# Lecture 11

# Today's class

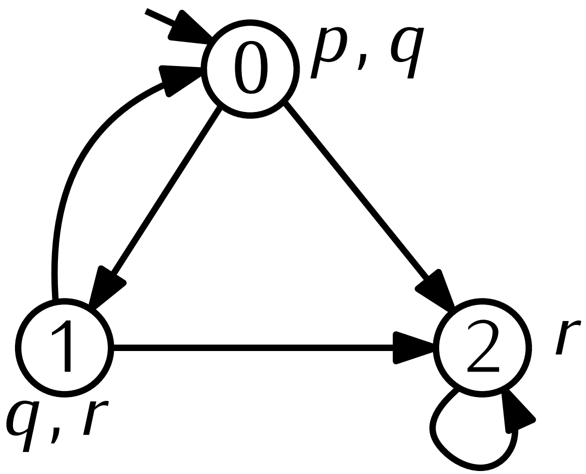
**Quick Review**

LTL and CTL

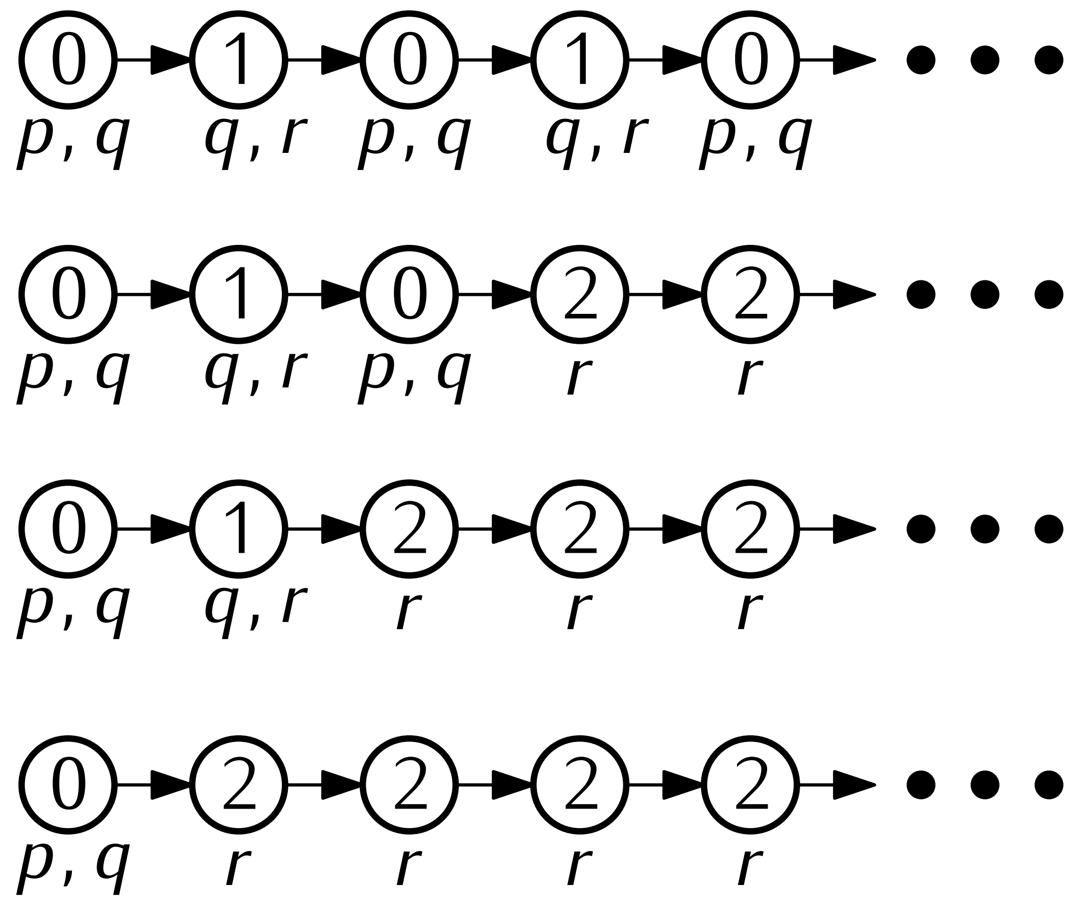
**Verifying properties of a stack**

Modeling, specifying, and verifiying stack properties.

# Linear Temporal Logic (LTL) Review



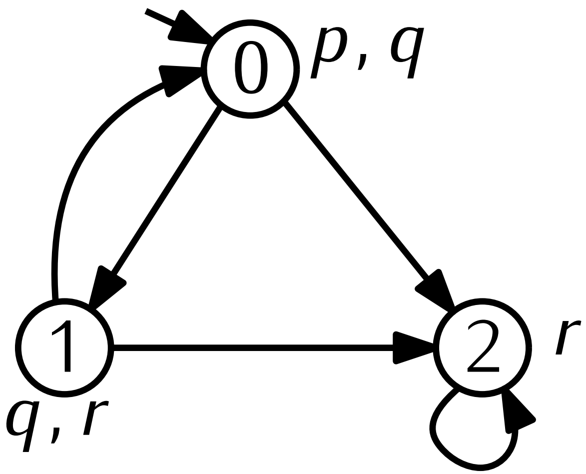

Some paths of $\mathcal{M}$



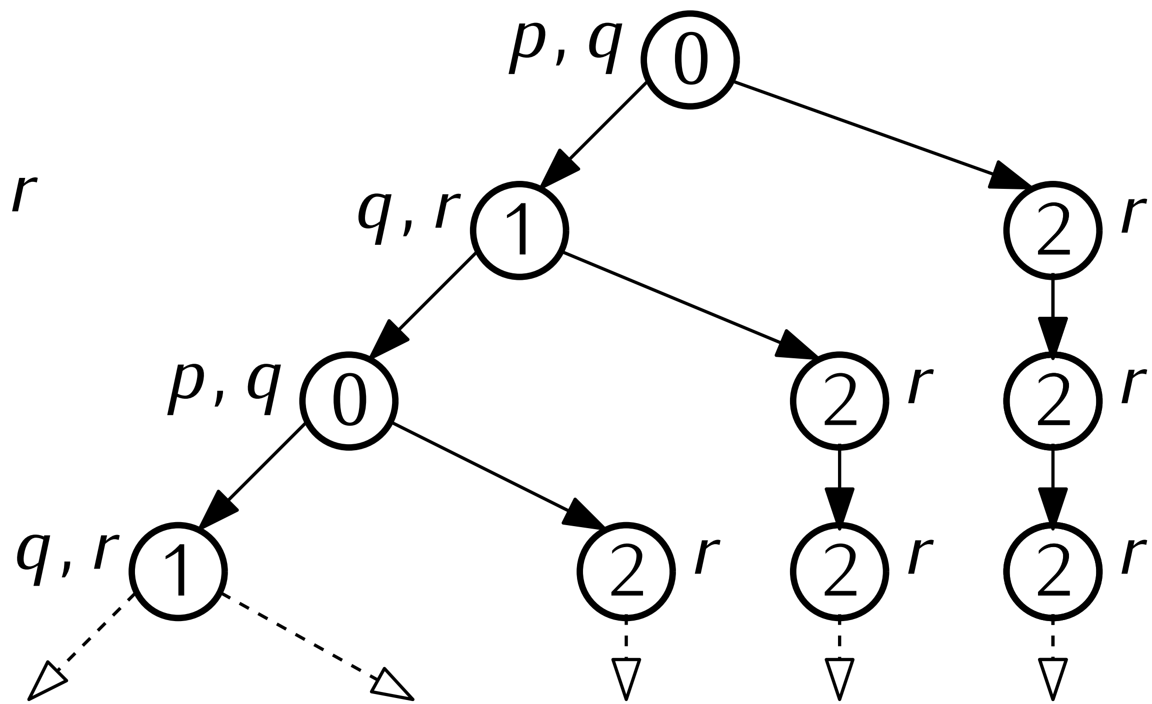

$$\mathcal{M} \models \phi \Leftrightarrow \forall \pi \ [\pi \models \phi]$$

LTL Model Checking

# Computation Tree Logic (CTL) Review




Computation tree for $\mathcal{M}$




Computation Tree Logic (CTL) expresses properties of "alternative timelines".

$$\mathcal{M} \models \phi \Leftrightarrow \forall s \in I \;\; s \models \phi$$

CTL Model Checking

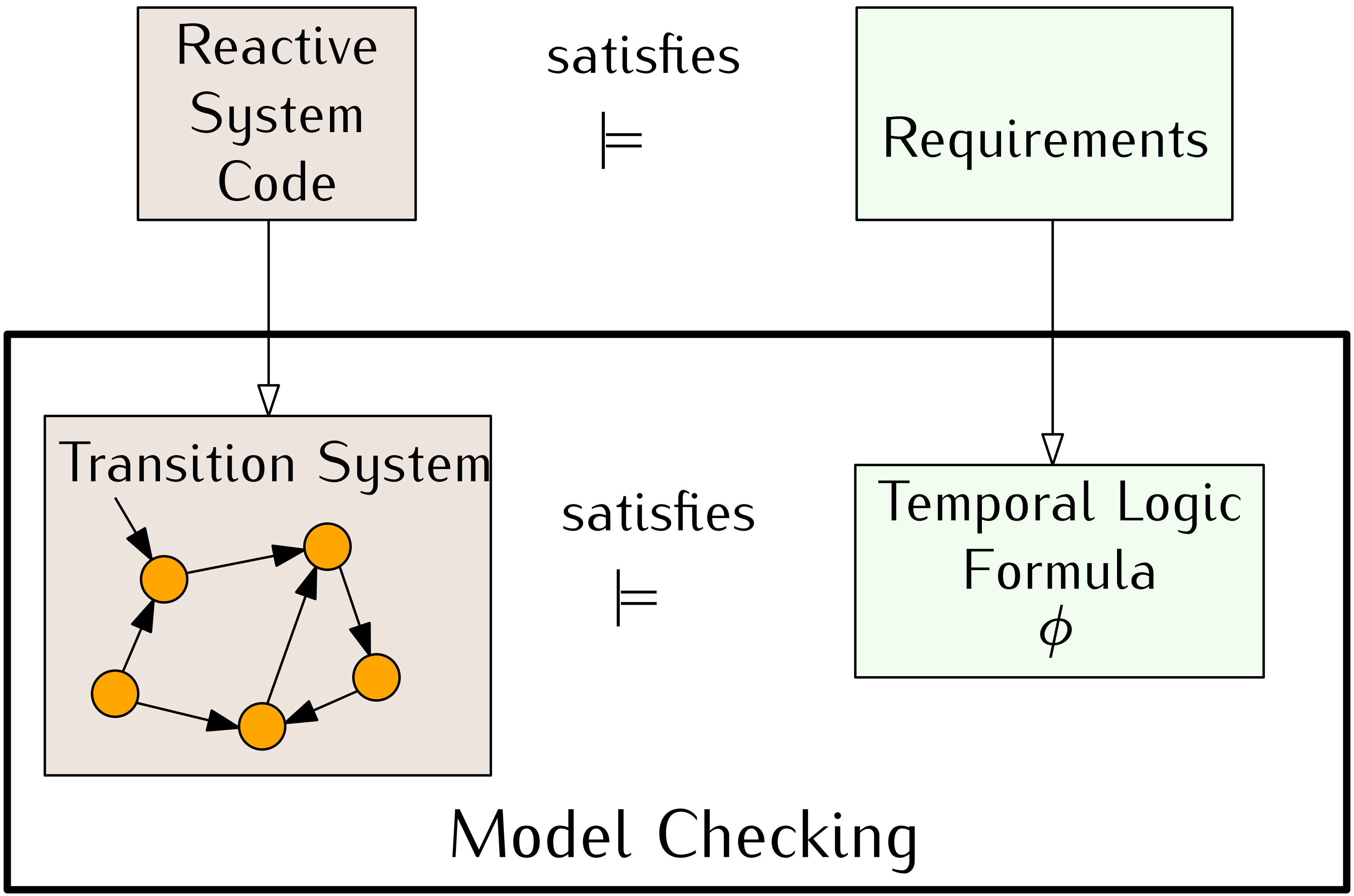
HW questions?

LTL questions?

CTL questions?

