

EECS498-008

Formal Verification

of Systems Software

Material and slides created by
Jon Howell and Manos Kapritsos

Some new Dafny syntax

Datatype member functions

```
datatype Pet = Dog | Cat | Ant | Spider {  
  function CountLegs() : int {  
    match this  
      case Dog => 4  
      case Cat => 4  
      case Ant => 6  
      case Spider => 8  
  }  
}  
  
function ShoesForTwo(pet: Pet) : int {  
  2 * pet.CountLegs()  
}
```

Some new Dafny syntax

Calc statements

```
assert a == b;  
assert b == c;  
assert c == d;
```

```
calc {  
  a;  
  b;  
  c;  
  d;  
}
```

Some new Dafny syntax

Calc statements

```
assert a == b;  
assert b == c;  
assert c == d;
```

```
calc {  
  a;  
  { MyUsefulLemma(a,b); }  
  b;  
  c;  
  d;  
}
```

Some new Dafny syntax

Calc statements

```
assert a == b;  
assert b == c;  
assert c == d;
```

```
calc ==> {  
  a;  
  { MyUsefulLemma(a,b); }  
  b;  
  c;  
  d;  
}
```

Some new Dafny syntax

Choose operator

```
assert 1 % 7 == 1;  
assert exists x :: x % 7 == 1;  
var x :| x % 7 == 1;
```

Choose x such that...

Administrivia

Remember that Problem Set 1 is due this Friday

I'm still missing some of your pictures. Please send me your picture (manosk@umich.edu) with the Subject “EECS498-008 picture”

