# Concurrent Programming: Critical Sections

CS 6410

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

[Robbert van Renesse]

# Concurrent Programming is Hard

Why?

- Concurrent programs are *non-deterministic*
  - run them twice with same input, get two different answers
  - or worse, one time it works and the second time it fails
- Program statements are executed non-atomically
  - **x += 1** compiles to something like
    - **LOAD x**
    - **ADD 1**
    - **STORE x**

# Enter *Harmony*

- A new concurrent programming language
  - heavily based on Python syntax to reduce learning curve for many
- A new underlying virtual machine

it tries *all* possible executions of a program (or rather, explores all possible reachable states) until it finds a problem, if any

(this is called "model checking")

# Non-Determinism

```
shared = True

def f(): assert shared
def g(): shared = False

f()
g()
```

(a) [code/prog1.hny] Sequential

```
shared = True

def f(): assert shared
def g(): shared = False

spawn f()
spawn g()
```

(b) [code/prog2.hny] Concurrent

Figure 3.1: A sequential and a concurrent program.

# Non-Determinism

```
shared = True

def f(): assert shared
def g(): shared = False

f()
g()
```

(a) [code/prog1.hny] Sequential

```
shared = True

def f(): assert shared
def g(): shared = False

spawn f()
spawn g()
```

(b) [code/prog2.hny] Concurrent

Figure 3.1: A sequential and a concurrent program.

#states 2
2 components, 0 bad states
No issues

#states 11
Safety Violation
T0: __init__() [0-3,17-25] { shared: True }
T2: g() [13-16] { shared: False }
T1: f() [4-8] { shared: False }
Harmony assertion failed

# Critical Section

Must be serialized due to shared memory access

T1

```
. . .
CSEnter()
  Critical section
CSExit()
. . .
```

T2

```
. . .
CSEnter()
  Critical section
CSExit()
. . .
```

<u>Goals</u>

**Mutual Exclusion:** 1 thread in a critical section at time

**Progress:** at least one thread makes it into the CS if desired and no other thread is there

**Fairness:** equal chances of getting into CS

… in practice, fairness rarely guaranteed or needed

# Mutual Exclusion and Progress

- Need both:
  - either one is trivial to achieve by itself

# Critical Sections in Harmony

```
def thread(self):
    while True:
        …    # code outside critical section
        …    # code to enter the critical section
        …    # critical section itself
        …    # code to exit the critical section

spawn thread(1)
spawn thread(2)
…
```

- How do we check mutual exclusion?
- How do we check progress?

# Critical Sections in Harmony

```
def thread(self):
    while True:
        …    # code outside critical section
        …    # code to enter the critical section
        cs: assert countLabel(cs) == 1
        …    # code to exit the critical section

spawn thread(1)
spawn thread(2)
…
```

- How do we check mutual exclusion?
- How do we check progress?

# Critical Sections in Harmony

```
def thread(self):
    while choose( { False, True } ):
        …    # code outside critical section
        …    # code to enter the critical section
        cs: assert countLabel(cs) == 1
        …    # code to exit the critical section

spawn thread(1)
spawn thread(2)
…
```

- How do we check mutual exclusion?
- How do we check progress?
  - *if code to enter/exit the critical section cannot terminate, Harmony with balk*

# *Sounds like you need a lock…*

- True, but this is an O.S. class!
- The question is:

*How does one build a lock?*

- Harmony is a concurrent programming language. *Really, doesn't Harmony have locks?*

*You have to program them!*

# First attempt: a naïve lock

```
lockTaken = False

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        await not lockTaken
        lockTaken = True

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        lockTaken = False

spawn thread(0)
spawn thread(1)
```

Figure 5.3: [code/naiveLock.hny] Naïve implementation of a shared lock.

# First attempt: a naïve lock

```
lockTaken = False

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        await not lockTaken
        lockTaken = True

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        lockTaken = False

spawn thread(0)
spawn thread(1)
```

wait till lock is free, then take it

Figure 5.3: [code/naiveLock.hny] Naïve implementation of a shared lock.

# First attempt: a naïve lock

```
lockTaken = False

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        await not lockTaken
        lockTaken = True

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        lockTaken = False

spawn thread(0)
spawn thread(1)
```

wait till lock is free, then take it

release the lock

Figure 5.3: [code/naiveLock.hny] Naïve implementation of a shared lock.

# First attempt: a naïve lock

```
1   lockTaken = False
2
3   def thread(self):
4       while choose({ False, True }):
5           # Enter critical section
6           await not lockTaken
7           lockTaken = True
8
9           # Critical section
10          @cs: assert atLabel(cs) == { (thread, self): 1 }
11
12          # Leave
13          lockTake
14
15  spawn thread(
16  spawn thread(
```

```
==== Safety violation ====
__init__/() [0,26-36]                         36 { lockTaken: False }
thread/0 [1-2,3(choose True),4-7]              8 { lockTaken: False }
thread/1 [1-2,3(choose True),4-8]              9 { lockTaken: True }
thread/0 [8-19]                               19 { lockTaken: True }
>>> Harmony Assertion (file=code/naiveLock.hny, line=10) failed
```

Figure 5.3: [code/naiveLock.hny] Naïve implementation of a shared lock.