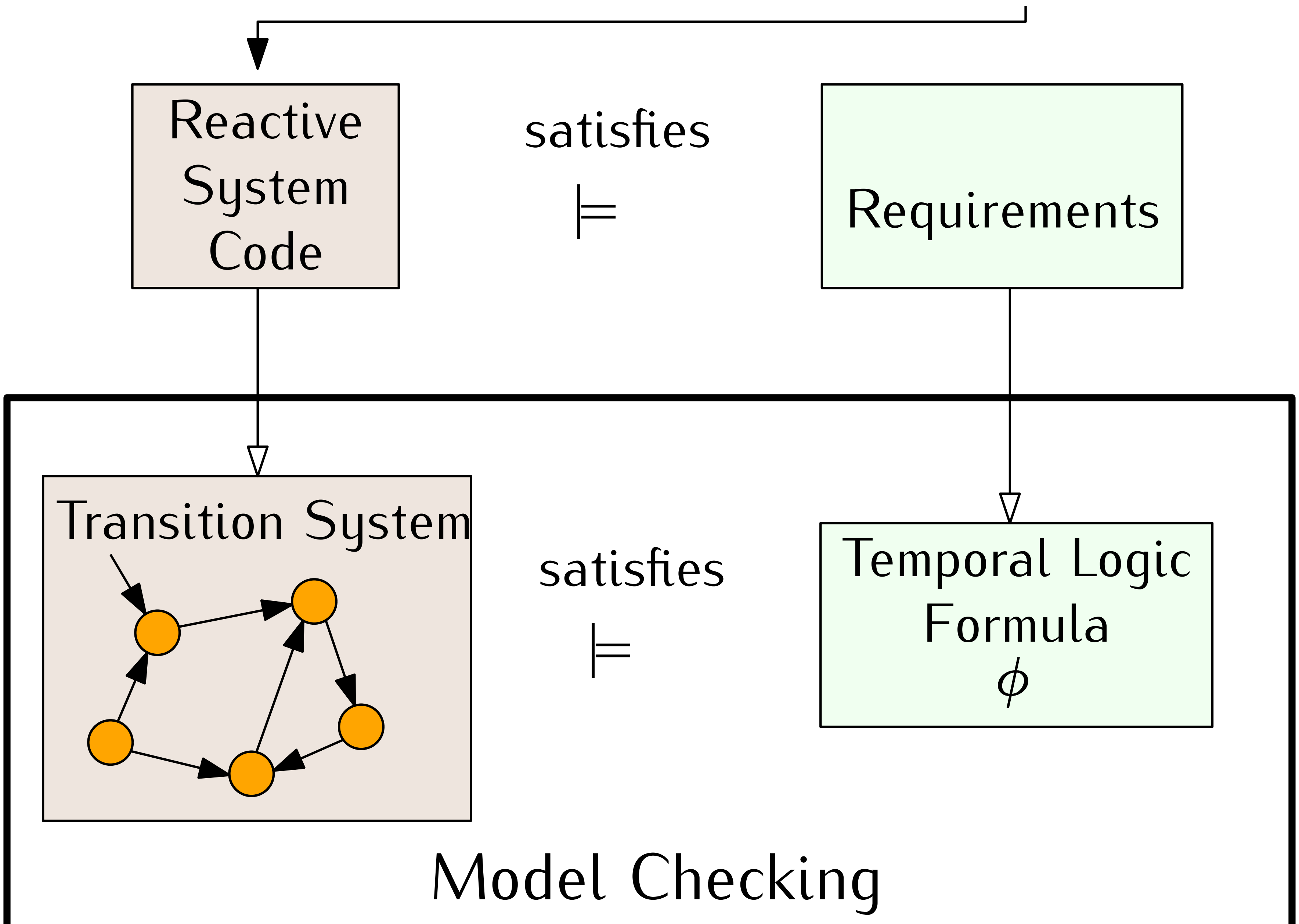
$\nu$SMV questions?

# The Big Picture

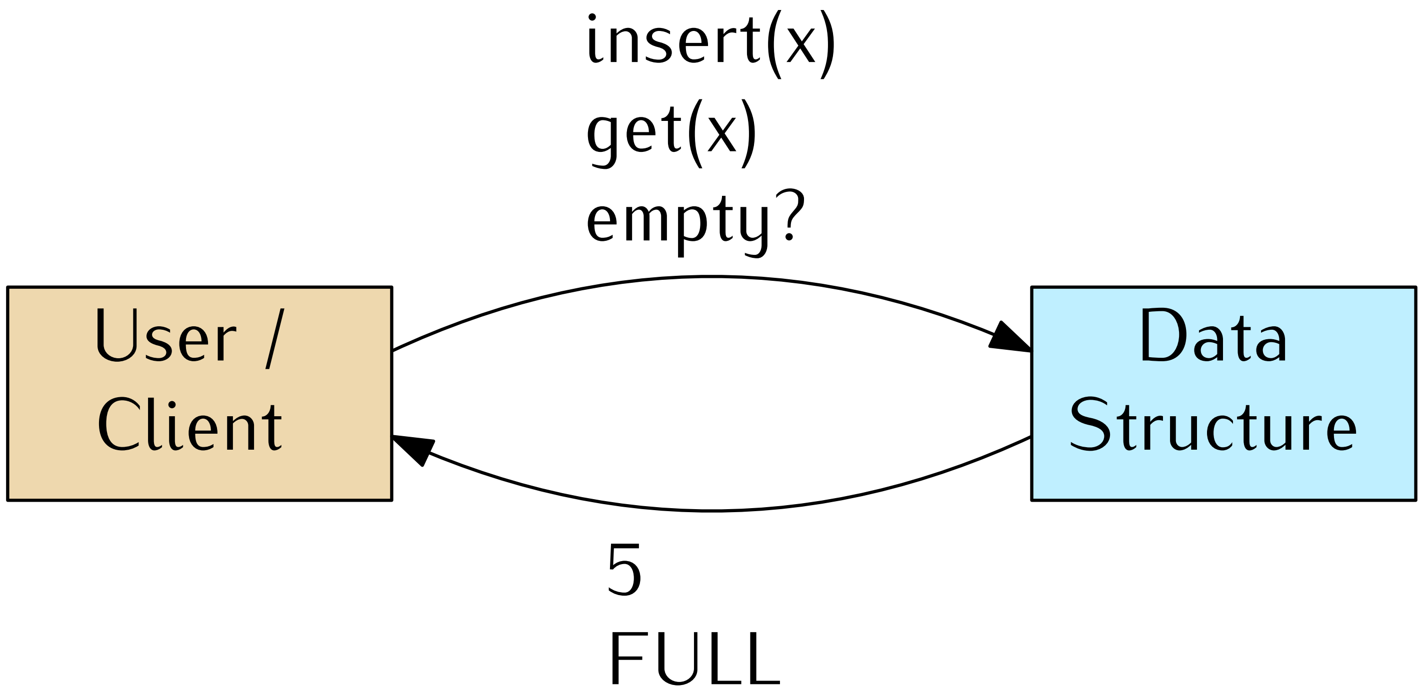


# The Big Picture

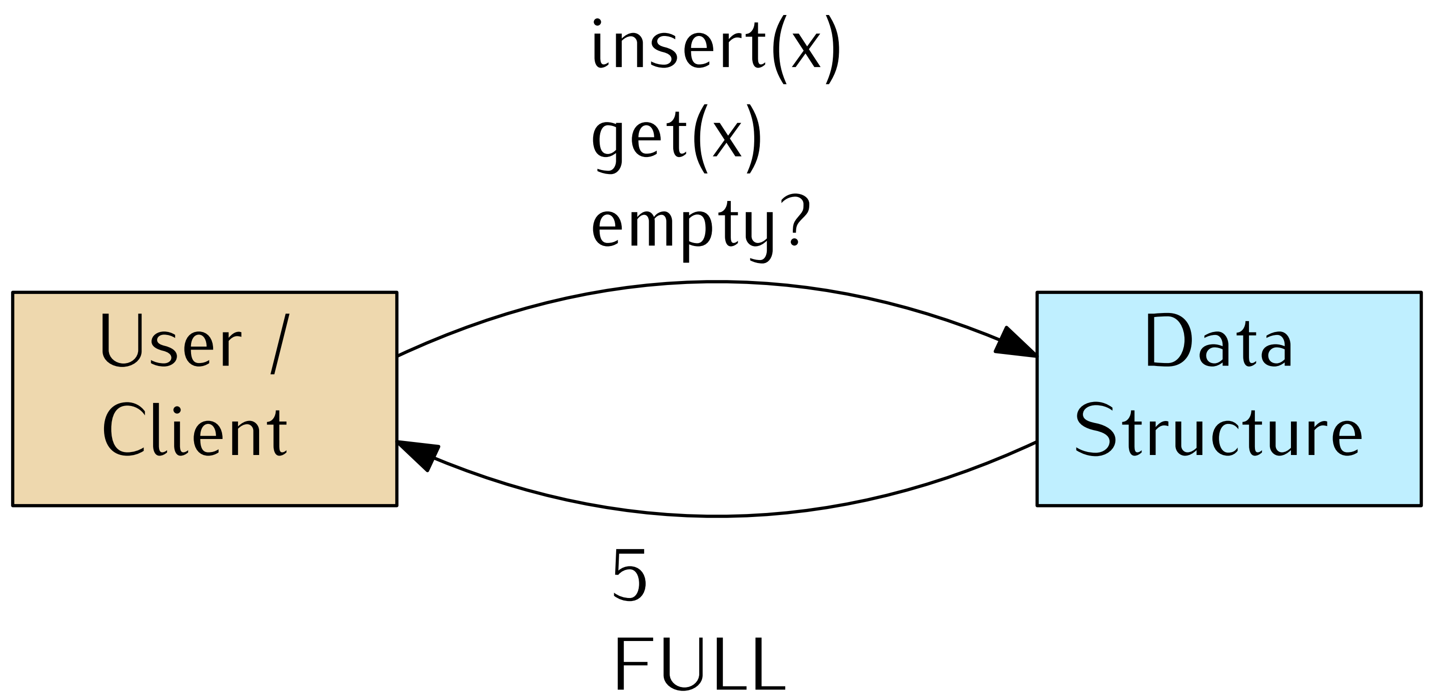
Can we think of a data structure as a reactive system?



# Data Structures as Reactive Systems



# Data Structures as Reactive Systems

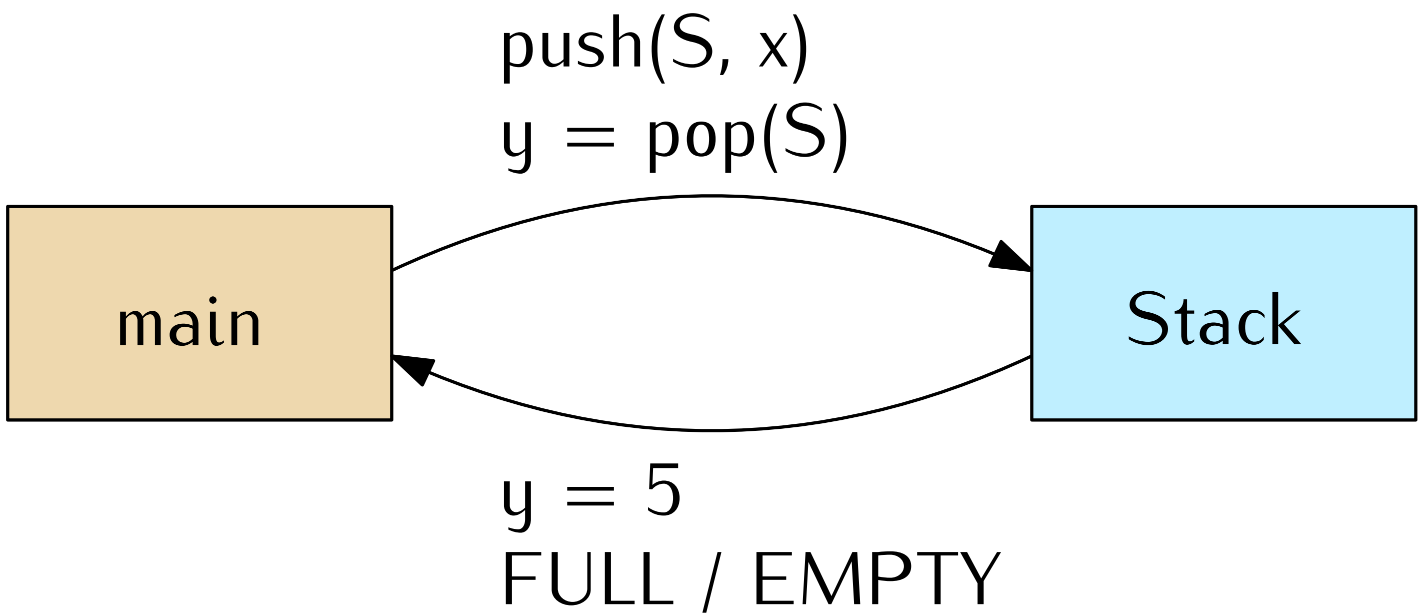



**Both the user / client and data structure:**

Internal state that changes over time (temporal aspects).

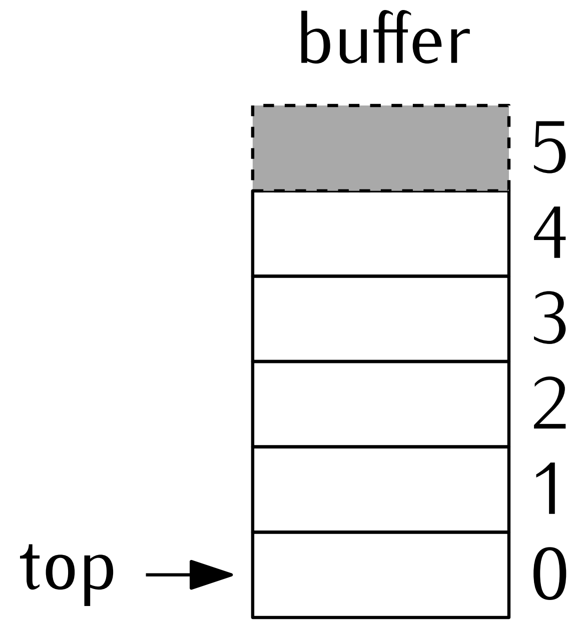
Indefinite lifetime, "runs" forever.

