Logic for Verification 3

Nisansala Yatapanage
ANU Logic Summer School

Today’s lecture

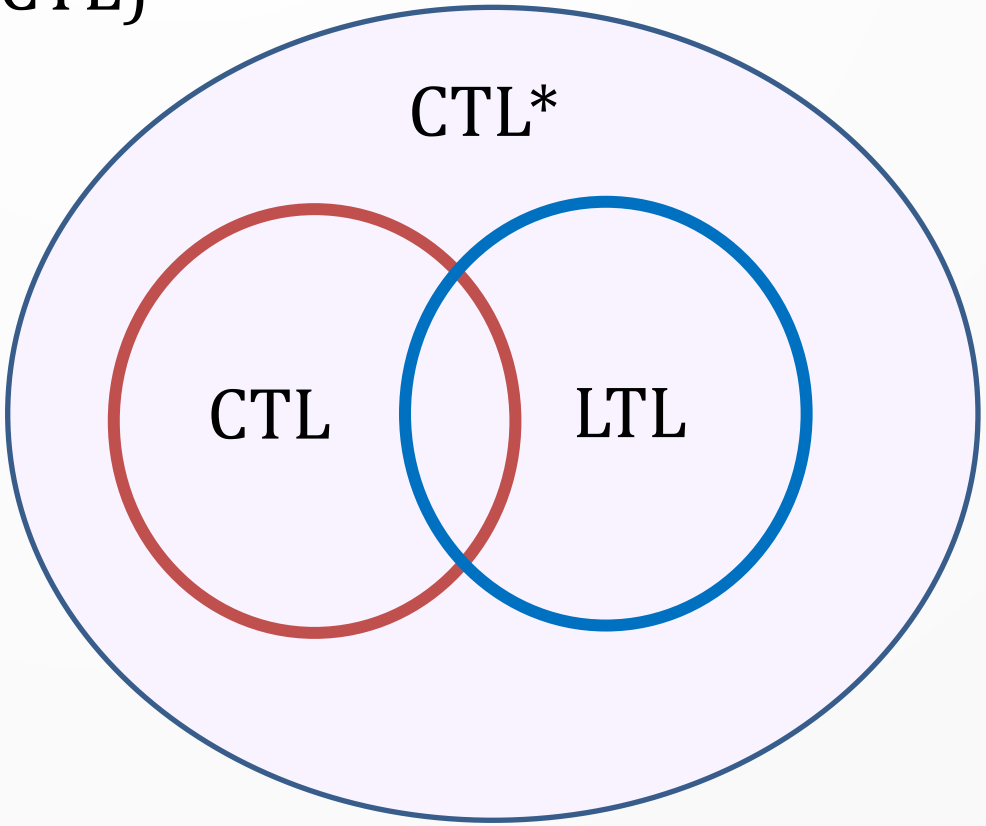
We’ll look at the following:

- LTL (linear temporal logic)
- CTL (computation tree logic)
- CTL*
- Transition systems
- Bisimulation

What is temporal logic?

Types of temporal logic include:

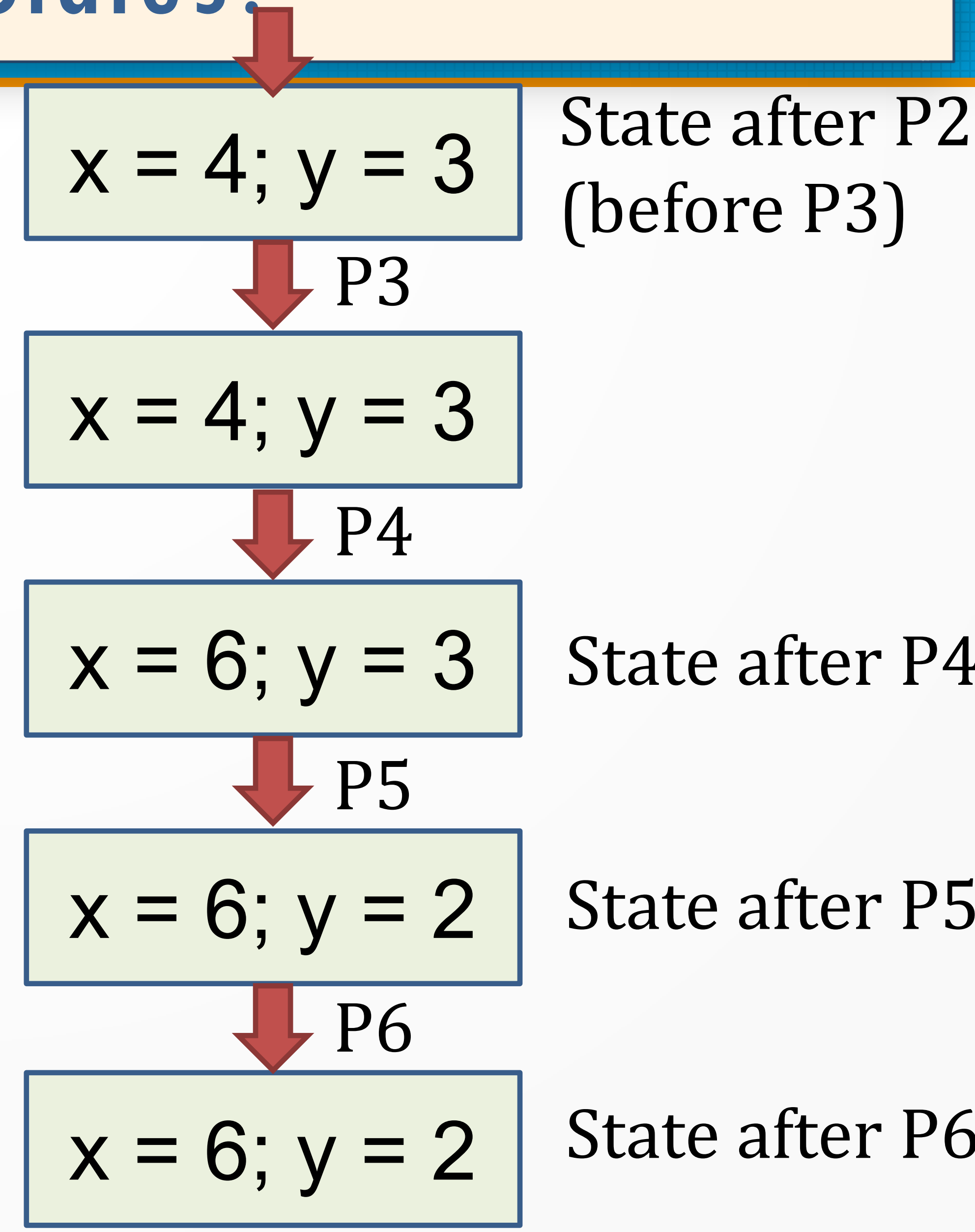
- ❑ Linear Temporal Logic (LTL)
- ❑ Computation Tree Logic (CTL)
- ❑ CTL*
- ❑ Lots of others too.



What are States?

Consider this program:

```
P1: int x = 4;
P2: int y = 3;
P3: if (x > 2){
P4:     x = x + 2;
    }
P5:  y = y – 1;
P6: System.out.println(y);
```



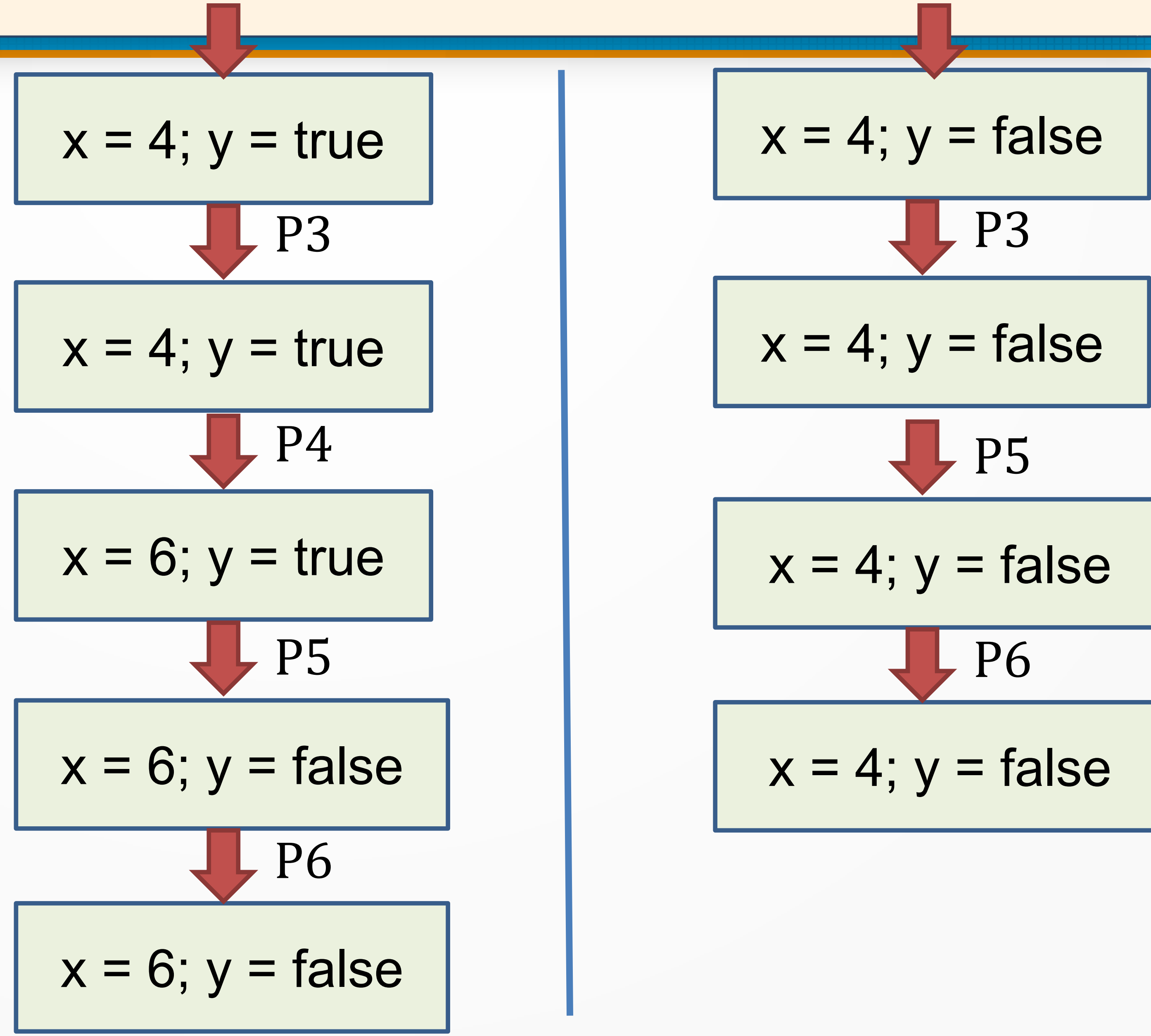
Note: These states are not really identical – the program counter values are different (not shown here).

What are States?

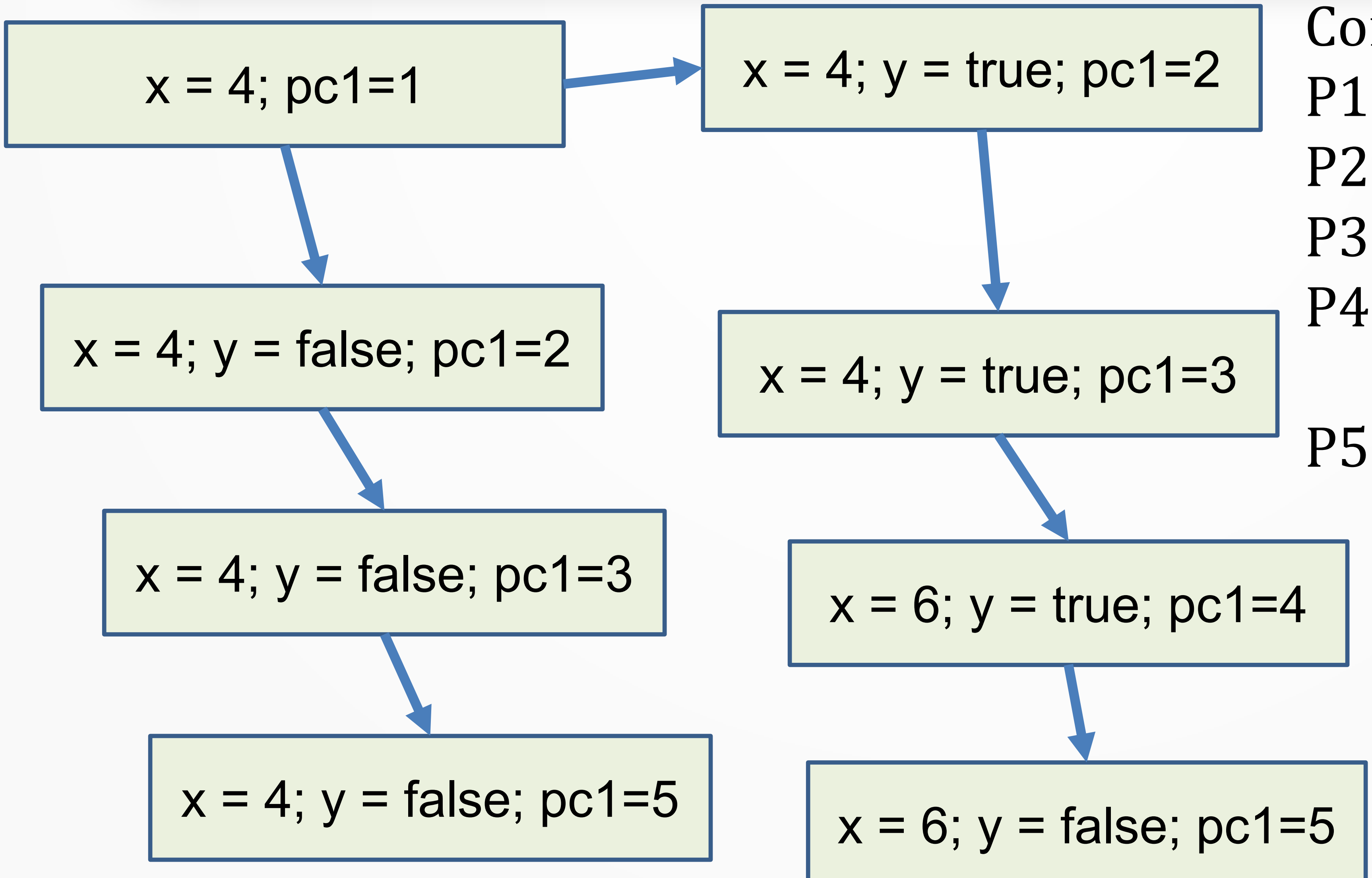
Consider this program:

```
P1: int x = 4
P2: boolean y = getInput();
P3: if (y){
P4:     x = x + 2;
    }
P5:  y = false;
P6: System.out.println(x);
```

This time there are two possible traces:



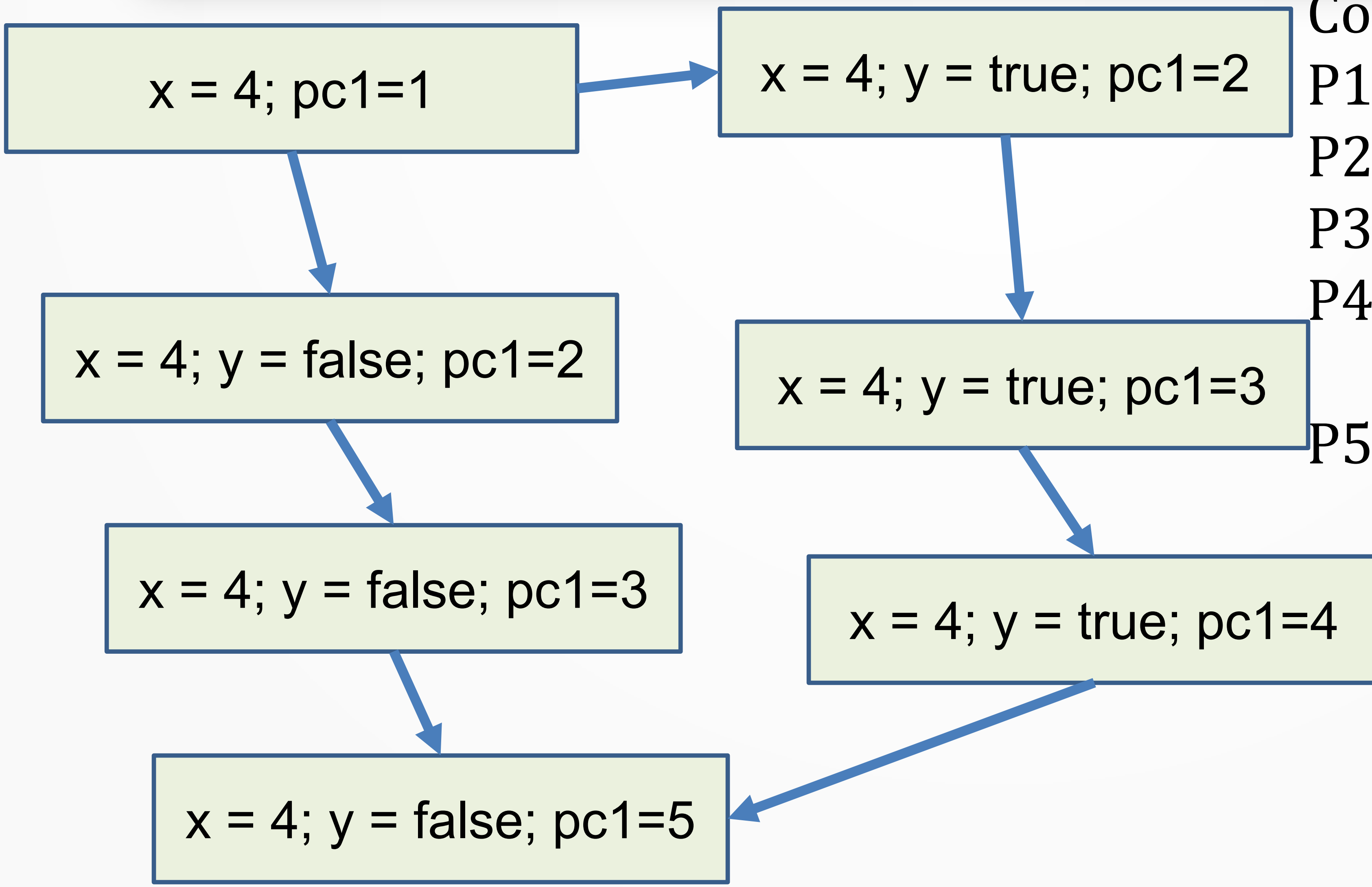
State Transition Diagram



Consider this program:

```
P1: int x = 4
P2: boolean y = getInput();
P3: if (y){
P4:     x = x + 2;
    }
P5:  y = false;
```

State Transition Diagram



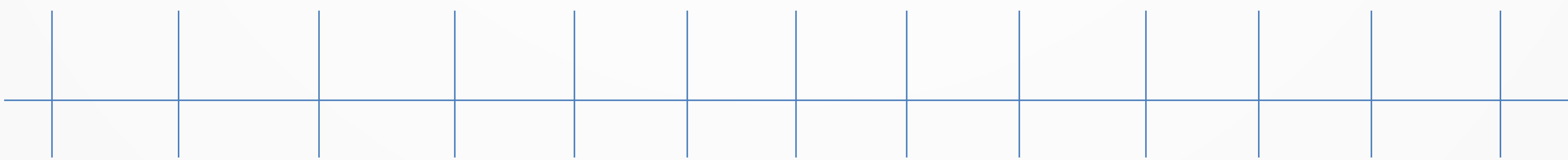
Consider this program:

```
P1: int x = 4
P2: boolean y = getInput();
P3: if (y){
P4:     System.out.println(x);
     }
P5:  y = false;
```

States

For non-temporal logics, we are thinking of the value just at one point, but for temporal logics, we need to think of the value across different points in time.

States s0 s1 s2 s3 s4 s5 and so on...



x = 0	x = 1	x = 2	x = 2	x = 3	and so on...
y = 0	y = 0	y = 0	y = 1	y = 1	

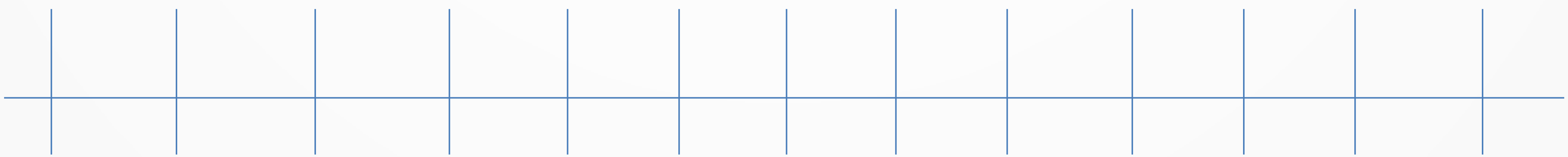
Simple Formulas

Does the property x = 0 hold at state s0? **Yes**

Does the property x = 0 hold at state s2? **No**

Does the property (x = 2 AND y = 0) hold at state s2? **Yes**

s0 s1 s2 s3 s4 s5 and so on...



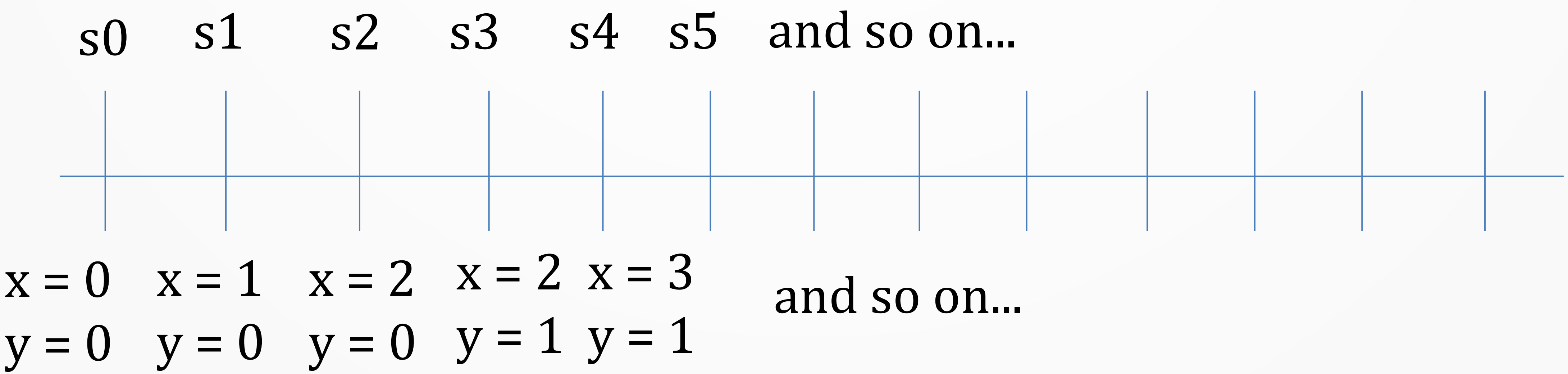
x = 0	x = 1	x = 2	x = 2	x = 3	and so on...
y = 0	y = 0	y = 0	y = 1	y = 1	

Simple Formulas

Does the property (x = 0) hold **on the execution trace starting at s0**?

This means the property must hold at **s0**. **Yes**

The rest of the states don't matter because there was no **G** operator (explained soon!)



Simple Formulas

Does the property ($x = 2$) hold **on the execution trace starting at s0**? **No**

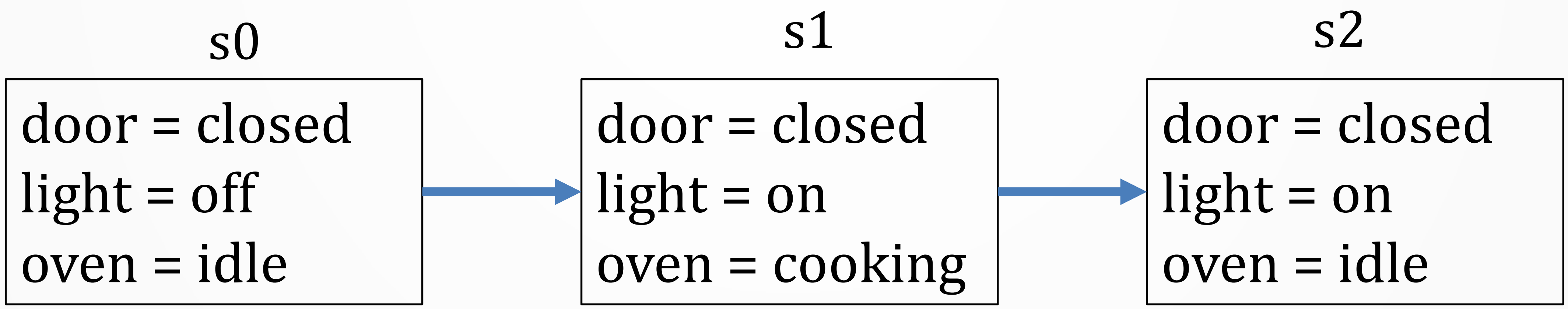
Does the property ($x = 0$ AND $y = 0$) hold **on the execution trace starting at s0**? **Yes**

Does the property ($x = 2$ AND $y = 0$) hold **on the execution trace starting at s0**? **No**

s0	s1	s2	s3	s4	s5	and so on...
$x = 0$ $y = 0$	$x = 1$ $y = 0$	$x = 2$ $y = 0$	$x = 2$ $y = 1$	$x = 3$ $y = 1$		and so on...

A Simple Case Study

Case study: A microwave oven.



Does (door = closed) hold on this transition system? **Yes**

Does (light = off) hold on this transition system? **Yes**

Does (light = on) hold on this transition system? **No**

Assume that it means starting at the first state, s0.

Safety Properties

A *labelled transition system* is a tuple (**S**, **I**, A,→), where **S** is a set of states, **I** is the set of initial states, A is a set of actions and →⊆ **S** x A x **S** is the transition relation.

A *Kripke Structure*, is a tuple T = (**S**, AP, **L**, **I**, →), where **S** is a set of states, AP is a set of atomic propositions, **L** is a labelling function which labels each state with the set of atomic propositions that hold in that state, **I** is a set of initial states and →⊆ **S** x **S** is the transition relation.

A *doubly-labelled transition system* labels both states and actions.

Safety Properties

A *labelled transition system* is a tuple (**S**, **I**, A,→), where **S** is a set of states, **I** is the set of initial states, A is a set of actions and →⊆ **S** x A x **S** is the transition relation.

A *Kripke Structure*, is a tuple T = (**S**, AP, **L**, **I**, →), where **S** is a set of states, AP is a set of atomic propositions, **L** is a labelling function which labels each state with the set of atomic propositions that hold in that state, **I** is a set of initial states and →⊆ **S** x **S** is the transition relation.

A *doubly-labelled transition system* labels both states and actions.

CTL* Properties

A CTL* *state formula* ψ is defined as follows, where $p \in AP$ is an atomic proposition, ψ_1 and ψ_2 are state formulas and φ is a *path* formula:

$$\psi = \mathit{true} \mid p \mid \psi_1 \land \psi_2 \mid !\, \psi_1 \mid E\varphi$$

A CTL* *path formula* is defined as follows, where φ1 and φ2 are path formulas and ψ is a state formula:

$$\varphi = \psi \mid \varphi_1 \land \varphi_2 \mid !\, \varphi_1 \mid X\varphi_1 \mid \varphi_1 \, U \, \varphi_2$$

CTL* Formulas

Let $T = (\mathbf{S}, AP, \mathbf{L}, \mathbf{I}, \rightarrow)$ be a transition system. A CTL* state formula ψ holds in a state $s \in \mathbf{S}$, denoted $T, s \models \psi$, or simply $s \models \psi$, according to the following, where ψ_1 and ψ_2 are CTL* state formulas and φ is a CTL* path formula:

$s \models true$,
$s \models a \in AP$ iff $a \in \mathbf{L}(s)$,
$s \models\ !\, \psi_1$ iff $s \not\models \psi_1$,
$s \models \psi_1 \land \psi_2$ iff $s \models \psi_1$ and $s \models \psi_2$,
$s \models \mathsf{E}\varphi$ iff there exists a path $\pi = < s_0, s_1, s_2, \ldots >$, such that $s_0 = s$ and $\pi \models \varphi$.

CTL* Formulas

A CTL* path formula φ holds for a path $\pi = < s_0, s_1, s_2, \ldots >$, denoted $\pi \vDash \varphi$, according to the following,
where φ_1 and φ_2 are CTL* path formulas and ψ_1 is a CTL* state formula:

$\pi \vDash \psi_1$ iff $s_0 \vDash \psi_1$,
$\pi \vDash \varphi_1 \wedge \varphi_2$ iff $\pi \vDash \varphi_1$ and $\pi \vDash \varphi_2$,
$\pi \vDash\ !\, \varphi_1$ iff $\pi \nvDash \varphi_1$,
$\pi \vDash X\varphi_1$ iff $\pi[s_1\ldots] \vDash \varphi_1$,
$\pi \vDash \varphi_1 \cup \varphi_2$ iff $\exists j > 0$ such that $\pi[s_j\ldots] \vDash \varphi_2$ and $\forall i$, where $0 \leq i < j$, $\pi[s_i\ldots] \vDash \varphi_1$.

CTL* Formulas

All LTL formulas implicitly hold over all paths.

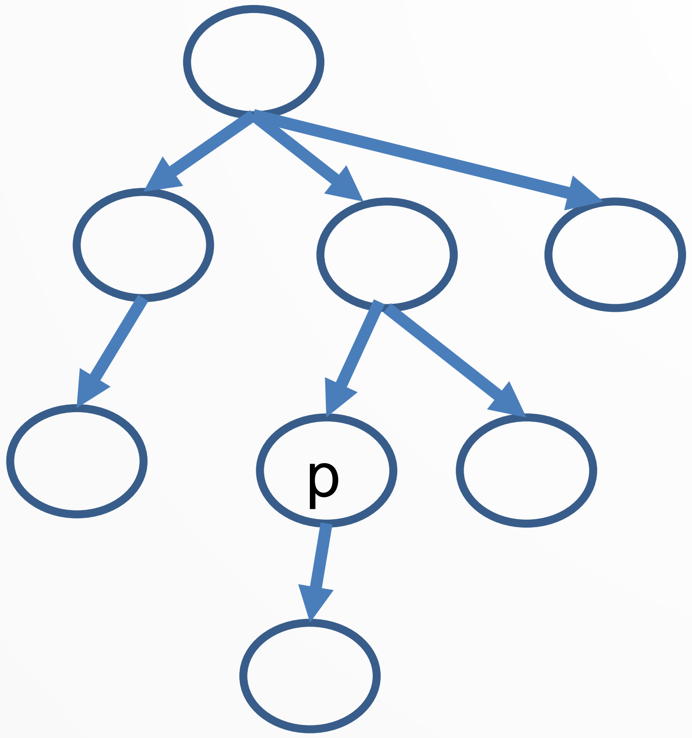
All CTL path operators have to be preceded by an A or E.

Some formulas can be expressed in LTL but not CTL and vice versa.

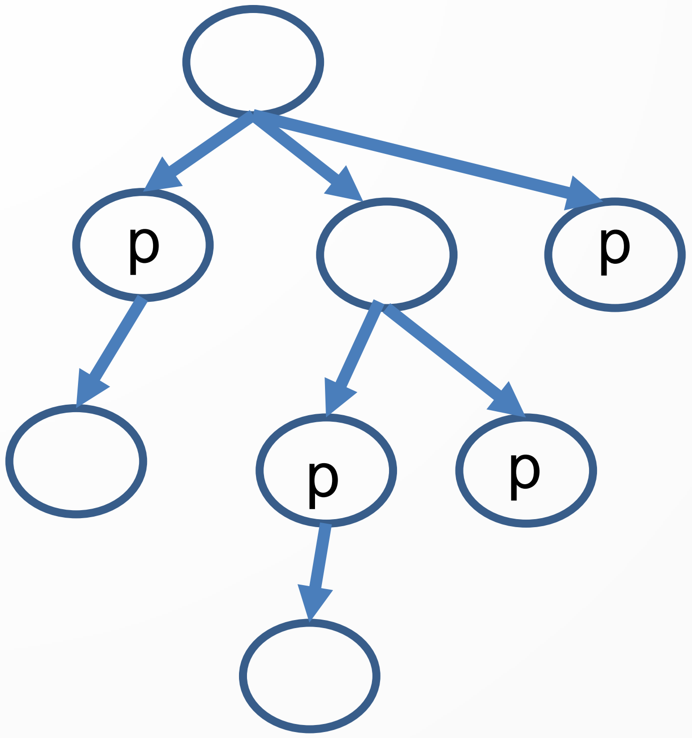
FGp (liveness) is a valid LTL formula but cannot be expressed in CTL.

$E(Xp \land XXq)$ can be expressed in CTL* but not in CTL.

CTL Formulas

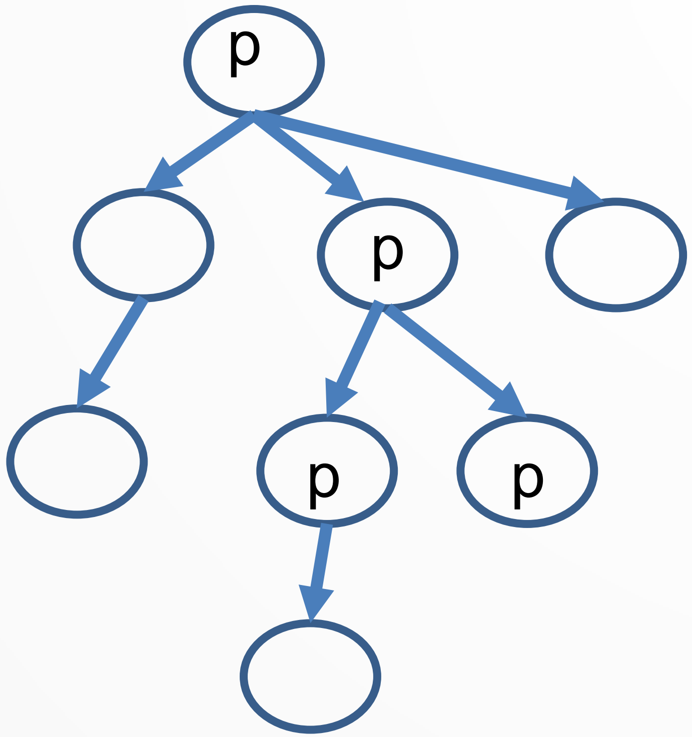


EFp

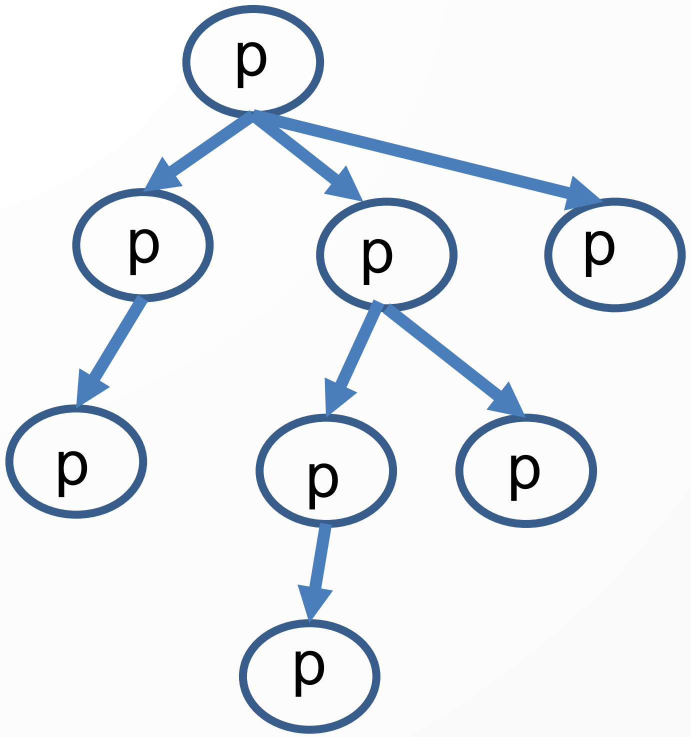


AFp

CTL Formulas



EGp



AGp

Global Operator

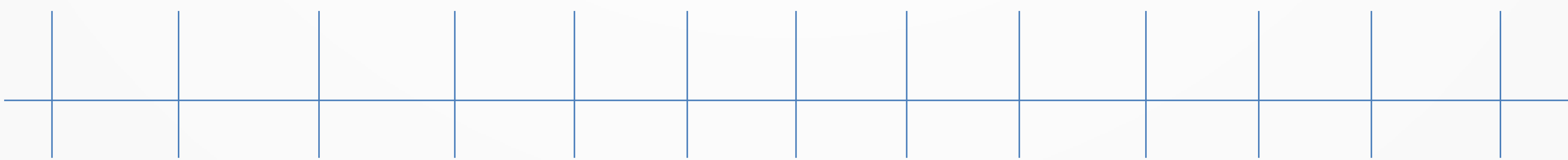
G means **G**lobally – holds on **all** states.

Does the property **G**(x = 0) hold on this execution trace? **No**

Does the property **G**(y < 2) hold on this execution trace? **Yes**

- If the question doesn't mention other states, assume it means s0.

s0 s1 s2 s3 s4 s5 and so on...



s0	s1	s2	s3	s4	
x = 0	x = 1	x = 2	x = 2	x = 3	and so on with no further changes to y.
y = 0	y = 0	y = 0	y = 1	y = 1	

A Simple Case Study

s0		s1		s2
door = closed light = off oven = idle	→	door = closed light = on oven = cooking	→	door = closed light = on oven = idle

Compare that to the answers if there is a **G** operator:

Does **G**(door = closed) hold on this transition system? **Yes**

Does **G**(light = off) hold on this transition system? **No**

Does **G**(light = on) hold on this transition system? **No**

A Simple Case Study

s0		s1		s2
door = closed light = off oven = idle	→	door = closed light = on oven = cooking	→	door = closed light = on oven = timeout

Does **G**((door = closed) AND (light = off)) hold on this transition system?

No

Counterexample: s0, s1.

This is because at s1 the property is violated.

