Model Checking with Automata An Overview

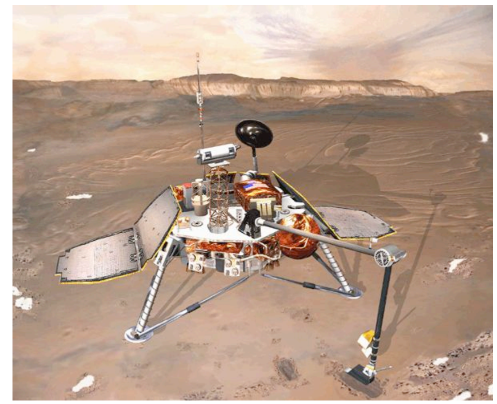
Vanessa D. Carson
Control and Dynamical Systems, Caltech

Contents

- Motivation
- Overview Software Verification Techniques
- Model Checking
  - System Modeling
  - Specification Modeling
  - Verification
- Next Lecture

Motivation

Software bugs are hard to find



- Example: Mars Polar Lander 1999
  - Study Martian weather, climate, water and $C0_2$ levels
  - Last telemetry sent prior to atmospheric entry
  - Potential software/hardware error
    - Logic for engine cutoff that engaged when lander legs were deployed ~40 m above ground
    - Some vibration caused sensors to trip engine cut-off
- Input and its effects on state not considered appropriately

Software systems are complex

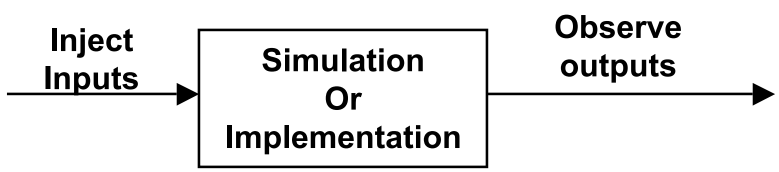
- Multiple processes running concurrently
  - sensors, planners, actuators
  - complexity of process interleavings
  - reasoning about distributed systems
- Interested in systems that do not halt

$\boxed{\text{Assure that system behaves as intended}}$

Techniques in Software Verification

Software Testing

Simulation and Testing

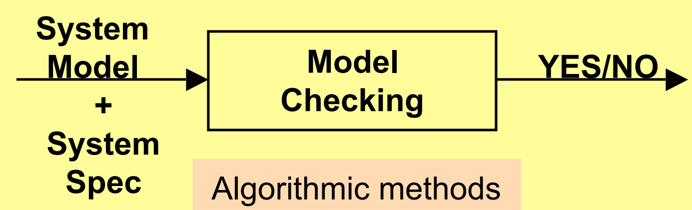


- Proving coverage
  - Rarely possible to check *all* software interactions
- Tools: Code coverage tools, parsers, random testing

Formal Verification

Deductive Verification
Model Checking

Proof methods

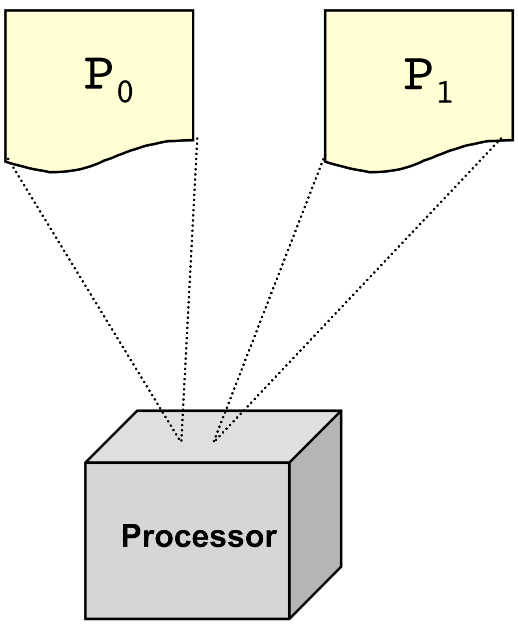


Algorithmic methods

- Complex systems => large models
  - DV: Length of proof/expertise
  - MC: physical limitations
- Tools: Isabelle, HOL, ACL2,PVS (DV), SPIN (MC)

Concurrency and Shared Variables

Two processes P_0 and P_1 executing on a single core CPU



P_0 and P_1 are loaded into the processor and executed one at a time according to a schedule

Scheduler
P_0
OS process
P_1
P_0

Determines what process
to put into context based
on some scheduling algorithm

P_0 and P_1 may share some memory and may read and write to it

Memory
P_0Var=100
writeBuf="xy"
OSVar='root'
P_1Var=50

Shared variable between process P0 and P1

Concurrency Issues

```
P0   while True do
         getScienceData(scienceData);
         writeToBuffer(writeBuff, scienceData);
         writeToFile(writeBuff, scienceFile);
         clearBuffer(writeBuff);
      end while;
```

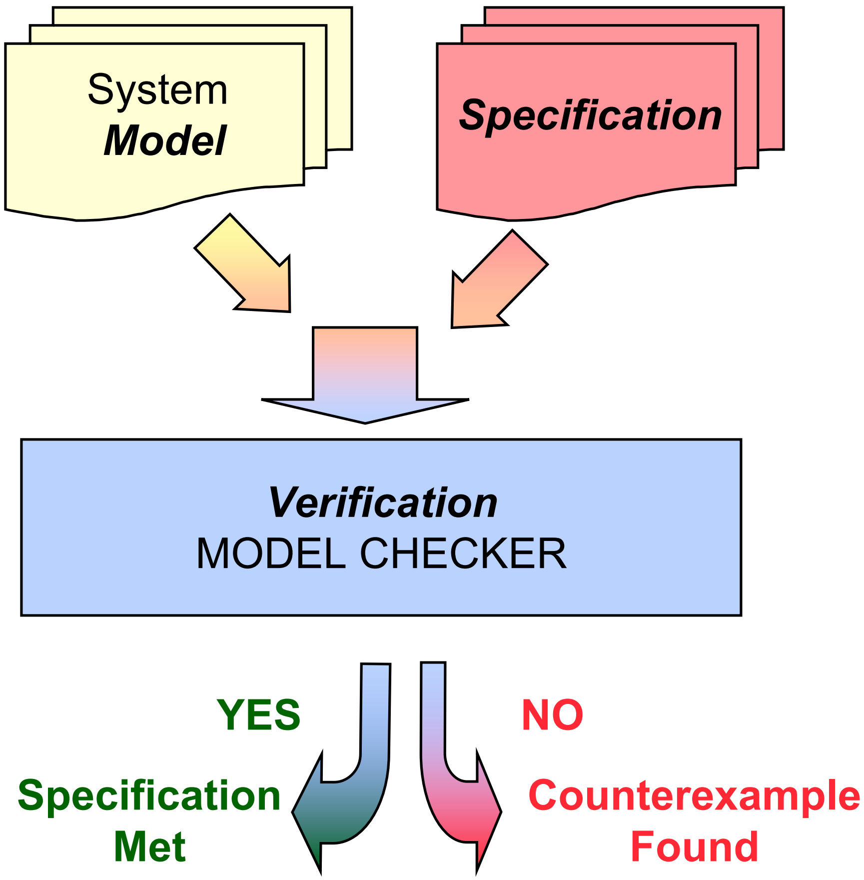
P_0 writes science data to scienceFile

Science Data Overwritten!

```
P1   while True do
         getSpacecraftHealthData(healthData);
         writeToBuffer( writeBuff, healthData);
         writeToFile(writeBuff, healthFile);
         clearBuffer(writeBuff);
     end while;
```

