Announcements

Succession of preps, readings, Assignments (check Canvas)

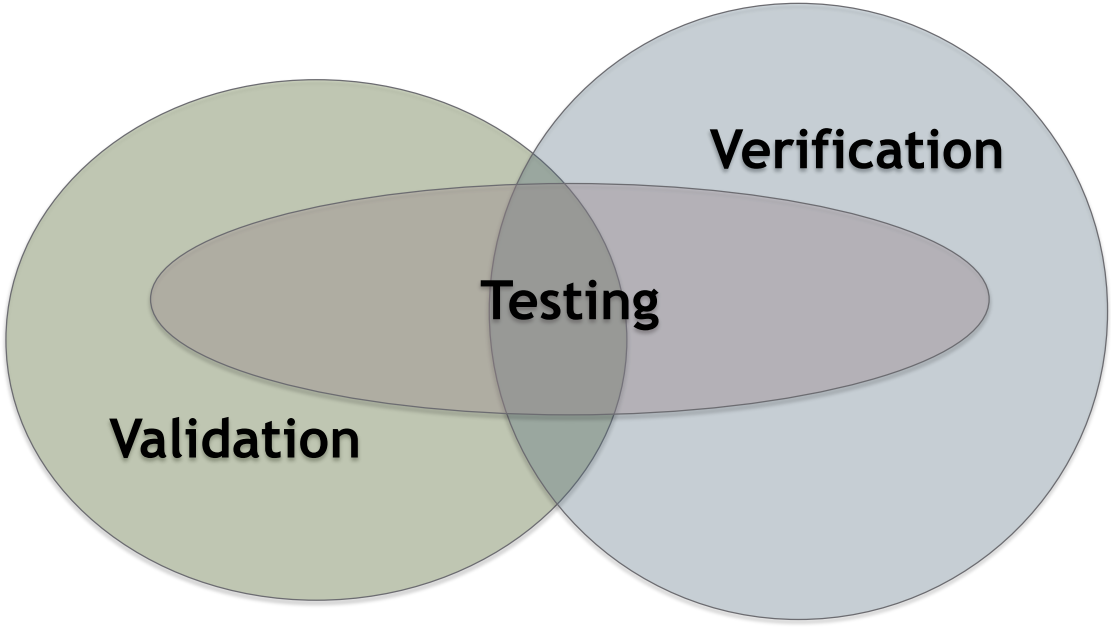
- This Thu: tool installation (Promela/Spin Prep) for class exercises and 2 articles (deadline for report for articles is Fri)
- Tue 11/14: 2 more articles, class exercises
- Thu 11/16: More lass exercises
- L4: Warmup for Promela/Spin Assignment (take-home): due date on Canvas
- A2: Promela/Spin Assignment (last assignment): due date on Canvas

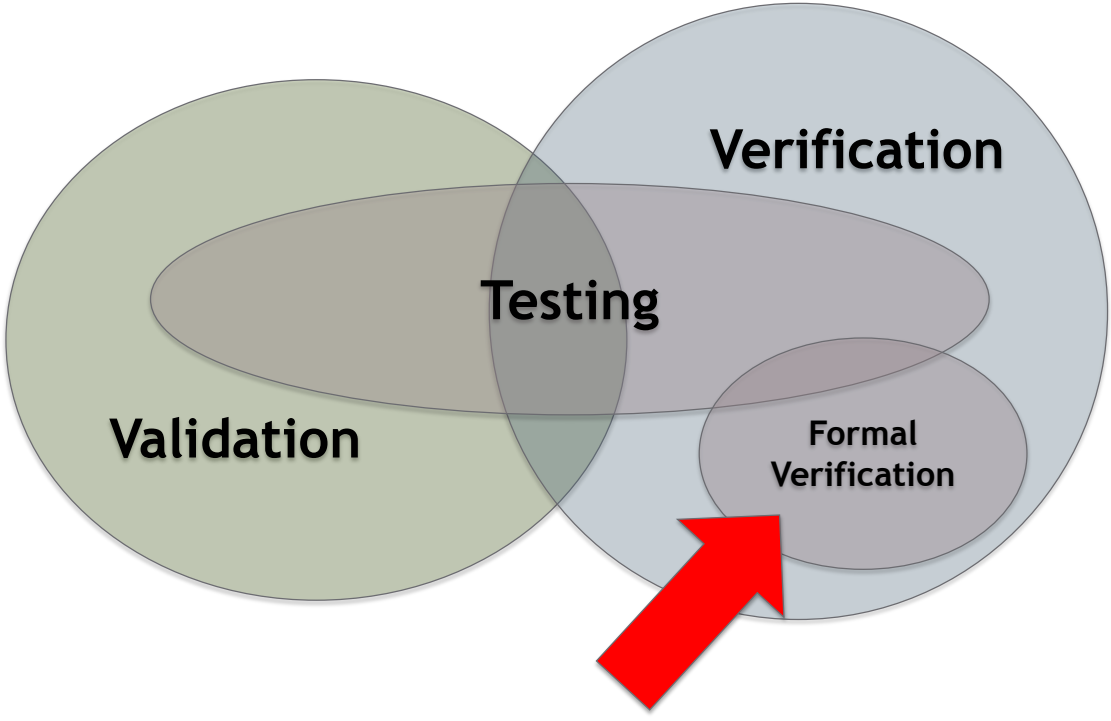
- Final Exam: Tue Dec 12, 2:30pm PT (room TBD)

formal = *opposite of a*d-hoc

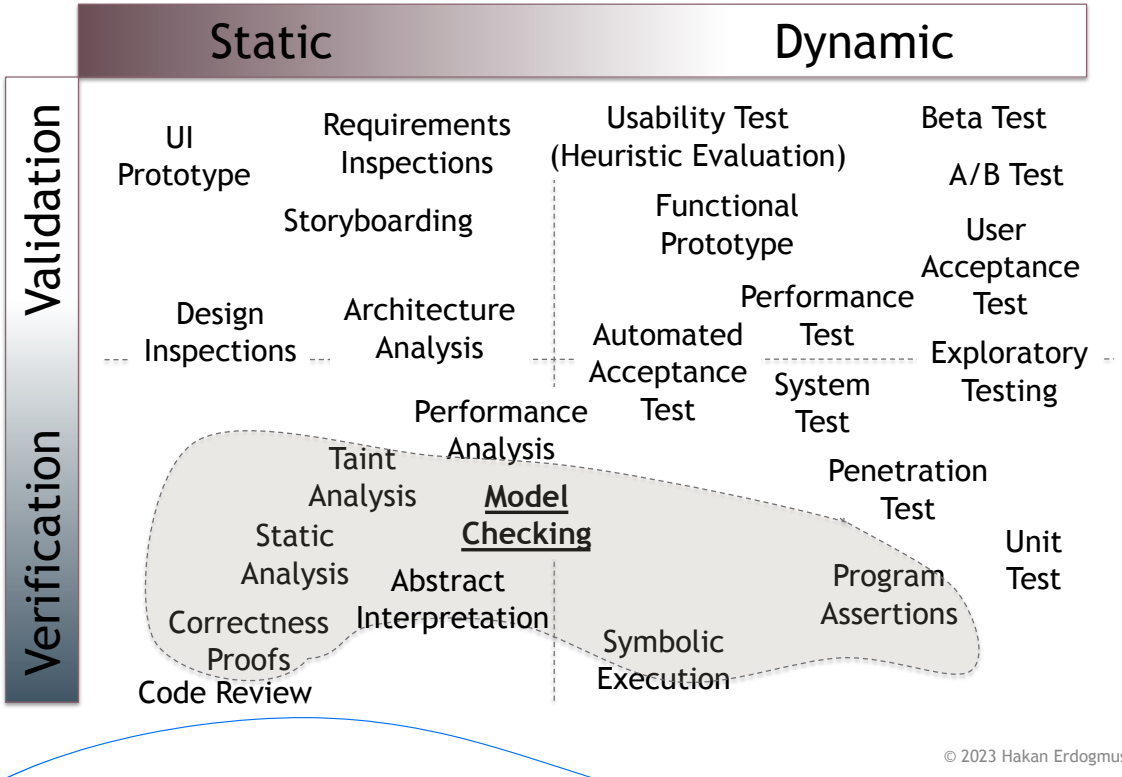
FORMAL VERIFICATION

rigorous, algorithmic, logic-, and mathematically-based verification techniques





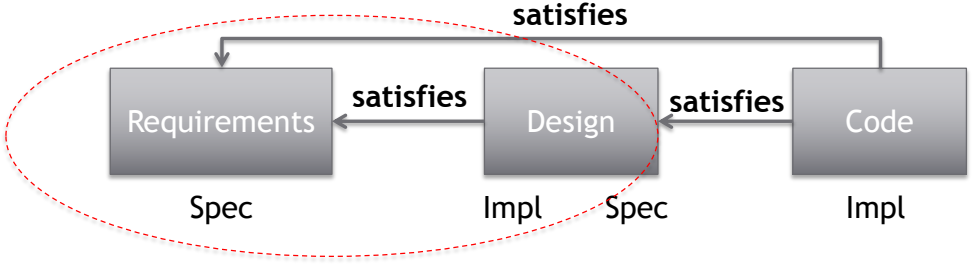
V&V Landscape - *in this course*



Formally: Verification is checking consistency of an *implementation* with respect to a *specification*

Impl **satisfies** *Spec*

*Implementation and specification are **roles**, not fixed artifacts*



Verification: checking consistency of a *model* with respect to a set of correctness *properties*

Impl Spec

Model **satisfies** *Property*

$$M \models P$$

Model is an abstraction of the implementation that includes only the details relevant to the property being verified (**approximation principle**)