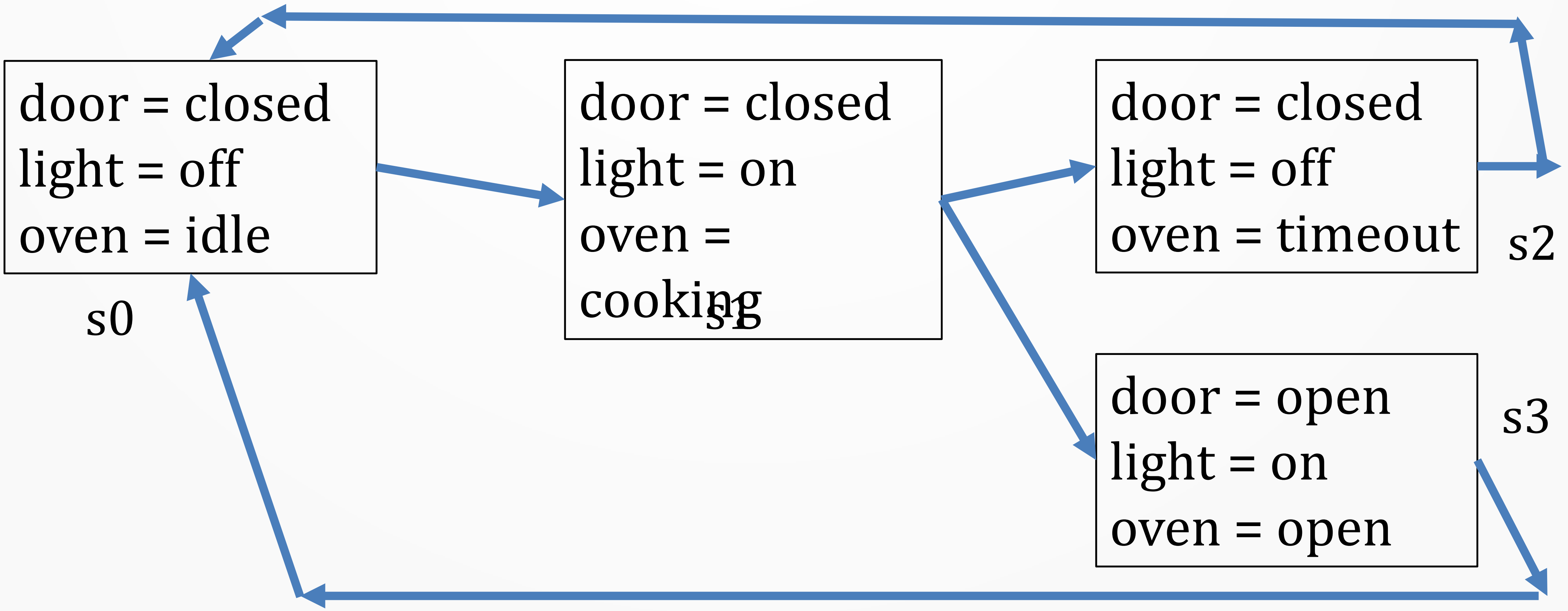
G means look at **all** the states.

It is also violated at s2 but the counterexample stops at the first violation.

Case Study

Each of those traces corresponds to a possible path in the transition system.

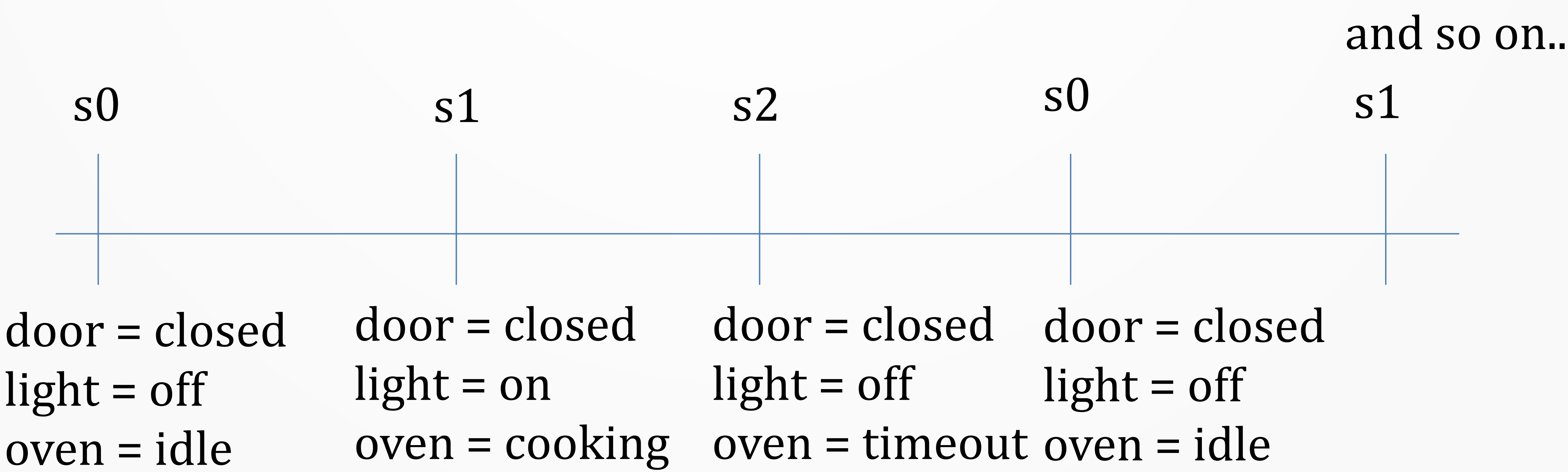
Here is a more complex transition system for the oven:



Execution Traces

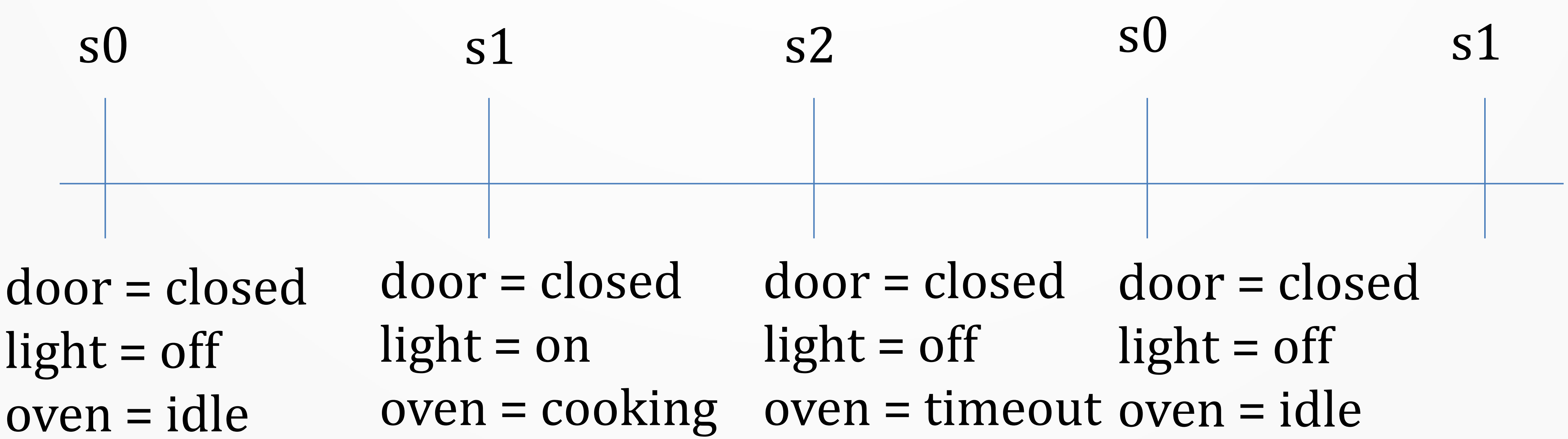
What are the possible execution traces?

Some are: s0, s1, s2, s0, ...
s0, s1, s3, s0, ...



Global Operator

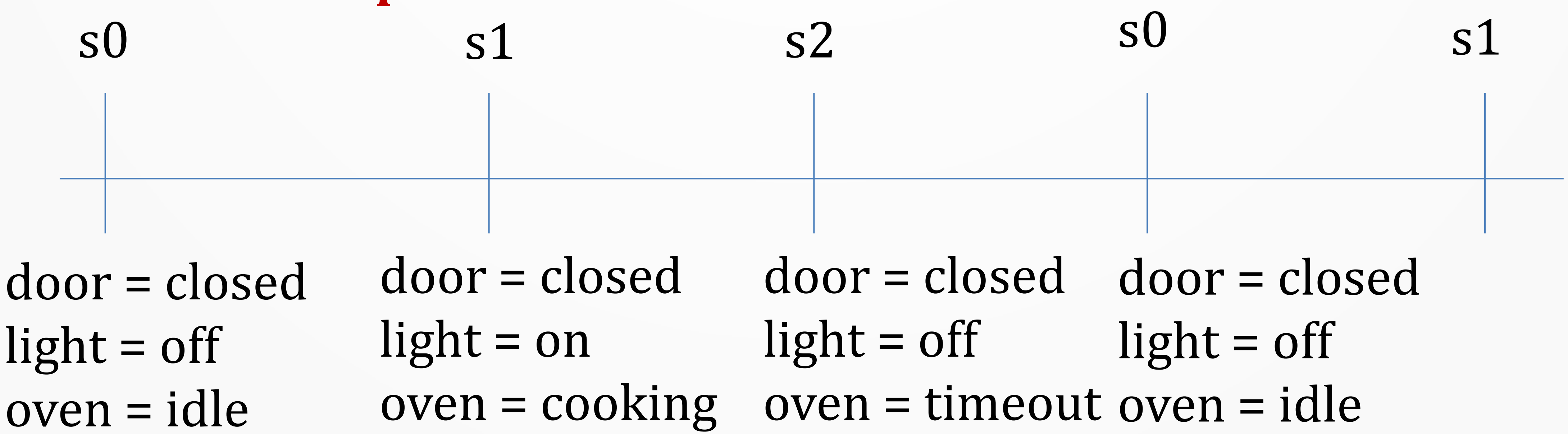
G(door = closed) holds on the execution trace below, but does it hold for **all** the possible traces?



Global Operator

When we ask: does **G**(door = closed) hold, we are talking about **all** the possible execution traces.

See whether you can find a trace where it **doesn't** hold – if no such trace exists, the property holds. Otherwise, it is false and that trace is a **counterexample**.



Global Operator

G(door = closed) holds on the execution trace below, but does it hold for **all** the possible traces?

What about this one below?

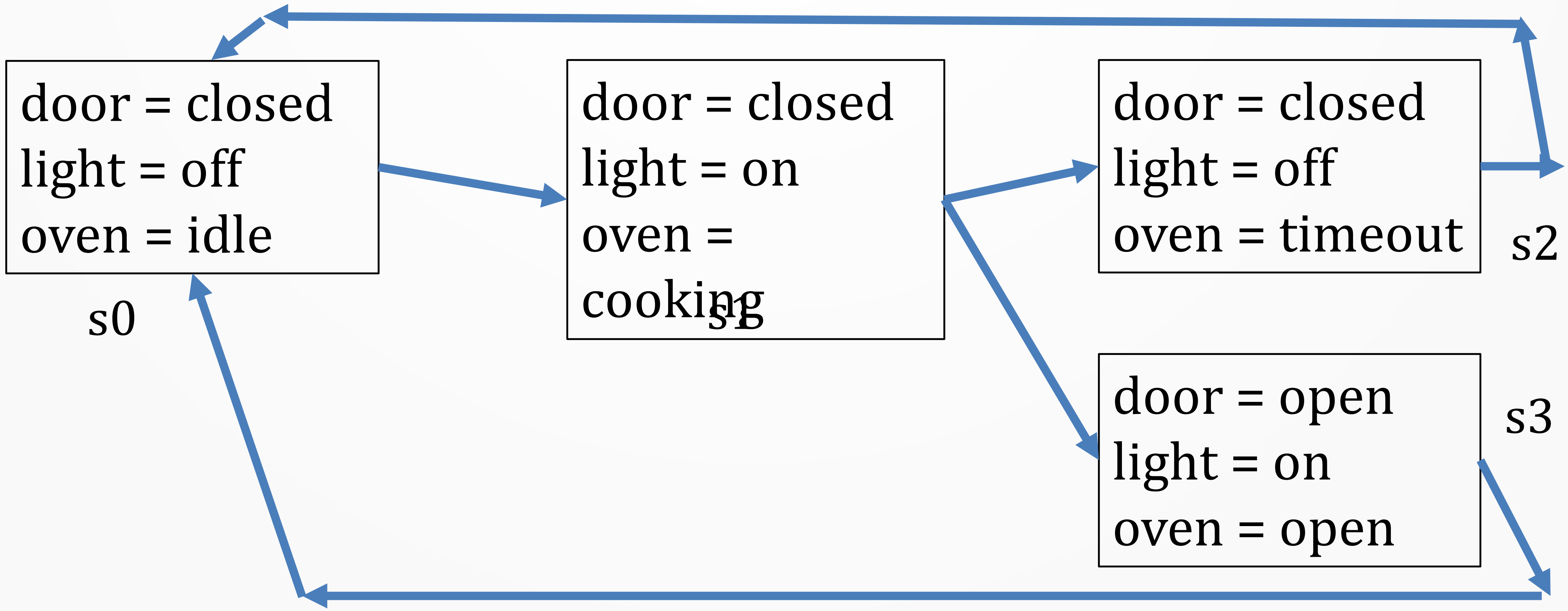
This is a **counterexample**. The property is false.

s0	s1	s3	s0	s1
door = closed light = off oven = idle	door = closed light = on oven = cooking	door = open light = on oven = open	door = closed light = off oven = idle	

Global Operator

What about **G**(door = open OR door = closed)? **Yes**

On every trace, on all the steps, either door = open or door = closed.



Future Operator

A new operator: **F** (**F**uture or eventually).

Does **F**(x = 3) hold on the trace below? **Yes**

Starting at s0, we can eventually get to a state where x = 3.

s0 s1 s2 s3 s4 s5 and so on...

s0	s1	s2	s3	s4	
x = 0	x = 1	x = 2	x = 2	**x = 3**	and so on...
y = 0	y = 0	y = 0	y = 1	y = 1	

Future Operator

Does **F**(x = 2) hold on the trace below? **Yes**

Starting at s0, we can eventually get to a state where x = 2.

What about **G**(**F**(x = 2))?

s0	s1	s2	s3	s4	s5	and so on...
x = 0	x = 1	**x = 2**	**x = 2**	x = 3		and so on with no states where x = 2 again.
y = 0	y = 0	y = 0	y = 1	y = 1		

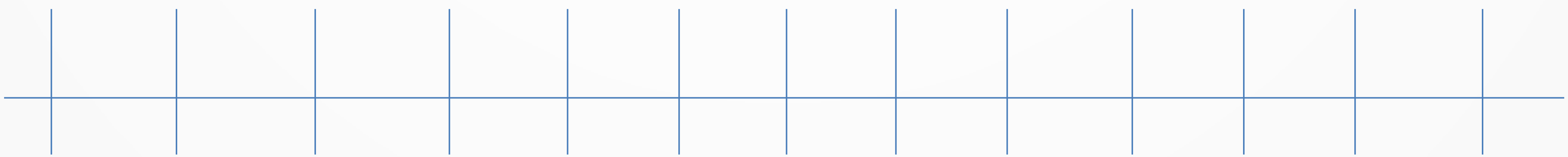
Future Operator

What about **G**(**F**(x = 2))?

Remember what we do for **G**(p) – the property p must hold on **every** state, not just the first one.

So at every state, check whether you can reach x = 2. **No – not from s4.**

s0 s1 s2 s3 s4 s5 and so on...



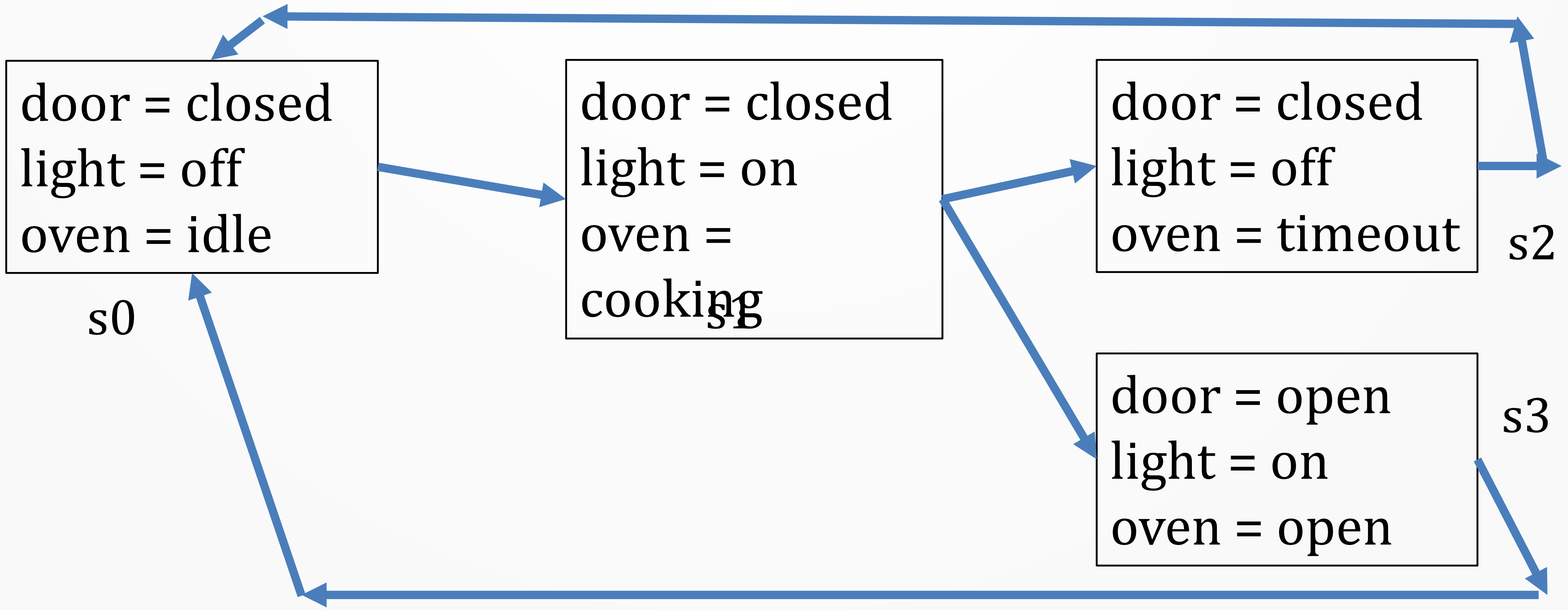
s0	s1	s2	s3	s4
x = 0	x = 1	**x = 2**	**x = 2**	x = 3
y = 0	y = 0	y = 0	y = 1	y = 1

and so on with no states where x = 2 again.

Future Operator

Does **G**(**F**(oven = idle) hold? **Yes**

On every trace, on all the steps, eventually you can get to oven = idle.



Future Operator

G(**F**(oven = idle):

Here is one trace. The state s0 is continuously reached, so from any state, you can eventually reach oven = idle.

s0	s1	s2	s0	s1
door = closed	door = closed	door = closed	door = closed	
light = off	light = on	light = off	light = off	
oven = idle	oven = cooking	oven = timeout	oven = idle	

Future Operator

G(**F**(oven = idle):

This is called **infinitely often**.

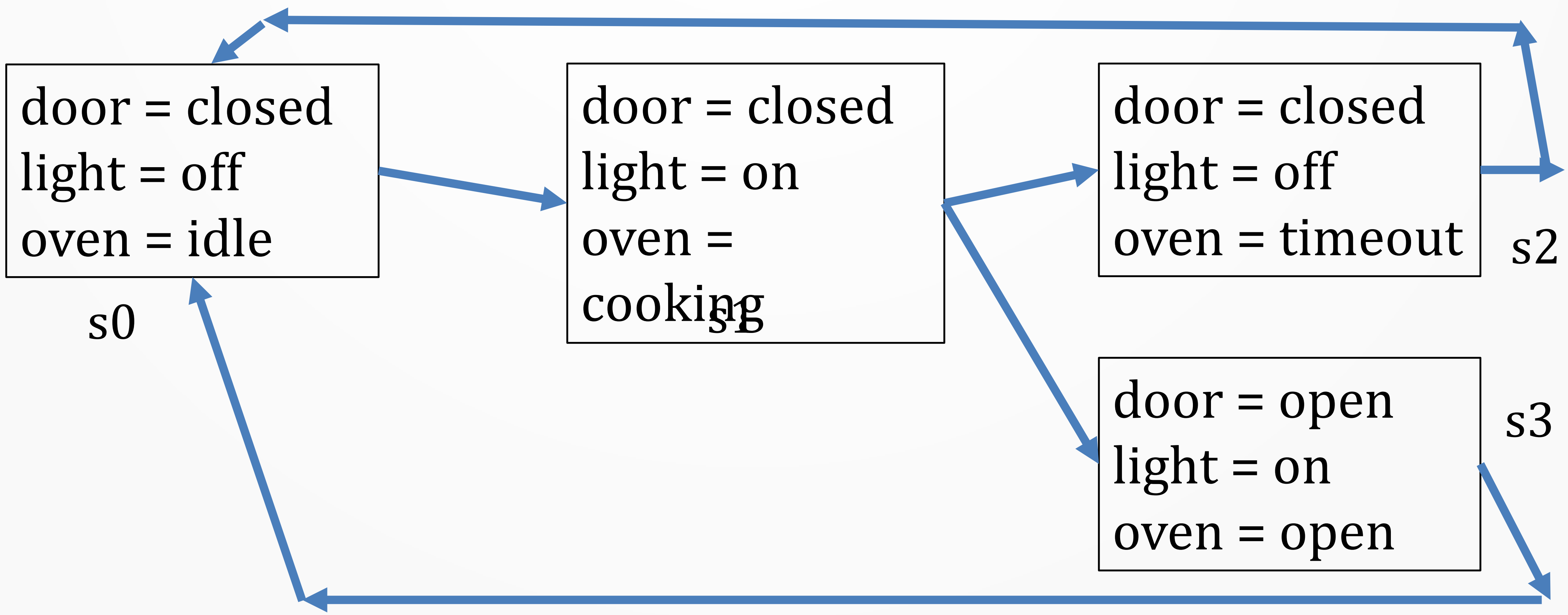
Here is *another* trace. The state s0 is continuously reached, so from any state, you can eventually reach oven = idle.

s0	s1	s3	s0	s1
door = closed light = off oven = idle	door = closed light = on oven = cooking	door = open light = on oven = open	door = closed light = off oven = idle	

Future Operator

Does **G**(**F**(door = open) hold? **No**

There is a counterexample!



Future Operator

G(**F**(door = open):

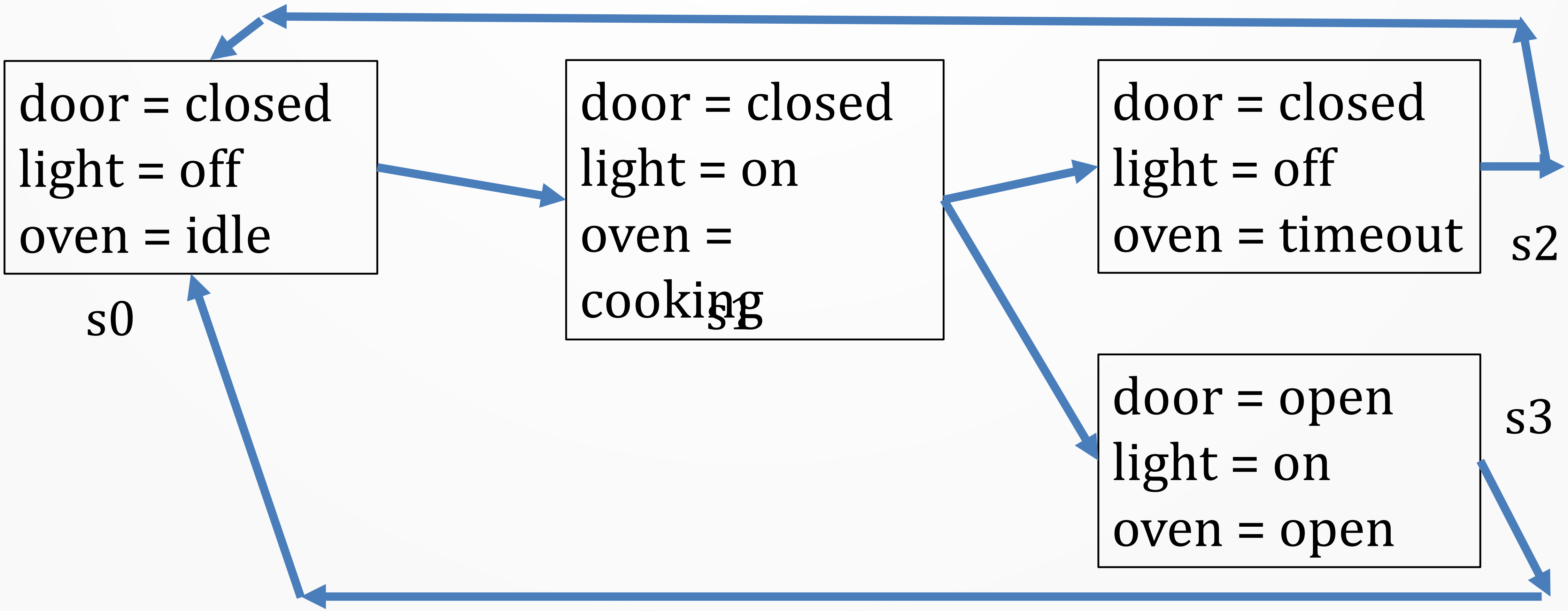
Here is the counterexample.

This trace can go on forever without reaching s3, where the door is open.

s0	s1	s2	s0	s1
door = closed	door = closed	door = closed	door = closed	
light = off	light = on	light = off	light = off	
oven = idle	oven = cooking	oven = timeout	oven = idle	

Future Operator

Counterexamples for **F** always have an infinite cycle, because we have to show that something will **never** happen.



Counterexamples

A counterexample is an execution trace showing how the property is violated.

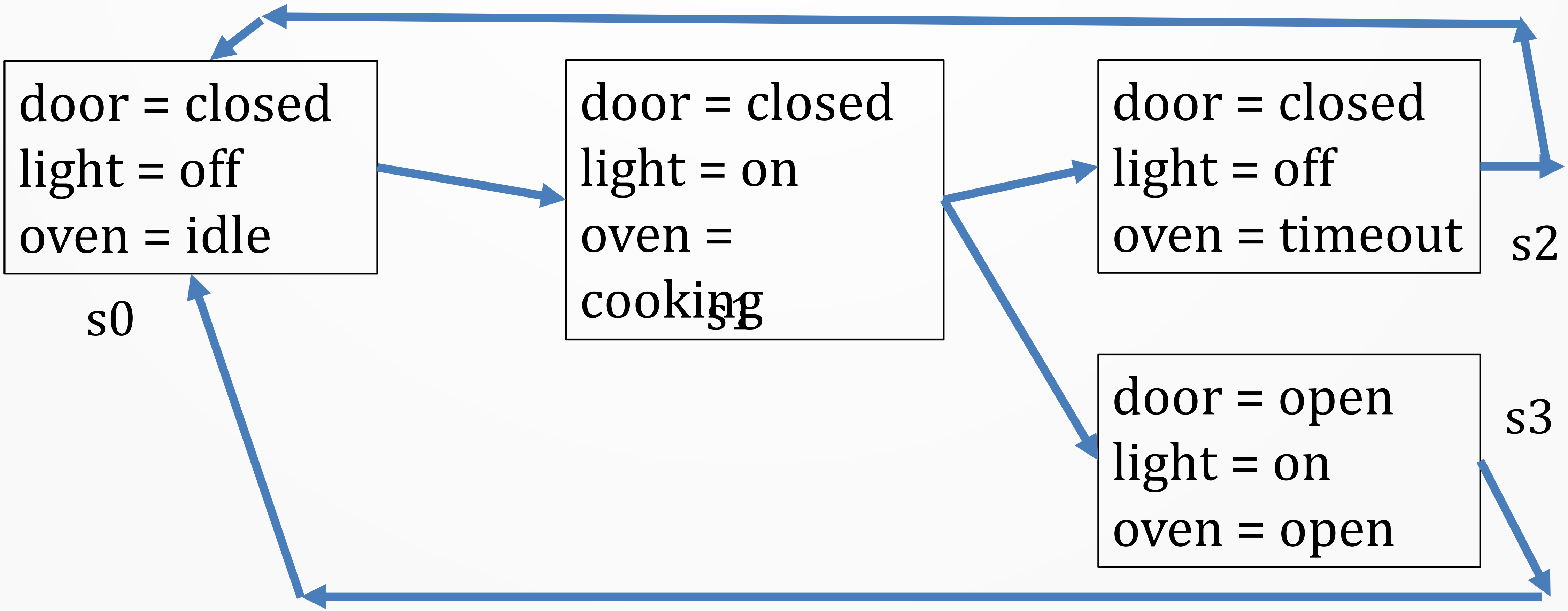
Look for the shortest counterexample if possible.

For properties like G(p) where p is an atomic proposition, then the counterexample is just a trace going up to the state where p does not hold.

Counterexamples must be full traces – start from the starting state.

Counterexamples

What is the counterexample for **G**(door=closed AND light=off)?



Counterexamples

What is the counterexample for **G**(door = closed AND light = off)?

s0 – door=closed; light=off; oven=idle

s1 – door=closed; light=on; oven=cooking

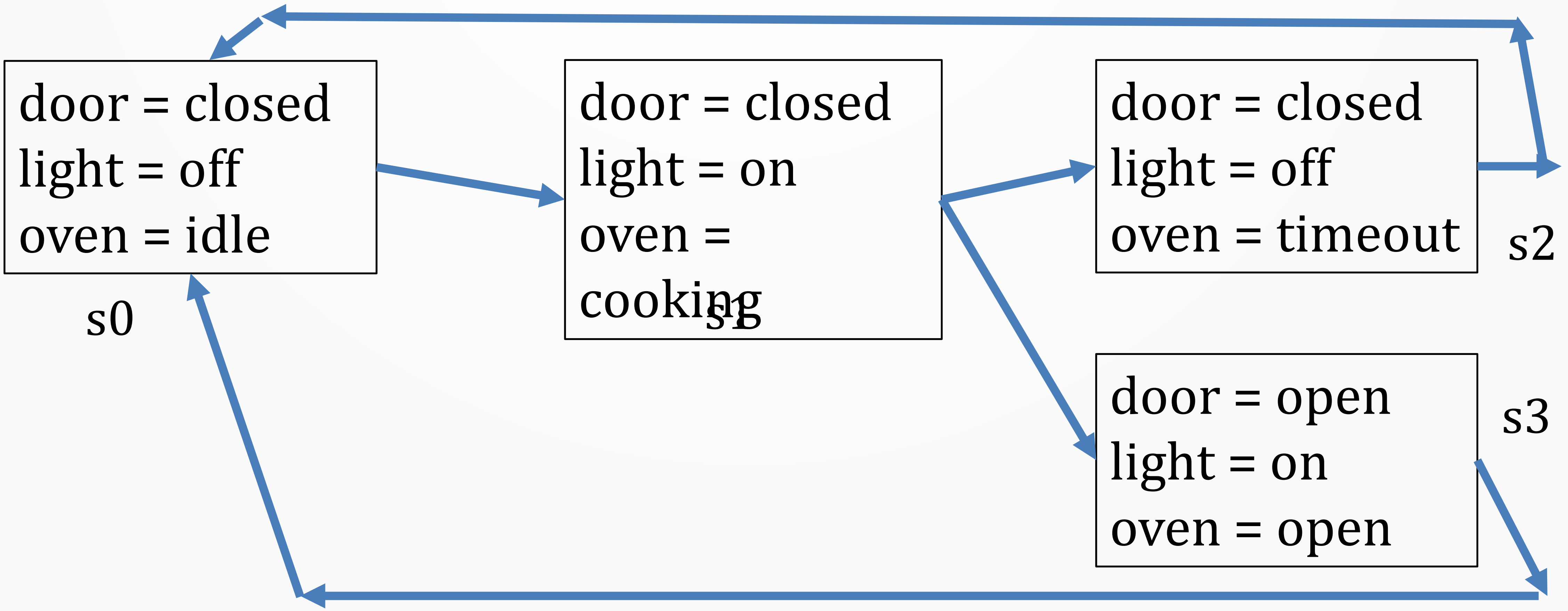
The property was violated!

Therefore the counterexample is: s0, s1.

Look for the first state where the conditions don’t hold and stop the trace there.

Counterexamples

What is the counterexample for
G(door=closed => (light=off OR oven = cooking))? Proved!



Counterexamples

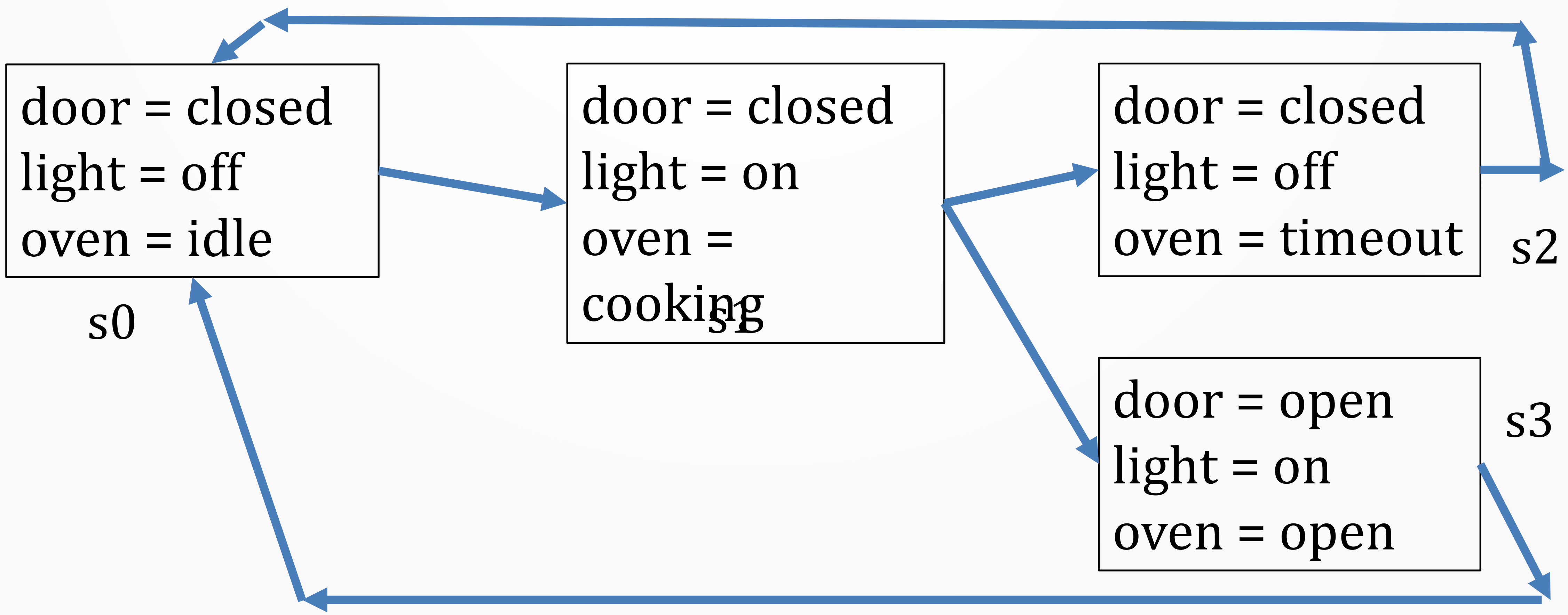
What is the counterexample for
G(door=closed => (light=off OR oven = cooking))?

Proved!

This is proved because on states s0 to s2, the door is closed and either the light is off or the oven is cooking. On state s3, the door is not closed, so the antecedent is **false**. Therefore, it doesn't matter that the light is not off and the oven is not cooking.

Counterexamples

What is the counterexample for **G**(**F**(oven = timeout))?



Counterexamples

What is the counterexample for **G**(**F**(oven = timeout))?

s0 – door=closed; light=off; oven=idle

s1 – door=closed; light=on; oven=cooking

s3 – door=open; light=on; oven=open

s0 – door=closed; light=off; oven=idle

… continue as a cycle.

Counterexample:
s0, s1, s3, s0, … cycle with
s1, s3, s0.

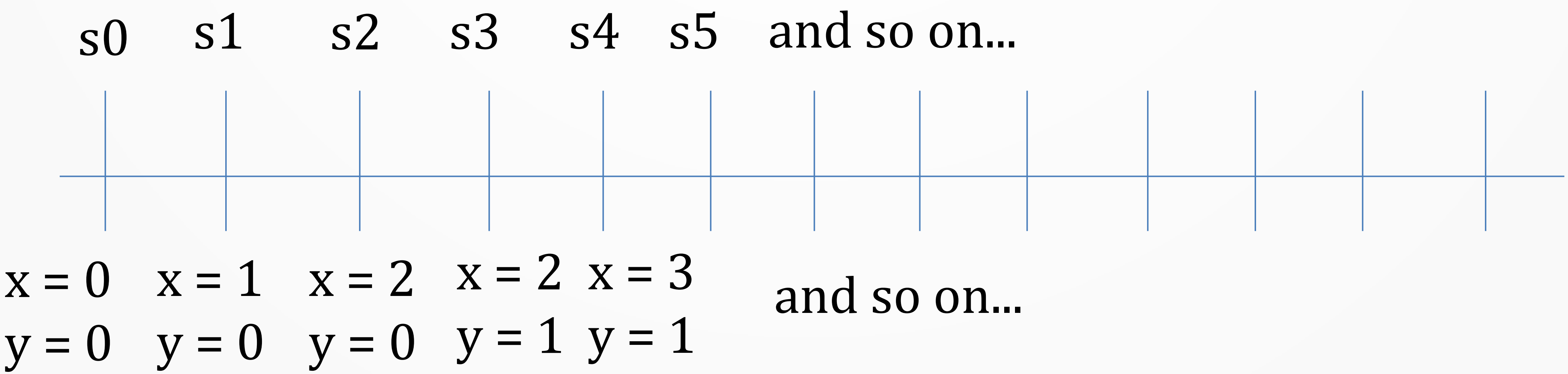
Counterexamples for **F** always need a cycle to show that it never reaches a state where the condition holds.

Implies with Globally

What about if it was **G**(x = 2 => y = 0)?