P_1 writes spacecraft health data to healthFile

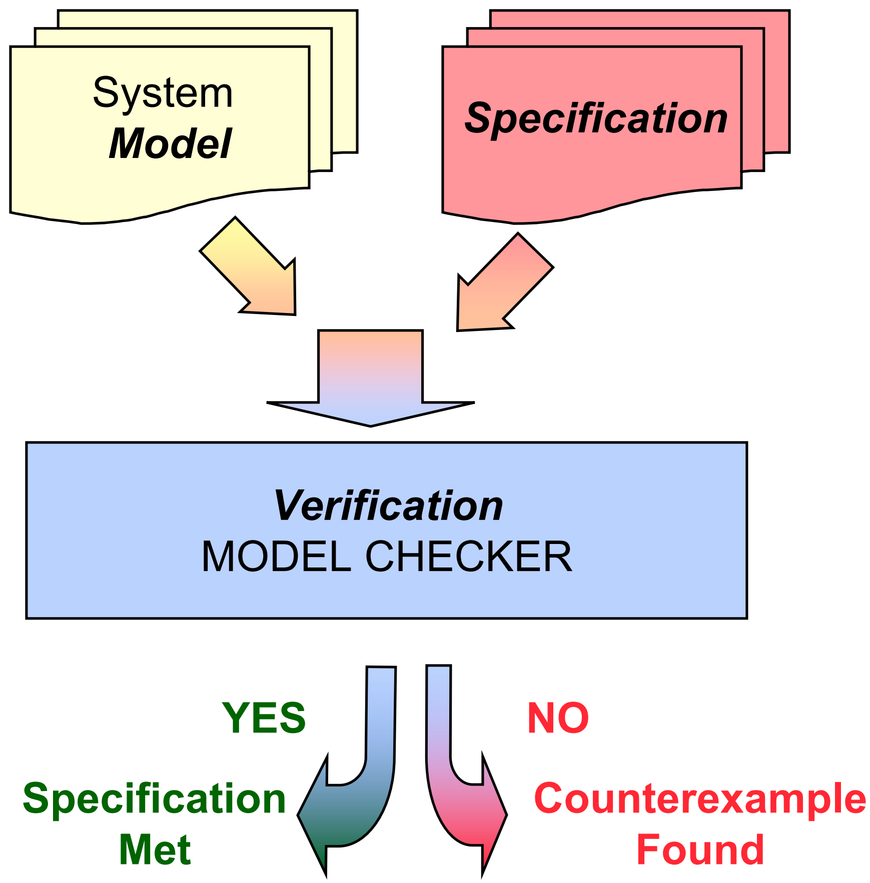
Model Checking Process



- ***Modeling:*** Convert a design to a formalism accepted by a model checker.
- ***Specification****:* State the properties that the design must satisfy.
- ***Verification****:* Verify correctness of specification with respect to the model.

Verification is performed automatically by an exhaustive search of the state space of the system.

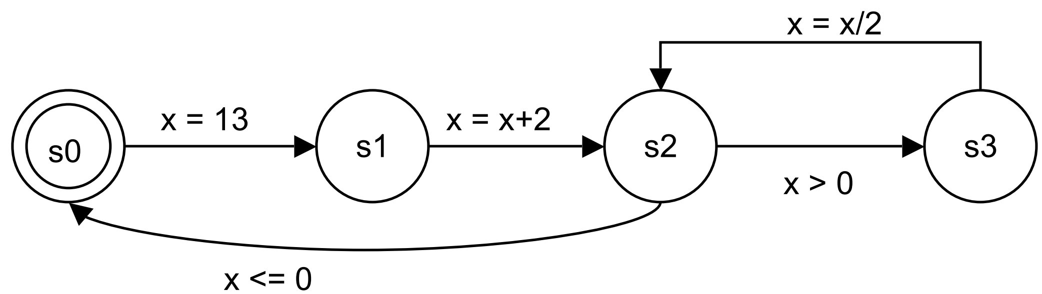
System Modeling



Modeling with Finite State Automata

```
int x = 13;

while (x == 13) {
  x = x + 2;
  while (x > 0) {
    x = x/2;
  }
}
```



A *finite state automaton* is a tuple $(\Sigma, S, S_0, \Delta, F)$ where

Σ is a finite alphabet

S is a finite set of *states*

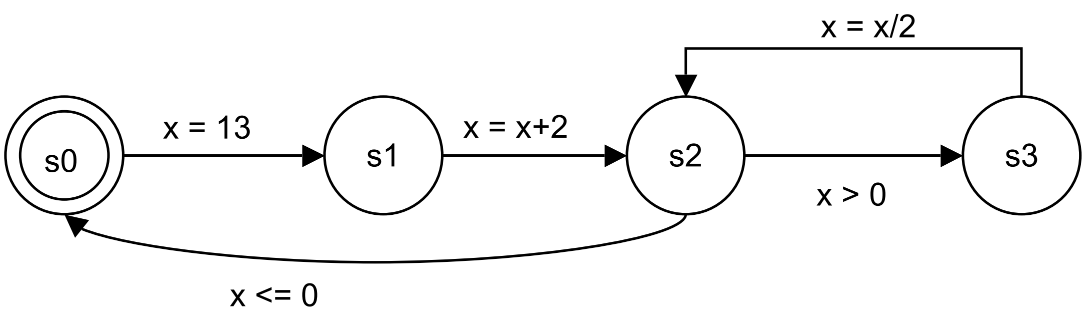
$S_0 \subseteq S$ is the set of initial states

$\Delta \subseteq (S \times \Sigma \times S)$ is a set of transition relations

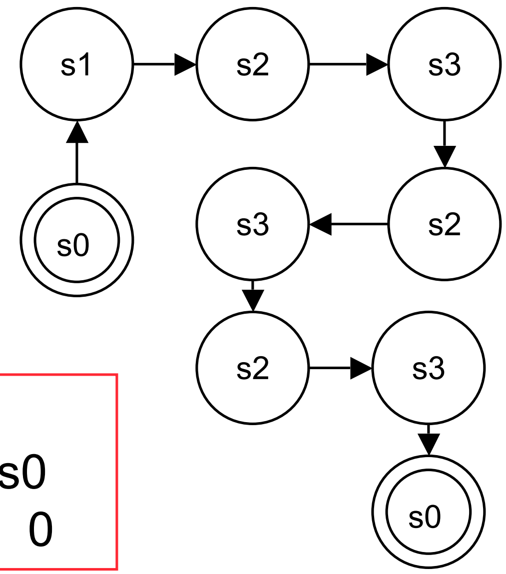
$F \subseteq S$ is a set of final states

Reasoning about Program Execution

The automaton A



An execution of A



A run of the automaton r = s0 s1 s2s3 s2s3 s2s3 s2s3 s0
13 15 7 3 1 0

A *run* of a finite state automaton A is a sequence of transitions $\rho = s_0 s_1 \ldots s_n$ of states $s_i \in S$ such that $s_0 \in S_0$ and $(s_i, l_i, s_{i+1}) \in \Delta, \forall i \in \mathbb{N}$.

A finite run ρ is *accepting* $\iff$ the final state $s_f \in F$.

Modeling with Büchi Automata

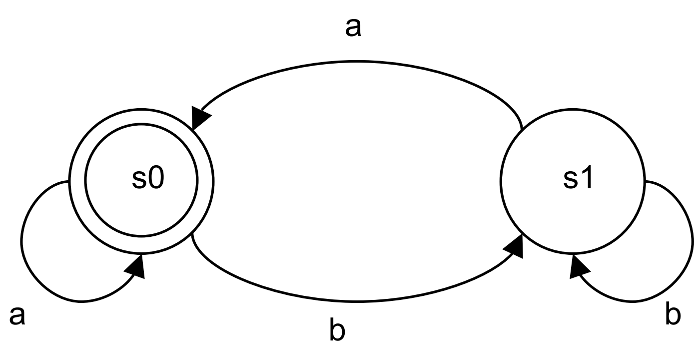
- Most concurrent systems do not to halt during normal execution
- Decide on acceptance of ongoing, potentially infinite executions
  - OS schedulers, control software
- Require Finite State Automata over *infinite* words

A ω-run of a finite state automaton A is an infinite sequence $\rho = s_0 s_1 \ldots s_n \ldots$ of states $s_i \in S$ such that $s_0 \in S_0$ and $(s_i, l_i, s_{i+1}) \in \Delta, \forall i \in \mathbb{N}$.

The run ρ is *accepting* $\iff \exists s \in F, \ni s_i = s$ for infinitely many $i \in \mathbb{N}$.
In other words, there exists $s \in F$ that appears infinitely often.

A ***Büchi*** automaton is a finite state automaton that accepts infinite runs.

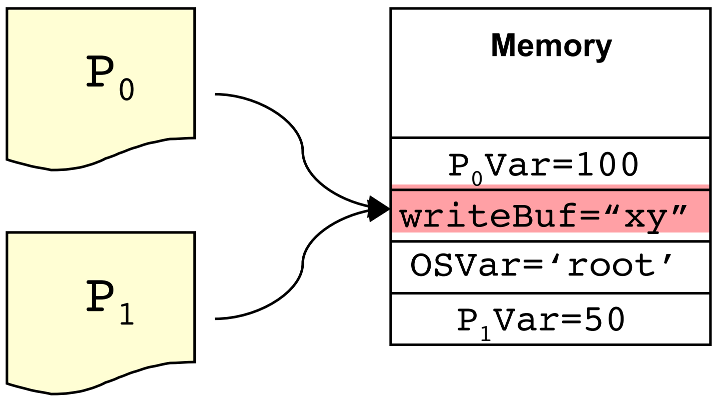
Büchi Automaton Language



- Two state Büchi automaton *A*
- Initial and accepting state s_0
- Accepts infinite number of symbol *a*
- *L(A)* ={set of *w*-words over {a,b} with infinitely many a's}
- $L(A) = \{(b^*a)^w\}$

The *language* of an automaton A, $L(A) \subseteq \Sigma^\omega$ is the set of ω-words for which there exists a run ρ of A and that run is accepting.

