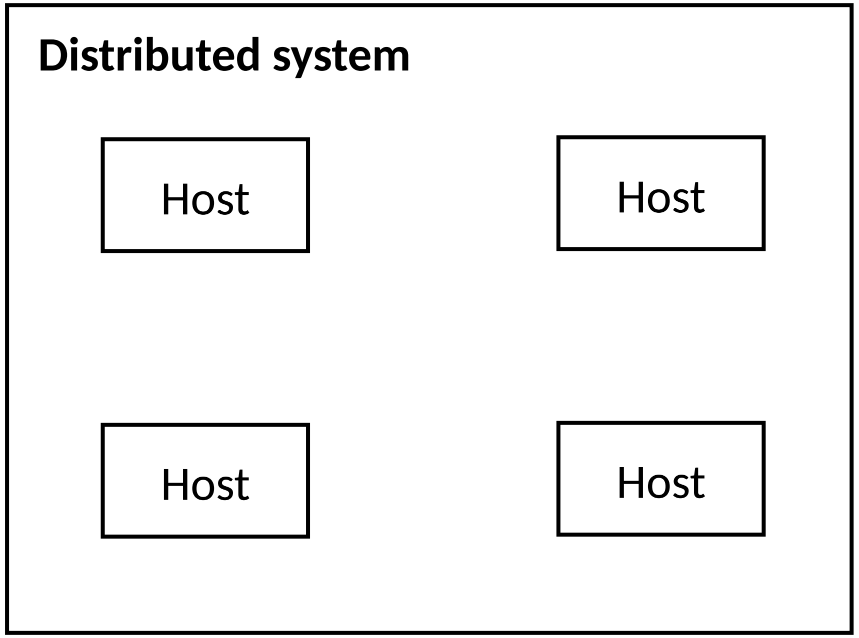
EECS498-003
Formal Verification of Systems Software

Material and slides created by

Jon Howell and Manos Kapritsos

Modeling distributed systems

A distributed system is composed of multiple hosts



Distributed System: attempt #1

```
module DistributedSystem {
  datatype Variables =
    Variables(hosts:seq<Host.Variables>)

  predicate Next (v:Variables, v’:Variables, hostid: nat)
{
    && Host.Next(v.hosts[hostid],v'.hosts[hostid]))
    && forall otherHost:nat | otherHost != hostid ::
        v’.hosts[otherHost] == v.hosts[otherHost]
  }
}
```

Modeling the network - Ordering

In order delivery

Out of order delivery

Modeling the network - Duplication

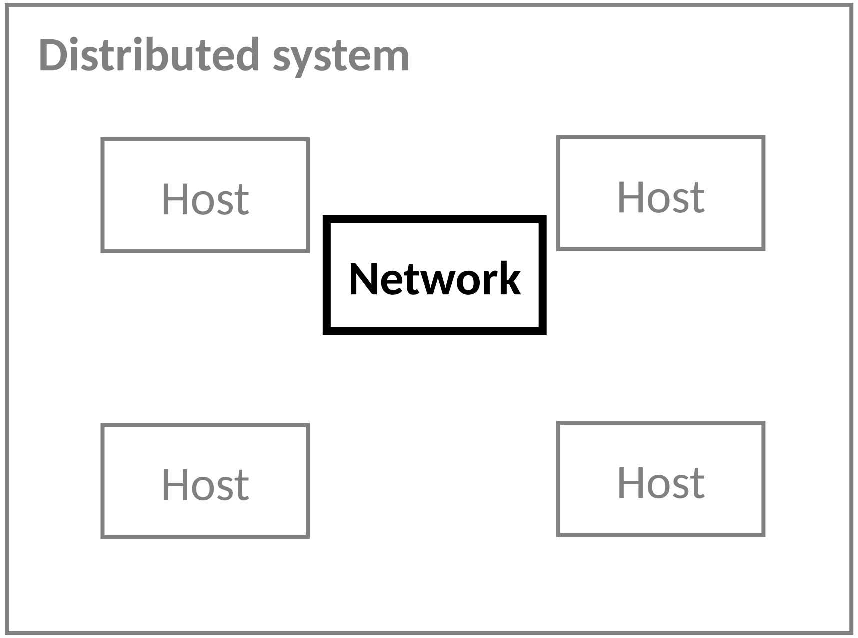
- Can the network duplicate messages?
- How does that affect our network model?