Does the property hold on the path below? **No**

At s0, s1 and s2 it holds, but not at s3.



Implies with Globally

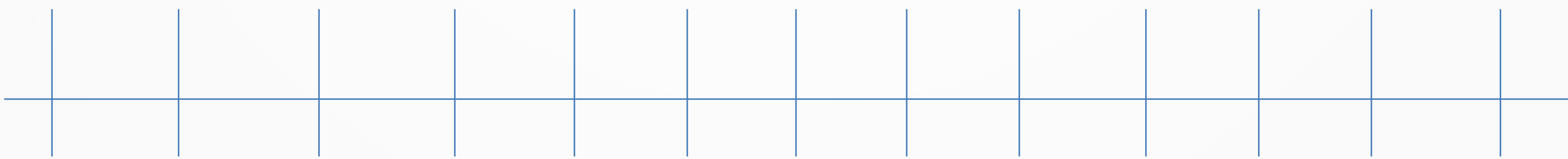
At s0, s1 and s2 it holds, but not at s3.

Counterexample: s0, s1, s2, s3.

G(x = 2 => y = 0)

Why does it hold at s0 and s1 when x is not 2? Remember how implies works.

s0 s1 s2 s3 s4 s5 and so on...



s0	s1	s2	s3	s4
x = 0	x = 1	x = 2	x = 2	x = 3
y = 0	y = 0	y = 0	y = 1	y = 1

and so on with no further changes to x or y.

Implies with Future

G(x = 2 => **F** (y = 1)) **Proved**

From every state where x = 2 holds, we can eventually reach a state where y = 1 holds (sometimes it is the same state).

s0	s1	s2	s3	s4	s5	and so on...
x = 0 y = 0	x = 1 y = 0	x = 2 y = 0	x = 2 y = 1	x = 3 y = 1		and so on with no further changes to x or y.

Implies with Future

$\mathbf{G}(x = 0 \Rightarrow \mathbf{F}\ (y = 0))$ **Proved**

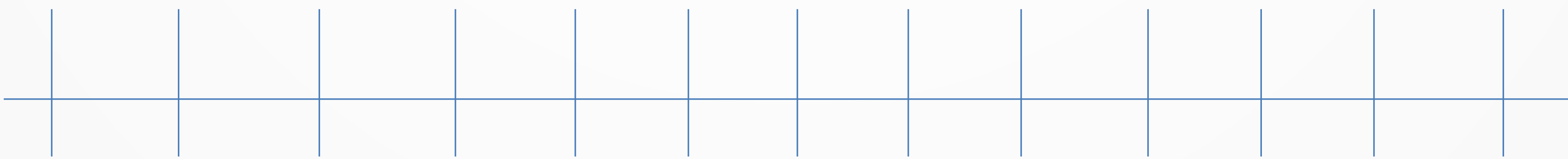
s0	s1	s2	s3	s4	s5	and so on...
x = 0	x = 1	x = 2	x = 2	x = 3		and so on with no further changes to x or y.
y = 0	y = 0	y = 0	y = 1	y = 1		

Implies with Future

$\mathbf{G}(x = 0 \Rightarrow \mathbf{F}\ (y = 0))$ **False**

No way to get from this state where x = 0 to one where y = 0.

s0 s1 s2 s3 s4 s5 and so on...



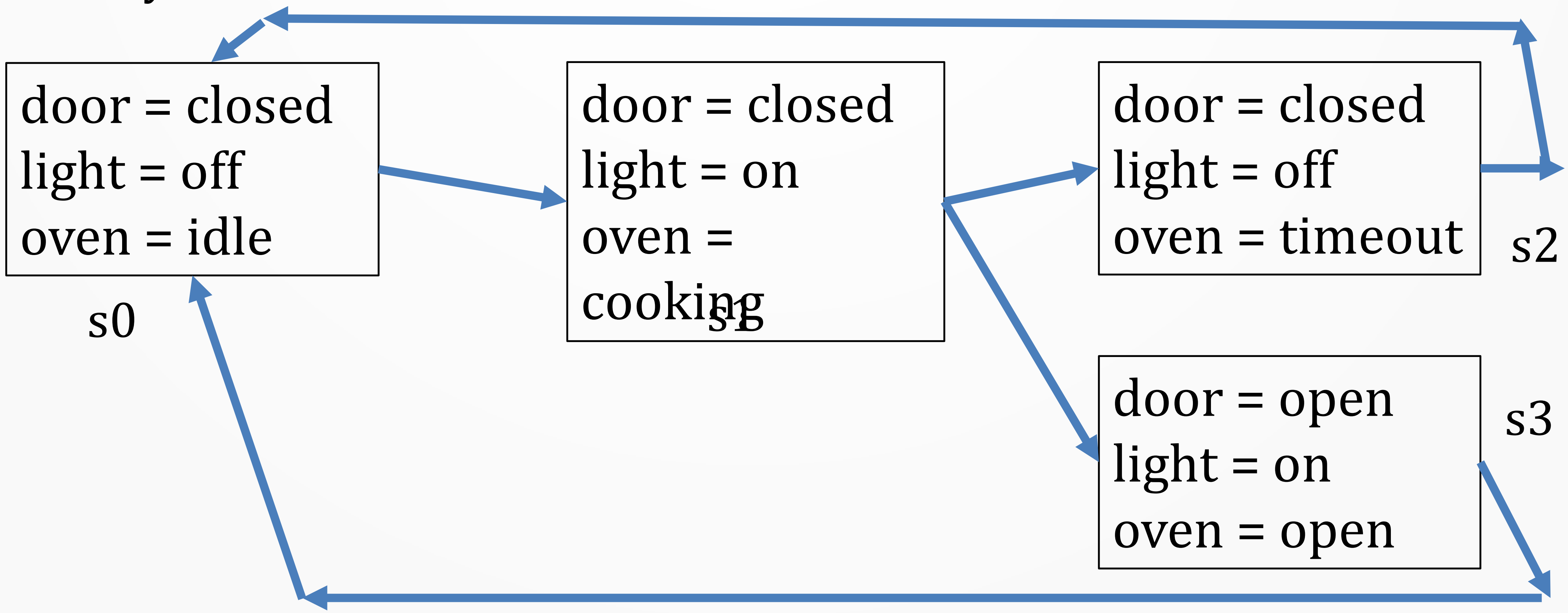
s0	s1	s2	s3	s4
x = 0	x = 1	x = 0	x = 2	x = 3
y = 0	y = 0	y = 1	y = 1	y = 1

and so on with no further changes to x or y.

Implies with Future

Does **G**(oven = idle => **F**(door = closed)) hold? **Yes**

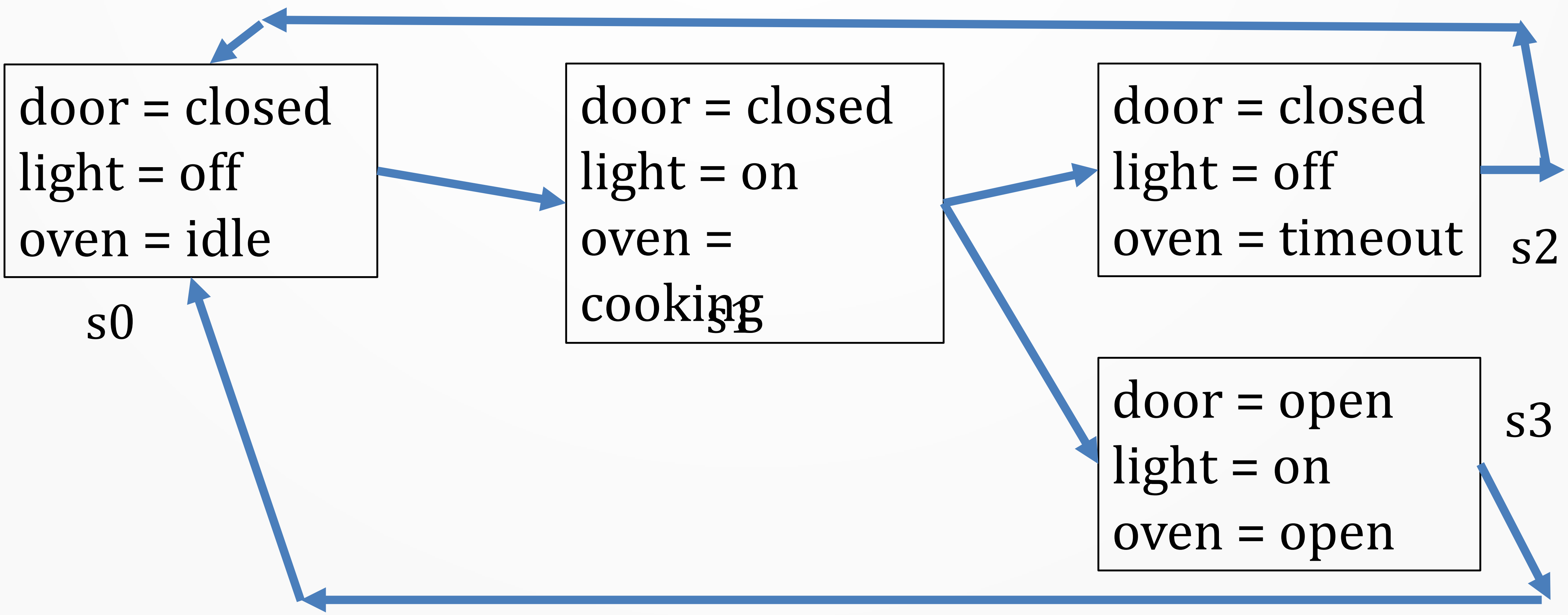
The only state where oven = idle is s0. From here, it is always possible to eventually reach a state where door = closed.



Implies with Future

Does **G**(door = closed => **F**(oven = open)) hold? **No**

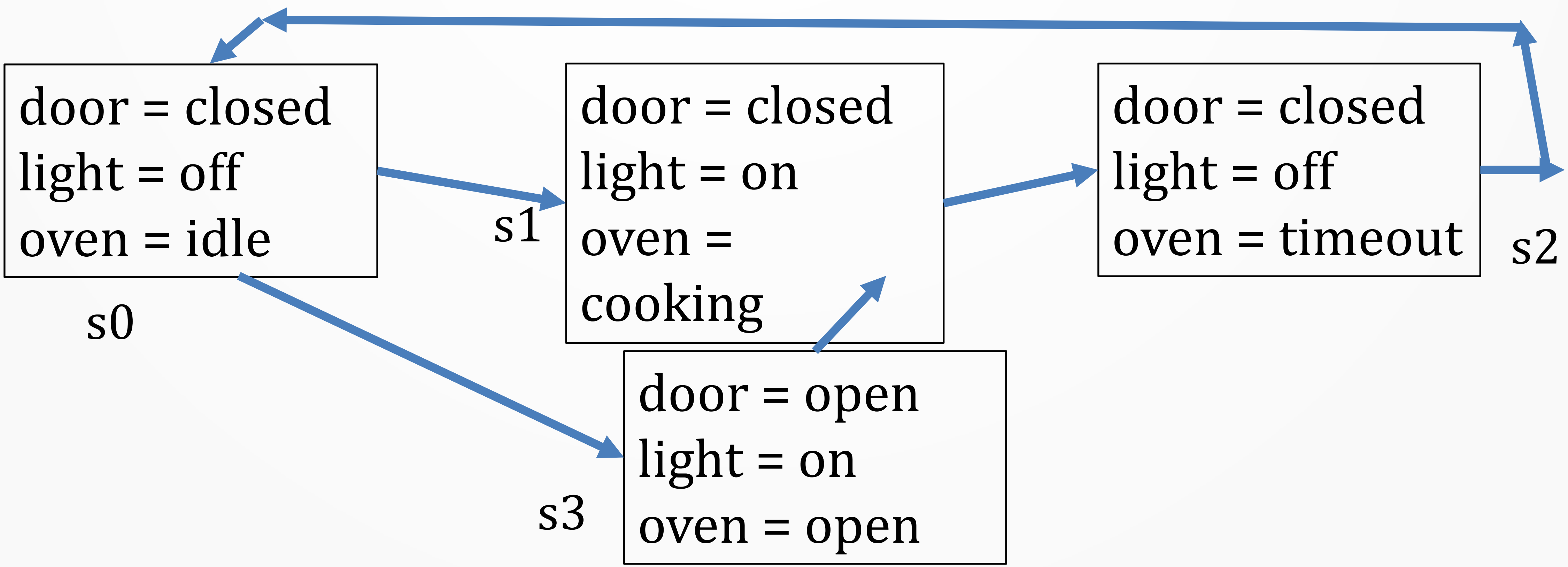
Counterexample: s0, s1, s2, then an infinite cycle with s0, s1, s2.



Implies with Future

Does **G**(door = closed => **F**(oven = timeout)) hold? **Yes**

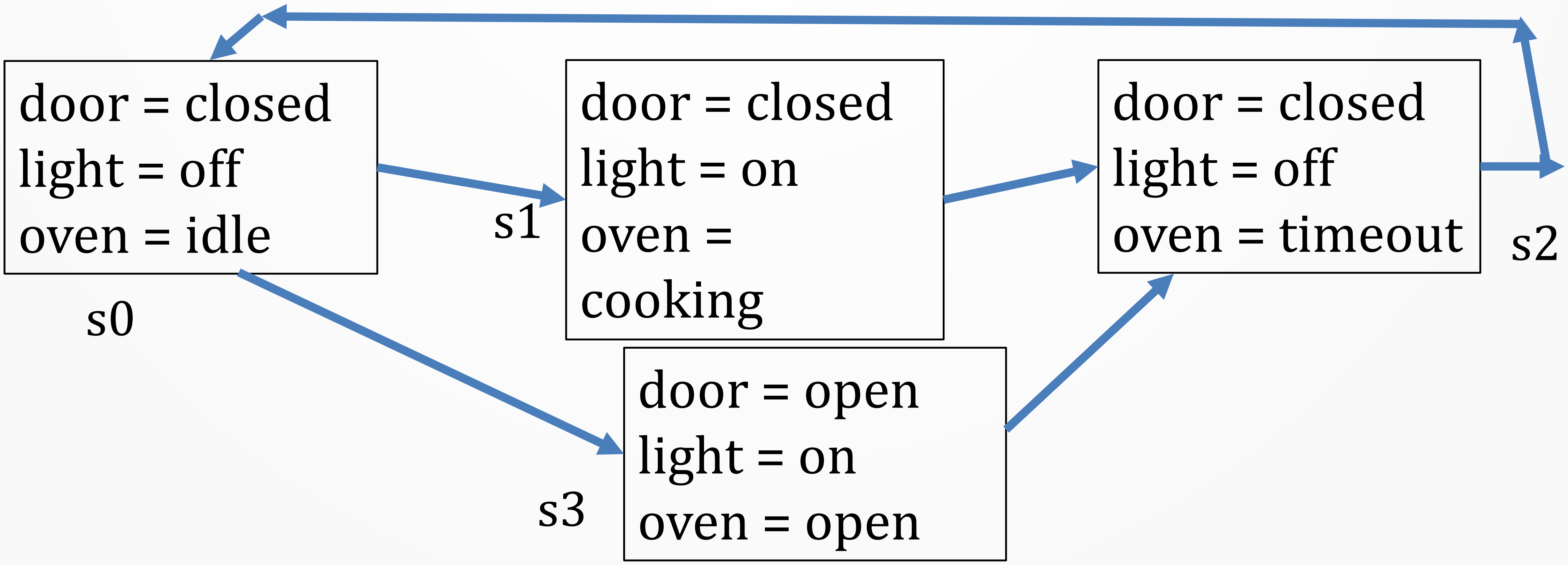
Even on paths that go through s3, s2 is always still reached.



Implies with Future

Does **G**(light = on => **F**(oven = cooking)) hold? **No**

At s1, oven is already cooking, but from s3, we could keep missing oven = cooking. Counterexample: s0, s3, cycle s2, s0, s3.



Next Operator

Another new operator: **X** (Ne**X**t).

Does **X**(x = 1) hold on the trace below? **Yes**

Starting at s0, in the next state, x = 1.

If the formula has just an **X** by itself like this one, then you only look at the 2nd state (s1 in the example).

s0	s1	s2	s3	s4	s5	and so on...
x = 0 y = 0	**x = 1** y = 0	x = 2 y = 0	x = 2 y = 1	x = 3 y = 1		and so on...

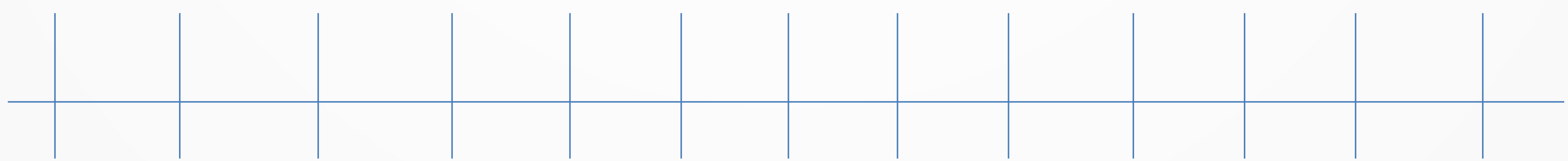
Implies with Next

What about if it was **G**(x = 2 => **X**(y = 1))?

Does the property hold on the trace below? **Yes**

Find the states where x = 2 holds and then look at the next states after them.

s0 s1 s2 s3 s4 s5 and so on...



s0	s1	s2	s3	s4	
x = 0 y = 0	x = 1 y = 0	x = 2 y = 0	x = 2 y = 1	x = 3 y = 1	and so on with no changes to x or y.

Implies with Next

Now try **G**(y = 0 => **X**(y = 1))?

Does the property hold on the trace below?

Find the states where y = 0 holds and then look at the next states after them.

s0 s1 s2 s3 s4 s5 and so on...

s0	s1	s2	s3	s4
x = 0	x = 1	x = 2	x = 2	x = 3
y = 0	y = 0	y = 0	y = 1	y = 1

and so on with no changes to x or y.

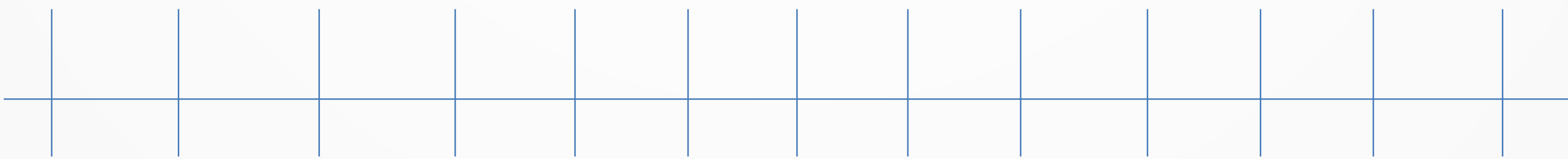
Implies with Next

Now try **G**(y = 0 => **X**(y = 1))?

Does the property hold on the trace below? **No**

Counterexample: s0. At this state, y = 0 but on the next state, s1, y is not 1.

s0 s1 s2 s3 s4 s5 and so on...



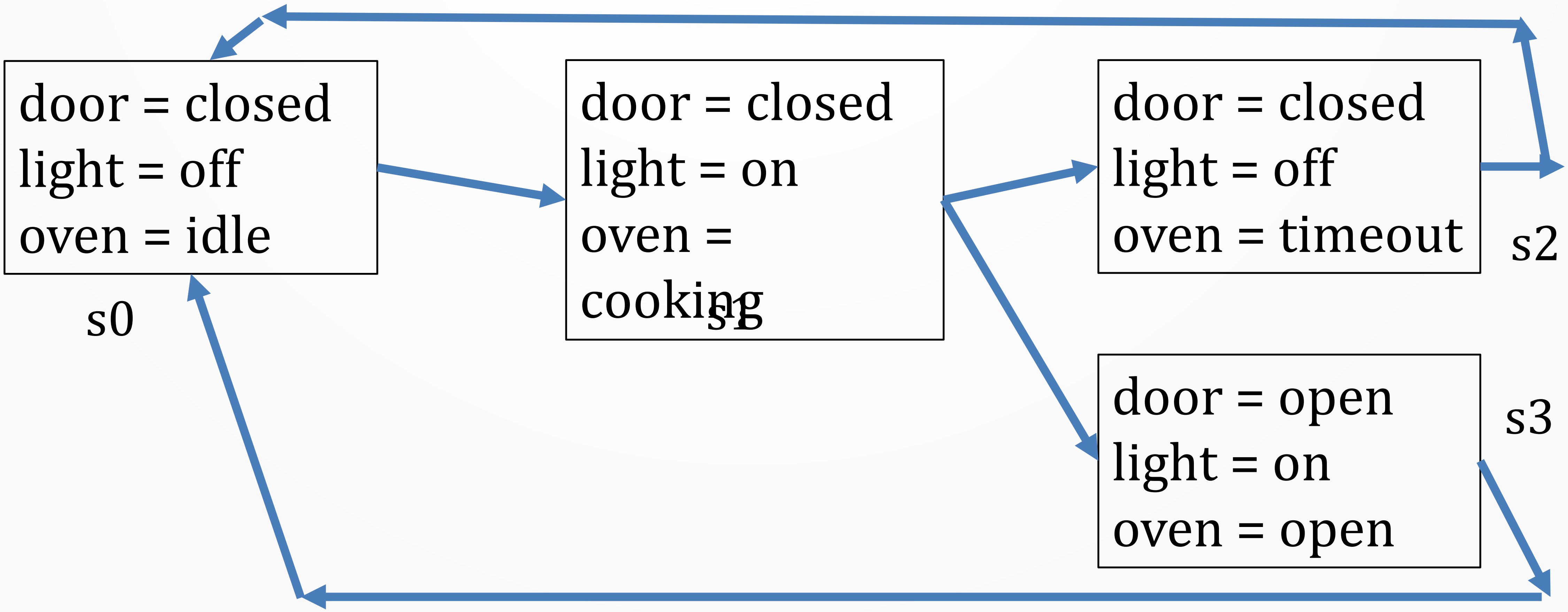
s0	s1	s2	s3	s4
x = 0	x = 1	x = 2	x = 2	x = 3
y = 0	y = 0	y = 0	y = 1	y = 1

and so on with no changes to x or y.

Implies with Next

Does **G**(oven = idle => **X**(door = closed)) hold? **Yes**

The only state where oven = idle is s0. From here, the only next step is s1.



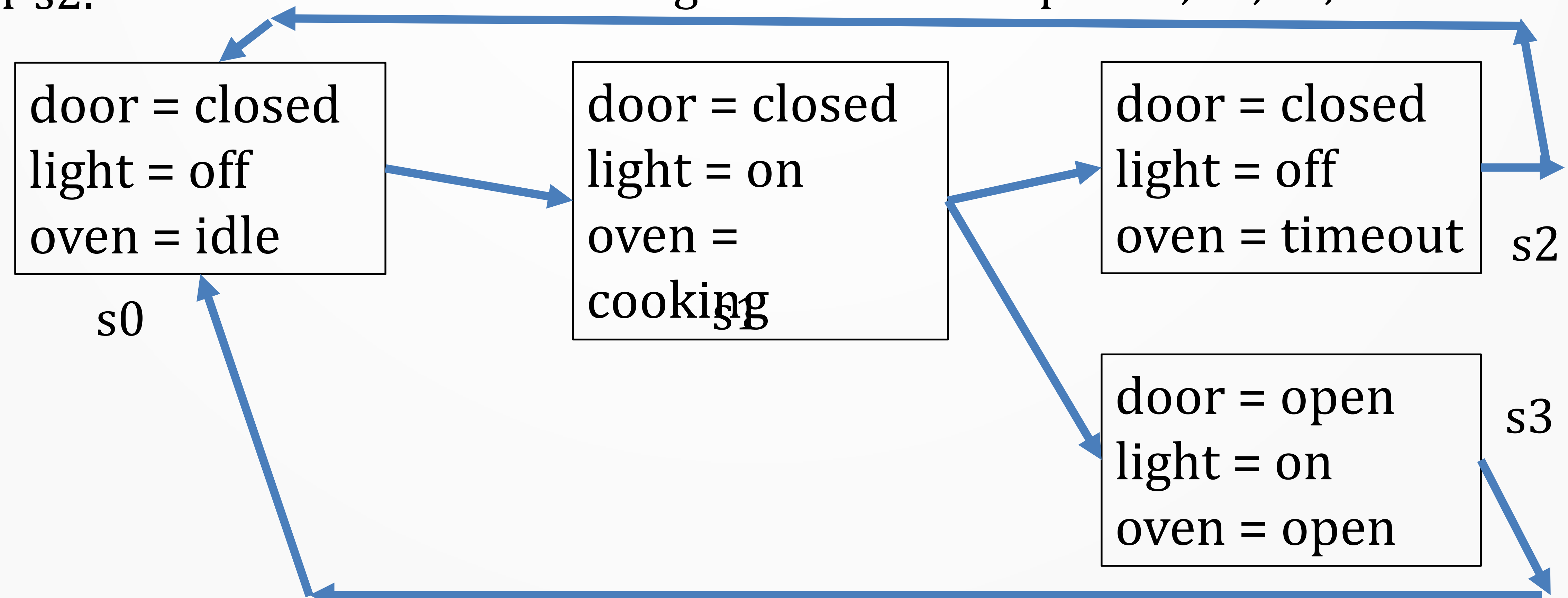
Implies with Next

Does **G**(door = closed => **X**(light = on)) hold? **No**

At s0, light = on in the next state, but not at s1 or s2.

Counterexample: s0, s1, s2.

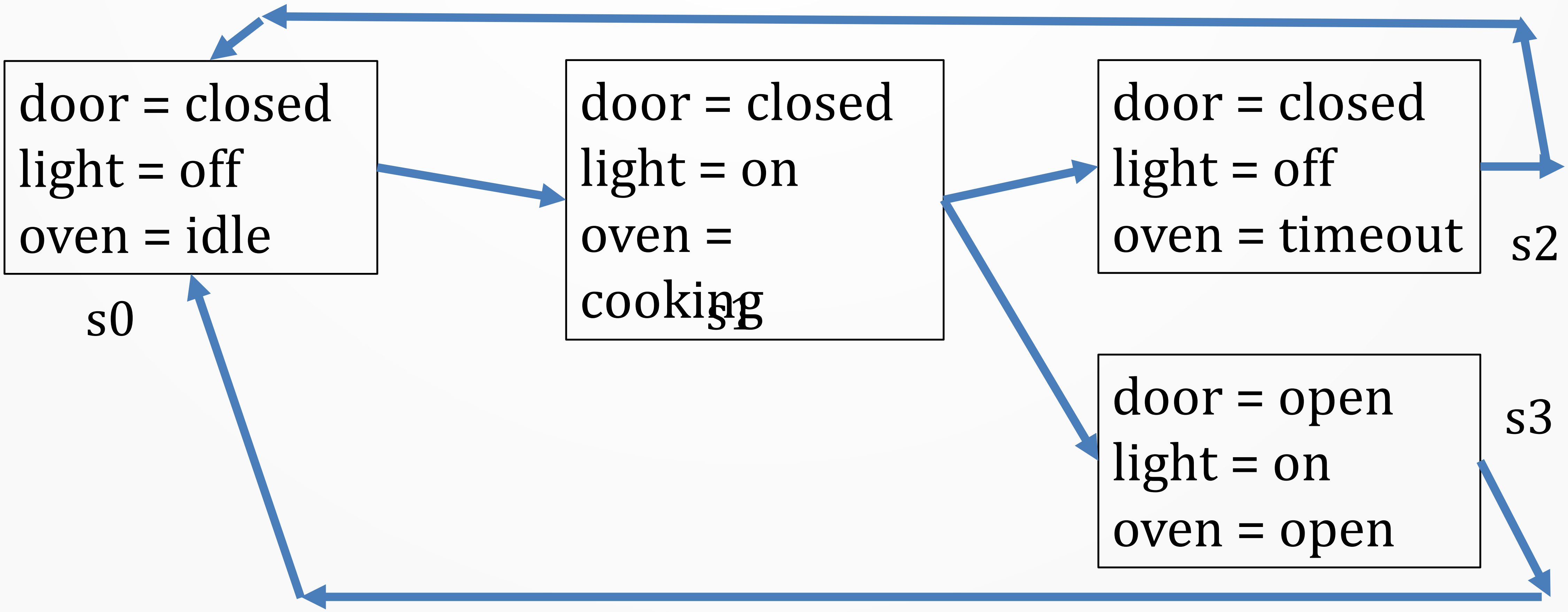
Alternative longer counterexample: s0, s1, s2, s0.



Implies with Next

Does **G**(light = off => **X**(oven = cooking)) hold? **No**

Counterexample: s0, s1, s2, s0. No cycle is needed for **X**.



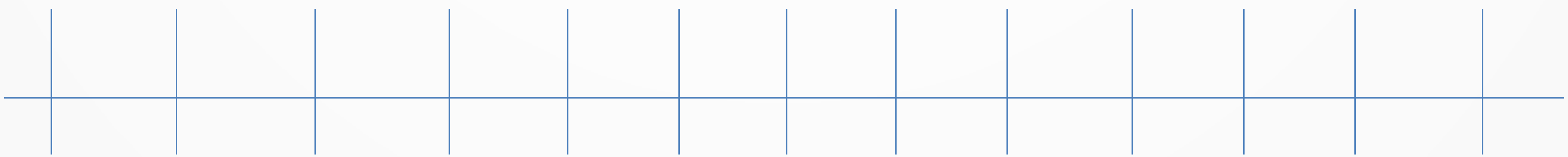
Until Operator

U (**U**ntil) p **U** q – p holds until q holds.

Does (y = 0) **U** (x = 2) hold on the trace below? **Yes**

y = 0 holds on all states until it reaches a state where x = 2 holds (s2).

s0 s1 s2 s3 s4 s5 and so on...



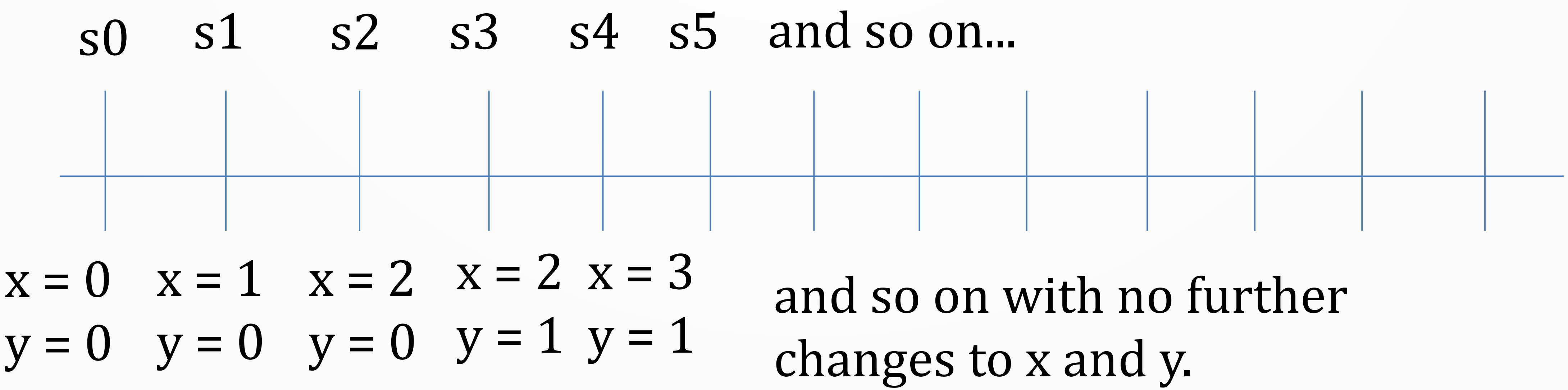
s0	s1	s2	s3	s4
x = 0	x = 1	x = 2	x = 2	x = 3
y = 0	y = 0	y = 0	y = 1	y = 1

and so on with no further changes to x and y.

Until Operator

Does (y = 0) **U** (x = 4) hold on the trace below? **No**

Remember that if the second clause (x = 4) doesn't ever hold, then the property is **false**.



Until Operator

Does **G**((y = 0) **U** (x = 2)) hold on the trace below? **No**

From any state, y should stay 0 until x is 2.

Is this true at s4? How about s5?

s0	s1	s2	s3	s4	s5	and so on...
x = 0	x = 1	x = 2	x = 2	x = 2	x = 3	x = 2
y = 0	y = 0	y = 0	y = 1	y = 1	y = 1	y = 1

and so on with no further changes to x and y.

Until Operator

Does **G**(x = 1 => ((y = 0) **U** (x = 2))) hold on the trace below? **Yes**

We're only interested in looking at s1. From here, y stays 0 until x is 2.

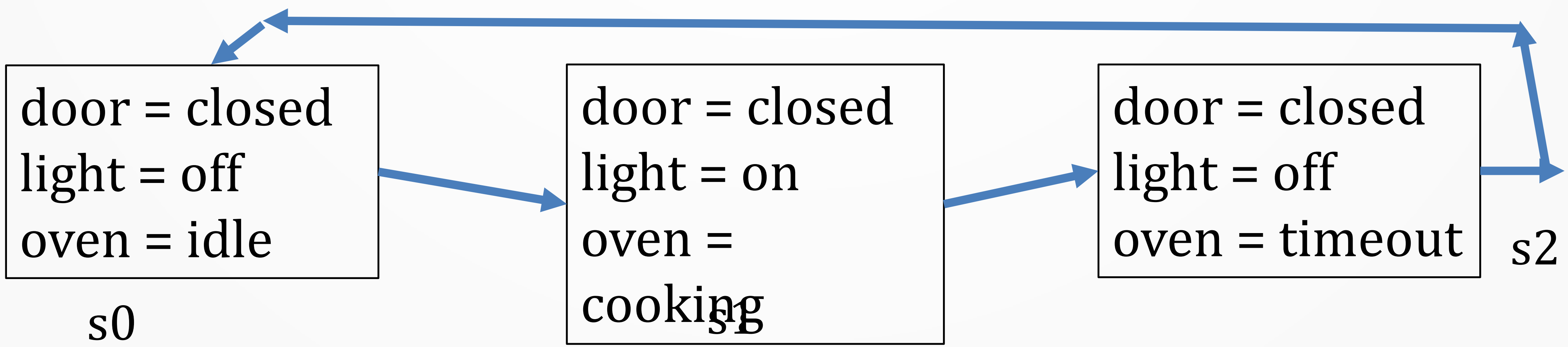
s0	s1	s2	s3	s4	s5	and so on...
x = 0	x = 1	x = 2	x = 2	x = 2	x = 3	x = 2
y = 0	y = 0	y = 0	y = 1	y = 1	y = 1	y = 1

and so on with no further changes to x and y.

Until

Does **G**(light = off **U** oven = cooking) hold? **Yes**

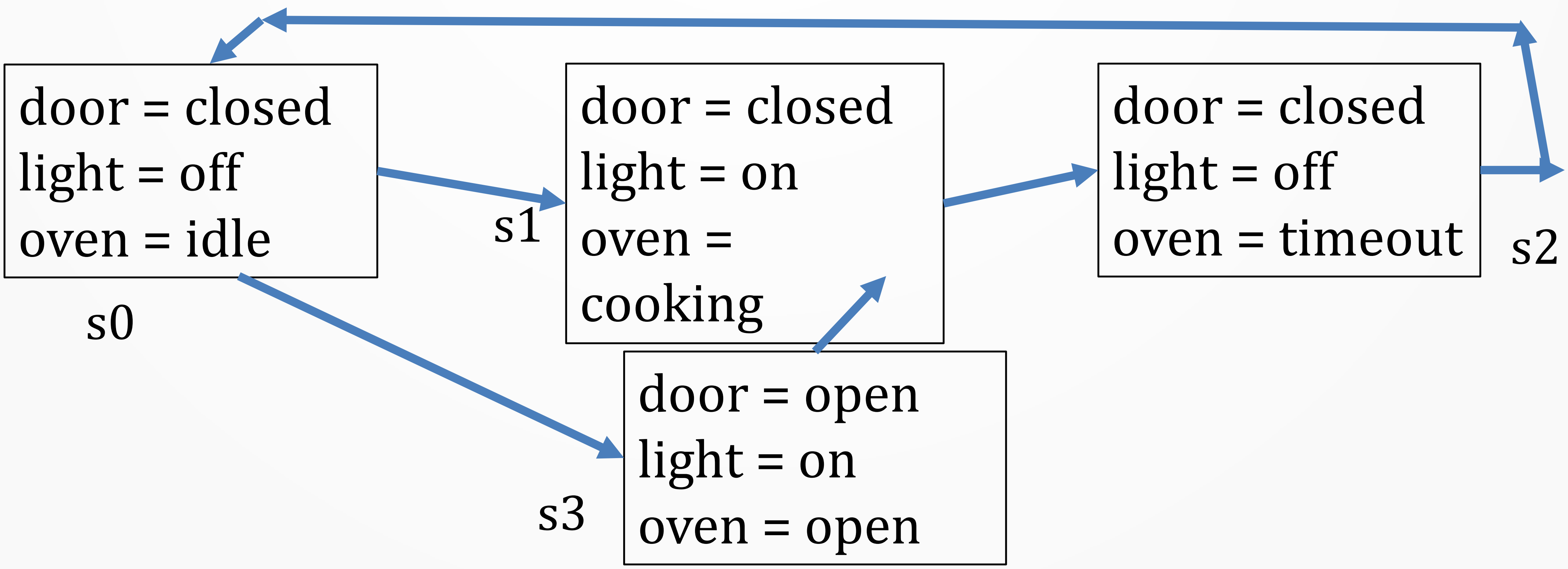
The light stays off until s1, where the oven is cooking.



