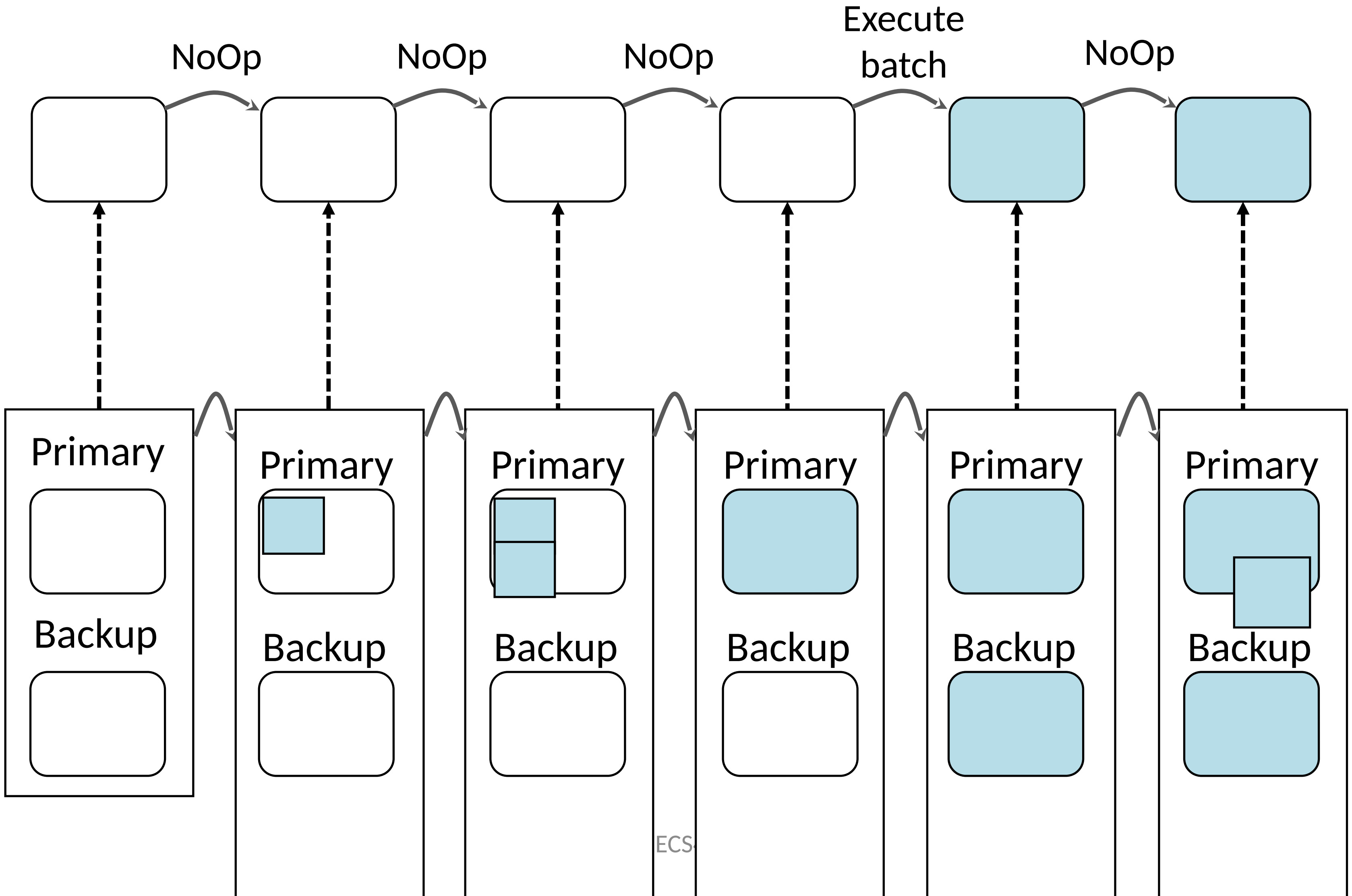
EECS498-003
Formal Verification of Systems Software