When: **Spec = Set of Properties = Set of Requirements**

$$M \models \{ P_1, P_2, \ldots, P_n \}$$

Main Principles of V&V still apply here

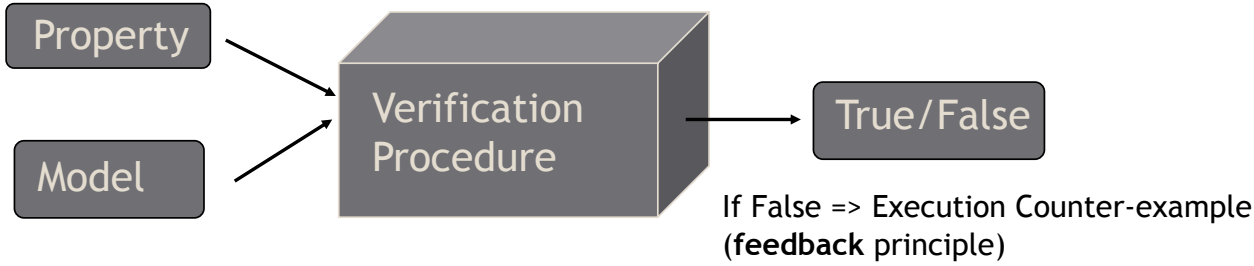
Main Principles of V&V still apply here

- **Partitioning**: divide and conquer
- **Approximation**: making the problem easier
- **Visibility**: making information accessible
- **Feedback**: providing actionable information
- **Repeatability**: better to fail every time than sometimes
- **Redundancy**: triangulating results by using different techniques

Examples of properties

- The system always shuts down when the pressure exceed 100psi
- The system never refuses inputs
- A request is always followed by an acknowledgement if it's accepted
- Every message sent will eventually be received
- The requests will be processed in the order they are sent
- After pressing the call button, the elevator will eventually go to the calling floor unless the cancel button is pressed

YOU CAN'T ~~ALWAYS~~ *ever* GET WHAT YOU WANT



PROVING CORRECTNESS PROPERTIES IS UNDECIDABLE IN ITS FULL GENERALITY…

Restrict to subset of ***relevant and useful properties*** *on* ***finite-state models***

Also important: How realistic/faithful is the model?

Depends on the property

- If all details relevant to the property being verified are included in the model, then realistic/faithful (*sound*)

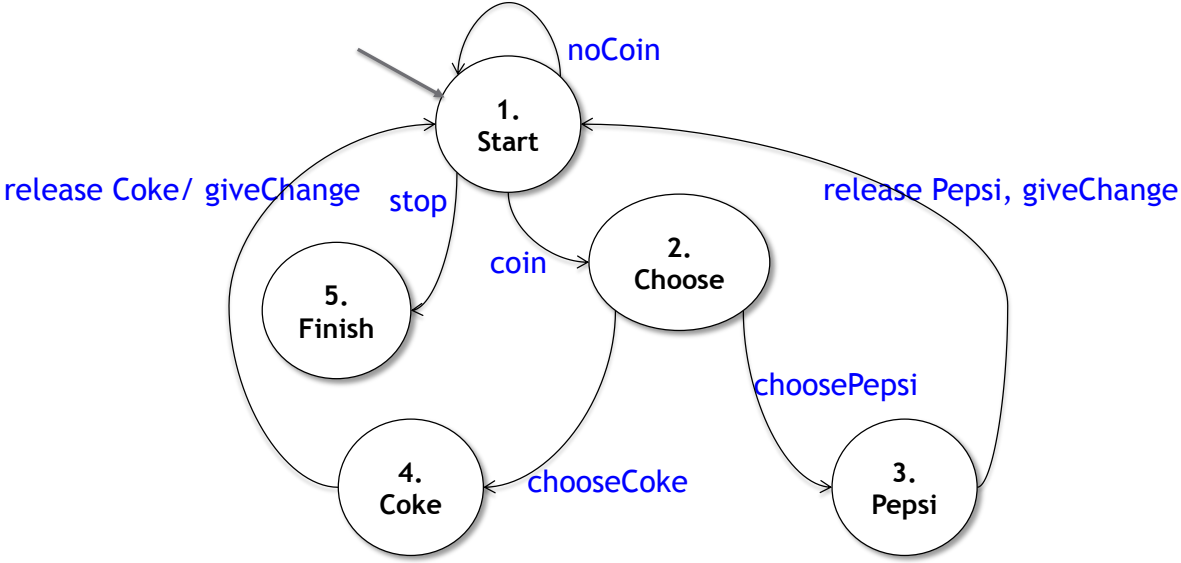
- Unrealistic/unfaithful to the extent of missing or changed detail relevant to the property being verified (*unsound*)

Building models

- What do we need to know to build a model?

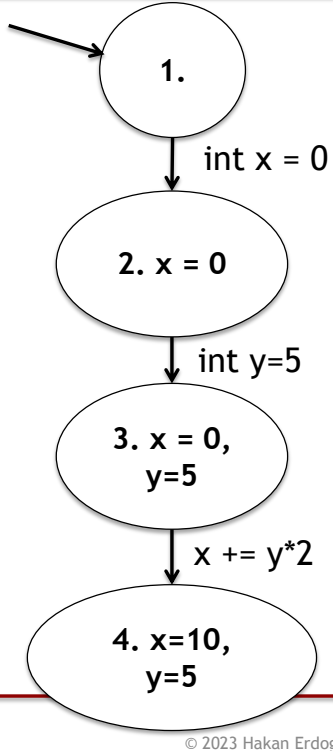
- Consider a simple vending machine
  - A customer inserts coins, selects a beverage and receives a can of soda
  - Basically, a state machine
    - Starts in initial state and changes based on user interaction
- We need to know
  - What *states* the vending machine can be in
  - How the machine *transitions* from one state to another
  - In which state the machine starts

VENDING MACHINE MODEL



Modeling a sample program as a state machine

```
int x = 0;
int y = 5;
x += y*2;
```



Formal verification techniques that use models, like static analysis, can find faults that testing is not good at finding!

Which types of faults testing is good at finding?

- Faults that we can predictably reproduce at execution
  - Often related to **computations and data flow in sequential and deterministic systems**

Which types of faults is testing poor at finding?

Faults that we cannot predictably reproduce at execution

- Often related to complex, unpredictable **control flow**
  - when there is inherent *nondeterminism*
- Common in **concurrent systems**
  - parallel programs
  - distributed systems

This is where formal verification techniques come into play

Concurrent systems are *composed of multiple communicating components*

- each component has own thread of execution
- components communicate or synchronize through different mechanisms
  - message passing
    - events
    - remote procedure call (RPC)
  - request-response protocol (HTTP)
  - shared data
  - *rendez-vous (handshake)*
- no guarantees with respect to relative speed of execution

Examples of concurrent systems

Examples of concurrent software systems

- Multi-threaded programs
- Actor systems / Agent systems
- Communicating processes
- Network applications
- Operating systems
- Embedded systems
- Sensor networks / IoT systems
- Distributed web applications
- Client-server applications
- Grid applications
- Communication protocols

…

Most software is concurrent

… at some sufficiently low-enough or sufficiently high-enough level of abstraction

Whether we treat (model) the system under verification as *concurrent* or *sequential* depends on the properties of interest…

What are some common concurrency-related faults?

Common concurrency-related faults

- Deadlocks
  - *Inability to make progress*
- Race conditions
  - *Untimely or out-of-order access to shared resources*
- Livelocks
  - *Unproductive progress ("spinning its wheels")*
- Starvation
  - *Unfair scheduling, unfair access to resources*

Think-Pair-Share: Find familiar examples of each category

- Deadlocks
  - *Inability to make progress*
- Race conditions
  - *Untimely or out-of-order access to shared resources*
- Livelocks
  - *Unproductive execution*
- Starvation
  - *Unfair scheduling, unfair access to resources*

Testing cannot easily catch concurrency faults due to inability to control OS, communication, concurrency infrastructure

- *relative execution speeds of underlying components/threads/processes*
- *opaque scheduling schemes of execution environments*
- *some nondeterminism being totally invisible*

We cannot easily reproduce concurrency faults during testing!

relevant

- We need to consider *all possible behaviors*, independent of any invisible nondeterminism!
- **Formal verification can do this, but we will need to model the system first!**

Motivating example: mutual exclusion

Two threads that need a resource `r` are trying to control access to `r`.
Shared variables `x, y` are used to to put a *lock* on `r`.
Another shared variable `stop` controls termination.

Thread X

```
stop = false;
while (!stop) {
  x = 1;                    // reserve
  while (y == 1) {}         // busy wait
  // can access r
  dSW(r);                   // access (critical section)
  x = 0;                    // release
}
```

Thread Y

```
stop = false;
while (!stop) {
  y = 1;
  while (x == 1) {}
  // can access r
  dSW(r);
  y = 0;
}
```

How can you write a test that makes sure only one thread accesses `r` at a time? (**mutex property**)
And that at least one thread is able to access it? (**progress property**)

Motivating example: how do we formulate these properties?

Introduce a shared *observation* variable mutex (init to 0) that track how many threads are in their critical section

Thread X

```
stop = false;
while (!stop) {
  x = 1;
  while (y == 1) {}
  mutex++;
  dSW(r);
  mutex--;
  x = 0;
}
```

Thread Y

```
stop = false;
while (!stop) {
  y = 1;
  while (x == 1) {}
  mutex++;
  dSW(r);
  mutex--;
  y = 0;
}
```

- **Mutex property**: value of `mutex` should never be greater than 1
- **Progress property**: one thread can enter its critical section; *i.e.*, `mutex` will eventually become 1

What now?

Let's try to do this with testing

Introduce a shared history array `h[]` and shared count `c` (init to 0). `h[c]` stores a snapshot of variable mutex. In `dSW(r)`, store value of mutex in `h[c]` and increment `c`.