Until

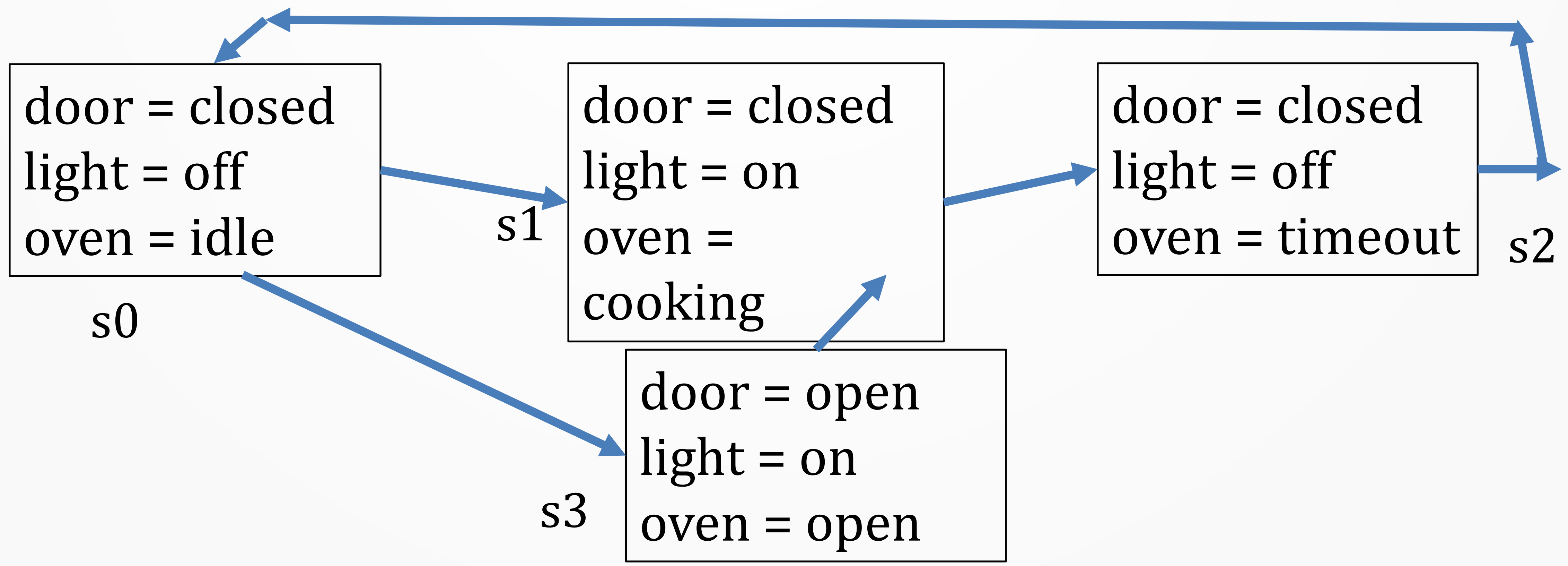
Does **G**(light = off **U** oven = cooking) hold? **No**

Counterexample: s0, s3, s1.



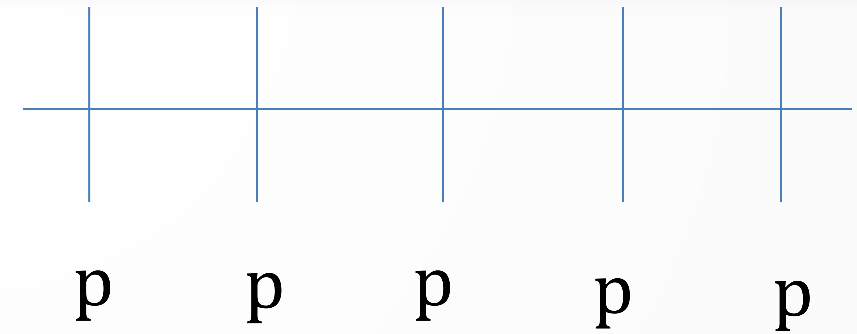
Implies with Until

Does **G**(oven = open => (light = on **U** oven = timeout)) hold? **Yes**

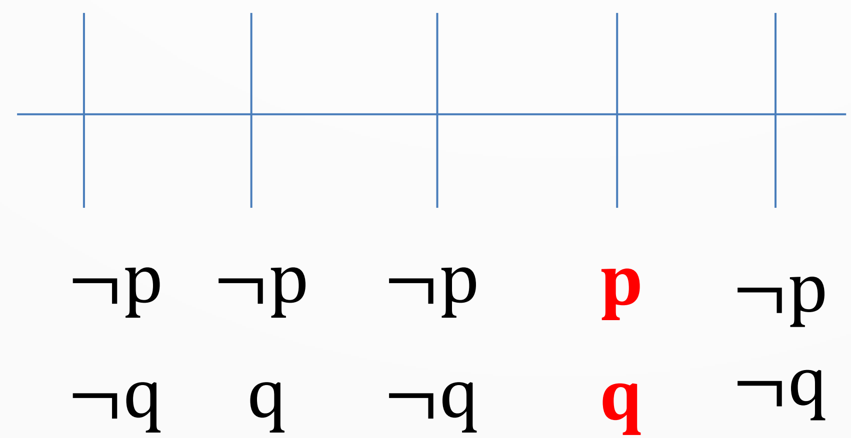


LTL Operators Overview

G(p) – p must hold on **every** state.



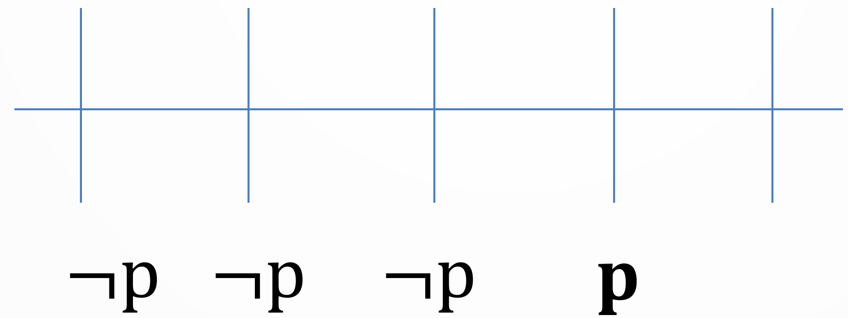
G(p => q) – on every state, if p holds then q holds.



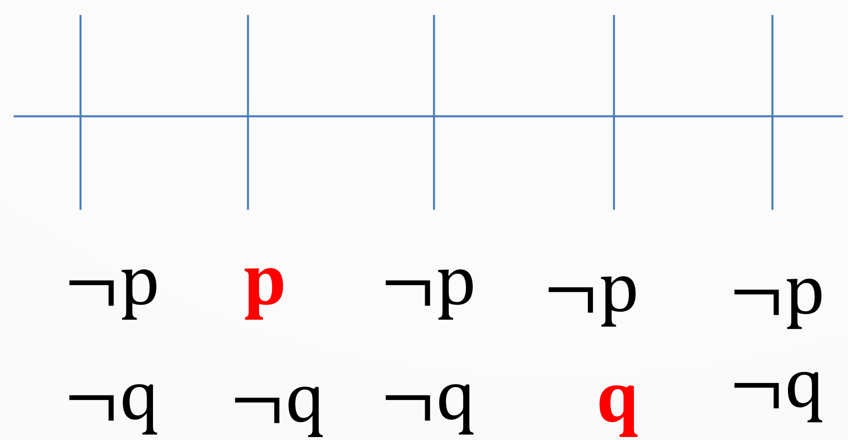
Remember that it doesn't matter whether q holds or not on the states where p doesn't hold.

LTL Operators Overview

F(p) – p must **eventually** hold. There must be a state in the future where p holds.



G(p => **F**(q)) – on every state, if p holds then eventually q holds.

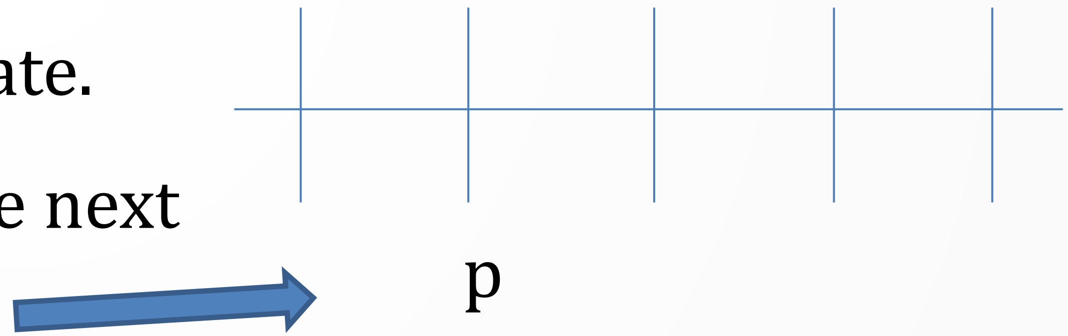


p doesn’t have to hold in the same state as q but it can.

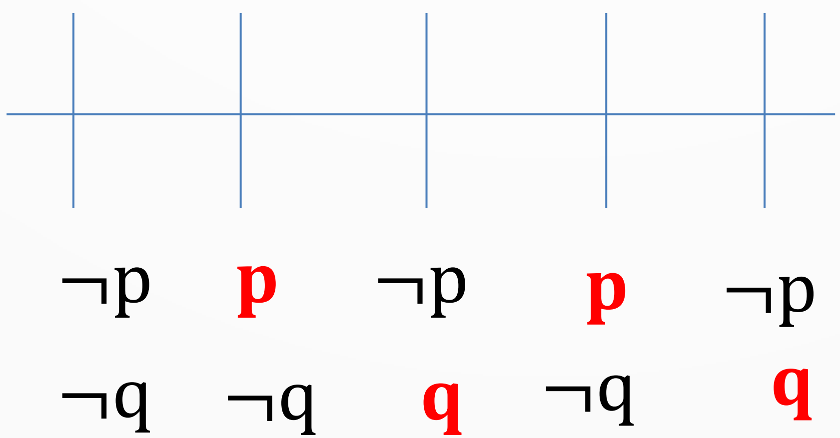
LTL Operators Overview

X(p) – p must hold on the **next** state.

If there's no **G** then it just means the next state after the starting state.

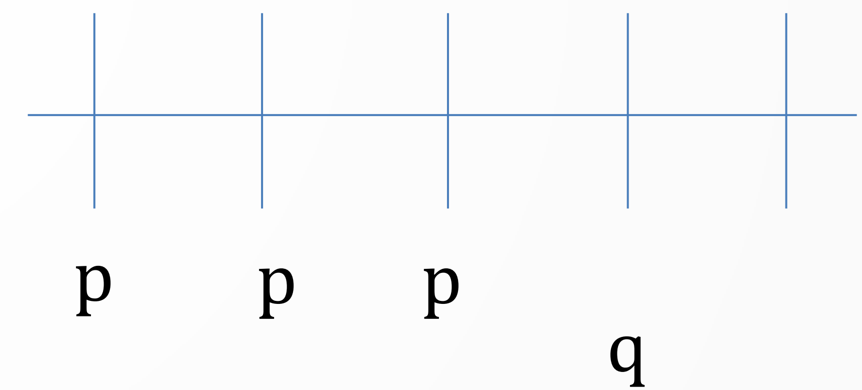


G(p => **X**(q)) – on every state, if p holds then in the next state q holds.

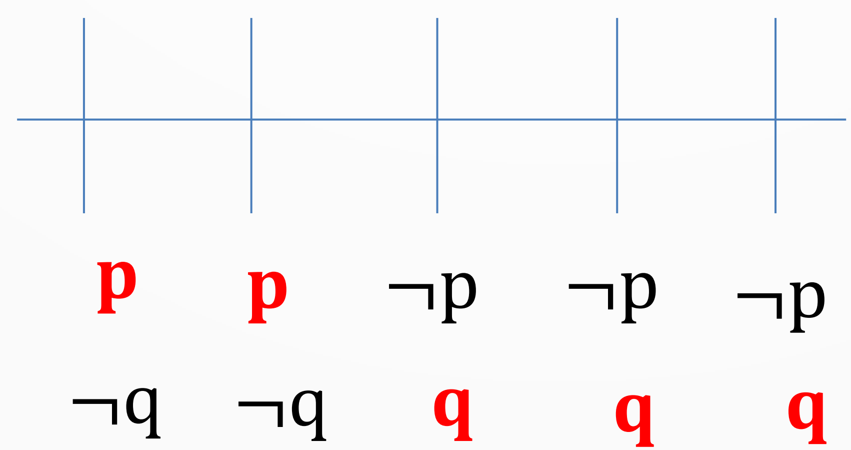


LTL Operators Overview

p **U** q – p must hold **until** q holds.

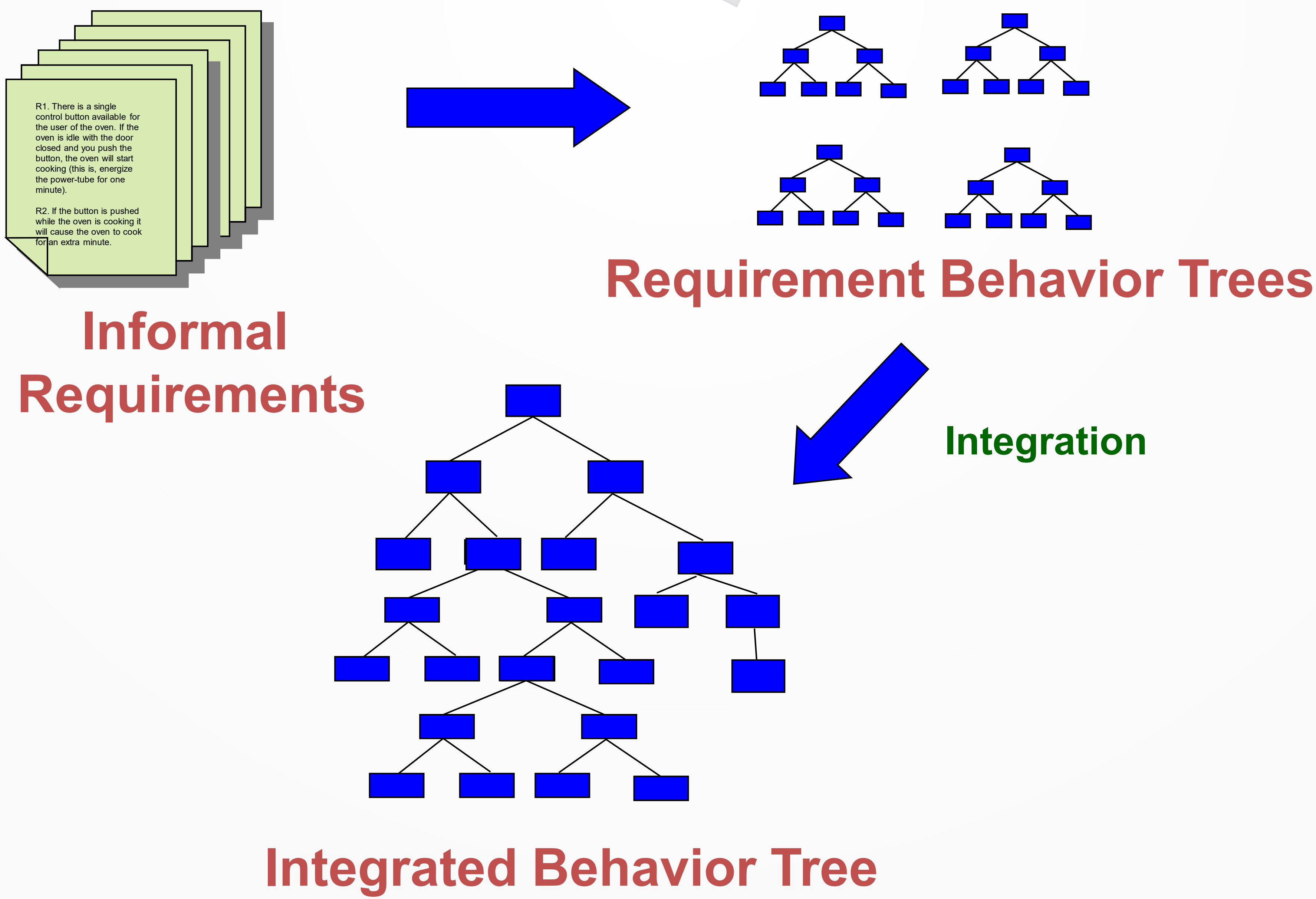


G(p **U** q) – from every state, p must hold until q holds.



Remember that from every state, q must hold eventually.

Behavior Trees



Behavior Tree Syntax

C
[s]

State-realisation

C
? s ?

Selection

C
??? s ???

Guard

C
> e <

Internal Input Event

C
< e >

Internal Output Event

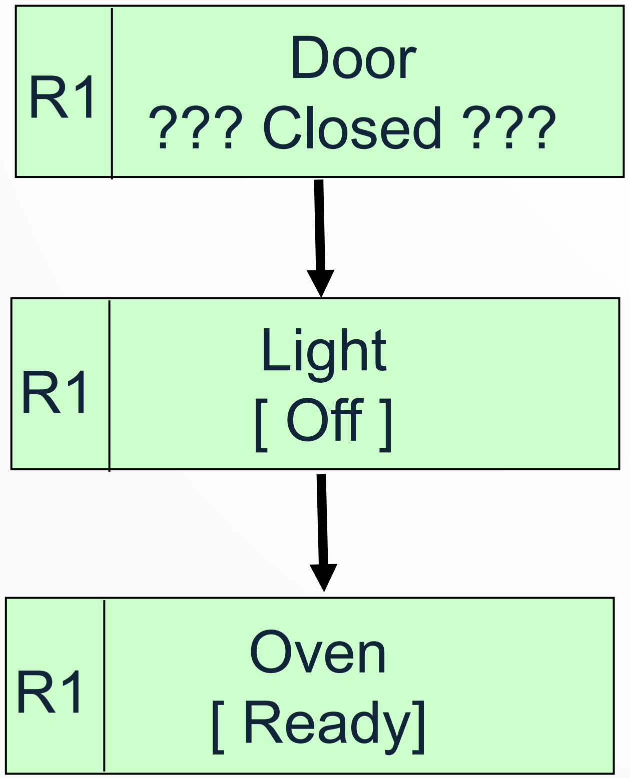
C
>> e <<

External Input Event

C
<< e >>

External Output Event

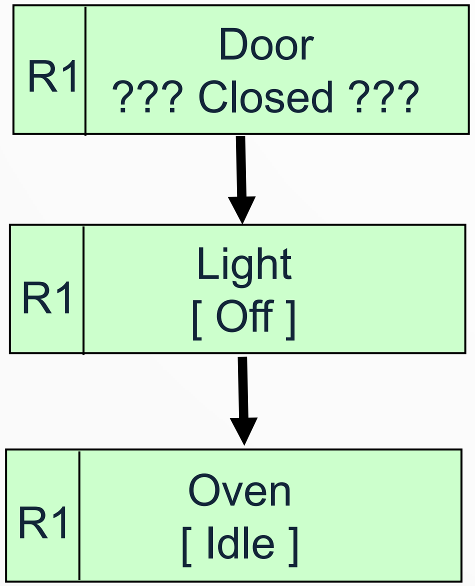
Behavior Tree Syntax



When the door is open, the light goes off.

Behavior Trees

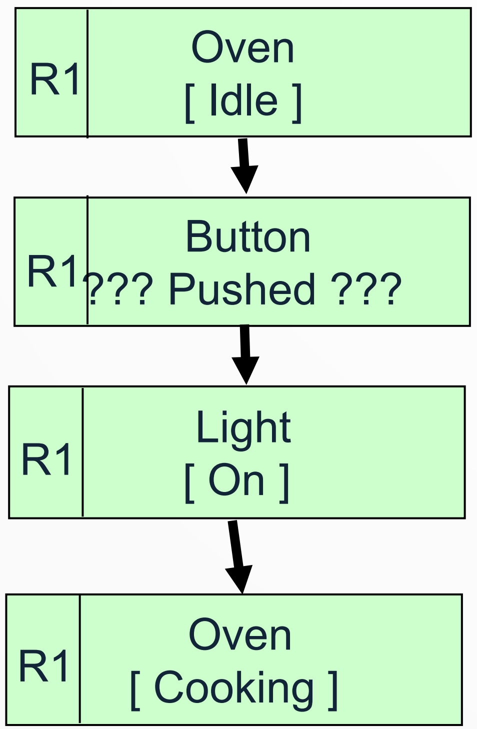
Behavior Trees is a language for writing about the *requirements* of a system.



Microwave Oven requirement 1:

When the door is closed, the light should go off and the oven goes into an idle state.

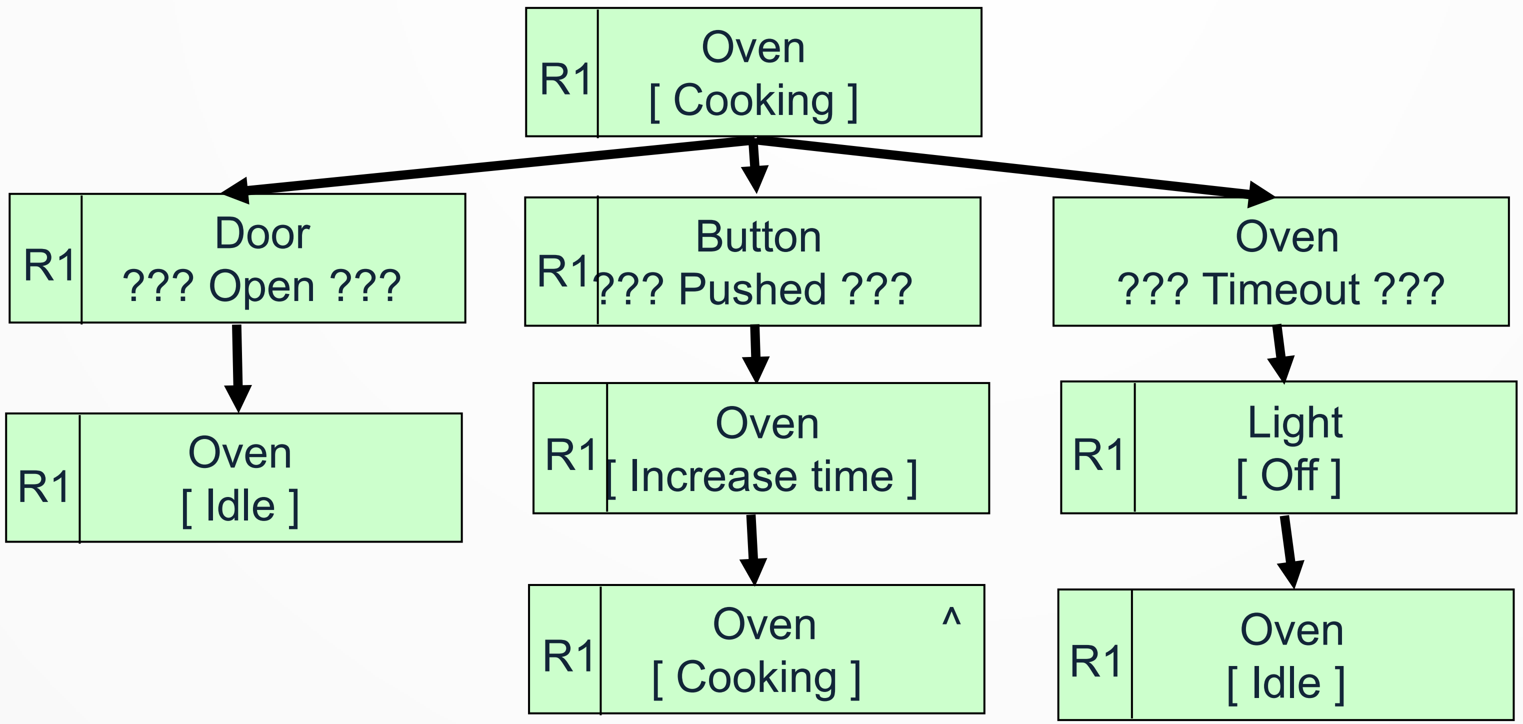
Behavior Trees



Microwave Oven requirement 2:

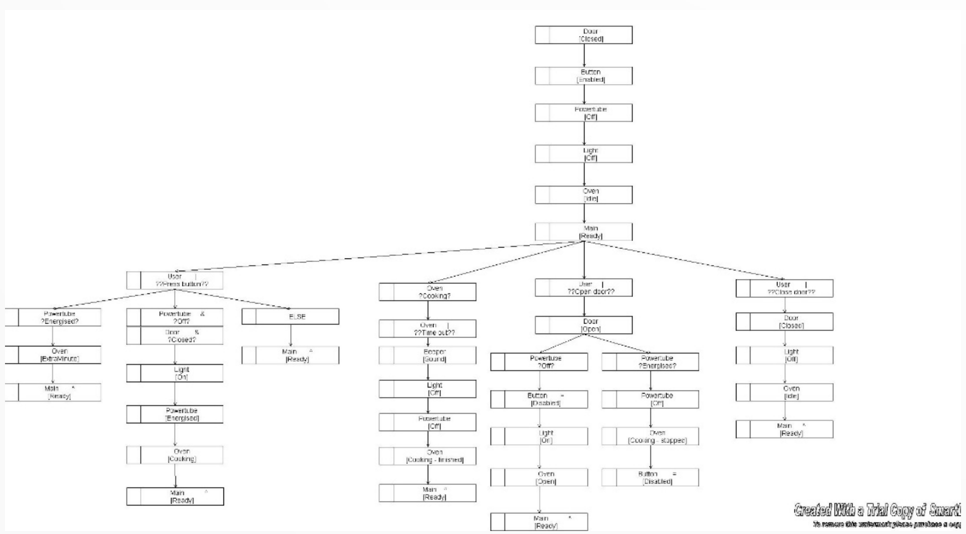
When the oven is idle, when the button is pushed, the light should go on and the oven should begin cooking.

Behavior Trees



Making the System

Requirements → Design → Implementation (Software)



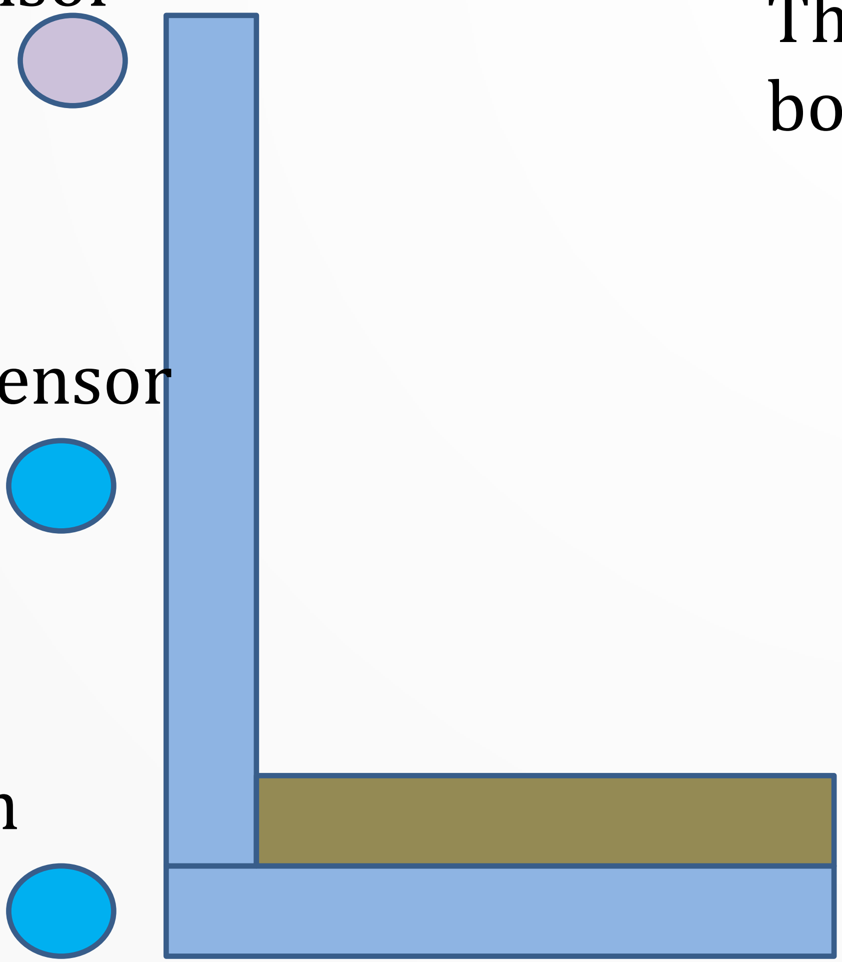
```
public void
closeDoor(){

door.closeDoor();

light.turnOff();
}
```

A Safety-Critical System

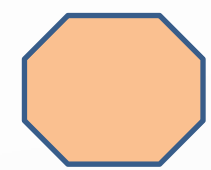
Top sensor low

PONR sensor high

Bottom sensor high

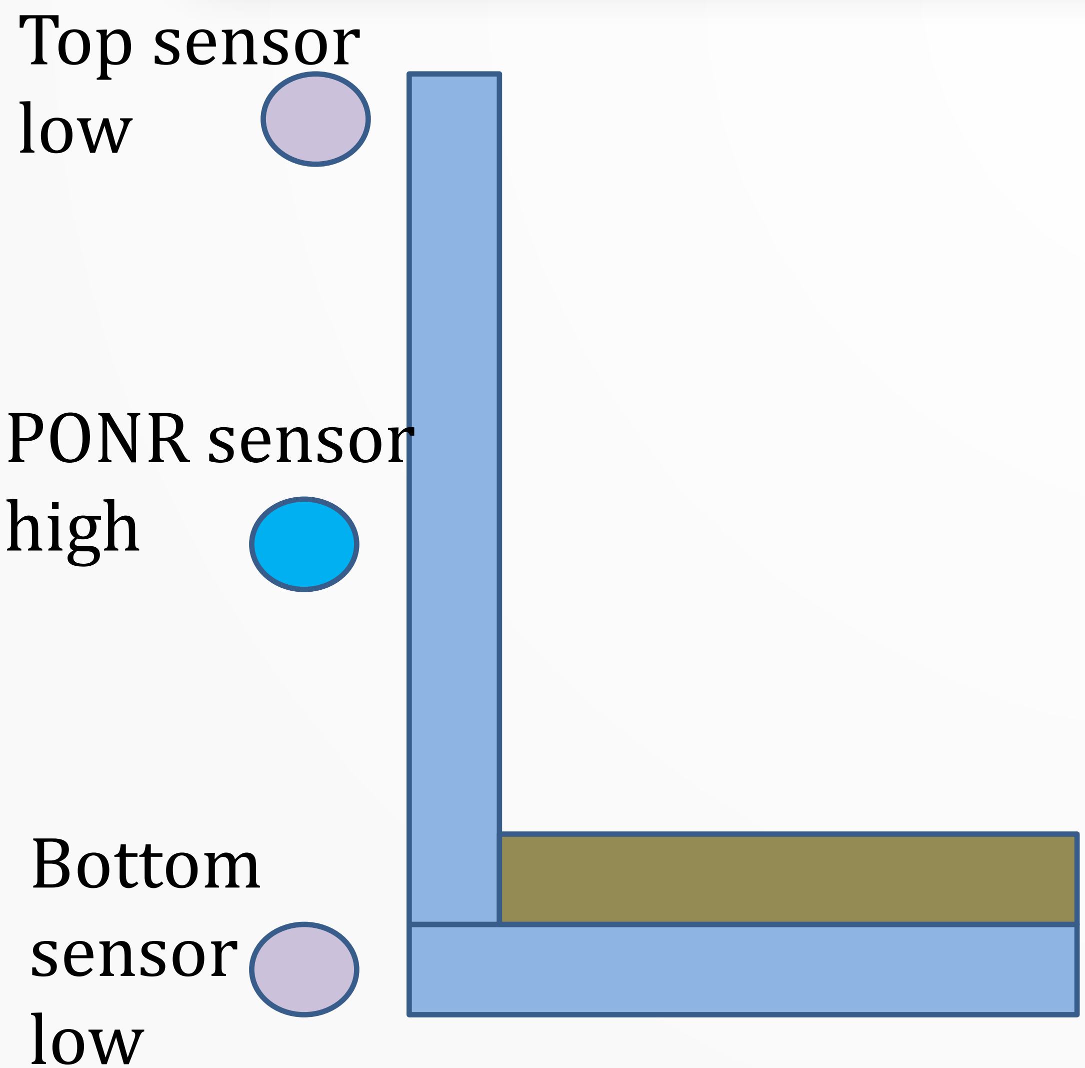


The motor is off. The plunger is at the bottom.



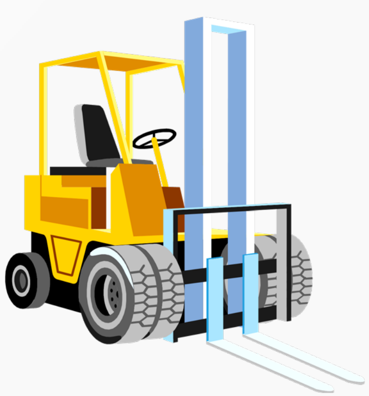
The user has released the button.

A Safety-Critical System

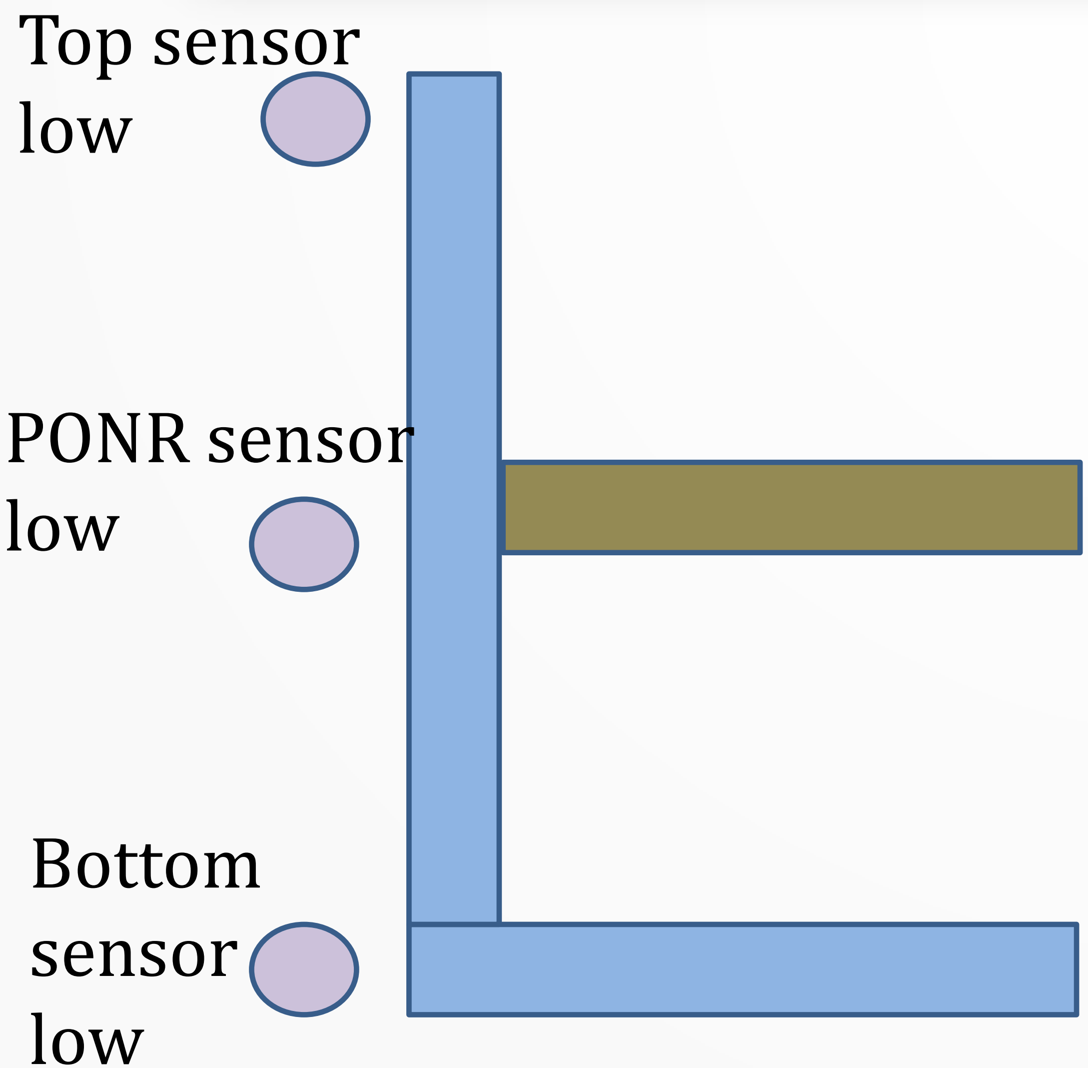


The motor turns on. The plunger starts rising.

The user has released the button.

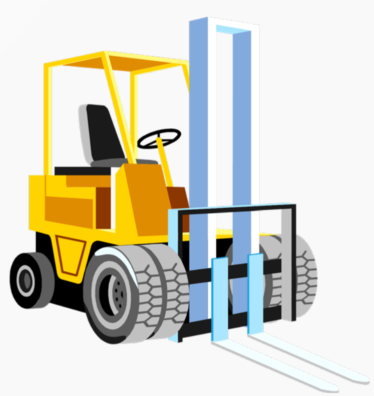


A Safety-Critical System

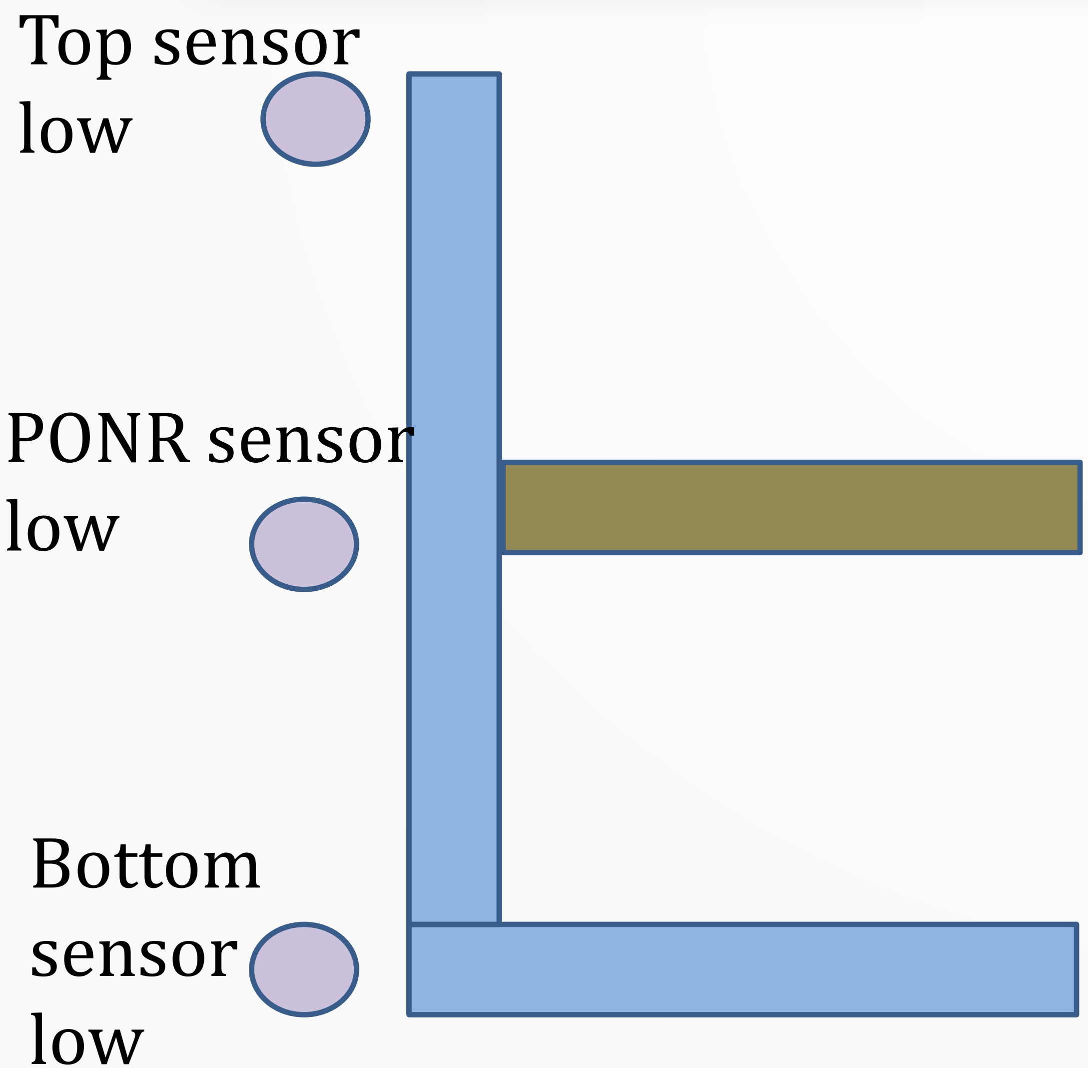


The motor turns on. The plunger starts rising.

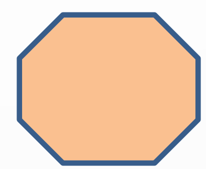
The user has released the button.



A Safety-Critical System

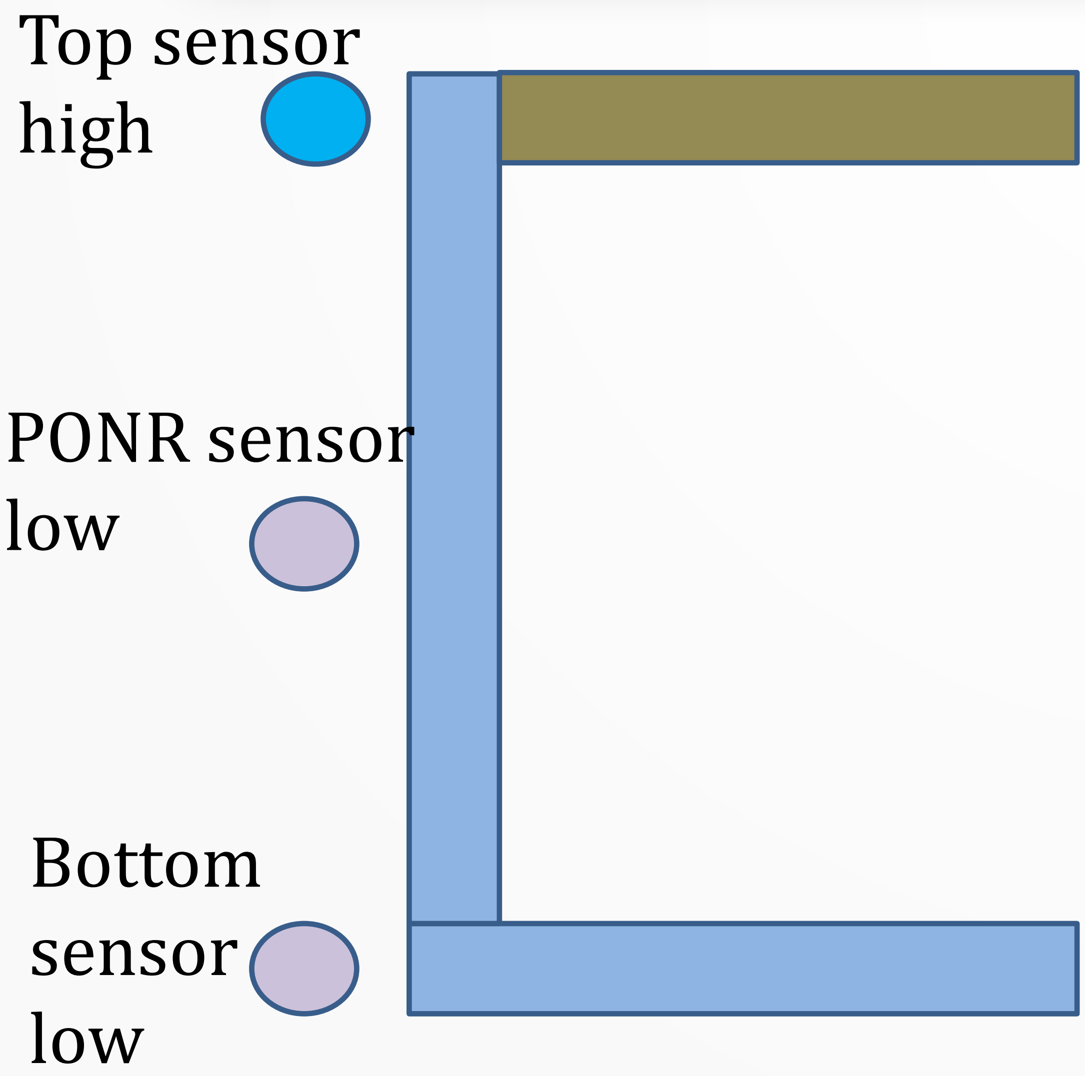


The motor turns on. The plunger starts rising.



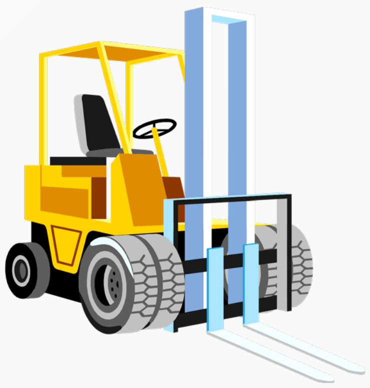
The user has released the button.

A Safety-Critical System

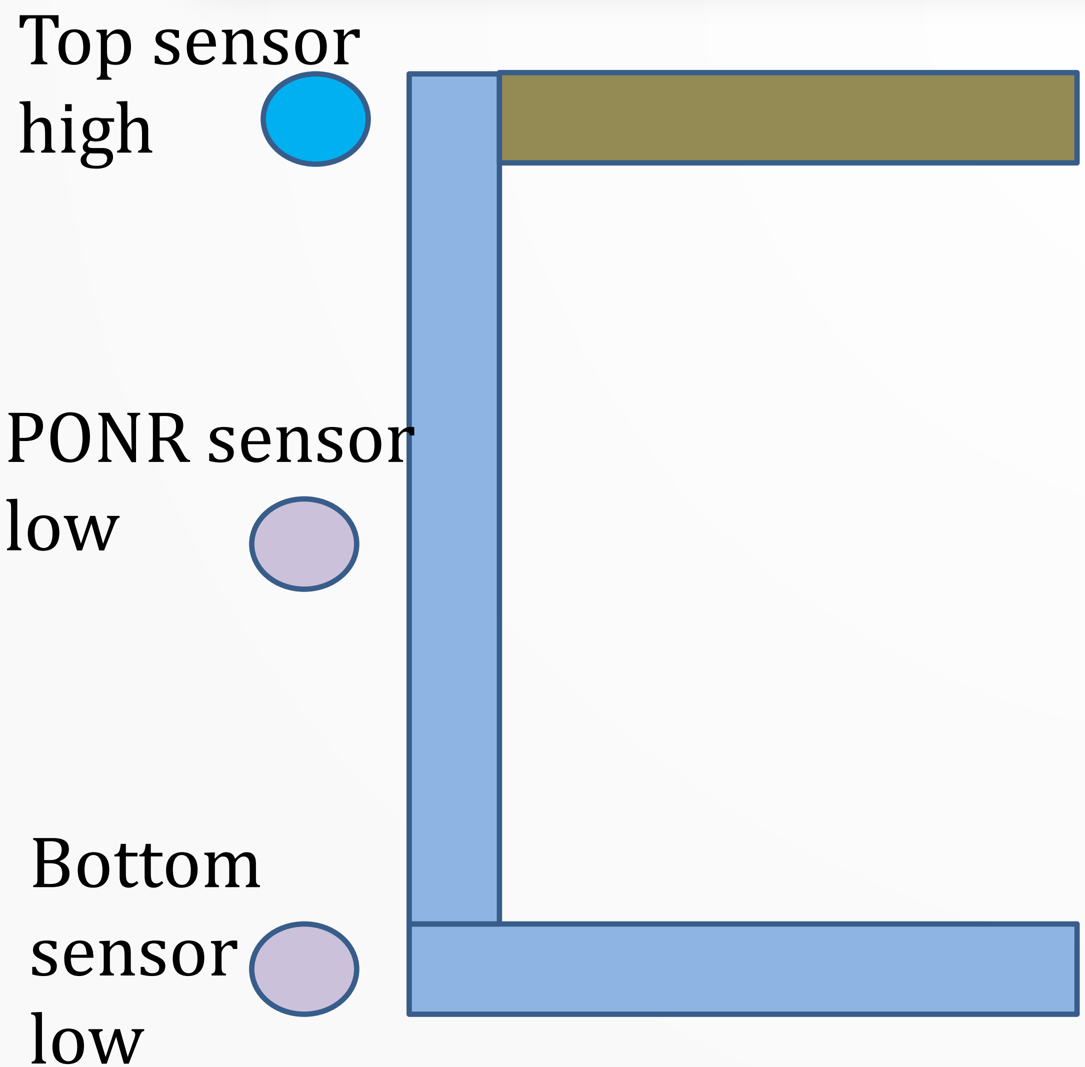


The motor stays on.
The plunger stays at the top.

The user has released the button.

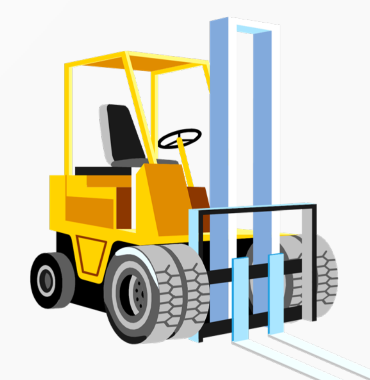


A Safety-Critical System



The motor stays on.
The plunger stays at the top.

The user pushes the button.

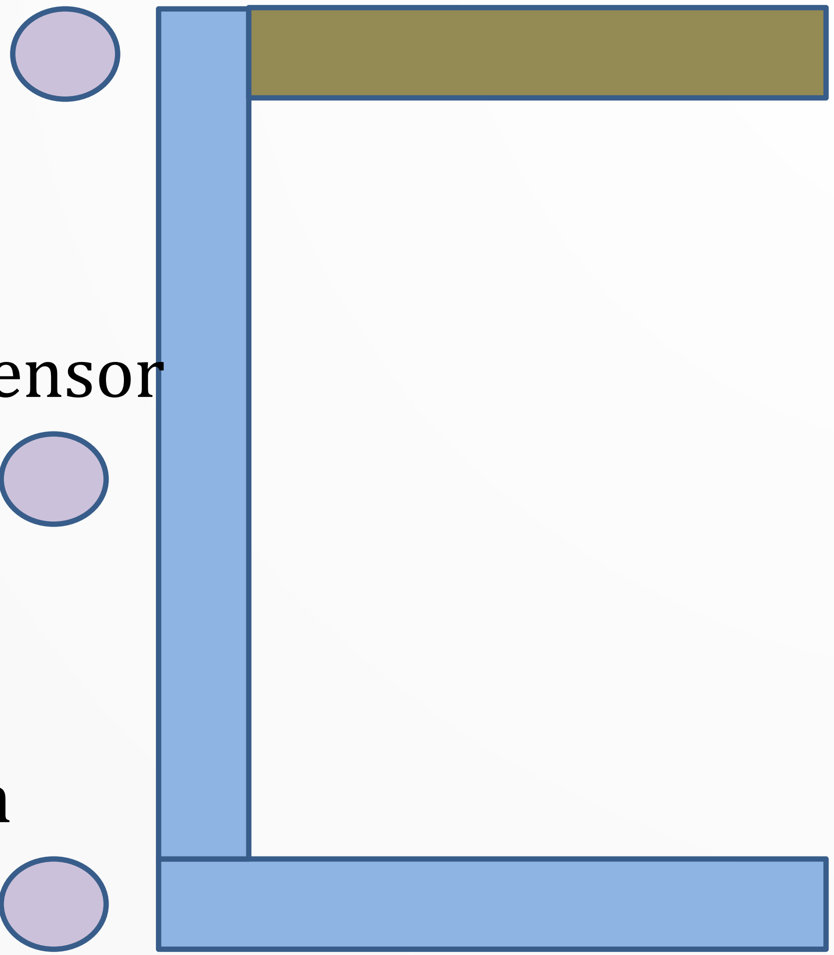


A Safety-Critical System

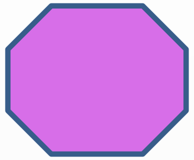
Top sensor low

PONR sensor low

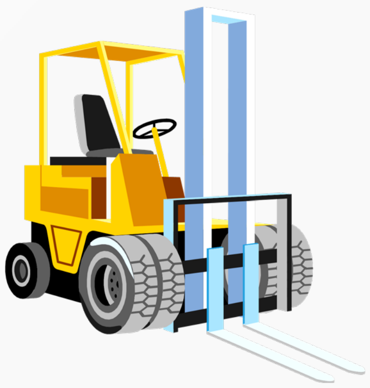
Bottom sensor low



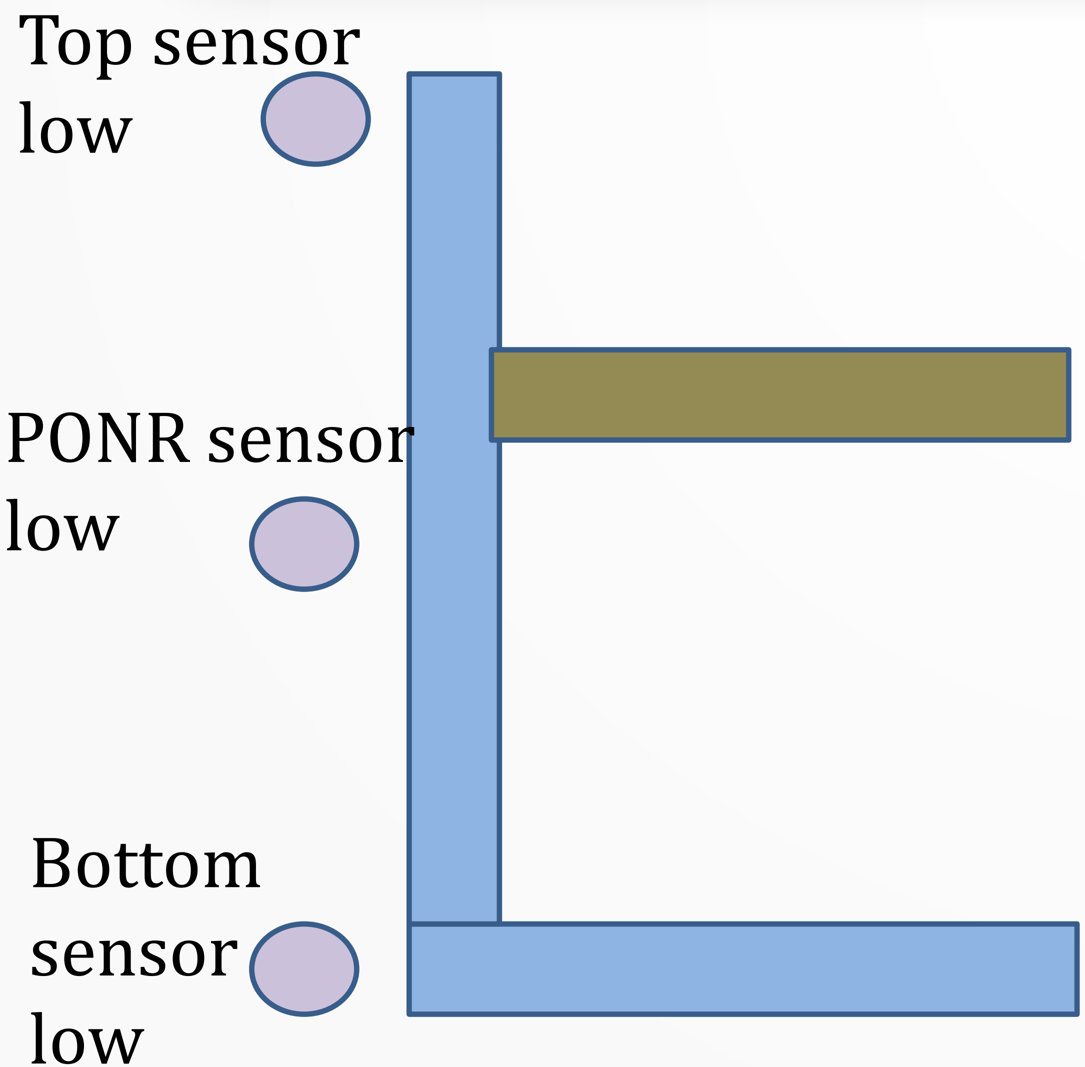
The motor turns off.
The plunger starts falling.



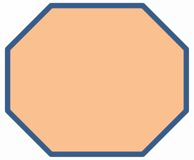
The user pushes the button.



A Safety-Critical System



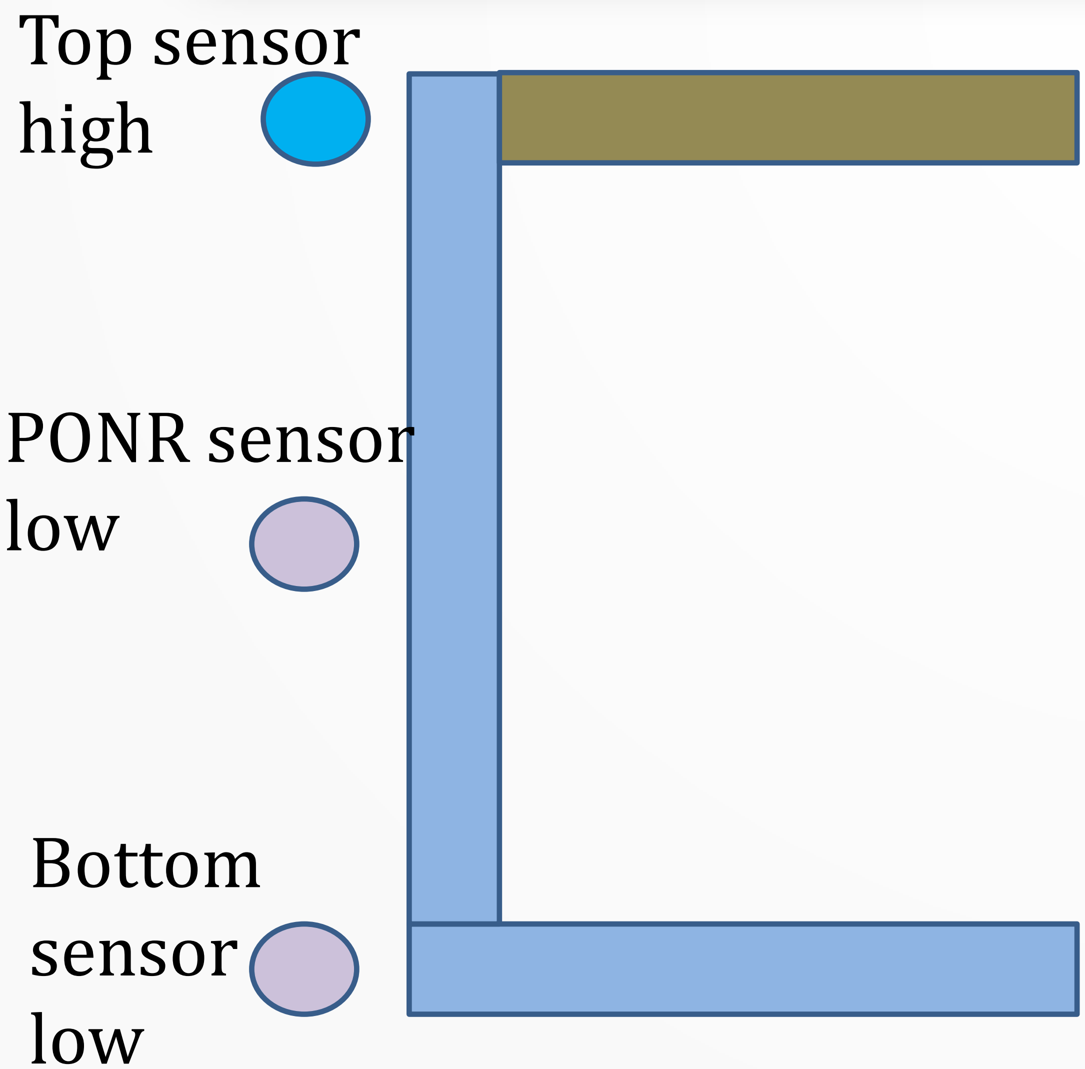
Safety abort:



The user releases the button.

The motor turns back on.
The plunger rises again.

A Safety-Critical System



Safety abort:

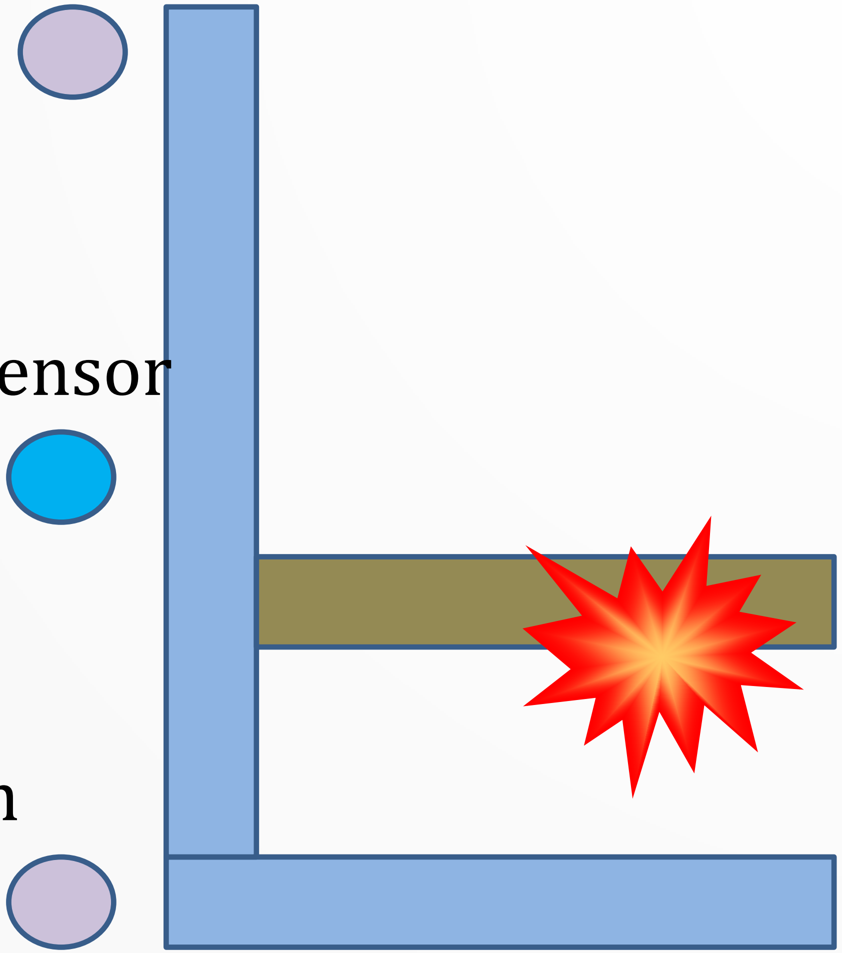
The user releases the button below the PONR.

The plunger is falling below the PONR. The motor is off.

A Safety-Critical System

Top sensor low

PONR sensor high



Bottom sensor low

The user releases the button.

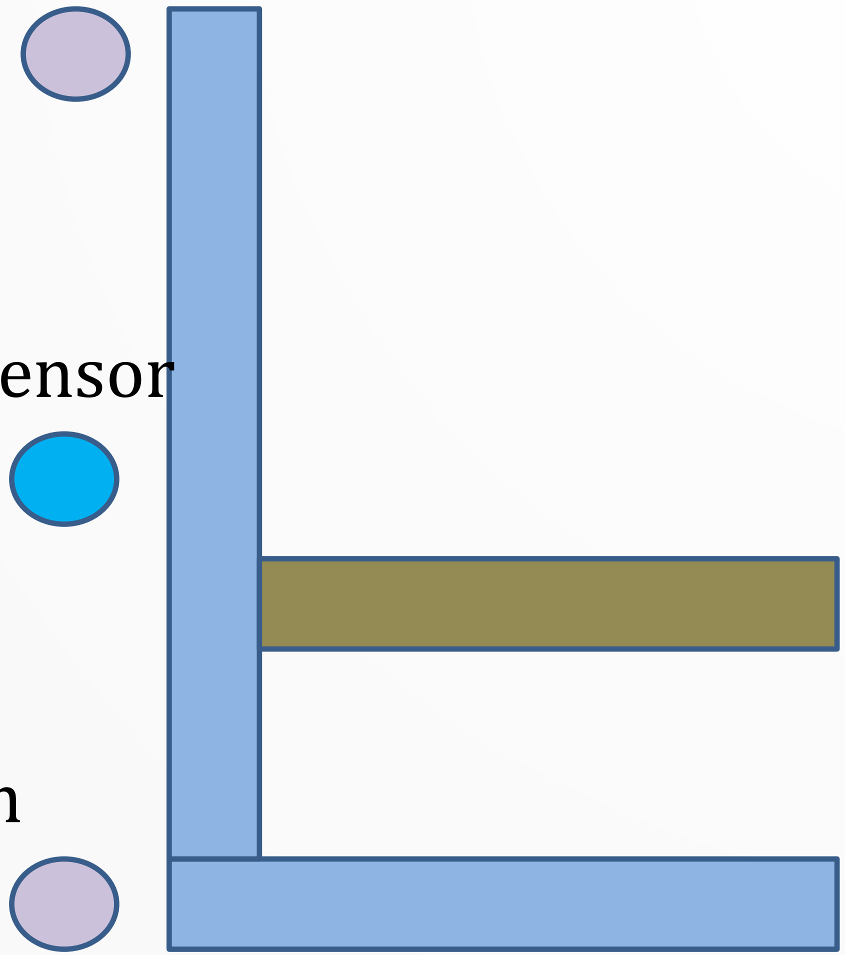
DANGER: The motor cannot turn on below the Point-of-no-return.

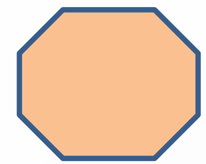
A Safety-Critical System

Top sensor low

PONR sensor high

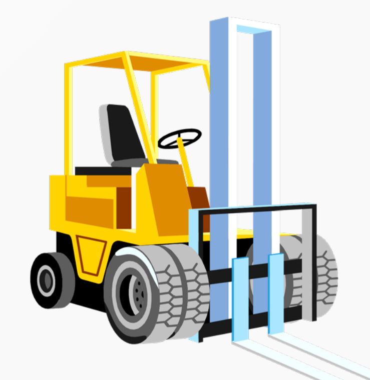
Bottom sensor low



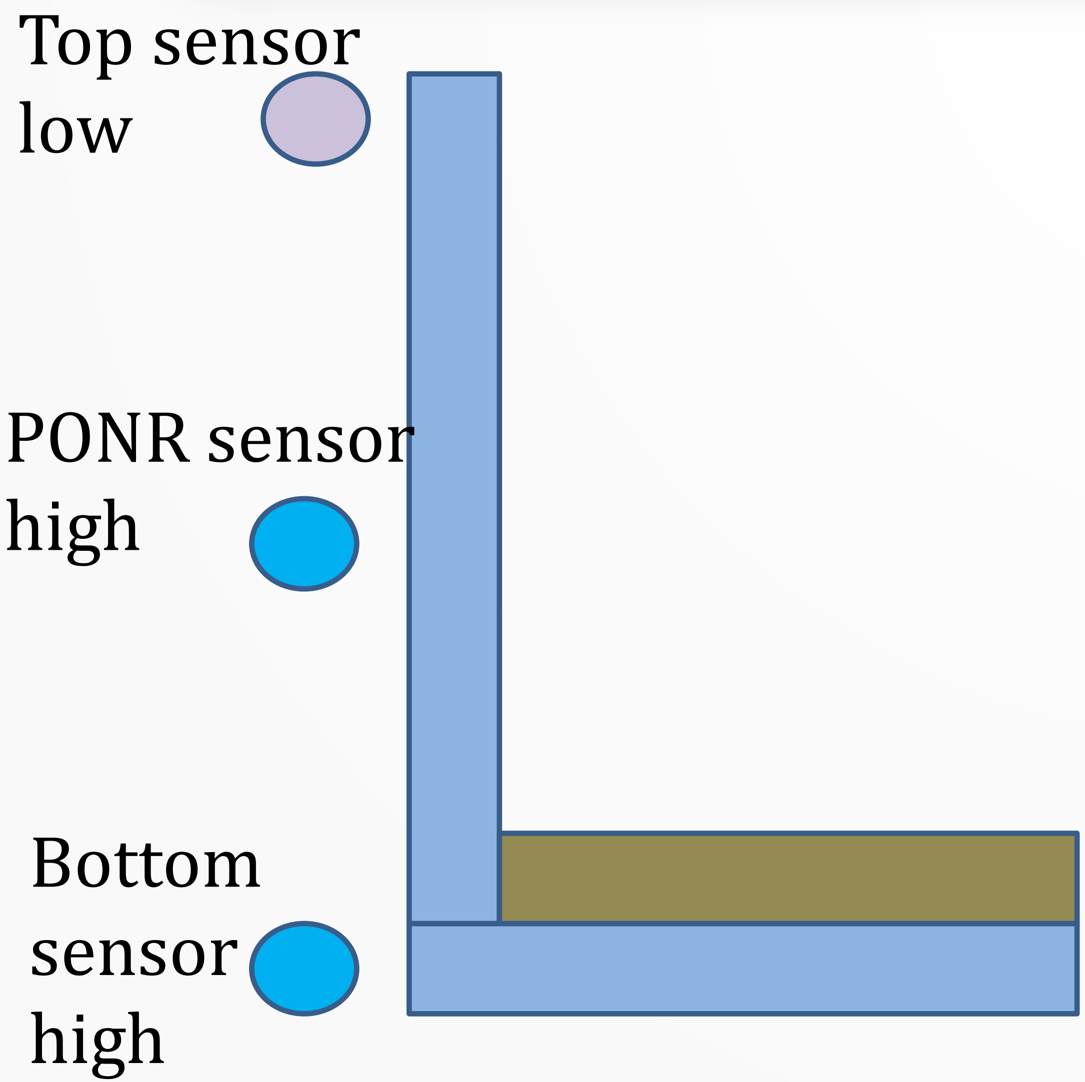


The user releases the button.

DANGER: The motor cannot turn on below the Point-of-no-return.

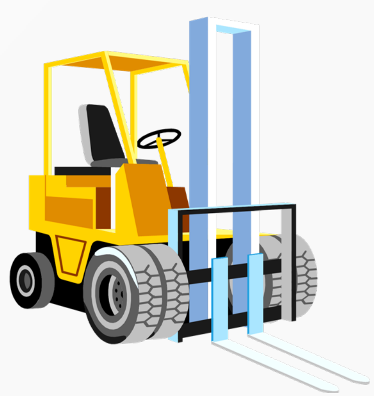


A Safety-Critical System

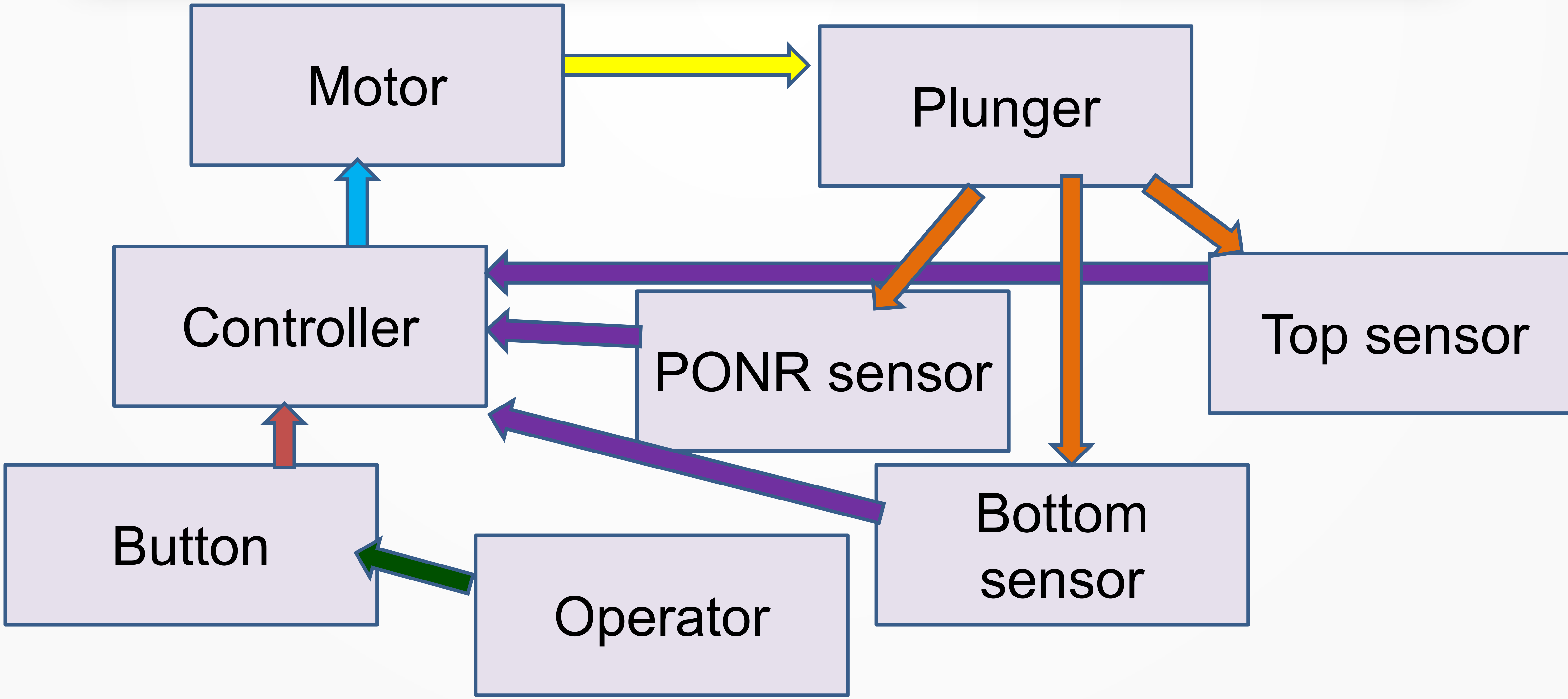


The user releases the button.

The plunger reaches the bottom and immediately starts rising again.



The System


Motor
Plunger
Controller
PONR sensor
Top sensor
Button
Operator
Bottom sensor

Communicating by Message-Passing

Concurrent processes can communicate by **shared-variable** or **message passing**.

The plunger falls below the PONR.

The PONR sensor detects this and changes to the high state (modelled by the plunger sending a message out to the PONR sensor).

The PONR sensor sends a message to the controller.

Example Trace

Example trace:
Plunger = falling
Plunger = atBottom
BottomSensor = high
Controller reads BottomSensor
Controller sends out TurnOn message to the motor
Motor = on
Plunger = risingBelowPONR
BottomSensor = low
Plunger = risingAbovePONR
PONRSensor = low

Model checking

Safety property in natural language

Th1: Uncommanded closing: Plunger should not start falling without the operator pressing the button.

Th2: Motor on below PONR: The motor should not turn on when the plunger is falling below the PONR.

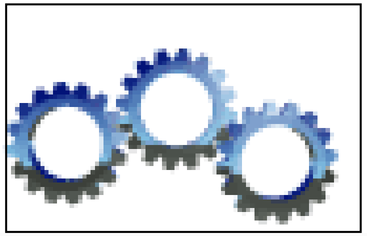
Th3: Loss of abort: If the plunger is falling above the PONR and the operator releases the button, the motor should turn on.

Th4: Plunger falling before reaching the top: The motor should not turn off unless the plunger is at the top.

```
th1:  THEOREM behavior |- G((plunger=plunger_at_top AND
operator=operator_released_button) =>
(electric_Motor=electric_Motor_on));

th2:  THEOREM behavior |- G((plunger=plunger_falling_fast) =>
(electric_Motor=electric_Motor_off));

th3:  THEOREM behavior |- G(F(plunger=plunger_falling_fast)) =>
G((plunger=plunger_falling_slow AND
operator=operator_released_button) =>
U(plunger=plunger_falling_slow, electric_Motor=electric_Motor_on));

th4:  THEOREM behavior |-
G(NOT((plunger=plunger_rising_below_PONR OR
plunger=plunger_rising_above_PONR) AND
(electric_Motor=electric_Motor_off)));
```

Temporal logic specification

Specification



Model checker

Proved

Counterexample

Safety Properties

Th1: Uncommanded closing: The plunger should not start falling without the operator pressing the button.

Th2: Motor on below PONR: The motor should not turn on when the plunger is falling below the PONR.

Th3: Loss of abort: If the plunger is falling above the PONR and the operator releases the button, the motor should turn on.

Th4: Plunger falling before reaching the top: The motor should not turn off unless the plunger is at the top.

Safety Properties

Th1: Uncommanded closing: The plunger should not start falling without the operator pressing the button.

G((plunger = atTop AND operator = releasedButton) => (electric_Motor = on));

Safety Properties

Th2: Motor on below PONR: The motor should not turn on when the plunger is falling below the PONR.

G((plunger = fallingFast) => (electric_Motor = off));

Safety Properties

Th3: Loss of abort: If the plunger is falling above the PONR and the operator releases the button, the motor should turn on.

G((plunger = fallingSlow AND operator = releasedButton) =>
F(electric_Motor = on));

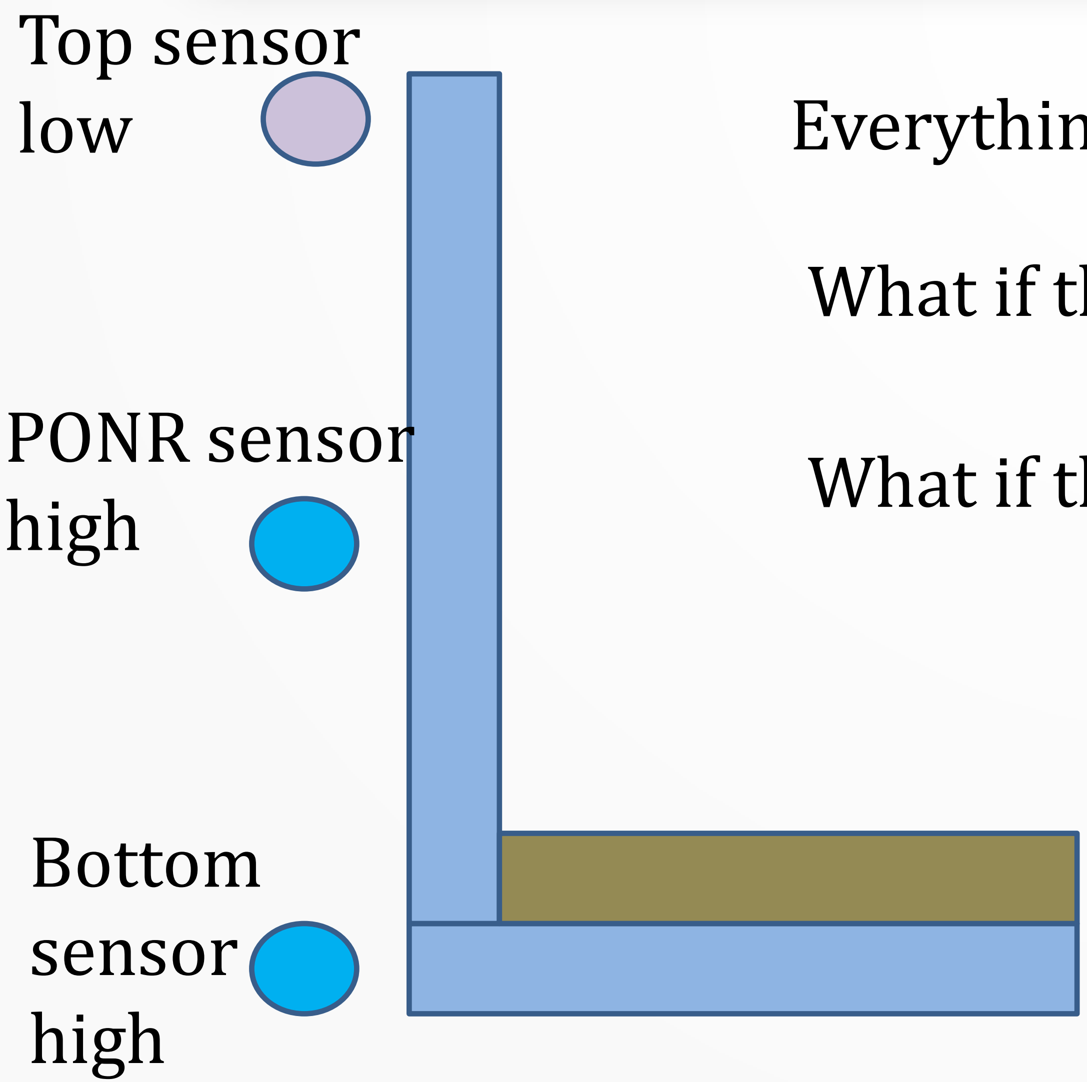
G((plunger = fallingSlow AND operator = releasedButton) =>
(plunger = fallingSlow **U** electric_Motor = on));

Safety Properties

Th4: Plunger falling before reaching the top: The motor should not turn off unless the plunger is at the top.

G(NOT((plunger = risingBelowPONR OR plunger = risingAbovePONR) AND (electric_Motor = off)));

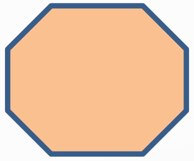
What can go wrong?



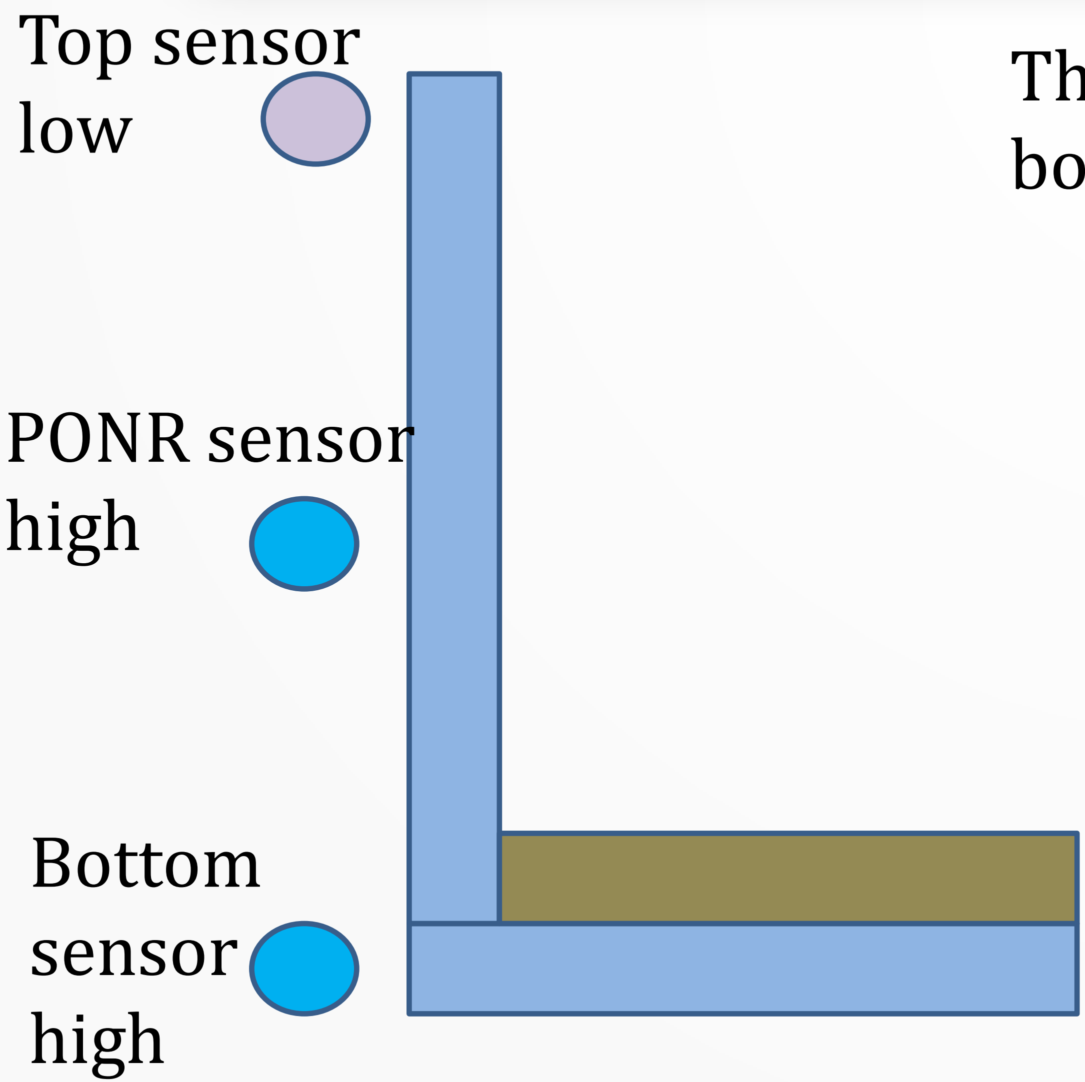
Everything looks ok. What can go wrong?

What if the sensors are broken?

What if the Bottom Sensor is stuck high?



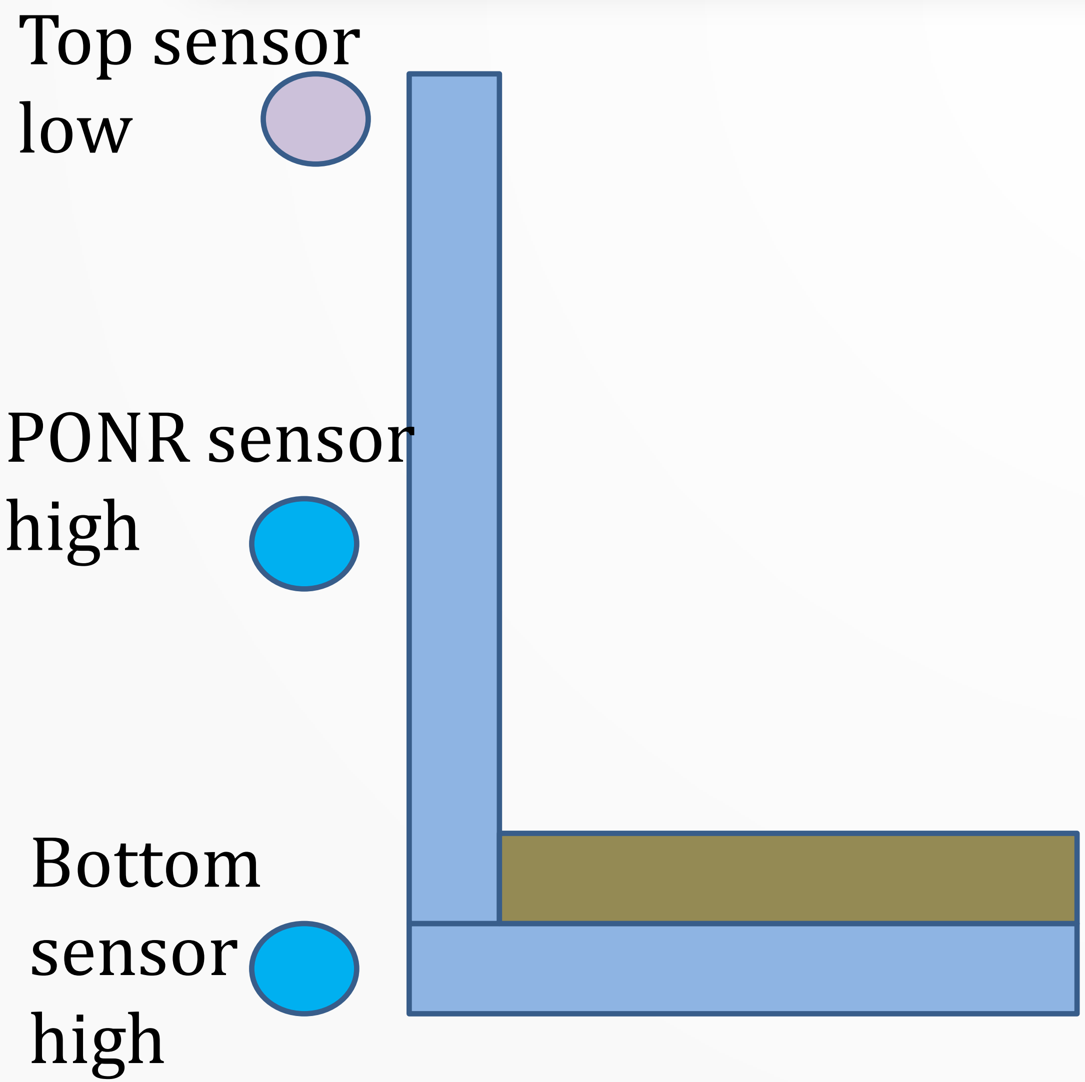
Bottom Sensor stuck high



The motor is off. The plunger is at the bottom.

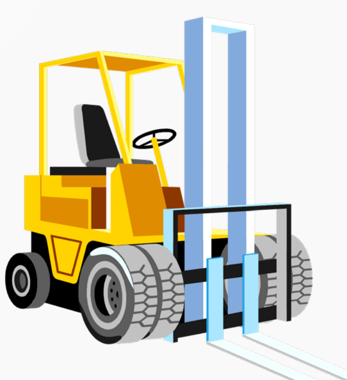
The user has released the button.

Bottom Sensor stuck high

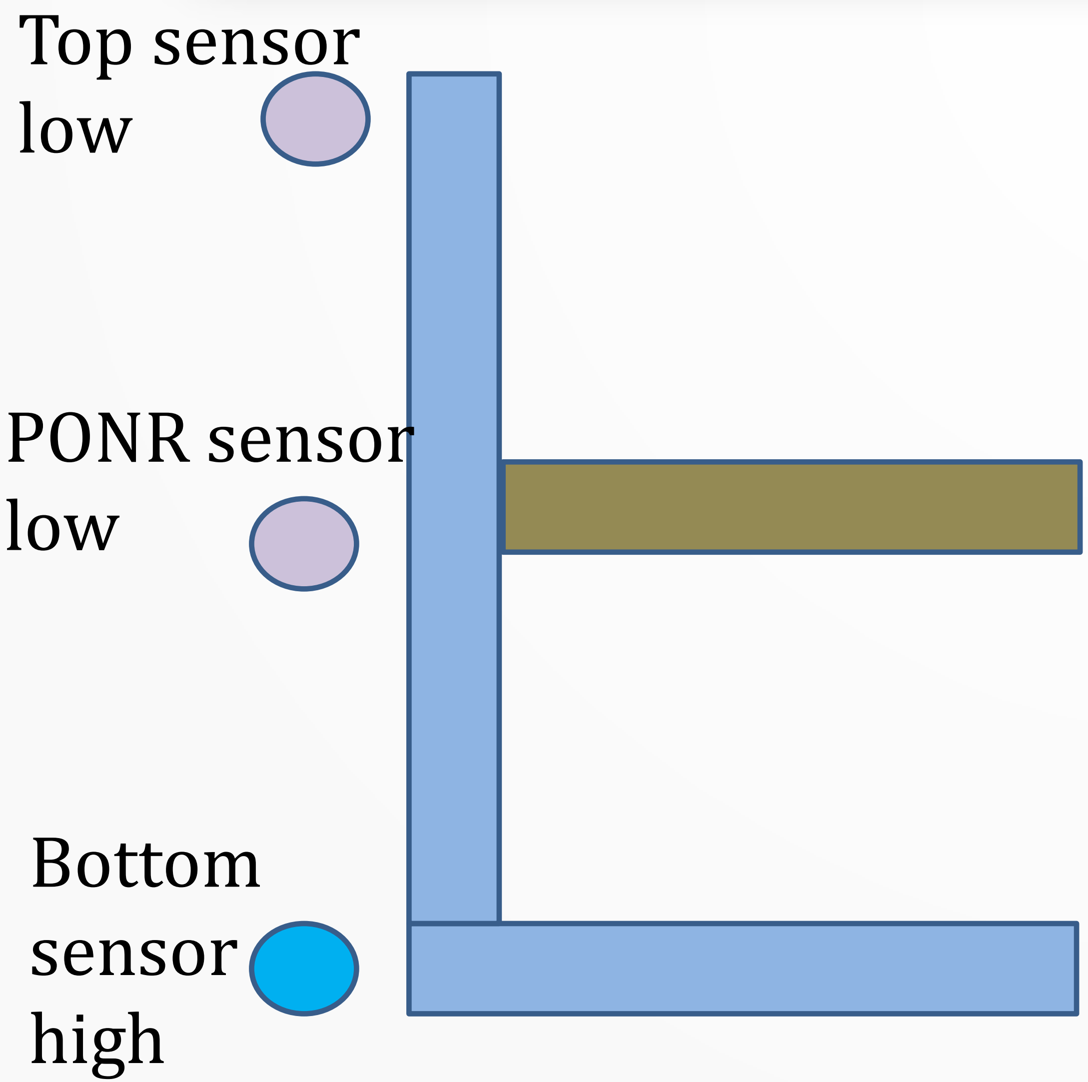


The motor turns on. The plunger starts rising.

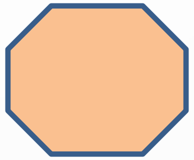
The user has released the button.



Bottom Sensor stuck high

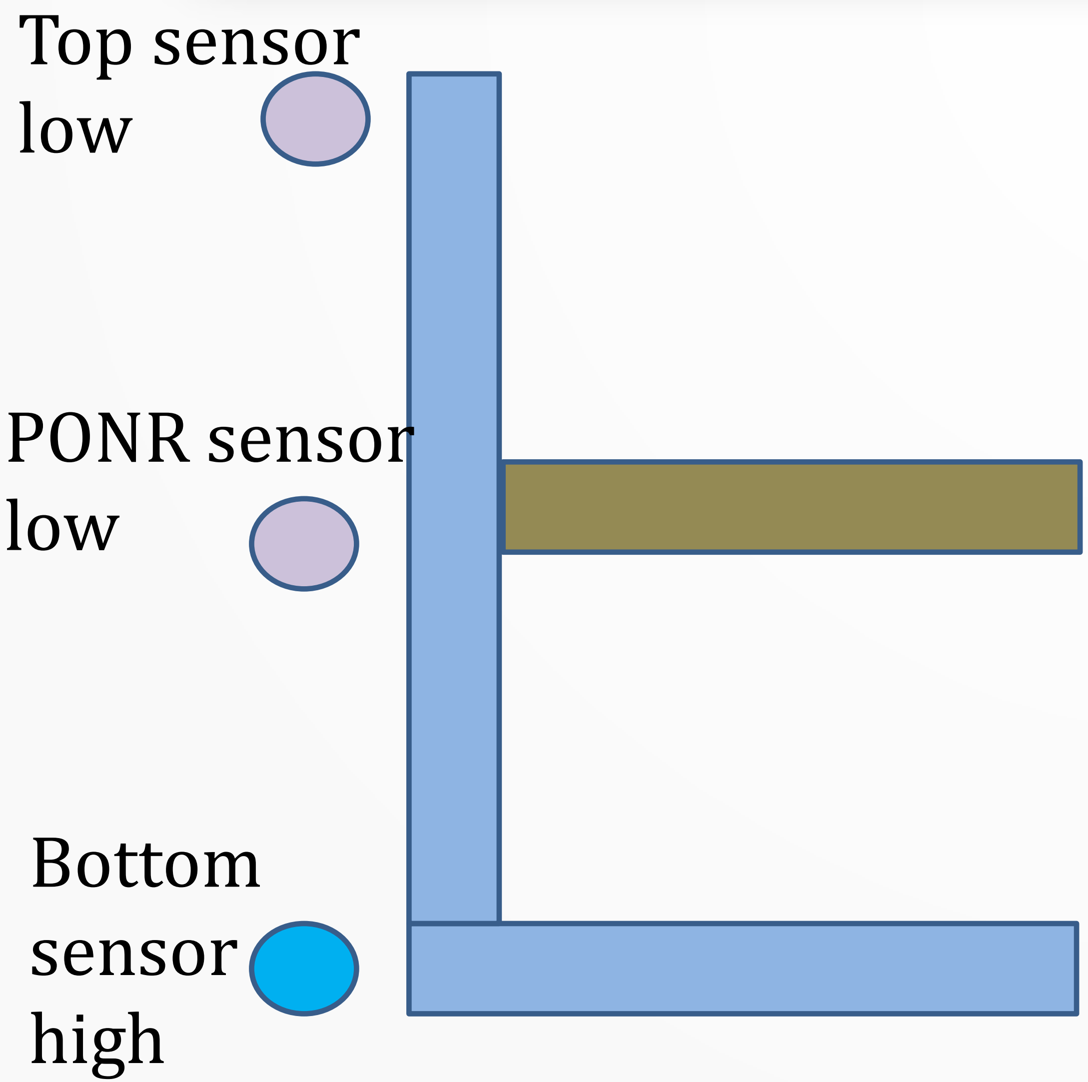


The motor turns on. The plunger starts rising.

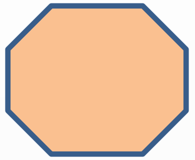


The user has released the button.

Bottom Sensor stuck high

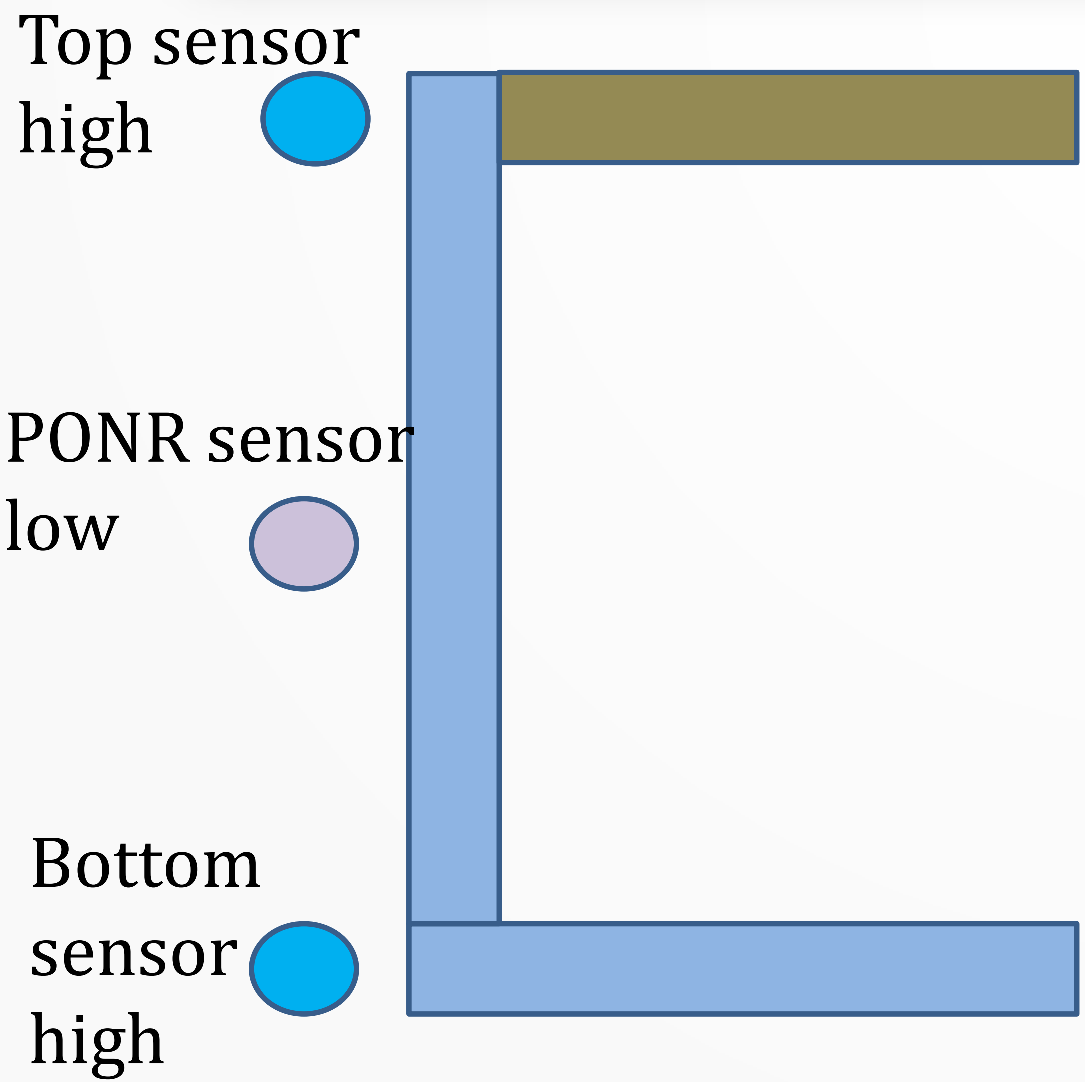


The motor turns on. The plunger starts rising.

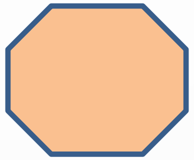


The user has released the button.

Bottom Sensor stuck high

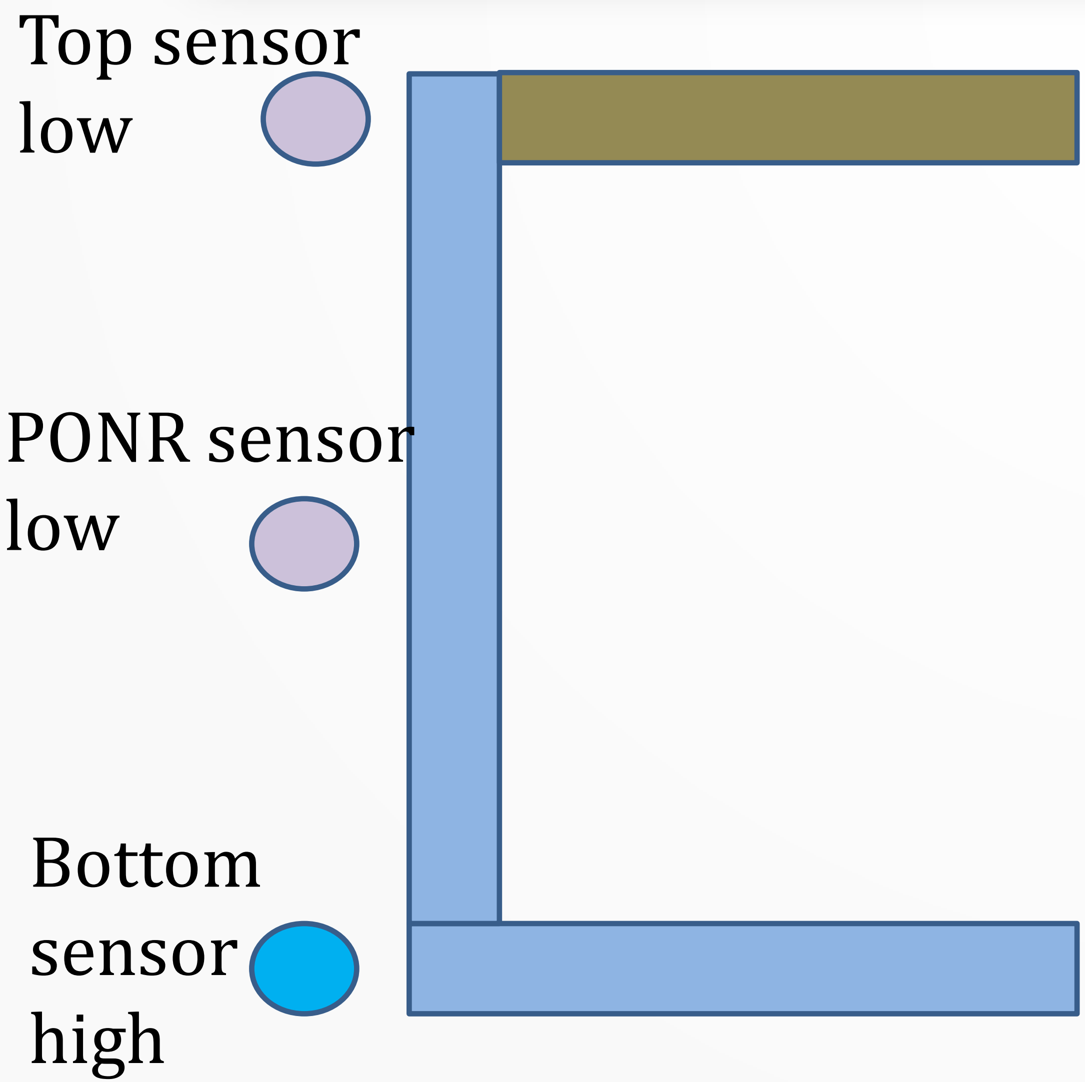


The motor stays on.
The plunger stays at the top.

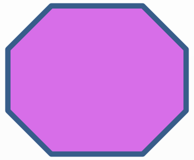


The user has released the button.

Bottom Sensor stuck high



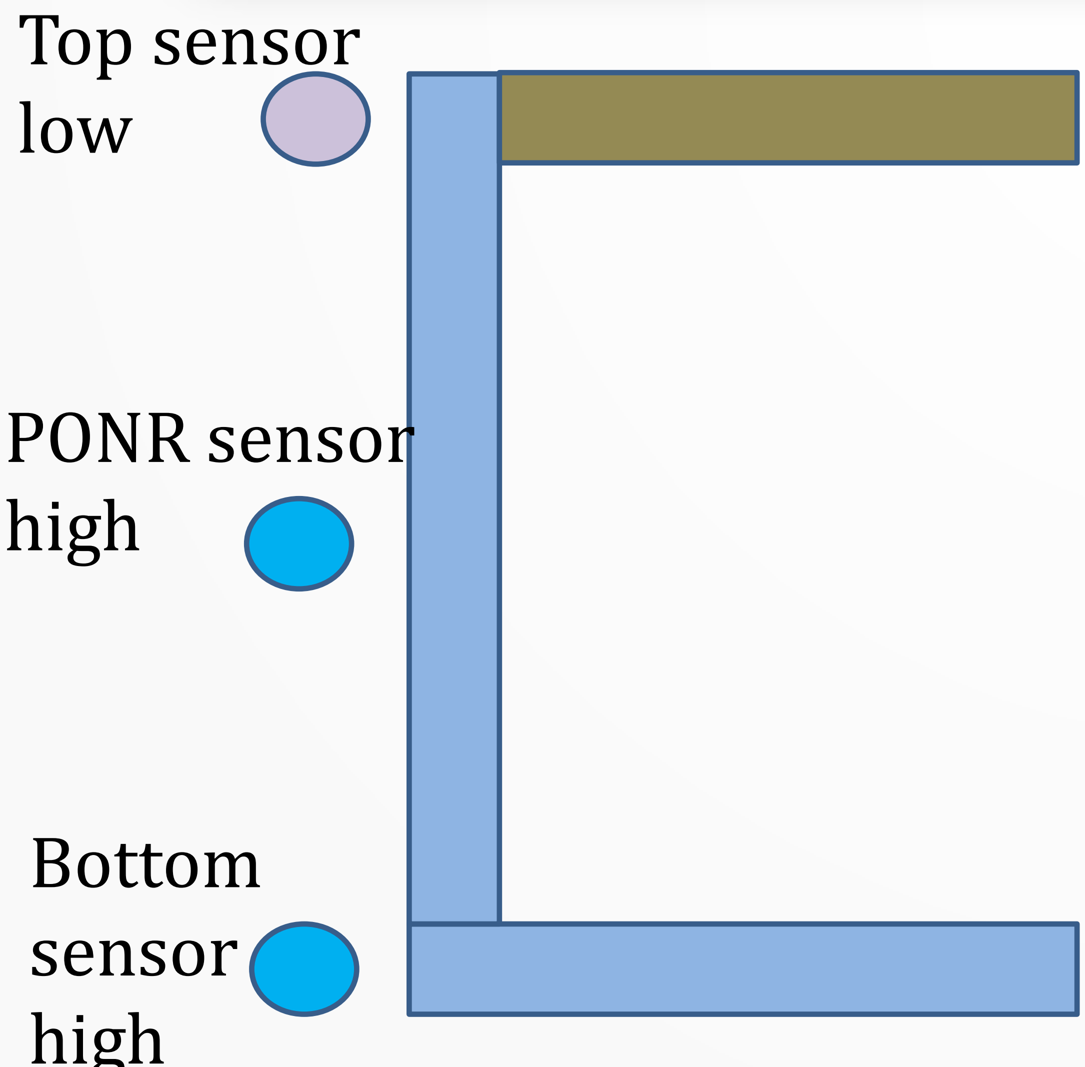
The plunger starts falling. The controller thinks it has reached the bottom and turns on the motor.



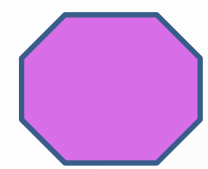
The user pushes the button.

No hazard, but the plunger won't fall.

Bottom Sensor stuck high half-way



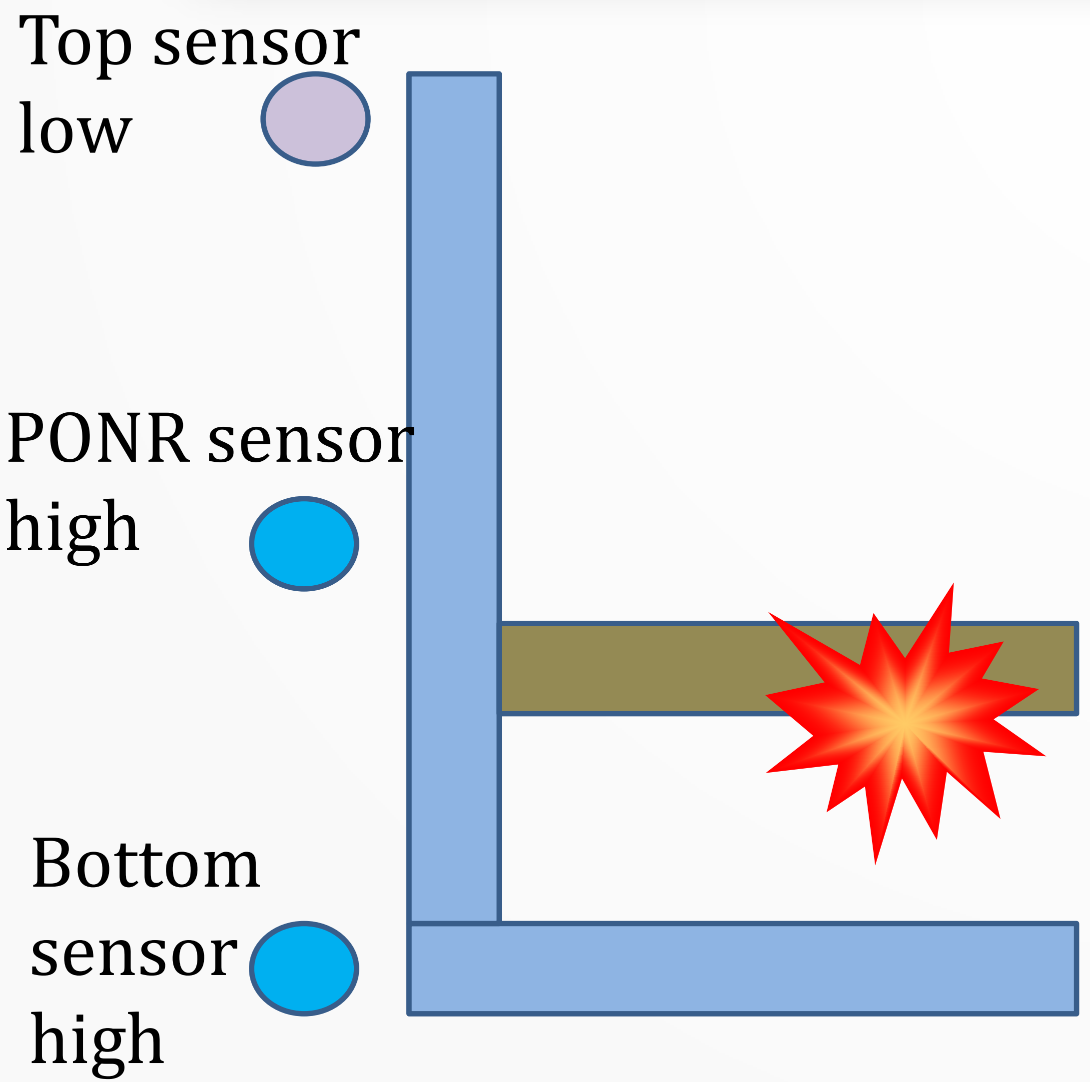
What if the bottom sensor breaks *after* the plunger has fallen below the PONR?



The user is still pushing the button.

The controller thinks the plunger reached the bottom and turns on the motor.

Bottom Sensor stuck high half-way



DANGER: The motor cannot turn on below the Point-of-no-return.

Bottom Sensor stuck high

Th2: Motor on below PONR: The motor should not turn on when the plunger is falling below the PONR.

The bottom sensor problem can violate Th2.

This is a serious safety hazard.

Another failure that can cause this is the **PONR sensor stuck low.**

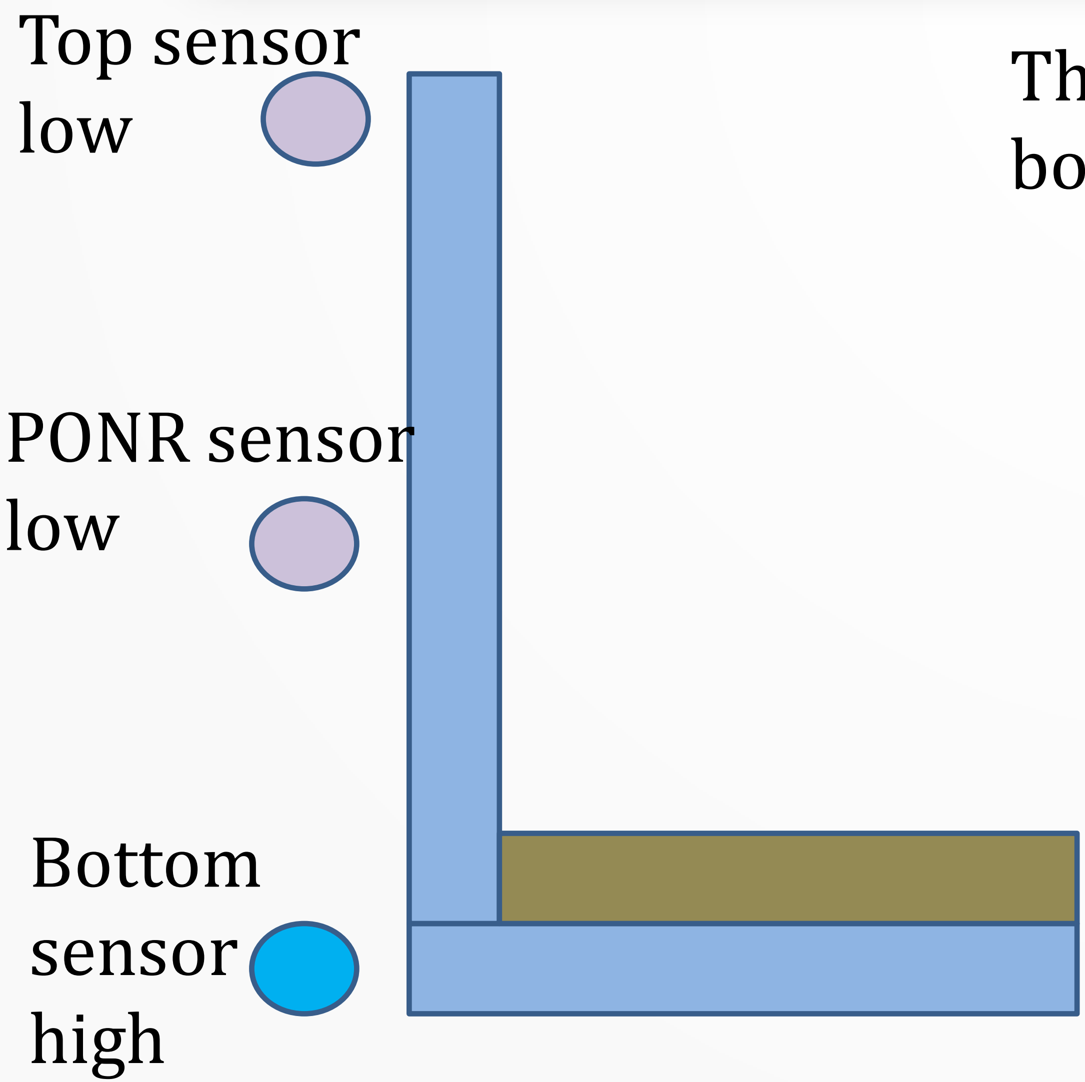
Safety Properties

Th2: Motor on below PONR: The motor should not turn on when the plunger is falling below the PONR.

