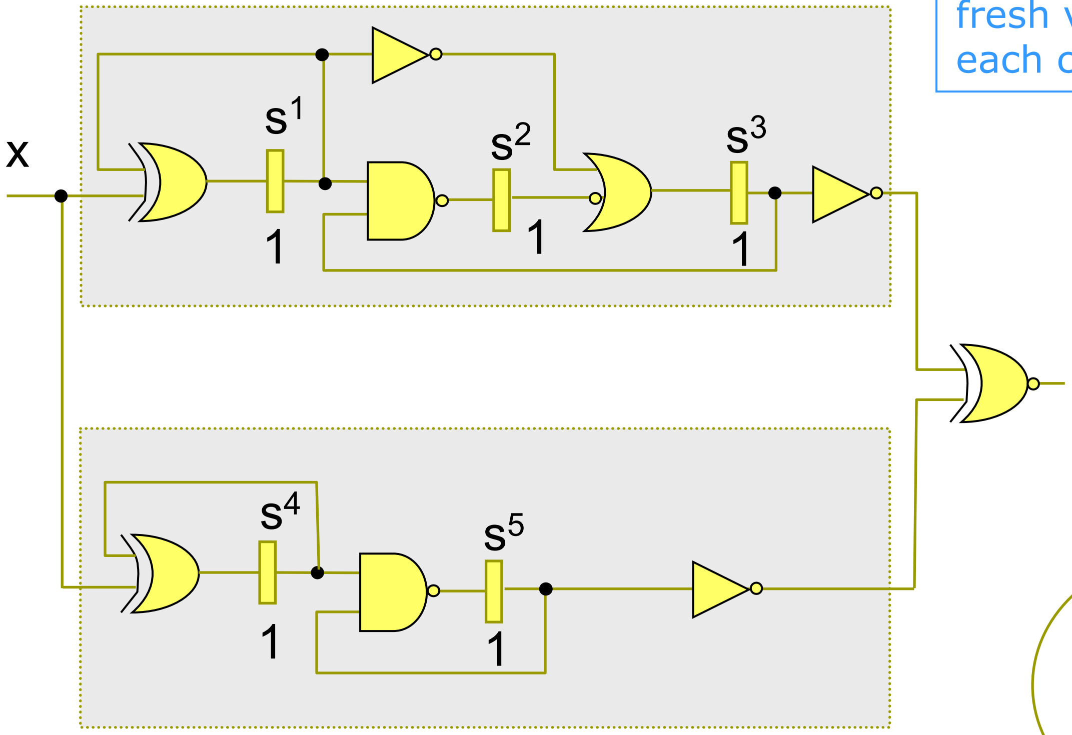
Register Correspondence

```
Algorithm REGISTER_CORRESPONDENCE {
  RC' = $\{(s^i,s^j) \mid s^i_0 = s^j_0\}$
   //start with registers with identical initial values
  do {
    RC    = RC'
    r(s)  = $\Pi_{\forall(si,sj)\in RC}\ (s^i \equiv s^j)$
    RC'   = $\{(s^i,s^j) \mid (s^i,s^j)\in RC \wedge (r(s) \rightarrow \delta^i(x,s)=\delta^j(x,s))\}$
    //$\delta^i$ is the transition function of $s^i$
  } while (RC' != RC)
  return RC
}
```

- In essence
  - The algorithm starts with an initial partitioning with two equivalence classes, one for each initial value
  - The algorithm computes iteratively the next-state function, assuming that the RC is correct
    - if yes, fixed point is reached and RC returned
    - if no, split equivalence classes along the mis-compares

Register Correspondence

- Example

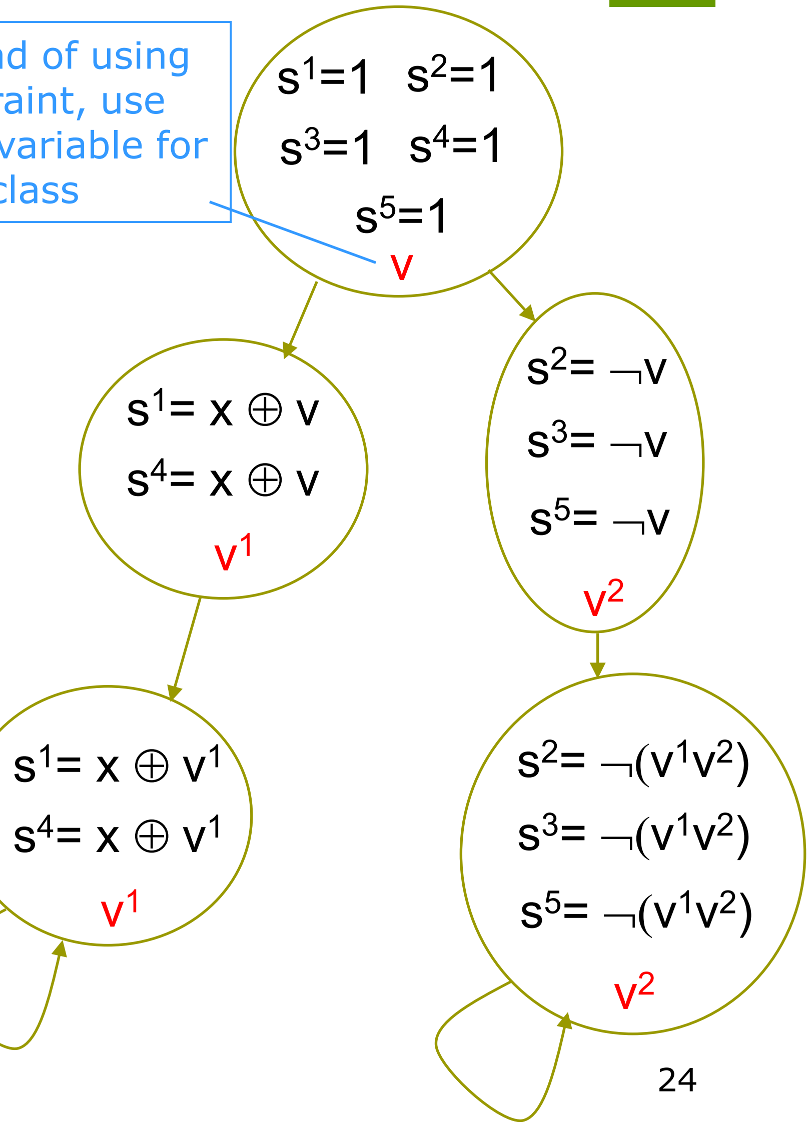


Result:

$\{s^1,s^4\}$

$\{s^2,s^3,s^5\}$

Instead of using constraint, use fresh variable for each class



$s^1=1$ $s^2=1$

$s^3=1$ $s^4=1$

$s^5=1$

v

$s^1= x \oplus v$

$s^4= x \oplus v$

v^1

$s^2= \neg v$

$s^3= \neg v$

$s^5= \neg v$

v^2

$s^1= x \oplus v^1$

$s^4= x \oplus v^1$

v^1

$s^2= \neg(v^1v^2)$

$s^3= \neg(v^1v^2)$

$s^5= \neg(v^1v^2)$

v^2

Register Correspondence

- Potential problems:
  - In case of mis-comparing designs
    - Effect of mis-compared cone may ripple through entire algorithm and split all equivalence classes until they contain only single registers
    - Difficult to debug since no hint of error location
    - Solution:
      - Relax equivalence criteria
        - E.g. structural register correspondence algorithm based on support set of registers
      - Combine with name mapping, functional/structural criteria

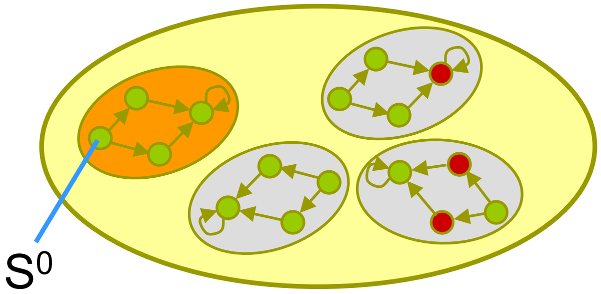
Sequential EC

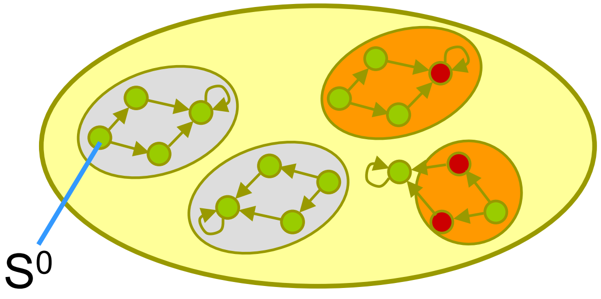
- In case that combinational EC model fails:
  - Use generalized register correspondence to also consider retiming
    - In essence, use all internal nets as candidates for possible matches

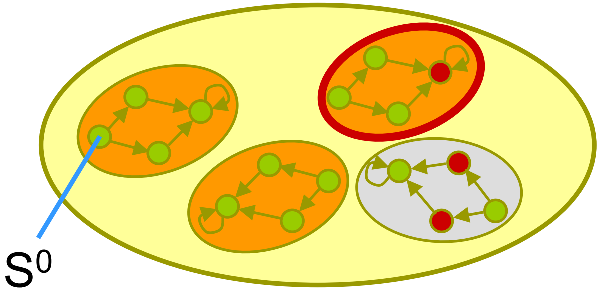
- Worst case: general sequential verification
  - Prove that the output of the product machine is not satisfiable (sequentially)
  - Special case of general property checking

Sequential EC

- State traversal
  - Forward
    - Start from initial state(s)
    - Traverse forward to check whether "bad" state(s) is reachable
  - Backward
    - Start from bad state(s)
    - Traverse backward to check whether initial state(s) can reach them
  - Hybrid
    - Compute over-approximation of reachable states by forward traversal
    - For all bad states in over-approximation, start backward traversal to see whether initial state can reach them







Sequential EC

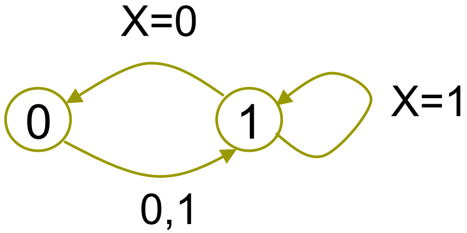
- Transition relation

Transition Relation t(s,s'): $t(s,s') = \begin{cases} 1 & \text{if there is a transition from s to s'} \\ 0 & \text{otherwise} \end{cases}$

$$t(s,s') = \exists x.(s' \equiv \delta(x,s))$$

- Example

x	s	$\delta(x,s)$	s'	$s' \equiv \delta(x,s)$	$t(s,s') = \exists x.(s' \equiv \delta(x,s))$
0	0	1	0	0	0
1	0	1	0	0	0
0	1	0	0	1	1
1	1	1	0	0	1
0	0	1	1	1	1
1	0	1	1	1	1
0	1	0	1	0	1
1	1	1	1	1	1



$\exists x.$

Sequential EC

- Image and pre-image of states

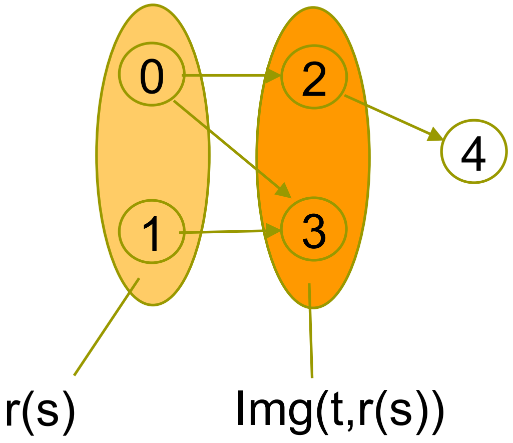
Image of a set of states r(s):

$$IMG(t,r) = \exists s.(r(s) \wedge t(s,s'))$$

Pre-Image of a set of states r(s):

$$PREIMG(t,r) = \exists s'.(r(s') \wedge t(s,s'))$$

- Example



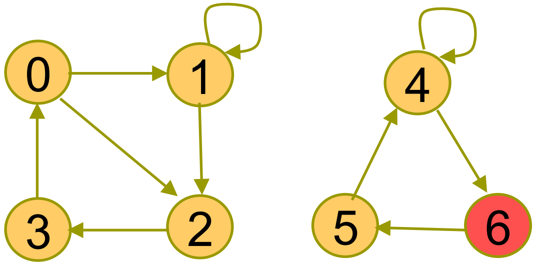
$r(s)$	$= (s \equiv 0) \vee (s \equiv 1)$	$\{0,1\}$
$t(s,s')$	$= (s \equiv 0) \wedge (s' \equiv 2) \vee$ $(s \equiv 0) \wedge (s' \equiv 3) \vee$ $(s \equiv 1) \wedge (s' \equiv 3) \vee$ $(s \equiv 2) \wedge (s' \equiv 4)$	$\{(0,2),$ $(0,3),$ $(1,3),$ $(2,4)\}$
$t \wedge r$	$= (s \equiv 0) \wedge (s' \equiv 2) \vee$ $(s \equiv 0) \wedge (s' \equiv 3) \vee$ $(s \equiv 1) \wedge (s' \equiv 3)$	$\{(0,2),$ $(0,3),$ $(1,3)\}$
$\exists s.(r \wedge t)$	$= (s' \equiv 2) \vee (s' \equiv 3)$	$\{2,3\}$

