Hands-on Interaction-oriented Programming

Amit K. Chopra[1] Matteo Baldoni[2] Samuel H. Christie V[3]
Munindar P. Singh[3]

[1]Lancaster University

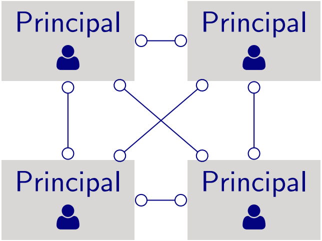
[2]University of Torino [3]North Carolina State University

AAMAS 2025, Detroit, Tutorial T8

Outline

Sociotechnical Systems

Long-lived engagements between autonomous principals

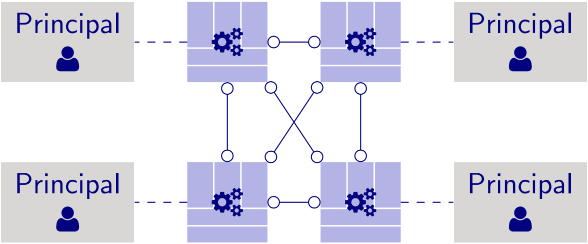


- In Ebusiness, health, finance,. . .
- Conceptually decentralized

Challenge

Realize an STS as a decentralized, loosely-coupled system that lets the principals interact with maximal flexibility?

Multiagent Systems: Agents Help Principals Exercise Autonomy

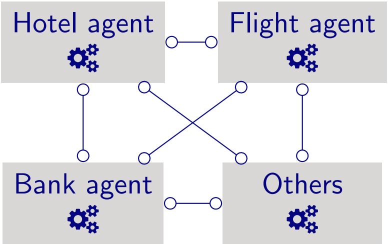


Agents

- Are heterogeneous in construction
- Encode decision making of their respective principals
- Interact based on *agreements* and asynchronous *protocols*

Agentic AI: Multiagent Paradigm

Flexible, Generative AI-powered agents that make real world decisions



Inflexible coordination via workflows defeats flexibility

- We abandoned workflows in the 90s
- Started working on *interaction meaning*

Interaction-Oriented Programming (IOP)

Empower stakeholders and programmers

Method

- Model a multiagent system in terms of interactions
- Compose and verify models
- Implement agents independently on the basis of models

High-level abstractions that

- Reflect stakeholder intuitions and
- Let programmers focus on the business logic

Outline

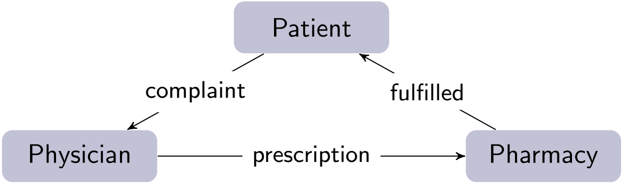
Communication Protocols

A protocol defines how the agents ought to communicate with one another
A protocol is a model of a decentralized application!

- What are the main requirements for protocol specifications?
- How can we specify a communication protocol?
  - Roles (abstracting over agents)
  - Message schemas, i.e., allowed content
  - Message emission and reception, point-to-point or multicast, between specified roles
  - Constraints on message occurrence
  - Constraints on message ordering
- Agents participate in a protocol by playing a role in it
- How can we develop agents suitable for a role?

Protocol for a Healthcare Application

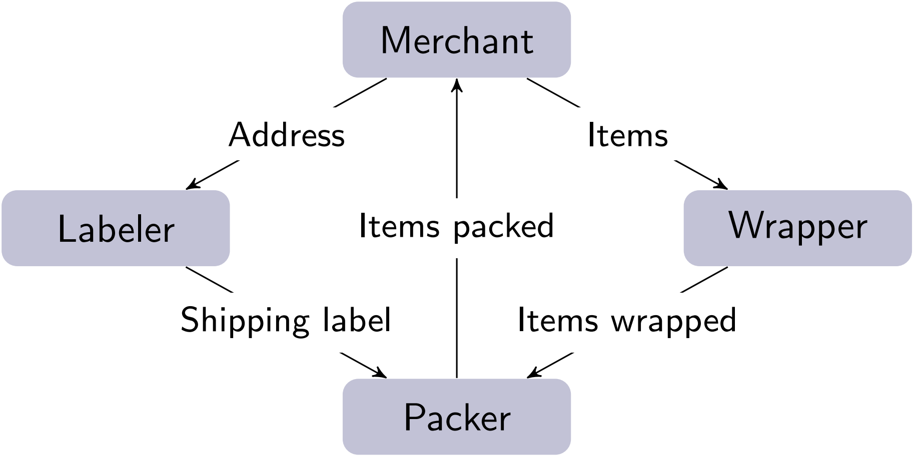
Patient sends a Complaint to Physician, who sends a Prescription to Pharmacy, who sends Fulfill to Patient