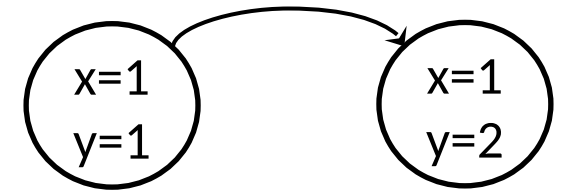
Chapter 3: Building state machines

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

An **action** is a transition from one state to another

An **execution** is a sequence of states

We will capture executions with **state machines**



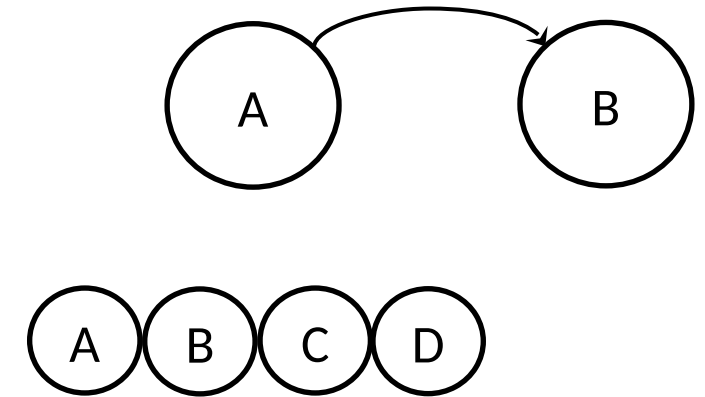
Building state machines

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

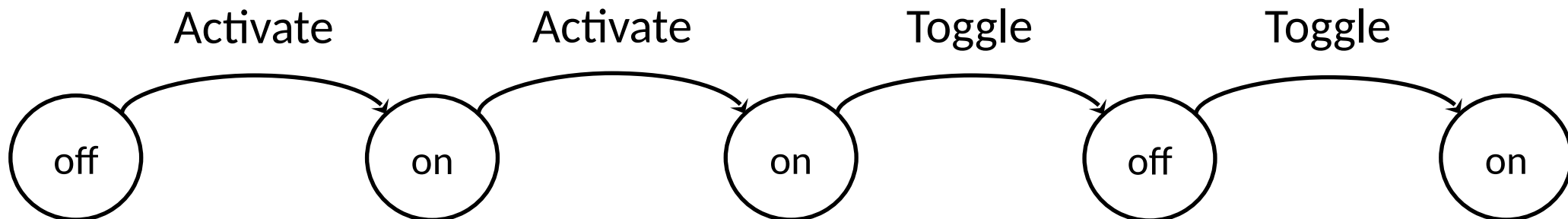
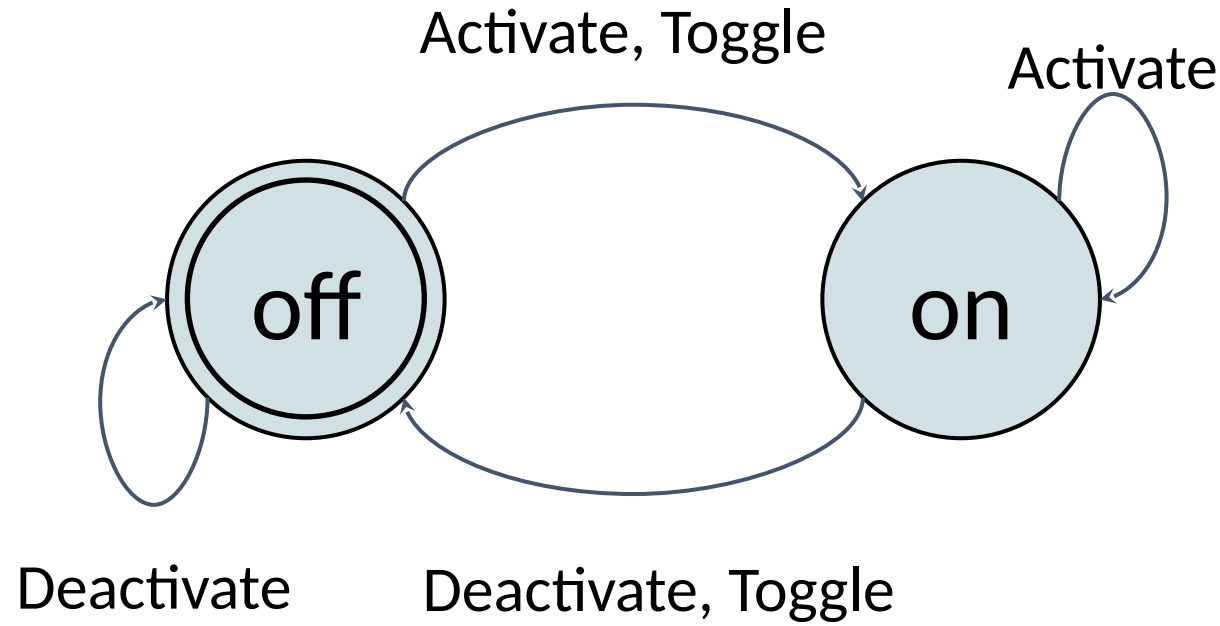
An **action** is a transition from one state to another

An **execution** is a sequence of states

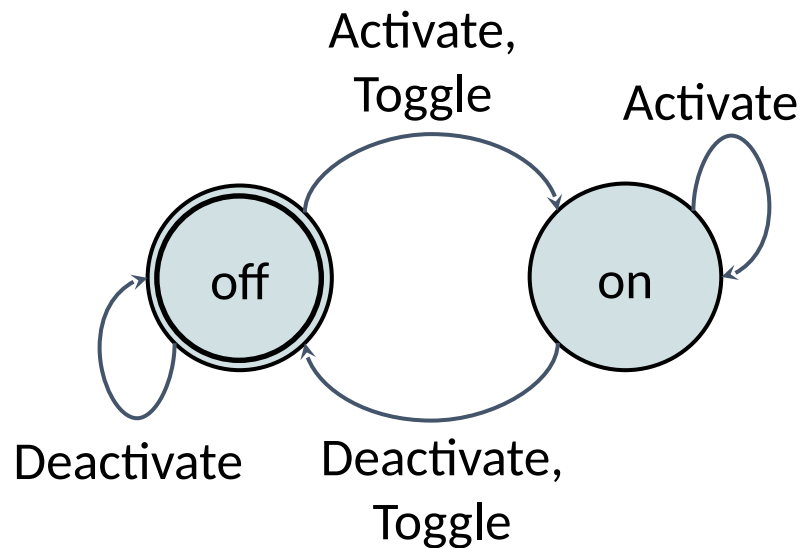
We will capture executions with **state machines**