Sequential EC

- Forward state traversal

```
Algorithm TRAVERSE_FORWARD(t, λ ,S0) {
  reached = ∅
  current = S0                             // start from init
  while (reached ≠ (reached ∨ current)) { // fixed point
    reached = reached ∨ current            // add new states
    next    = IMG(t,current)               // one step transition
    current = next                         // rename variable
  }
  return ∃x.(λ(x,s) ∧ reached)
}
```

- Example



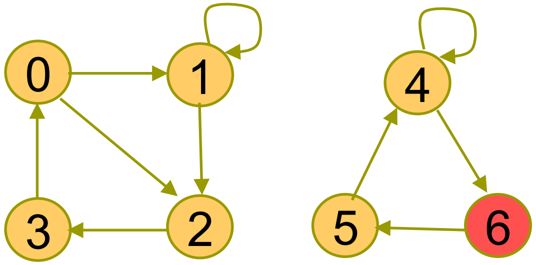
Iteration:	1	2	3
Reached:	{0}	{0,1,2}	{0,1,2,3}
Current:	{0}	{1,2}	{1,2,3}
Next:	{1,2}	{1,2,3}	{0,1,2,3}

Sequential EC

- Backward state traversal

```
Algorithm TRAVERSE_BACKWARD(t, λ ,S0) {
  reached = ∅
  current = ∃x.(λ(x,s)=1)                         // start from bad
  while (reached ≠ (reached ∨ current)) {  // fixed point
    reached  = reached ∨ current                // add new states
    previous = PRE_IMG(t,current)               // one step transition
    current  = previous                         // rename variable
  }
  return (S0 ∧ reached)
}
```

- Example



Iteration:	1	2	3
Reached:	{6}	{4,6}	{4,5,6}
Current:	{6}	{4}	{4,5}
Previous:	{4}	{4,5}	{4,5,6}

Sequential EC

- Explicit reachability analysis
  - Represent states explicitly (e.g. as bit string) => limited capacity
  - Use hashtable to find quickly whether state was reached before
  - Image operation: simple simulation
  - Preimage operation: SAT run
- Symbolic reachability analysis
  - Represent states and transition relation symbolically
    - E.g. BDDs, circuits, DNF, etc.
  - Use BDD operations to perform image and preimage operation (simple AND or AND_EXIST)
  - Lots of heuristic improvements to keep BDD size under control

Sequential EC

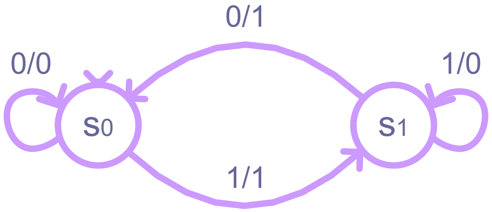
- Let R(s) be the characteristic function of the set of reachable states of the product FSM $M_{1\times2}$ obtained from forward reachability analysis. Then FSMs M_1 and M_2 are equivalent if and only if

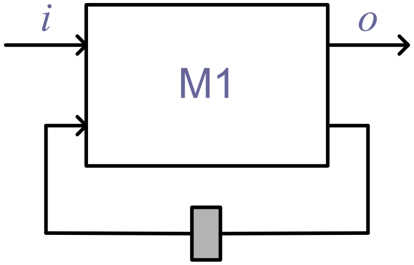
  $$\lambda_{1\times2}(x,s) \wedge R(s)$$

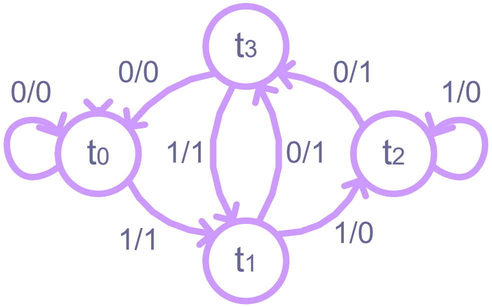
  is constant 0 for all valuations on input variables x and state variables s
  - This can be checked in constant time for BDD