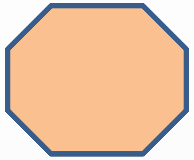
G((plunger = fallingFast) => (electric_Motor = off));

Counterexample: The plunger is falling fast, then the bottom sensor turns high, the controller reads the bottom sensor as high and turns on the motor.

PONR Sensor stuck low

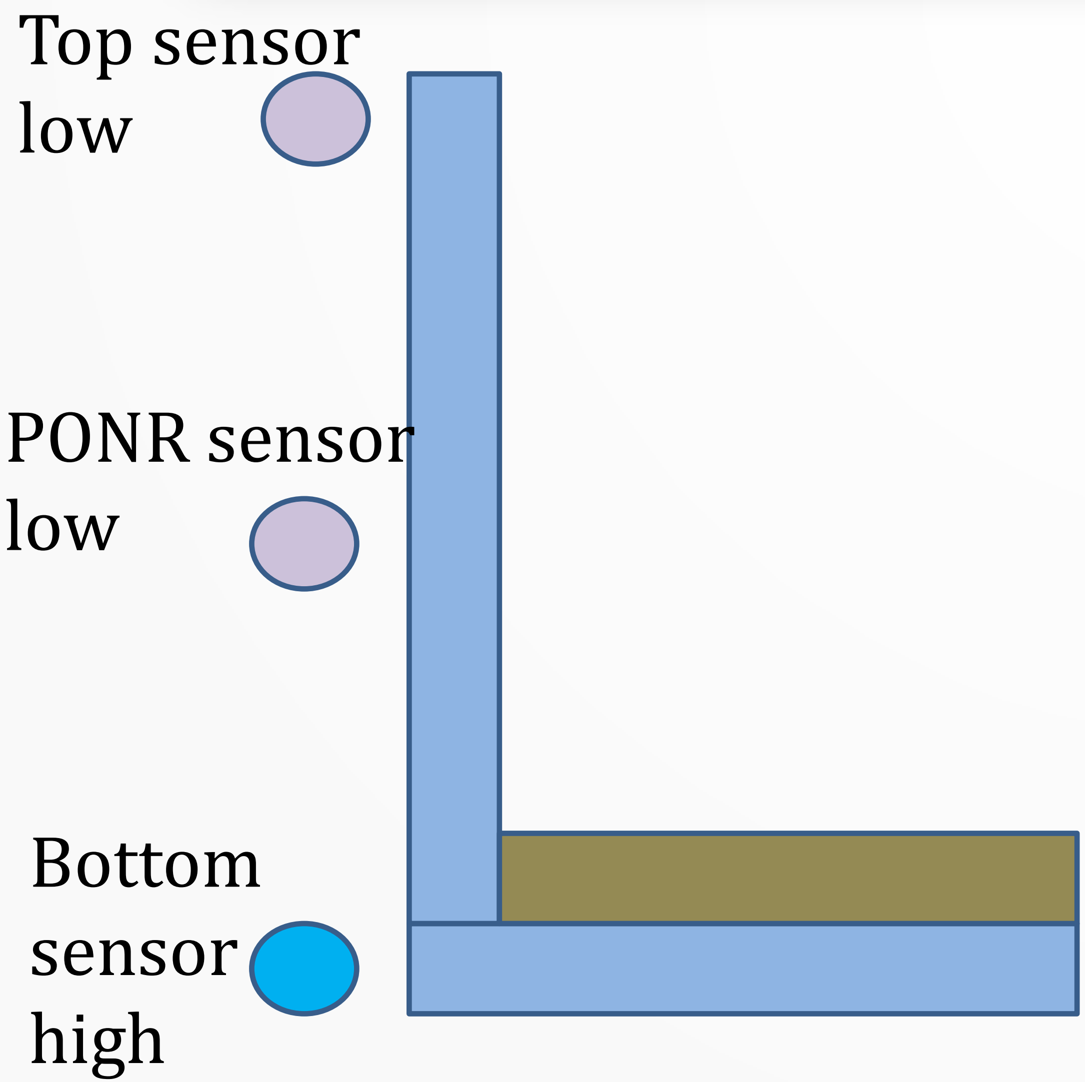


The motor is off. The plunger is at the bottom.



The user has released the button.

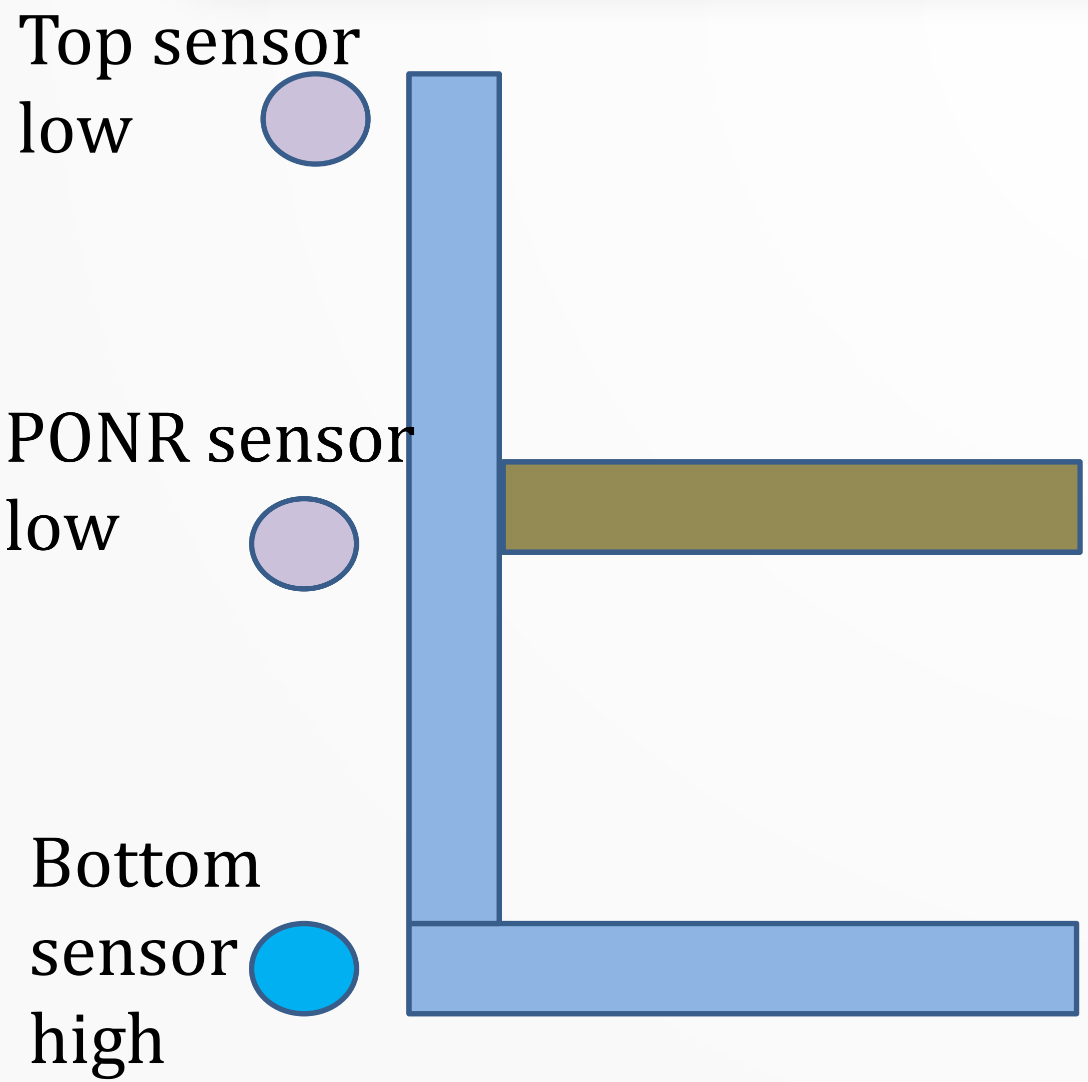
PONR Sensor stuck low



The motor turns on. The plunger starts rising.

The user has released the button.

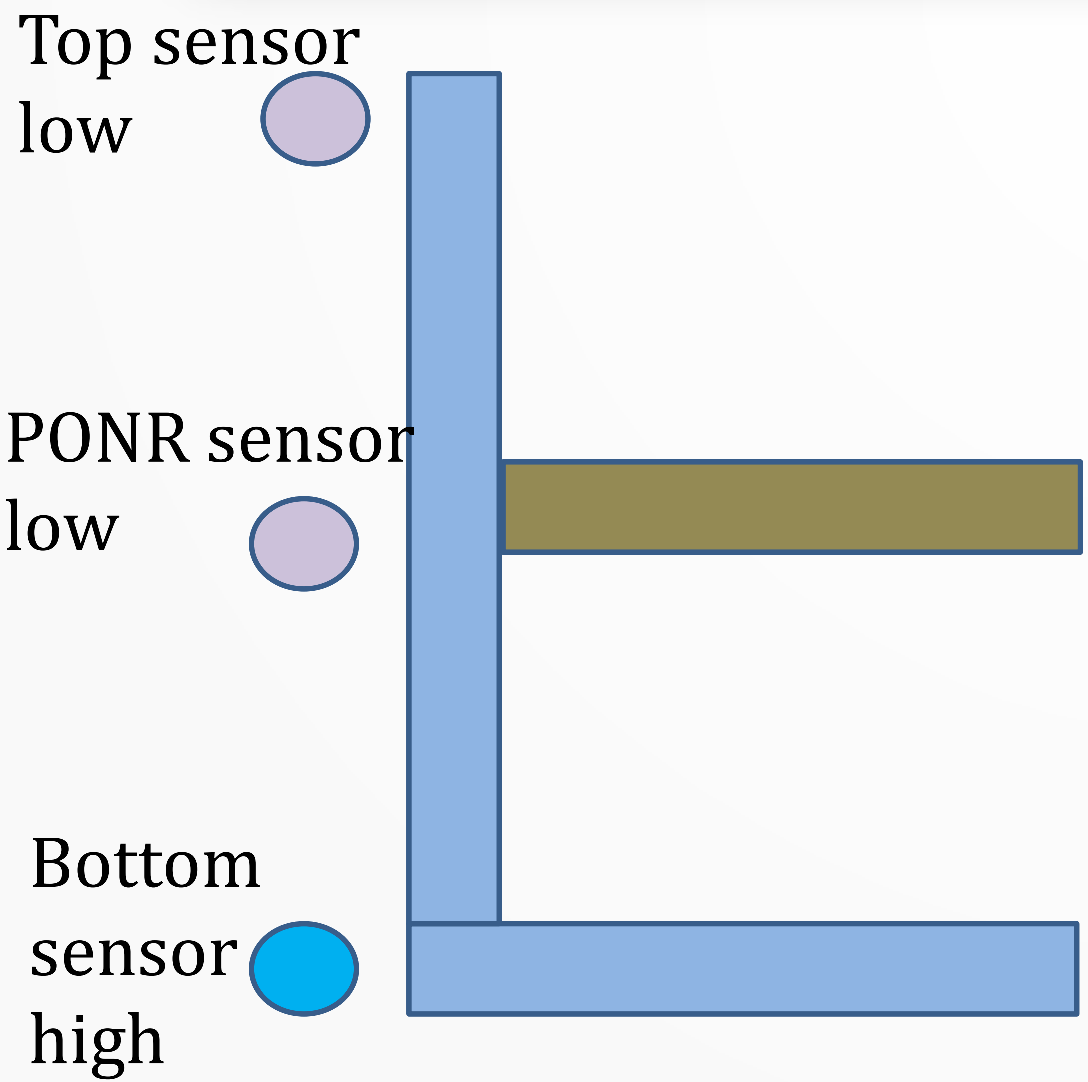
PONR Sensor stuck low



The motor turns on. The plunger starts rising.

The user has released the button.

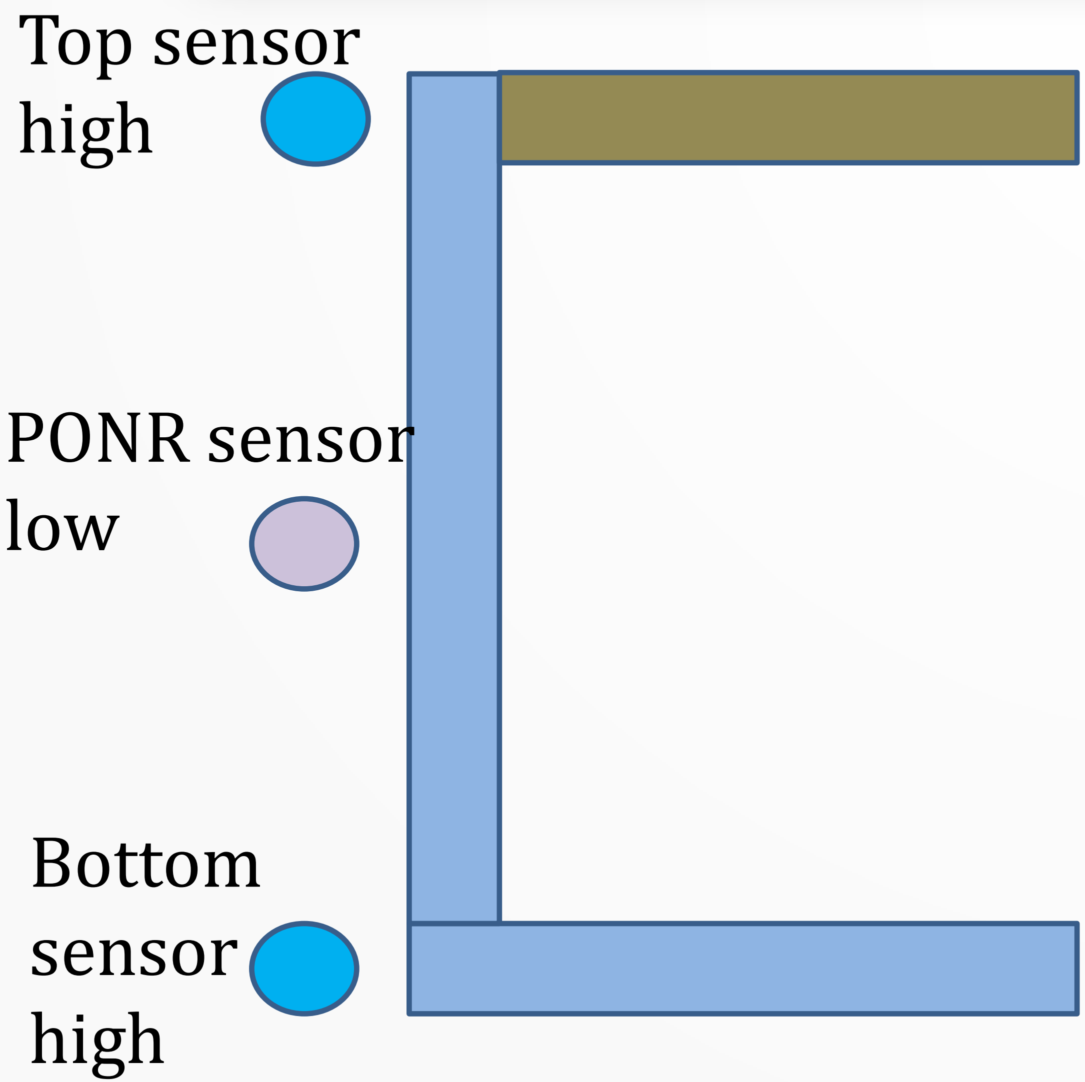
PONR Sensor stuck low



The motor turns on. The plunger starts rising.

The user has released the button.

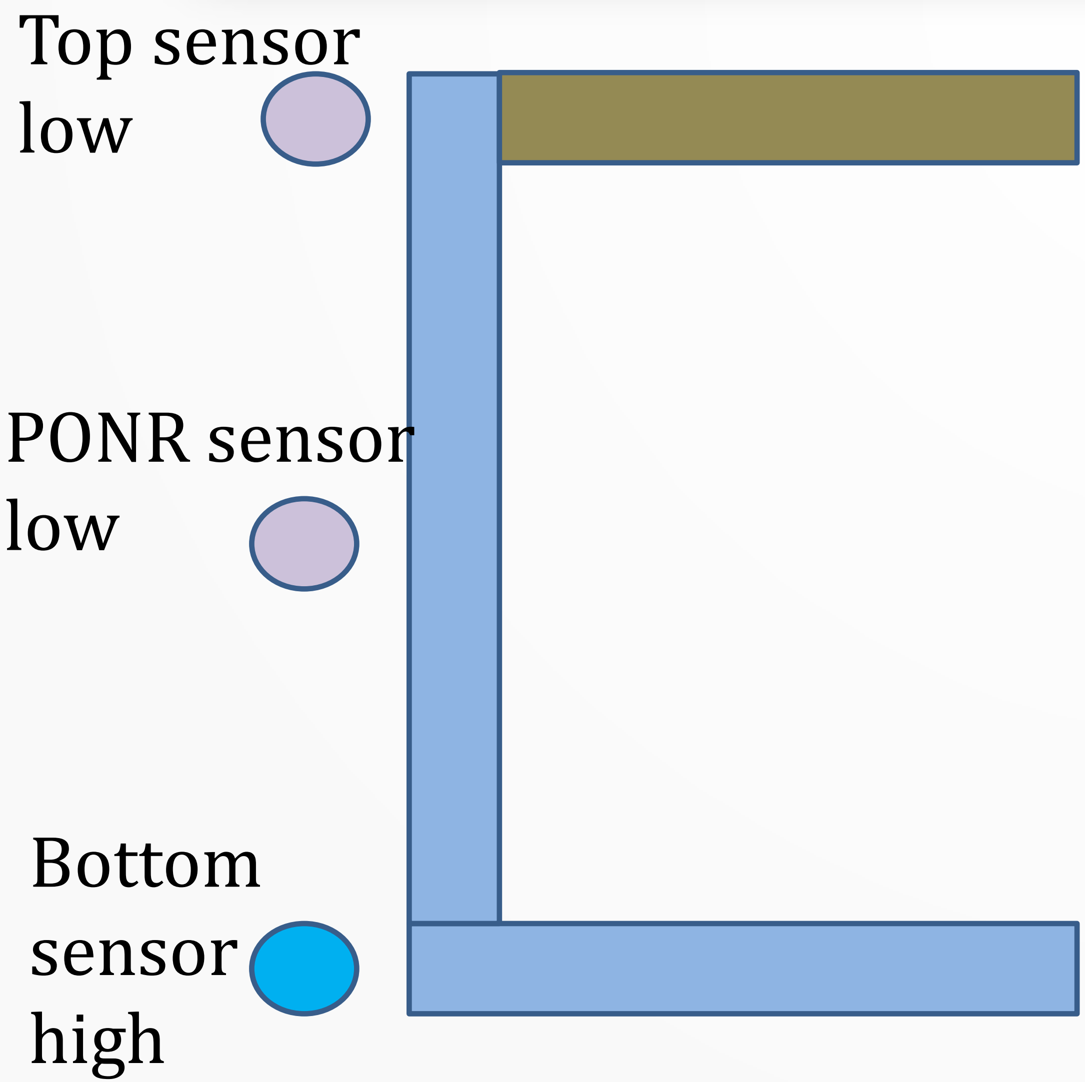
PONR Sensor stuck low



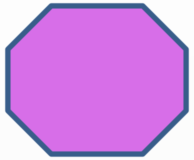
The motor stays on.
The plunger stays at the top.

The user has released the button.

PONR Sensor stuck low

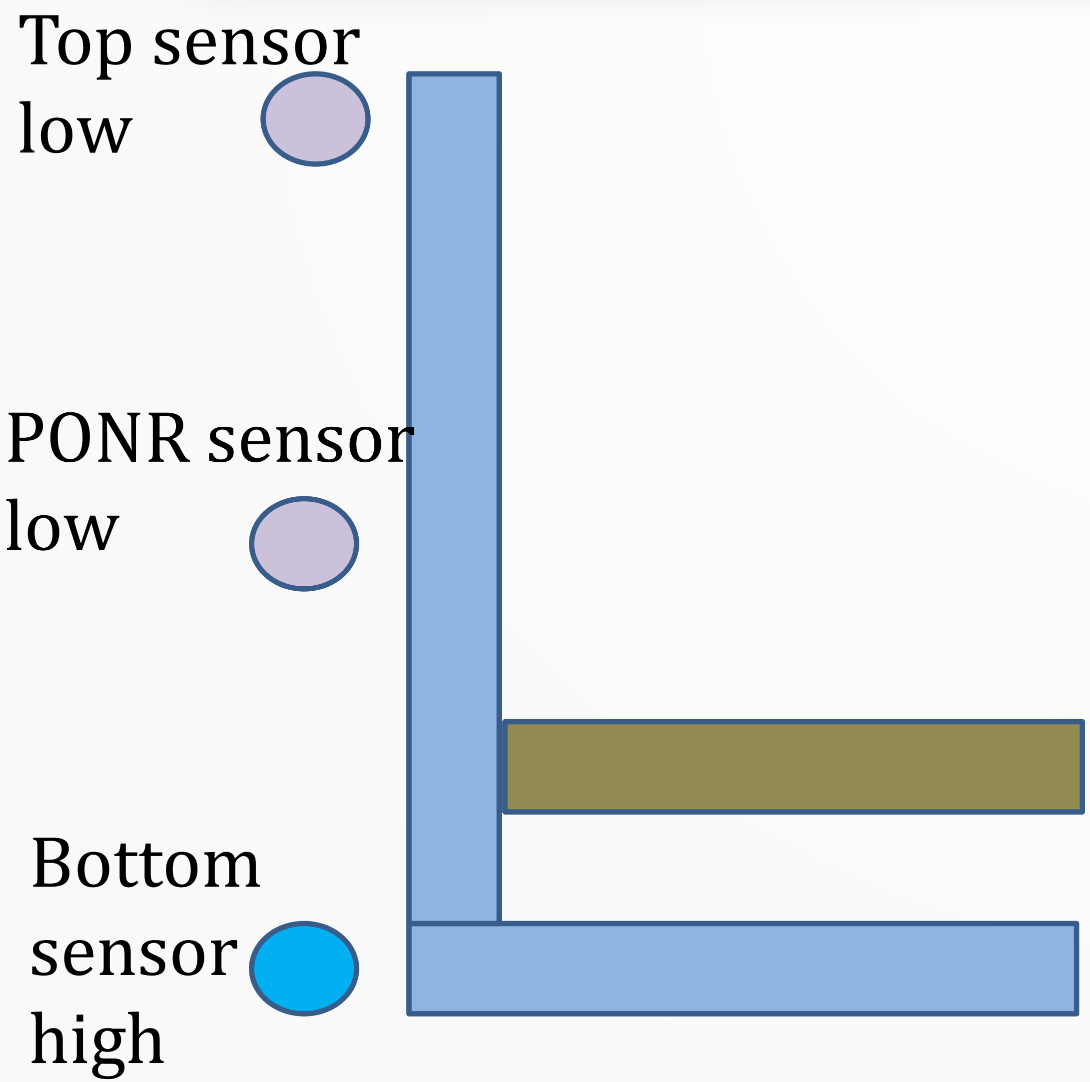


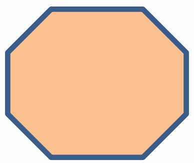
The motor turns off. The plunger starts falling.



The user pushes the button.

PONR Sensor stuck low

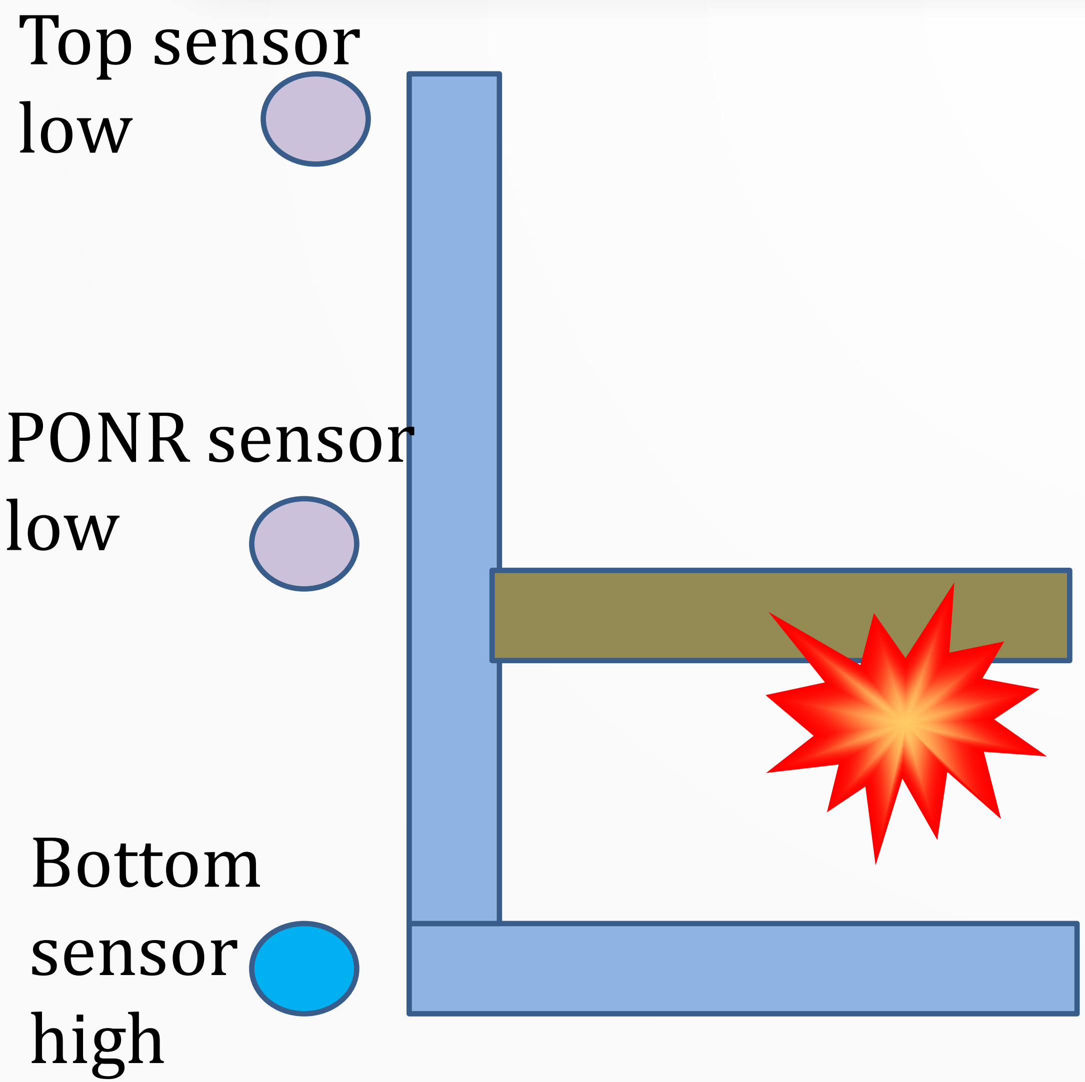




The user releases the button.

The controller thinks the plunger is still above the PONR and allows the abort. The motor turns on.

PONR Sensor stuck low



DANGER: The motor cannot turn on below the Point-of-no-return.

Safety Properties

Th2: Motor on below PONR: The motor should not turn on when the plunger is falling below the PONR.

G((plunger = fallingFast) => (electric_Motor = off));

Counterexample when the PONR sensor is stuck low: The plunger is falling fast, then the operator releases the button. The controller reads the PONR sensor value as low and thinks it is ok to turn on the motor. The controller turns on the motor.

What else can happen?

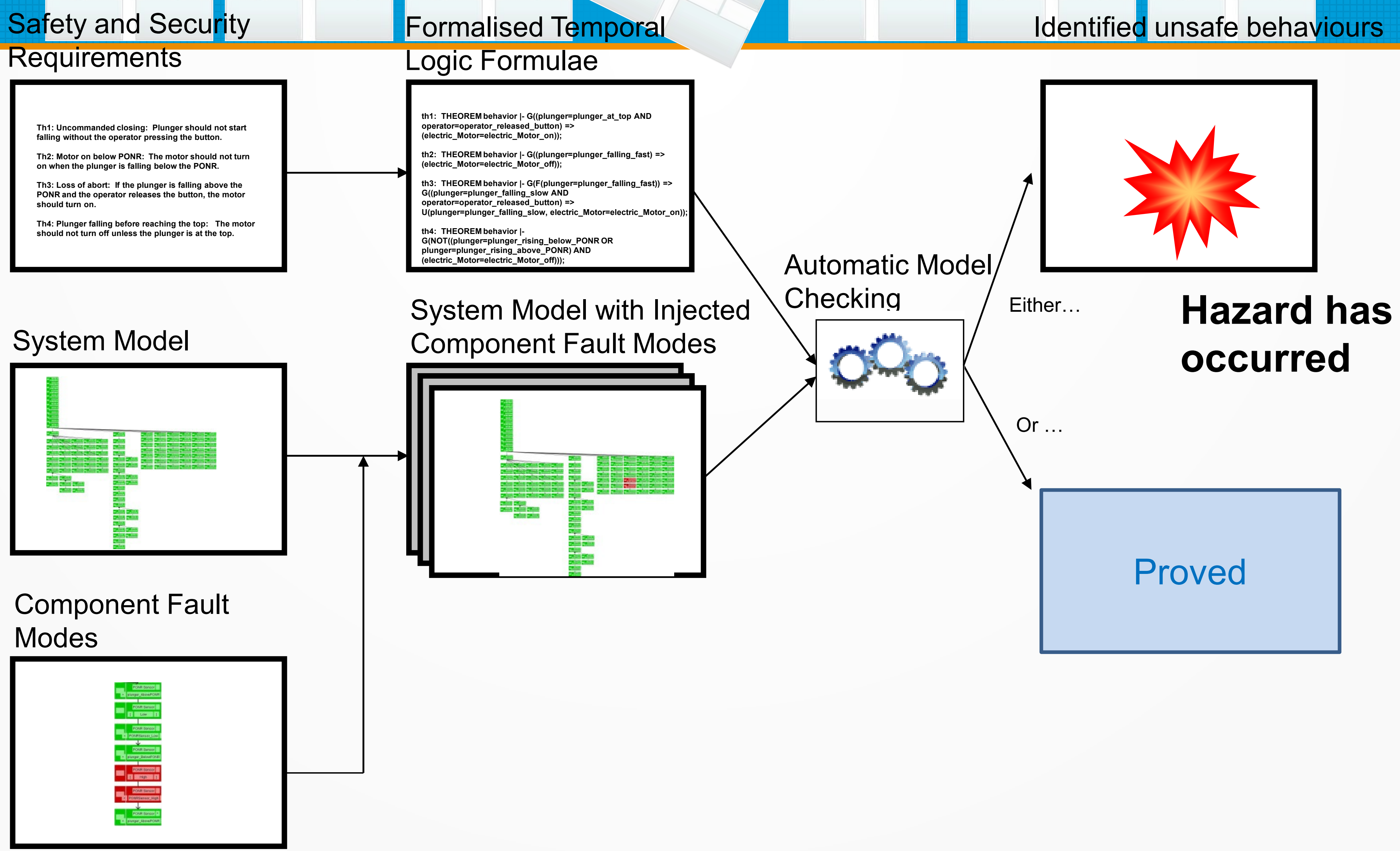
Table 1. Results of model-checking each component failure mode against the four hazard conditions

Component Failure	**HC1**	**HC2**	**HC3**	**HC4**
No failures	$\surd$	$\surd$	$\surd$	$\surd$
Top Sensor stuck Low	$\surd$	$\surd$	$\surd$	$\surd$
Top Sensor stuck High	$\surd$	$\surd$	$\surd$	X
Bottom Sensor stuck Low	$\surd$	$\surd$	$\surd$	$\surd$
Bottom Sensor stuck High	$\surd$	X	$\surd$	$\surd$
PONR Sensor stuck Low	$\surd$	X	$\surd$	$\surd$
PONR Sensor stuck High	$\surd$	$\surd$	X	$\surd$
Button stuck released	$\surd$	$\surd$	$\surd$	$\surd$
Button stuck pushed	X	$\surd$	X	$\surd$
Motor stuck on	$\surd$	$\surd$	$\surd$	$\surd$
Motor stuck off	$\surd$	$\surd$	$\surd$	$\surd$

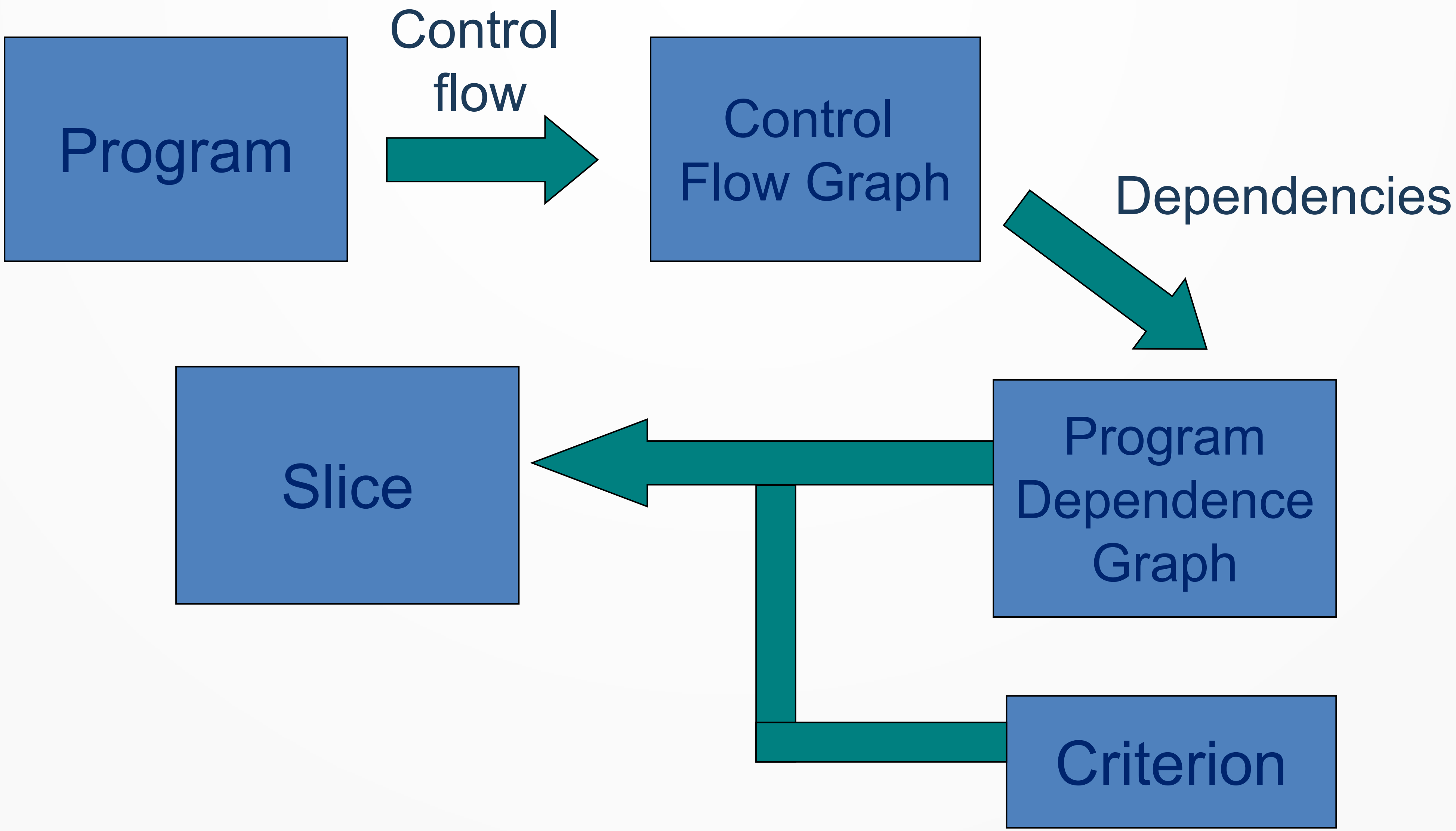
Key: $\surd$ = hazard condition does not arise, X = hazard condition can occur

From: Grunske, L., Lindsay, P., **Yatapanage, N**. and Winter, K. (2005). An Automated Failure Mode and Effect Analysis based on High-Level Design Specification with Behavior Trees. *Integrated Formal Methods: 5th International Conference (IFM 2005), Proc.,* Lecture Notes in Computer Science. Springer-Verlag. 3771:129-149.