- Autonomy means no one needs to send any message!
- Three parties, not client server
- Healthcare standards: Health Level 7 (HL7), Integrate the Healthcare Enterprise (IHE)
  - Informally described interactions: difficult to implement correctly

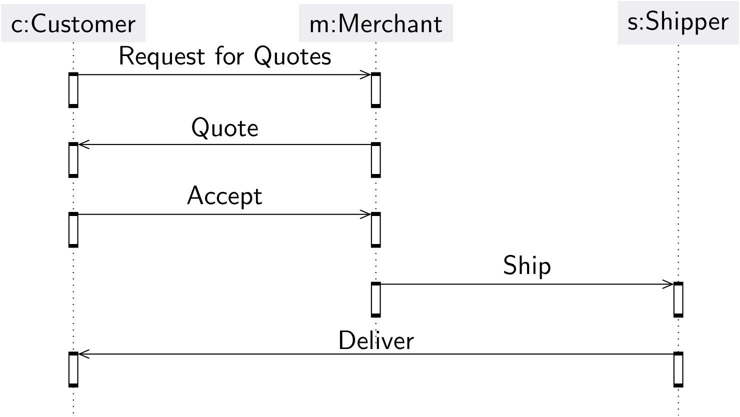
Protocol for Purchase Order (PO) Fulfillment Application

Several items in a PO that may be wrapped and packed independently to create a shipment



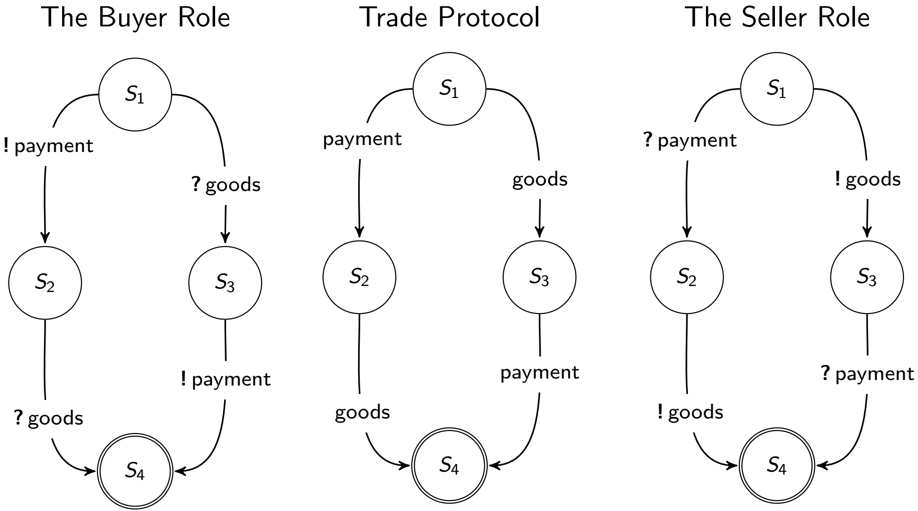
A Purchase Protocol (Just the Happy Path)

Specified as a UML interaction diagram
Unhappy path (featuring Reject) would be in another UML diagram



Protocols and Roles as State Machines

Protocol: shared view; roles: each local view
! and ? mean send and receive, respectively



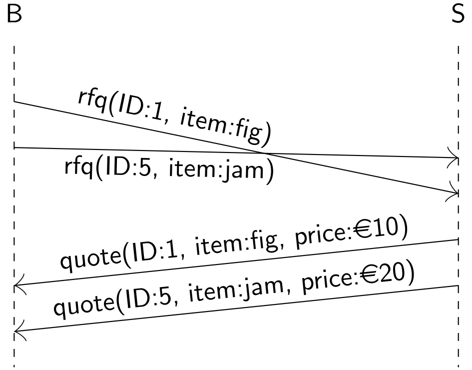
Protocols Promote Interoperation, Autonomy, Heterogeneity

- Autonomy by
  - Minimally constraining agents' decision making and interactions
- Interoperation by specifying
  - Schemas of messages exchanged
  - Meanings of messages, which determine the *state* of the interaction
  - Correct behaviors
- Heterogeneity by
  - Providing the standard to which agents are implemented
  - Defining the extent of heterogeneity: the agents can be heterogeneous with regard to everything else
- All of the above contribute to loose coupling!