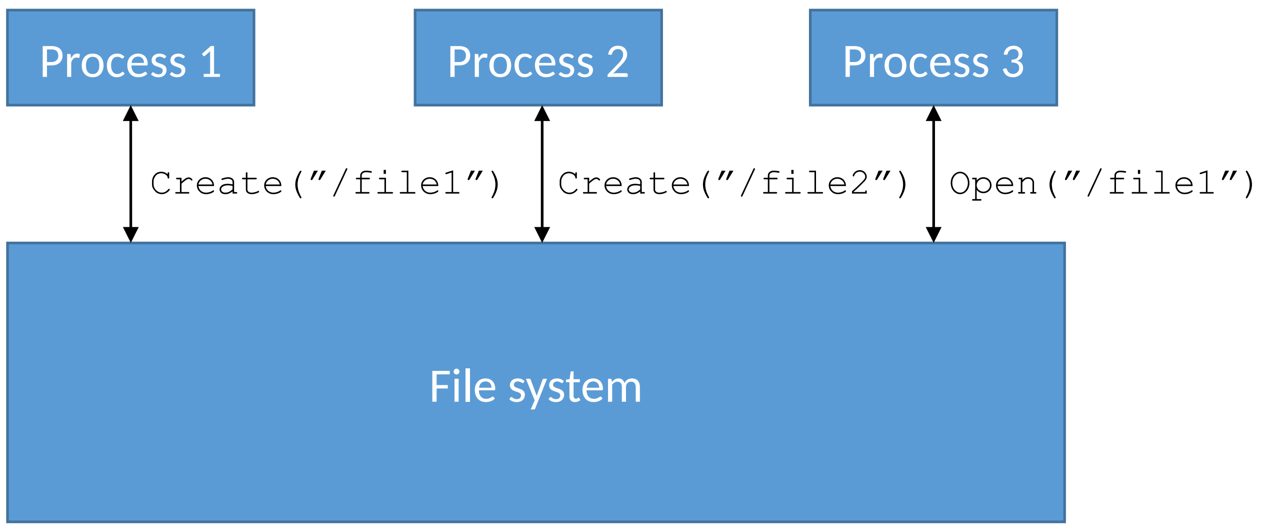
Material and slides created by

Jon Howell and Manos Kapritsos

A primary-backup protocol



World-visible events



Which of these behaviors are correct?
(assuming an initially empty file system)

Behavior #1

```
Create(f, "/file1")     (returns OK)
Create(f, "/file2")     (returns OK)
Create(d, "/dir")       (returns OK)
Create(f, "/dir/file1") (returns OK)
```

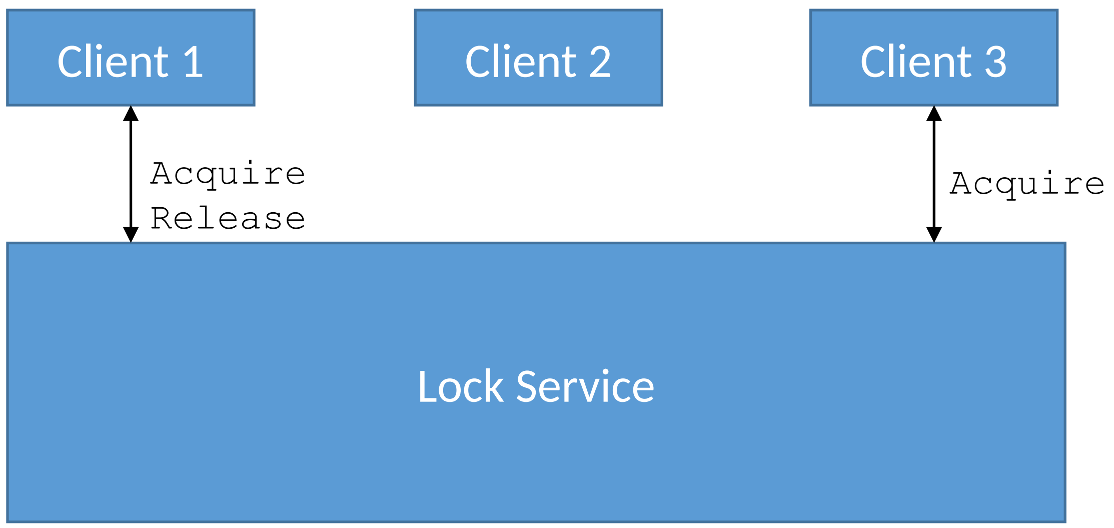
Behavior #2

```
Create(f, "/file1")     (returns OK)
Create(f, "/file2")     (returns OK)
Create(f, "/dir/file1") (returns Err)
```

Behavior #3

```
Create(f, "/file1")     (returns OK)
Write(f, "/file2")      (returns OK)
Create(d, "/dir")       (returns OK)
Create(f, "/dir/file1") (returns OK)
```

World-visible events



Which of these behaviors are correct?
(assuming no one holds the lock initially)

Behavior #1

```
Acquire(client1)
Acquire(client1)
Release(client1)
Release(client1)
```