The Switch state machine



The Switch state machine



```

datatype SwitchState = On | Off
datatype Variables =
    Variables(switch:SwitchState)
predicate Init(v:Variables) {
    v.switch == Off
}

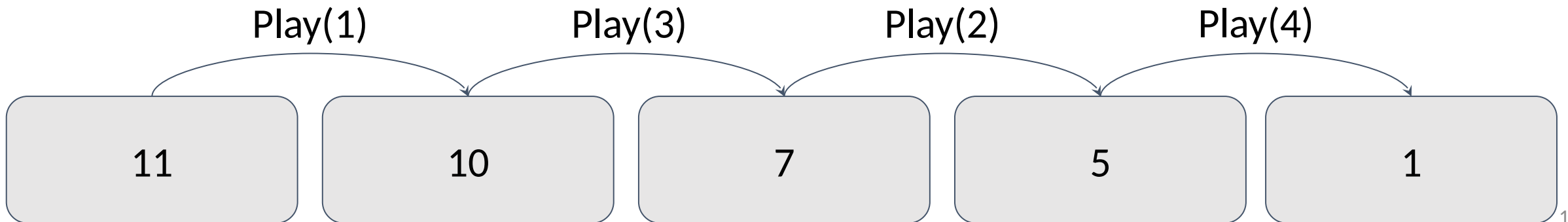
```

```

predicate Activate(v:Variables, v':Variables) {
    v'.switch == On
}
predicate Deactivate(v:Variables, v':Variables) {
    v'.switch == Off
}
predicate Toggle(v:Variables, v':Variables) {
    On v'.switch == if v.switch.On? then Off else
    On
}
predicate Next(v:Variables, v':Variables) {
    || Activate(v, v')
    || Deactivate(v, v')
    || Toggle(v, v')
}

```

The Game of Nim



The Nim state machine

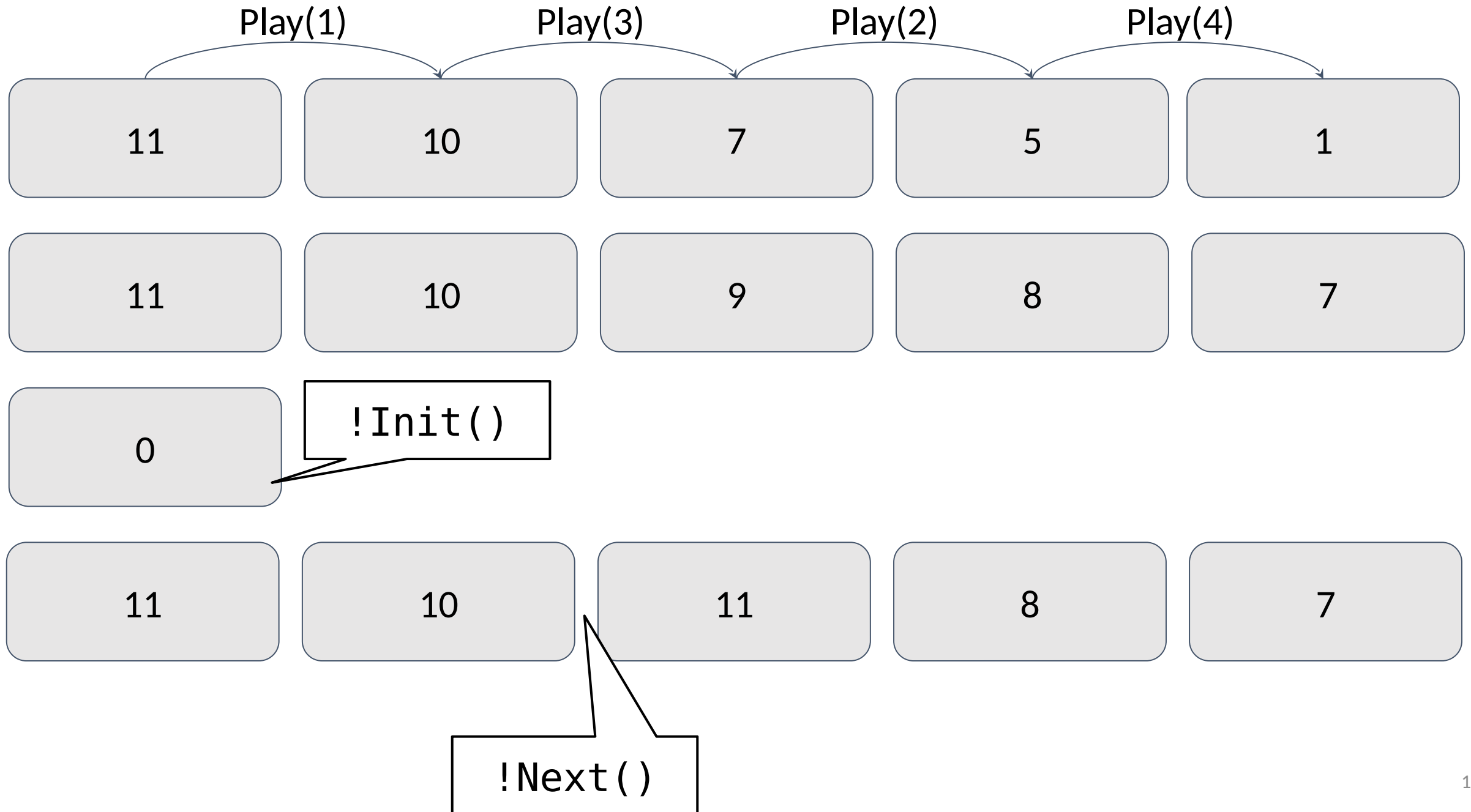
```
datatype Variables = Variables(tokens:nat)
predicate Init(v:Variables) {
    v.tokens > 0
}

predicate Play(v:Variables, v':Variables, take:nat) {
    && 1 <= take <= 4
    && take <= v.tokens
    && v'.tokens == v.tokens - take
}

predicate Next(v:Variables, v':Variables)
{
    exists take :: Play(v, v', take)
}
```