# Second attempt: *flags*

```
flags = [ False, False ]

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        await not flags[1 − self]

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

Figure 5.5: [code/naiveFlags.hny] Naïve use of flags to solve mutual exclusion.

# Second attempt: *flags*

```
flags = [ False, False ]

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        await not flags[1 – self]

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

enter, then wait for other

Figure 5.5: [code/naiveFlags.hny] Naïve use of flags to solve mutual exclusion.

# Second attempt: *flags*

```
flags = [ False, False ]

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        await not flags[1 - self]

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

enter, then wait for other

leave

Figure 5.5: [code/naiveFlags.hny] Naïve use of flags to solve mutual exclusion.

# Second attempt: *flags*

```
flags = [ False, False ]

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        await not flags[1 − self]

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

Figure 5.5: [code/naiveFlags.hny] Naïve use of flags to solve mutual exclusion.

# Second attempt: *flags*

```
flags = [ False, False ]

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        await not flags[1 – self]

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

Figure 5.5: [code/n

```
==== Non-terminating State ===
__init__/() [0,36-46]                      46 { flags: [False, False] }
thread/0 [1-2,3(choose True),4-12] 13 { flags: [True,  False] }
thread/1 [1-2,3(choose True),4-12] 13 { flags: [True,  True] }
blocked thread: thread/1 pc = 13
blocked thread: thread/0 pc = 13
```

# Third attempt: *turn* variable

```
turn = 0

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        turn = 1 - self
        await turn == self

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section

spawn thread(0)
spawn thread(1)
```

Figure 5.7: [code/naiveTurn.hny] Naïve use of turn variable to solve mutual exclusion.

# Third attempt: *turn* variable

```
turn = 0

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        turn = 1 − self
        await turn == self

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section

spawn thread(0)
spawn thread(1)
```

after you...

Figure 5.7: [code/naiveTurn.hny] Naïve use of turn variable to solve mutual exclusion.

# Third attempt: *turn* variable

```
turn = 0

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        turn = 1 − self
        await turn == self

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section

spawn thread(0)
spawn thread(1)
```

after you...

wait for your turn

Figure 5.7: [code/naiveTurn.hny] Naïve use of turn variable to solve mutual exclusion.

# Third attempt: *turn* variable

```
turn = 0

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        turn = 1 – self
        await turn == self

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section

spawn thread(0)
spawn thread(1)
```

```
==== Non-terminating State ===
__init__/() [0,28-38]                                          38 { turn: 0 }
thread/0 [1-2,3(choose True),4-26,2,3(choose True),4]   5 { turn: 1 }
thread/1 [1-2,3(choose False),4,27]                            27 { turn: 1 }
blocked thread: thread/0 pc = 5
```

Figure 5.7: [co

# Peterson's Algorithm: *flags & turn*

```
sequential flags, turn

flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 – self
        await (not flags[1 – self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

# Peterson's Algorithm: *flags & turn*

latest version of Harmony only

```
sequential flags, turn

flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 − self
        await (not flags[1 − self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

# Peterson's Algorithm: *flags & turn*

```
sequential flags, turn

flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 − self
        await (not flags[1 − self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

"you go first"

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

# Peterson's Algorithm: *flags & turn*

```
sequential flags, turn

flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 − self
        await (not flags[1 − self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

"you go first"

wait until alone or it's my turn

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

# Peterson's Algorithm: *flags & turn*

```
sequential flags, turn

flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 − self
        await (not flags[1 − self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

"you go first"

wait until alone or it's my turn

leave

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

# Peterson's Algorithm: *flags & turn*

```
sequential flags, turn

flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 − self
        await (not flags[1 − self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

#states = 104 diameter = 5
#components: 37
no issues found

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

So, we proved Peterson's Algorithm correct by brute force, enumerating all possible executions. We now know *that* it works.

*But how does one prove it by deduction?*

*so one understands why it works…*

# What and how?

- Need to show that, for any execution, all states reached satisfy mutual exclusion
    - in other words, mutual exclusion is *invariant*

*invariant = predicate that holds in every reachable state*

# How to prove an invariant?

- Need to show that, for any execution, all states reached satisfy the invariant
- Sounds similar to sorting:
  - Need to show that, for any list of numbers, the resulting list is ordered
- Let’s try *proof by induction* on the length of an execution

# Proof by induction

You want to prove that some *Induction Hypothesis* IH(n) holds for any n:

- Base Case:
  - show that IH(0) holds
- Induction Step:
  - show that if IH(i) holds, then so does IH(i+1)

# Proof by induction in our case

To show that some IH holds for an *execution* E of any number of *steps*:

- Base Case:
  - show that IH holds in the initial state(s)
- Induction Step:
  - show that if IH holds in a state produced by E, then for any possible next step s, IH also holds in the state produced by E + [s]

# But there's a problem

- How do we characterize a "state produced by E"?
  - or how do we characterize a *reachable state*?
- Instead, it's much easier if we proved a so-called "inductive invariant":
  - Base Case:
    - show that IH holds in the initial state(s)
  - Induction Step:
    - show that if IH holds in ***any*** state, then for any possible next step, IH also holds in the resulting state

# First question: what should IH be?

- Obvious answer: mutual exclusion itself
  - if $T0$ is in the critical section, then $T1$ is not
    - without loss of generality…
  - Formally: $T0@cs \Longrightarrow \neg T1@cs$

- Unfortunately, this won't work…

# State before T1 takes a step:

```
sequential flags, turn

flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 – self
        await (not flags[1 – self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

T1

T0

flags = [ True, True ]
turn = 1

mutual exclusion holds

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

# State after T1 takes a step:

```
sequential flags, turn

flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 – self
        await (not flags[1 – self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

flags = [ True, True ]
turn = 1

T0

T1

mutual exclusion violated

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

# So, is Peterson's Algorithm broken?

# *No, it'll turn out this prior state cannot be reached from the initial state (see later)*

```
sequential flags, turn

flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 − self
T1      await (not flags[1 − self]) or (turn == self)

        # critical section is here
T0      @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

flags = [ True, True ]
turn = 1

mutual exclusion holds

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

# Useful and obvious but insufficient invariant

```
sequential flags, turn

flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 − self
        await (not flags[1 − self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

$$Tx@cs \Rightarrow flags[x]$$

mutual exclusion holds

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

# What else do we expect to hold @cs?

```
sequential flags, turn

flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 – self
        await (not flags[1 – self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

? ? ?

mutual exclusion holds

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

# Another obvious IH to try

- Based on the **await** condition:

$$T0@cs \Longrightarrow \neg flags[1] \vee turn = 0$$

- Promising because if $T0@cs \wedge T1@cs$ then

$$\left.\begin{array}{r} T0@cs \Longrightarrow \neg flags[1] \vee turn = 0 \wedge \\ T1@cs \Longrightarrow \neg flags[0] \vee turn = 1 \end{array}\right\} \Rightarrow \left\{\begin{array}{l} turn = 0 \;\wedge \\ \;turn = 1 \end{array}\right.$$

$\Longrightarrow$ False (therefore mutual exclusion)

- Unfortunately, this is not an invariant…

# Another obvious IH to try

- Based on the **await** condition:

$$T0@cs \Longrightarrow \neg flags[1] \lor turn = 0$$

Easy to check with Harmony
Just run it with the following:
@cs: assert (not flags[1 - self]) or (turn == self)

- Unfortunately, this is not an invariant…

# State before T1 takes a step:

flags = [ True, False ]
turn = 1

```
sequential flags, turn

flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
T1      # Enter critical section
        flags[self] = True
        turn = 1 − self
        await (not flags[1 − self]) or (turn == self)

        # critical section is here
T0      @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical s
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

$T0@cs \Longrightarrow \neg flags[1] \vee turn = 0$ holds

note: this is a reachable state

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

# State after T1 takes a step:

```
sequential flags, turn

flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 − self
        await (not flags[1 − self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical s
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

T1

T0

flags = [ True, True ]
turn = 1

$T0@cs \Longrightarrow \neg flags[1] \vee turn = 0$ violated

note: this is also a reachable state

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

# But suggests an improved hypothesis

$$T0@cs \Longrightarrow \neg flags[1] \lor turn = 0 \lor T1@gate$$

```
flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
T1 →    @gate: turn = 1 − self
        await (not flags[1 − self]) or (turn == self)

        # Critical section
T0 →    @cs: assert (not flags[1 − self]) or (turn == self) or
                 (atLabel(gate) == {(thread, 1 − self): 1})

        # Leave critical section
        flags[self] = False
```

# But suggests an improved hypothesis

$$T0@cs \Longrightarrow \neg flags[1] \vee turn = 0 \vee T1@gate$$

```
flags = [ False, False ]
```

Also easy to check with Harmony

Proves that it is invariant, but not necessarily an *inductive invariant*

T0

```
        @cs: assert (not flags[1 − self]) or (turn == self) or
                        (atLabel(gate) == {(thread, 1 − self): 1})

        # Leave critical section
        flags[self] = False
```

# Inductive Invariance Proof

Let $I$ be the induction hypothesis:

$$I \triangleq T0@cs \Longrightarrow \neg flags[1] \vee turn == 0 \vee T1@gate$$

$I$ clearly holds in the initial state because $\neg T0@cs$ (false implies anything)

We are going to show: if $I$ holds in a state (*reachable or not*), then $I$ also holds in any state after either $T0$ or $T1$ takes a step

# Tricky Case 1:

$\neg T0@cs$ and $T0$ takes a step so that $T0@cs$

This must mean that $\neg flags[1] \vee turn = 0$ before the step (see code line 11)

```
11        await (not flags[1 − self]) or (turn == self)
12
13        # critical section is here
14        @cs: assert atLabel(cs) == { (thread, self): 1 }
```

But then $\neg flags[1] \vee turn = 0$ still holds after the step

So $T0@cs \Longrightarrow \neg flags[1] \vee turn = 0 \vee T1@gate$

# Tricky Case 2:

$T0@cs$ and $T1$ takes a step

This must mean that before the step

$\neg flags[1] \lor turn = 0 \lor T1@gate$ (by IH).