Challenge: Information Integrity in Interactions

Interactions must compute consistent information objects

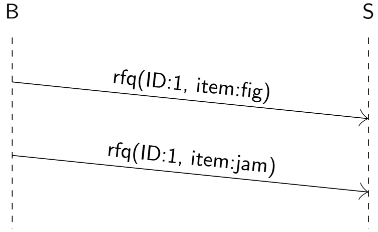
- An information object specification: [ID **key**, item, price]



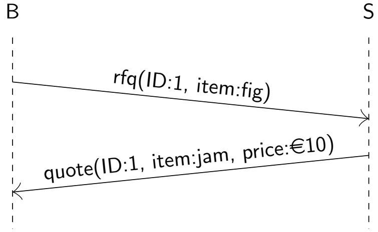
Integrity Violations

Can Be Avoided By Local Checks

- Object: [ID **key**, item, price]



(a) B messes up.

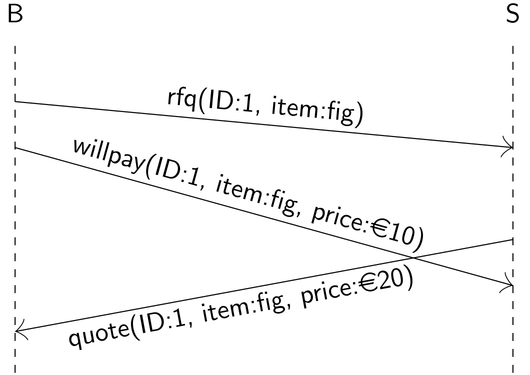


(b) S messes up.

Integrity Violation: Race Condition

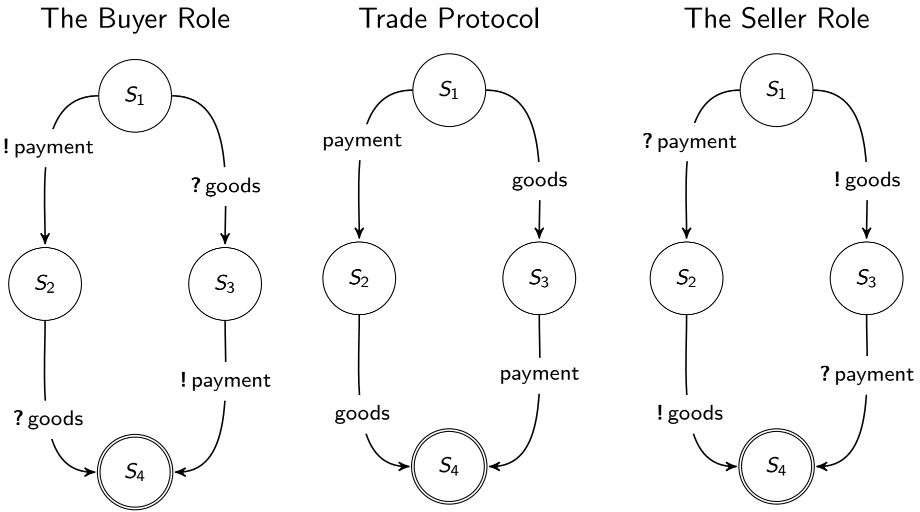
Cannot be Avoided By Local Checks; Requires Verification

- Object: [ID **key**, item, price]



Challenge: Interactions Must Not Deadlock

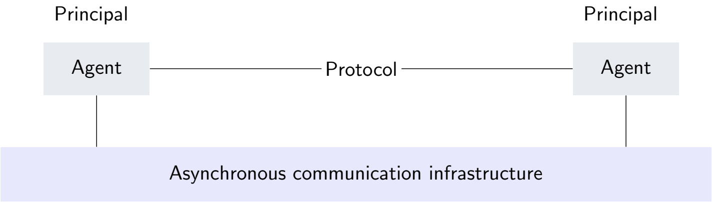
There must always be a path that agents can take that leads to a final state



- Deadlock: if B chooses **?** goods and S chooses **?** payment

Challenge: Asynchronous Communication

Practical; Offered by the Internet



- Senders and receivers do not have to rendezvous (synchronize) to perform a communication
  - Sender puts a message into the communication infrastructure regardless of the state of the receiver
  - Receiver receives the message whenever the infrastructure delivers it
- Decouples senders and receivers
- Conducive to autonomy

Properties of the Communication Infrastructure

How can we achieve each property?