Modeling the network - Integrity

- Can the network tamper with messages?
- How does that affect our network model?

Modeling the network

```
datatype Option<T> = Some(value:T) | None
datatype MessageOps = MessageOps(
                     recv:Option<Message>,
send:Option<Message>)
```

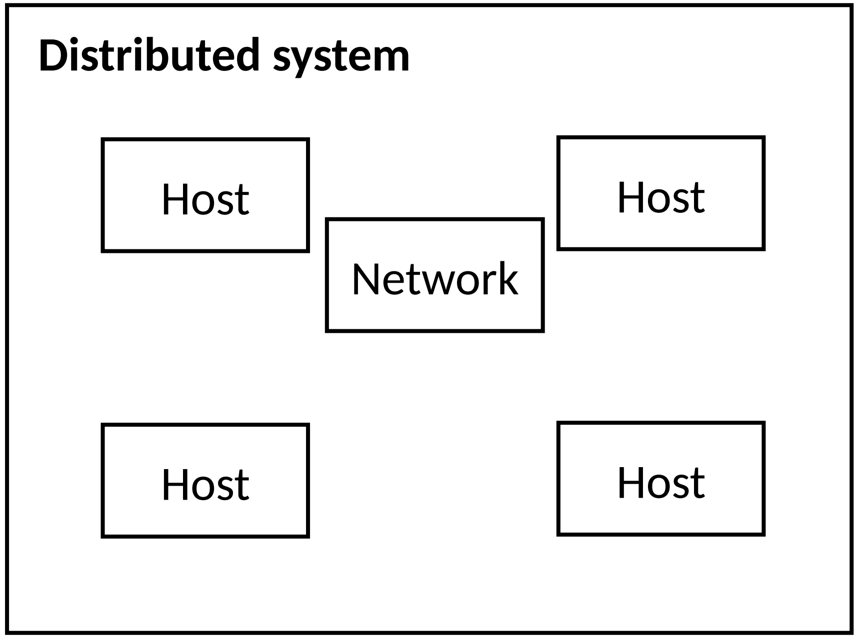


Network module

```
module Network {
  datatype Variables =
    Variables(sentMsgs: set<Message>)

  predicate Next(v, v’, msgOps:MessageOps) {
    // can only receive messages that have been sent
    && (msgOps.recv.Some? ==> msgOps.recv.value in
v.sentMsgs)
    // Record the sent message, if there was one
    && v'.sentMsgs ==
       v.sentMsgs + if msgOps.send.None? then {}
                    else {msgOps.send.value}
  }
}
```