Thread X

```
stop = false;
while (!stop) {
  x = 1;
  while (y == 1) {}
  mutex++;
  dSW(r);
  mutex--;
  x = 0;
}
```

`h[c++] = mutex;`

Thread Y

```
stop = false;
while (!stop) {
  y = 1;
  while (x == 1) {}
  mutex++;
  dSW(r);
  mutex--;
  y = 0;
}
```

Test:

```
threadX.start(); threadY.start(); Thread.sleep(50);
stop = true; end = c; progress = false;
threadX.join(200); threadY.join(200);
for (int i = 0; i < end; i++) {
    assertTrue(h[i] <= 1) ; // mutex
    if (h[i] > 0) progress = true;
}
assertTrue(progress); // progress
```

Run this test 10 times. It always passes. Your strategy is working. You're done!

Wait: something is wrong!

Change nothing. Come back next day. Re-execute the test. Now it suddenly fails? Test reveals that `c == 0` at the end. What the heck?

Thread X

```
stop = false;
while (!stop) {
  x = 1;
  while (y == 1) {}
  mutex++;
  dSW(r);
  mutex--;
  x = 0;
}
```

Thread Y

```
stop = false;
while (!stop) {
  y = 1;
  while (x == 1) {}
  mutex++;
  dSW(r);
  mutex--;
  y = 0;
}
```

Test:

```
threadX.start(); threadY.start(); Thread.sleep(50);
stop = true; end = c; progress = false;
threadX.join(200); threadY.join(200);
for (int i = 0; i < end; i++) { // loop body never executed
    assertTrue(h[i] <= 1) ; // mutex
    if (h[i] > 0) progress = true;
}
assertTrue(progress); // progress
```

Come back tomorrow. Run the test. Now it fails! Why?

Wait: something is wrong!

Change nothing. Come back next day. Re-execute the test. Now it suddenly fails? Test reveals that `c == 0` at the end. What the heck?

Thread X

```
stop = false;
while (!stop) {
  x = 1;
  while (y == 1) {}
  mutex++;
  dSW(r);
  mutex--;
  x = 0;
}
```

Thread Y

```
stop = false;
while (!stop) {
  y = 1;
  while (x == 1) {}
  mutex++;
  dSW(r);
  mutex--;
  y = 0;
}
```

Test:

```
threadX.start(); threadY.start(); Thread.sleep(50);
stop = true; end = c; progress = false;
threadX.join(200); threadY.join(200);
for (int i = 0; i < end; i++) { // loop body never executed
    assertTrue(h[i] <= 1) ; // mutex
    if (h[i] > 0) progress = true;
}
assertTrue(progress); // progress
```

Come back tomorrow. Run the test. Now it fails! Why?

Java Mutex Example – Generalized Version

For a generalized version of the previous example with N clients/threads and observe its unpredictability, try:

https://github.com/cmusv-svvt/Mutex-Java

Let's try a different approach

- Model the system as communicating finite state automata
- Search the global state space
- See if there are any undesirable states:
  - that violate the *mutex* property
  - that violate the *progress* property

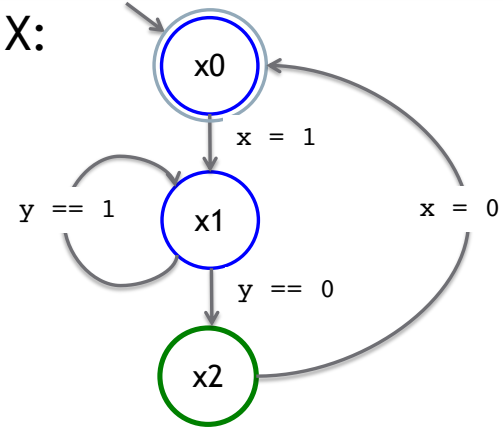
Thread X

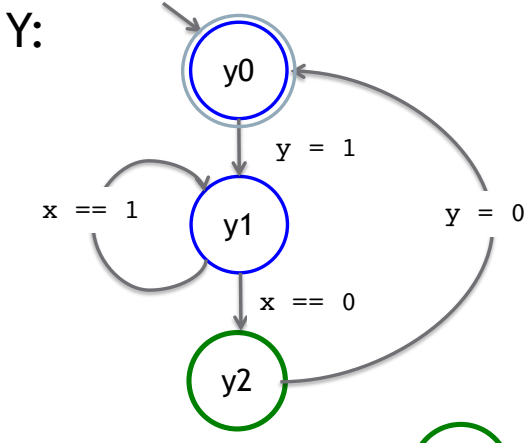
```
stop = false;
while (!stop) {
    x = 1;
    while (y == 1) {}
    mutex++;
    dSW(r);
    mutex--;
    x = 0;
}
```

Thread Y

```
stop = false;
while (!stop) {
    y = 1;
    while (x == 1) {}
    mutex++;
    dSW(r);
    mutex--;
    y = 0;
}
```

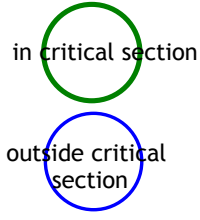
Finite State Automata for X and Y



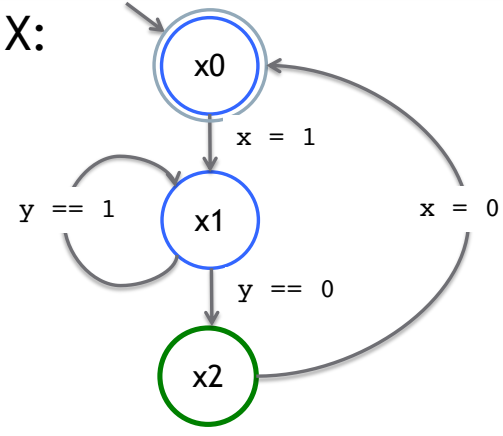


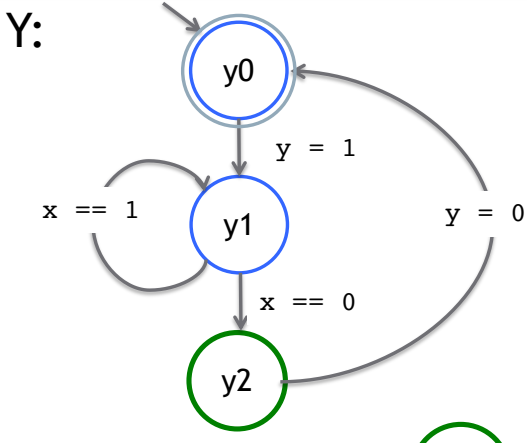

```
while (true) {
  x = 1;
  while (y == 1) {}
  dSW(r);
  x = 0;
}
```

```
while (true) {
  y = 1;
  while (x == 1) {}
  dSW(r);
  y = 0;
}
```



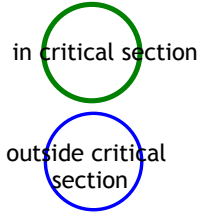
Finite State Automata for X and Y






```
while (true) {
  x = 1;
  while (y == 1) {}
  x = 0;
}
```

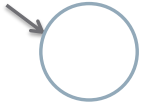
```
while (true) {
  y = 1;
  while (x == 1) {}
  y = 0;
}
```



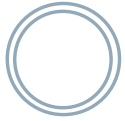
What happened to `stop`, `mutex++`, `mutex--`, `dSW(r)`?

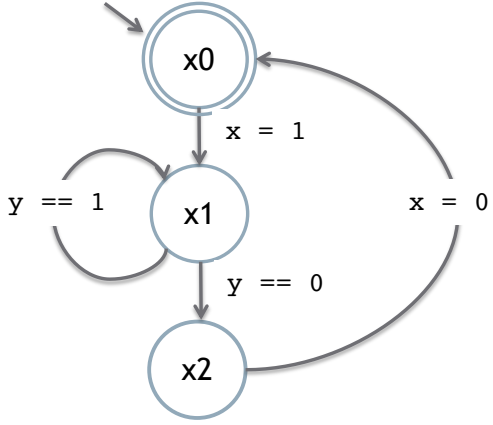
Finite State Automata

- States and transitions (both finite)
- Start state



- A set of end states



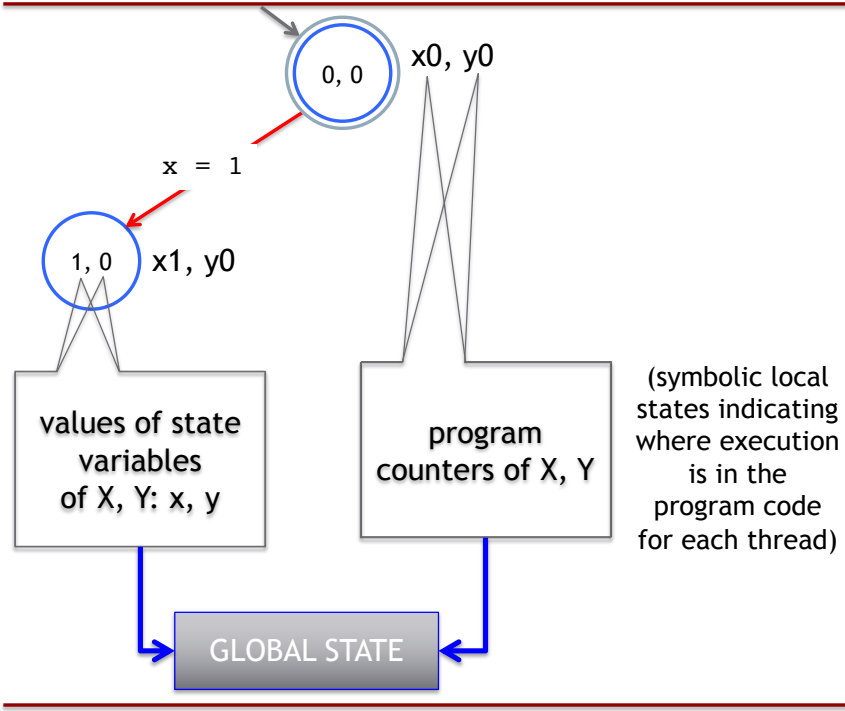


- Transitions are labeled
  - *condition*: enabled if evaluates to true
  - *action (command)* (e.g., assignment): always enabled

