

EECS498-008

Formal Verification

of Systems Software

Material and slides created by
Jon Howell and Manos Kapritsos

A **state** is an assignment of values to variables

```
datatype Card = Shelf | Patron(name: string)
datatype Book = Book(title: string)
type Variables = map<Book, Card>
```

The **state space** is the set of possible assignments.

The Martian: Shelf
Snow Crash: Shelf

The Martian: Shelf
Snow Crash: Jon

~~The Martian: Jon
Snow Crash: Jon~~

The Martian: Manos
Snow Crash: Jon

A state machine definition

```

predicate Init(v: Variables) {
  forall book | book in v :: v[book] == Shelf
}

datatype Card = Shelf | Patron(name: string)
datatype Book = Book(title: string)
type Variables = map<Book, Card>

predicate CheckOut(v : Variables, v' : Variables, book: Book, name: string) {
  && book in v
  && v[book] == Shelf
  && (forall book | book in v :: v[book] != Patron(name))
  && v' == v[book := Patron(name)]
}

predicate CheckIn(v : Variables, v' : Variables, book: Book, name: string) {
  && book in v
  && v[book] == Patron(name)
  && v' == v[book := Shelf]
}

predicate Next(v: Variables, v': Variables) {
  || (exists book, name :: CheckOut(v, v', book, name))
  || (exists book, name :: CheckIn(v, v', book, name))
}

```

enabling condition

“update”

Nondeterministic definition

A **behavior** is the set of **all possible executions**

```

predicate CheckOut(v, v', book, name) {
  && book in v
  && v[book] == Shelf
  && (forall book | book in v :: v[book] !=
Patron(name))
  && v' == v[book := Patron(name)]
}
predicate CheckIn(v, v', book, name) {
  && book in v
  && v[book] == Patron(name)
  && v' == v[book := Shelf]
}

```

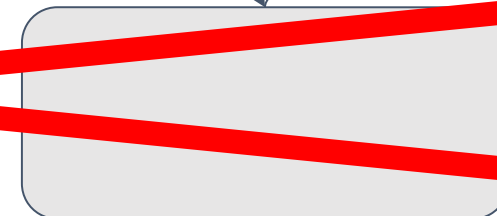
check out

???

The Martian: Shelf
Snow Crash: Shelf

The Martian: Shelf
Snow Crash: Jon

The Martian: Shelf
Snow Crash: Rob



State machine strengths

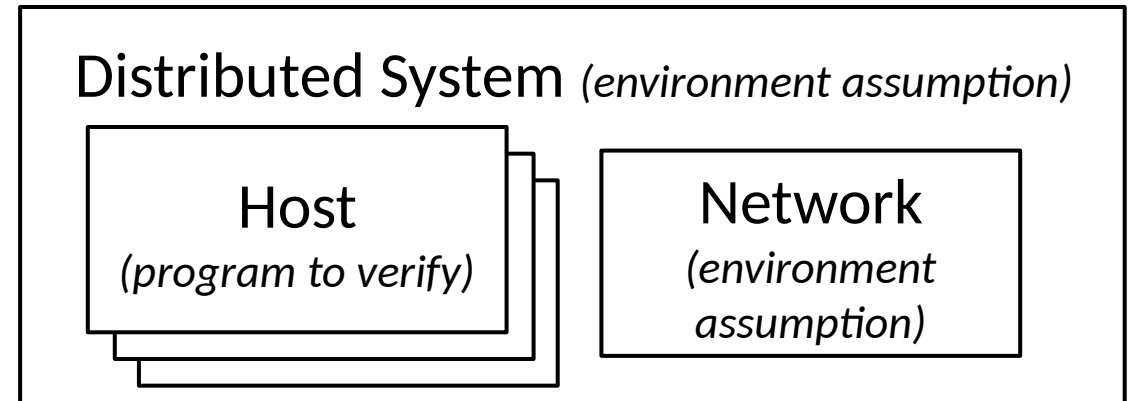
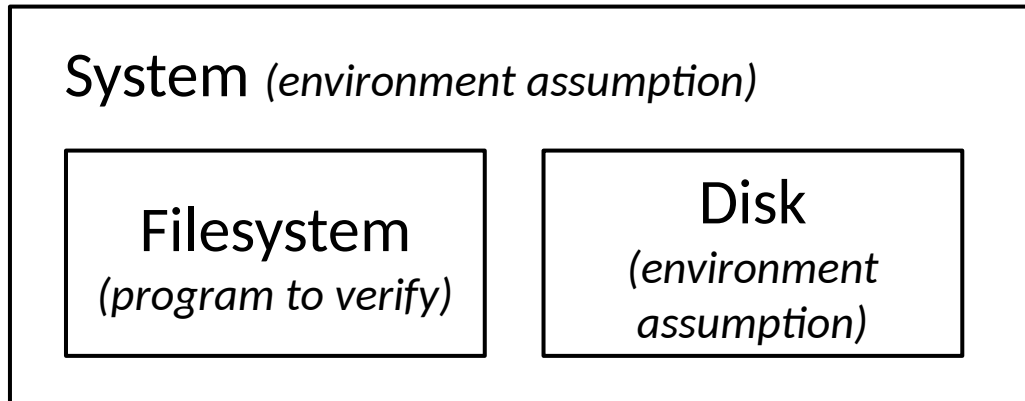
- Abstraction
 - States can be abstract
 - Model an infinite map instead of an efficient pivot table
 - Next predicate is nondeterministic:
 - Implementation may only select some of the choices
 - Can model Murphy's law (e.g. crash tolerance) or an adversary