A distributed system is composed of multiple hosts **and a network**



Distributed system: attempt #2

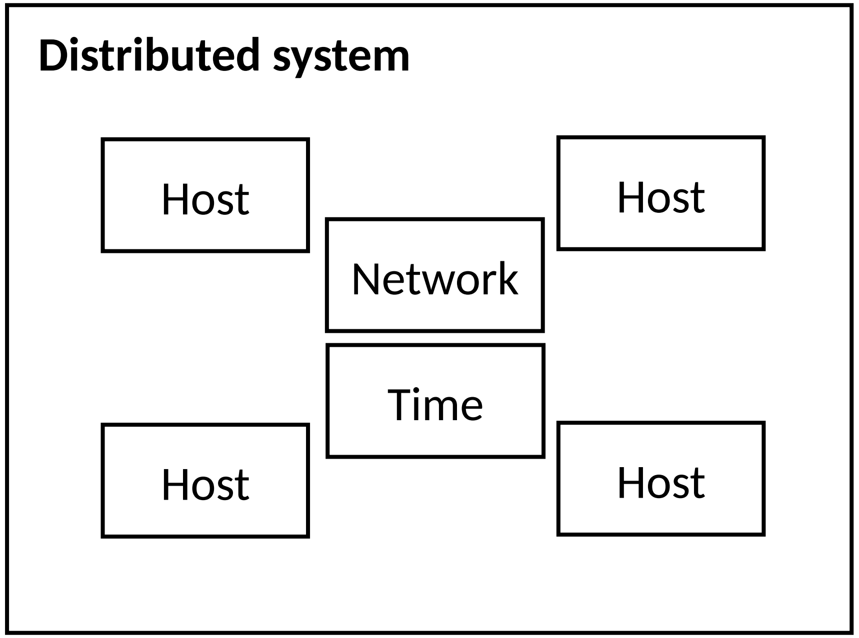
```
module DistributedSystem {
  datatype Variables =
    Variables(hosts:seq<Host.Variables>,
              network: Network.Variables)

  predicate HostAction(v, v’, hostid, msgOps) {
    && Host.Next(v.hosts[hostid],v'.hosts[hostid],msgOps))
    && forall otherHost:nat | otherHost != hostid ::
        v’.hosts[otherHost] == v.hosts[otherHost]
  }

  predicate Next(v, v’, hostid, msgOps: MessageOps) {
    && HostAction(v, v’, hostid, msgOps)
    && Network.Next(v, v’, msgOps)
  }
}
```

Binding variable

A distributed system is composed of multiple hosts, **a network and clocks**

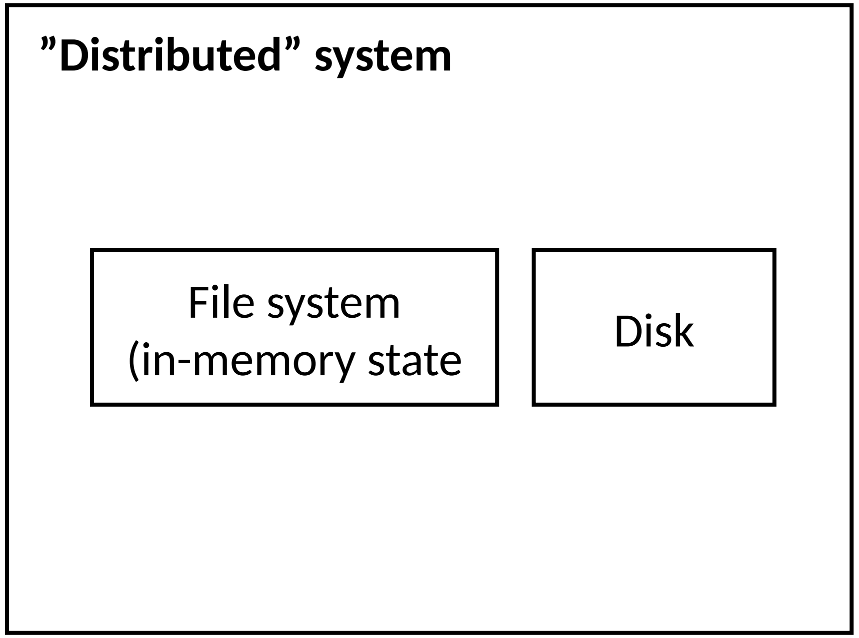


Distributed system: attempt #3

```
module DistributedSystem {
  datatype Variables =
    Variables(hosts:seq<Host.Variables>,
              network: Network.Variables,
              time: Time.Variables)

  predicate Next(v, v', hostid, msgOps: MessageOps,
clk:Time) {
    || (&& HostAction(v, v', hostid, msgOps)
        && Network.Next(v, v', msgOps)
        && Time.Read(v.time, clk))
    || (&& Time.Advance(v.time, v'.time)
        && v'.hosts == v.hosts
        && v'.network == v.network)
  }
}
```