# Modeling a Stack



# Modeling a Stack

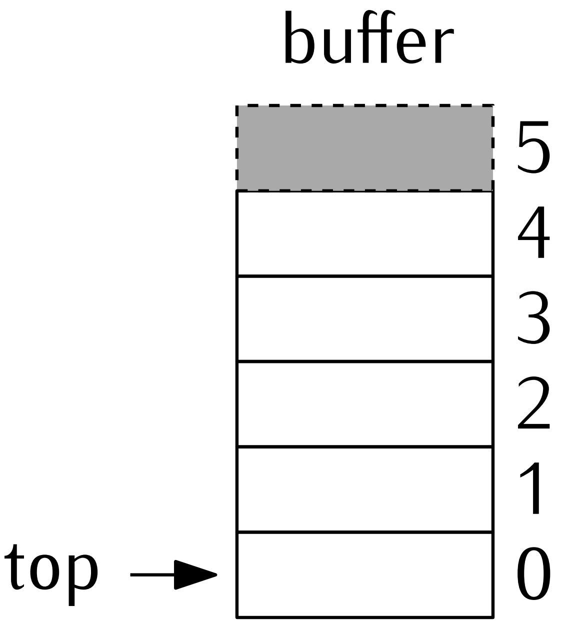
A stack has a finite **buffer** with integer indices and a pointer to the **top** of the stack.



# Modeling a Stack

A stack has a finite **buffer** with integer indices and a pointer to the **top** of the stack.

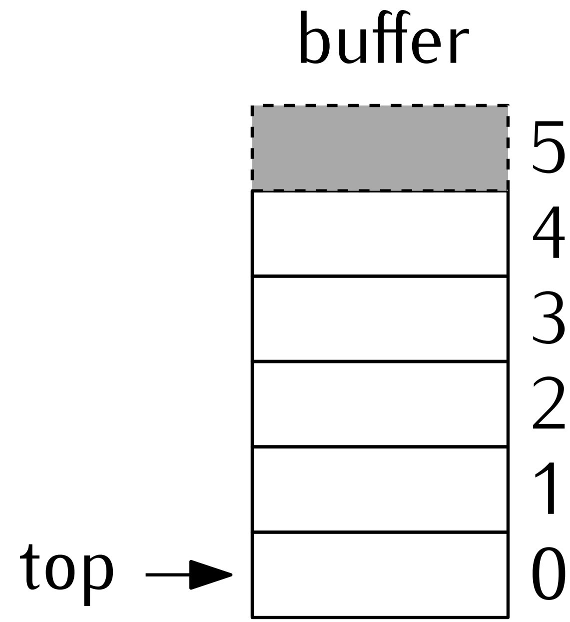
The stack pointer, top, indicates the next available place to store a value.




Pushing increments top, popping decrements top.

# Modeling a Stack

A stack has a finite **buffer** with integer indices and a pointer to the **top** of the stack.




The stack pointer, top, indicates the next available place to store a value.

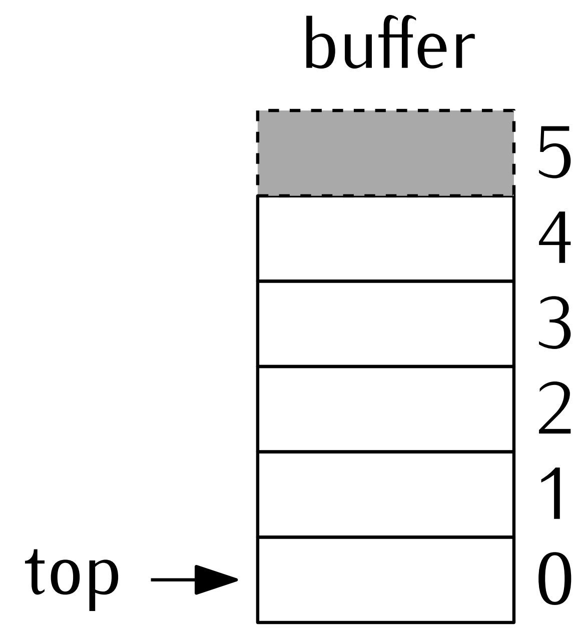
Pushing increments top, popping decrements top.

When top $= 0$, stack is empty.

# Modeling a Stack

A stack has a finite **buffer** with integer indices and a pointer to the **top** of the stack.

The stack pointer, top, indicates the next available place to store a value.



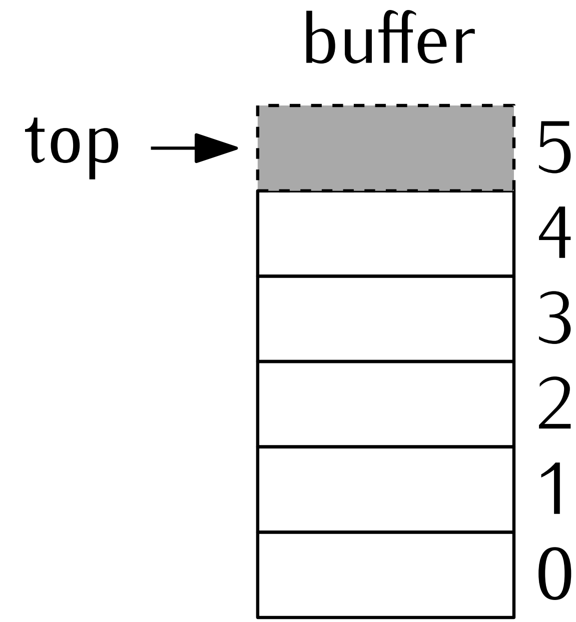

Pushing increments top, popping decrements top.

When top = 0, stack is empty.

Attempting to pop an empty stack returns NULL.

# Modeling a Stack

A stack has a finite **buffer** with integer indices and a pointer to the **top** of the stack.




The stack pointer, top, indicates the next available place to store a value.

Pushing increments top, popping decrements top.

When top = 0, stack is empty.

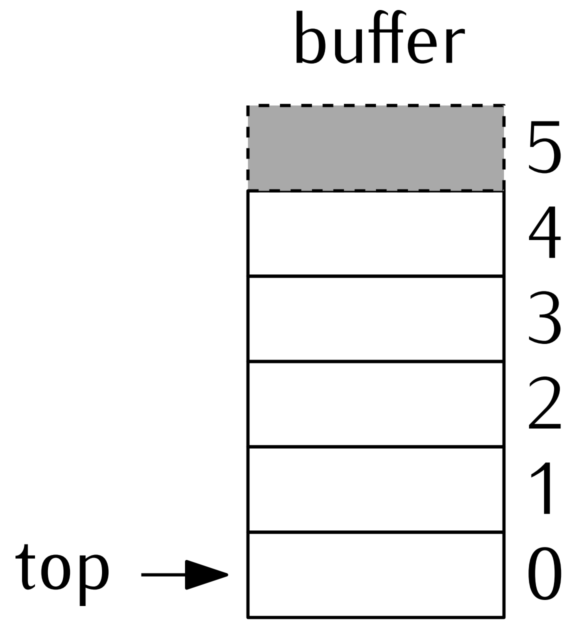
Attempting to pop an empty stack returns NULL.

When top = SIZE, stack is full.

Attempting to push when stack is full does nothing.

# Modeling a Stack

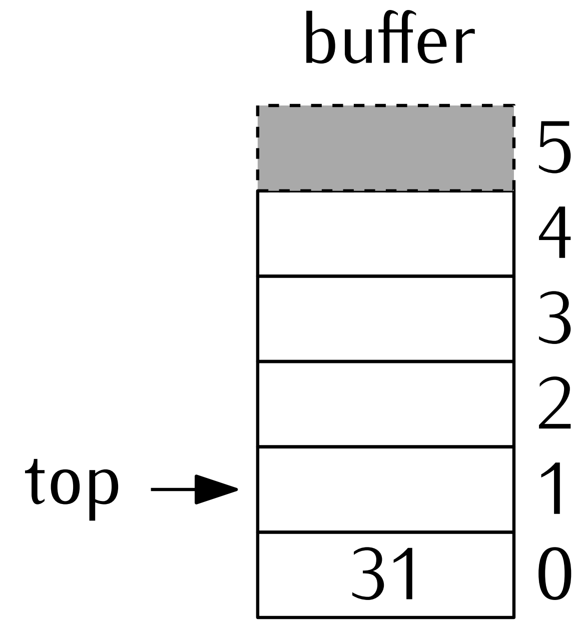
An example execution



# Modeling a Stack

An example execution

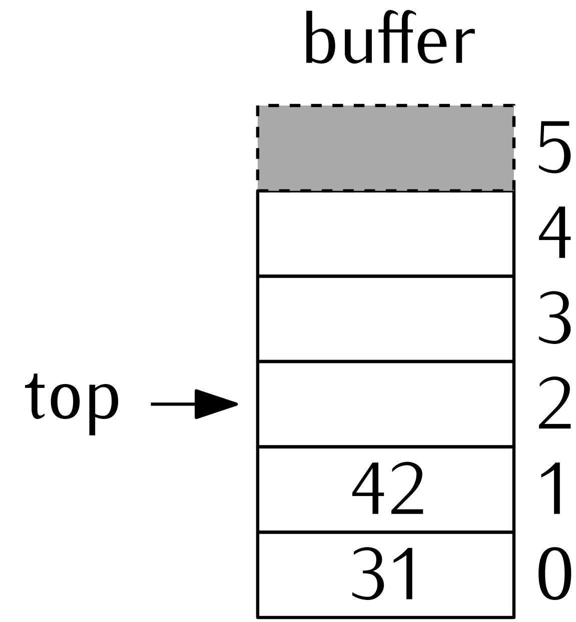
push 31



# Modeling a Stack

An example execution

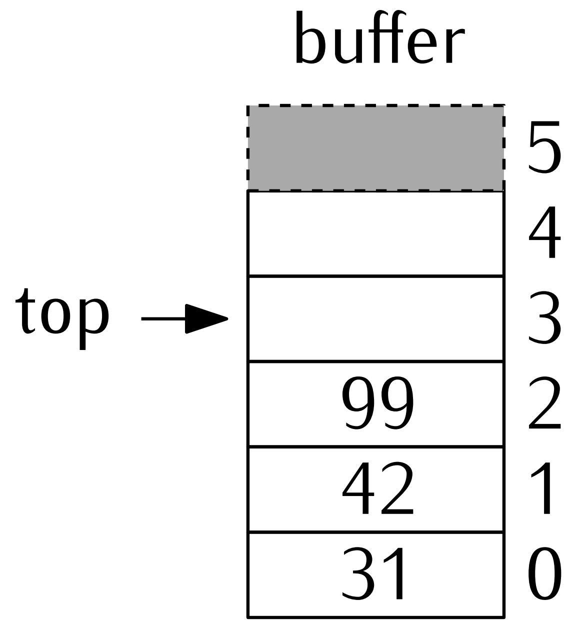
push 31
push 42



# Modeling a Stack

An example execution

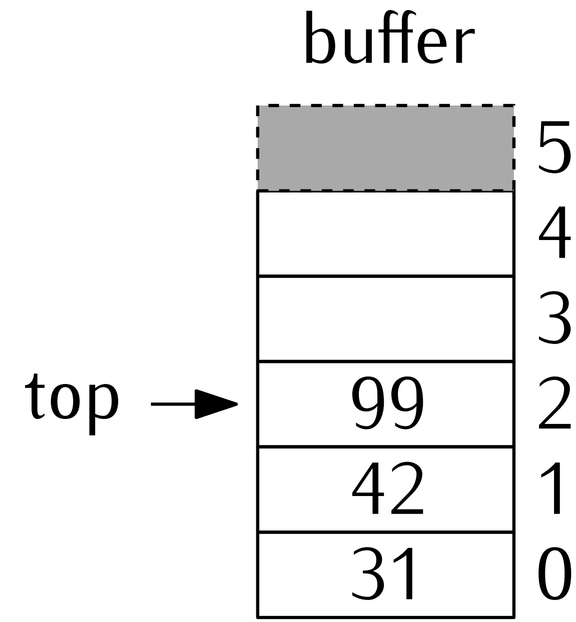
push 31
push 42
push 99



# Modeling a Stack

An example execution

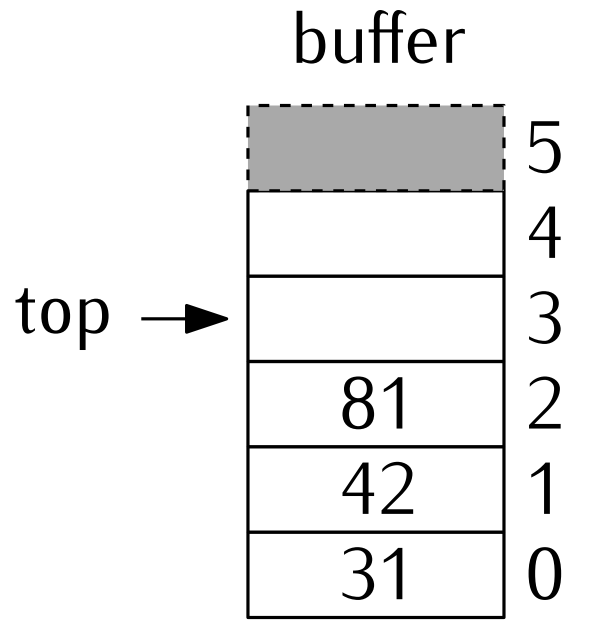
push 31
push 42
push 99
v = pop   v = 99