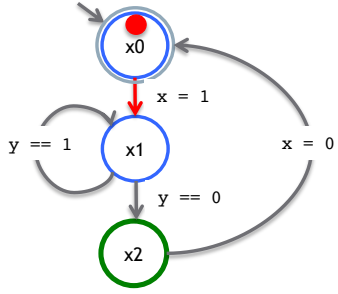
The product FSA: X ⊗ Y

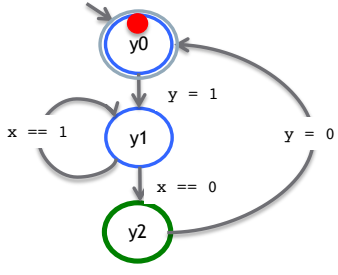
- We want to search the combined (global) state space
- One FSA can execute an enabled transition at a time (emulating a single-core CPU multiplexing different threads)
- In case of multiple enabled transitions, one is randomly chosen
- We can do a depth-first or breadth-first search to visit all global states
- When we visit an already visited global state, we backtrack and look for another enabled transition to take

Creating X ⊗ Y (1)



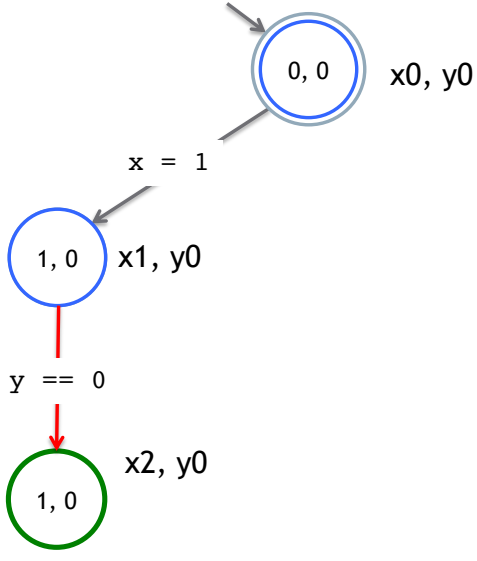
(symbolic local states indicating where execution is in the program code for each thread)

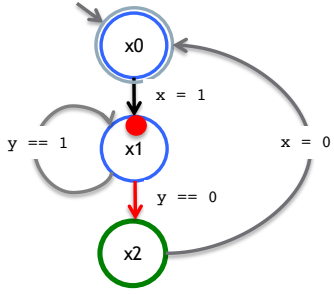


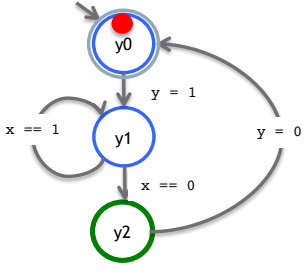


every combination is a distinct global state

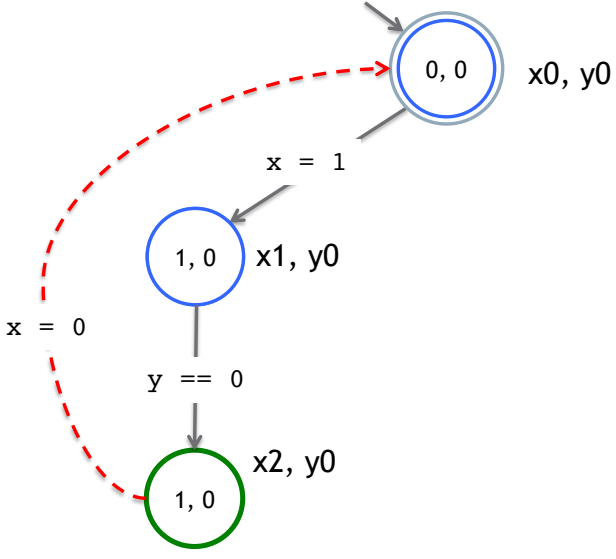
Creating X ⊗ Y (2)

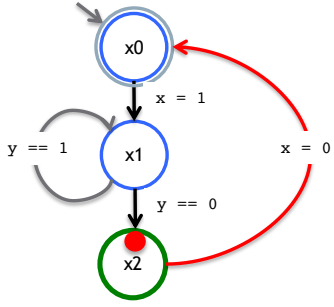


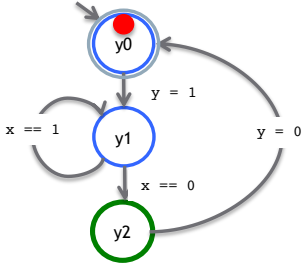




Creating X ⊗ Y (3)

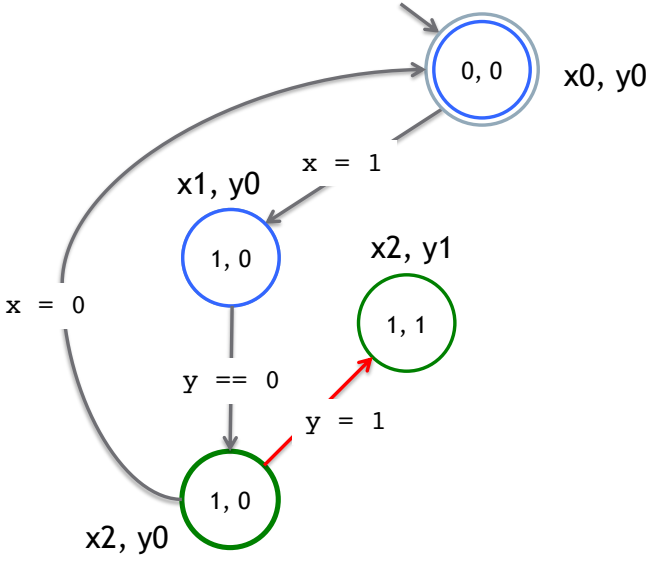


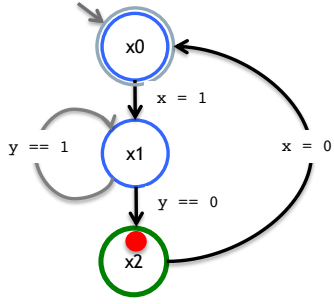


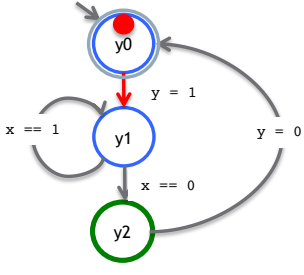


Hit previously visited state: stop and backtrack to x2, y0

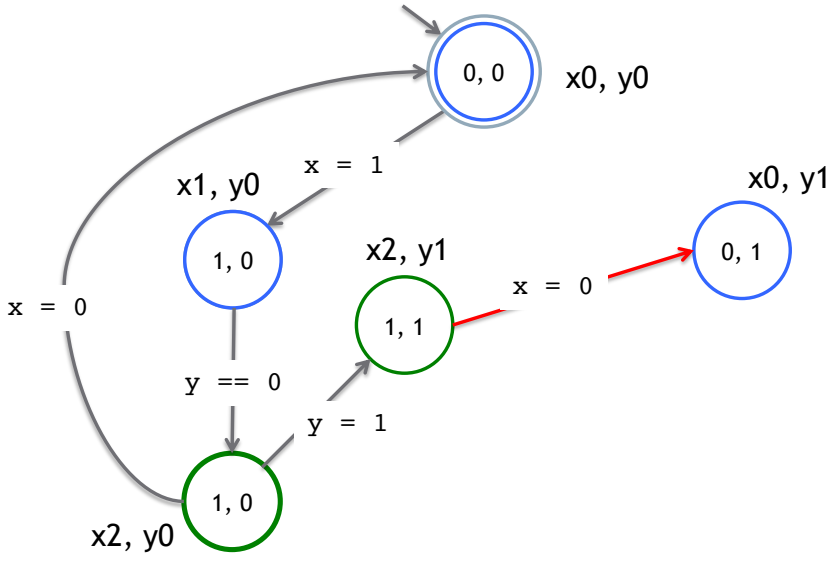
Creating X ⊗ Y (4)

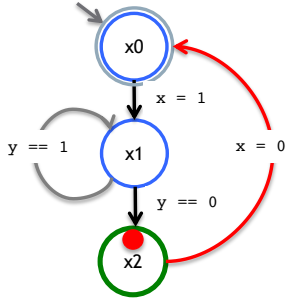


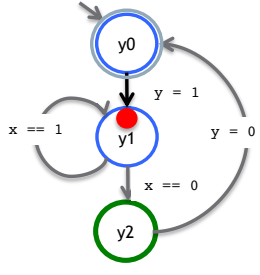




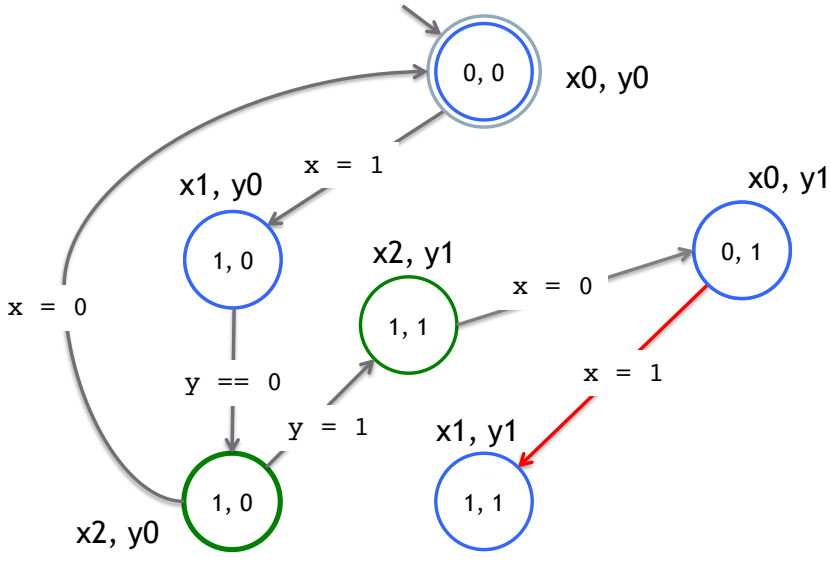
Creating X ⊗ Y (5)