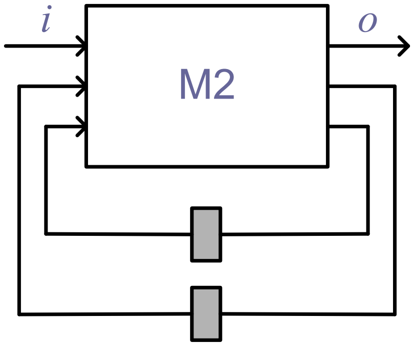
Sequential EC

- Example
  - To check: The equivalence of M_1 and M_2



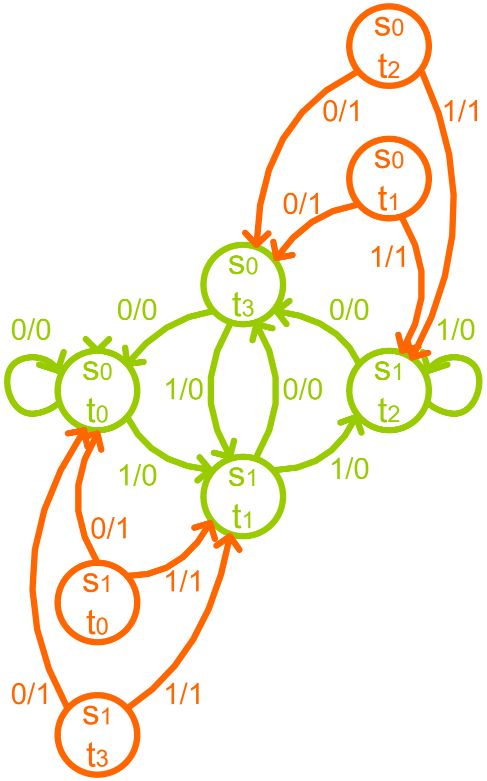


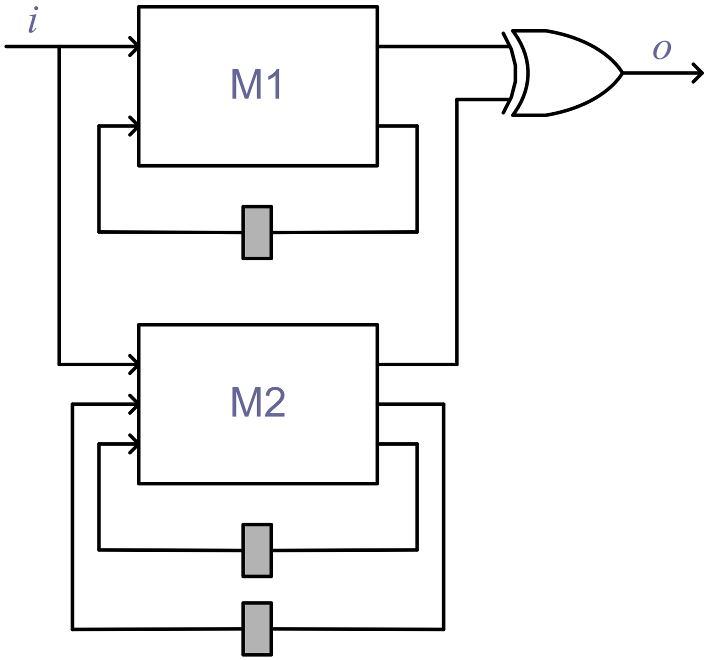




Sequential EC

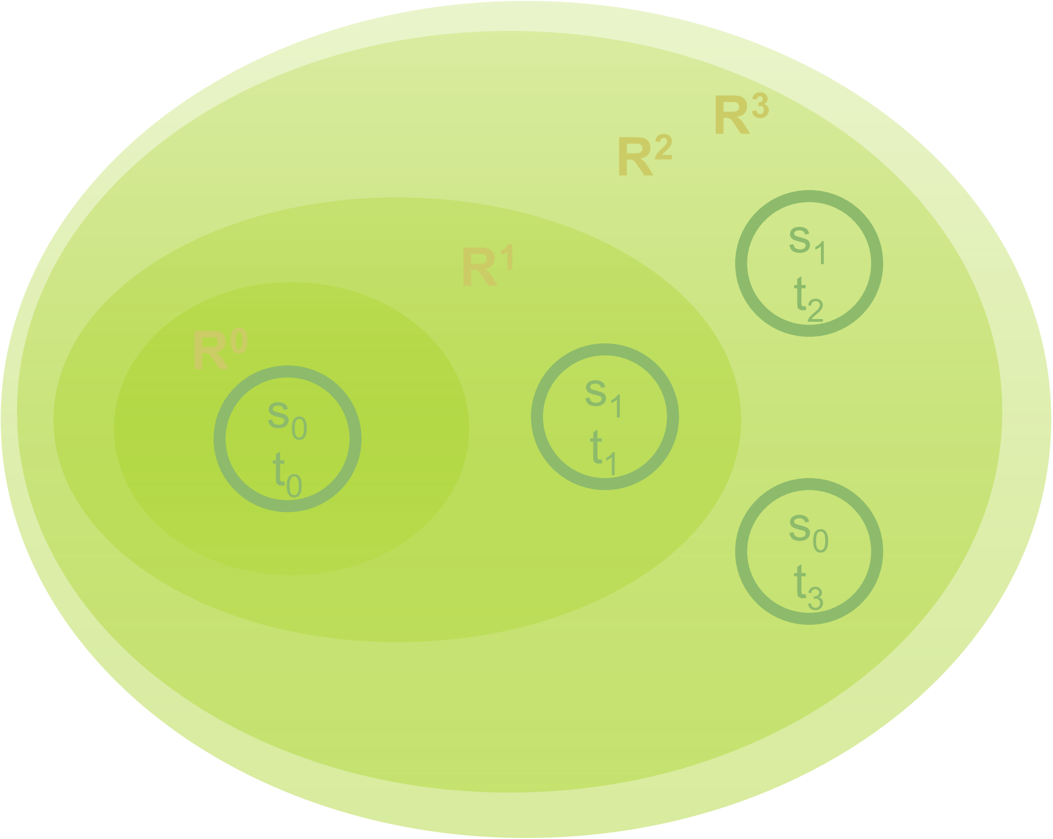
- Example (cont'd)
  - Construct product FSM of M_1 and M_2

