Final Exam

- Mandatory reading for final exam: "The Mars Code" by Gerard Holzmann
- Dec 12 @ 2:30pm-4:30pm Rm 109/110 and 118
- Closed book – BE CAREFUL ABOUT AIV!
- No notes, no electronic devices
- Only covers Static Analysis, Formal Verification, Model Checking, and above reading
- Download latest versions of lecture slides
- Before the exam: update your LockDown browser instance (otherwise it may force you to update it at the exam), and test it using the Sample Final exam!

18654 Final Review

Formal Verification & Static Analysis

Main Principles of SVT

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

Main Principles: Verification

- **Redundancy**: testing + verification
  - different types of verification/testing
  - techniques that combine verification & testing
- **Partitioning**: decompose model or property
- **Approximation**: modeling – reducing the problem, data abstraction in static analysis, AST as a substitute for code
- **Visibility:** properties as spec – explicit requirements
- **Feedback**: failure reason, counter-example when property is false
- **Repeatability**: deterministic results, a property either holds or not

Formal verification is good for…

Detecting faults caused by concurrency and non-determinism that are hard to reveal with testing

- Deadlocks (safety)
  - *Inability to make progress*
- Race conditions (liveness or safety)
  - *Untimely access to shared resources*
- Livelocks (liveness)
  - *Unproductive progress*
- Starvation (liveness)
  - *Unfair scheduling, unfair access to resources*

How to generate global state space: mutex example

Two threads accessing an exclusive resource z, trying to control access to z.
Introduce an shared variable `mutex` (init to 0) that serves as a monitor.

Thread X

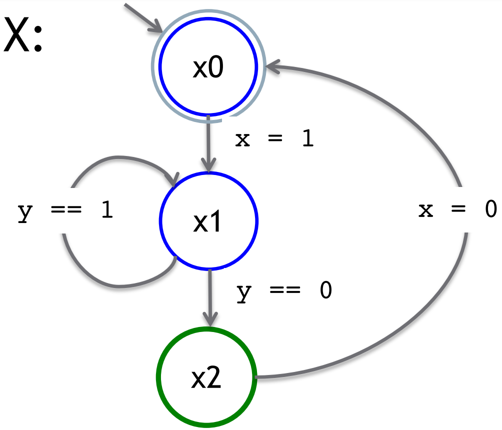
```
while (true) {
  x = 1;
  while (y == 1) {}
  mutex++;
  dSW(z);
  mutex--;
  x = 0;
}
```

Thread Y

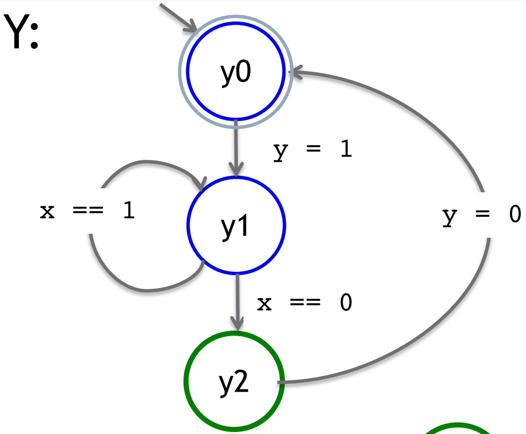
```
while (true) {
  y = 1;
  while (x == 1) {}
  mutex++;
  dSW(z);
  mutex--;
  y = 0;
}
```

- Mutex property: value of `mutex` should never be greater than 1
- No-livelock property: both threads can enter their critical section, *i.e.*, `mutex` can always become 1