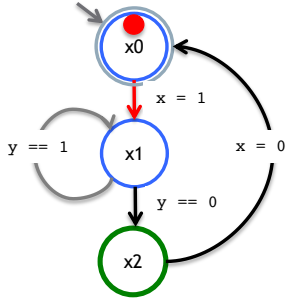


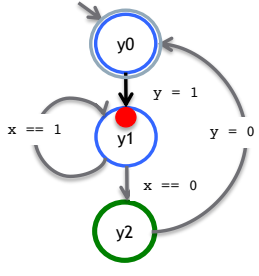




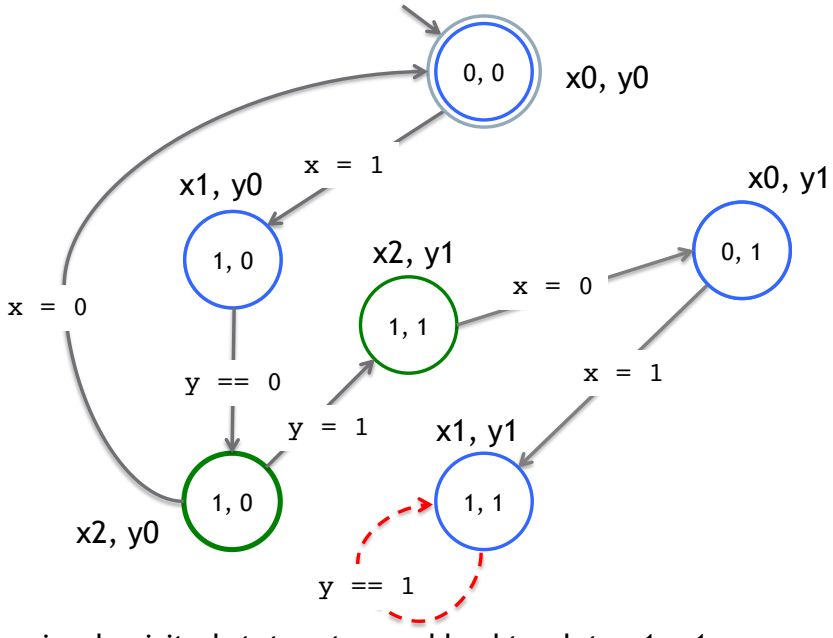
Creating X ⊗ Y (6)

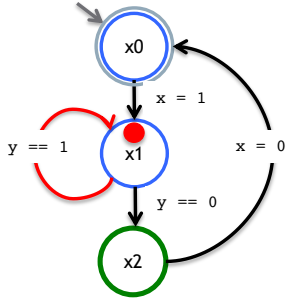


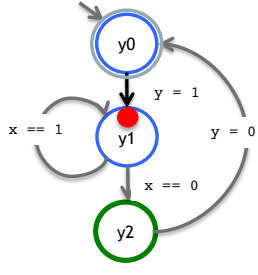




Creating X ⊗ Y (7)

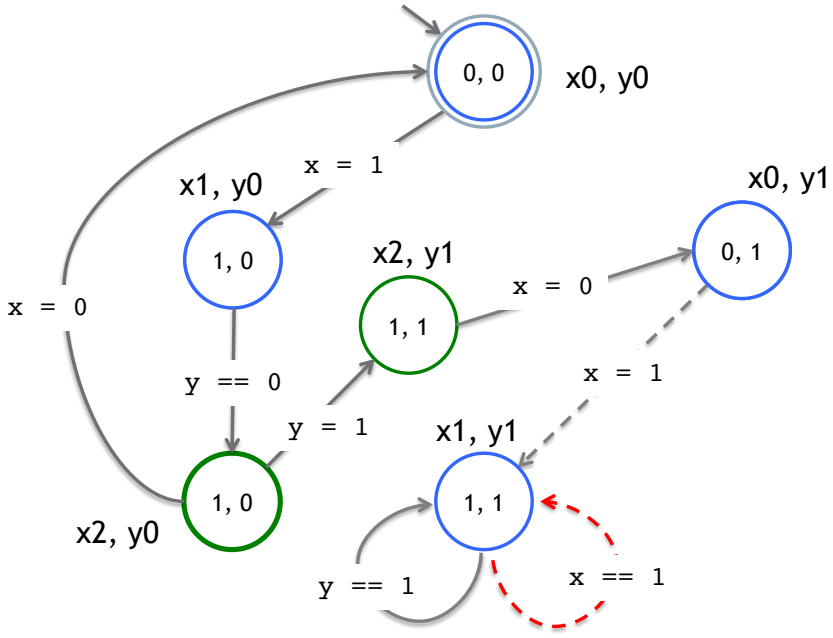


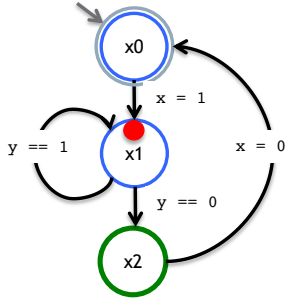


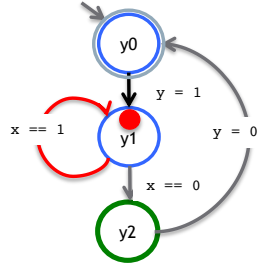


Hit previously visited state: stop and backtrack to x1, y1

Creating X ⊗ Y (8)