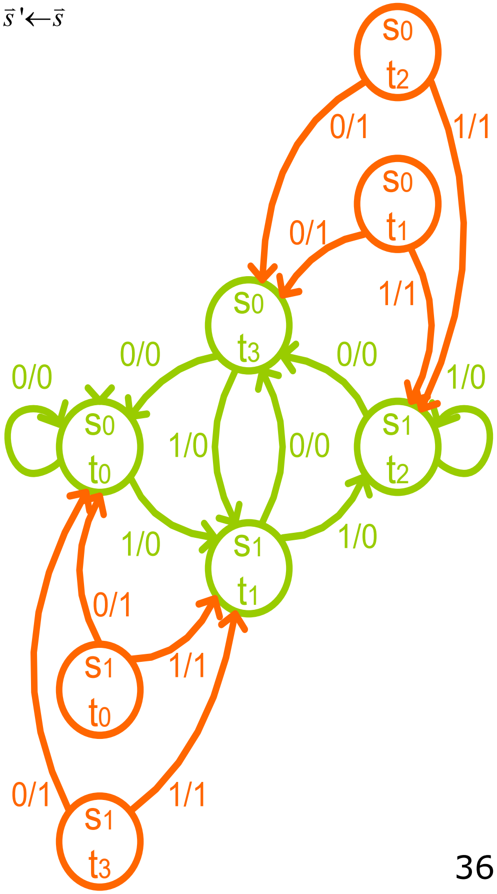




Sequential EC

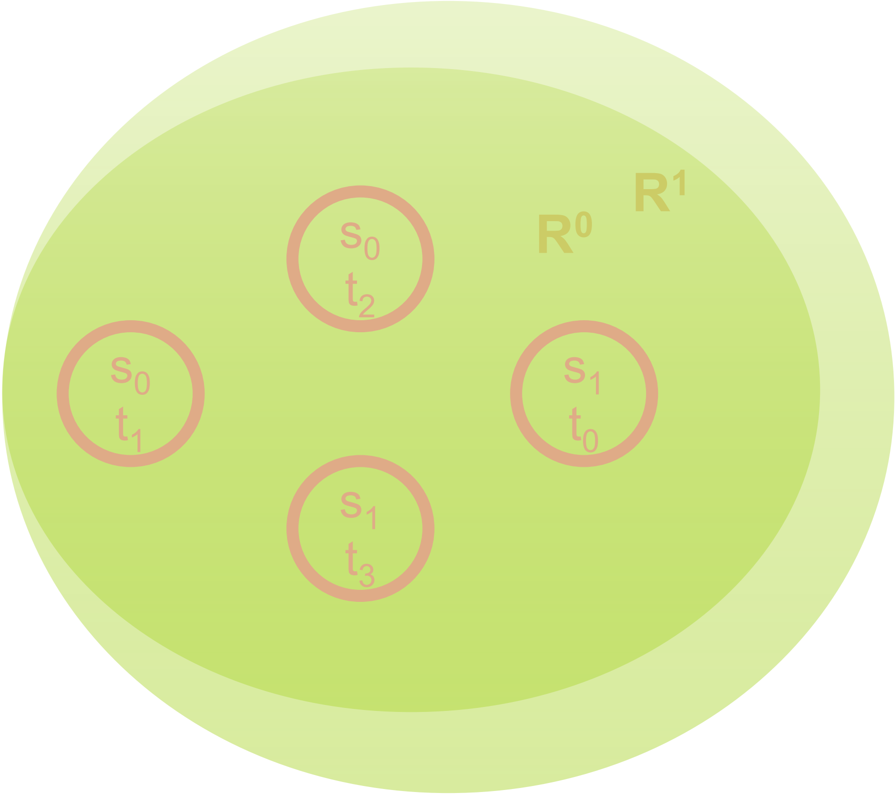
- Example (cont'd)
  - Forward reachability analysis based on image computation $Img(C,T) = [\exists \vec{x}, \vec{s}. T(\vec{x},\vec{s},\vec{s}\,') \wedge C(\vec{s})]_{\vec{s}\,' \leftarrow \vec{s}}$

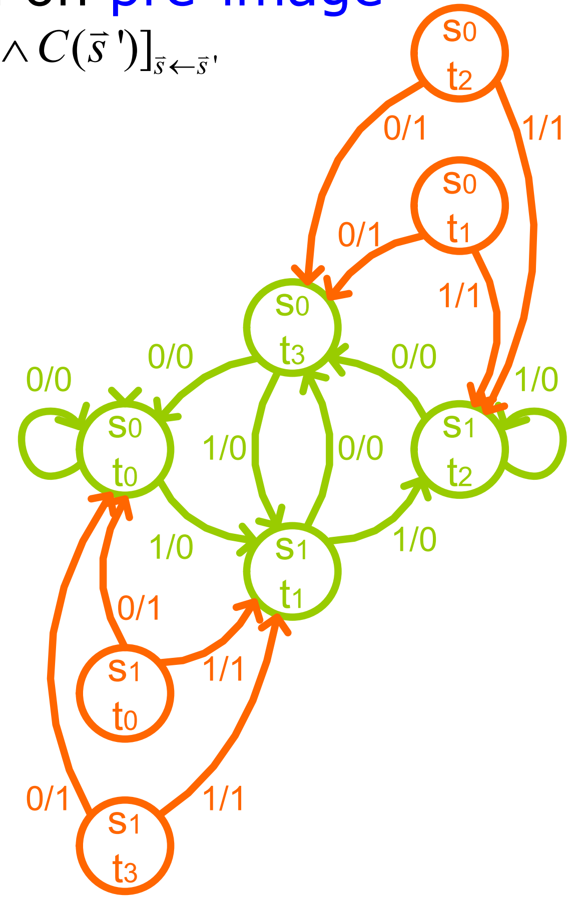




Sequential EC

- Example (cont'd)
  - Backward reachability analysis based on pre-image computation $PreImg(C,T) = [\exists \vec{x}, \vec{s}\,'.T(\vec{x},\vec{s},\vec{s}\,') \wedge C(\vec{s}\,')]_{\vec{s} \leftarrow \vec{s}\,'}$



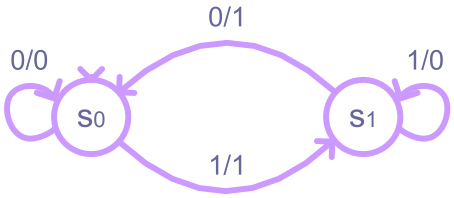


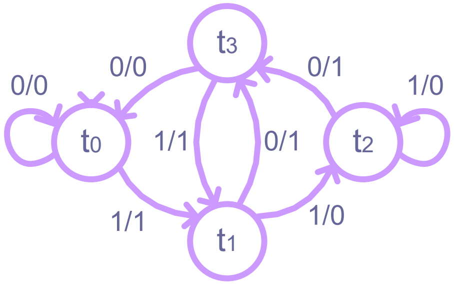
Sequential EC

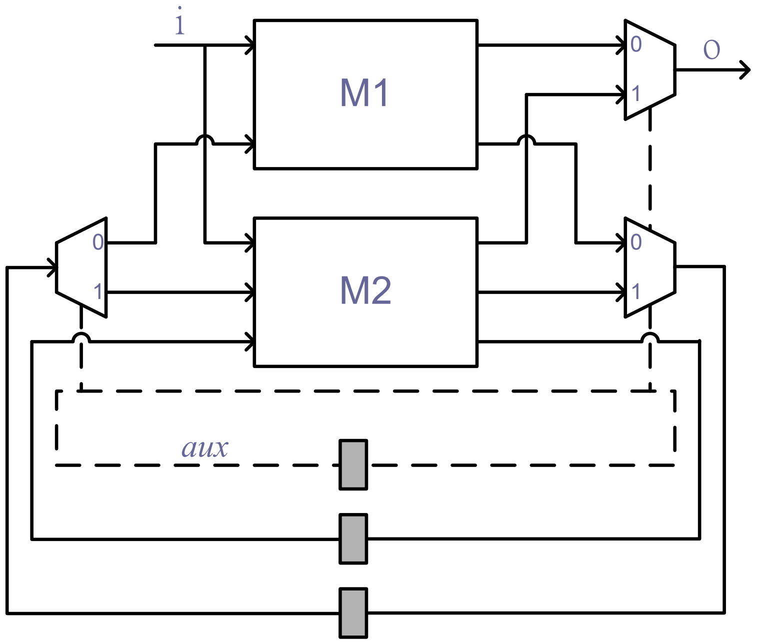
- Alternative approach beyond reachability analysis
  - Based on state equivalence
    - Two FSMs are equivalent if and only if their initial states are equivalent
      - Two states of an FSM are equivalent if starting these two states the FSM behaves indistinguishably
  - Explicit algorithm (based on state transition graph enumeration) is known
    - Used in state minimization where equivalent states must be identified
  - How about implicit algorithm (based on Boolean manipulation) ?