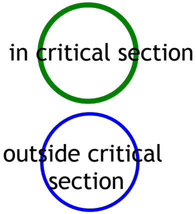
Finite State Automata for X and Y




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




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



Product Automata

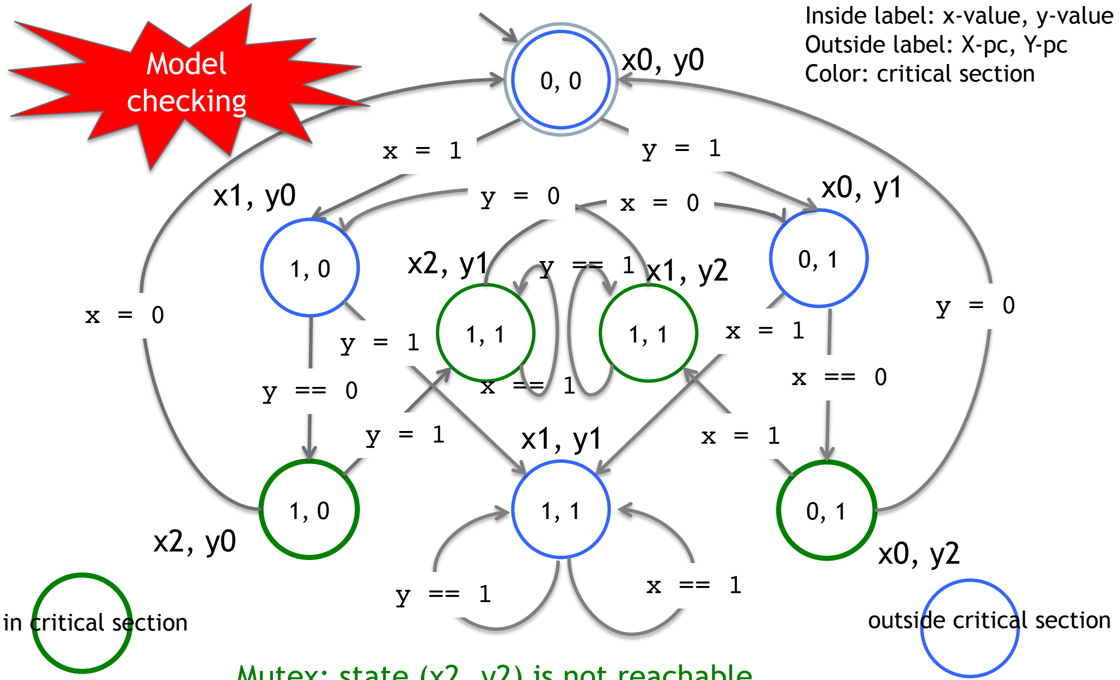
$$X \otimes Y$$

- Construction $X \otimes Y$ using interleaving semantics
- *Basically a search graph of the global state space*

Do you understand interleaving semantics?

The product FSA showing mutual exclusion and livelock

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



Mutex: state (x2, y2) is not reachable

Livelock: system gets stuck in state (x1, y1)

The imag e part with relati

Know how to translate a process to automaton!

Know how to construct the product automaton using interleaving semantics!

Understand the relationship between product automaton and global state space.

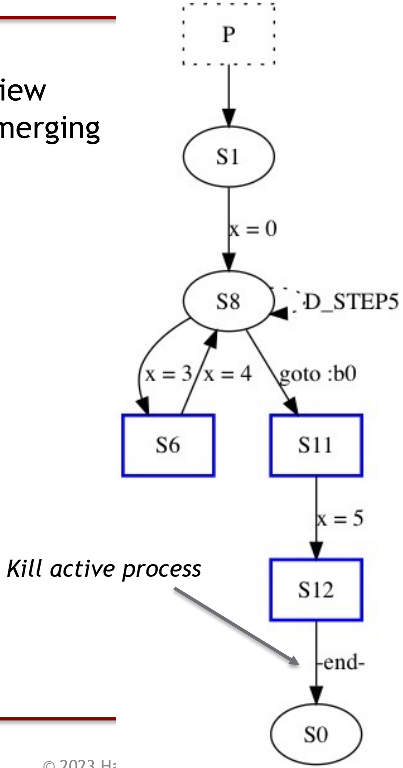
Convert this Promela active process to a FSA, manually

```
active proctype P() {
   int x;
   x = 0;
   do
   :: d_step { x = 1; x = 2; }
   :: x = 3; x = 4;
   :: break;
   od;
   x = 5;
}
```

Covert this Promela process instance to a FSA

```
active proctype P() {
   int x;
   x = 0;
   do
   :: d_step { x = 1; x = 2; }
   :: x = 3; x = 4;
   :: break;
   od;
   x = 5;
}
```

