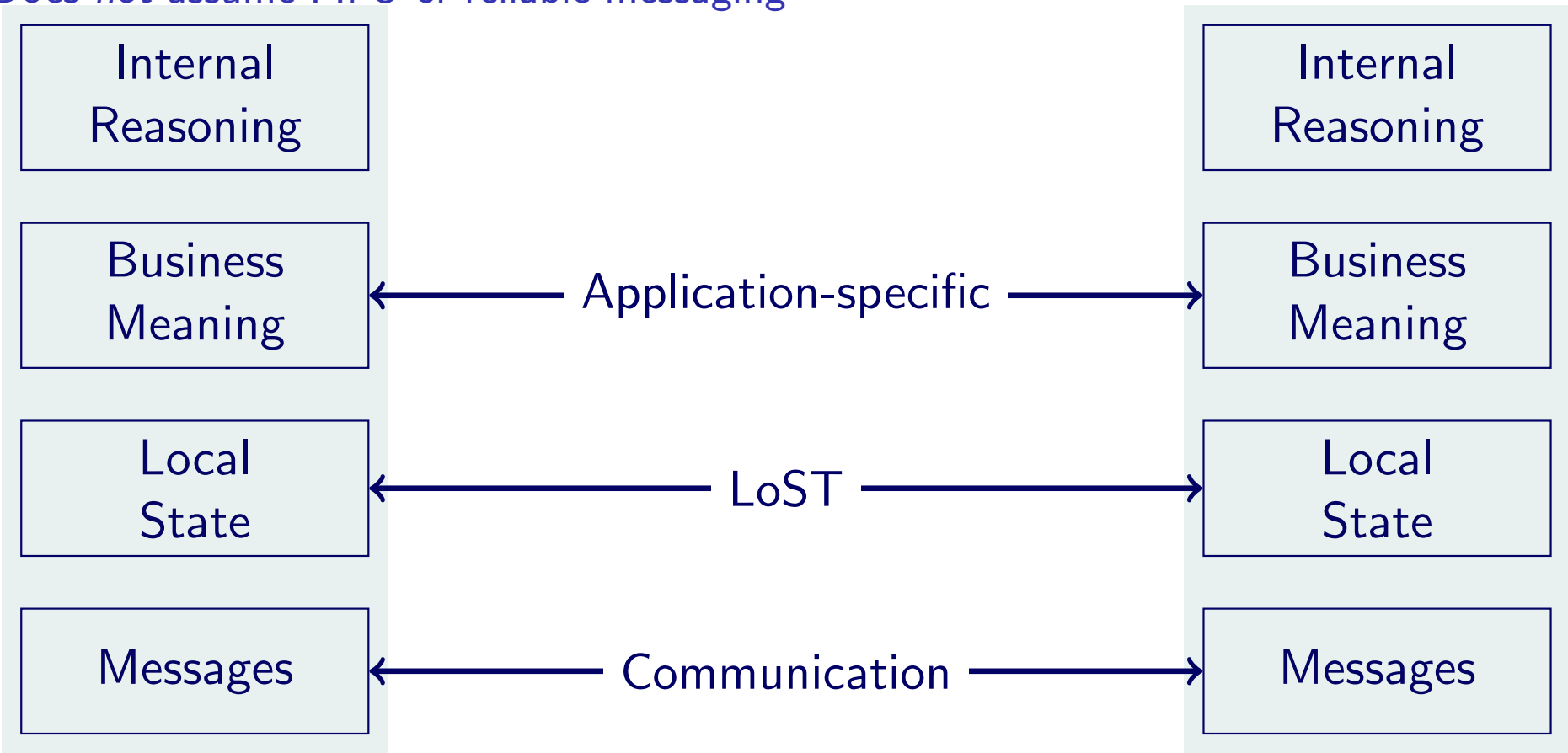
Engineering with Agent Communication

- Begin from a protocol
- Generate roles from the protocol
- For each role, implement one or more agents who realize ("flesh out") it
  - Map incoming and outgoing messages to changes in the local state
  - Implement methods to process each incoming message
  - Implement methods to send messages allowed by the protocol
- Challenge: Generating roles that ensure interoperation
  - Not trivial when a protocol involves more than two roles
  - The protocol must be such that such roles are derivable from it

Realizing BSPL via LoST (Local State Transfer)

Does *not* assume FIFO or reliable messaging



- Unique messages
- Integrity checks on incoming messages
- Consistency of local choices on outgoing messages

Implementing LoST

Think of the message logs you want

- For each role
  - For each message that it sends or receives
    - Maintain a *local* relation of the same schema as the message
- Receive and store any message provided
  - It is not a duplicate
  - Its integrity checks with respect to parameter bindings
  - Garbage collect expired sessions: requires additional annotations
- Send any unique message provided
  - Parameter bindings agree with previous bindings for the same keys for ⌜in⌝ parameters
  - No bindings for ⌜out⌝ and ⌜nil⌝ parameters exist

Enacting the *Offer* Protocol

```
Offer {
roles B, S
parameters out ID key, out item, out price

B ↦ S: rfq[out ID, out item]
S ↦ B: quote[in ID, in item, out price]
}
```

Offer (virtual)

ID	item	price
1	fig	

B:rfq

ID	item
1	fig

B:quote

ID	item	price

S:rfq

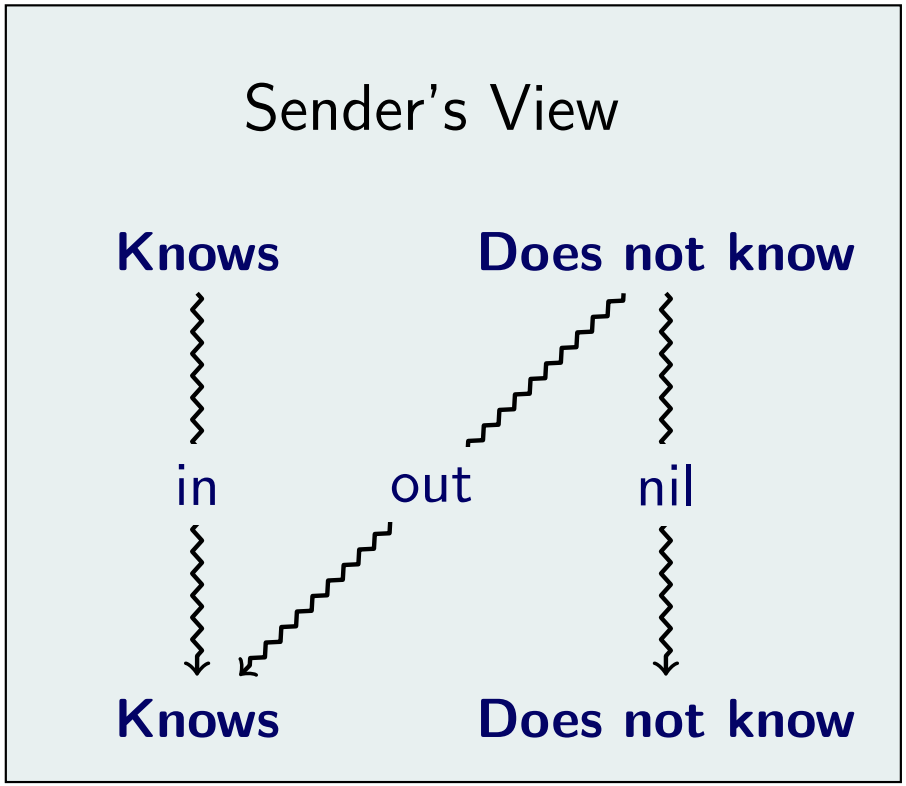
ID	item
1	fig

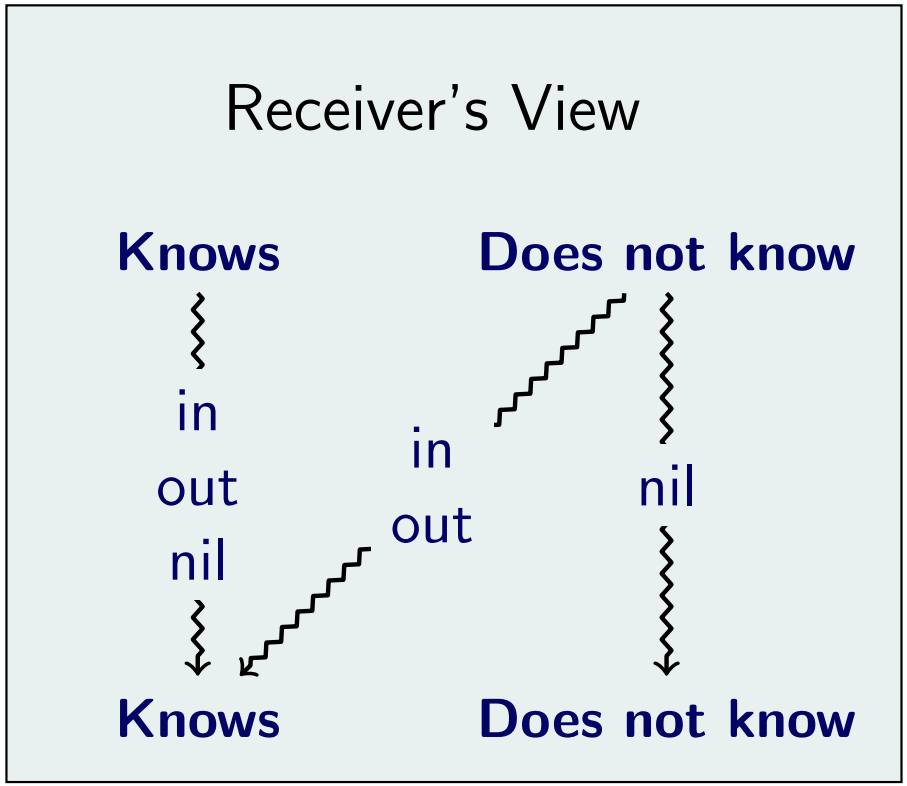
S:quote

ID	item	price

Knowledge and Viability

When is a message viable? What effect does it have on a role's local knowledge?

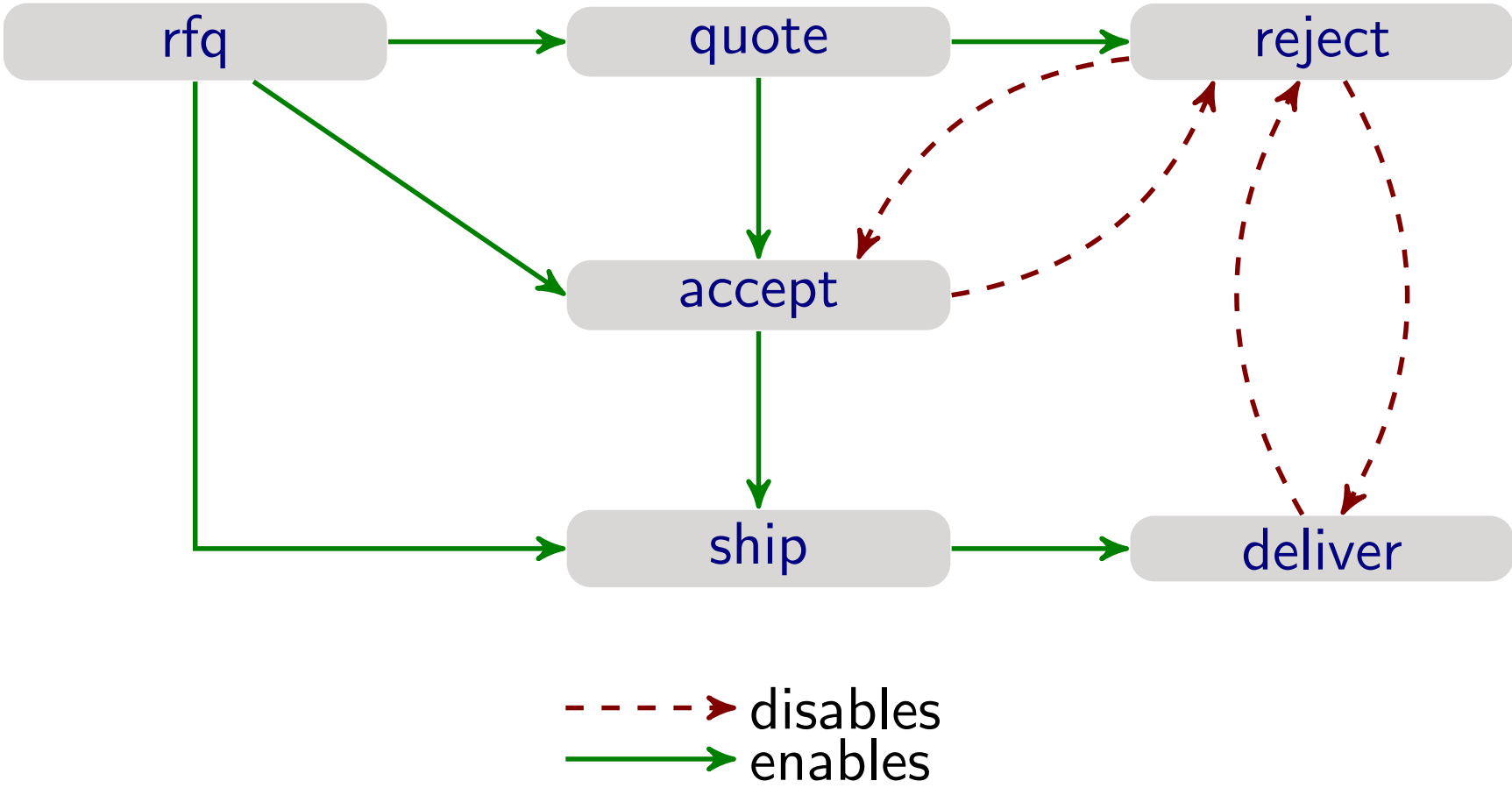




- Knowledge increases monotonically at each role
- An ⌜out⌝ parameter **creates** and transmits knowledge
- An ⌜in⌝ parameter transmits knowledge
- Repetitions through multiple paths are harmless and superfluous

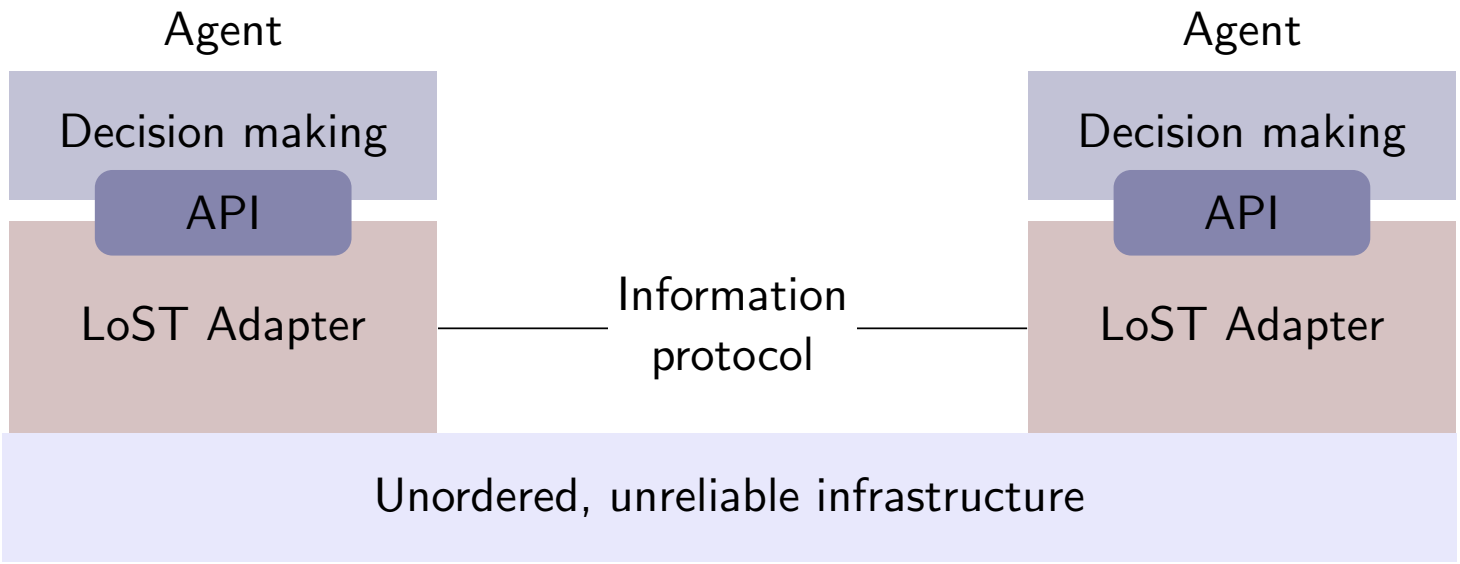
Purchase Pictorially

Informal notation; place relevant parameters as edge labels



Programming an Agent to Play a Role in an Information Protocols

Kiko (Python), Orpheus (Beliefs, Goals), Azorus (Beliefs, Goals, Commitments)

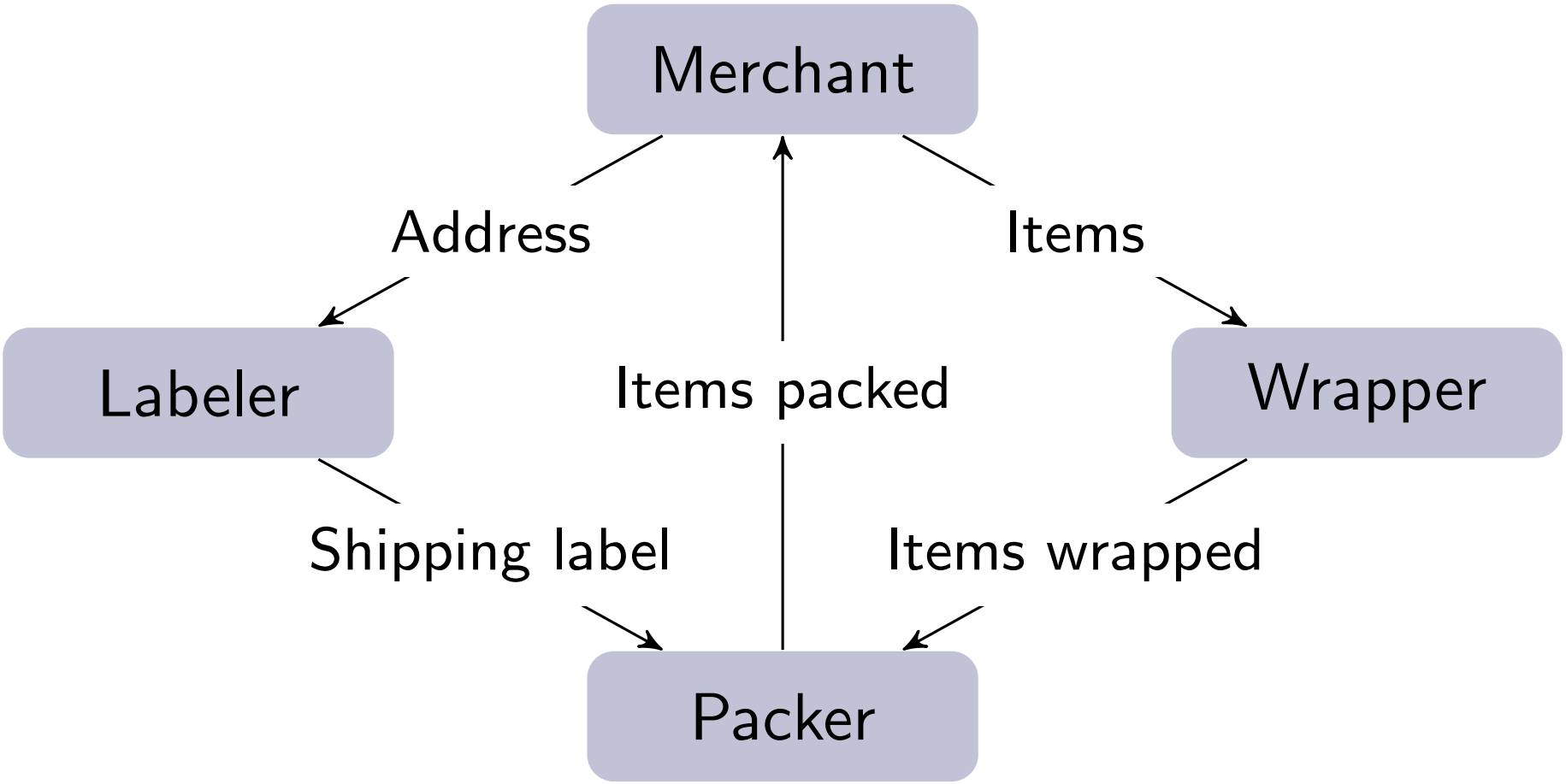


Specify information causality, not message ordering

- Constraints on message emission
- No constraints on message reception
- Messages may be retransmitted or forwarded

Logistics Scenario

Handling of Customer's Purchase Orders (POs). Several items in a PO that may be wrapped and packed independently to create a shipment



The *Logistics* Protocol

Showing just the messages for brevity

```
Merchant ↦ Labeler: RequestLabel[out oID key,
  out address]

Merchant ↦ Wrapper: RequestWrapping[in oID key,
  out iID key, out item]

Wrapper ↦ Packer: Wrapped[in oID key, in iID key,
  in item, out wrapping]

Labeler ↦ Packer: Labeled[in oID key, in address,
  out label]

Packer ↦ Merchant: Packed[in oID key, in iID key,
  in item, in wrapping, in label, out status]
```

Adapter Computes What Messages are Enabled to Send

Consider a particular snapshot of the Packer's local state

```
Packer ↦ Merchant: Packed[in oID key, in iID key,
  in item, in wrapping, in label, out status]
```

P:Labeled

oID	address	label
1	UK	1234
2	US	abcd

P:Wrapped

oID	iID	item	wrapping
2	a2	jam	silk

The *Packed* message is enabled

- All its ⌜in⌝ parameters are bound based on the local state
- All its ⌜out⌝ parameters (just one here) are to be bound by the agent via its reasoning
- *Packed*(oID: 2, iID: a2, item: `jam`, wrapping: `silk`, label: `abcd`, status:???)

Enablement-Based Programming Model for Agents

Adapter invokes message enablement handlers that encode decision making

```
enabled(Packed p)
p.status = "Better!"
MC.send(p)
```

Enablement

- Agent programmer writes one method for each message the adopted role may send
- The adapter maintains the local state for the agent
- Whenever a message is enabled, the adapter calls the corresponding method, specifying the bindings of all ⌜in⌝ parameters
- The method figures out
  - Whether to send an instance of the message
  - What bindings to use for its ⌜out⌝ parameters

Safety: *Purchase Unsafe*

Remove conflict between *accept* and *reject*

```
Purchase Unsafe {
roles B, S, Shipper
parameters out ID key, out item, out price, out outcome
private address, resp

B ↦ S: rfq[out ID, out item]
S ↦ B: quote[in ID, in item, out price]
B ↦ S: accept[in ID, in item, in price, out address]
B ↦ S: reject[in ID, in item, in price, out outcome]

S ↦ Shipper: ship[in ID, in item, in address]
Shipper ↦ B: deliver[in ID, in item, in address,
  out outcome]
}
```

- B can send both *accept* and *reject*
- Thus outcome can be bound twice in the same enactment

Liveness: *Purchase No Ship*

Omit *ship*

```
Purchase Minus Ship {
roles B, S, Shipper
parameters out ID key, out item, out price, out outcome
private address, resp

B ↦ S: rfq[out ID, out item]
S ↦ B: quote[in ID, in item, out price]
B ↦ S: accept[in ID, in item, in price, out address,
  out resp]
B ↦ S: reject[in ID, in item, in price, out outcome,
  out resp]

Shipper ↦ B: deliver[in ID, in item, in address,
  out outcome]
}
```

- If B sends *reject*, the enactment completes
- If B sends *accept*, the enactment deadlocks

Exercise: *Abruptly Cancel*

Solution added

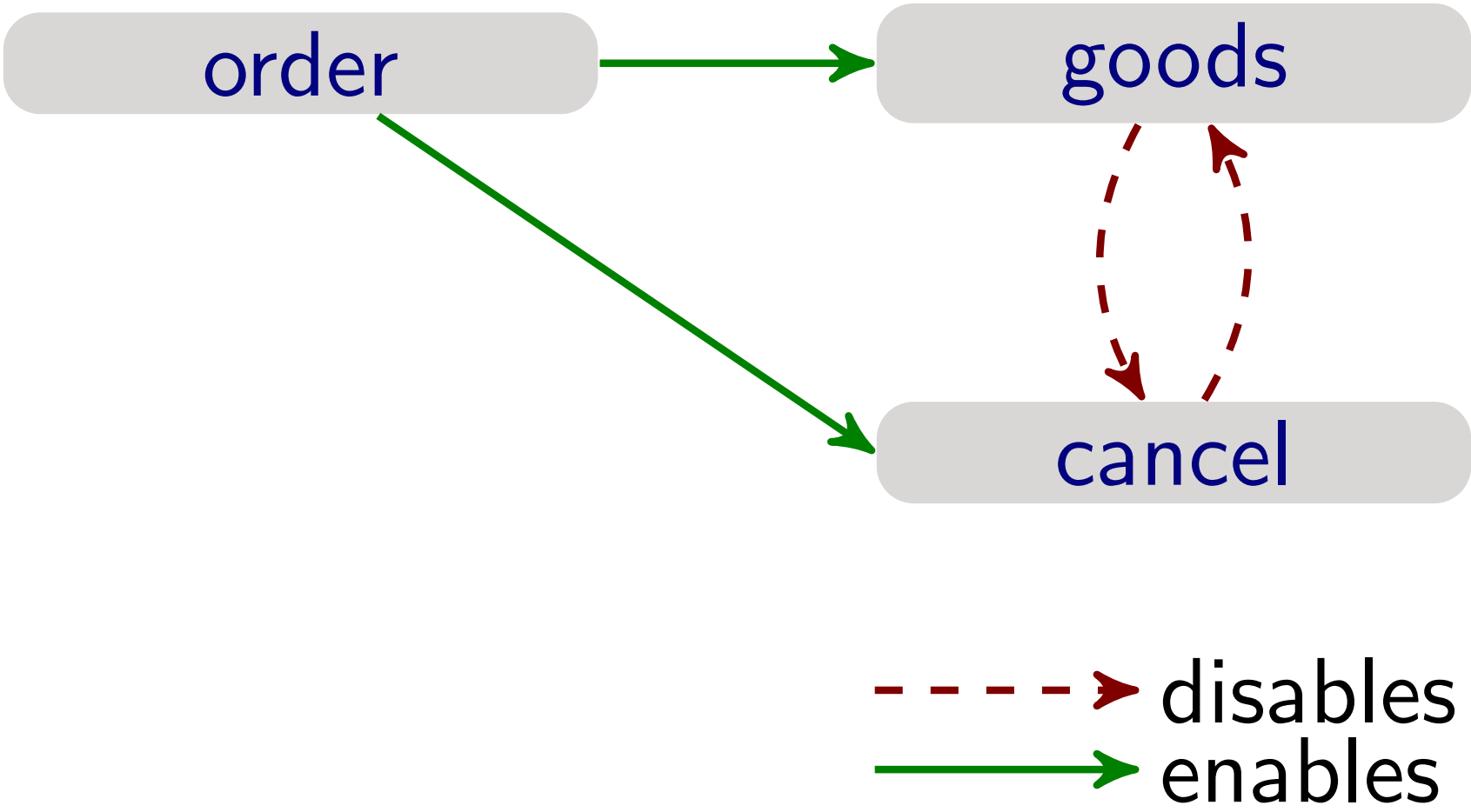
```
Abruptly Cancel {
roles B, S
parameters out ID key, out item, out outcome

B ↦ S: order [out ID, out item]
B ↦ S: cancel [in ID, in item, out outcome]
S ↦ B: goods [in ID, in item, out outcome]
}
```

- Is this protocol safe? **No**
- Is this protocol live? **Yes**

Abrupt Cancel Unsafe

Nonlocal choice



Exercise: *Abruptly Cancel* Modified (with ⌜nil⌝)

Solution added

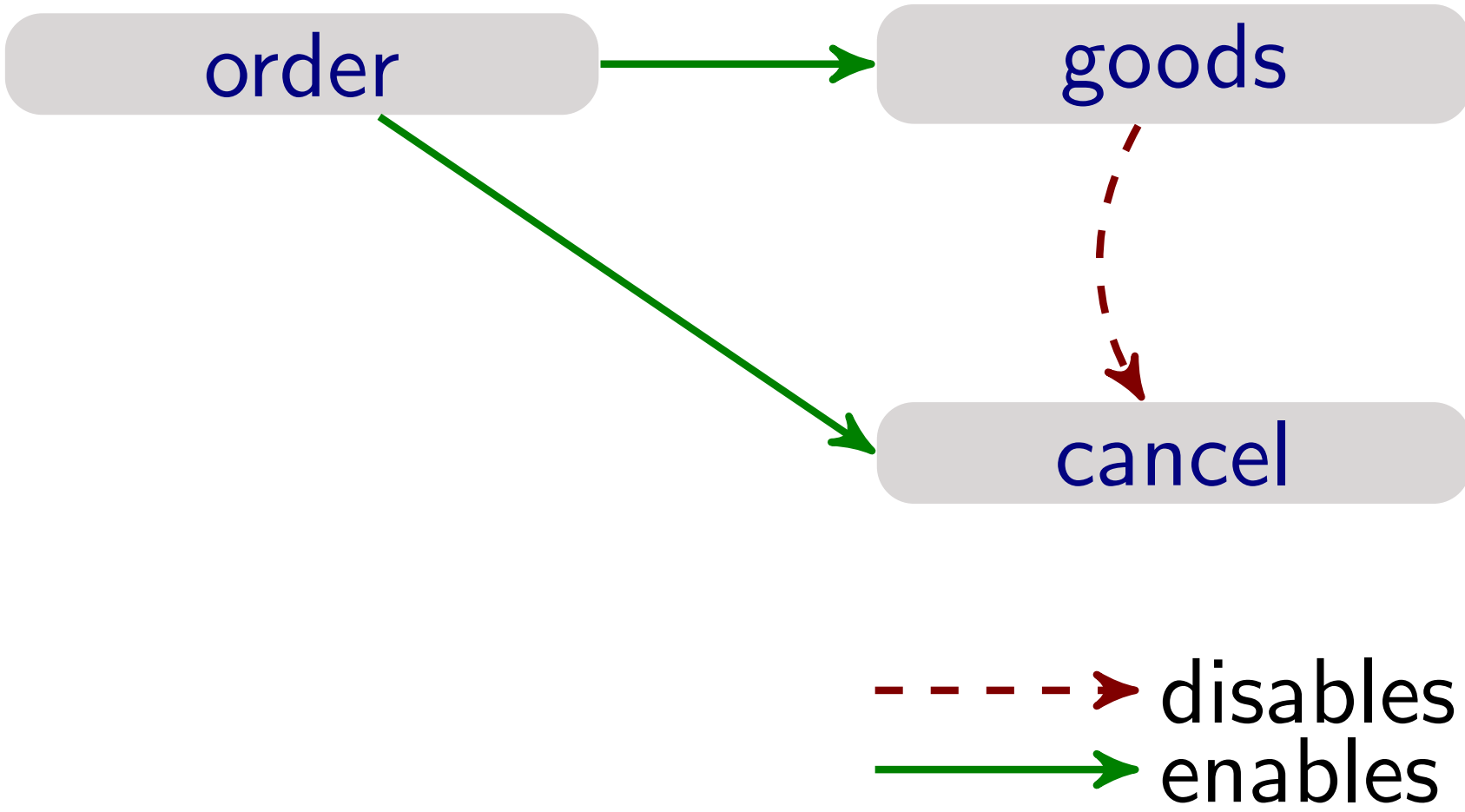
```
Abruptly Cancel {
roles B, S
parameters out ID key, out item, out outcome

B ↦ S: order [out ID, out item]
B ↦ S: cancel [in ID, in item, nil outcome]
S ↦ B: goods [in ID, in item, out outcome]
}
```

- Is this protocol safe? **Yes**
- Is this protocol live? **Yes**
  - But it lacks business realism because the SELLER may send *goods* even after receiving *cancel*

Abrupt Cancel Safe

Removed nonlocal choice



Validation in Systems Engineering

Solving the right problem

Validation Challenges

Does the protocol capture precisely what the stakeholders wanted?

- Verification $\not\Rightarrow$ Validation
  - See *Abruptly Cancel* Modified above
- Inherently an iterative process because stakeholders don't know what they want
- Made harder by decentralization

Verification in Systems Engineering

Solving the problem right

Verification Challenges

- Impossible for a complete decentralized system
- Only possible under assumptions of how other parties behave
- Protocols capture *some* such assumptions formally

Compositionality

Each part is correct $\Rightarrow$ Whole is correct

Compositionality

Does the protocol guarantee that if each party is correct, the multiagent system is correct?

Correctness of Roles Given a Protocol

- Is the design objective of each role obvious?
- Does the role include some illegal enactments?
- Does the role preclude some legal enactments?

Correctness of Protocol Independent of Agents

- Liveness
- Safety
- Domain-specific properties