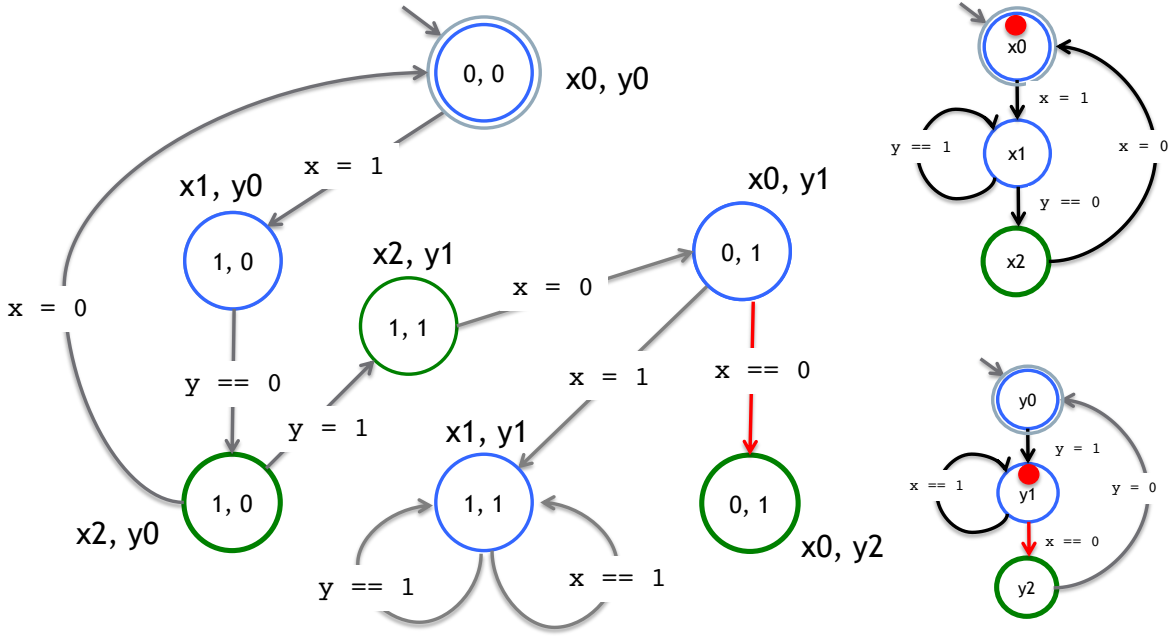




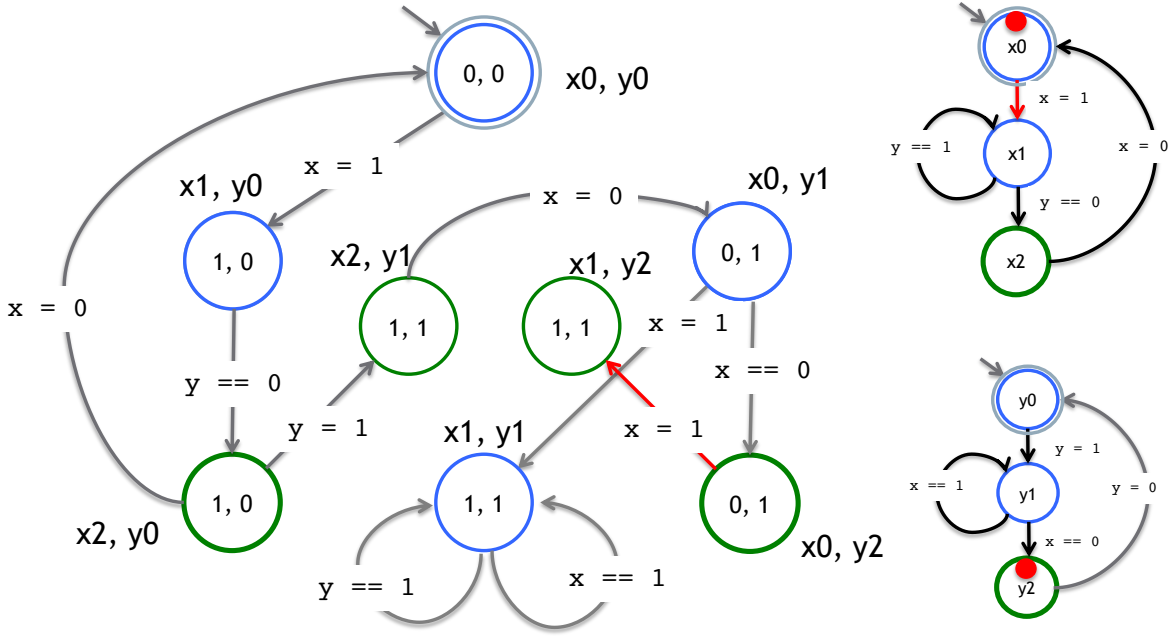


No further transitions to explore at x1, y1: backtrack to x0, y1

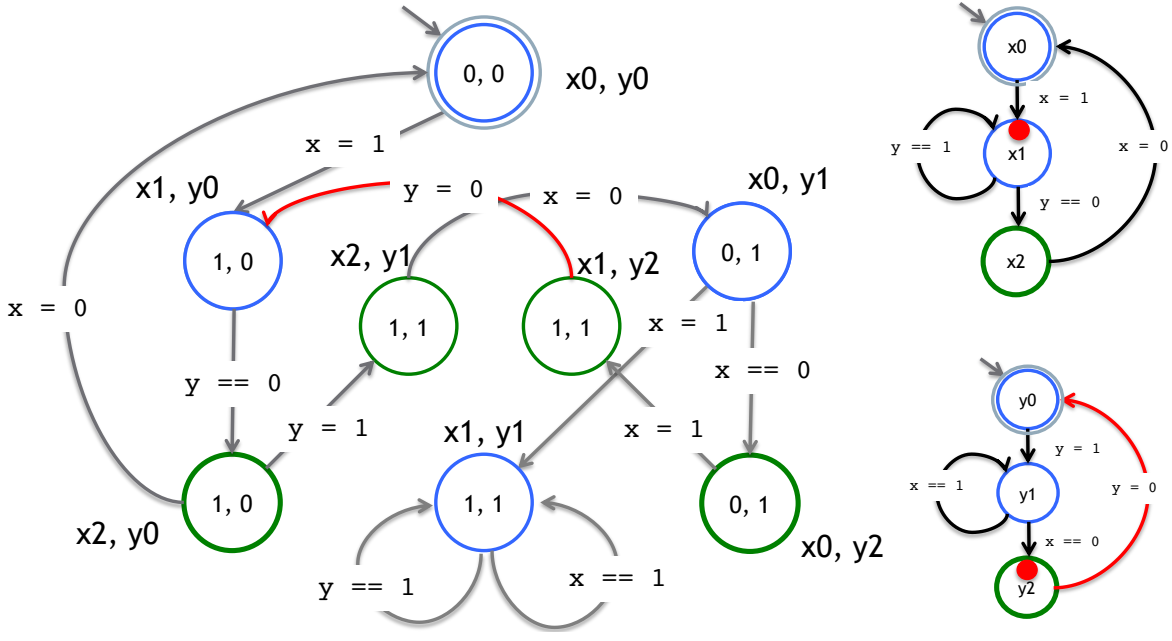
Creating X ⊗ Y (9)



Creating X ⊗ Y (10)

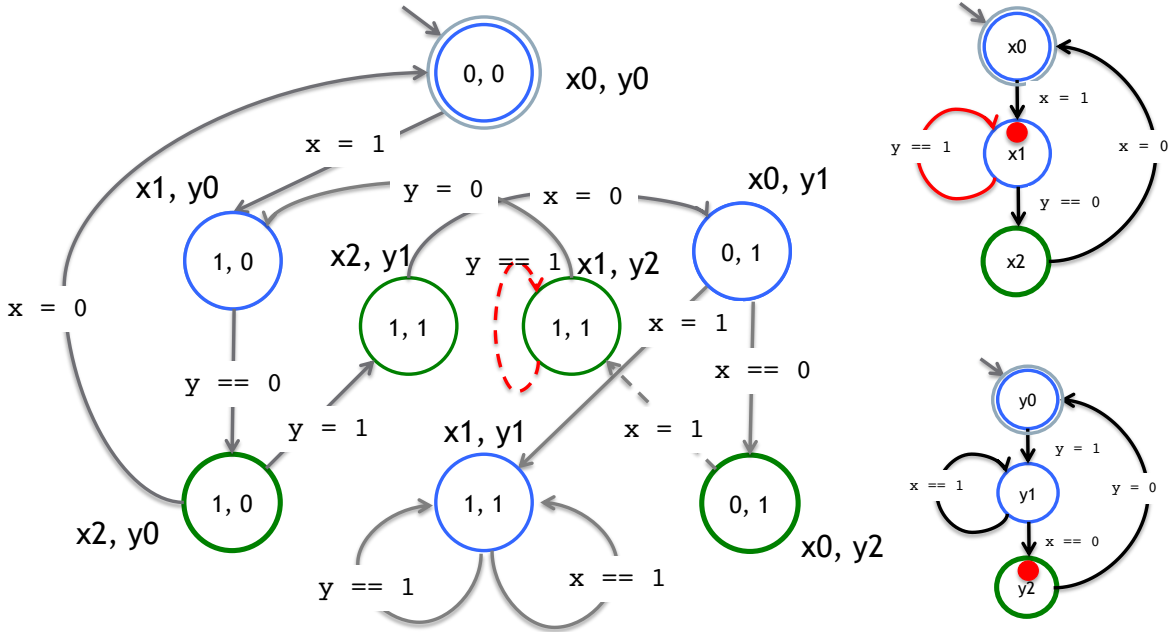


Creating X ⊗ Y (11)



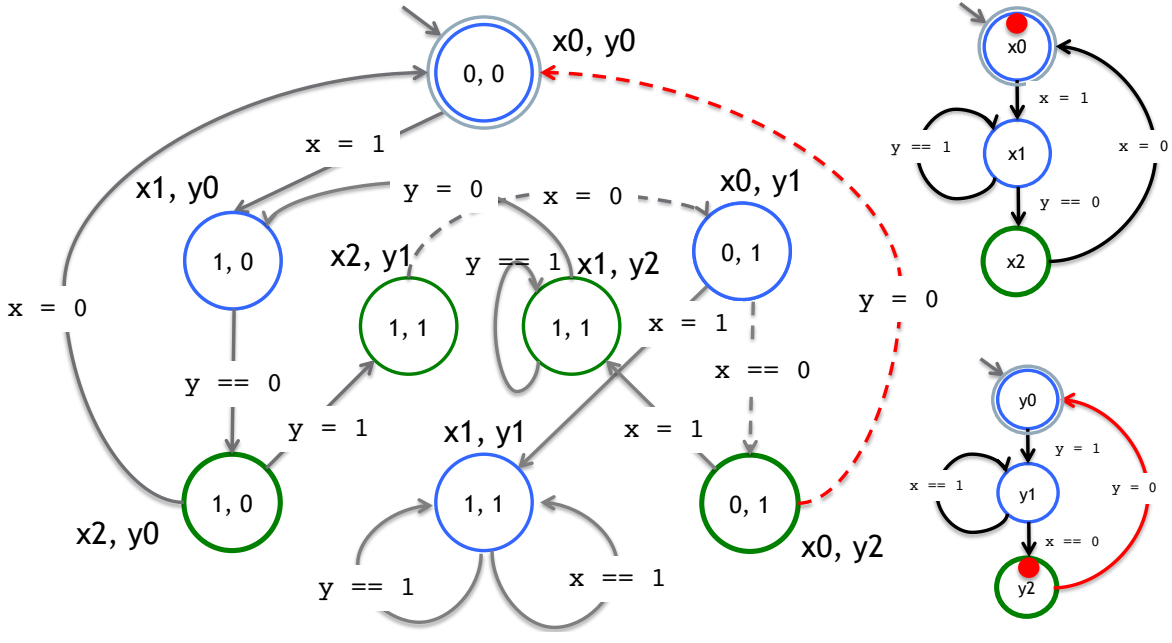
Hit a previously visited state: backtrack to x1, y2

Creating X ⊗ Y (12)