Mutual Exclusion



Problem

- P_0 alters the variable *writeBuf* over some execution steps
- P_1 gets triggered while P_0 is in the process of overwriting variable *writeBuf*
- *writeBuf* is in an inconsistent and unpredictable state

Solution

- Avoid simultaneous use of a common resource
- Divide code into ***critical sections*** to protect shared data
- ***Mutual exclusion*** algorithms exist
  - Lamport's Bakery, Peterson's, …

Mutual Exclusion Example

```
P0::l0: while True do
         NC0: wait(turn = 0);
         CR0:  turn := 1;
        end while;
    l'0
```

```
P1::l1: while True do
         NC1: wait(turn = 1);
         CR1:  turn := 0;
        end while;
       l'1
```

Problem Description

- Two asynchronous processes P_0 and P_1
- P_0 and P_1 share a variable *turn*
- P_0 and P_1 can not be in their critical section at the same time
- P_0 shall eventually enter into its critical region
- Model variables of interest
  - Shared variable state
  - Location of execution with program counter

Program Translation

- Manna, Pnueli (1995) Program translation formula
  - takes a sequential program and transforms to a first order formula that represents the set of transitions of the program

The initial states of each process P_I are described by the formula

$$S_0(V, PC) \equiv pc = m \wedge pc_o = \perp \wedge pc_1 = \perp$$

where $\perp$ indicates the process has been activated.

Apply translation procedure C, then for each process P_i

$$pc_i = l_i \wedge pc_i' = NC_i \wedge True \wedge turn' = turn$$

$$pc_i = NC_i \wedge pc_i' = CR_i \wedge turn = i \wedge turn' = turn$$

$$pc_i = CR_i \wedge pc_i' = l_i \wedge turn' = (i+1) mod(2)$$

$$pc_i = NC_i \wedge pc_i' = NC_i \wedge turn \neq i \wedge turn' = turn$$

$$pc_i = l_i \wedge pc_i' = l_i' \wedge False \wedge turn' = turn$$

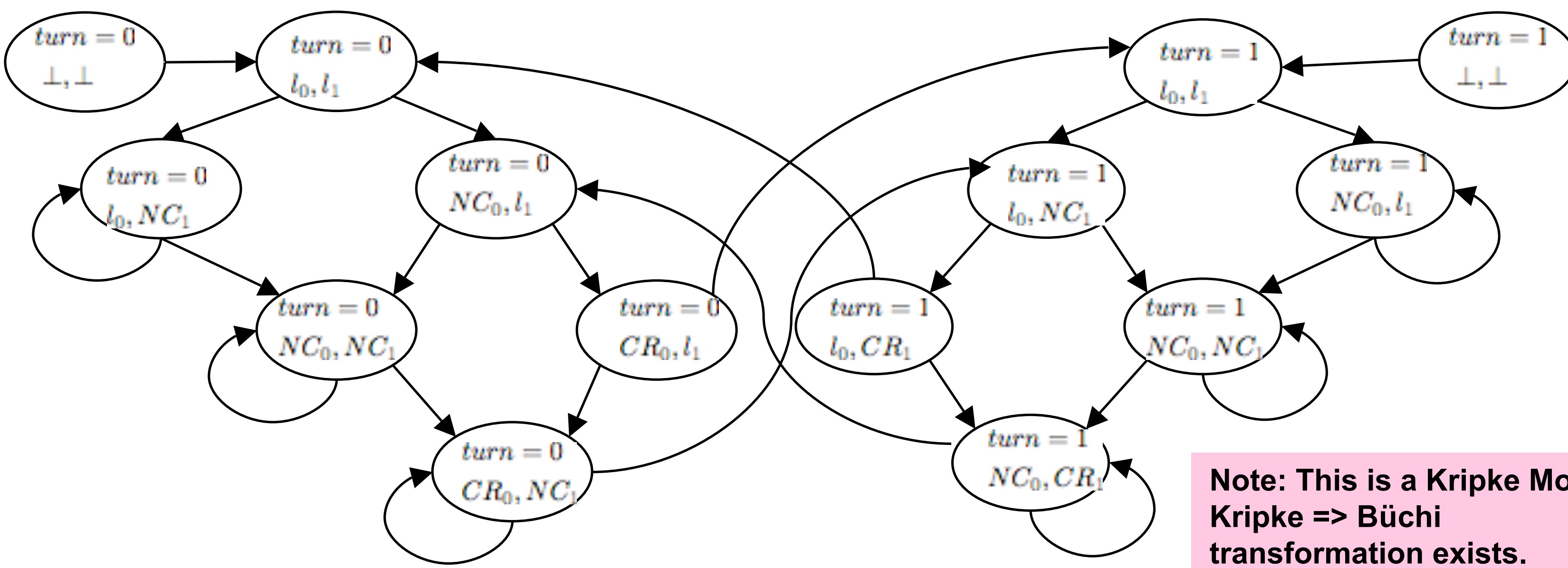
Mutex Model

P_0 has lock

```
P0::l0: while True do
          NC0: wait(turn = 0);
          CR0:  turn := 1;
        end while;
    l'0
```

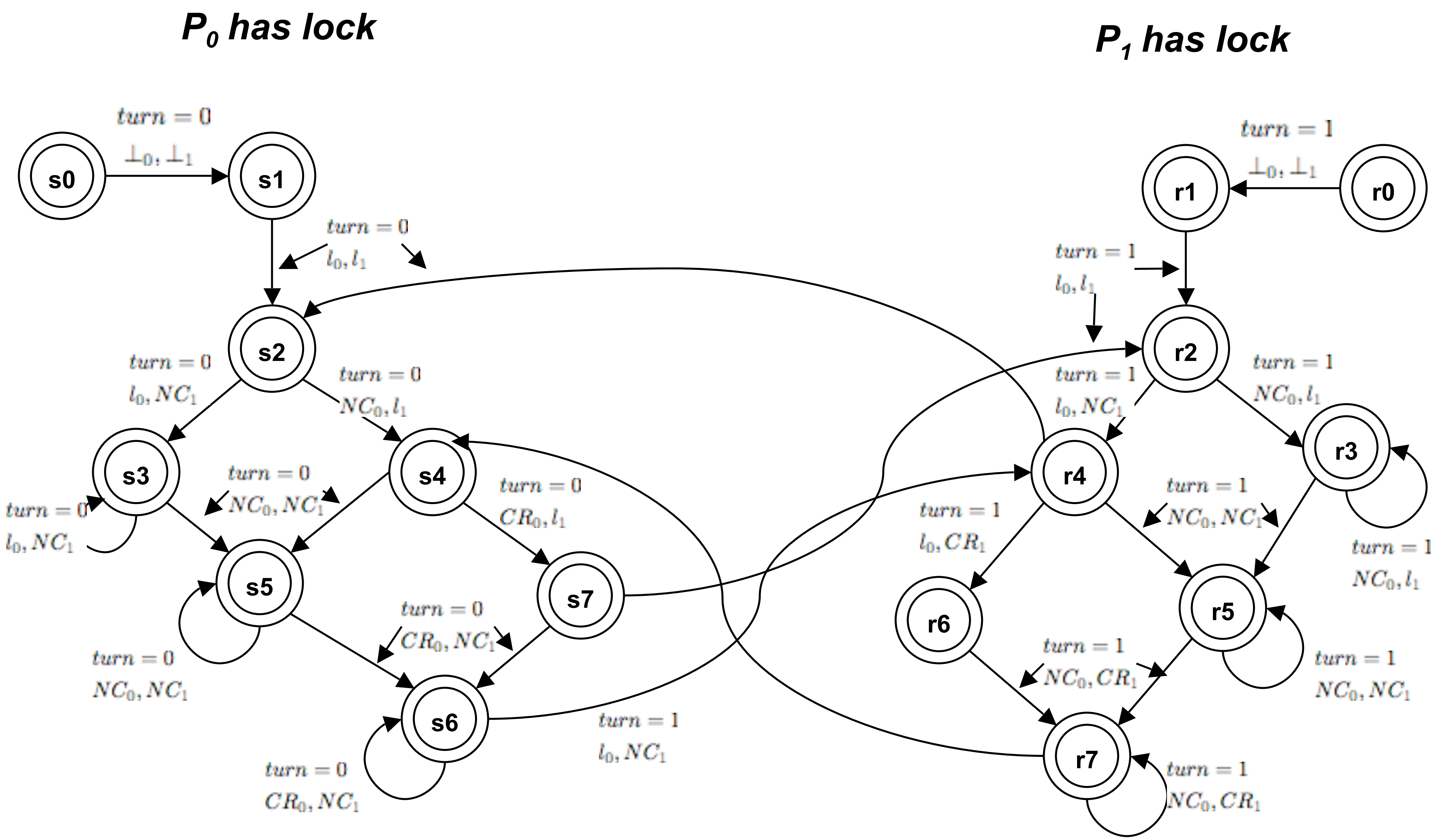
P_1 has lock

```
P1::l1: while True do
          NC1: wait(turn = 1);
          CR1:  turn := 0;
        end while;
    l'1
```