Sequential EC

- State partitioning based sequential EC
  - Construct and multiplexed FSM (disjoint union of the state graphs)
- Example



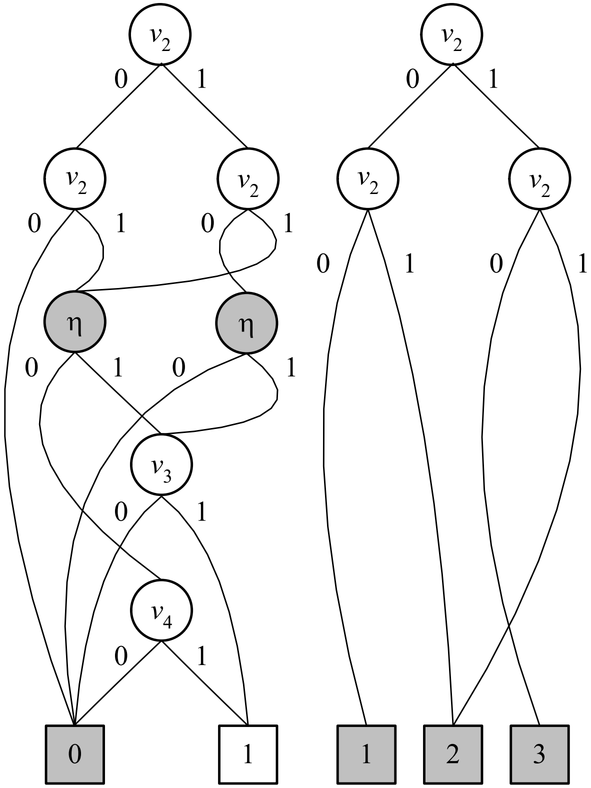




Sequential EC

- State partitioning over multiplexed FSM
  - Using BDD-based functional decomposition
- Example (cont'd)

$\eta\ v_3\ v_4$ \ $v_1\ v_2$	0 0	0 1	1 0	1 1
0 0 0	0	0	0	0
0 0 1	0	1	0	1
0 1 0	0	0	0	0
0 1 1	0	1	0	1
1 0 0	0	0	0	0
1 0 1	0	0	0	0
1 1 0	0	1	1	1
1 1 1	0	1	1	1

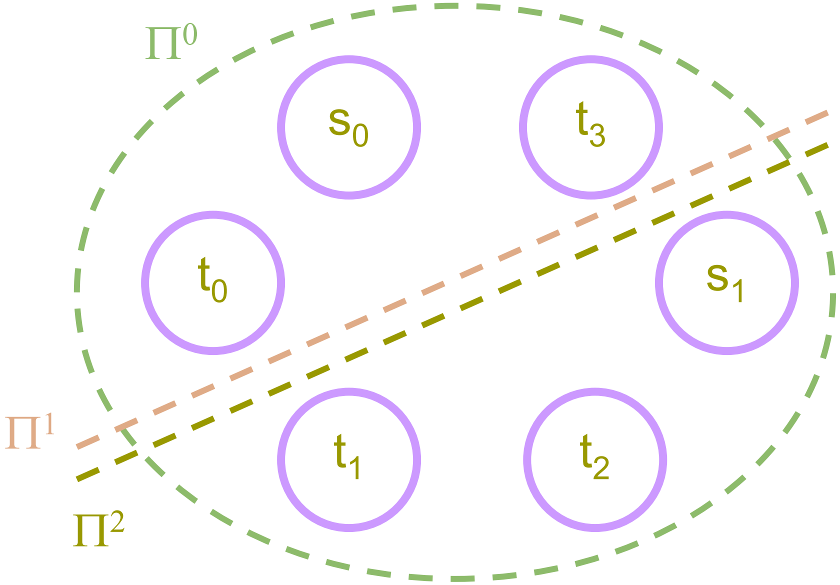


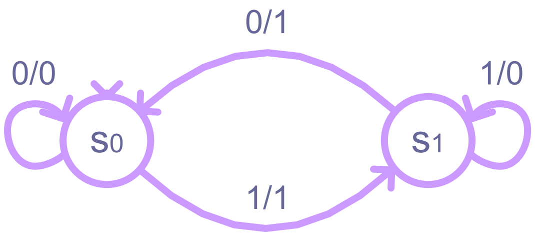
Sequential EC

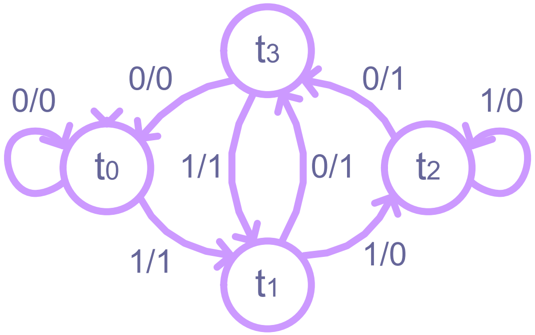
- State partitioning based sequential EC
  - BDD-based functional decomposition
    - Bound set variables (top): state variables
    - Free set variables (bottom): others
    - Cutset: free-set nodes with incoming edges from bound-set nodes
  - Paths leading to a node in the cutset form an equivalence class of states (for an iteration)
  - Iterate functional decomposition over composed functions