enabling condition

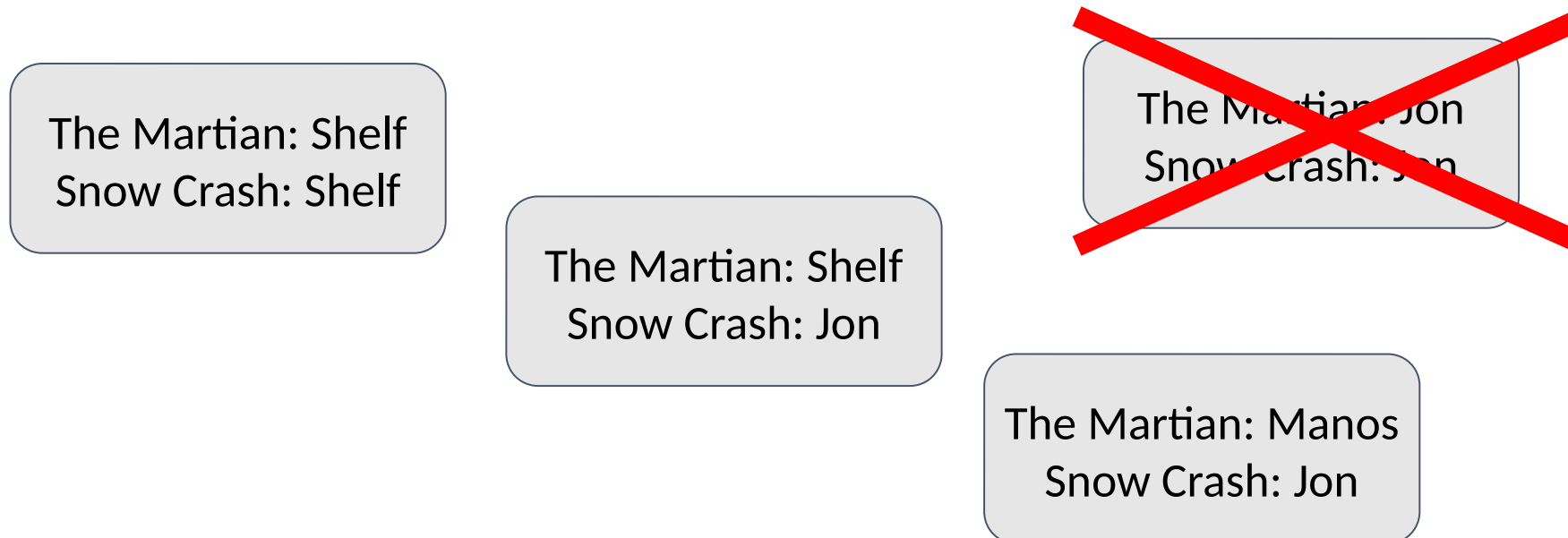
“update”