State machine strengths

- Abstraction
- Asynchrony
 - Each step of a state machine is conceptually atomic
 - Interleaved steps capture asynchrony: threads, host processes, adversaries
 - Designer decides how precisely to model interleaving; can refine/reduce

State machine strengths

- Abstraction
- Asynchrony
- Environment
 - Model a proposed program with one state machine (verified)
 - Model (adversarial) environment with another (trusted)
 - Compound state machine models their interactions (trusted)

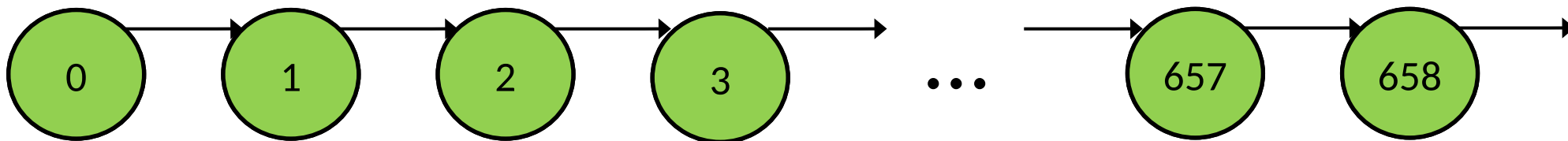


Chapter 4: Proving properties

Expressing a system as a state machine allows us to **prove** that it has certain properties

- We will focus on safety properties
 - i.e. properties that hold throughout the execution

Basic tool: induction



- Show that the property holds on state 0
- Show that if the property holds on state k , it must hold on state $k+1$

Let's prove a safety invariant!

```
predicate Safety(v:Variables) {  
  true // TBD  
}
```

```
lemma SafetyProof()  
  ensures forall v :: Init(v) ==> Safety(v)  
  ensures forall v, v' :: Safety(v) && Next(v, v') ==> Safety(v')  
{  
}
```

Base case

Inductive Step

Let's prove a safety invariant!

Interactive proof development in editor

*Bisection debugging,
case analysis,
existential instantiation*



As you begin writing more interesting specs, proofs will be nontrivial.

Pull all the nondeterminism into one place, and get a receipt.



Jay Normal Form

```
datatype Step =  
  | Action1Step( <parameters> )  
  | Action2Step( <parameters> )  
  ...  
  
predicate NextStep(v: Variables, v': Variables, step:Step)  
{  
  match step  
    case Action1Step(<parameters>) => Action1(v, v', <parameters>)  
    case Action2Step(<parameters>) => Action2(v, v', <parameters>)  
    ...  
}  
predicate Next(v: Variables, v': Variables)  
{  
  exists step :: NextStep(v, v', step)  
}
```