This modeling applies to all asynchronous systems

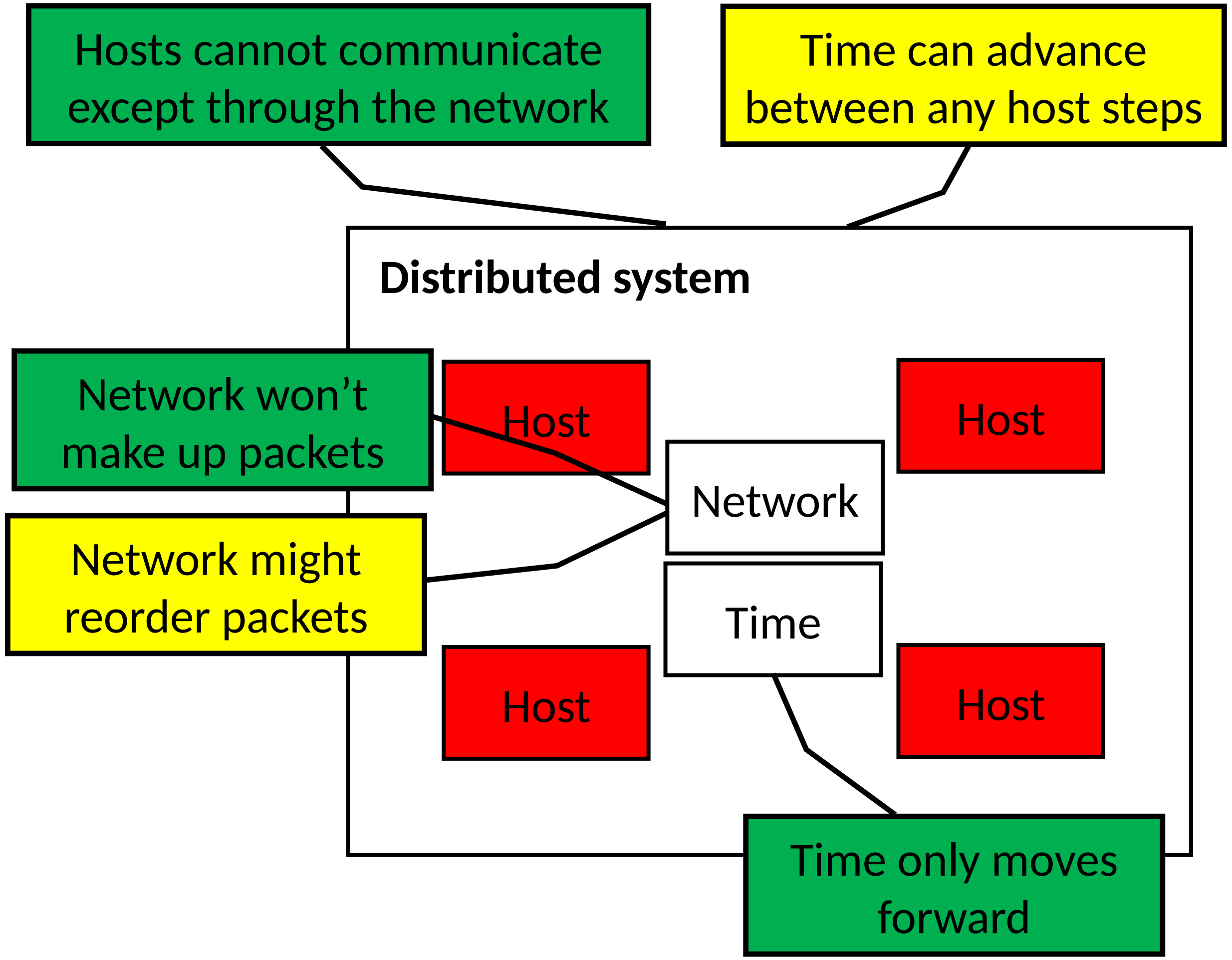


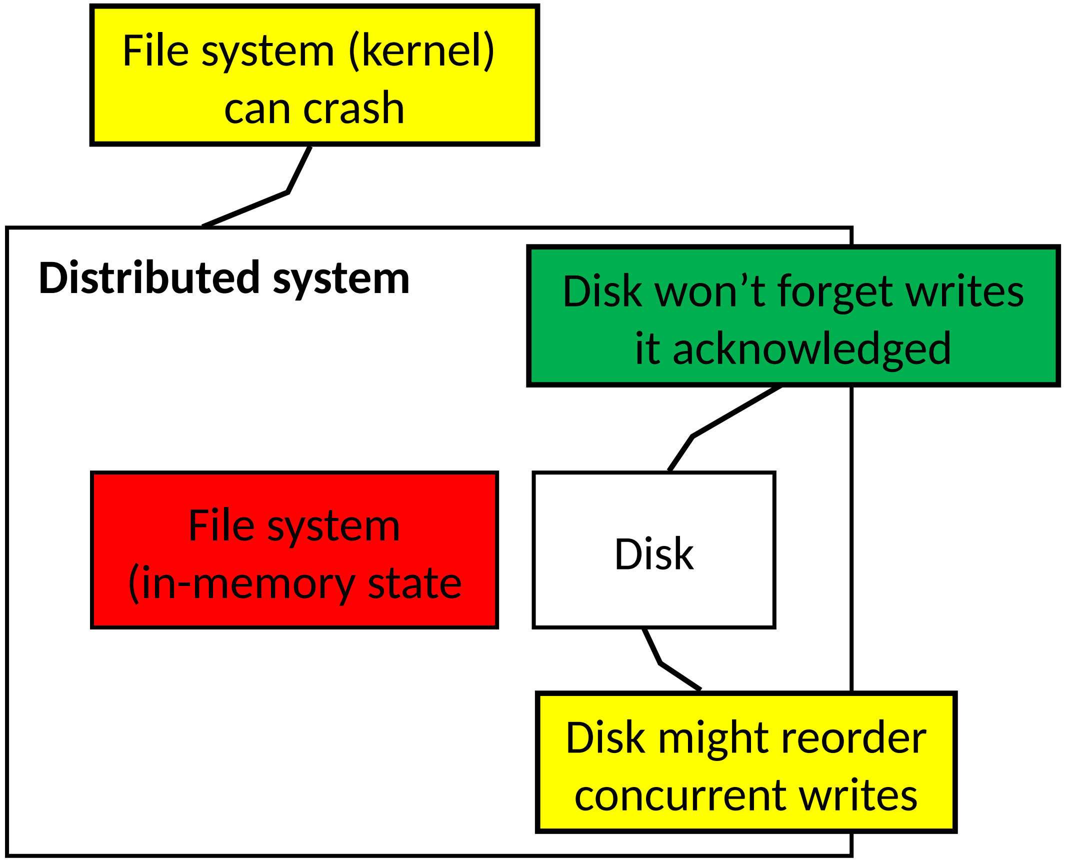

```
module DistributedSystem {
  datatype Variables =
    Variables(fs: FileSystem.Variables,
              disk: Disk.Variables)

  predicate Next(v, v’) {
    || (exists io ::
        && FileSystem.Next(v.fs, v’.fs, io)
        && Disk.Next(v.disk, v’.disk, io)
    || ( // Crash!
        && FileSystem.Init(v’.fs)
        && v’.disk == v.disk
       )
  }
}
```