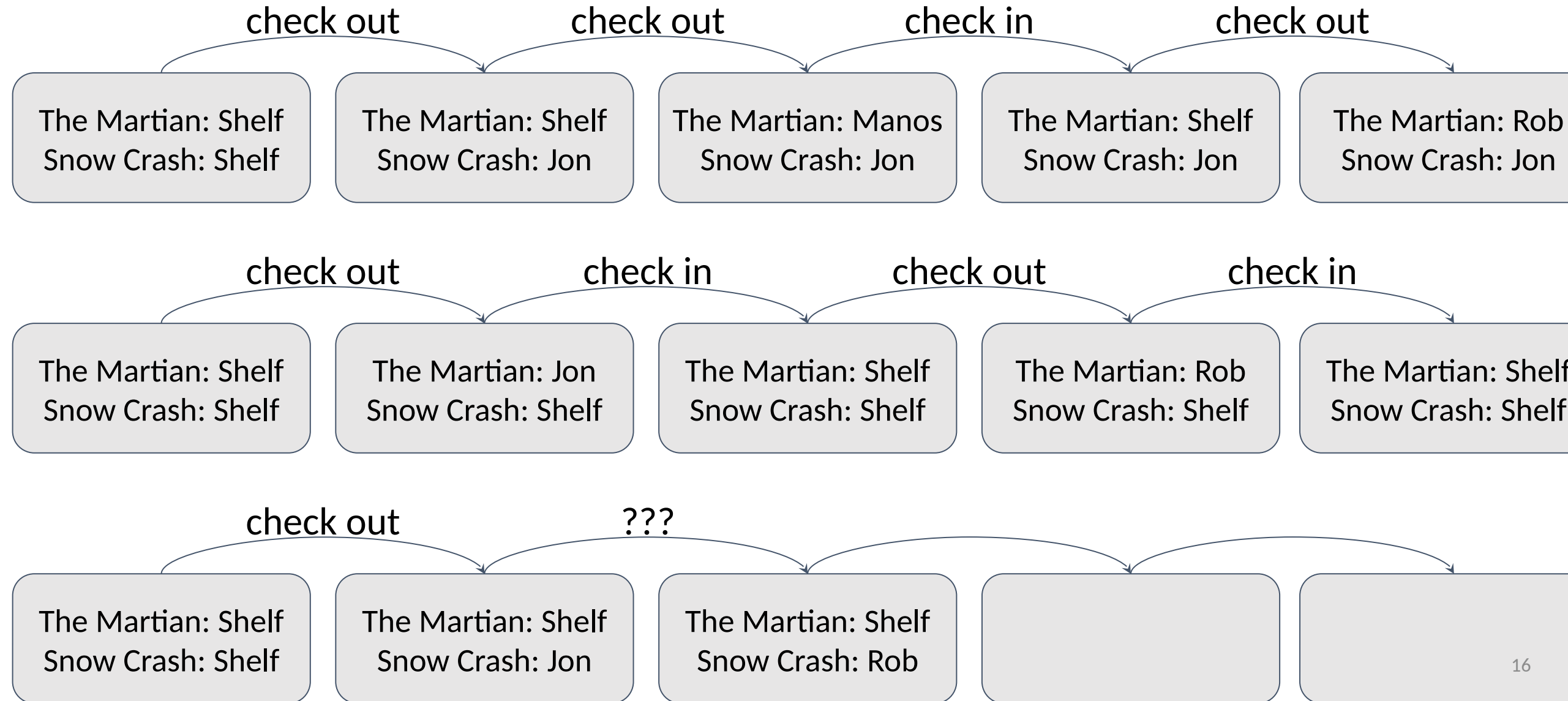
A **state** is an assignment of values to variables