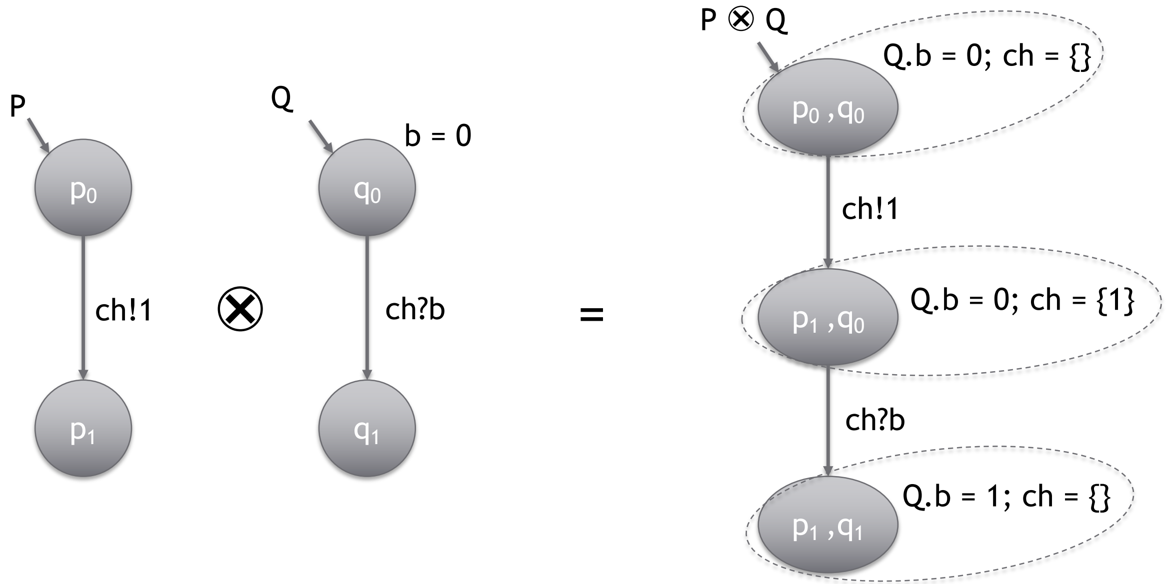
SpinRCP automata view
Turn off statement merging



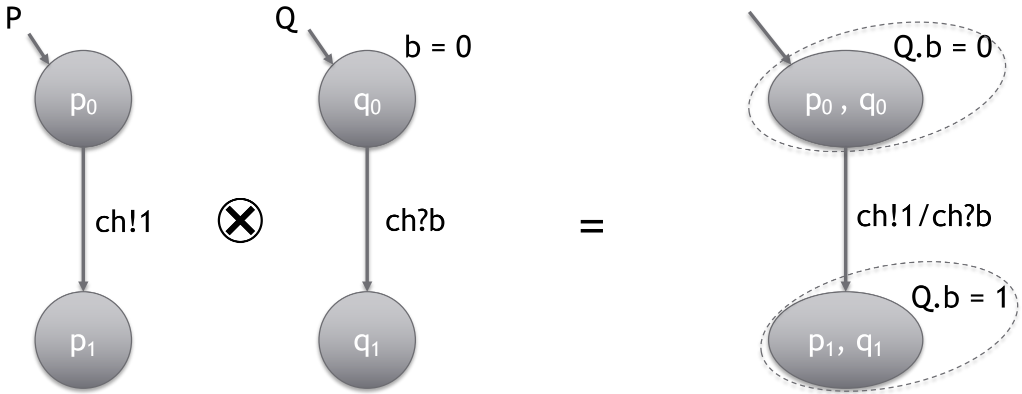
Process communication

- Asynchronous
  - message passing via asynchronous channels
  - shared global variables
- Synchronous
  - rendez-vous (message passing via synchronous channels – capacity of 0, stateless)