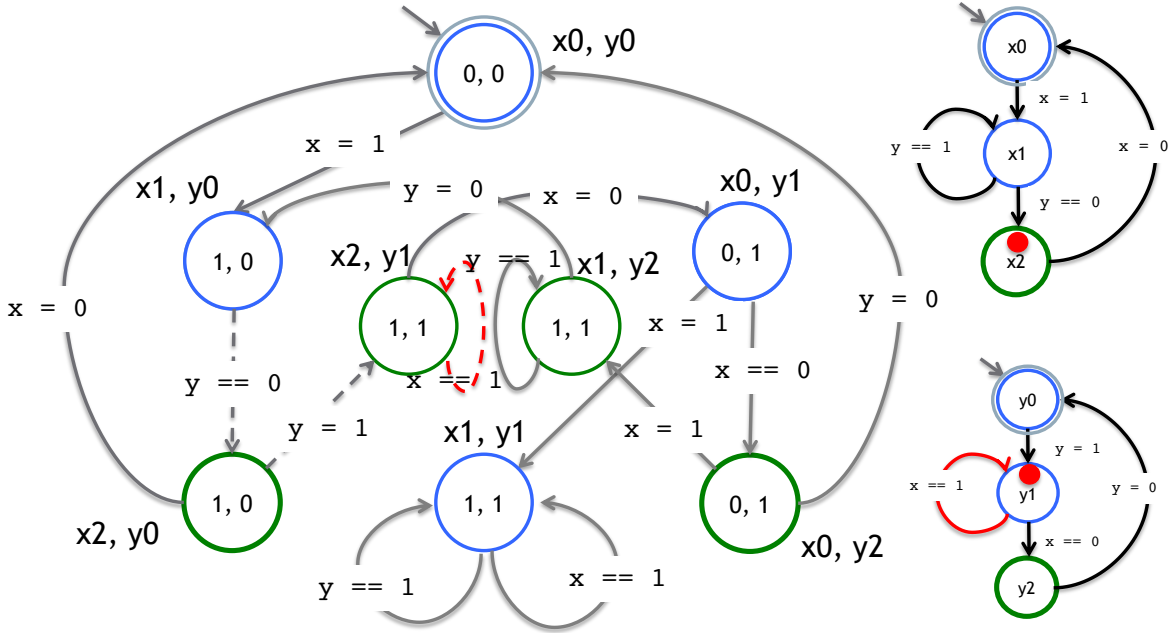
Hit a previously visited state x1, y2: backtrack to x1, y2 and then to x0, y2

Creating X ⊗ Y (13)



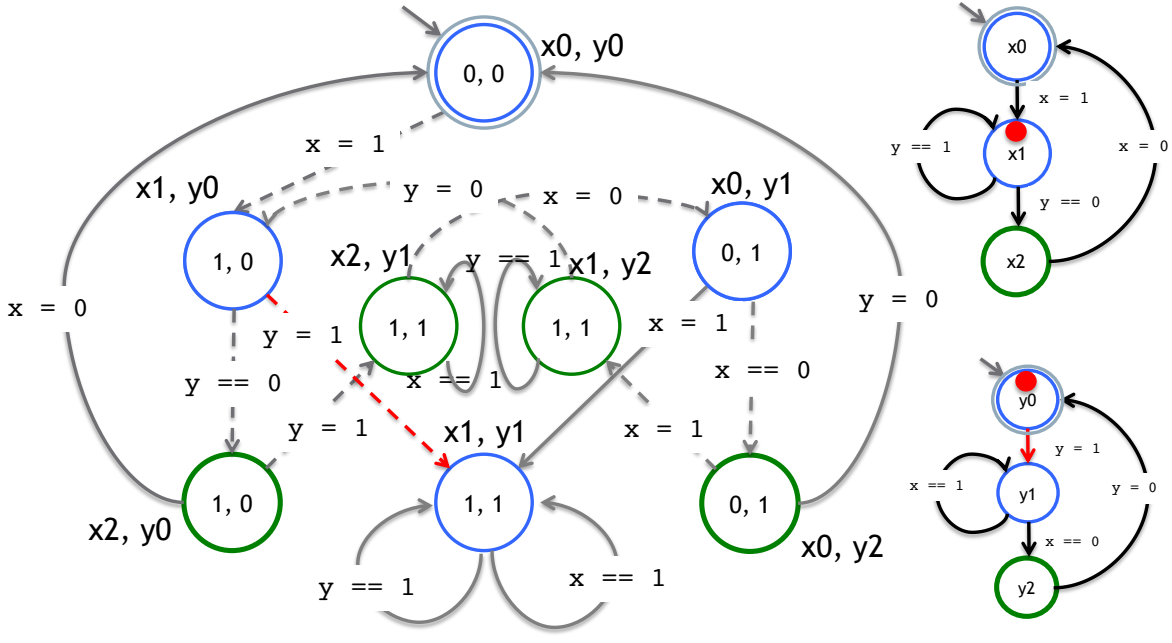
Hit a previously visited state x0, y0: backtrack to x0, y2; then to x0, y1; then x2, y1

Creating X ⊗ Y (14)



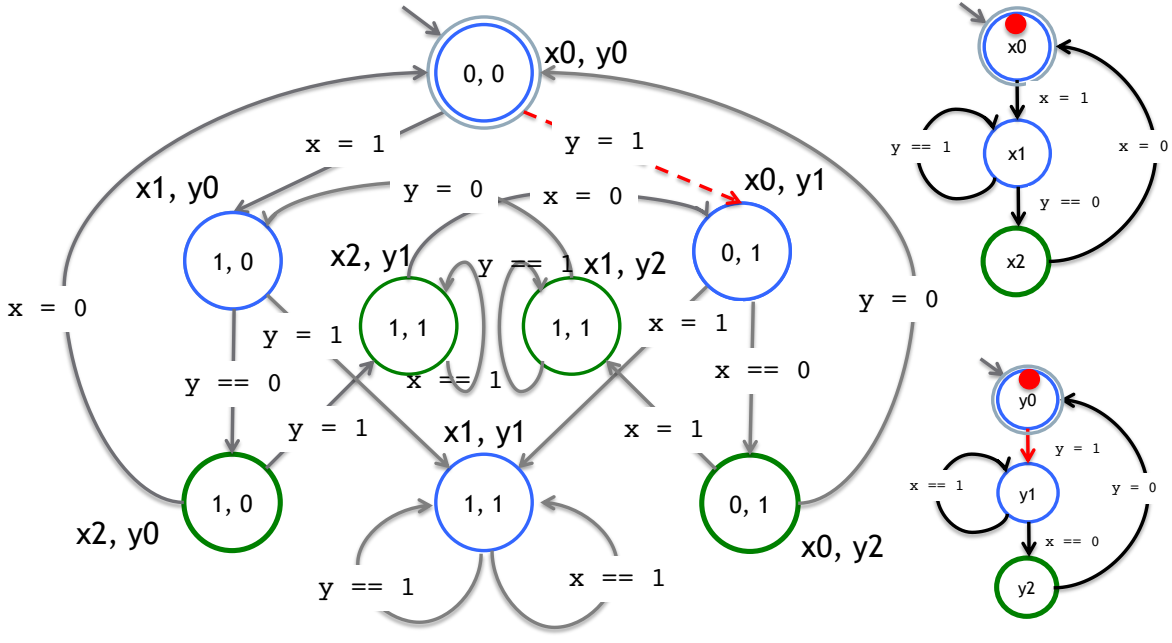
Hit a previously visited state x2, y1: backtrack to x2, y1; then to x2, y0; then x1, y0

Creating X ⊗ Y (15)



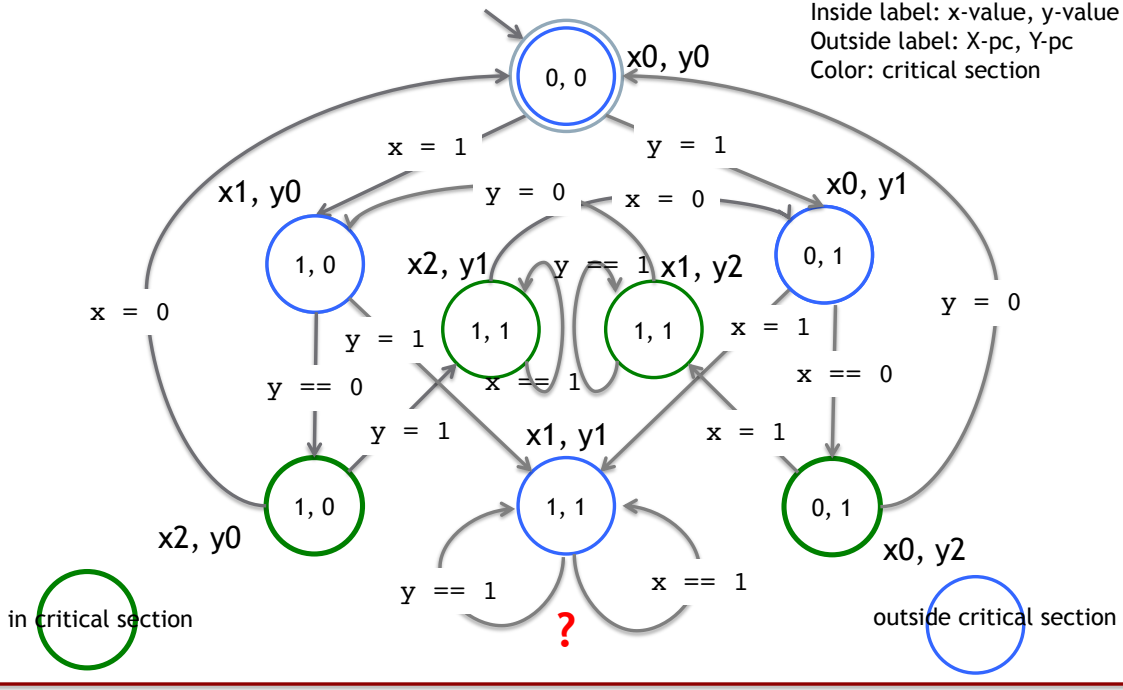
Hit a previously visited state x1, y1: backtrack to x1, y0; then x1, y2; then x0, y2; …. all the way to x0, y0!