# Modeling a Stack

An example execution

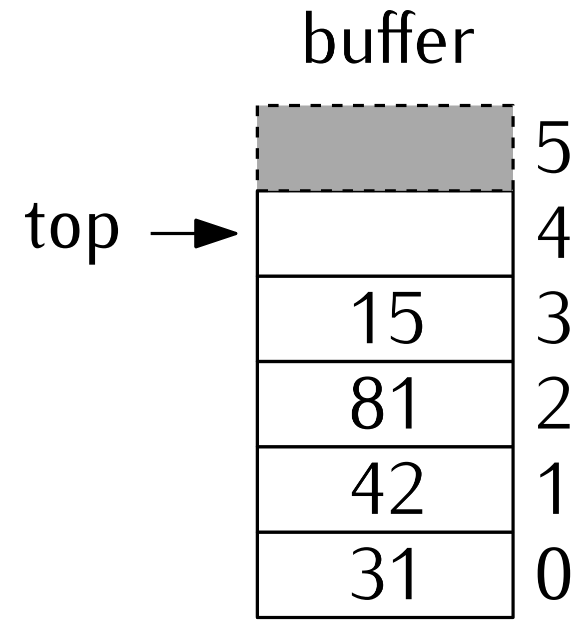
push 31
push 42
push 99
v = pop   v = 99
push 81



# Modeling a Stack

An example execution

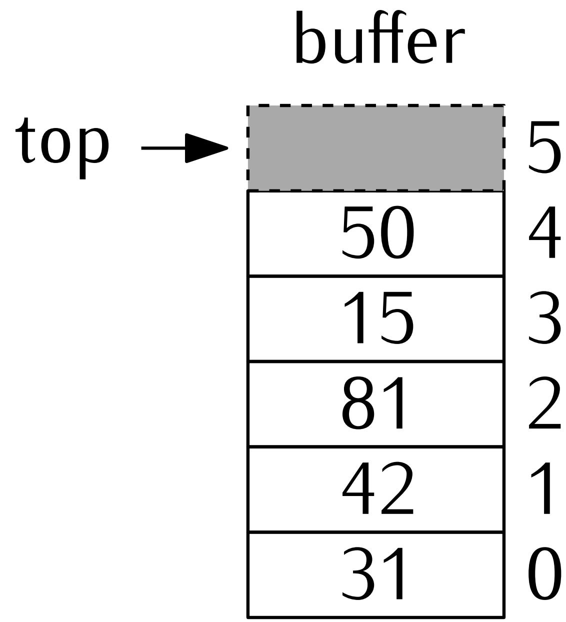
push 31
push 42
push 99
v = pop    v = 99
push 81
push 15



# Modeling a Stack

An example execution

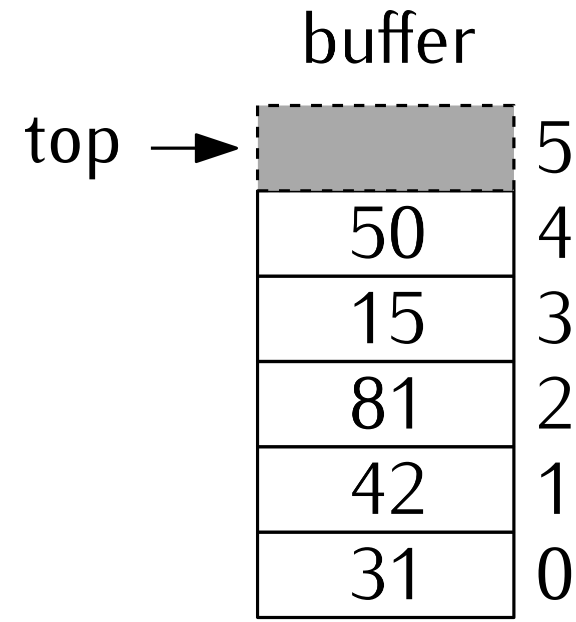
push 31
push 42
push 99
v = pop   v = 99
push 81
push 15
push 50



# Modeling a Stack

An example execution

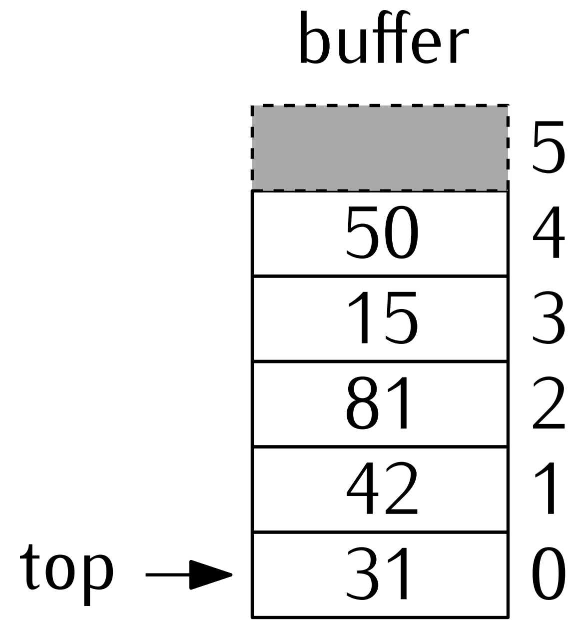
push 31
push 42
push 99
v = pop   v = 99
push 81
push 15
push 50
push 25   nothing happens



# Modeling a Stack

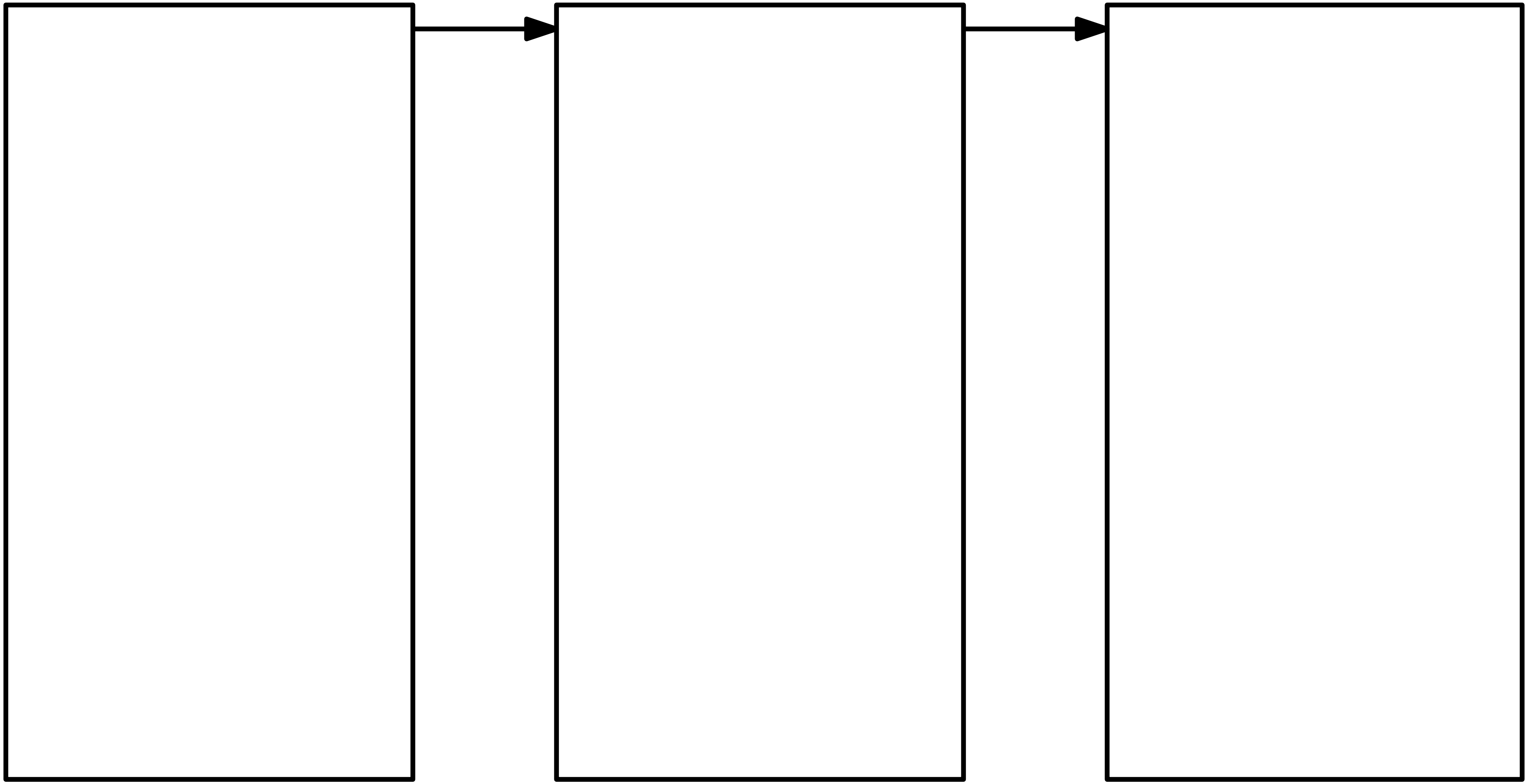
An example execution

| | |
|---|---|
| push 31 | |
| push 42 | |
| push 99 | |
| v = pop | v = 99 |
| push 81 | |
| push 15 | |
| push 50 | |
| push 25 | nothing happens |
| v = pop | v = 50 |
| v = pop | v = 15 |
| v = pop | v = 81 |
| v = pop | v = 42 |
| v = pop | v = 31 |
| v = pop | v = NULL |



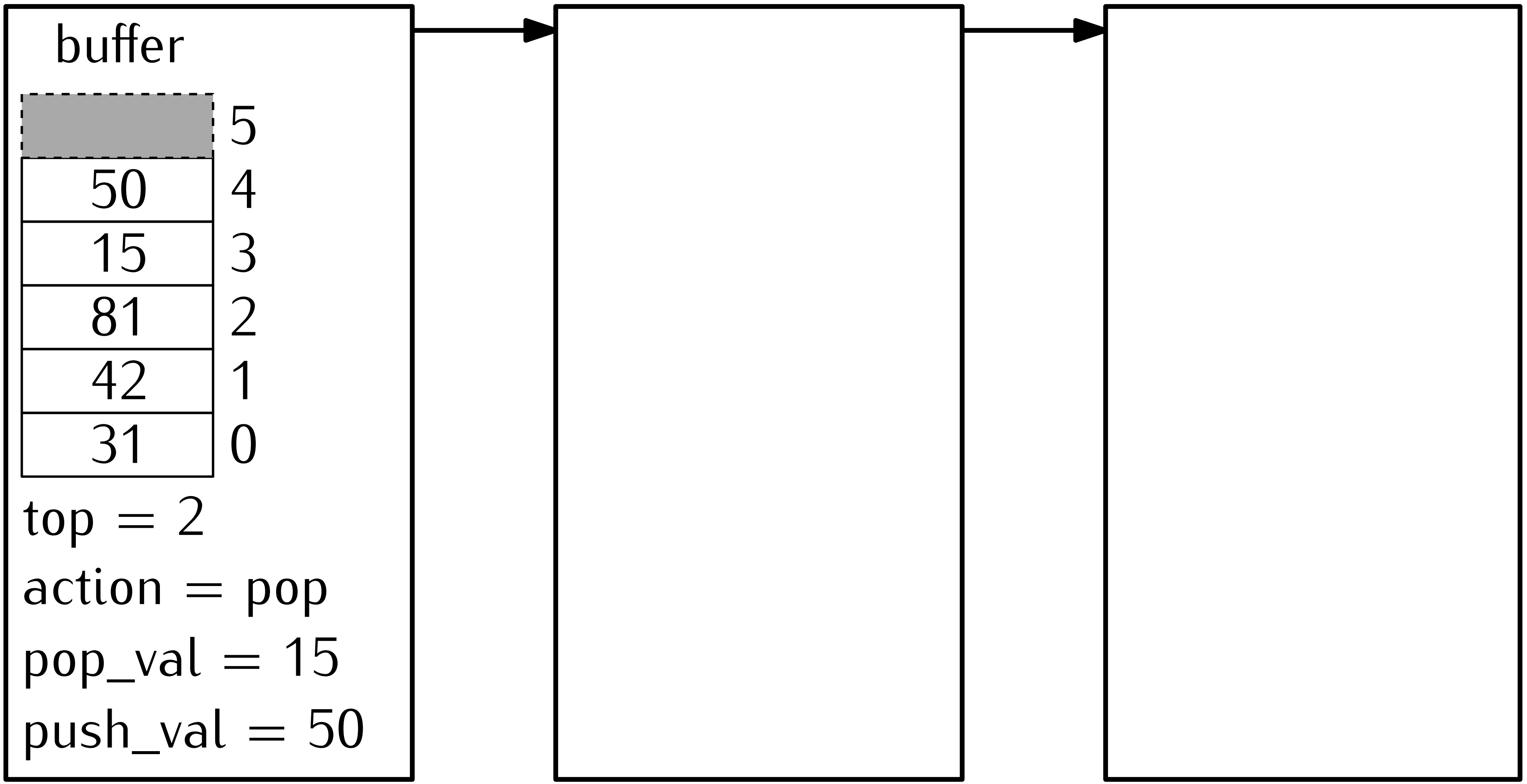
# Modeling a Stack in *ν*SMV

What do the states and transitions look like?