Sequential EC

- Example (cont'd)
  - State partitioning







Sequential EC

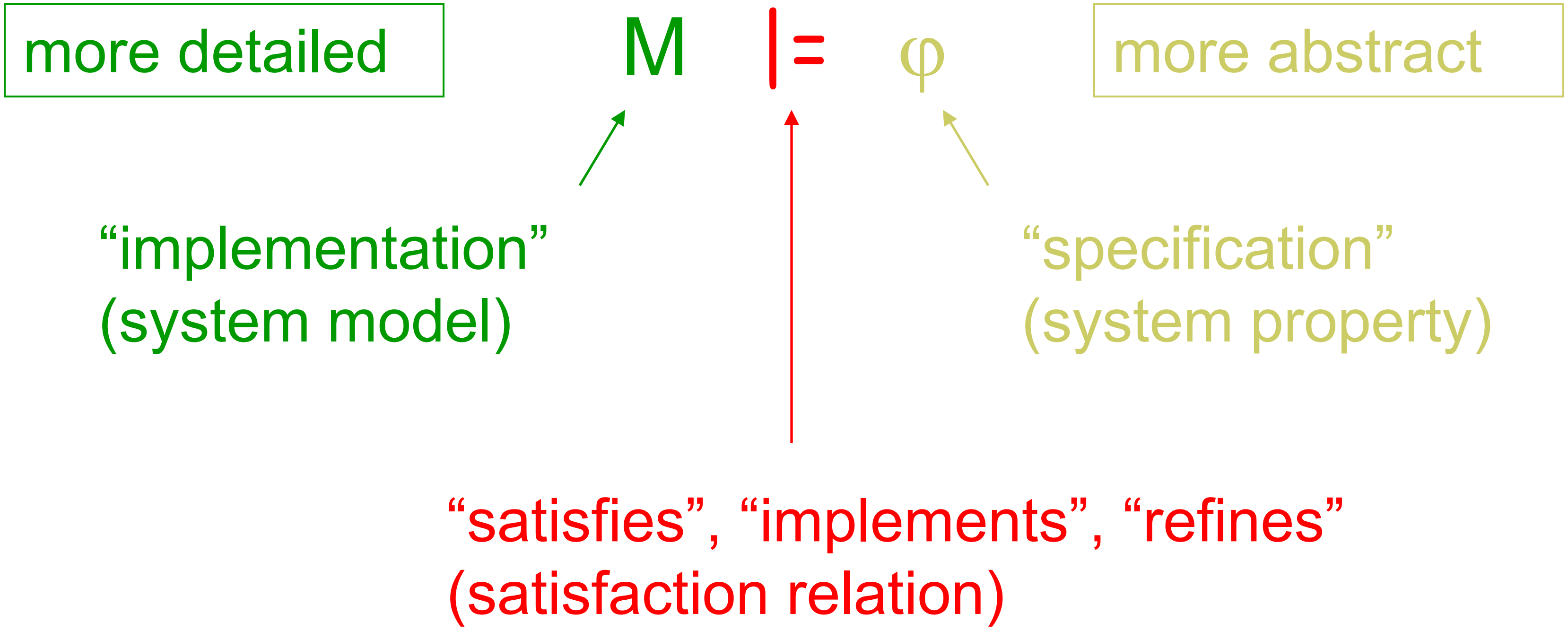
- Connection between reachability based SEC and state partitioning based SEC
  - Backward reachability analysis can be considered as state partitioning in the product state space

Sequential EC

- Summary
  - Industrial EC checkers almost exclusively use an combinational EC paradigm even for sequential EC
    - Sequential EC is too complex and can only be applied to design with a few hundred state bits
    - Structure similarity should be identified to simplify sequential EC
  - Besides sequential equivalence checking, reachability analysis is useful in sequential circuit optimization
    - Recall in sequential optimization that unreachable states can be used as sequential don’t cares to optimize a sequential circuits

Model Checking

- A model checking problem is defined by



more detailed

$M \;\; |= \;\; \varphi$

more abstract

"implementation"
(system model)

"specification"
(system property)

"satisfies", "implements", "refines"
(satisfaction relation)

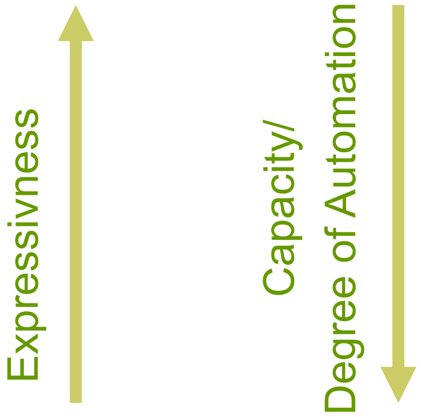
Model Checking

- M |= φ
  - Check if system model M satisfies a system property φ
  - System model M is described with a state transition system
    - finite state or infinite state
  - Temporal property φ can be described with three orthogonal choices:
    1. operational vs. declarative: automata vs. logic
    2. may vs. must: branching vs. linear time
    3. prohibiting bad vs. desiring good behavior: safety vs. liveness

Different choices lead to different model checking problems.

Property Checking

- Assertion-based verification
  - Properties are expressed as RTL annotations in terms or assertions (“This statement must hold true”)
  - E.g. AG(x=y) “For all paths from the initial state and all successor states x=y”
- Formal verification methods:
  - Exhaustive, do not require simulation vectors
- Main methods:
  - Theorem proving
  - Model Checking
    - Liveness property checking
    - Safety property checking
  - Refinement checking
  - Equivalence checking
  - Bounded property checking



Expressivness

Capacity/
Degree of Automation

Property Checking

- Safety property: Something “bad” will never happen
  - Safety property violation always has a finite witness
    - if something bad happens on an infinite run, then it happens already on some finite prefix
  - Example
    - Two processes cannot be in their critical sections simultaneously

- Liveness property: Something “good” will eventually happen
  - Liveness property violation never has a finite witness
    - no matter what happens along a finite run, something good could still happen later
  - Example
    - Whenever process P1 wants to enter the critical section, provided process P2 never stays in the critical section forever, P1 gets to enter eventually

For finite state systems, liveness can be converted to safety!

Safety Property Checking

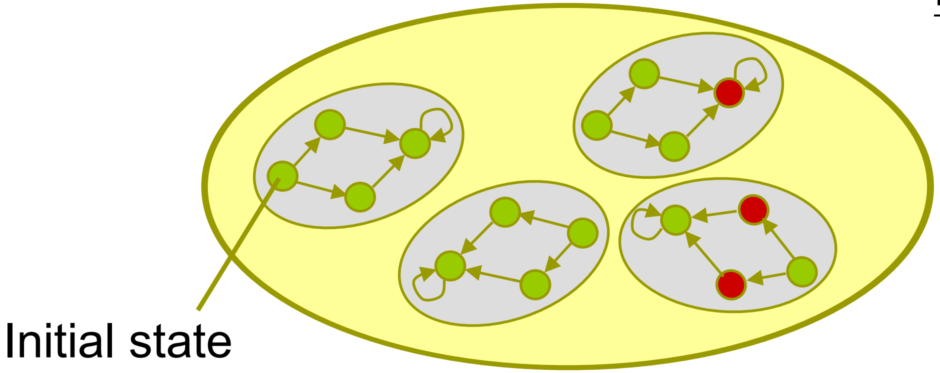
- Safety property checking can be formulated as a reachability problem
  - Are bad states reachable from good states?