Creating X ⊗ Y (16)

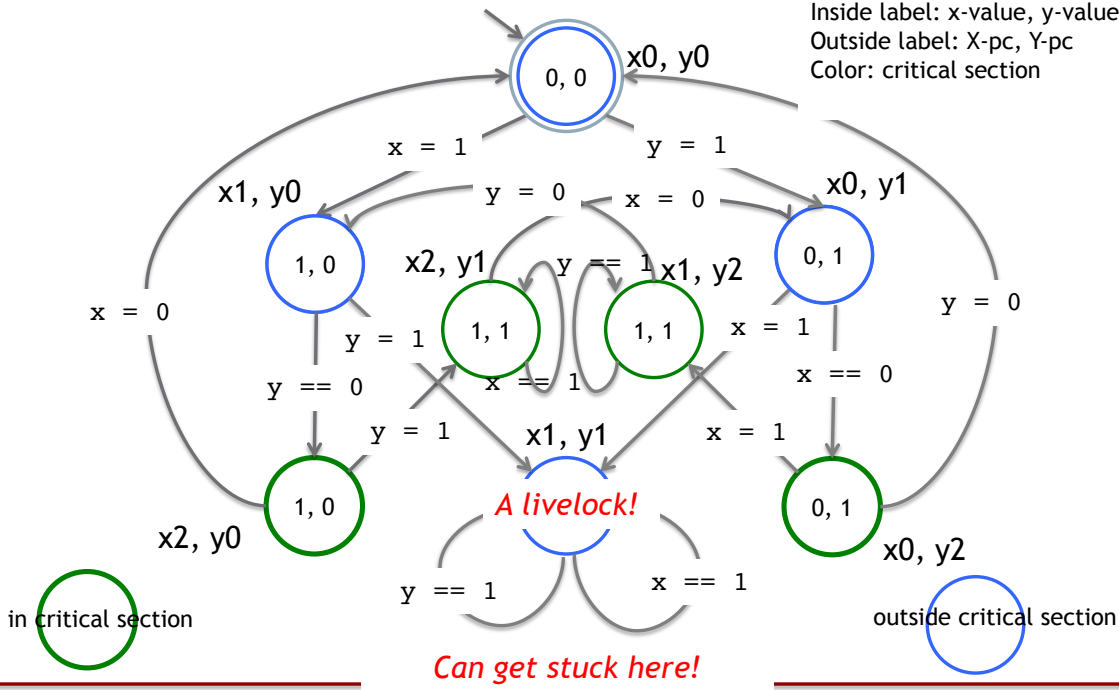


Hit a previously visited state x0, y1: backtrack to x0, y0;
No further transitions to explore: DONE!

The product FSA: final result

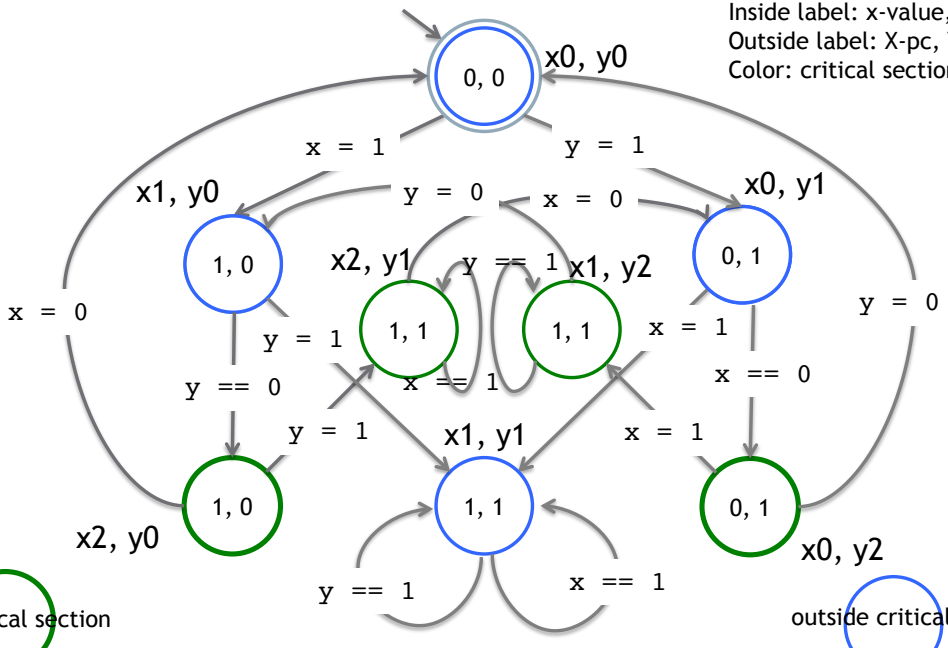


The product FSA showing a livelock



The product FSA showing mutual exclusion

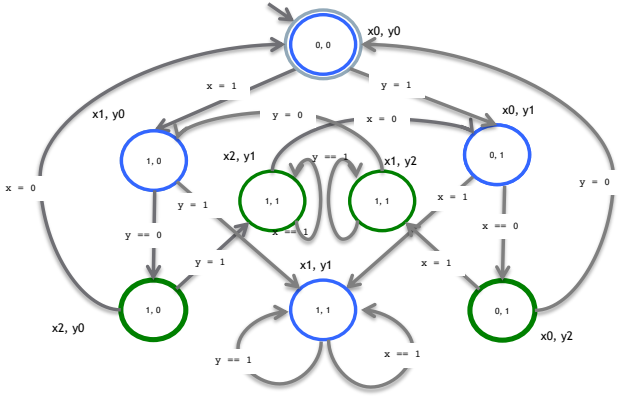
States:
Inside label: x-value, y-value
Outside label: X-pc, Y-pc
Color: critical section



State x2, y2 is not reachable (which is good)!

Mutex property satisfied.

The product FSA: properties summary



- Represents all possible global states: all possible, reachable interleavings of X and Y
- Any path through this graph is a serialized execution path of the whole system
- Includes only relevant detail for what we want to verify
- **Progress** property: detects the livelock state (fault)
- **Mutex** property: in every state, only one thread is in its critical section (state x2, y2 is not reachable) (good)

What did we just do?

model checking!

```
stop = false;
while (!stop) {
  x = 1;
  while (y == 1) {}
  mutex++;
  dSW(z);
  mutex--;
  x = 0;
}
```

S = Mutex-Program

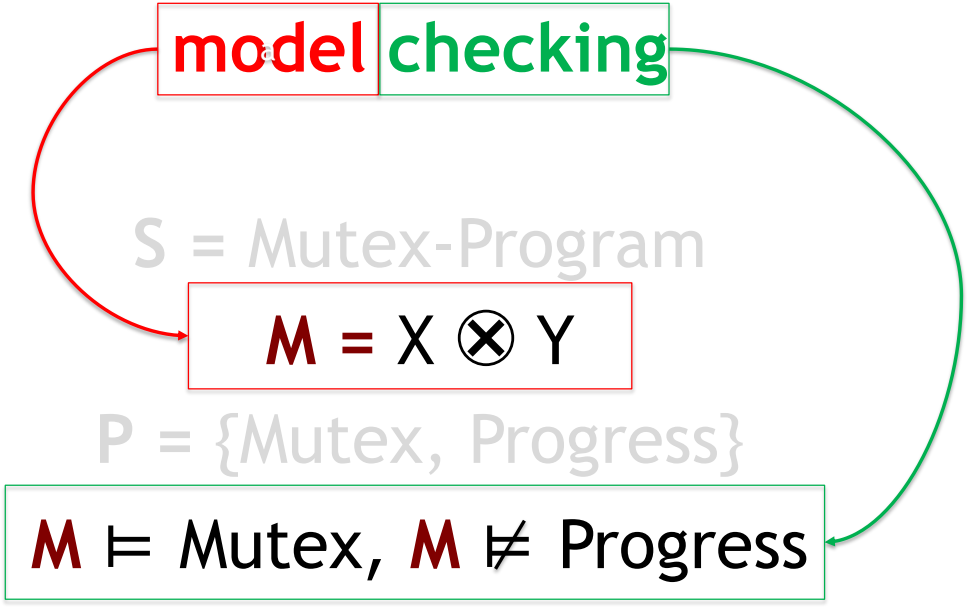
M = X ⊗ Y

P = { Mutex, Progress }

M ⊨ Mutex, **M** ⊭ Progress, **M** ⊭ P

```
stop = false;
while (!stop) {
  y = 1;
  while (x == 1) {}
  mutex++;
  dSW(z);
  mutex--;
  y = 0;
}
```

What did we just do?



[Clarke & Emerson 1981]:

"Model checking is an automated technique that, given a finite-state model of a system and a logical property, systematically checks whether this property holds for that model [starting from an initial state]."

Edmund Clarke, CMU & Allen Emerson, UT-Austin
2007 Turing Award Winners

Model checking is subject to limitations

- Models must be loyal/realistic enough
- Models should not restrict behavior with respect to properties
- Models must be finite-state
- Models must be sufficiently abstract (shouldn't have irrelevant details) so that model-checking is tractable

We can apply similar process to much more complex models and properties

- Decide which details to eliminate/keep in system
- Create the model in a formal modeling language with precise semantics
- Express properties we want to verify using formal logic
- Automatically generate the product FSA and check properties against this FSA
- If product FSA is too big, manage the state space via further reasonable optimizations/approximations

We can apply similar process to much more complex models and properties

- Decide which details to eliminate/keep in system
- Create the model in a formal modeling language with precise semantics (Promela)
- Express properties we want to verify using formal logic (Linear Temporal Logic)
- Automatically generate the product FSA and check properties against this FSA (Spin)
- If product FSA is too big, manage the state space via further reasonable optimizations/approximations (various Spin options)

Model checking is not the only formal verification technique

- **Abstract interpretation** (already discussed)
- **Symbolic execution**
- **Formal correctness proofs**

Formal verification techniques

- **Model checking** (what we started to describe)
  - Based on exploration of state space by a formal model of program
- **Abstract interpretation**: execute program with abstract inputs (as in division-by-zero analysis) and explore its abstract state space
- **Symbolic execution**: execute program with symbolic inputs (raw variables, no values) explore its symbolic state space (applied to automatic test case generation)
- **Formal correctness proofs**
  - Based on formal program semantics: axiomatic, operational, denotational
  - *Tools*: logic(s), inference, induction
  - Often based on **Hoare Logic** via pre- and post-conditions
  - May operate on formal models in a specific notation or directly on source programs (Java, C)