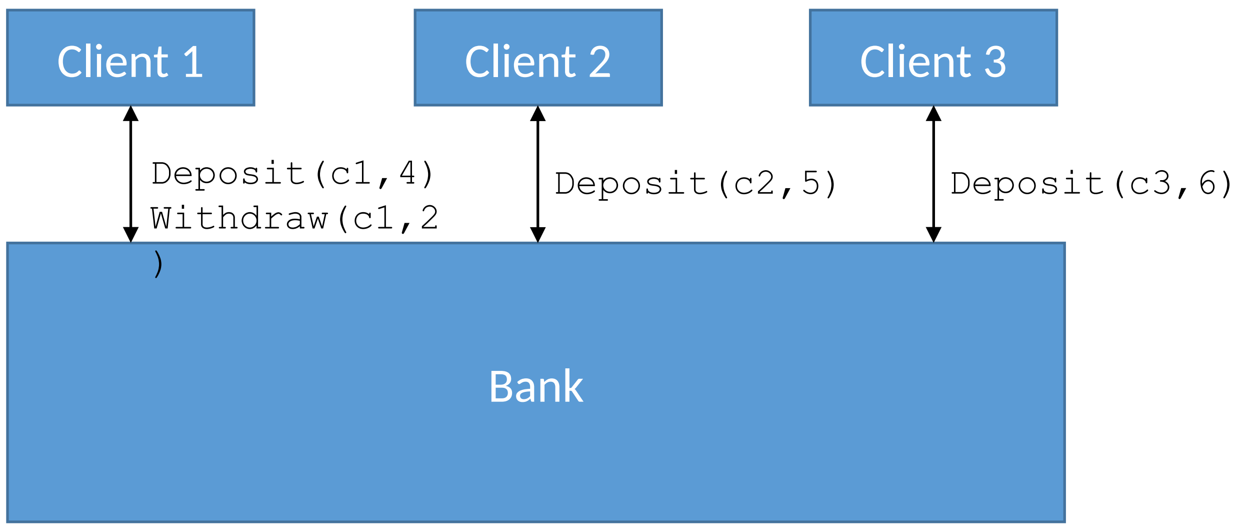
Behavior #2

```
Release(client2)
Acquire(client1)
Release(client1)
```

Behavior #3

```
Acquire(client1)
Release(client1)
Acquire(client2)
```

World-visible events



Which of these behaviors are correct?

(assuming all account are initially empty)

Behavior #1

```
Deposit(client1, 6)     (returns OK)
Withdraw(client1, 3)    (returns OK)
Withdraw(client1, 2)    (returns OK)
Deposit(client1, 3)     (returns Err)
```

Behavior #2

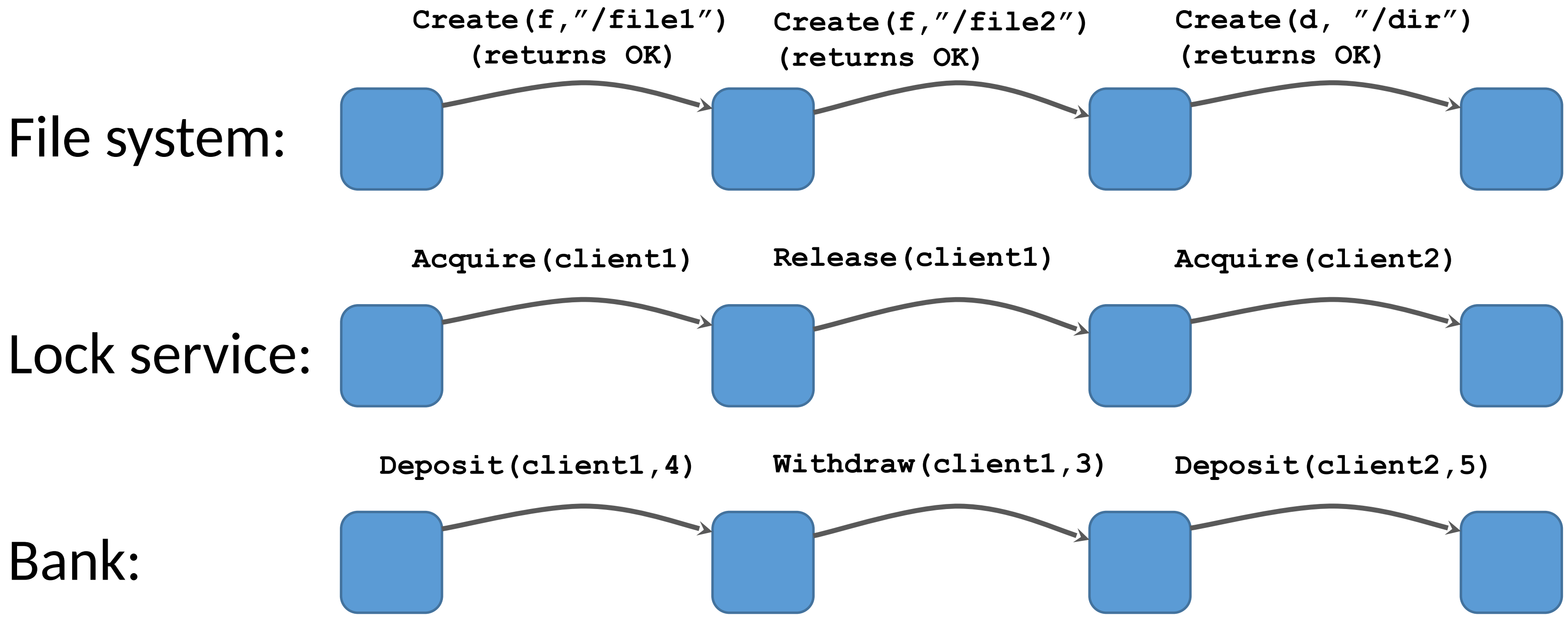
```
Deposit(client1, 6)     (returns OK)
Withdraw(client1, 3)    (returns OK)
Withdraw(client2, 2)    (returns OK)
```