- Sequential equivalence checking can be considered as one kind of safety property checking
  - M : product machine
  - φ : all states reachable from initial states has output 0

Safety Property Checking

- Concept:
  - Counter example has finite length
  - Specification in terms of “bad behavior” that should not happen
  - E.g. specify a state with a bad property or a bad output condition
  - Handles 95% of practical properties

- Basic approach:
  - Express property as formula on state and inputs
  - Single reachability analysis sufficient to decide about correctness



Property:

AG(^overflow)

“The history buffer never overflows”

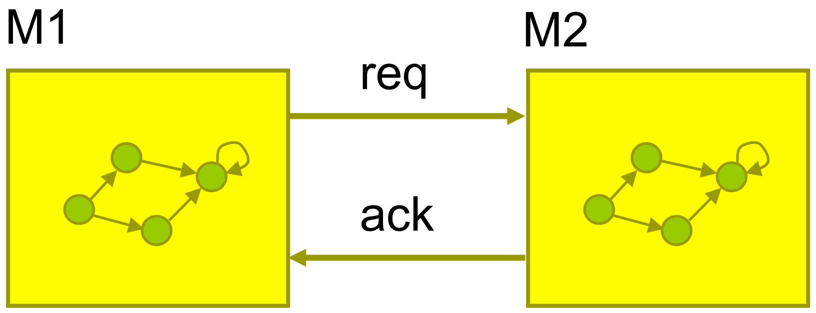
● Bad state (overflow)

● Good state (no overflow)

Liveness Property Checking

- Concept:
  - Counter example has infinite length
  - Specification in terms of “good behavior” that should always happen
  - E.g. AG(req=>AF ack)

- Basic approach:
  - Nested reachability analysis according to formula



Property:

AG(req=> AF ack)

“A request from M1 will always be acknowledged by M2”

Model Checking

- Data structure evolution in model checking
  - State graph (late 70s-80s)
    - Problem size ~10^4 states
  - BDD (late 80s-90s) – symbolic model checking
    - Problem size ~10^{20} states
    - Critical resource: memory
  - SAT (late 90s-) – bounded/unbounded model checking
    - GRASP, SATO, chaff, berkmin
    - Problem size ~10^{100} (?) states
    - Critical resource: CPU time

Bounded Model Checking

- Bounded Model Checking (Biere, et al., TACAS 1999):
  - Property checking method based on finite unfolding of transition relation interleaved with checks of the property
    - Sound: in its pure form no false positives are possible
    - Incomplete: cannot guarantee correctness of property
  - Basic method:
    - CNF-based:
      - Use CNF-based SAT solver to represent unfolding and proof UNSAT for correctness of property
    - Circuit-based:
      - Use ATPG-like reasoning to show untestability
    - Hybrid:
      - Use circuit rewriting and SAT checking interleaved
        - e.g. based on AND/INV graphs

Bounded Model Checking

- Notation
  - Variables for current and next state: *s, s'*
  - Predicate for transition relation: *t(s,s')*
    - *t(s,s')*=1 iff there is a transition from *s* to *s'*
  - Predicate for initial states: *i(s)*
    - *i(s)*=1 iff s is an initial state
  - Predicate for property: *p(s)*
    - *p(s)*=1 iff s satisfies property *p*
  - Predicate for all paths of length *k*:

    $t^k(s_0,s_k) = \prod_{0\leq i<k} t(s_i,s_{i+1})$
    - $t^k(s_0,s_k)$=1 iff there is a transition path of length *k* from s_0 to s_k

Bounded Model Checking

- BMC for length *k*

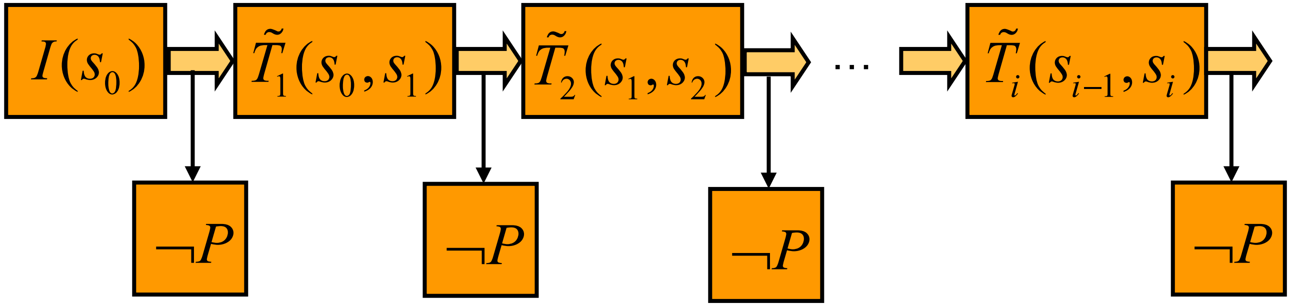
  $BMC_k = i(s_0) \wedge t^k(s_0,s_k) \wedge \neg p(s_k)$

- BMC loop

```
Algorithm BMC(max_length){
  forall 0 ≤ k < max_length do {
    if(SAT($BMC_k$)) return FAIL
  }
  return SUCCESS;
}
```

Bounded Model Checking

- BMC unfolding
  - Time-frame expansion



Comments:

- Any SAT technique can be used for checking frames
- Combination with random simulation, parallel runs etc.

Unbounded Model Checking

- K-step induction [Sheeran, FMCAD 2000]
  - Assert correctness of properties proven for previous frames

    $$tp^k(s_0,s_k) = \bigwedge_{0\le i<k} p(s_i) \wedge t(s_i,s_{i+1})$$

  - Simple path constraint
    - No state visited twice

    $$tp^k_{simple}(s_0,s_k) = \bigwedge_{0\le i<k} p(s_i) \wedge t(s_i,s_{i+1}) \wedge \bigwedge_{0\le i<j\le k} s_i \ne s_j$$

  - K-step inductiveness
    - In addition to *BMC_k* check also

    $$inv^k = tp^k(s_0,s_k) \wedge \neg p(s_k)$$

- Interpolation [McMillan, CAV 2003]

- SAT-based model checking without unrolling [Bradley, VMCAI 2011]

Model Checking

- Summary
  - Temporal logic is a variation of mathematical logic and is concerned with temporal reasoning
    - Developed since 1970's
  - Model checking is concerned with algorithmic verification of temporal properties
    - Developed since 1980's
    - Hardware model checking techniques are being applied in the software domain
  - Reference
    - K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993
    - M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999