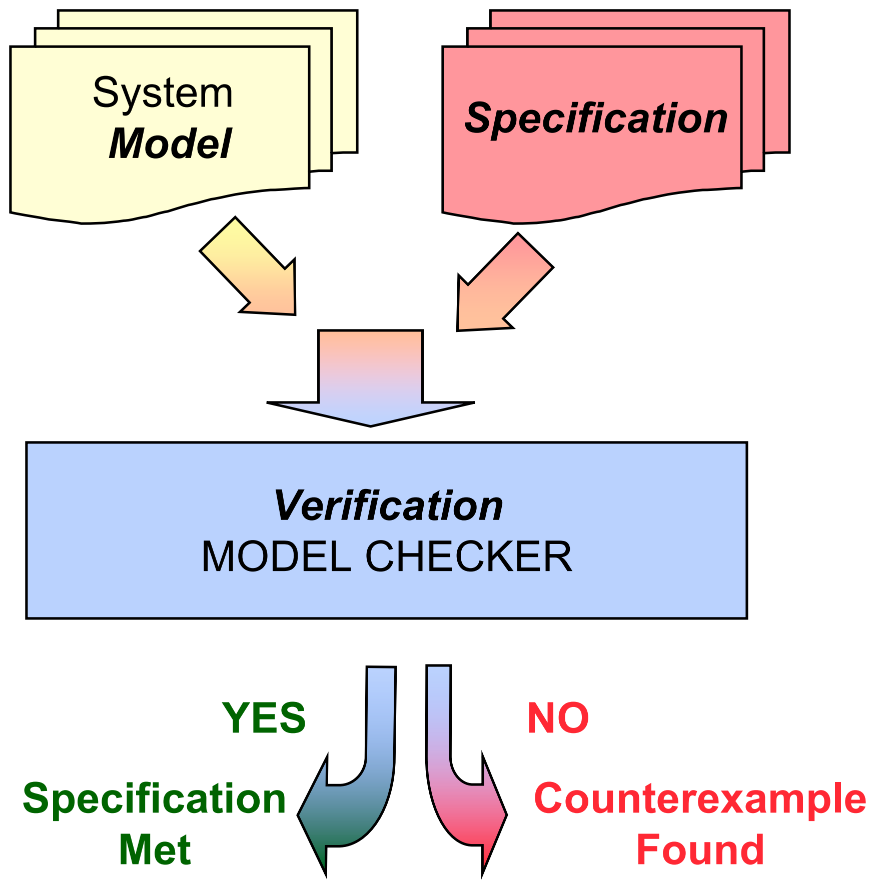


Note: This is a Kripke Model.
Kripke => Büchi transformation exists.

Mutex Büchi Model



Specification



Modeling Specifications

Goal: Model desirable properties of a system as correctness claims.

- Proving essential ***logical correctness*** properties ***independent*** *of*
  - Execution speeds
    - relative speeds of of processes, instruction execution time
  - Probability of occurrence of events
    - packet loss, failure of external device
- Two types of ***correctness claims*** [Lamport 1983, Pnueli 1995]
  - ***Safety*** set of properties the system may not violate
    - ***State properties****:* Claims about reachable/unreachable states
      - System invariant: holds in every reachable state
      - Process assertion: holds in specific reachable states
  - ***Liveness*** set of properties the system must satisfy
    - ***Path properties****:* Claims about feasible/unfeasible executions
- Several techniques available
  - LTL, Propositional Logic, Büchi Automata

LTL Specification

Safety	$\Box\neg\phi$
Liveness	$\Box\Diamond\psi$

Commonly used LTL formulas

FORMULA	DESCRIPTION	TYPE
$\Box p$	Always p	*Invariance*
$\Diamond p$	Eventually p	*Guarantee*
$p \rightarrow \Diamond q$	p implies eventually q	*Response*
$p \rightarrow q \cup r$	p implies q until r	*Precedence*
$\Box\Diamond p$	Always eventually p	*Recurrence*
$\Diamond\Box p$	Eventually always p	*Stability*
$\Diamond p \rightarrow \Diamond q$	Eventually p implies eventually q	*Correlation*

Let E be the complete set of ω-runs and let ϕ be a correctness property formalized as an LTL property.

The system satisfies the property ϕ if and only if all the ω-runs in E do.

Mutual Exclusion Example

```
P0::l0: while True do
          NC0: wait(turn = 0);
          CR0:  turn := 1;
         end while;
    l'0
```

```
P1::l1: while True do
          NC1: wait(turn = 1);
          CR1:  turn := 0;
         end while;
      l'1
```

Problem Description

- Two asynchronous processes P_0 and P_1
- P_0 and P_1 share a variable *turn*
- P_0 and P_1 can not be in their critical section at the same time
- P_0 shall eventually enter into its critical region
- Model variables of interest
  - Shared variable state
  - Location of execution with program counter

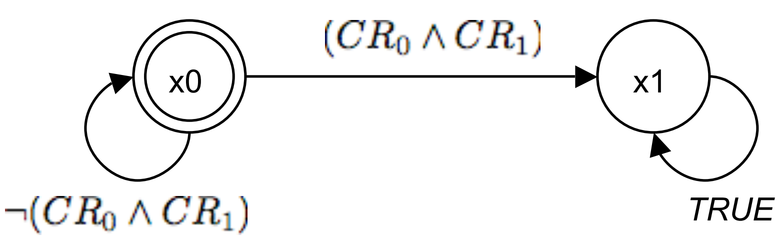
Mutex Specification

"Both processes can not simultaneously be in their critical regions"

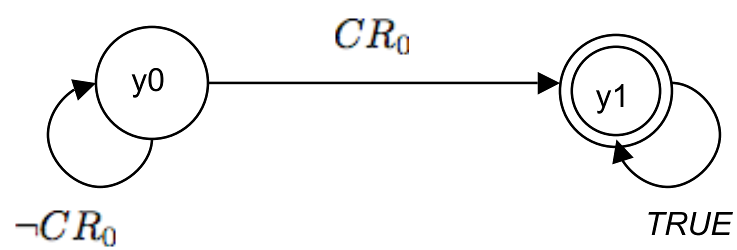
"The process P_0 will eventually enter its critical region"

$$\Box\neg(CR_0 \wedge CR_1)$$

$$\Diamond\, CR_0$$



Mutual Exclusion Property



Liveness Property

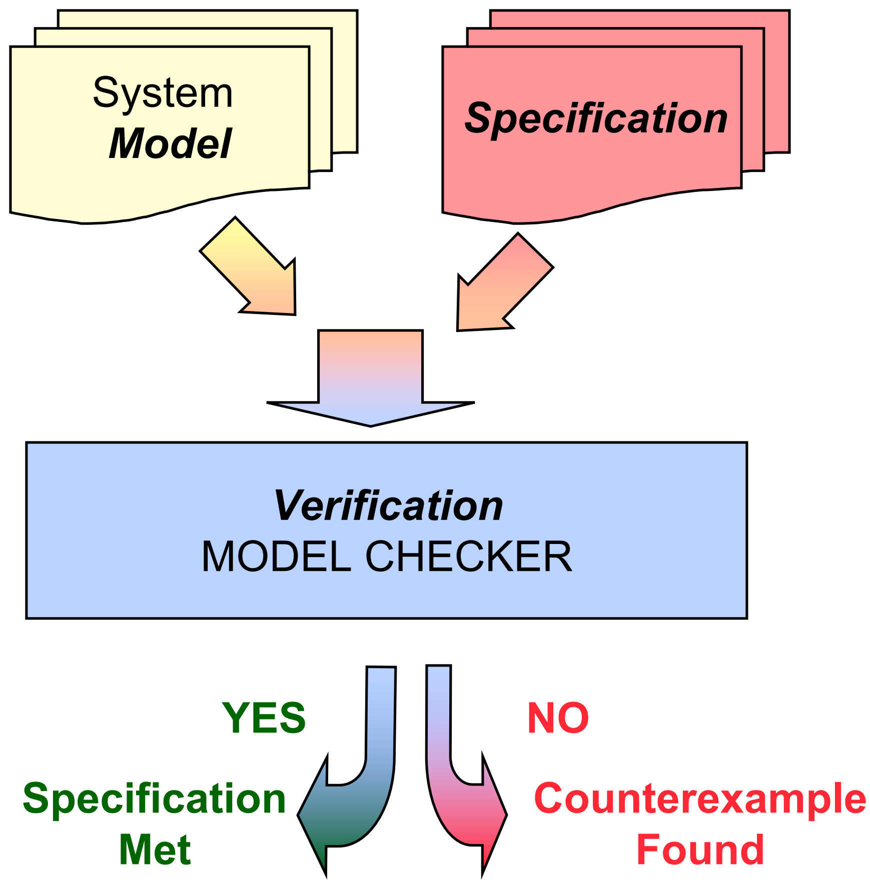
For every temporal logic formula there exists a Büchi automaton that accepts precisely those runs that satisfy the formula.

Properties of Büchi Automata

- Closed under intersection and complementation.
  - There exists an automaton that accepts exactly the intersection of the languages of a set of automata
  - There exists an automaton that recognizes the complement of the language of the given automaton
- Language emptiness is decidable
  - Whether the set of accepting runs is empty

The verification problem is equivalent to an emptiness test for an intersection product of Büchi automata.