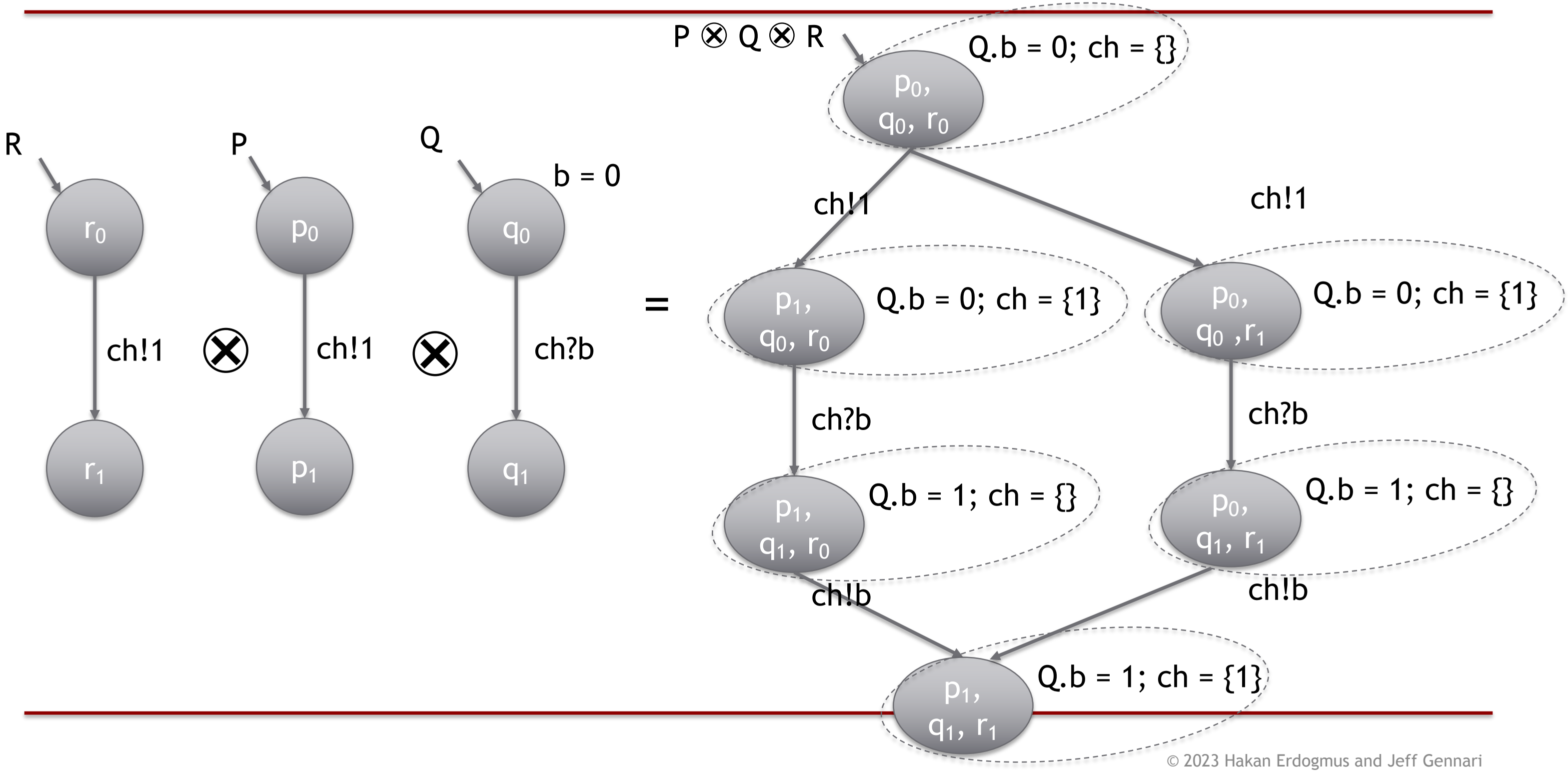
Asynchronous channel



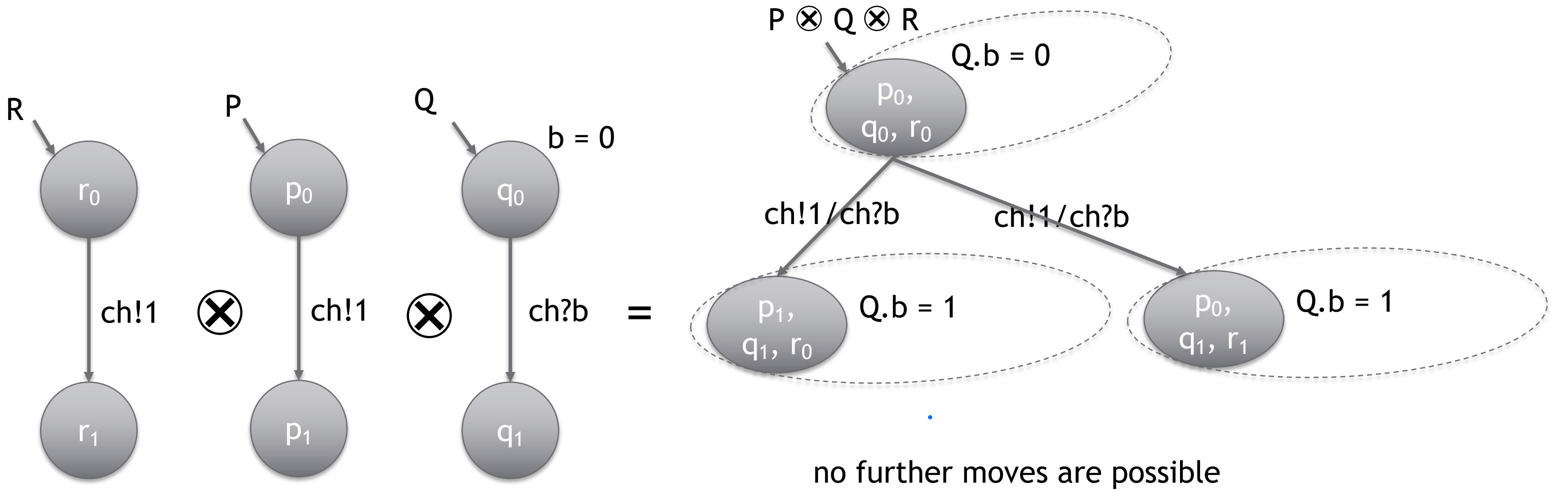
Rendez-vous – channel has no state!



Asynchronous channel



Rendez-vous - channel has no state!



no further moves are possible

Model Checking

$$M = M_1 \otimes M_2 \otimes \ldots \otimes M_n$$

$$M \vDash \Phi \; ?$$

Model checking: caveats

State explosion

$$|M| = |M_1| \times |M_2| \times \ldots \times |M_n|$$

You can estimate the worst-case state space and memory...

Estimating worst-case state space: L4 – Part 1

Let M be a Promela model that has two active processes P and Q that communicate with each other via two channels.

- Each channel has a maximum size of two.
- A slot in each channel can hold one of five different message types (an mytpe with 5 values).
- The FSA of P and Q each has 10 states.
- P and Q each has a local variable of type bool.
- M also has a global variable of type byte shared between the two processes.

1. What is the maximum number of global system states that M can have?
2. By what factor would the maximum size of the global state space increase if you increased the capacity of only one channel by 1?

总的来说就是 and = channel states * global variable states * process variable states

Estimating worst-case state space: L4 - Part 1