Behavior #3

```
Deposit(client1, 6)     (returns OK)
Withdraw(client1, 3)    (returns OK)
Withdraw(client1, 2)    (returns OK)
Withdraw(client1, 3)     (returns Err)
```

Events define correctness

One should be able to evaluate the correctness of the system by inspecting a behavior (sequence) consisting of world-visible events



Event-enriched spec state machines

We will be adding events to our spec state machines

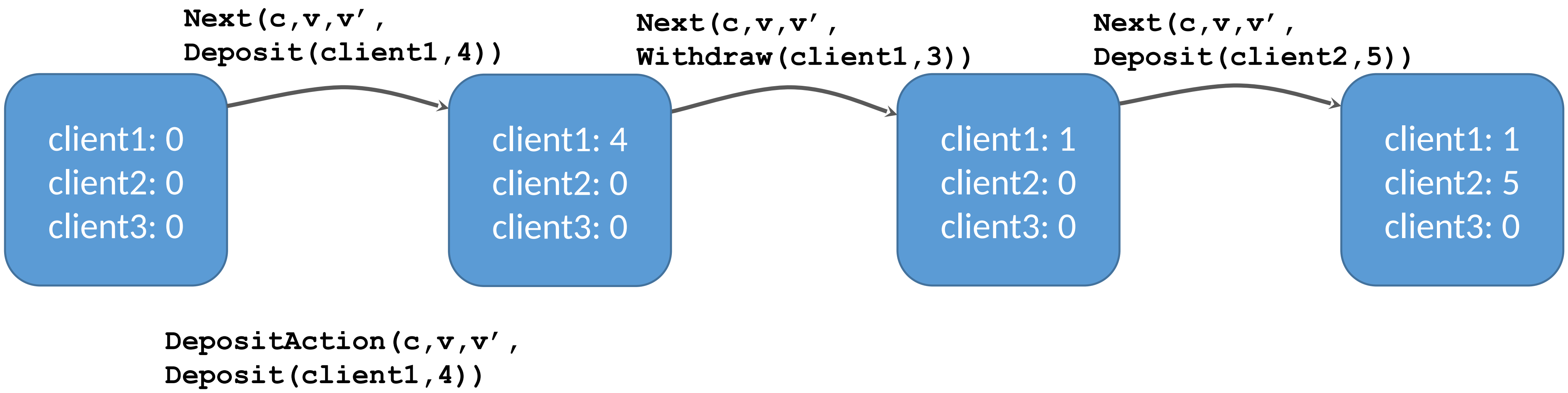
For example, the lock service would use this Event datatype:

```
datatype Event = Acquire(clientId:nat) | Release(clientId:nat) | NoOp
```

The Next() transition will now be parameterized by an Event:

```
ghost predicate Next(c: Constants, v: Variables, v': Variables, evt: Event)
```

Example: Bank spec state machine



Event-enriched protocol state machines

We will **also** be adding events to our protocol state machines

Using the exact same type as the spec state machine uses

E.g. for lock service

```
datatype Event = Acquire(clientId:nat) | Release(clientId:nat) | NoOp
```

The Next() transition of both Host and DistributedSystem will now be parameterized by an Event:

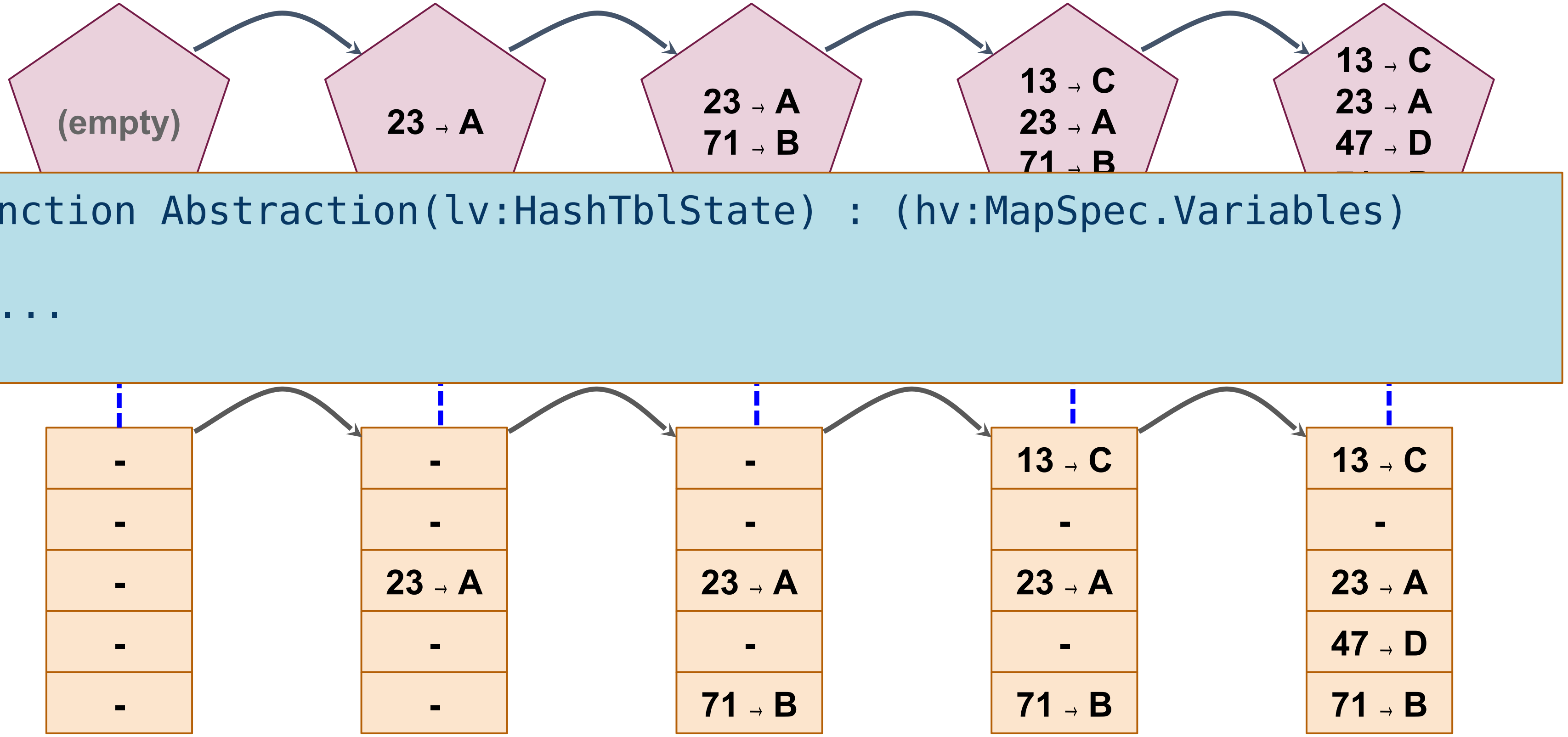
```
ghost predicate Next(c: Constants, v: Variables, v': Variables, evt: Event)
```

Event-enriched state machines

…and bound together using the Event as a binding variable

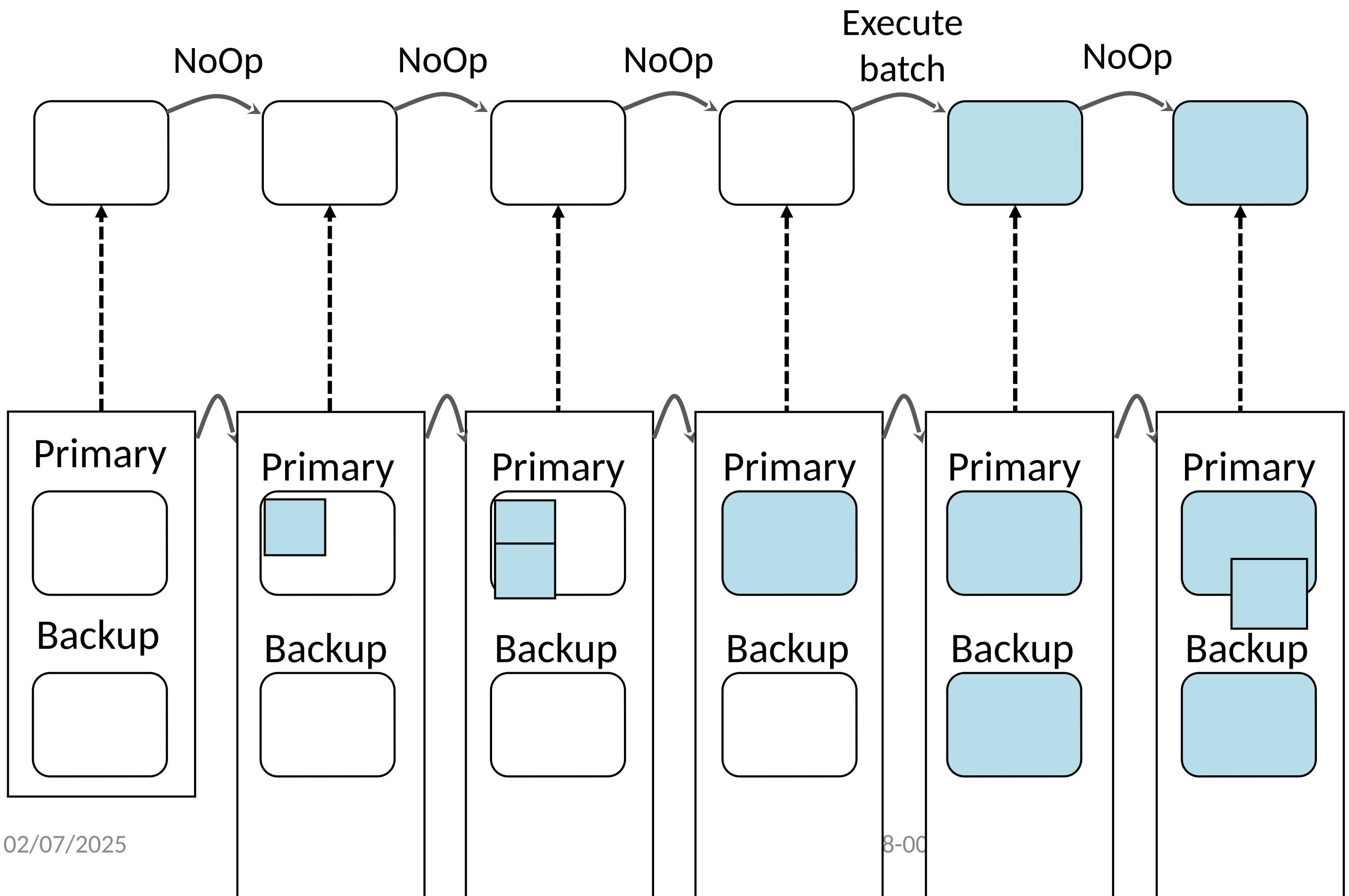
```
module DistributedSystem {
...
ghost predicate NextStep(c: Constants, v: Variables, v': Variables, evt: Event,
step: Step)
{
  // HostAction calls Host.Next with evt
  && HostAction(c, v, v', evt, step.hostid, step.msgOps)
  && Network.Next(c.network, v.network, v'.network, step.msgOps)
}

ghost predicate Next(c: Constants, v: Variables, v': Variables, evt: Event)
{
  exists step :: NextStep(c, v, v', evt, step)
}
```