Verification



Verification Condition

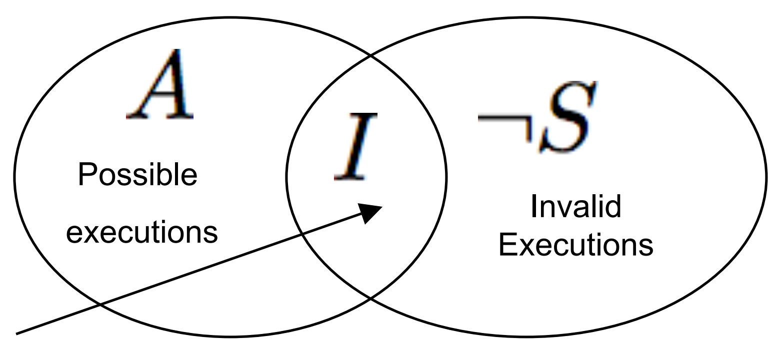
Goal: Verify that all possible behaviors of the model of the system A satisfy the specification S.

The system A satisfies the specification S when $L(A) \subseteq L(S)$.

$$\Updownarrow$$

Let $\overline{L(S)}$ be the language $\Sigma^\omega - L(S)$, then $L(A) \cap \overline{L(S)} = \emptyset$.

If I is empty, then A satisfies S.
If I is not empty, then A can violate S, and I contains at least one complete counterexample that proves it.



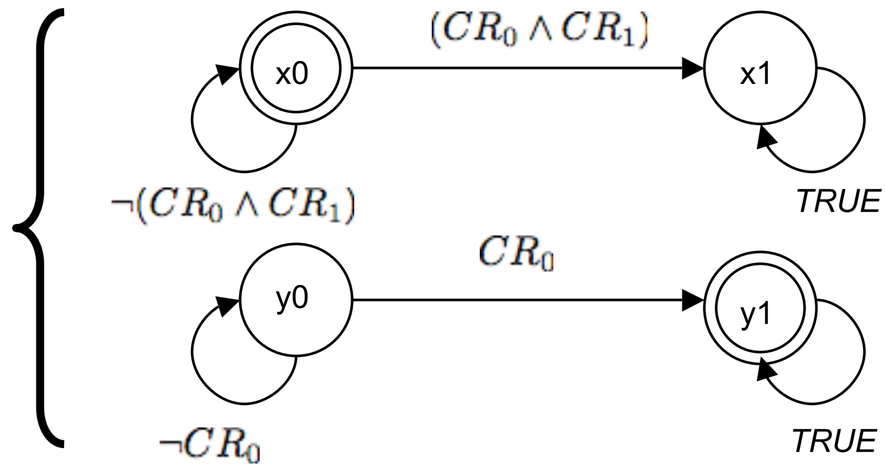
Executions that are possible and invalid

Complementing the Specification

LTL Specification

$$S \equiv (\Diamond CR_0) \wedge (\Box \neg (CR_0 \wedge CR_1))$$

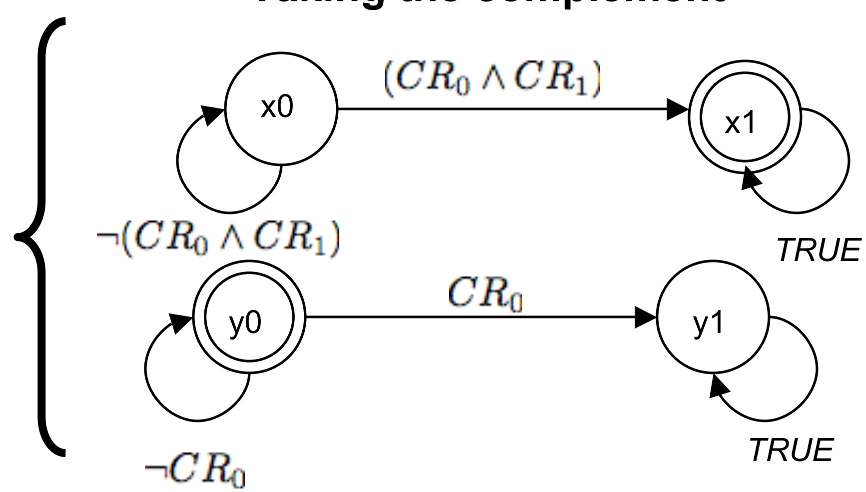
Büchi Automata Specification



Taking the negation

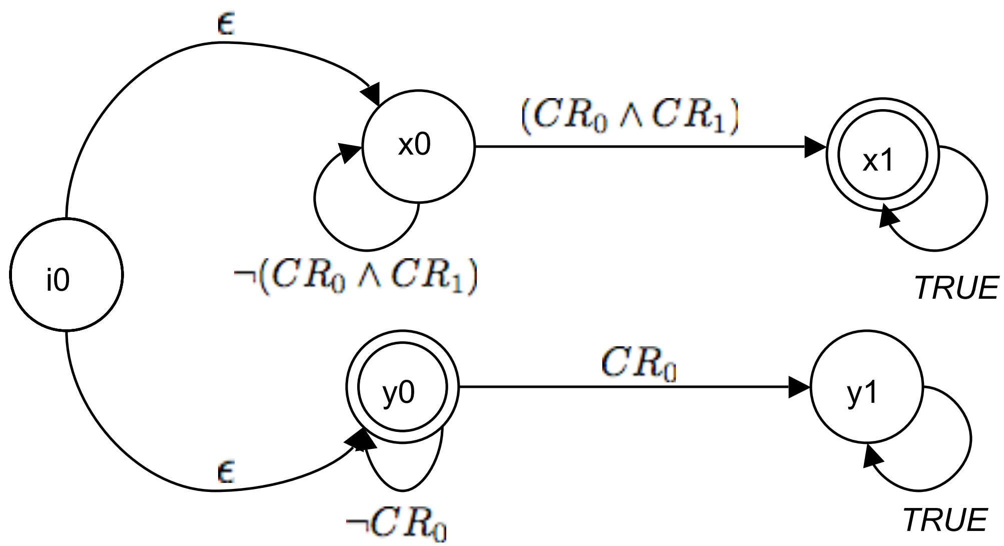
$$\neg S \equiv \neg(\Diamond CR_0 \wedge \Box \neg (CR_0 \wedge CR_1))$$
$$\Longleftrightarrow \quad \Box \neg CR_0 \vee \Diamond (CR_0 \wedge CR_1))$$

Taking the complement



Union of Complement Specification

Goal: Construct an automaton that recognizes the union of languages of both automata.



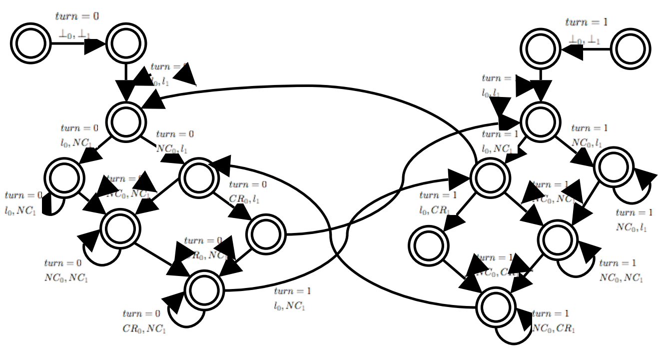
$$\overline{L(S)} = \{\ \epsilon(\neg CR_0 \wedge CR_1)^*(CR_0 \wedge CR_1)(\omega)^*,\ \epsilon(\neg CR_0)^*\ \}$$

Intersection Of Automata

$$\boxed{L(A) \cap \overline{L(S)}}$$

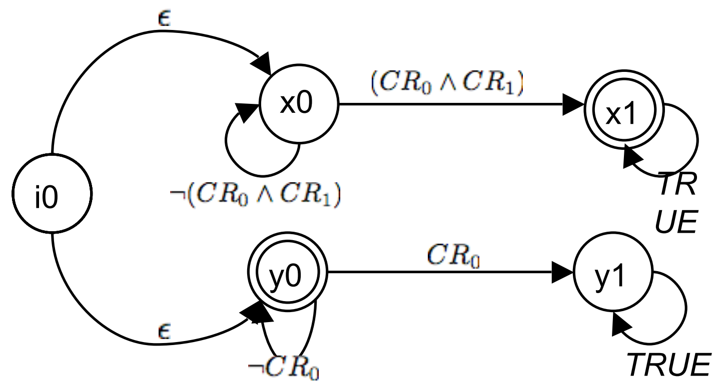
Goal: Construct an automaton that recognizes the intersection of languages of both automata.