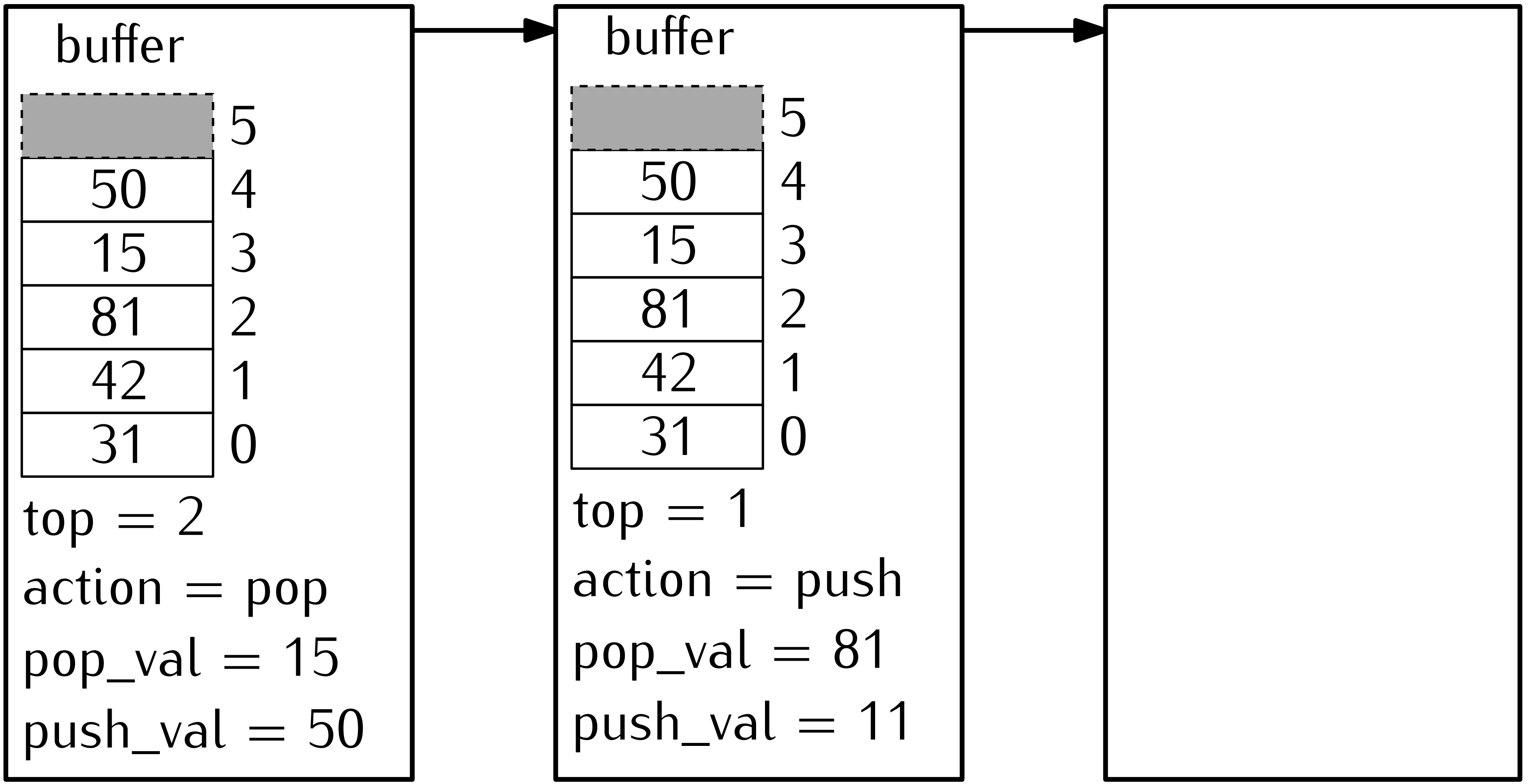
# Modeling a Stack in $\nu$SMV

What do the states and transitions look like?



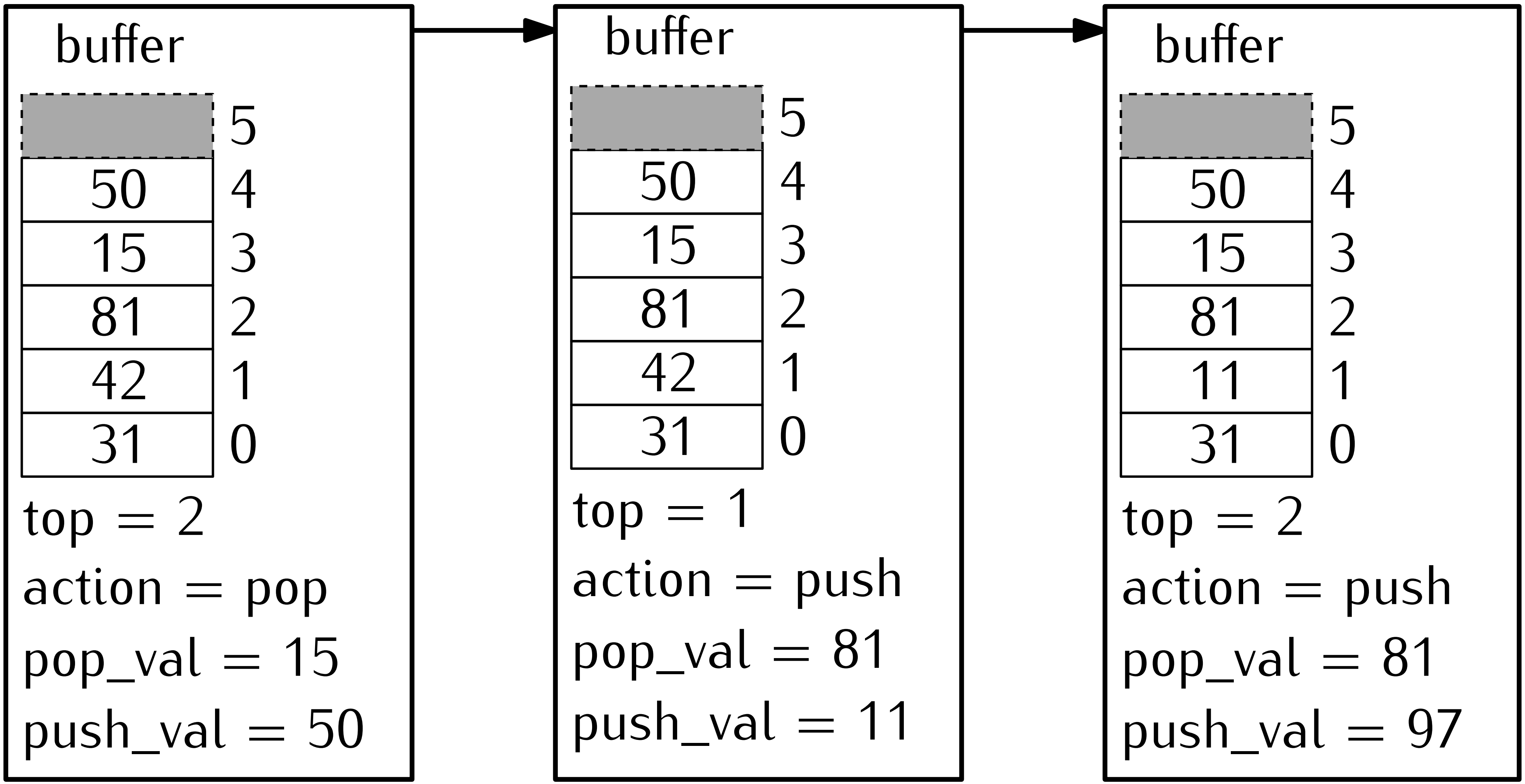
# Modeling a Stack in $\nu$SMV

What do the states and transitions look like?



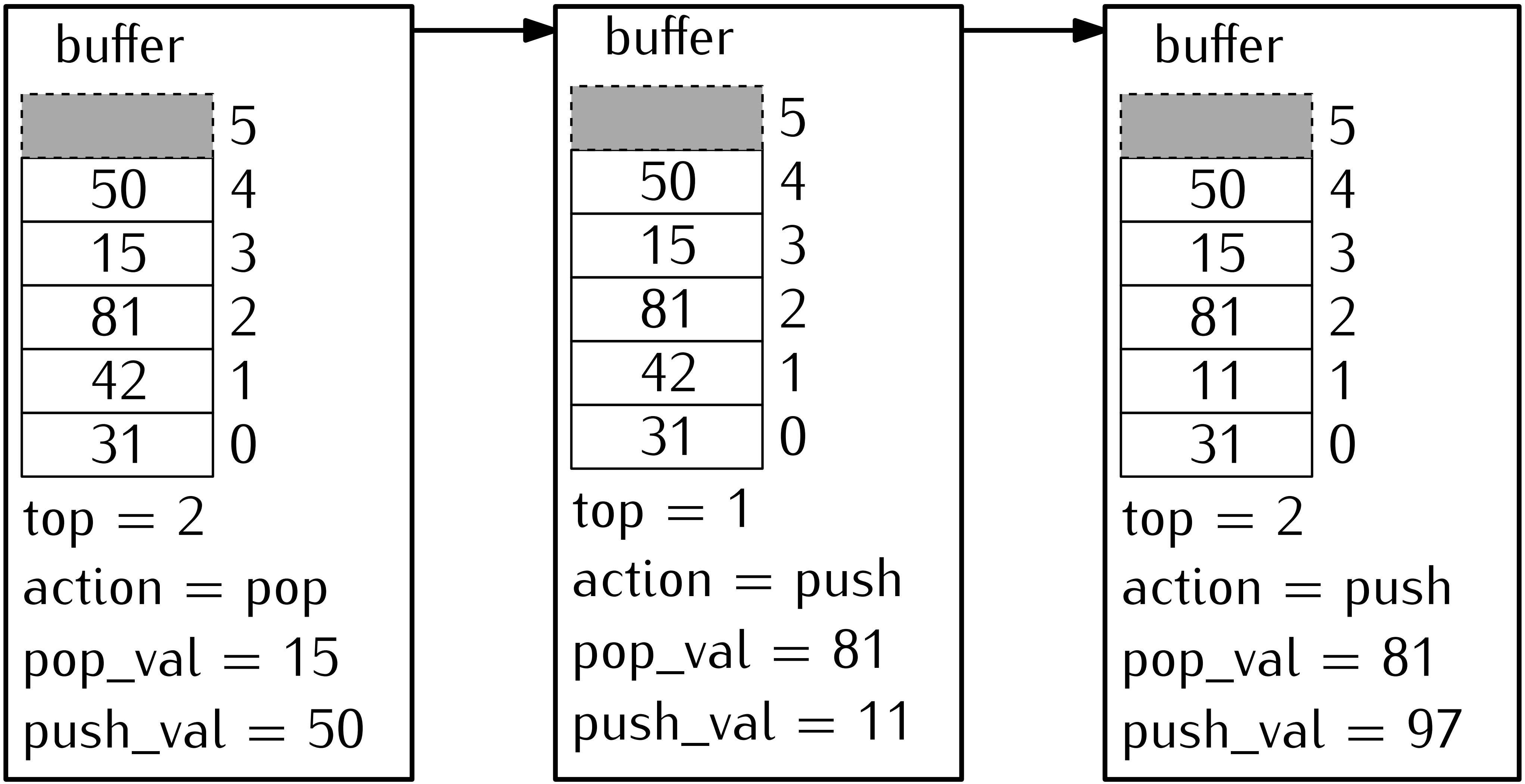
# Modeling a Stack in $\nu$SMV

What do the states and transitions look like?