The Abstraction function



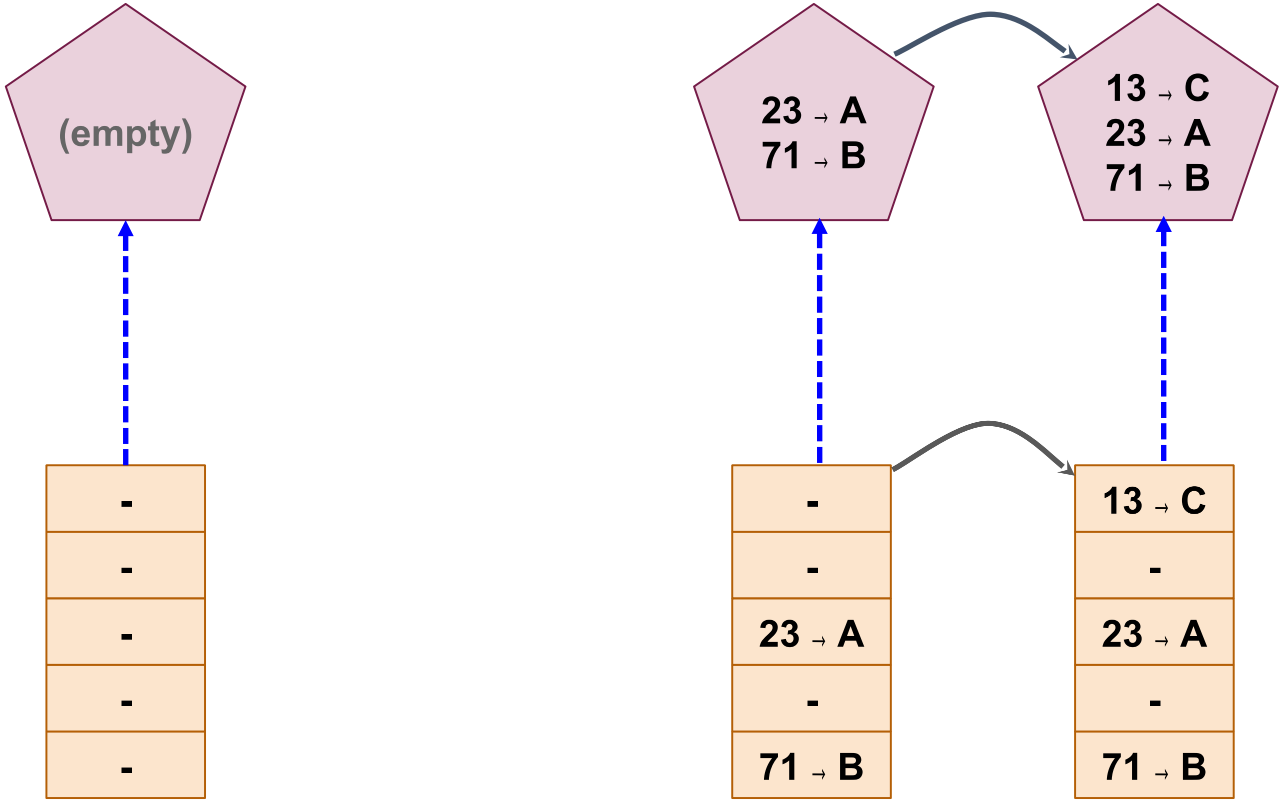

```
function Abstraction(lv:HashTblState) : (hv:MapSpec.Variables)
{
  ...
}
```

A primary-backup protocol



What is the abstraction function?

A refinement proof



A refinement proof

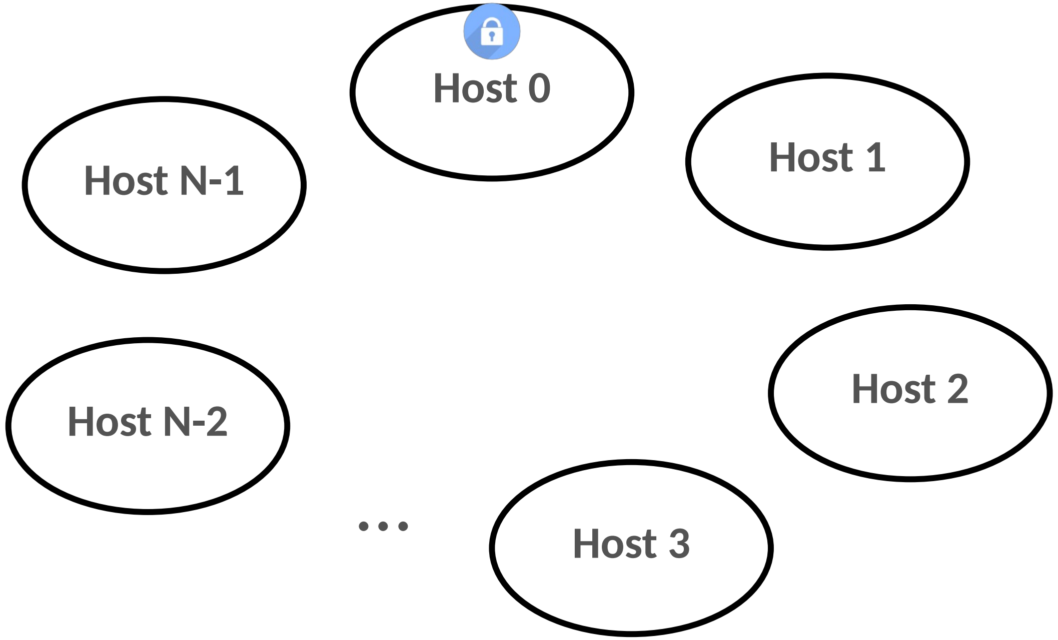
```
function Abstraction(v:Variables) : Spec.Variables
predicate Inv(v:Variables)

lemma RefinementInit(v:Variables)
    requires Init(v)

    ensures Spec.Init(Abstraction(v))  // Refinement base case

lemma RefinementNext(v:Variables, v':Variables)
    requires Next(v, v’, evt)


    ensures Spec.Next(Abstraction(v), Abstraction(v’), evt) // Refinement
inductive step
            || Abstraction(v) == Abstraction(v’) && evt == NoOp // OR stutter step
```