System Model A



$\cap$

Complement of Specification, $\overline{S}$

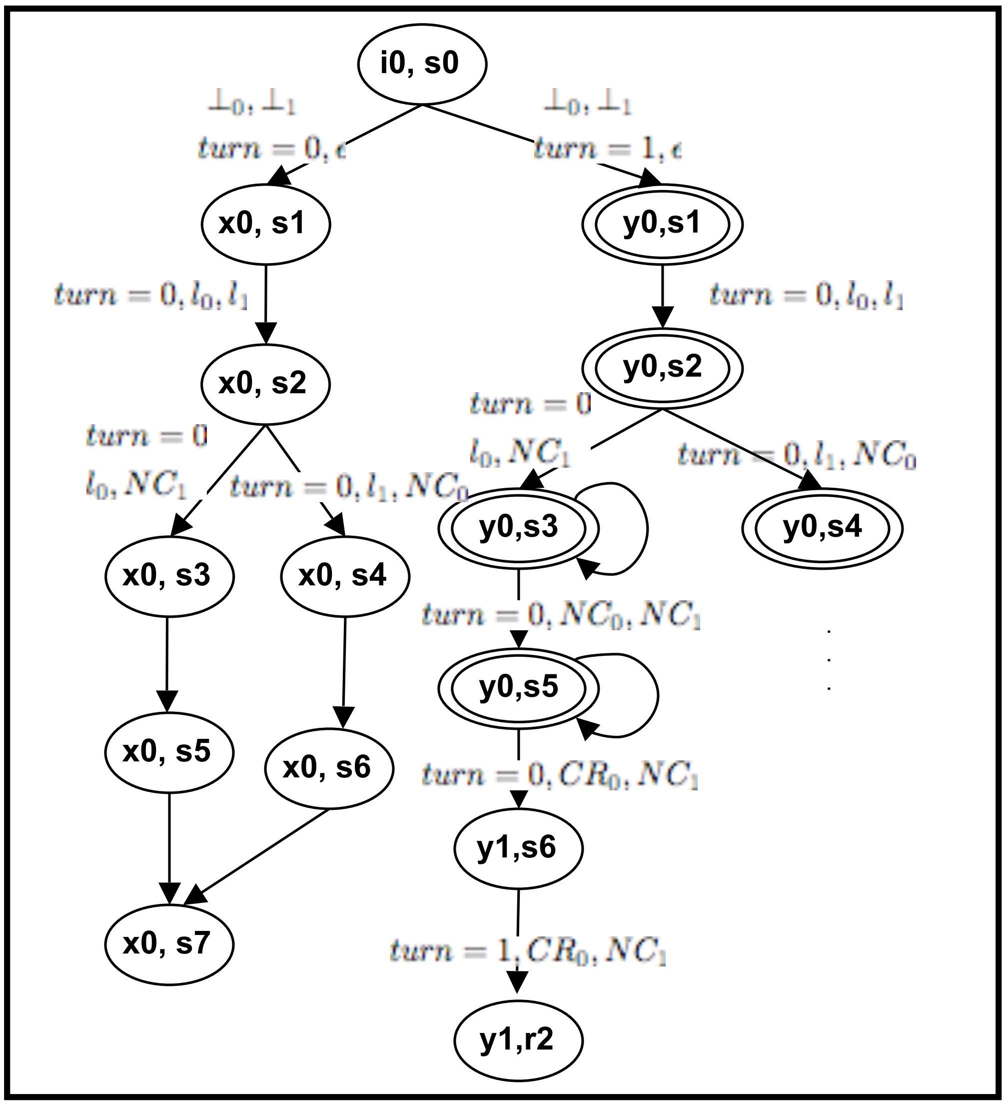


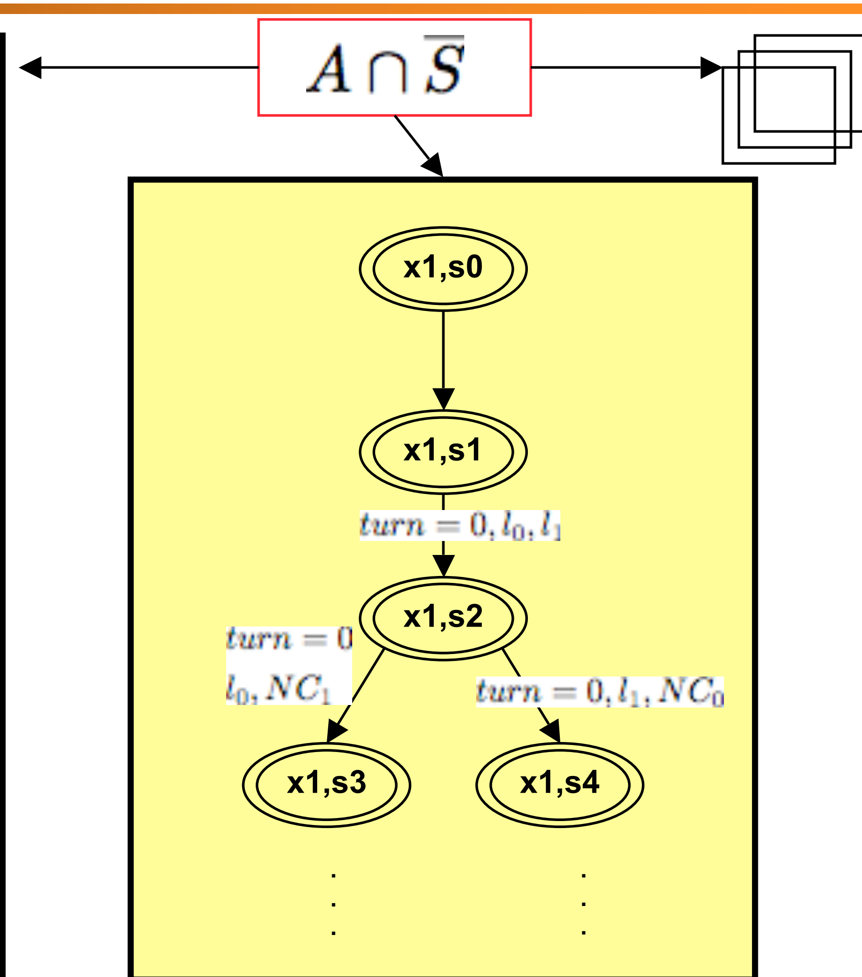
$$A \cap \overline{S} = \{\Sigma, Q_A \times Q_{\overline{S}}, \Delta', Q_A^0 \times Q_{\overline{S}}^0, S_A \times F_{\overline{S}}\}$$

where Q_A are the states of the model A and $Q_{\overline{S}}$ are the states of the specification $\overline{S}$.

Also, $(< s_i, x_j >, a, < s_m, x_n >) \in \Delta' \iff (s_i, a, s_m,) \in \Delta_A$ and $(x_j, a, x_n) \in \Delta_{\overline{S}}$

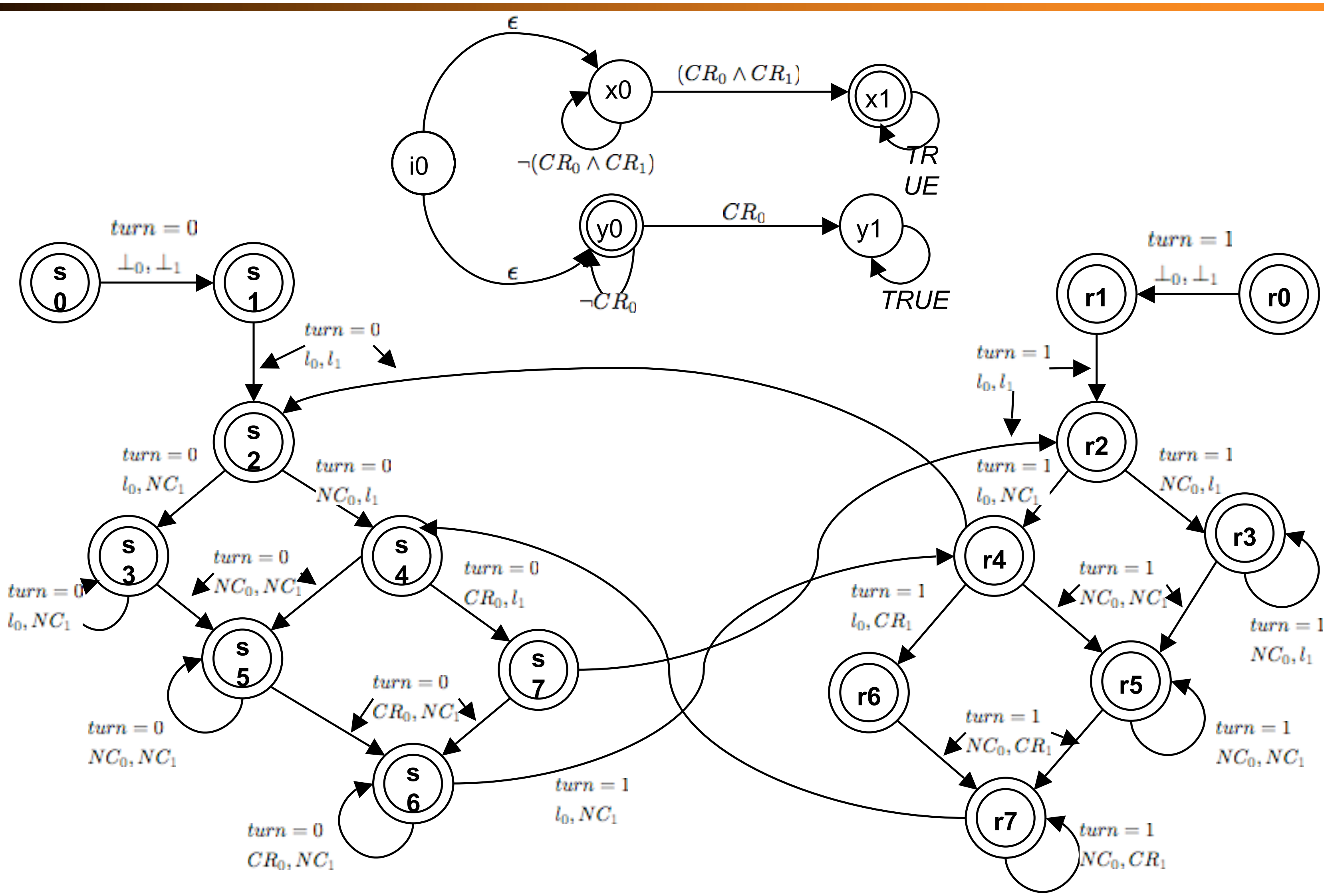
Partial Representations of Intersection Automaton



