

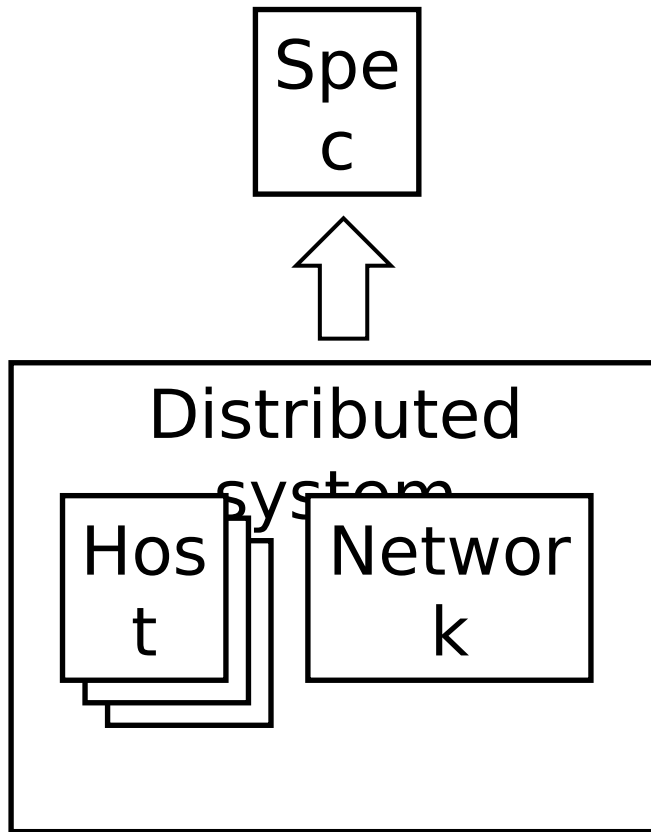
EECS498-008

Formal Verification

of Systems Software

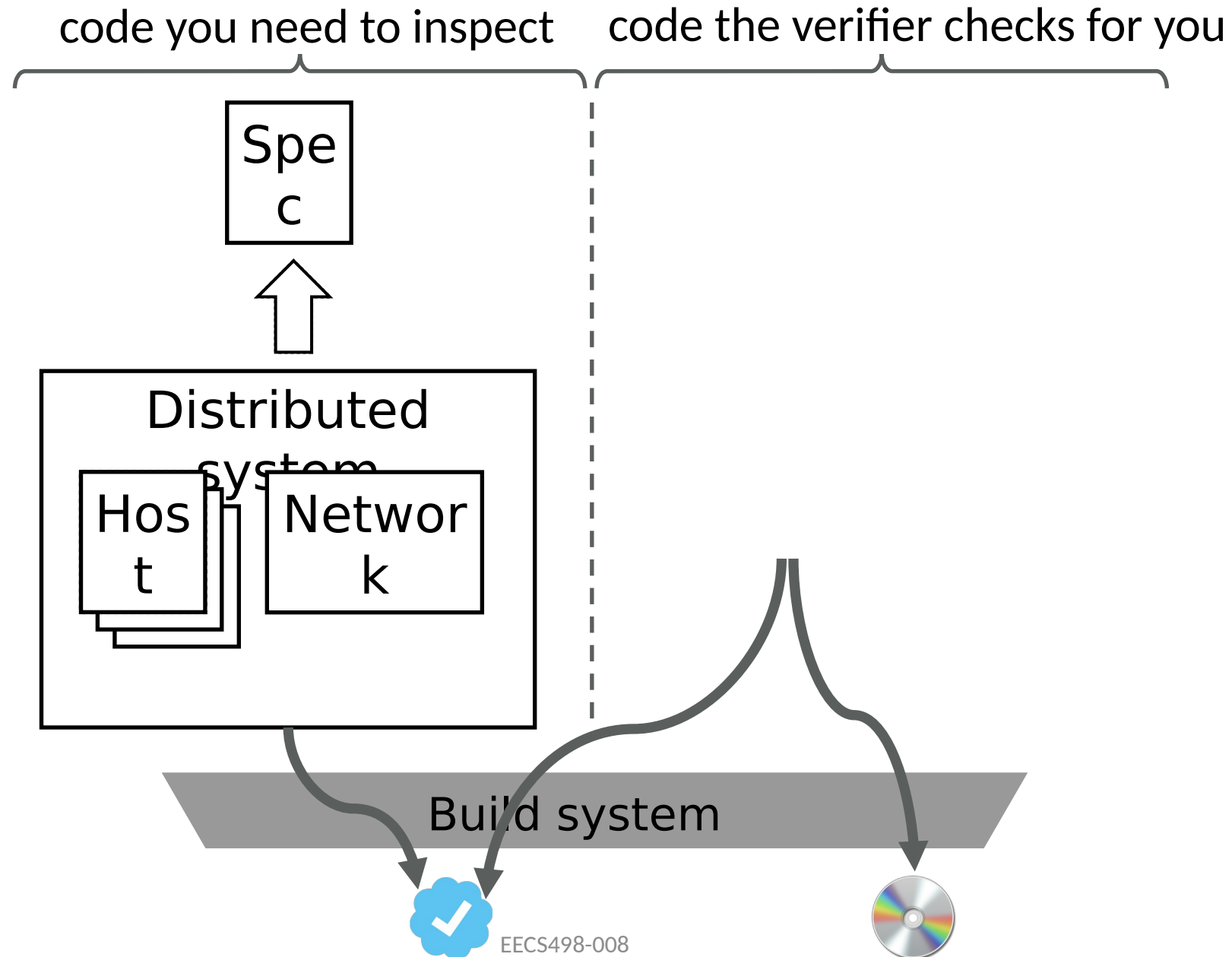
Material and slides created by
Jon Howell and Manos Kapritsos