- Possible number of states of each channel: $\sum_{i=0..2} 5^i$ = **31** (includes empty channel)

  1 + 5 + 5*5 = 31

- Possible number of states of a channel with increased capacity of one channel:
  $\sum_{i=0..3} 5^i$ = **156**

- Possible values of global variable: **256**

- Possible local states of each process: **10 × 2 = 20** (10 states and 1 bit local variable)

1. Max number of global states: (31 × 31) × (20 × 20) × 256 = **98,406,400** = **~100 million!**
2. Max number of global states with increased channel capacity of one channel:
   (31 × 156) × (20 × 20) × 256 = **495,206,400** = **~1/2 billion!**

   495,206,400/98,406,400 = **5.03** => Max state space would increase by a factor of **~5**!

Possible Number of states of channel: (1+N+N*N+..+N^chan_capacity) ^num_chan

All possible states of all process = product of all process's (# states of FSA of that process * # local variables in that process) *

<u>Safety</u> properties…

… express behavior that a system should avoid

Something bad cannot happen!

Safety properties are about **states**…

- reachable vs. unreachable states
- system **invariants** that hold
  - *globally*: in all system states
  - *locally*: in selected model states

Designer's perspective

Verifier's perspective

<u>Liveness</u> properties…

… express behavior that a system must allow

Something good should happen!