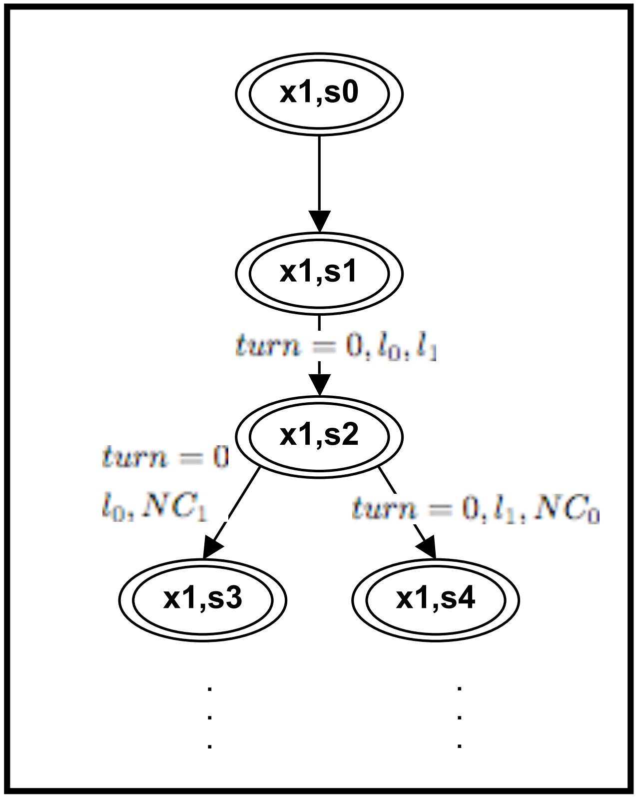


No initial conditions to sub-automaton

Mutex Büchi Model/Spec



Verification of Mutual Exclusion

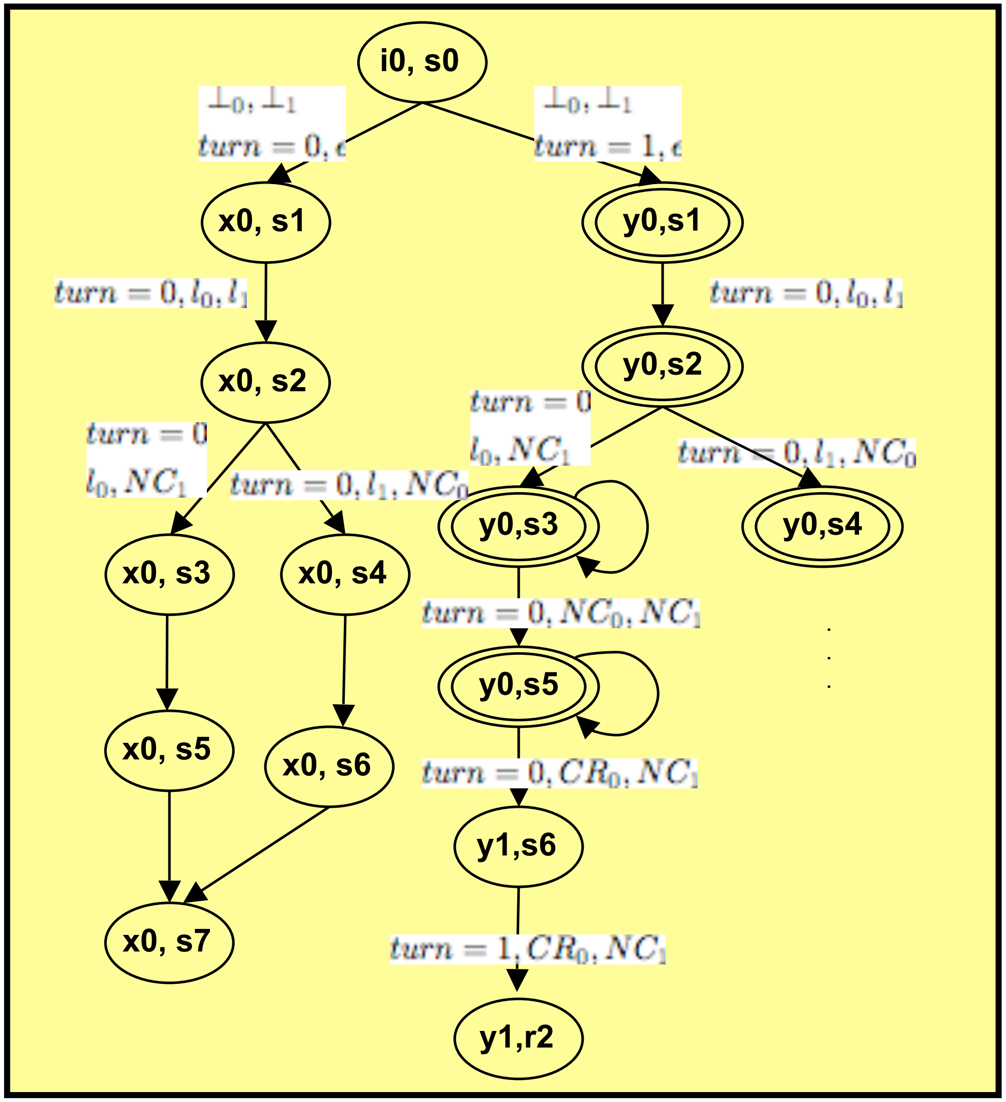


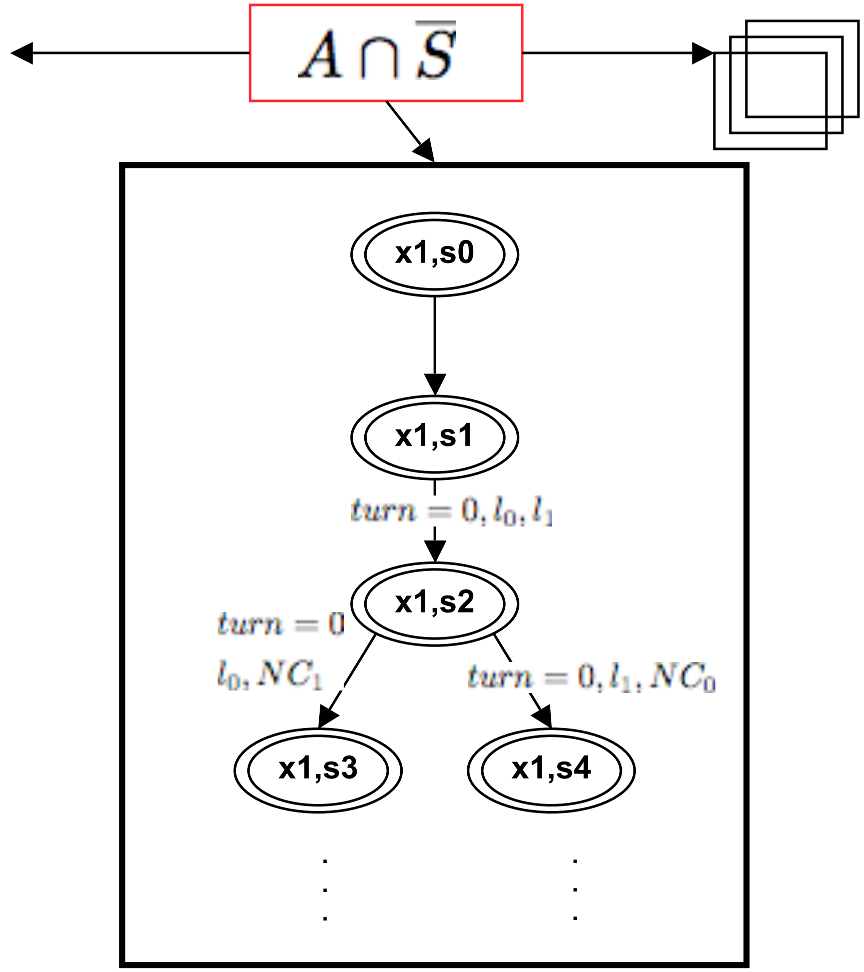
- *Any state of the form (x1,sk) is not reachable from any initial state*
- *The transition to x1 implies both critical regions have been entered simultaneously*

The system satisfies the *mutual exclusion* property.