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

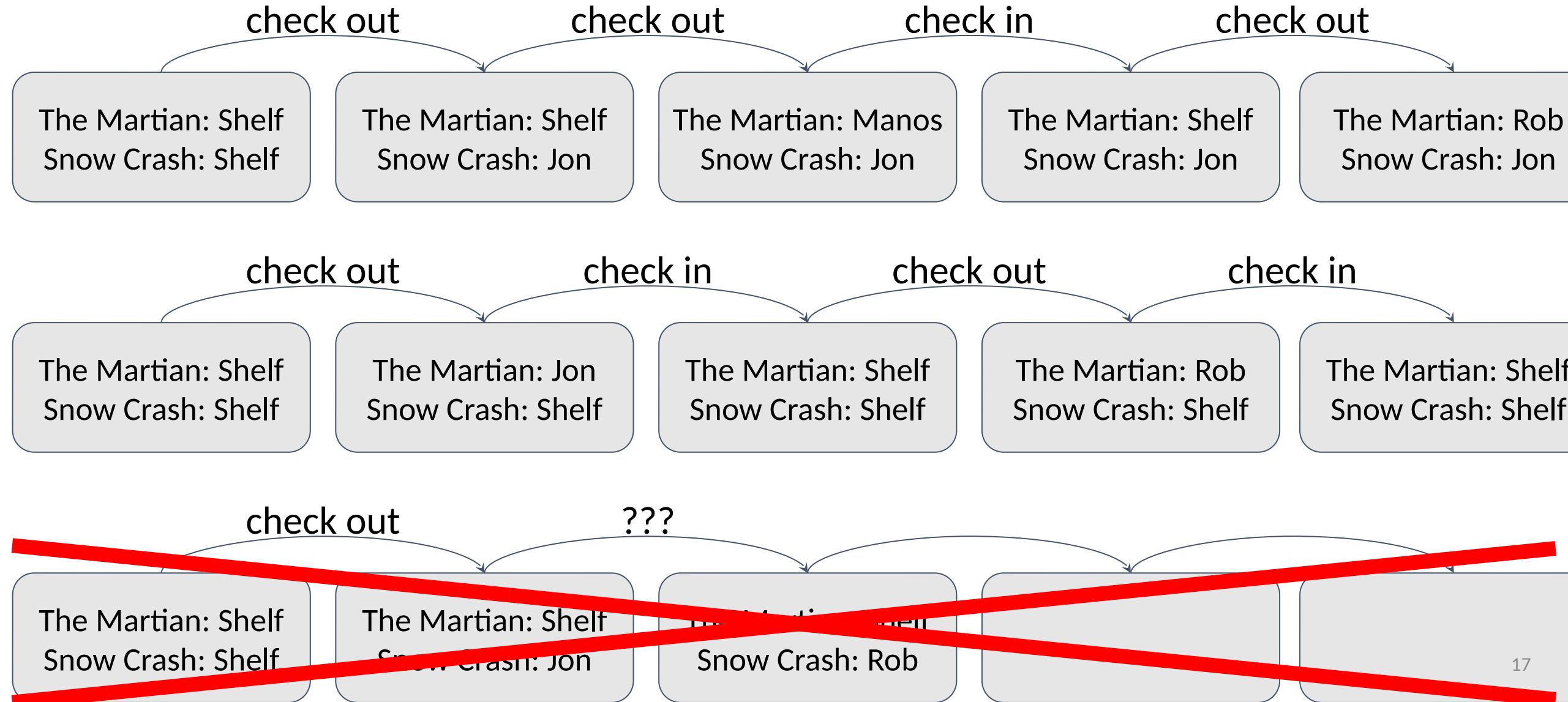
The **state space** is the set of possible assignments.



An **execution** is an infinite sequence of states



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



How should we define a behavior?

With a **program**?

Its variables define its state space
Its executions define its behavior

Weaknesses:

- concreteness
- nondeterminism
- asynchrony
- environment

How should we define a behavior?

With a state machine

Its **type** defines its state space

Its **initial states** and **transitions** define its behavior

A state machine definition

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

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

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

} enabling condition

} “update”