- *termination*, *fairness*, *progress*

Designer's perspective

Liveness properties are about **paths**…

- combine state properties along an execution path to express dependencies between them
- temporal ordering of events (precedence, response, correlation)

Verifier's perspective

Tools for Specifying Properties

Safety (state) properties

- assertions
- deadlocks
- end-state labels
- LTL
  - []p where p doesn't have temporal operators
  - p, where p doesn't have temporal operators, just verifies the initial state

Liveness (path) properties

- LTL

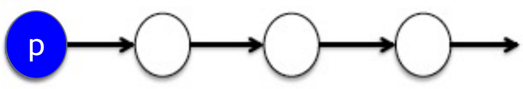
LTL operators you should know

Always	$[]\phi$	φ is true throughout
Eventually	$<>\phi$	φ is eventually true
Next	$X\phi$	φ is true in the next step (state)
Until (strong)	$\phi_1 \; U \; \phi_2$	ϕ_1 stays true until ϕ_2 becomes true (and ϕ_2 is eventually true)

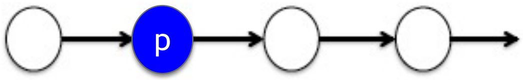
All have implicit universal quantification: for All paths…

LTL intuitive interpretation
(timeline diagram for a single path)

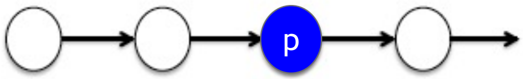
```
p: holds in current state
```



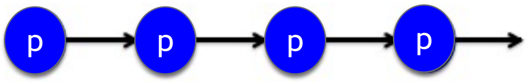

```
Xp: p holds in the next state
```



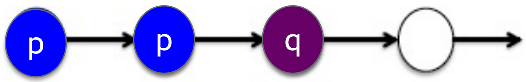

```
<>p: p holds eventually
```




```
[]p: p holds from now on
```




```
p U q: p holds until q holds
      (and q must eventually hold)
```



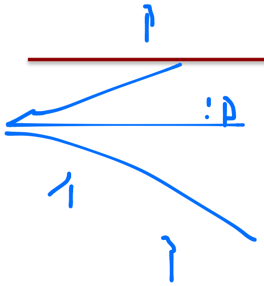
Or use LTL patterns (need to adapt them)

S: safety
L: liveness

Formula	Pronounced	Interpretation	Type
[]p	always p	invariance	S
<>p	eventually p	guarantee	L
p -> <>q	p implies eventually q	response	L
p -> (<>(q || r)	p implies eventually q or r	objective	L
[]<>p	always eventually p	recurrence (progress)	L
[](p -> <>q)	always, p implies eventually q	recurrent response	L
<>[]p	eventually always p	stability (non-progress)	L
!q U (p && !q)	p (strictly) precedes q	precedence	L
<>p -> <>q	eventually p implies eventually q	correlation	L

http://people.cis.ksu.edu/~dwyer/SPAT/ltl.html

Common LTL rules and false LTL rules



![]p	⇔	<>!p
!<>p	⇔	[]!p
[](p && q)	⇔	[]p && []q
<>(p || q)	⇔	<>p || <>q

~~![]p~~	~~⇔~~	~~[]!p~~
~~!<>p~~	~~⇔~~	~~<>!p~~

What about these?

<><>p = ?

[][]p = ?

What about these?

<><>p = <>p

[][]p = []p

Caution!

$M \nvDash \varphi$ does <u>not</u> imply $M \vDash\ !\varphi$

Reason: implicit universal quantification in LTL
($M \vDash \varphi$ iff "for all traces" τ of M, $\tau \vDash \varphi$)

Example

- Let φ = <>p and p holds somewhere in some traces of M, but it holds nowhere in other traces
- Then neither φ nor its negation $!\varphi$ = []!p will hold for M

"Fail" or "Existential" properties: especially applicable to LTL properties with preconditions

Consider the formula: `[](p -> <>q)`

- You want to show that this formula doesn't hold trivially because of the precondition p never being true
- That is, you want p to be true somewhere in at least one trace, if not somewhere in all traces

- You may separately check `<>p`, but this would require p to be true somewhere in all traces: *too strong!*
- **Solution:** Check `!<>p = []!p` and make sure it fails!