Checking non-emptiness of Büchi automaton B is equivalent to finding a strongly connected component that is reachable from an initial state and contains an accepting state.

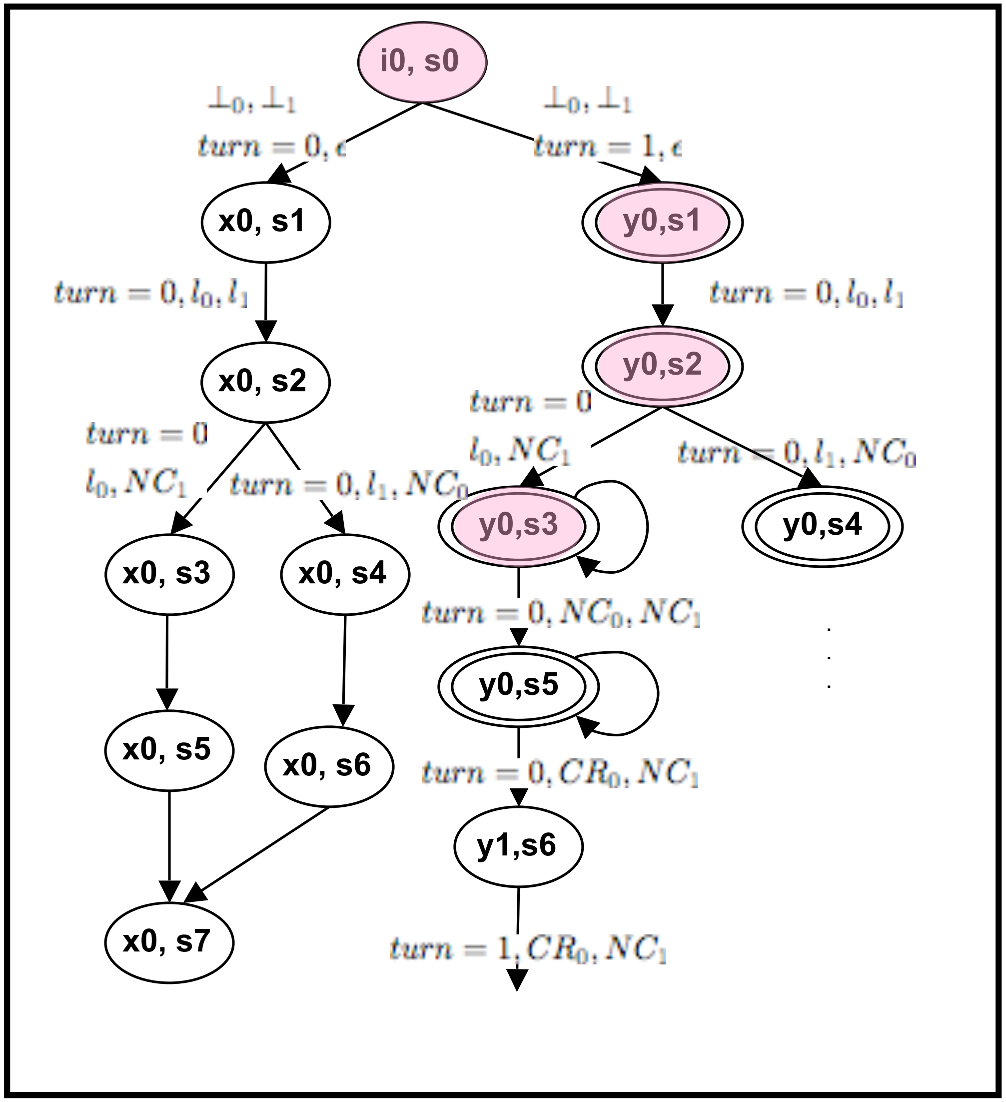
Partial Representations of Intersection Automaton





No initial conditions to sub-automaton

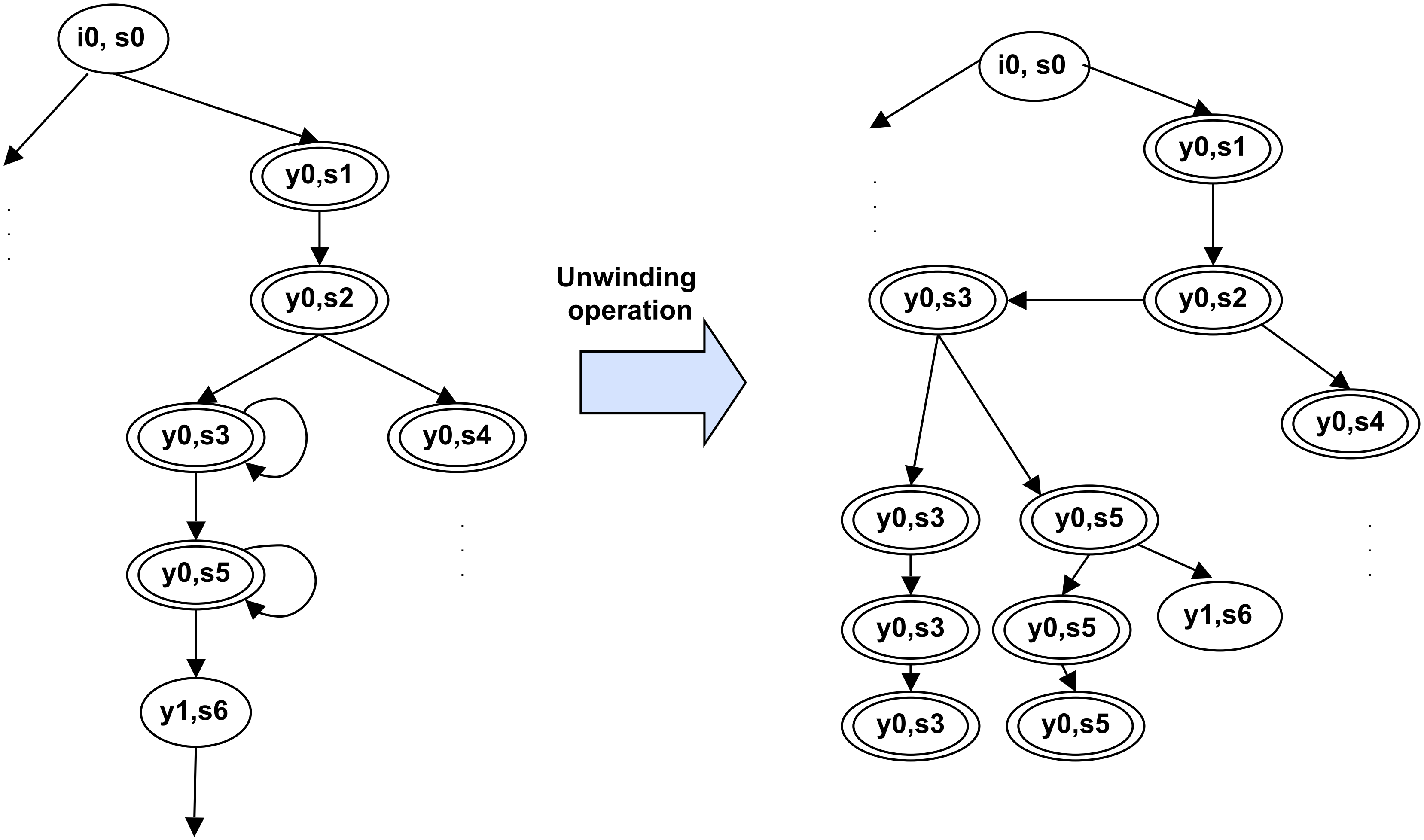
Verification of Liveness Property



- *Tarjan's DFS algorithm for finding strongly connected components*
  - *O(num states + num transitions)*
  - *Double DFS*
- *Found accepting run that has an accepting state with cycle back to itself*
- *Counterexample found*
  - *<i0,s0><y0,s1><y0,s2><y0,s3>**

The system does not satisfy the *absence of starvation* property.

Note on Automaton Unwinding



Automaton ***Computational Path***

Next Time

- State space reduction
  - Abstractions
  - Partial Order Reduction
  - Compositional Reasoning
  - Symbolic Model Checking

References

- Clarke, E.M. *Model Checking,* 1999.
- Holzman, G. *The SPIN Model Checker,* 2003.
- Sipser, M., *Introduction to the Theory of Computation,* 2005.