# Modeling a Stack in $\nu$SMV

What do the states and transitions look like?




Our $\nu$SMV model should define all possible state transitions.

# Modeling a Stack in $\nu$SMV

Overview

# Modeling a Stack in *v*SMV

Overview

action = { push, pop }

# Modeling a Stack in *v*SMV

Overview

action = { push, pop }

next(top) := increment or decrement depending on push or pop. Keep top in the correct range.

# Modeling a Stack in *v*SMV

Overview

action = { push, pop }

next(top) := increment or decrement depending on push or pop. Keep top in the correct range.

next(pop_val) := the value under the stack pointer if the current action is pop and the stack is not empty. Otherwise NULL.

# Modeling a Stack in *ν*SMV

Overview

action = { push, pop }

next(top) := increment or decrement depending on push or pop. Keep top in the correct range.

next(pop_val) := the value under the stack pointer if the current action is pop and the stack is not empty. Otherwise NULL.

next(buffer) := update a location in the buffer based on top if action = push. If action = pop, no update.

# Modeling a Stack in $\nu$SMV

# Modeling a Stack in $\nu$SMV

```
#define SIZE 5

MODULE main
  VAR
    pop_val : {NULL, x, y, z};
    push_val : {x,y,z};
    action : {push, pop};
    s : stack(action, push_val, pop_val);
```

# Modeling a Stack in *v*SMV

```
#define SIZE 5

MODULE main
  VAR
    pop_val : {NULL, x, y, z};
    push_val : {x,y,z};
    action : {push, pop};
    s : stack(action, push_val, pop_val);
```

Why $\{x, y, z\}$?

# Modeling a Stack in $\nu$SMV

```
#define SIZE 5

MODULE main
  VAR
    pop_val : {NULL, x, y, z};
    push_val : {x,y,z};
    action : {push, pop};
    s : stack(action, push_val, pop_val);
```

Why $\{x, y, z\}$?

We have abstracted away the type of the stack. This abstraction will allow us to make statements about three distinct values in the stack, without worrying about what they are.

# Modeling a Stack in *ν*SMV

```
MODULE stack(action, push_val, pop_val)
  VAR
    top : 0 .. SIZE;
    buffer : array 0 .. SIZE - 1 of {NULL, x, y, z};

  DEFINE
    full := top = SIZE;
    empty := top = 0;

  ASSIGN
    init(top) := 0;

    next(top) :=
      case
        (action = push) & (top < SIZE)  : top + 1;
        (action = pop) & (top > 0) : top - 1;
        TRUE : top;
      esac;

    next(pop_val) :=
      case
        action = pop & !empty : buffer[top];
        TRUE : NULL;
      esac;
```

# Modeling a Stack in *ν*SMV

How to update the state of the buffer?

We'd like to write something like `next(buffer) := ??`

In *ν*SMV, we have to update individual array elements.

# Modeling a Stack in *v*SMV

How to update the state of the buffer?

We'd like to write something like `next(buffer) := ??`

In *v*SMV, we have to update individual array elements.

```
next(buffer[3]) :=                         ?               :
```

conditional test then-exp else-exp

# Modeling a Stack in *v*SMV

How to update the state of the buffer?

We'd like to write something like `next(buffer) := ??`

In *v*SMV, we have to update individual array elements.

```
next(buffer[3]) := action = push & top = 3 ?  push_val :  buffer[3]
```

conditional test then-exp else-exp

# Modeling a Stack in *ν*SMV

How to update the state of the buffer?

We'd like to write something like `next(buffer) := ??`

In *ν*SMV, we have to update individual array elements.

```
next(buffer[3]) := action = push & top = 3 ?  push_val :  buffer[3]
```

conditional test then-exp else-exp

# Modeling a Stack in *v*SMV

How to update the state of the buffer?

We'd like to write something like `next(buffer) := ??`

In *v*SMV, we have to update individual array elements.

```
next(buffer[3]) := action = push & top = 3 ?  push_val :  buffer[3]
```

conditional test then-exp else-exp

```
next(buffer[0]) := top = 0 & action = push ? push_val : buffer[0];
next(buffer[1]) := top = 1 & action = push ? push_val : buffer[1];
next(buffer[2]) := top = 2 & action = push ? push_val : buffer[2];
next(buffer[3]) := top = 3 & action = push ? push_val : buffer[3];
next(buffer[4]) := top = 4 & action = push ? push_val : buffer[4];
```

# Stack Properties to Verify

$G(0 \leq \texttt{s.top} \wedge \texttt{s.top} \leq \texttt{SIZE})$

LTL property: the stack pointer is always in the correct range.

# Stack Properties to Verify

$G(0 \leq \texttt{s.top} \wedge \texttt{s.top} \leq \texttt{SIZE})$

LTL property: the stack pointer is always in the correct range.

$G\ \neg(\texttt{s.full}\ \wedge\ \texttt{s.empty})$

LTL property: the stack is never empty and full at the same time.

# Stack Properties to Verify

$G(0 \leq \texttt{s.top} \wedge \texttt{s.top} \leq \texttt{SIZE})$

LTL property: the stack pointer is always in the correct range.

$G\ \neg(\texttt{s.full}\ \wedge\ \texttt{s.empty})$

LTL property: the stack is never empty and full at the same time.

$AG\ (\texttt{s.full}\ \rightarrow\ EF\ \texttt{s.empty})$

# Stack Properties to Verify

$G(0 \leq \texttt{s.top} \wedge \texttt{s.top} \leq \texttt{SIZE})$

LTL property: the stack pointer is always in the correct range.

$G\ \neg(\texttt{s.full}\ \wedge\ \texttt{s.empty})$

LTL property: the stack is never empty and full at the same time.

$AG\ (\texttt{s.full}\ \rightarrow\ EF\ \texttt{s.empty})$

CTL property: For all possible system states, if the stack is full, then it is possible that the stack is eventually empty.

# Stack Properties to Verify

The most important property of a stack:

# Stack Properties to Verify

The most important property of a stack:

Last In, First Out (LIFO)

This will be part of your next HW.

# Coming up with specifications

# Coming up with specifications

Karl Popper: The Logic of Scientific Discovery

> “My proposal is based on an *asymmetry* between verifiability and falsifiability; an asymmetry which results from the logical form of universal statements. For these are never derivable from singular statements, but can be contradicted by singular statements."

Edsger Dijkstra

> Program testing can be used to show the presence of bugs, but never to show their absence!

# Coming up with specifications

Titus Bartik, et. al.: Designing for Dystopia: Software Engineering Research for the Post-apocalypse. (FSE 2016)

# Coming up with specifications

Titus Bartik, et. al.: Designing for Dystopia: Software Engineering Research for the Post-apocalypse. (FSE 2016)

Literary theorists have long recognized the trade-offs in optimistic and pessimistic thinking through utopias and dystopias.

Research suggests that scientists are overwhelmingly optimistic, and subject to the effect of optimism bias [1].

Software engineering researchers have a tendency to be optimistic.

Though useful, optimism bias bolsters unrealistic expectations towards desirable outcomes.

Framing software engineering research through dystopias mitigates optimism bias and engender more diverse and thought-provoking research directions.

[1] D. A. Armor and S. E. Taylor. When predictions fail: The dilemma of unrealistic optimism.

# Coming up with specifications

In class activity:

1. Come up with one or two interesting stack properties that would be important to verify. Write it down as a legible English sentence.

2. In a group of two or three, swap properties. Identitfy if the property is LTL or CTL. Translate the property to LTL or CTL.

3. Regroup and discuss your properties and translations.

4. Choose one to write on the board (both in English and CTL / LTL) to explain to the rest of the class.

# Coming up with specifications

Some hints:

What could go wrong? Negate that property.

What should go right? Assert that property.

What should happen if `push x` is followed directly by `pop`?

What should happen if we try to pop an empty stack?

Our examples from earlier:

$$G(0 \leq \texttt{s.top} \wedge \texttt{s.top} \leq SIZE)$$

$$G\ \neg(\texttt{s.full}\ \wedge\ \texttt{s.empty})$$

$$AG\ (\texttt{s.full}\ \rightarrow\ EF\ \texttt{s.empty})$$

# Future homework

Translate and verify some stack properties.

Model and verify a queue.

# Equivalence of properties

Given two temporal logic formulas $\alpha$ and $\beta$, when can we say that the two formulas are equivalent?

# Equivalence of properties

Given two temporal logic formulas $\alpha$ and $\beta$, when can we say that the two formulas are equivalent?

We say that $\alpha \equiv \beta$ iff

# Equivalence of properties

Given two temporal logic formulas $\alpha$ and $\beta$, when can we say that the two formulas are equivalent?

We say that $\alpha \equiv \beta$ iff

$$\forall \mathcal{M} \quad (\mathcal{M} \models \alpha \Leftrightarrow \mathcal{M} \models \beta)$$

# Equivalence of properties

Given two temporal logic formulas $\alpha$ and $\beta$, when can we say that the two formulas are equivalent?

We say that $\alpha \equiv \beta$ iff

$$\forall \mathcal{M} \quad (\mathcal{M} \models \alpha \Leftrightarrow \mathcal{M} \models \beta)$$

We say that $\alpha \not\equiv \beta$ iff

# Equivalence of properties

Given two temporal logic formulas $\alpha$ and $\beta$, when can we say that the two formulas are equivalent?

We say that $\alpha \equiv \beta$ iff

$$\forall \mathcal{M} \quad (\mathcal{M} \models \alpha \Leftrightarrow \mathcal{M} \models \beta)$$

We say that $\alpha \not\equiv \beta$ iff

$$\exists \mathcal{M} \quad ((\mathcal{M} \models \alpha \wedge \mathcal{M} \not\models \beta) \vee (\mathcal{M} \not\models \alpha \wedge \mathcal{M} \models \beta))$$

# Equivalence of properties

Given two temporal logic formulas $\alpha$ and $\beta$, when can we say that the two formulas are equivalent?

We say that $\alpha \equiv \beta$ iff

$$\forall \mathcal{M} \quad (\mathcal{M} \models \alpha \Leftrightarrow \mathcal{M} \models \beta)$$

We say that $\alpha \not\equiv \beta$ iff

$$\exists \mathcal{M} \quad ((\mathcal{M} \models \alpha \wedge \mathcal{M} \not\models \beta) \vee (\mathcal{M} \not\models \alpha \wedge \mathcal{M} \models \beta))$$

In words, two formulas are not equivalent if we can find a transition system that satisifes one formula but not the other.

## Showing that $\alpha \not\equiv \beta$

Consider these two temporal formulas

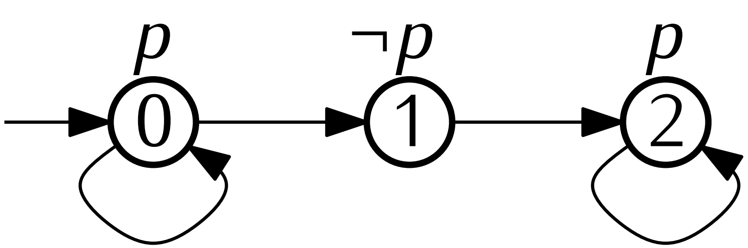
$F\ G\ p$ $\qquad$ $AF\ AG\ p$

# Showing that $\alpha \not\equiv \beta$

Consider these two temporal formulas

$F\ G\ p$ $\qquad\qquad$ $AF\ AG\ p$

Consider this transition system, $\mathcal{M}$:

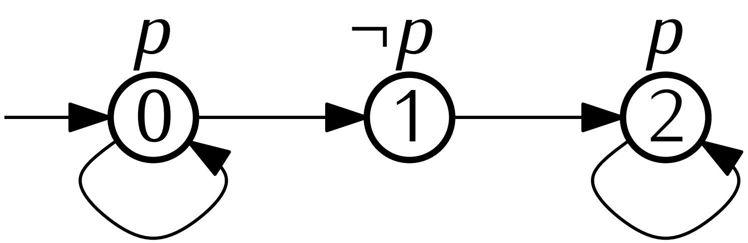


# Showing that $\alpha \not\equiv \beta$

Consider these two temporal formulas

$F\ G\ p$ $\qquad\qquad$ $AF\ AG\ p$

Consider this transition system, $\mathcal{M}$:




Paths of $\mathcal{M}$ look like:

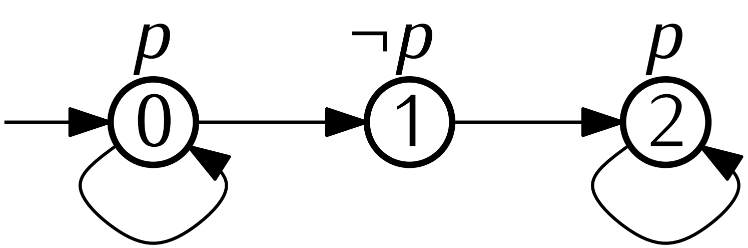
$0^\omega$ $\quad$ or $\quad$ $0^*1\ 2^\omega$