  → p is possible on some trace

Example: FAIL properties

```
#define eventual ... // a state proposition indicating some stability condition
#define stable ... // a state proposition indicating some eventuality condition

ltl ifStabilityThenEventuality { <>[]stable -> []<>eventual }
```

precondition

```
ltl notStability_FAIL { !<>[]stable }
```

!precondition

```
// if we want to prove that ifStabilityThenEventuality precondition is meaningful
// then we must show that !precondition fails
```

Static Analysis

Static Analysis -- data flow analysis with abstraction: detecting possible division by zero

```
x = 10;
y = x;
z = 0;
while (y > -1) {
    x = x/y;
    y = y – 1;
    z = z + 1;
}
```

Iteration 1	Iteration 2	Iteration 3
x:P		
x:P, y:P		
x:P, y:P, z:Z		
x:P, y:P, z:Z	x:P, y:M, z:P	x:W, y:M, z:P
x:P, y:P, z:Z	x:W, y:M, z:P	x:W, y:M, z:P
x:P, y:M, z:Z	x:W, y:M, z:P	x:W, y:M, z:P
x:P, y:M, z:P	x:W, y:M, z:P	x:W, y:M, z:P

Fixpoint

Over-approximation of x: P, N, Z, M, W, E

The image part with relati

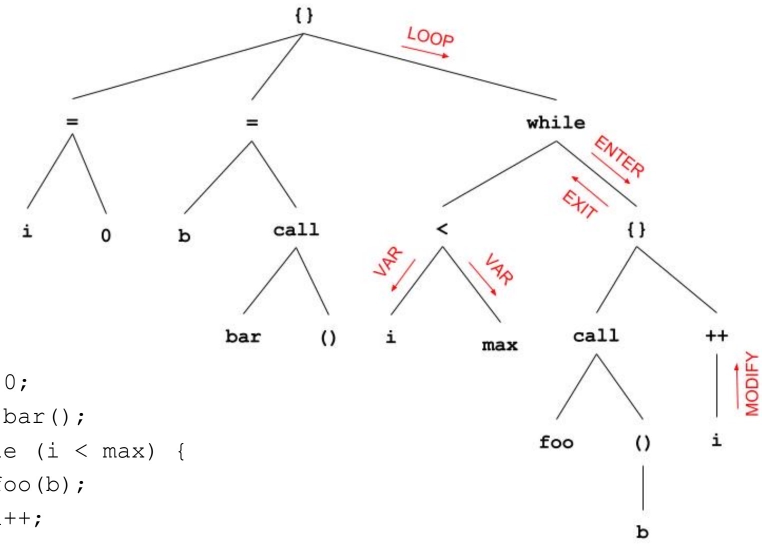
Pattern Checking Analysis on AST: Example

- A static analysis tool aims to detect possible infinite loops
- The extension walks through AST using a depth-first search algorithm while checking a specified pattern
- Pattern is defined as a FSA with actions: **LOOP, VAR, ENTER, MODIFY, EXIT**
- If at end of search: FSA is in accepting state: ok
- If FSA moves to non-accepting state: warning
- An auxiliary variable cvars stores variables appearing in the loop condition

AST Infinite Loop
Pattern Checking Rules (actions of FSA checker)

Action	*Trigger / Consequence*
LOOP	`while` construct encountered; sets `cvars` to empty
VAR	a variable identifier encountered in loop condition of `while`; adds encountered identifier to `cvars`
ENTER	`while` block entered
MODIFY	a variable in `cvars` is updated in `while` block (a variable is deemed updated if it appears on the LHS of an assignment or is the operand of an increment or decrement statement)
EXIT	`while` block exited