Binding variable

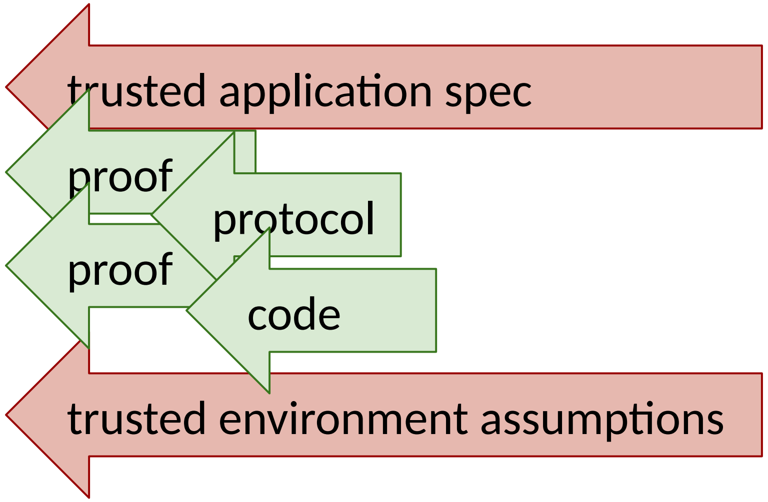
Trusted vs proven





SPECIFICATION: the systems specification sandwich

image: pixabay



Midterm logistics

- Time: Thursday, October 17, 6-8pm
- Location: BBB 1670

- Closed-book exam, allowed one ”cheat-sheet”, double-sided, 10pt minimum

- We assume knowledge of Dafny, but no ”guessing”

Administrivia

- No lectures next week
  - Tuesday is Fall study break
  - Thursday is the midterm

- Also, no lab next week
- I will still hold OH next Thursday

- Please fill out midterm evaluations
  - Grad students: 80%
  - Undergrad students: 17%

Recap of Chapters 1-4

Recap of Chapter 1: Dafny mechanics

- Primitive types
- Quantifiers
- Assertions
- Recursion
- Loop invariants
- Datatypes
- Triggers

Triggers

- **Q:** Does Dafny verify this code?

```
predicate P(x:int)
predicate Q(x:int)

method test()
  requires forall x :: P(x) && Q(x)
  ensures Q(0)
{
}
```

A: Only if it’s smart enough to pick the right trigger

Imagine you are the solver

```
requires forall x :: P(x) && Q(x)
```

I wonder if P(0) is a useful fact…

I wonder if P(1) is a useful fact…

I wonder if P(2) is a useful fact…

I wonder if P(3) is a useful fact…

I wonder if P(4) is a useful fact…

I wonder if P(5) is a useful fact…

I wonder if P(6) is a useful fact…

I wonder if P(7) is a useful fact…

I wonder if P(8) is a useful fact…

I wonder if P(9) is a useful fact…

I wonder if P(10) is a useful fact…

I wonder if P(11) is a useful fact…

I wonder if P(12) is a useful fact…

I wonder if P(13) is a useful fact…

I wonder if P(14) is a useful fact…

I wonder if P(15) is a useful fact…

I wonder if P(16) is a useful fact…

I wonder if P(17) is a useful fact…

I wonder if Q(0) is a useful fact…

I wonder if Q(1) is a useful fact…

I wonder if Q(2) is a useful fact…

I wonder if Q(3) is a useful fact…

I wonder if Q(4) is a useful fact…

I wonder if Q(5) is a useful fact…

I wonder if Q(6) is a useful fact…

I wonder if Q(7) is a useful fact…

I wonder if Q(8) is a useful fact…

I wonder if Q(9) is a useful fact…

I wonder if Q(10) is a useful fact…

I wonder if Q(11) is a useful fact…

I wonder if Q(12) is a useful fact…

I wonder if Q(13) is a useful fact…

I wonder if Q(14) is a useful fact…

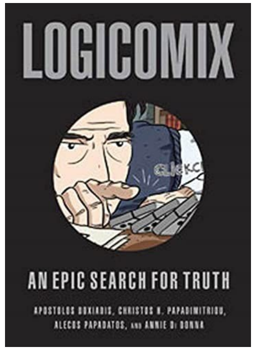
I wonder if Q(15) is a useful fact…

I wonder if Q(16) is a useful fact…

I wonder if Q(17) is a useful fact…

Completeness vs Soundness

- Proving a program correct is undecidable
  - i.e. it is impossible to write a program that always correctly answers the question: is this program correct
- Side note:
  - Logicomix
  - Veritasium



- Provers embrace incompleteness while guarding soundness
  - Incompleteness: the prover **may say "no"** to a correct program
  - Soundness: the prover **will never say "yes"** to an incorrect program