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

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

} 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]  
}
```

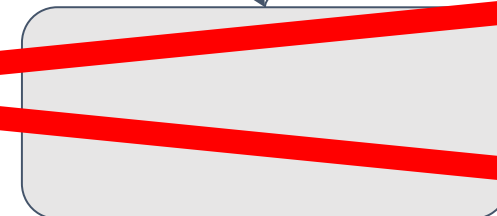
check out

???

The Martian: Shelf
Snow Crash: Shelf

The Martian: Shelf
Snow Crash: Jon

The Martian: Shelf
Snow Crash: Rob



State machine strengths

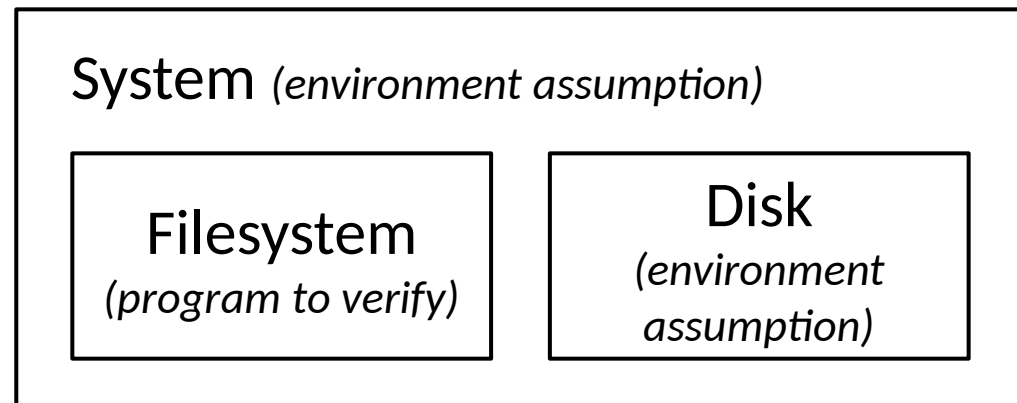
- Abstraction
 - States can be abstract
 - Model an infinite map instead of an efficient pivot table
 - Next predicate is nondeterministic:
 - Implementation may only select some of the choices
 - Can model Murphy's law (e.g. crash tolerance) or an adversary

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

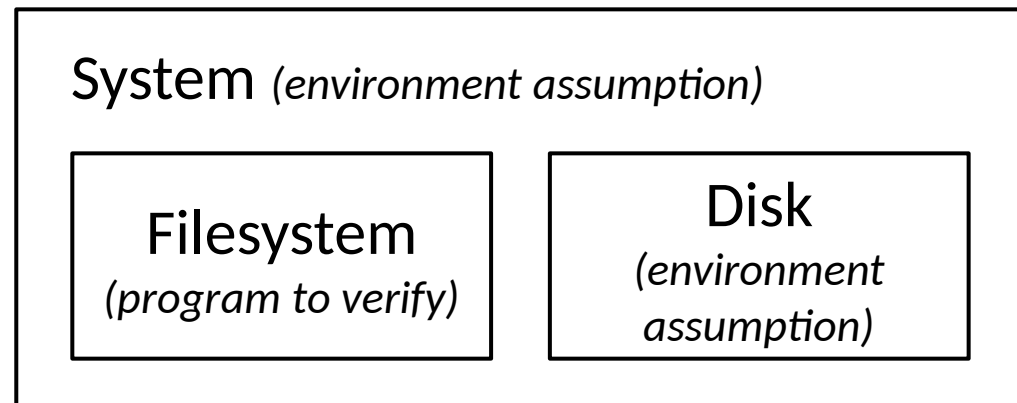
State machine strengths

- 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)



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)

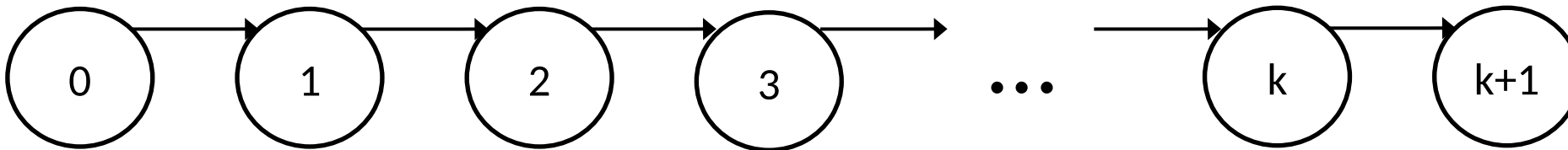


Chapter 4: Proving properties

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

- We will focus on safety properties

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:Library) {  
  true // TBD  
}
```

Base case

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

Inductive Step

Let's prove a safety invariant!

Interactive proof development in editor

Bisection debugging,

case analysis,

existential instantiation