# Showing that $\alpha \not\equiv \beta$

Consider these two temporal formulas

$F\ G\ p$ $\qquad\qquad\qquad\qquad$ $AF\ AG\ p$

Consider this transition system, $\mathcal{M}$:




Paths of $\mathcal{M}$ look like:

$0^\omega$ or $0^*1\ 2^\omega$

Sequences of propositions:

$p, p, p, p, p, \ldots$

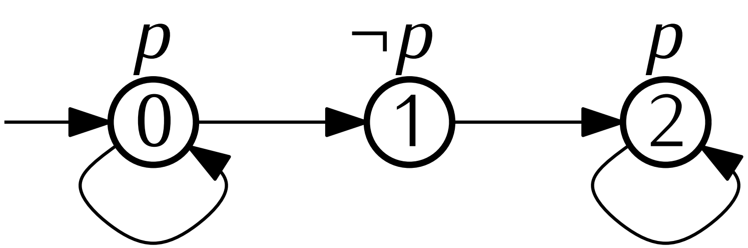
$p, p, p, \ldots, \neg p, p, p, p, \ldots$

# Showing that $\alpha \not\equiv \beta$

Consider these two temporal formulas

$F\ G\ p$ $\qquad\qquad$ $AF\ AG\ p$

Consider this transition system, $\mathcal{M}$:




Paths of $\mathcal{M}$ look like:

$0^\omega$ or $0^*1\ 2^\omega$

Sequences of propositions:

$p, p, p, p, p, \ldots$

$p, p, p, \ldots, \neg p, p, p, p, \ldots$

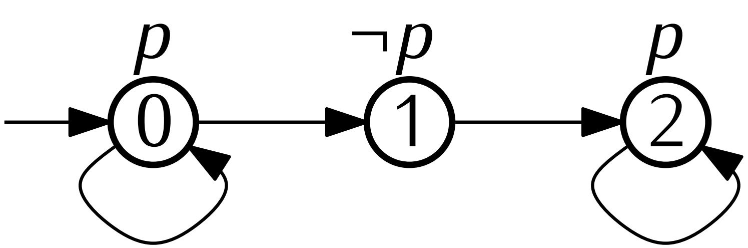
$$\mathcal{M} \models F\ G\ p$$

# Showing that $\alpha \not\equiv \beta$

Consider these two temporal formulas

$F\ G\ p$ $\qquad\qquad$ $AF\ AG\ p$

Consider this transition system, $\mathcal{M}$:




Computation tree:

Paths of $\mathcal{M}$ look like:

$0^{\omega}$ or $0^{*}1\ 2^{\omega}$

Sequences of propositions:

$p, p, p, p, p, \ldots$

$p, p, p, \ldots, \neg p, p, p, p, \ldots$

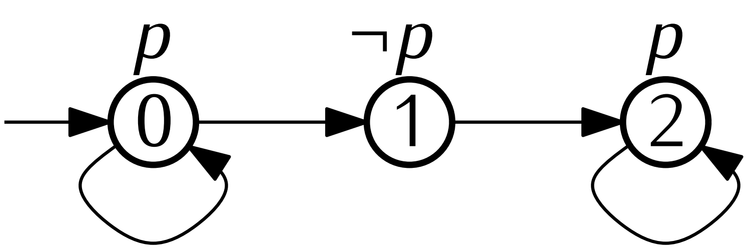
$\mathcal{M} \models F\ G\ p$

# Showing that $\alpha \not\equiv \beta$

Consider these two temporal formulas

$F\ G\ p$ $\qquad\qquad$ $AF\ AG\ p$

Consider this transition system, $\mathcal{M}$:




Computation tree:


Paths of $\mathcal{M}$ look like:

$0^\omega$ $\quad$ or $\quad$ $0^*1\ 2^\omega$

Sequences of propositions:

$p, p, p, p, p, \ldots$

$p, p, p, \ldots, \neg p, p, p, p, \ldots$

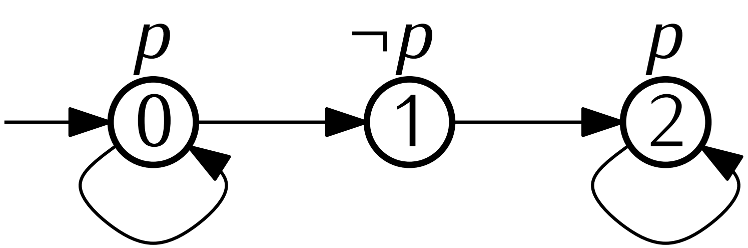
$$\mathcal{M} \models F\ G\ p$$

# Showing that $\alpha \not\equiv \beta$

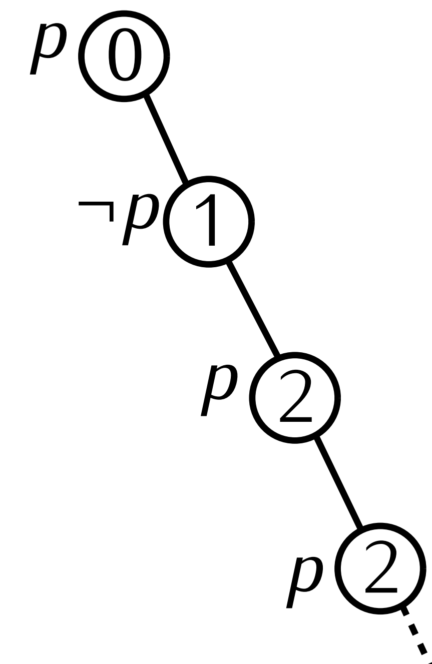
Consider these two temporal formulas

$F\ G\ p$ $\qquad\qquad$ $AF\ AG\ p$

Consider this transition system, $\mathcal{M}$:




Computation tree:




Paths of $\mathcal{M}$ look like:

$0^\omega$ $\quad$ or $\quad$ $0^*1\ 2^\omega$

Sequences of propositions:

$p, p, p, p, p, \ldots$

$p, p, p, \ldots, \neg p, p, p, p, \ldots$

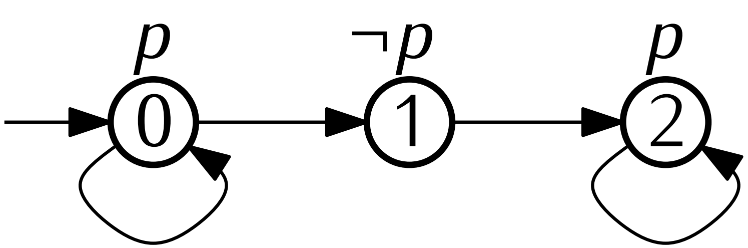
$$\mathcal{M} \models\ F\ G\ p$$

# Showing that $\alpha \not\equiv \beta$

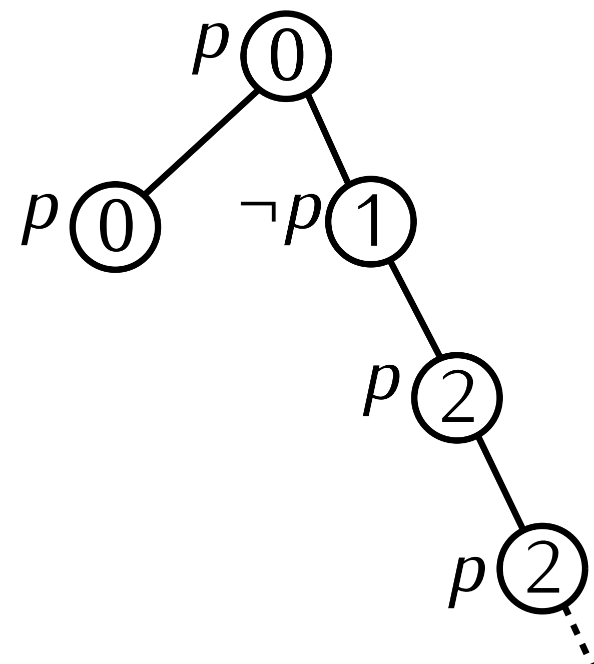
Consider these two temporal formulas

$F\ G\ p$ $\qquad$ $AF\ AG\ p$

Consider this transition system, $\mathcal{M}$:




Computation tree:




Paths of $\mathcal{M}$ look like:

$0^{\omega}$ or $0^{*}1\ 2^{\omega}$

Sequences of propositions:

$p, p, p, p, p, \ldots$

$p, p, p, \ldots, \neg p, p, p, p, \ldots$

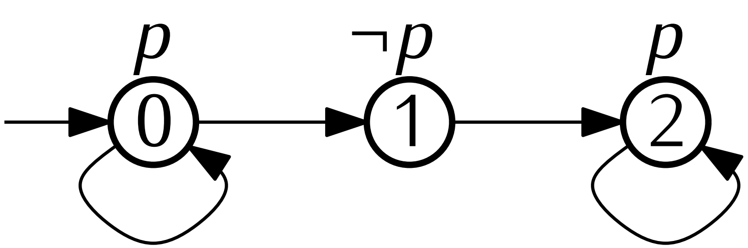
$\mathcal{M} \models\ F\ G\ p$

# Showing that $\alpha \not\equiv \beta$

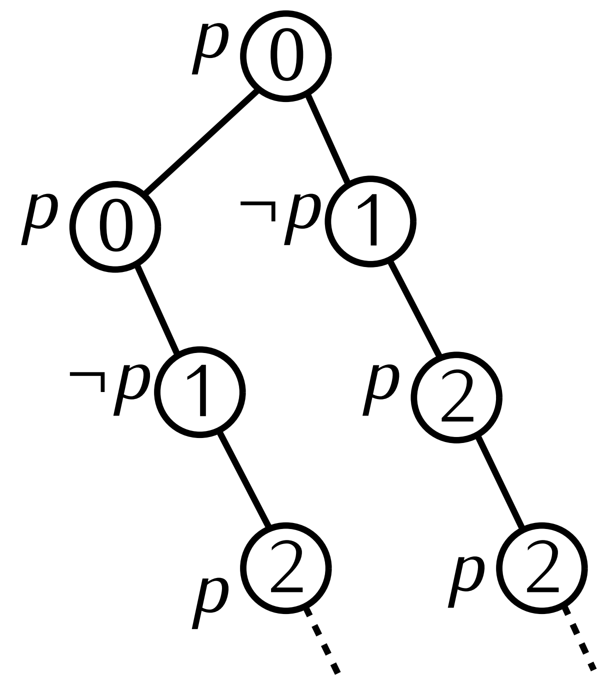
Consider these two temporal formulas

$F\ G\ p$ $AF\ AG\ p$

Consider this transition system, $\mathcal{M}$:




Computation tree:




Paths of $\mathcal{M}$ look like:

$0^{\omega}$ or $0^{*}1\ 2^{\omega}$

Sequences of propositions:

$p, p, p, p, p, \ldots$

$p, p, p, \ldots, \neg p, p, p, p, \ldots$

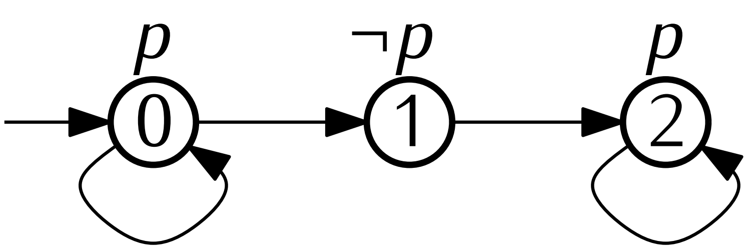
$$\mathcal{M} \models F\ G\ p$$

# Showing that $\alpha \not\equiv \beta$

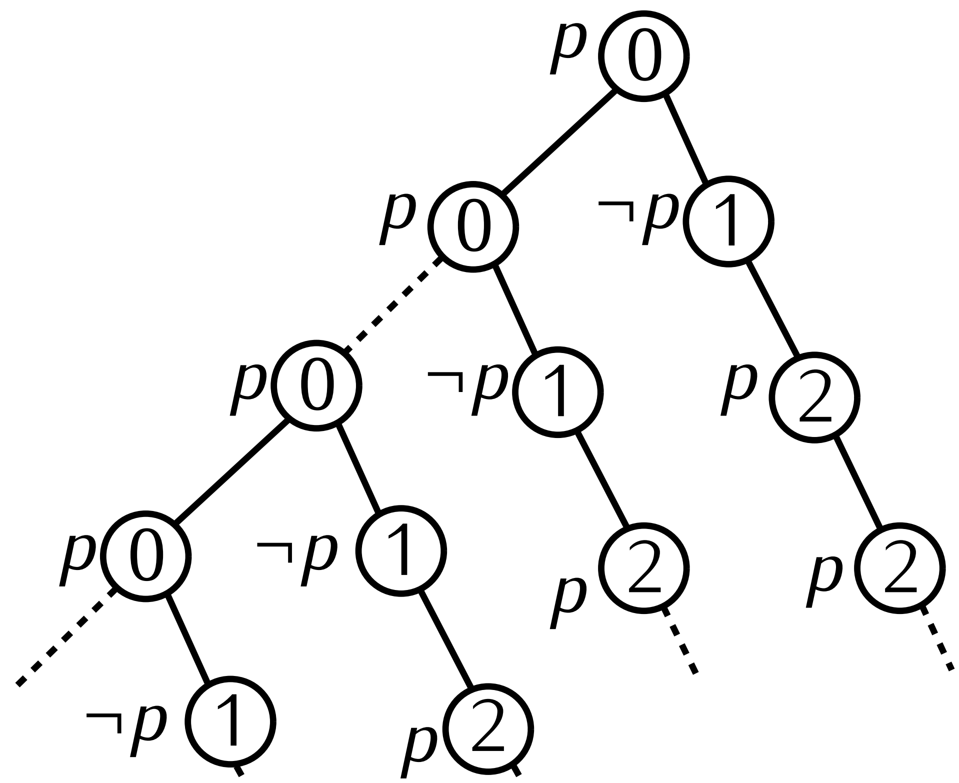
Consider these two temporal formulas

$F\ G\ p$ $\qquad\qquad$ $AF\ AG\ p$

Consider this transition system, $\mathcal{M}$:




Computation tree:




Paths of $\mathcal{M}$ look like:

$0^\omega$ or $0^*1\ 2^\omega$

Sequences of propositions:

$p, p, p, p, p, \ldots$

$p, p, p, \ldots, \neg p, p, p, p, \ldots$

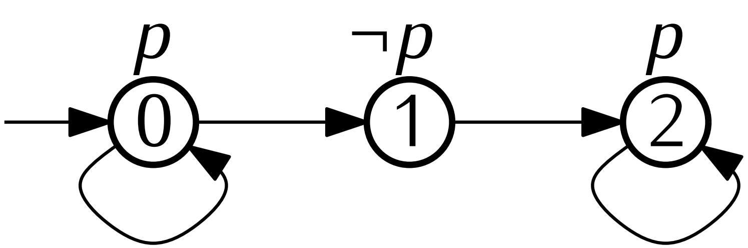
$\mathcal{M} \models\ F\ G\ p$

# Showing that $\alpha \not\equiv \beta$

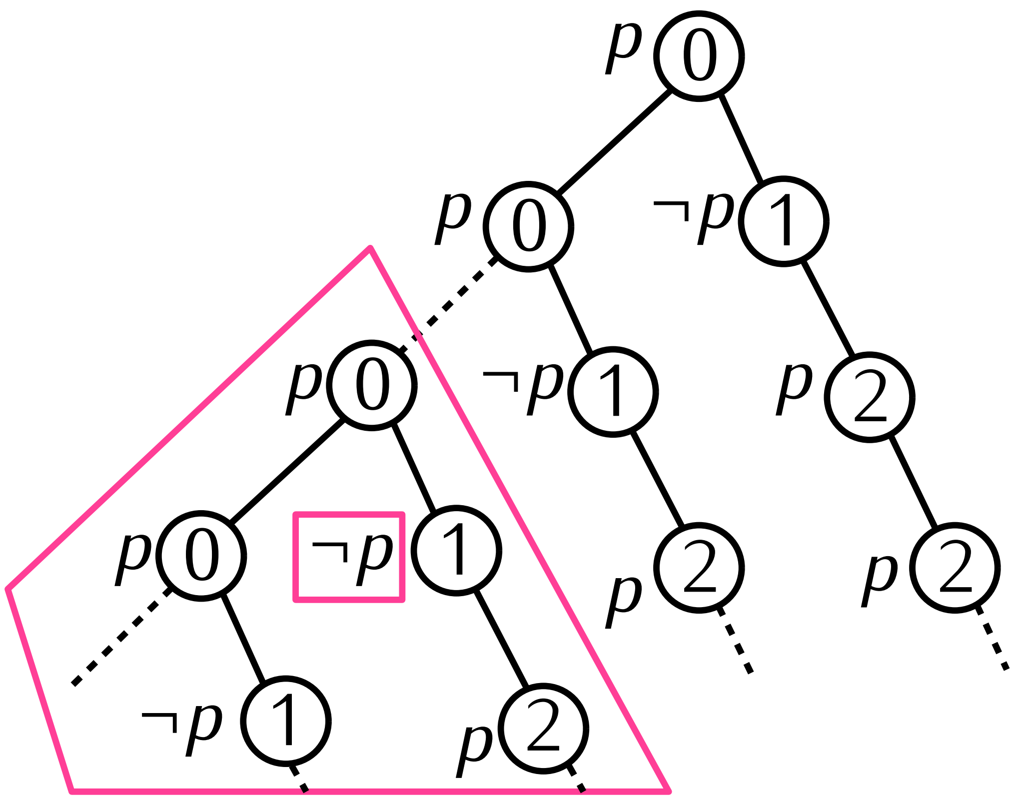
Consider these two temporal formulas

$F\ G\ p$ $\qquad\qquad$ $AF\ AG\ p$

Consider this transition system, $\mathcal{M}$:




Computation tree:




Paths of $\mathcal{M}$ look like:

$0^\omega$ $\quad$ or $\quad$ $0^* 1\ 2^\omega$

Sequences of propositions:

$p, p, p, p, p, \ldots$

$p, p, p, \ldots, \neg p, p, p, p, \ldots$

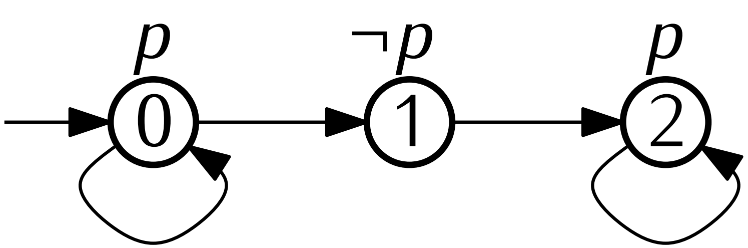
$\mathcal{M} \models\ F\ G\ p$

# Showing that $\alpha \not\equiv \beta$

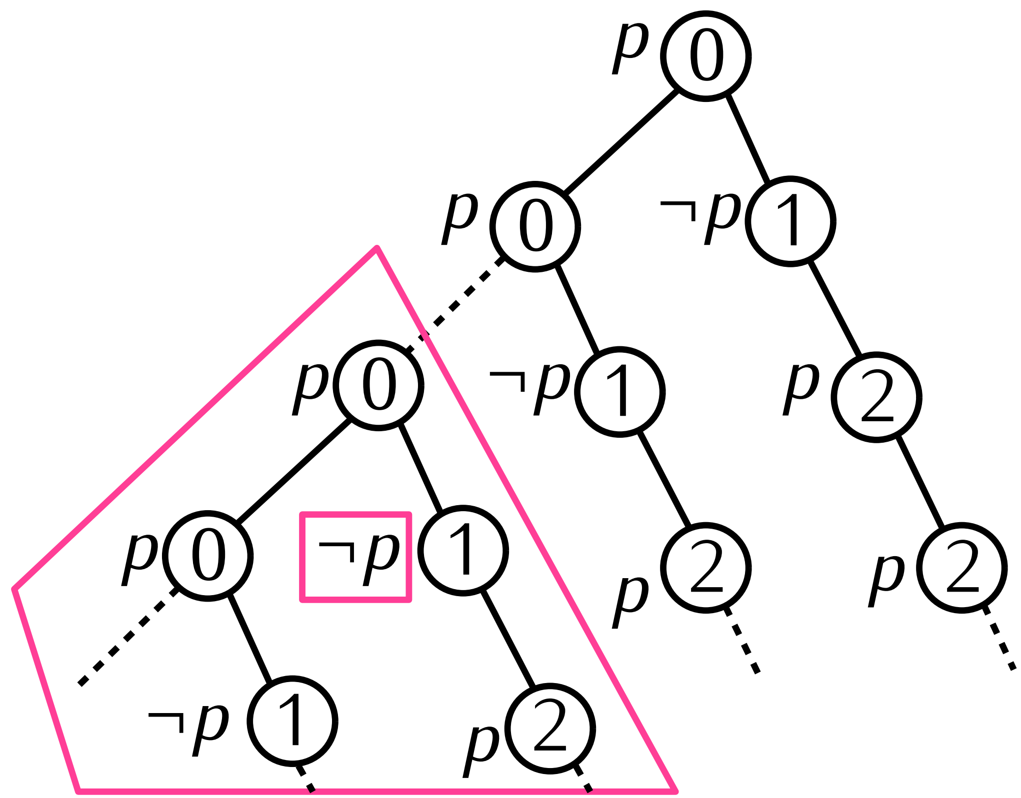
Consider these two temporal formulas

$F\ G\ p$ $\qquad\qquad$ $AF\ AG\ p$

Consider this transition system, $\mathcal{M}$:




Computation tree:




Paths of $\mathcal{M}$ look like:

$0^\omega$ $\quad$ or $\quad$ $0^*1\ 2^\omega$

Sequences of propositions:

$p, p, p, p, p, \ldots$

$p, p, p, \ldots, \neg p, p, p, p, \ldots$

$$\mathcal{M} \models F\ G\ p$$

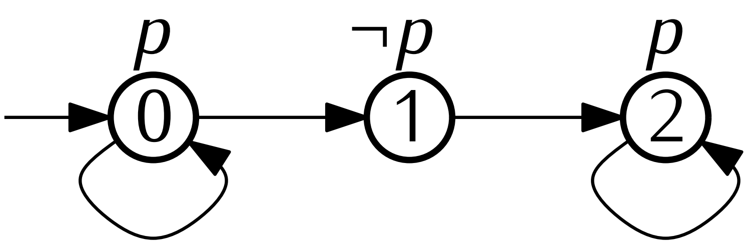
$$\mathcal{M} \not\models AF\ AG\ p$$

# Showing that $\alpha \not\equiv \beta$

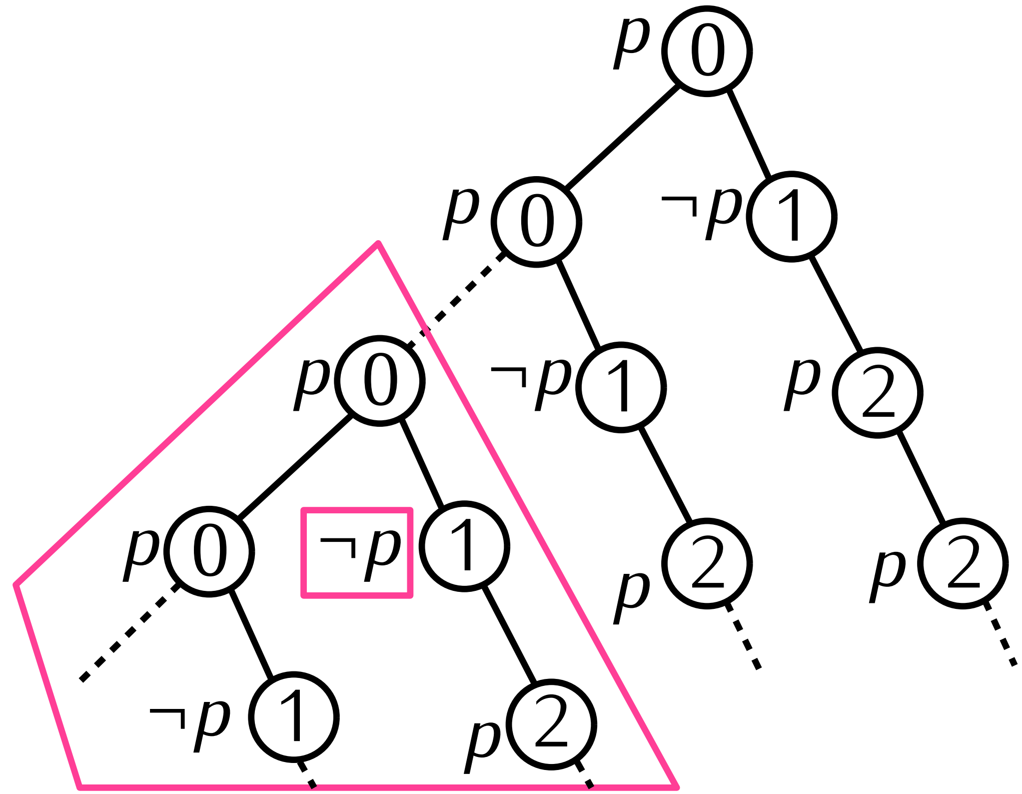
Consider these two temporal formulas

$F\ G\ p$ $\qquad\qquad$ $AF\ AG\ p$

Consider this transition system, $\mathcal{M}$:




Computation tree:




Paths of $\mathcal{M}$ look like:

$0^\omega$ or $0^*1\ 2^\omega$

Sequences of propositions:

$p, p, p, p, p, \ldots$

$p, p, p, \ldots, \neg p, p, p, p, \ldots$

$\mathcal{M} \models\ F\ G\ p$

$\mathcal{M} \not\models\ AF\ AG\ p$

On your HW: Show two formulas not equivalent.