Triggers

- **What is a trigger?**

A syntactic pattern involving quantified variables

A heuristic to let the solver know when to **instantiate** the quantifier

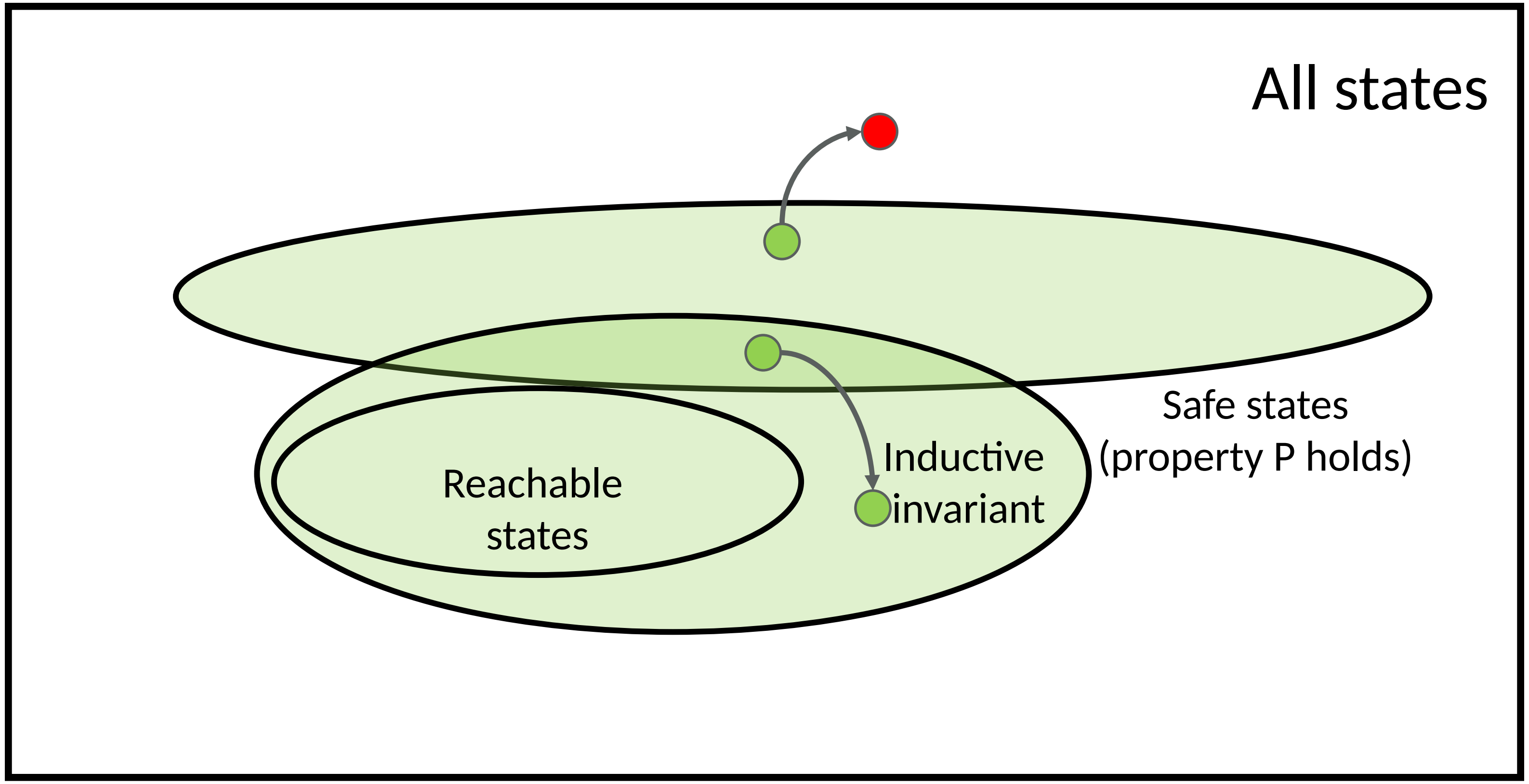
Recap of Chapter 2: Specification

Specifications are trusted!

Recap of Chapter 3: State machines

- Express the behavior of a system
- Main components: Constants/Variables, Init() and Next() predicates
- Advanced usage: Jay Normal Form

Recap of Chapter 4: Inductive invariants



Good luck with the midterm!