So, 3 cases to consider:

- $\neg flags[1] \Rightarrow flags[1]$
  - ➔ this means $T1@gate$ after the step
- $turn = 0 \Rightarrow turn = 1$
  - ➔ can't happen (only $T0$ sets $turn$ to 1)
- $T1@gate \Rightarrow \neg T1@gate$
  - ➔ this means $turn = 0$ after step

# *Enter critical section*
$flags[self] = \texttt{True}$
@gate: $turn = 1 - self$

So, $T0@cs \Longrightarrow \neg flags[1] \lor turn = 0 \lor T1@gate$

# Finally, prove mutual exclusion

$$T0@cs \land T1@cs \Longrightarrow$$

$$\begin{cases} \neg flags[1] \lor turn = 0 \lor T1@gate \\ \neg flags[0] \lor turn = 1 \lor T0@gate \end{cases} \land$$

$$\Longrightarrow turn = 0 \land \mathrm{turn} = 1$$

$$\Longrightarrow False$$

# Finally, prove mutual exclusion

$$T0@cs \wedge T1@cs \Longrightarrow$$

$$\begin{cases} \neg flags[1] \vee turn = 0 \vee T1@gate \\ \neg flags[0] \vee turn = 1 \vee T0@gate \end{cases} \wedge$$

$$\Longrightarrow turn = 0 \wedge \mathrm{turn} = 1$$

$$\Longrightarrow False$$

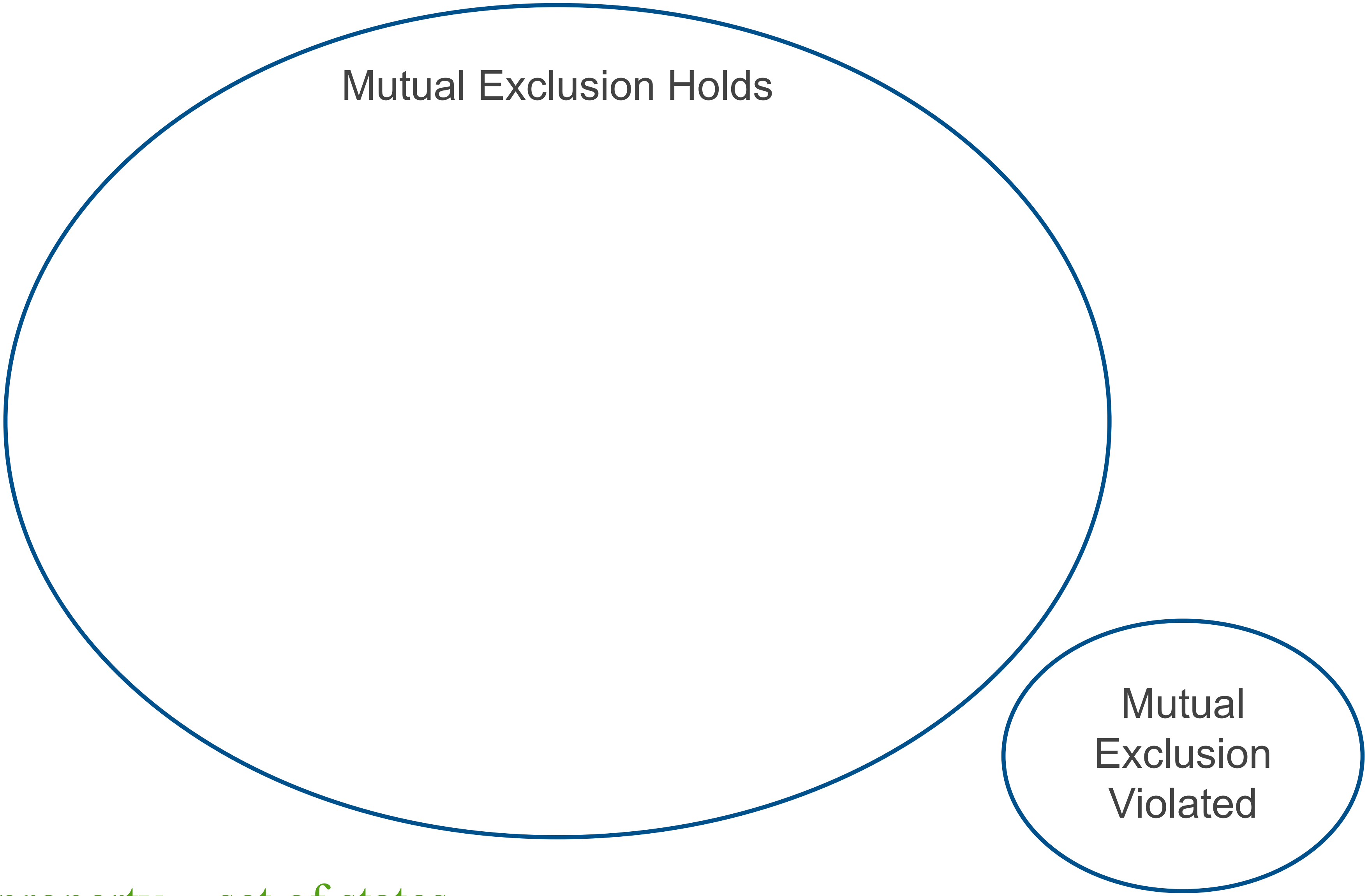