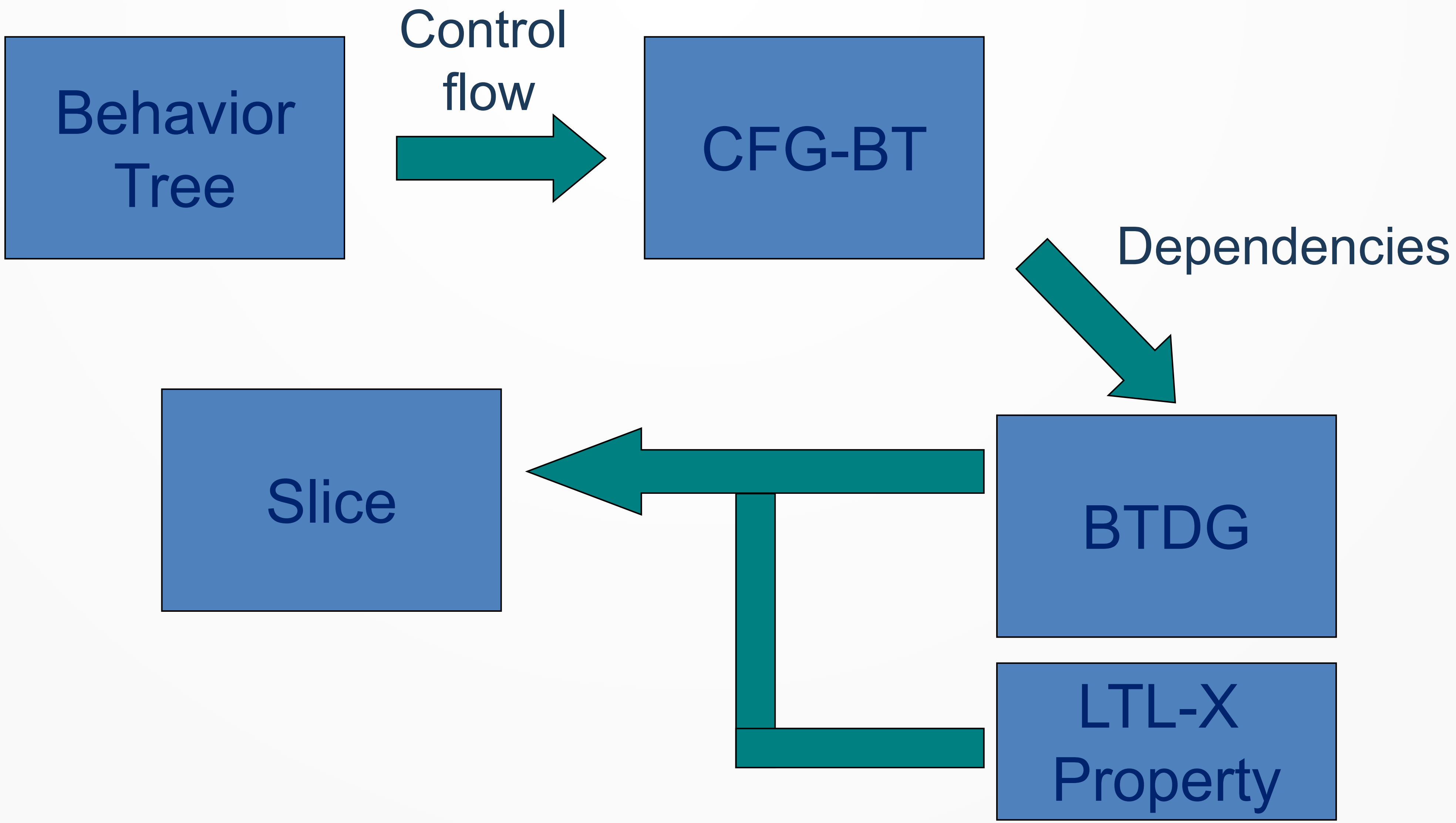
Behavior Tree Syntax



Program Slicing



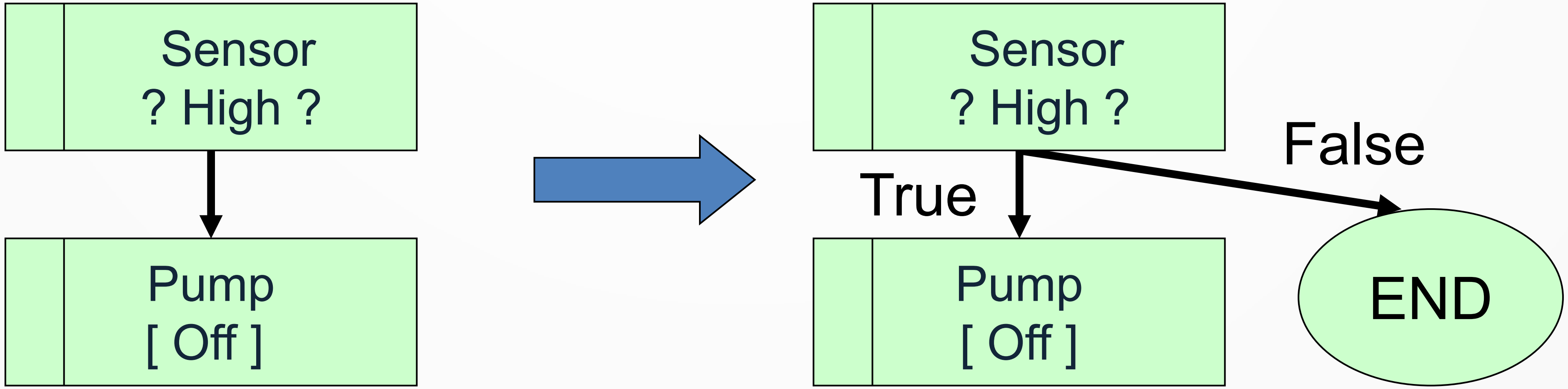
Slicing of BTs



BT-Control Flow Graph

A BT-CFG is similar to a BT except:

- Selections, guards and input events have an additional branch to show the false behaviour.



- Reversion / reference nodes are replaced by edges.

Control Dependence

Node *q* is control-dependent on node *p* iff:

- *p* has at least 2 successors *m* and *n*, where NOT(Alt(*m*,*n*)) and NOT(Conc(*m*,*n*)),
- for all maximal paths from *m*, *q* always occurs and
- there exists a maximal path from *n* on which *q* never occurs.

Control dependence occurs if *p* is a guard, selection or input event.

Program Slicing

- Automatically removes parts of the program which are irrelevant to a given criterion.
- Originally developed by Weiser (1981) for debugging programs.
- Start at a slicing criterion and then follow back dependencies, e.g. control & data dependencies.

Data Dependence

Node *q* is data/interference-dependent on node *p* iff:

- $\exists c \in \mathrm{DEF}(p)$ such that $c \in \mathrm{REF}(q)$ and
- $\forall k \in \mathit{Path}(p,q)$, $c \notin \mathrm{DEF}(k)$.

Data dependence = same thread
Interference dependence = parallel threads

Interference dependence is intransitive, so can lead to less precise, but still correct slices.

Other Dependencies

Node *q* is message-dependent on node *p* iff:

- *type*(*p*) = internalOutput and *behavior*(*q*) = *m*
- *type*(*p*) = internalInput and *behavior*(*q*) = *m.*

Node *q* is synchronisation-dependent on node *p* iff:

- *flag*(*p*) = *flag*(*q*) = synchronisation and
- *matching*(*p*,*q*).

Node *q* is *alternate-dependent* on node *p* iff:

- *p* and *q* have the same parent node and
- *p* and *q* are connected by an alternate branching point.

Creating the Slice

- Start at nodes which modify variables that are in the property.

$$\mathit{SliceCrit}(p) = \{n : BTNode \mid \exists c \in REF(p) \bullet c \in DEF(n)\}$$

- Traverse BTDG backwards, collecting all the nodes encountered.

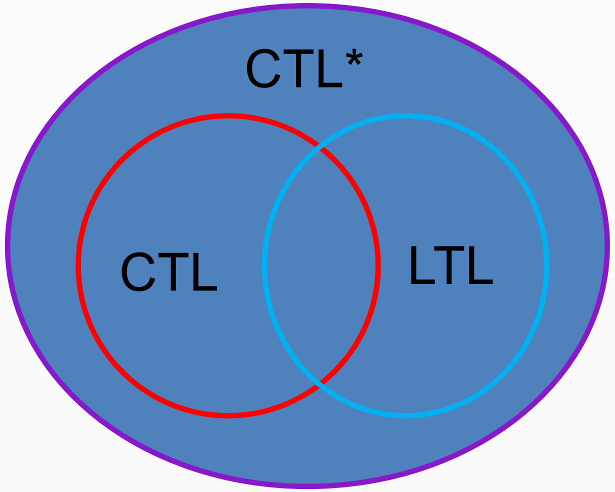
Creating the Slice

- Re-form into a Behavior Tree by adding blank place-holder nodes.
- Put reversions and reference nodes back into the slice, unless the entire sub-tree is not in the slice.
- Put for-all and for-some nodes back into the slice, unless the parameter is no longer used in the sub-tree below.

Bisimulation

Strong Bisimulation – matches every step
– preserves full CTL*

- Too restrictive for some applications,
e.g. slicing, where a model is reduced by eliminating stuttering



Weak Bisimulation

Weak forms of bisimulation

– do not match every step

– are suitable for applications like slicing

– do not preserve the X operator

- e.g. Branching bisimulation with explicit divergence

  - preserves CTL^*_{-X}

However, the next operator is useful in practice

- e.g. for safety properties such as *failure* => X(*set-alarm*)

Branching Bisimulation with Explicit Divergence

If a state *s* takes a step α to *s'*,

- then if $\alpha = \tau$, *bb*(*s'*, *t*) or

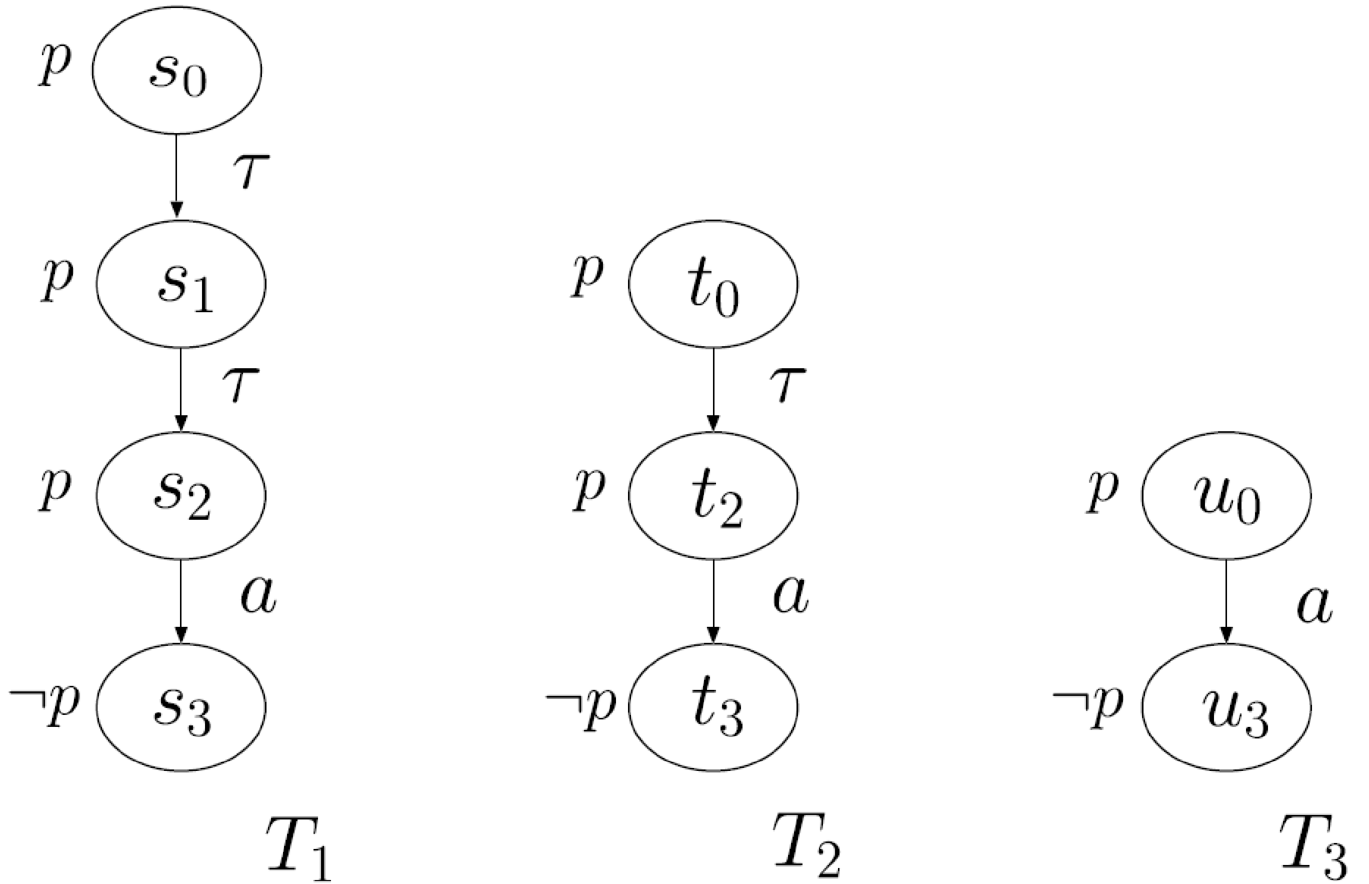
- there exist *t'*, *t''* such that *t* is followed by any number of stuttering steps to *t'*, which is then followed by the α step to *t''*, where *bb*(*s*, *t'*) and *bb*(*s'*, *t''*)

- if there exists an infinite stuttering path after *s* then there exists an infinite stuttering path after *t*.

The relation is symmetric.

It is defined so that branching logics can be preserved.
- It preserves CTL*-X.

Eliminating Stuttering



Example

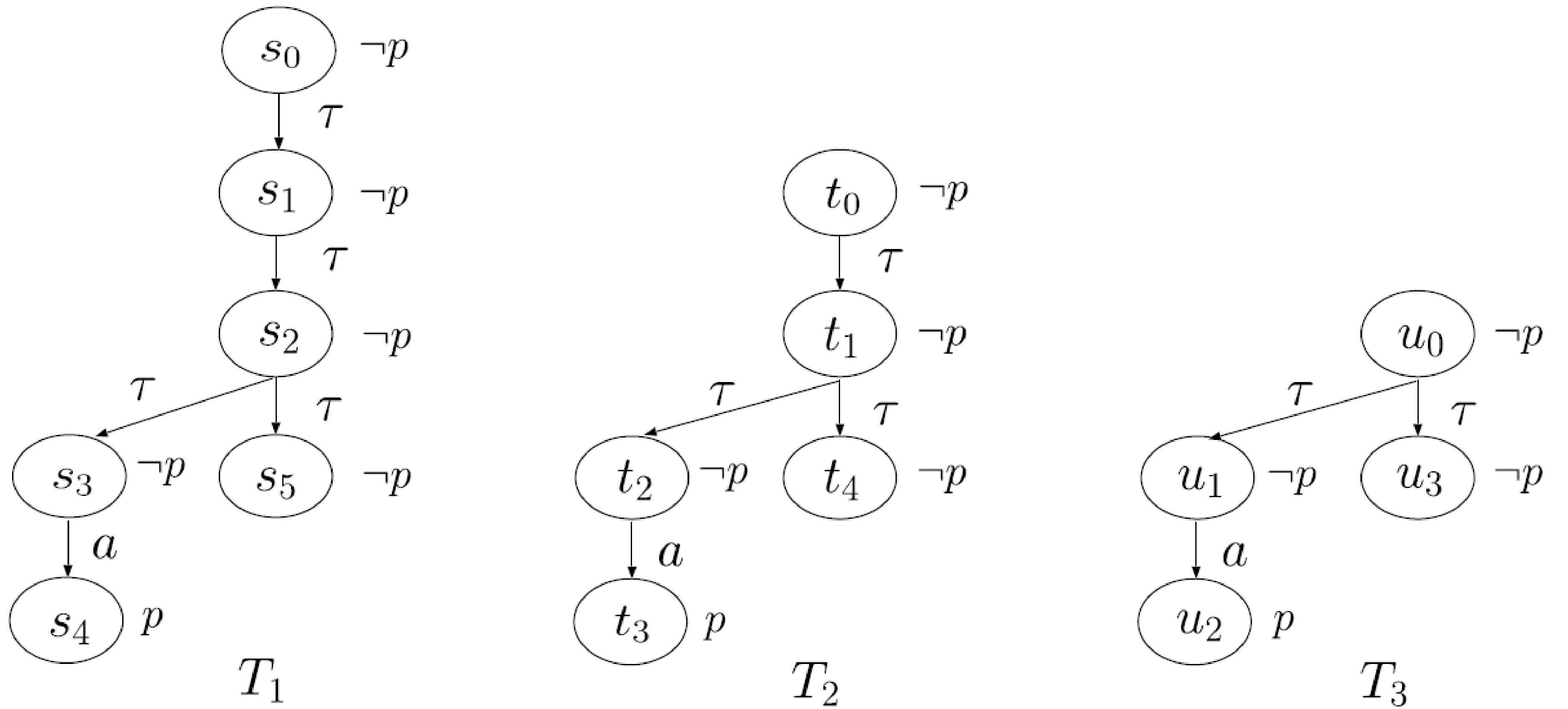
$T_1, s_0 \models \mathbf{AXp}$ $\quad$ $T_2, t_0 \models \mathbf{AXp}$ $\quad$ $T_3, u_0 \not\models \mathbf{AXp}$

Eliminating Stuttering

In the previous example, all three transition systems are related by branching bisimulation, but they don't satisfy the same properties with X.

- shows that the stuttering before an observable step is important.

Eliminating Stuttering



$T_1, s_0 \models \mathbf{AXE(F}p\mathbf{)}$ $\quad$ $T_2, t_0 \models \mathbf{AXE(F}p\mathbf{)}$ $\quad$ $T_3, u_0 \not\models \mathbf{AXE(F}p\mathbf{)}$

Observable Steps

These examples illustrate the notion of *observable step*.

An *observable step* is one which either:
- performs an observable (non-stuttering) action,
- passes a critical branching point, or
- performs a *relevant stuttering step* with respect to a particular formula.

Branching bisimulation covers only *observable actions* and *critical branching points* (called *bb-observable steps*).

Next-preserving branching bisimulation additionally considers *relevant stuttering steps.*

Relevant Stuttering Steps

The *relevant stuttering steps* are the ones which must be preserved, determined according to the *xdepth* of a formula:

$$xdepth(\varphi) = 0, \text{ where } \varphi \in AP,$$

$$xdepth(\psi_1 \wedge \psi_2) = max(xdepth(\psi_1), xdepth(\psi_2)),$$

$$xdepth(\neg\psi_1) = xdepth(\psi_1),$$

$$xdepth(\mathbf{E}\,\varphi) = xdepth(\varphi),$$

$$xdepth(\varphi_1\,\mathbf{U}\,\varphi_2) = max(xdepth(\varphi_1), xdepth(\varphi_2)),$$

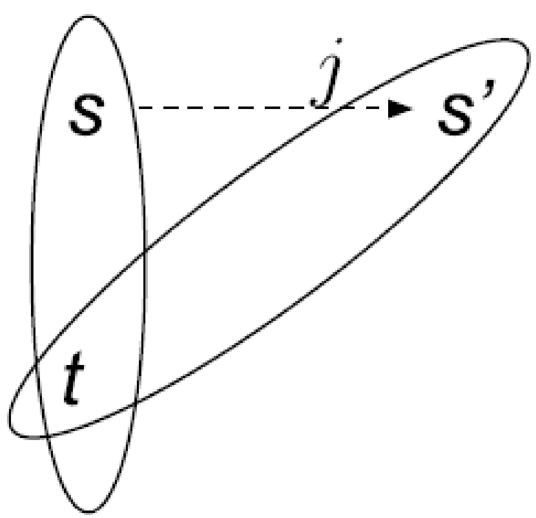
$$xdepth(\mathbf{X}\,\varphi) = xdepth(\varphi) + 1.$$

e.g. xdepth(Xp) = 1
xdepth((Xp) U (XXq)) = 2

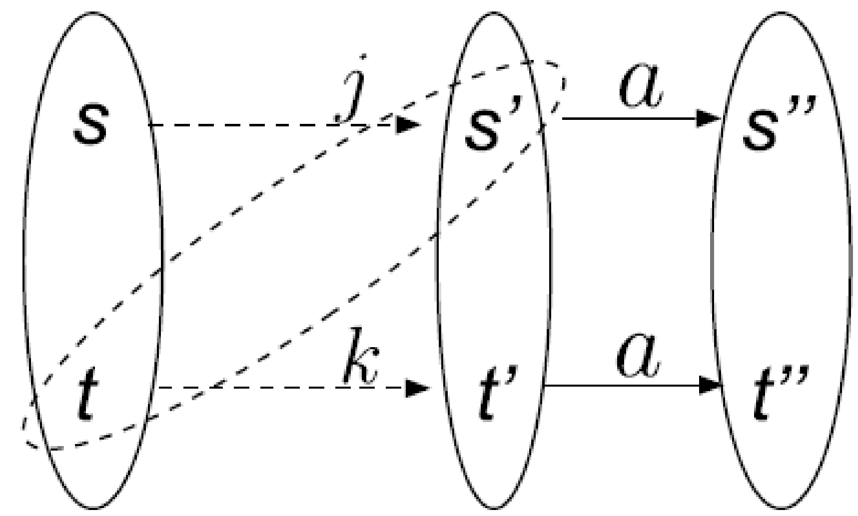
Relevant Stuttering Steps

If a transition system has more than xdepth(φ) stuttering steps before a bb-observable step, then the validity of φ does not change along the excess stuttering steps (the stuttering steps that are more than xdepth(φ) steps away from the bb-observable step).

Next-preserving Branching Bisimulation



The solid ellipses represent next-preserving branching bisimulation.



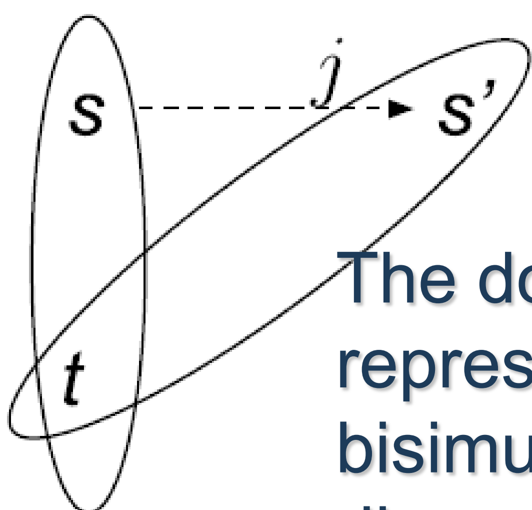
If a τ step occurs from *s* to *s'*, which is not bb-observable, and *s'* is npbb with depth *xd* to state *t*, then *t* is not required to do a matching step.

Next-preserving Branching Bisimulation

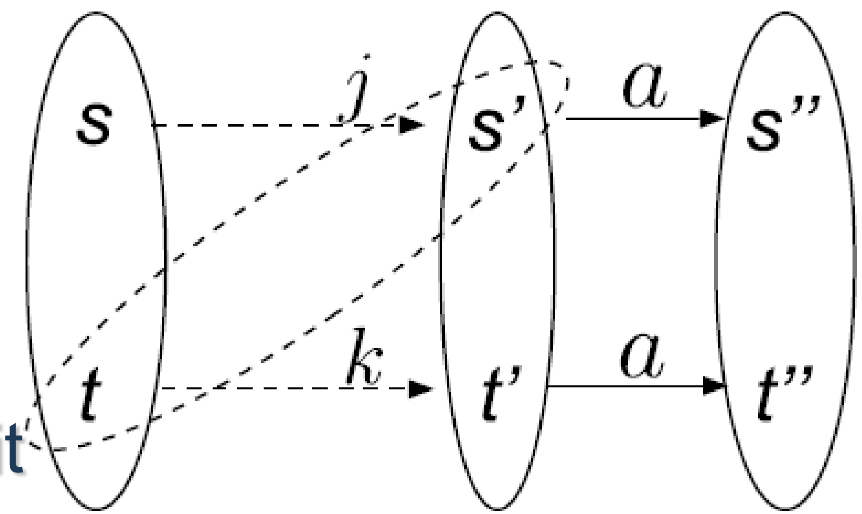
If a bb-observable step α occurs after *s*, possibly preceded by non-bb-observable steps, then this must be matched by *t*, also possibly preceded by non-bb-observable steps.
- The non-bb-observable steps which are *xd-relevant* must be preserved.

If a bb-observable step α occurs after *s*, possibly preceded by non-bb-observable steps, then this must be matched by *t*, also possibly preceded by non-bb-observable steps.
- The non-bb-observable steps which are *xd-relevant* must be preserved.

Next-preserving Branching Bisimulation



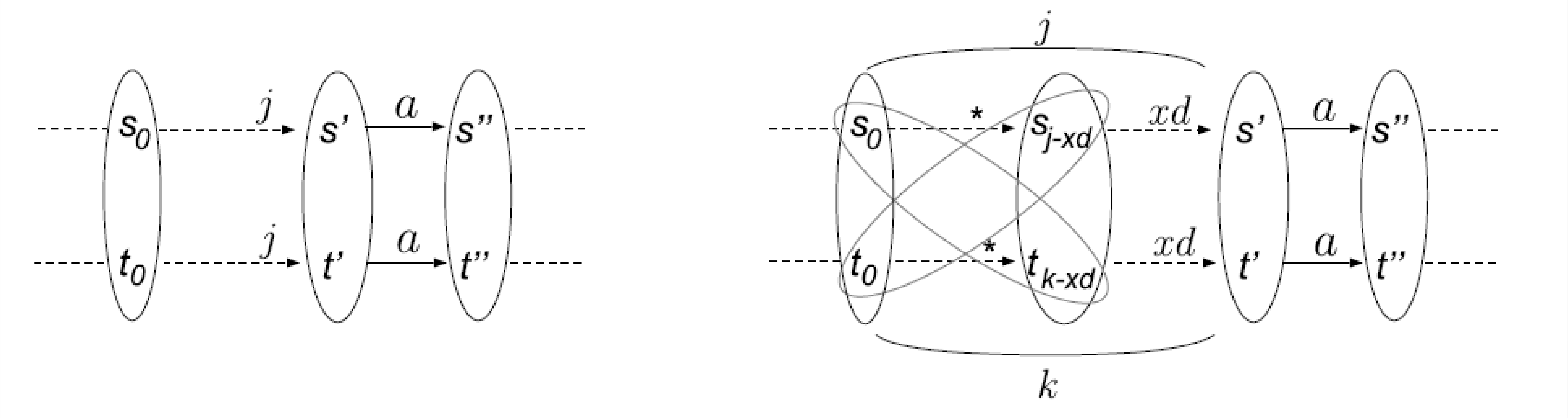
The dotted ellipse represents branching bisimulation with explicit divergence.



- if j < *xd* then k = j
- if j ≥ *xd* then k ≥ *xd*

Note that the relation is symmetric.

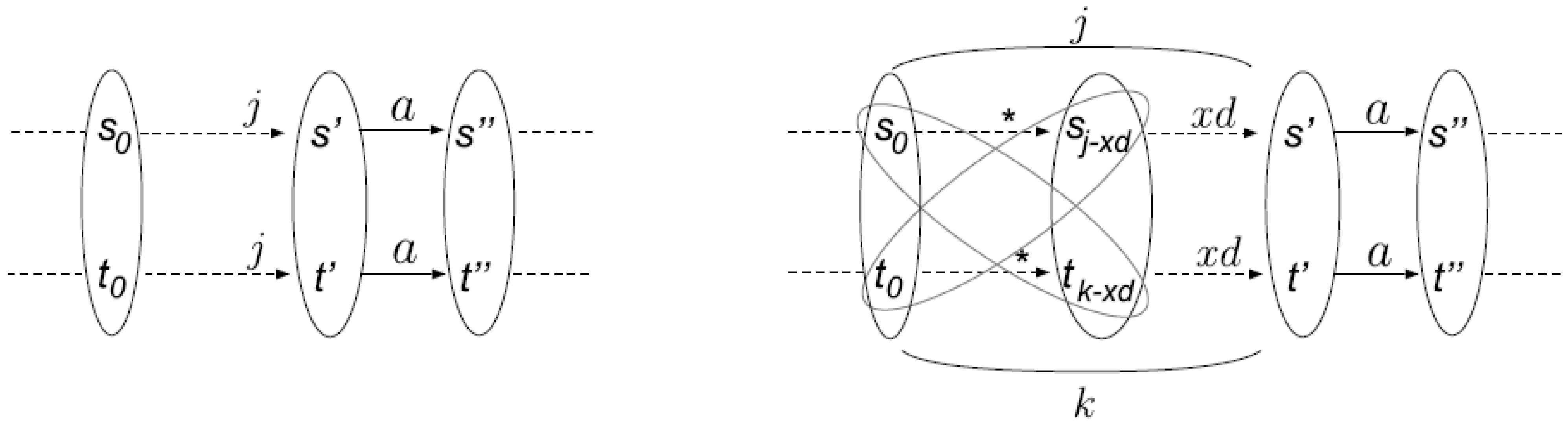
Next-preserving Branching Bisimulation (Paths)



If j < *xd*, then all the stuttering steps are *xd-relevant*, so must be matched.

Otherwise, only *xd* stuttering steps must be matched.

Next-preserving Branching Bisimulation (Paths)



This gives rise to an alternative definition using *xd-equivalent partitions*.

Each *xd-equivalent partition* of a path has to be matched by an *xd-equivalent partition* in the other path.

Next-preserving Branching Bisimulation

The parameterised next-preserving branching bisimulation gives rise to a heirarchy.

When $xd = 0$, the equivalence coincides with branching bisimulation with explicit divergence. This is the weakest next-preserving branching bisimulation.

When $xd = \infty$, the equivalence coincides with strong bisimulation. This is the strongest next-preserving branching bisimulation.

Next-preserving Branching Bisimulation

strong bisimulation

$$\vdots$$

$$\cap\!|$$

$$npb_4$$

$$\cap\!|$$

$$npb_3$$

$$\cap\!|$$

$$npb_2$$

$$\cap\!|$$

$$npb_1$$

$$\cap\!|$$

$$bb$$

$$\boxed{\begin{array}{c} npb_{xd} \\ \cap\!| \\ \vdots \\ npb_{\varphi_3} \\ \cap\!| \\ npb_{\varphi_2} \\ \cap\!| \\ npb_{\varphi_1} \end{array}}$$

Next-preserving Slicing

The next-preserving branching bisimulation definition was used to create a slicing method that preserves CTL*.

- requires extra stuttering nodes to be kept in the slice.

- Extra nodes are placed before critical nodes and branching points in the transition system – equivalent to several constructs in the BT diagram, i.e. alternative branching, concurrent branching, conditional nodes.

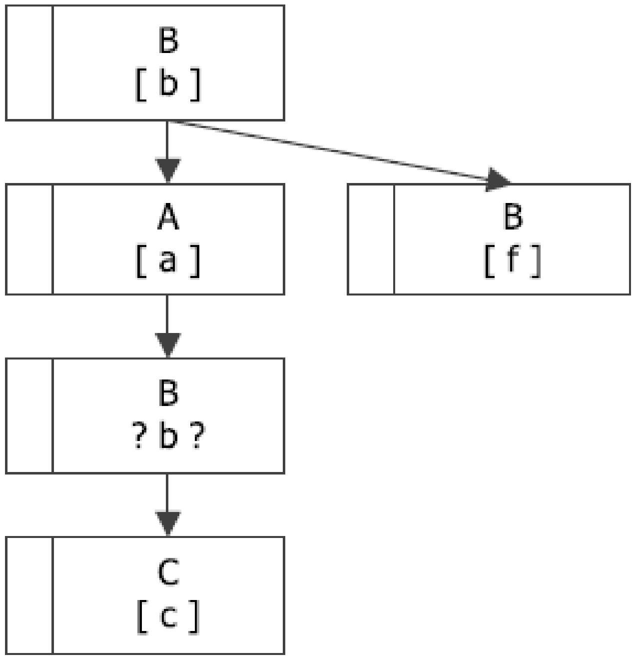
Next-preserving Slicing

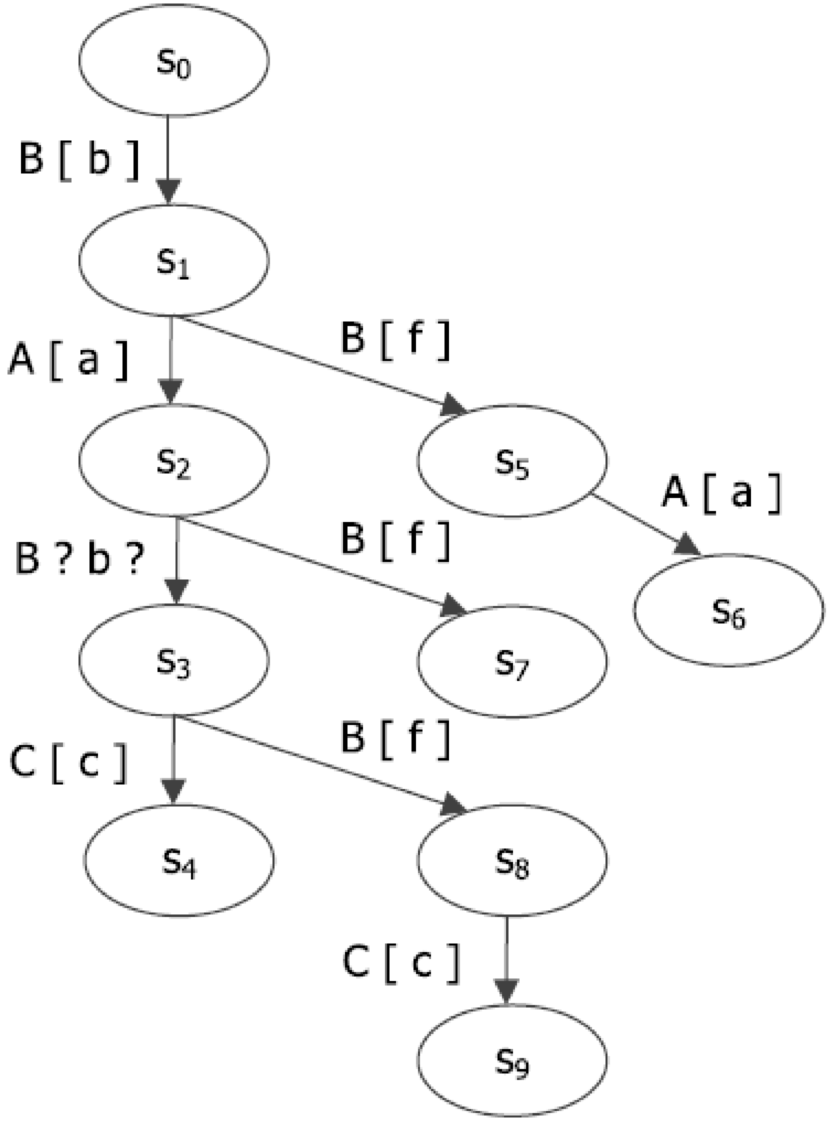
Alternative branching – need to look for cases where one branch leads to an observable node where the other doesn't.

Concurrent branching – non-determinism arises from the interleaved execution of nodes in different threads

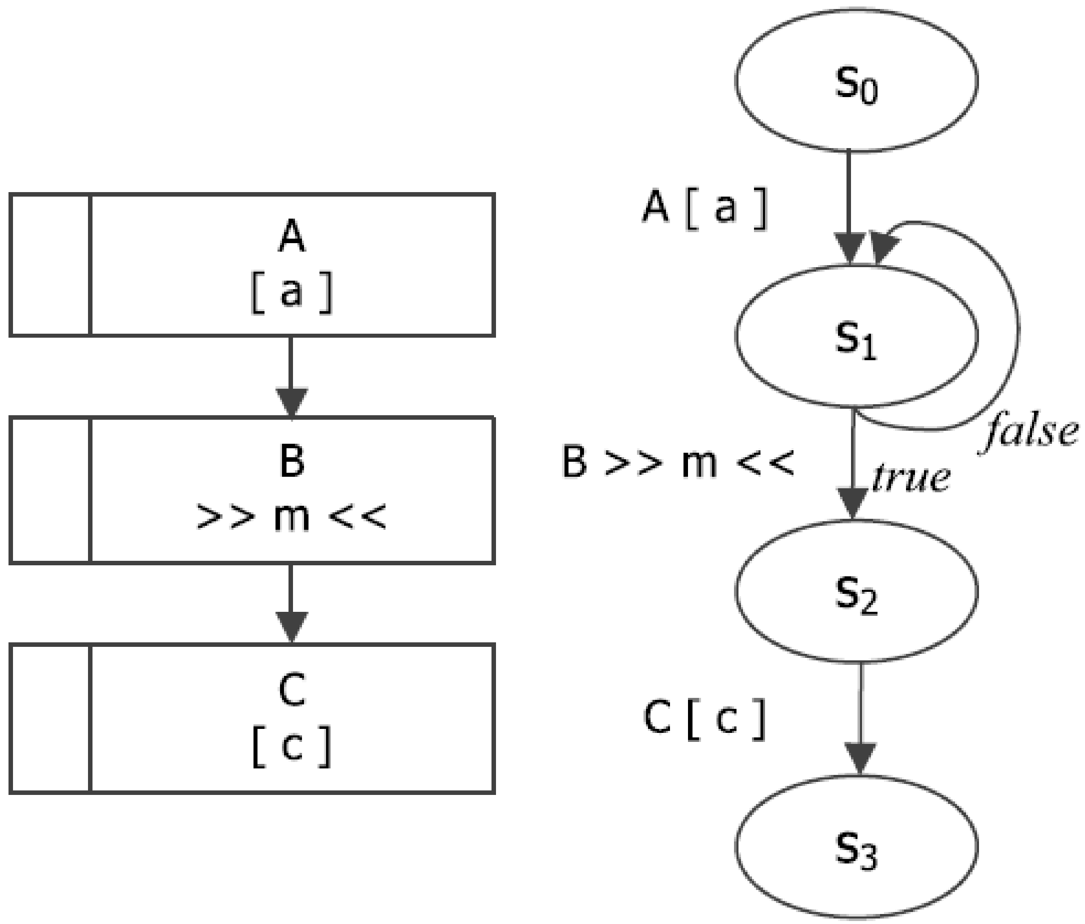
- Therefore any node in a concurrent branch is a branching point
- but most cases are not critical since they do not prevent the other nodes from executing

Next-preserving Slicing





Next-preserving Slicing



Conditional nodes – not really branching in the transition system, except if it's external input nodes.

Next-preserving Branching Bisimulation

- Proof of correctness that it preserves CTL* including X.

- Useful for applications where the X operator is required, e.g. slicing.