Noncreative: Must only sent messages be received?
- Will an infrastructure create messages?

Reliable: Must a message that is sent be received?
- Will an infrastructure drop messages?

Ordered: Must the messages from a sender to a receiver be delivered in the order in which they were sent?
- Will the infrastructure deliver messages in any order?

Global: Must the messages from different senders to the same receiver be received in the order in which they were sent?
- Called "causal" ordering in the literature but that term refers to potential causality

Challenge: Unordered, Asynchronous Communication

The Gold Standard!

- Today: Commonplace to rely on communication infrastructures that provide first-in first-out (FIFO) delivery
  - E.g., as provided by TCP and message queues
- But ordered delivery has drawbacks
  - Hidden synchronization
  - An additional assumption for the multiagent system to work correctly
  - Couples the agents through the infrastructure
- Challenge: Coordinate decentralized computation without assuming ordered delivery infrastructure

Challenge: Engineering with Agent Communication

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

Challenge: Modeling Application Meaning

Meaning lies in the application domain

- Offers, Shipments, Payments are domain objects that have meaning to users and on the basis of which they perform their decision making
  - Offer is a domain object with a unique identifier and an associated item and price
  - Each Offer sets up a commitment from Seller to Buyer that if Payment is made within 3 days, then Shipment will be made within 5 days.
- Challenge: Protocols must enable capturing application meaning
  - Communications compute (create and change state of) domain objects, which capture the state of the application!

Inadequate: Control Flow-Based Approaches For Modeling Protocols

UML interaction diagrams, state machines, ...

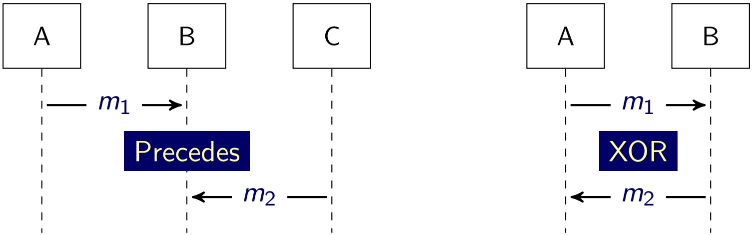
- Cannot model application meaning, which is necessarily grounded in information!
- Cannot meet the foregoing challenges
- Need an information-based approach for modeling protocols

Exercises: Specifying Protocols

Outline

Traditional Specifications: Procedural

Low-level, over-specified protocols, easily wrong



- Traditional approaches
  - Emphasize arbitrary ordering and occurrence constraints
  - Then work hard to deal with those constraints
- Our philosophy: The Zen of Distributed Computing
  - Necessary ordering constraints fall out from *causality*
  - Necessary occurrence constraints fall out from *integrity*
  - Unnecessary constraints: simply *ignore* such

Properties of Participants

- Autonomy
- Myopia
  - All choices must be local
  - Correctness must not rely on future interactions
- Heterogeneity: local $\neq$ internal
  - Local state (projection of global state, which is stored nowhere)
    - Public or observable
    - Typically, must be revealed for correctness
  - Internal state
    - Private
    - Must never be revealed: to avoid false coupling
- Shared nothing representation of local state
  - Enact via messaging

BSPL, the Blindingly Simple Protocol Language

Main ideas

- Only *two* syntactic notions
  - Declare a message schema: as an atomic protocol
  - Declare a composite protocol: as a bag of references to protocols
- Parameters are central
  - Provide a basis for expressing meaning in terms of bindings in protocol instances
  - Yield unambiguous specification of compositions through public parameters
  - Capture progression of a role's knowledge
  - Capture the completeness of a protocol enactment
  - Capture uniqueness of enactments through keys
- Separate structure (parameters) from meaning (bindings)
  - Capture many important constraints purely structurally

Key Parameters in BSPL

Marked as ⌜key⌝

- All the key parameters *together* form the key
- Each protocol must define at least one key parameter
- Each message or protocol reference must have at least one key parameter in common with the protocol in whose declaration it occurs
- The key of a protocol provides a basis for the uniqueness of its enactments

Parameter Adornments in BSPL

Capture the essential causal structure of a protocol (for simplicity, assume all parameters are strings)

- ⌜in⌝: Information that must be provided to instantiate a protocol
  - Bindings must exist locally in order to proceed
  - Bindings must be produced through some other protocol
- ⌜out⌝: Information that is generated by the protocol instances
  - Bindings can be fed into other protocols through their ⌜in⌝ parameters, thereby accomplishing composition
  - A standalone protocol must adorn all its public parameters ⌜out⌝
- ⌜nil⌝: Information that is absent from the protocol instance
  - Bindings must not exist

Protocol in BSPL: Main Ideas

- Declarative
  - No control flow, no control state
- Information-based
  - Specifies the computation of distributed information object
    - Message specification is atomic protocol
  - Specified via parameters
- Explicit causality
  - The messages an agent can send depends upon what it knows
  - Via parameter adornments ⌜out⌝, ⌜in⌝, ⌜nil⌝
- Integrity
  - Agent only sends messages that preserve consistency of objects
  - Via key constraints
- Asynchronous messaging
- Requires no ordering from infrastructure
- Composition and verification

The *Initiate* protocol

```
Initiate {
 role B, S
 parameter out ID key, out item

 B ↦ S: rfq[out ID key, out item]
}
```

The *Initiate* protocol

```
Initiate {
 role B, S
 parameter out ID key, out item

 B ↦ S: rfq[out ID key, out item]
}
```

Initiate (virtual)	
ID	item

B:rfq	
ID	item

S:rfq	
ID	item

The *Initiate* protocol

```
Initiate {
 role B, S
 parameter out ID key, out item

 B ↦ S: rfq[out ID key, out item]
}
```

Initiate (virtual)

ID	item
1	fig

B:rfq

ID	item
1	fig

S:rfq

ID	item

The *Initiate* protocol

```
Initiate {
 role B, S
 parameter out ID key, out item

 B ↦ S: rfq[out ID key, out item]
}
```

Initiate (virtual)

ID	item
1	fig
5	jam

B:rfq

ID	item
1	fig
5	jam

S:rfq

ID	item

The *Initiate* protocol

```
Initiate {
 role B, S
 parameter out ID key, out item

 B ↦ S: rfq[out ID key, out item]
}
```

Initiate (virtual)

ID	item
1	fig
5	jam

B:rfq

ID	item
1	fig
5	jam

S:rfq

ID	item
5	jam

The *Initiate* protocol

```
Initiate {
 role B, S
 parameter out ID key, out item

 B ↦ S: rfq[out ID key, out item]
}
```

Initiate (virtual)

ID	item
1	fig
5	jam

B:rfq

ID	item
1	fig
5	jam
×1	apple

S:rfq

ID	item
5	jam

The *Initiate* protocol

```
Initiate {
 role B, S
 parameter out ID key, out item

 B ↦ S: rfq[out ID key, out item]
}
```

Initiate (virtual)

ID	item
1	fig
5	jam
8	fig

B:rfq

ID	item
1	fig
5	jam
8	fig

S:rfq

ID	item
5	jam

The *Offer* Protocol

```
Offer {
 role B, S
 parameter out ID key, out item, out price

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

The *Offer* Protocol

```
Offer {
 role B, S
 parameter 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

The *Offer* Protocol

```
Offer {
 role B, S
 parameter 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	10

B:rfq

ID	item
1	fig

B:quote

ID	item	price

S:rfq

ID	item
1	fig

S:quote

ID	item	price
1	fig	10

The *Offer* Protocol

```
Offer {
 role B, S
 parameter 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	10

B:rfq

ID	item
1	fig

B:quote

ID	item	price
1	fig	10

S:rfq

ID	item
1	fig

S:quote

ID	item	price
1	fig	10

The *Offer* Protocol

```
Offer {
 role B, S
 parameter 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	10

B:rfq

ID	item
1	fig

B:quote

ID	item	price
1	fig	10

S:rfq

ID	item
1	fig

S:quote

ID	item	price
1	fig	10
×4	fig	10

The *Decide Offer* Protocol

Choice: *accept* and a *reject* with the same ID *cannot* both occur

```
Decide Offer {
 role B, S
 parameter out ID key, out item, out price, out decision

 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 decision]
 B ↦ S: reject[in ID, in item, in price, out decision]
}
```

The *Decide Offer* Protocol

Choice: *accept* and a *reject* with the same ID *cannot* both occur

```
Decide Offer {
 role B, S
 parameter out ID key, out item, out price, out decision

 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 decision]
 B ↦ S: reject[in ID, in item, in price, out decision]
}
```

Decide Offer (virtual)

ID	item	price	decision
1	fig	10	

B:rfq

ID	item
1	fig

B:quote

ID	item	price
1	fig	10

B:accept

ID	item	price	decision

B:reject

ID	item	price	decision

The *Decide Offer* Protocol

Choice: *accept* and a *reject* with the same ID *cannot* both occur

```
Decide Offer {
 role B, S
 parameter out ID key, out item, out price, out decision

 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 decision]
 B ↦ S: reject[in ID, in item, in price, out decision]
}
```

Decide Offer (virtual)

ID	item	price	decision
1	fig	10	nice

B:rfq

ID	item
1	fig

B:quote

ID	item	price
1	fig	10

B:accept

ID	item	price	decision
1	fig	10	nice

B:reject

ID	item	price	decision

The *Decide Offer* Protocol

Choice: *accept* and a *reject* with the same ID *cannot* both occur

```
Decide Offer {
 role B, S
 parameter out ID key, out item, out price, out decision

 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 decision]
 B ↦ S: reject[in ID, in item, in price, out decision]
}
```

Decide Offer (virtual)

ID	item	price	decision
1	fig	10	nice

B:rfq

ID	item
1	fig

B:quote

ID	item	price
1	fig	10

B:accept

ID	item	price	decision
1	fig	10	nice

B:reject

ID	item	price	decision
×1	fig	10	nice

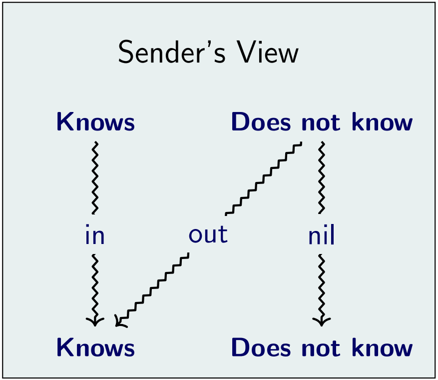
The *Purchase* Protocol

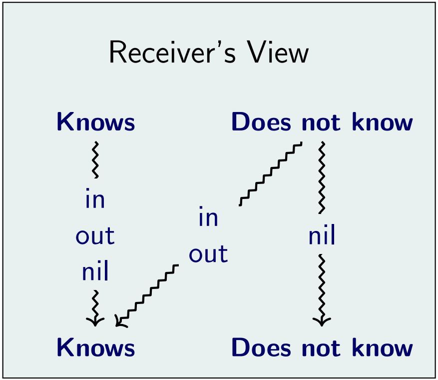
```
Purchase {
 role B, S, Shipper
 parameter out ID key, out item, out price, out outcome
 private address, resp

 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]

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

- *reject* conflicts with *accept* on resp (a *private* parameter)
- *reject* or *deliver* must occur for completion (to bind outcome)

Knowledge and Viability

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

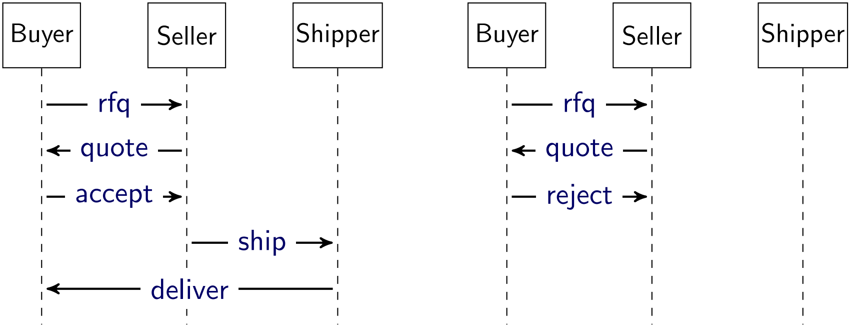




- Knowledge increases monotonically at each role
- An ⌜out⌝ parameter **creates** and transmits knowledge
- An ⌜in⌝ parameter transmits knowledge
- Repetitions through multiple paths are harmless and superfluous