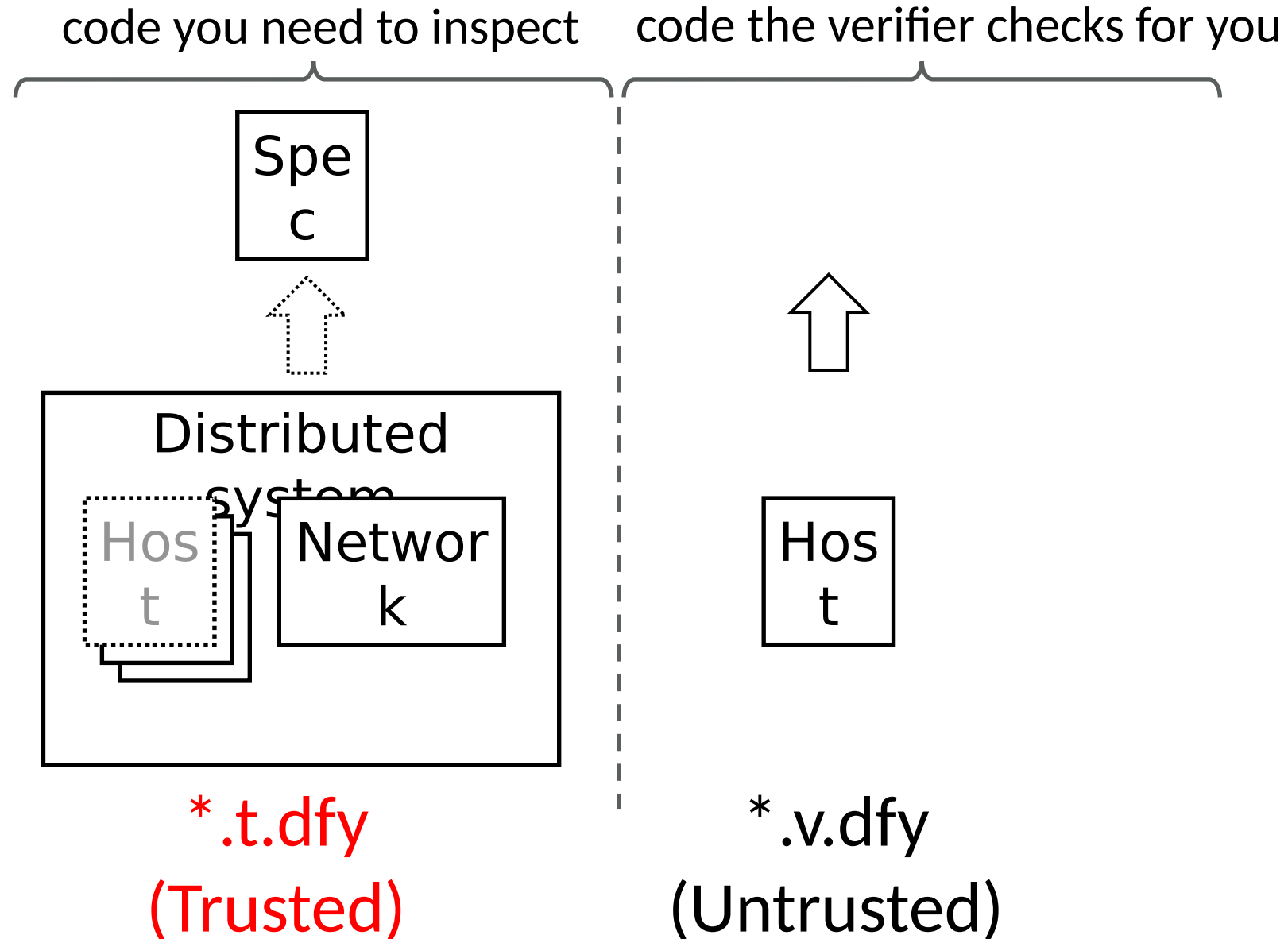
Refinement recap



```
function A(v:Variables) : Spec.Variables
predicate Inv(v:Variables) { ... }

lemma Refinement(v, v')
  ensures Init(v)  $\Rightarrow$  SpecInit(A(v)) && Inv(v)
  ensures Next(v, v') && Inv(v)
     $\Rightarrow$  ( || SpecNext(A(v), A(v')) && Inv(v')
        || A(v) == A(v')
        )
```





The verification game

- Player 1: the benign verification expert 

- Player 2: the malicious engineer 

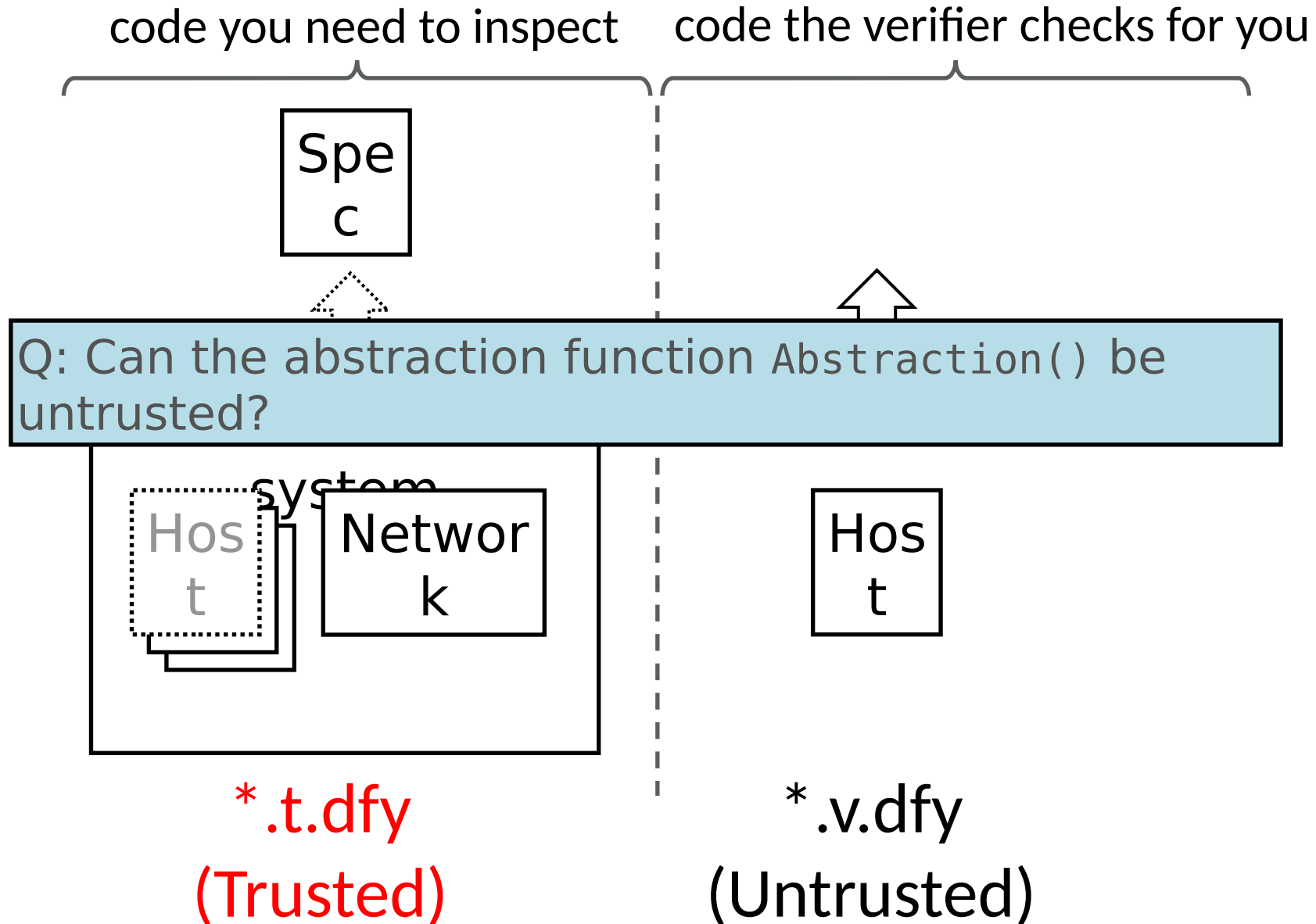


Player 1 sets up the trusted environment
(i.e. all `.t.dfy` files)

Player 2 writes the implementation and proof
(i.e. all `.v.dfy` files)



Player 1 runs the build system



What if the abstraction function pretended nothing ever happened?

Always returns the initial state

```
function Abstraction(v:Variables) :  
    Spec.Variables {  
    var a0 :| SpecInit(a0);  
    a0  
}  
  
predicate Inv(v:Variables)  
{ true }
```

...or just made up a fake story?

Returns fake
state

```
datatype Variables =  
  Variables(actualState: Stuff, fakeState:  
    HostState)  
  
function Abstraction(v:Variables) :  
  spec.Variables {  
    v.fakeState  
  }
```


Maybe someone should inspect Abstraction()...

Make it **Abstraction.t.dfy** and have an examiner examine it...

...ugh, that's a bad idea! The examiner would have to read
the entire protocol description

Application correspondence

Idea: use a trusted client-facing interface to constrain function `Abstraction()`

- Step 1: define a **trusted interface** that records requests and replies

```
module TrustedABI {  
  datatype Variables = Variables(requests:set<Input>,replies:set<Output>)  
  
  predicate AcceptRequest(v:Variables, v':Variables, request: Input)  
  { ... }  
  predicate DeliverReply(v:Variables, v':Variables, reply: Output)  
  { ... }  
  predicate ExecuteOp(c: Constants, v: Variables, v': Variables, abiOps:  
ABIOps)  
  {  
    // Type of binding variable between Host and TrustedABI.  
    // Analogous to Network.MsgOps  
    datatype ABIOps = ABIOps(request:Option<Input>,  
    reply:Option<Output>)  
  }  
}
```

Application correspondence

- Step 2: bind the transitions of this interface to the Host transitions

In DistributedSystem:

```
predicate HostNext(c: Constants, v: Variables, v':Variables,  
hostIdx:HostIdx, abiOps: TrustedABI.ABIOps) {  
    ...  
    && Host.Next(c.hosts[hostIdx], v.hosts[hostIdx], v'.hosts[hostIdx],  
abiOps)  
    && TrustedABI.ExecuteOp(c.abi, v.abi, v'.abi, abiOps)  
    ...  
}
```

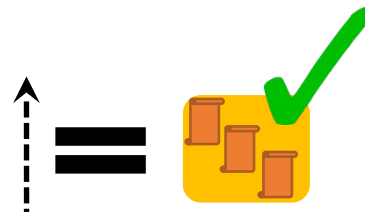
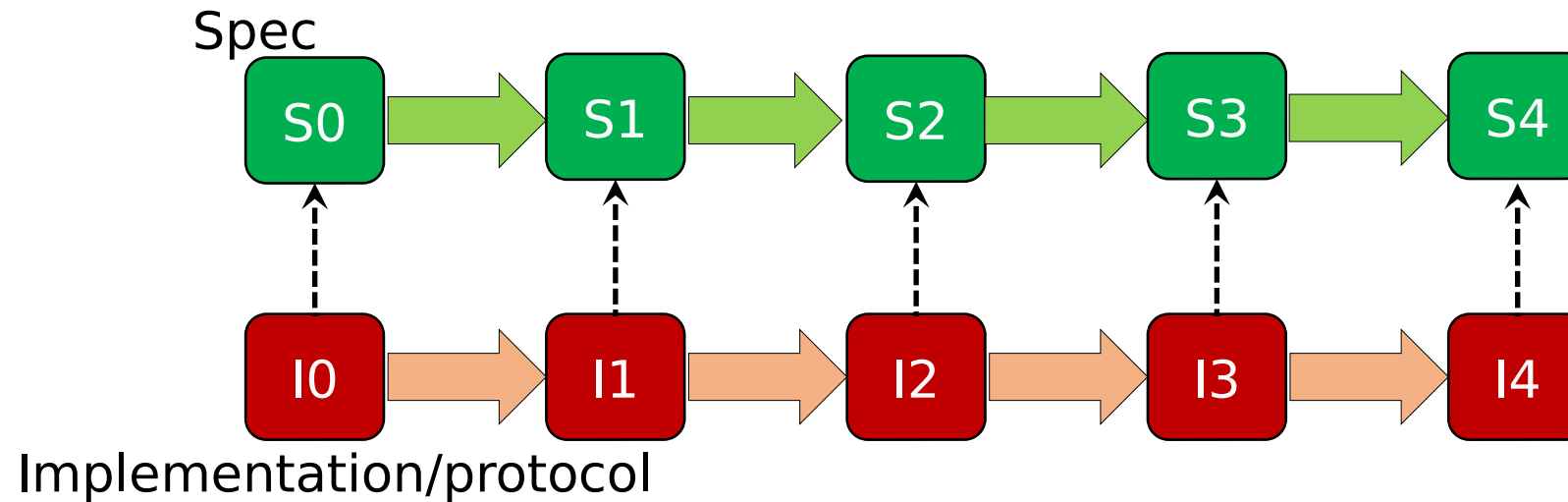
Application correspondence

- Step 3: add a refinement proof **obligation**

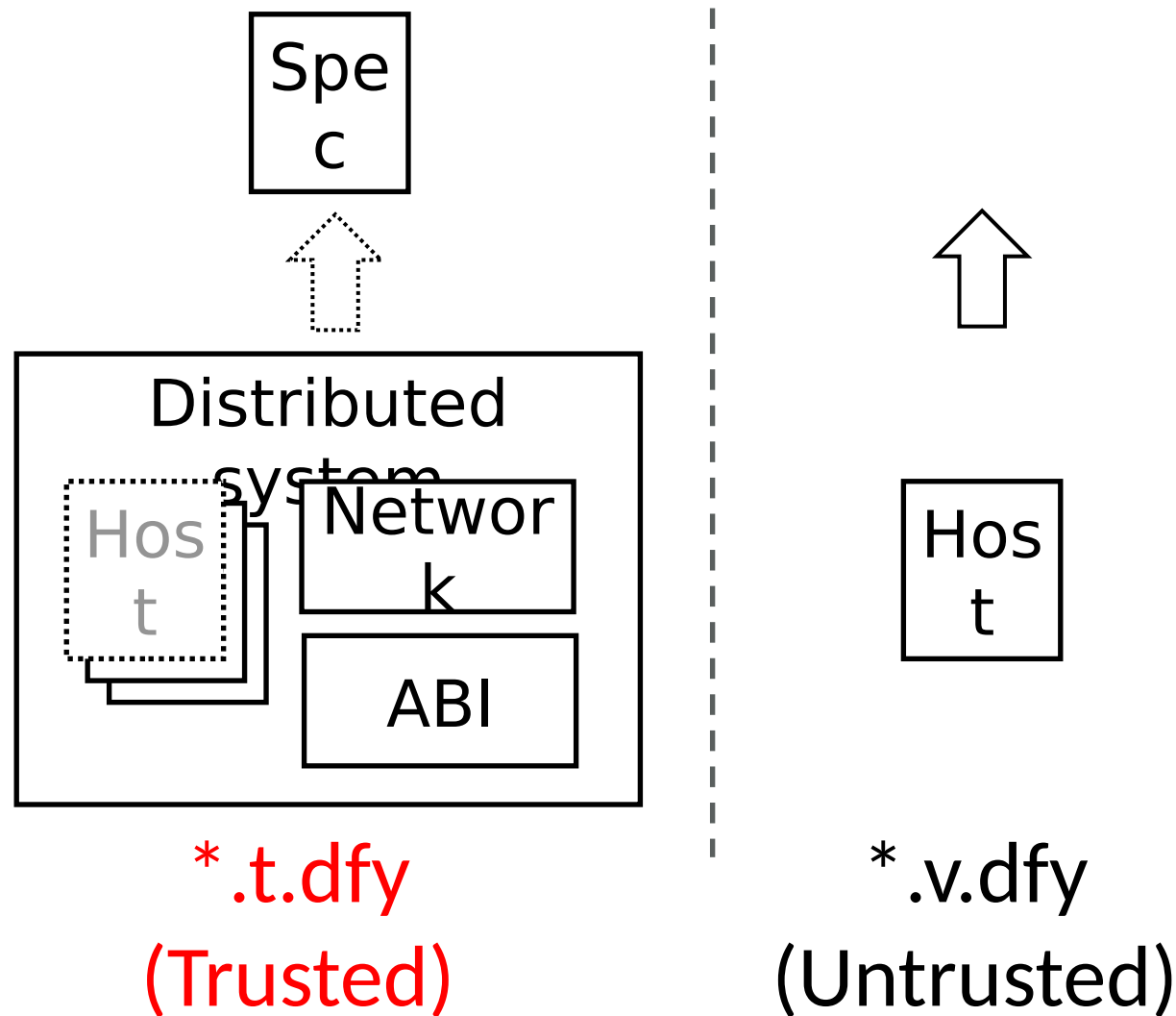
```
lemma RefinementHonorsApplicationCorrespondence(c: Constants, v: Variables)
  requires Inv(c, v)
  ensures Abstraction(c, v).requests == v.abi.requests
  ensures Abstraction(c, v).replies == v.abi.replies
{
}
```

There is no longer a reason to inspect `Abstraction()`. It is just part of the proof that constructs `Spec.Variables` as `Abstraction(Variables)`.

Application correspondence



Revisiting the big picture



Administrivia

- Monday lecture given by Jon Howell
- Also, Jon's broader verification talk, Monday 11am, BBB 3725
 - Title: **The End of Testing?**
The Promise of Verification-Driven Software Engineering
 - I strongly encourage you to attend, if you are available

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)
```

[illegible]

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...

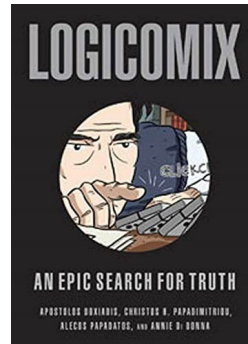
08 17

Completeness vs Soundness

- Proving a program correct is undecidable
 - i.e. it is impossible to design 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 will say “no” to some correct programs
 - 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

Triggers

- Q: Does Dafny verify this code?

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

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