# Now we can see why this state cannot be reached!

```
sequential flags, turn

flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 − self
T1      await (not flags[1 − self]) or (turn == self)

        # critical section is here
T0      @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

flags = [ True, True ]
turn = 1

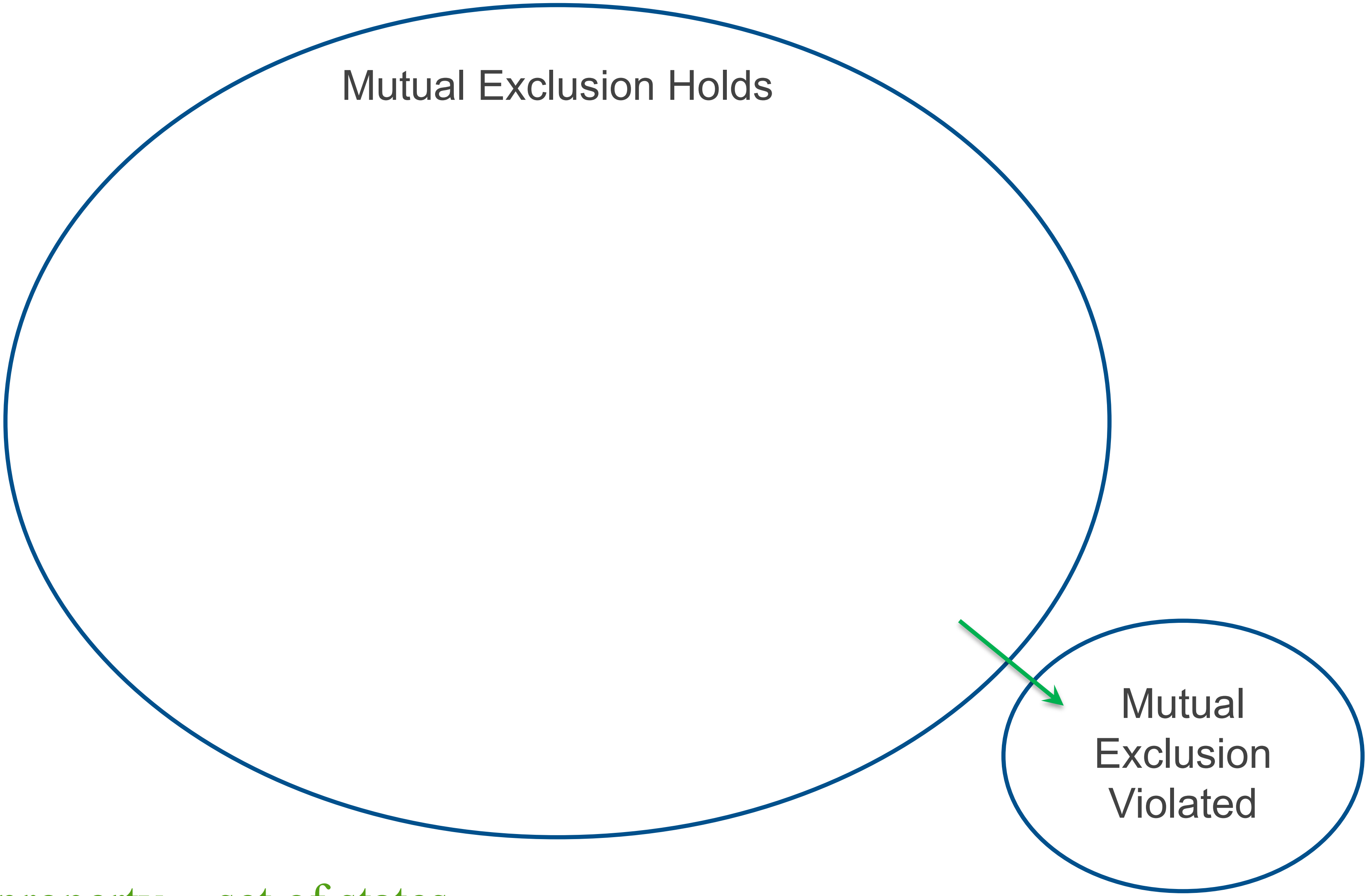
$$T0@cs \Longrightarrow \neg flags[1] \vee turn = 0 \vee T1@gate \quad \times$$

# Review in Pictures: State Space



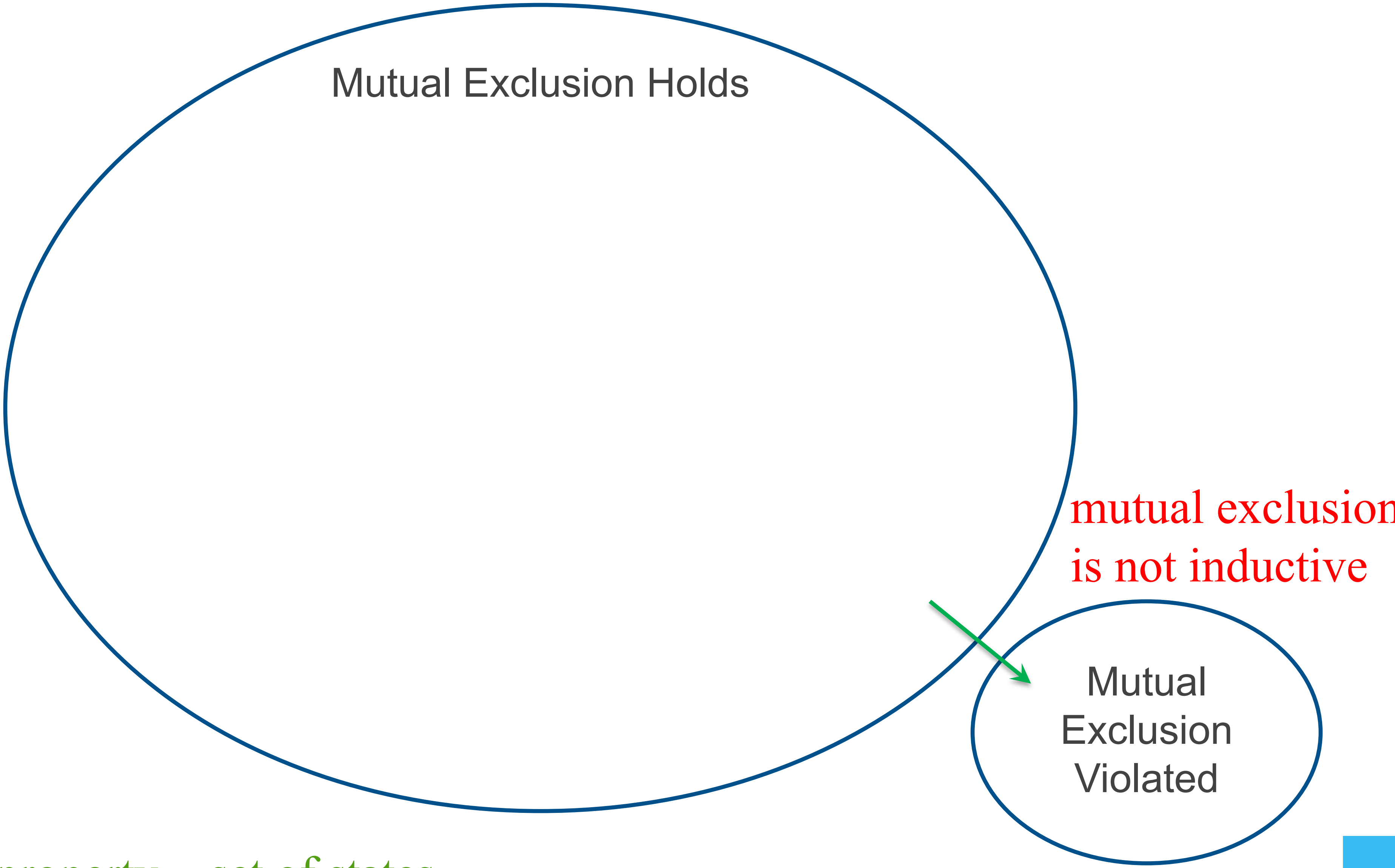

property = set of states