Project 1: Distributed lock service

Differences from centralized lock server

- **No centralized server** that coordinates who holds the lock
  - The hosts pass the lock amongst themselves
- The hosts communicate via **asynchronous messages**
  - A single state machine transition **cannot** read/update the state of two hosts

Distributed lock server



- N = numHosts, defined in network.t.dfy
- Messages are asynchronous (i.e. sending and receiving are two separate steps)

Distributed lock server

The lock is associated with a monotonically increasing epoch number

Accept an incoming message only if it has a higher epoch number than your current epoch

Distributed lock server

Safety property:

The desirable property is the same as the centralized lock server: at most one node holds the lock at any given time

Project files

Framework files
(trusted/immutable)

network.t.dfy

distributed_system.t.dfy

Host and proof files
(for you to complete)

host.v.dfy

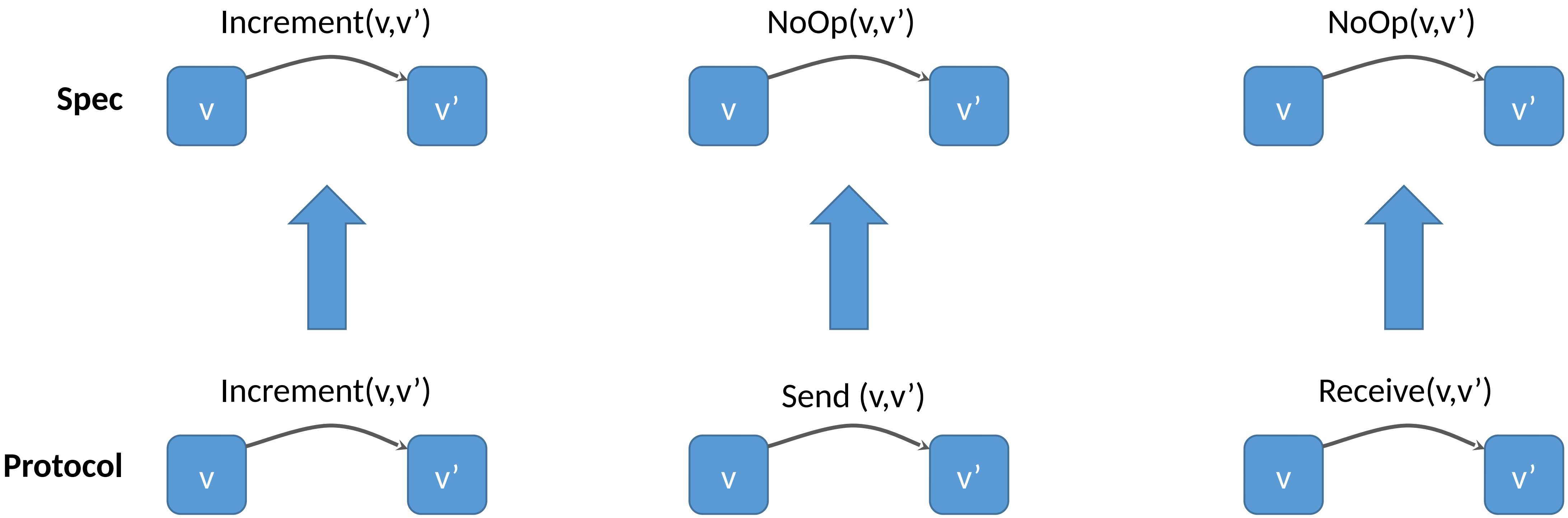
exercise01.dfy

Case study: a moving counter

- Hosts pass a counter around
- They can increment it or send it to someone else
  - Three types of protocol steps: Increment, Send, Receive
- No duplicates in the network

- Spec: a counter

Case study: a moving counter



VSCode transition