AST Infinite Loop Pattern Checking: Code Example



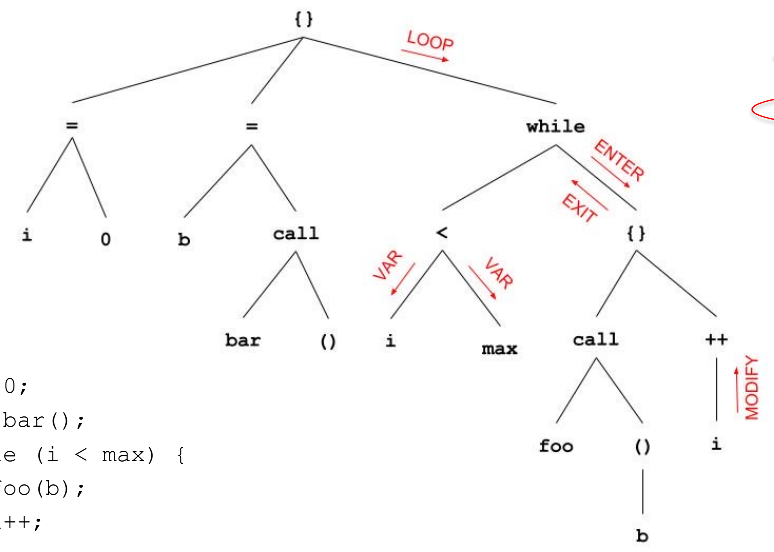

```
{
  i = 0;
  b = bar();
  while (i < max) {
     foo(b);
     i++;
    }
}
```

Definitely infinite loop?

1. (LOOP) (ENTER) (EXIT)
2. (LOOP) (VAR) (VAR) (ENTER) (EXIT)
3. (LOOP) (ENTER) (MODIFY) (EXIT)
4. (LOOP) (VAR) (ENTER) (MODIFY) (EXIT)
5. None of the above

FSA checker checks the undesirable pattern while performing a depth-first search of the AST

AST Infinite Loop Pattern Checking: Code Example




```
{
  i = 0;
  b = bar();
  while (i < max) {
     foo(b);
     i++;
    }
}
```

Definitely infinite loop?

1. (LOOP) (ENTER) (EXIT) report
2. (LOOP) (VAR) (VAR) (ENTER) (EXIT) report
3. ~~(LOOP) (ENTER) (MODIFY) (EXIT)~~ Syntactically impossible
4. (LOOP) (VAR) (ENTER) (MODIFY) (EXIT) ok
5. None of the above

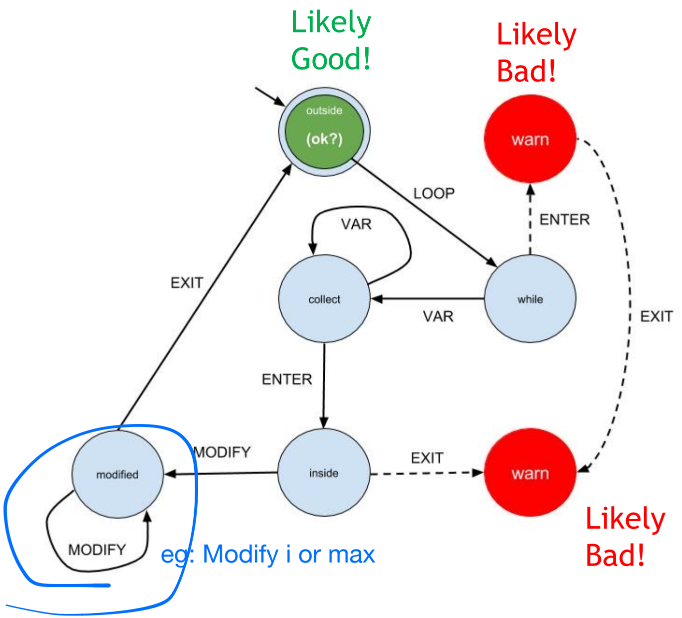
None of the above because even though some are signs of an infinite loop, none are guaranteed to be infinite loops at run-time!

Corresponds to example: ok pattern, but still does not guarantee absence of infinite loop

(LOOP)(ENTER)(VAR)(VAR)(MODIFY)(EXIT)

FSA to Recognize Possible Infinite Loop Pattern

...for syntactically valid code



Static analysis takeaways

- Low-hanging fruit: easy, fully automated, push-button
- Can easily be integrated into build process
- May produce too many spurious warnings, but most tools can be fine-tuned
- May detect easy-to-miss errors that may be hard to detect with testing, but have severe consequences

Some general concept question about splint taintness on the final

Good Luck!