# Review in Pictures: State Space



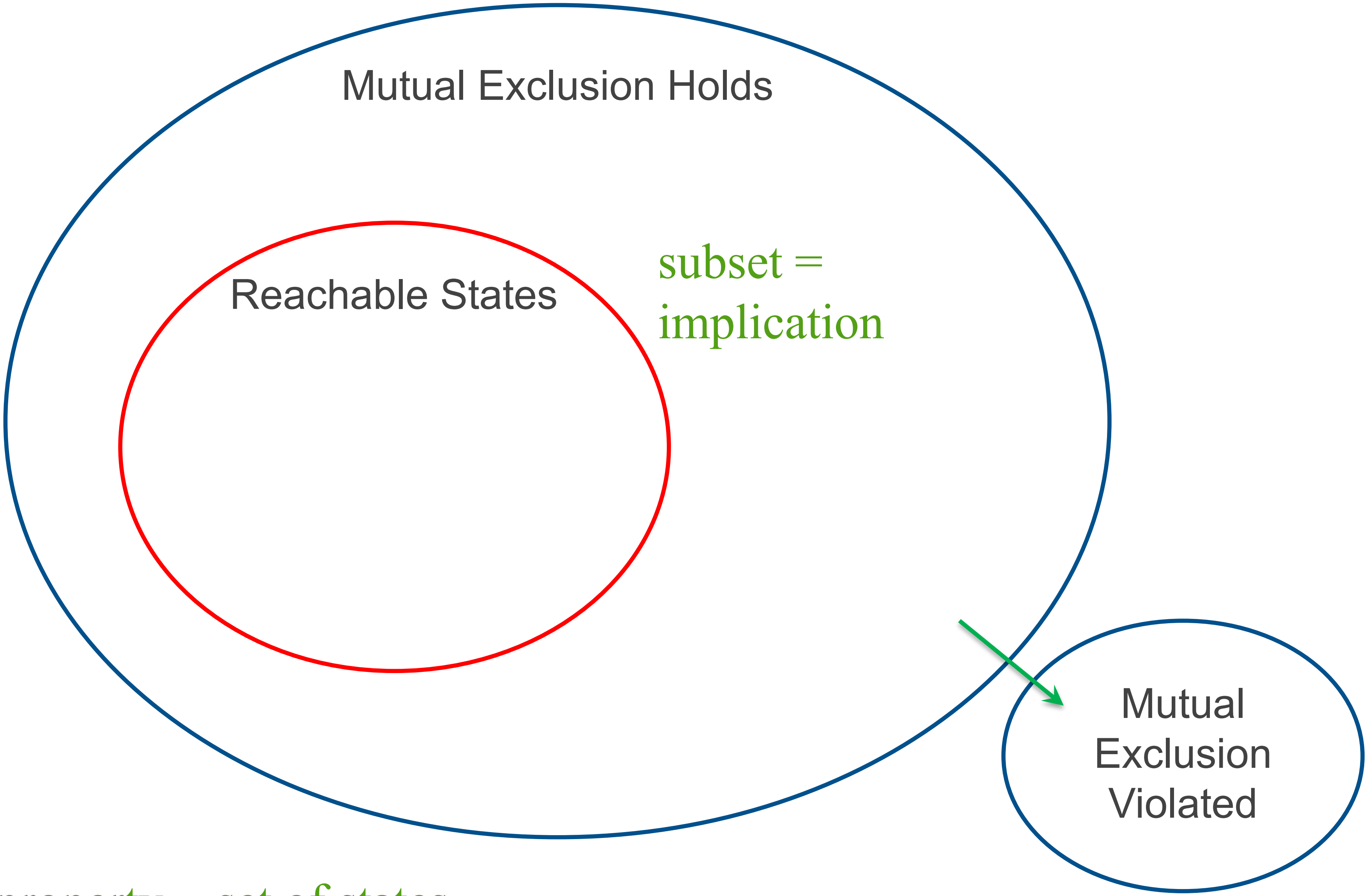

property = set of states

# Review in Pictures: State Space



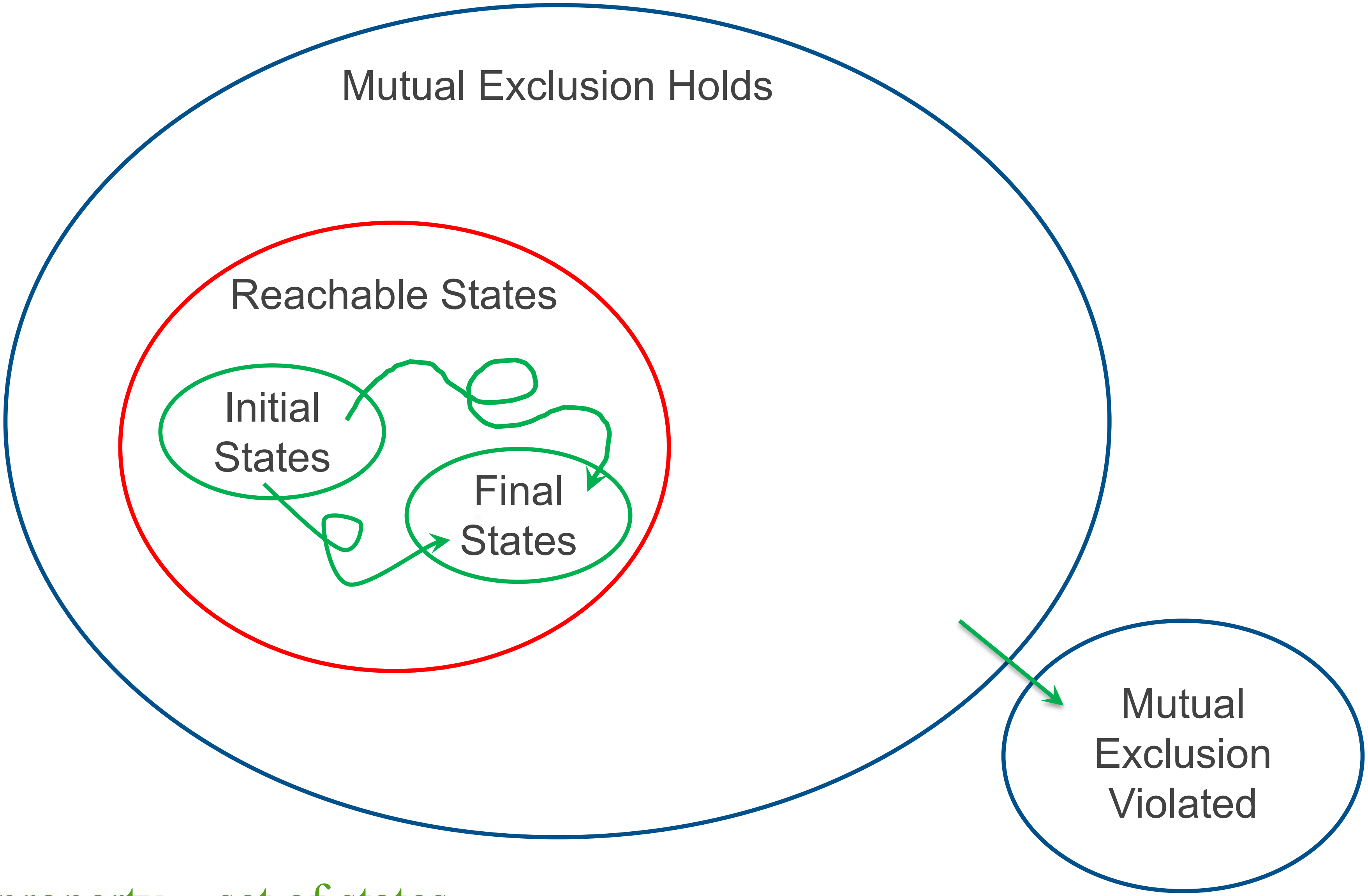

property = set of states

# Review in Pictures: State Space



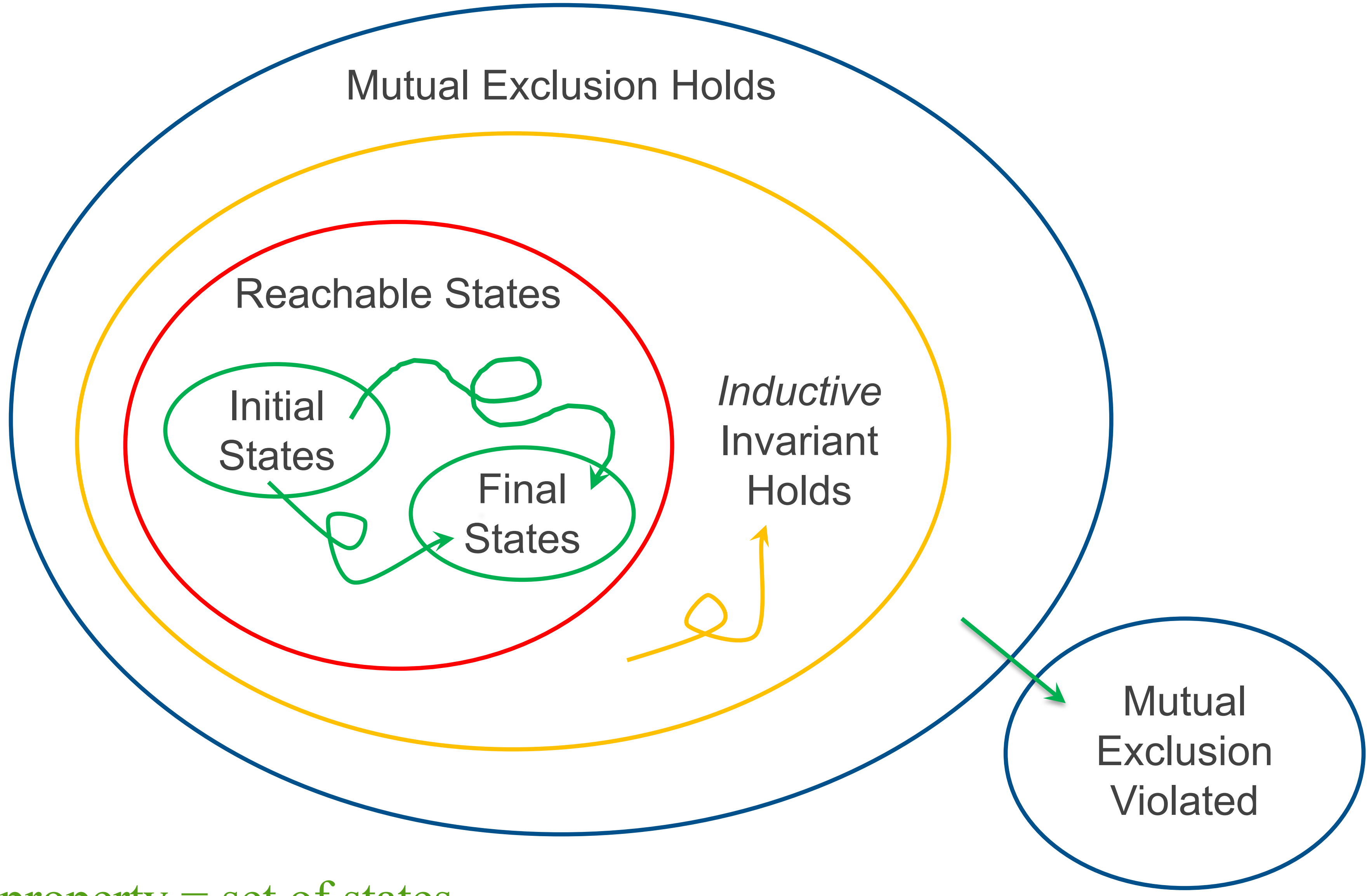

property = set of states

# Review in Pictures: State Space




property = set of states

# Review in Pictures: State Space




property = set of states

# Swapping lines 9 and 10?

```
1  sequential flags, turn
2
3  flags = [ False, False ]
4  turn = choose({0, 1})
5
6  def thread(self):
7      while choose({ False, True }):
8          # Enter critical section
9          flags[self] = True
10         turn = 1 − self
11         await (not flags[1 − self]) or (turn == self)
12
13         # critical section is here
14         @cs: assert atLabel(cs) == { (thread, self): 1 }
15
16         # Leave critical section
17         flags[self] = False
18
19 spawn thread(0)
20 spawn thread(1)
```

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm