The Blindingly Simple Protocol Language: Advanced Notions and Programming Model

Amit K. Chopra

Lancaster University

Polymorphism

Same message schema, but different adornments

```
Flexible-Offer {
 role B, S
 parameter in ID key, out item, out price, out qID

 B ↦ S: rfq[in ID, out item, nil price]
 B ↦ S: rfq[in ID, out item, out price]

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

- B has priority on generating price, but if it chooses not to (by sending *rfq* without price), then S can generate it

Safety: *Purchase Unsafe*

A protocol is safe iff no parameter can be bound twice in an enactment

```
Purchase Unsafe {
 role B, S, Shipper
 parameter 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]
}
```

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

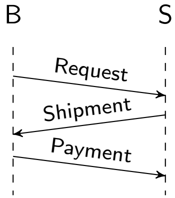
Liveness: *Purchase No Ship* (Omit *ship*)

A protocol is live iff any enactment (including the "empty" enactment) may progress to completion

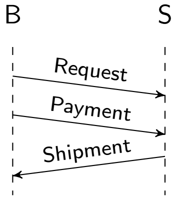
```
Purchase No Ship {
 role B, S, Shipper
 parameter out ID key, out item, out price, out outcome
 private address, resp

 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

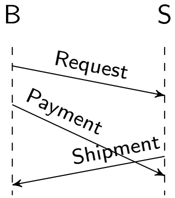
Concurrency: Can All These Enactments Be Supported?



(a) Shipment first



(b) Payment first

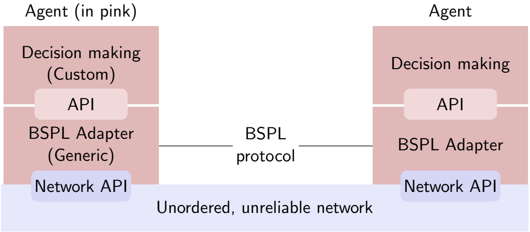


(c) Concurrent

Flexible Purchase: No Deadlock!

```
Flexible Purchase {
 role B, S
 parameter out ID key, out item, out shipped, out paid

 B ↦ S: Request[out ID, out item]
 S ↦ B: Shipment[in ID, in item, out shipped]
 B ↦ S: Payment[in ID, in item, out paid]
}
```

BSPL Adapter-Based Programming Model



Realizing a BSPL adapter ("interpreter" for BSPL)

- For each role (agent playing the 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
- 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

Remark on Control versus Information Flow

- Control flow
  - Natural within a single computational thread
  - Exemplified by conditional branching
  - Presumes master-slave relationship across threads
  - Impossible between mutually autonomous parties because neither controls the other
  - May sound appropriate, but only because of long habit
- Information flow
  - Natural across computational threads
  - Explicitly tied to causality

Information Centrism

Characterize each interaction purely in terms of information

- Explicit causality
  - Flow of information coincides with flow of causality
  - No hidden coordination
- Keys
  - Uniqueness of enactments
  - Integrity (parameter has at most one value in any enactment)
  - Basis for completion
- Immutability
  - Durability
  - Robustness: insensitivity to reordering by infrastructure

BSPL References

- Required Reading: See paper on the Blindingly Simple Protocol Language, available in the section for lecture 9 on Moodle.
- Required Reading: See paper on verifying information protocols, available in the section for lecture 10 on Moodle. Make sure you understand the concept of safety and liveness; you can omit the part of verifying them.
- Additional Reading: See paper on BSPL adapter (LoST), available in the section for lecture 10 on Moodle.

Decision: Abstraction for Agent Programming

Select a *set* of *enabled* messages to flesh out and emit

```
@adapter.schedule_decision("1s")
def decision_gifts(enabled):

    accepts = maximizeItems(budget,
        enabled.messages(Accept))

    for a in accepts
        a.bind(addr="Lyngby", dec=True)


    posRejects = enabled.message(Reject)
    rejects   = posRejects ▷ accepts

    for r in rejects
        r.bind(dec=True, status="done")

    emit(accepts ∪ rejects)
```

Complex Correlation; Abstraction over Message Order

```
W ↦ P: Wrapped[in oID key, in iID key, in item,
                out wrapping]

L ↦ P: Labeled[in oID key, in address, out label]

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

```
def decision_packing(enabled):

    packeds = enabled.messages(Packed)

    for p in packeds
        p.bind(status=True)

    emit(packeds)
```

Protocol-Based Application Design is the Future

- Erlang and Google's Go are languages that already promote messaging-based coordination, but would benefit from protocols
- Current network technologies ill-suited to modern decentralized applications
  - IoT,
  - Microservices-based
  - Business contract
- What we've covered in the last three lectures is the foundation!