Possible Enactments as Sets of Local Histories

Each participant's local history: set of messages sent and received



Standing Order

Composite keys and Composition

```
Insurance-Claims {
 role Vendor (V), Subscriber (S)
 parameter in pID key, out cID key, out claim, out response
 S ↦ V: claimRequest[in pID, out cID, out claim]
 V ↦ S: claimResponse[in pID, in cID, out response]
}

Create-Policy {
 role V, S,
 parameter out pID, out details
 ...
 Insurance-Claims(V, S, in pID key, out cID key, out claim, out
  response)
}
```

- A policy (identified by pID) may be associated with multiple claims (identified by cID)
- Create-Policy composes Insurance-Claims, supplying it with pID

in-out Polymorphism

price could be ⌜in⌝ or ⌜out⌝

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

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

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

- The price can be adorned ⌜in⌝ or ⌜out⌝ in a reference to this protocol

Flexible Sourcing of out Parameters

Buyer or Seller Offer

```
Buyer−or−Seller−Offer {
 role Buyer, Seller
 parameter in ID key, out item, out price, out confirmed

 Buyer ↦ Seller: rfq[in ID, out item, nil price]
 Buyer ↦ Seller: rfq[in ID, out item, out price]

 Seller ↦ Buyer: quote[in ID, in item, out price, out confirmed]
 Seller ↦ Buyer: quote[in ID, in item, in price, out confirmed]
}
```

- The BUYER or the SELLER may determine the binding
- The BUYER has first dibs in this example

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

Summary: Main Ideas

Taking a declarative, information-centric view of interaction to the limit

- Specification
  - A message is an atomic protocol
  - A composite protocol is a set of references to protocols
  - Each protocol is given by a name and a set of parameters (including keys)
  - Each protocol has *inputs* and *outputs*
- Representation
  - A protocol corresponds to a relation (table)
  - Integrity constraints apply on the relations
- Enactment via LoST: Local State Transfer
  - Information represented: local $\neq$ internal
  - Purely decentralized at each role
  - Materialize the relations *only* for messages

Realizing BSPL via 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

Information Centrism

Characterize each interaction purely in terms of information

- Explicit causality
  - Flow of information coincides with flow of causality
  - No hidden control flows
  - No backchannel for coordination
- Keys
  - Uniqueness
  - Basis for completion
- Integrity
  - Parameter has only one value (relative to its value of its key)
- Immutability
  - Durability
  - Robustness: insensitivity to
    - Reordering by infrastructure
    - Retransmission: one delivery is all it needs

Safety: *Purchase Unsafe*

Remove conflict between *accept* and *reject*

```
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]
}
```

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

Liveness: *Purchase No Ship*

Omit *ship*

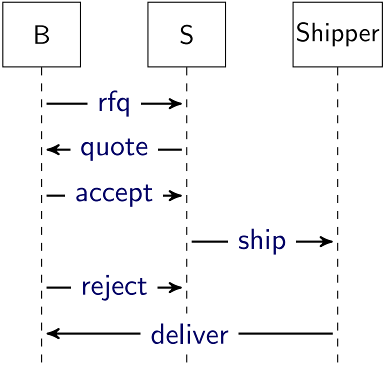
```
Purchase No Ship {
 role B, S, Shipper
 parameter out ID key, out item, out price, out outcome
 private address, resp

 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

Safety and Liveness Violations

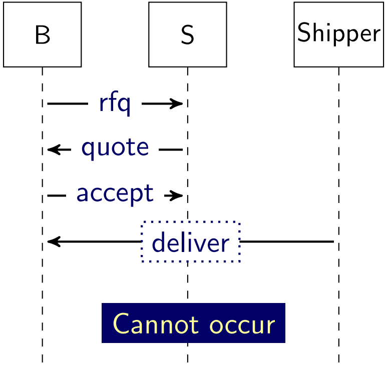
Encode a protocol's causal structure in temporal logic and evaluate propertes

Purchase Unsafe



Safety Violation

Purchase No Ship



Liveness Violation

Encode Causal Structure as Temporal Constraints

- *Reception*. If a message is received, it was previously sent.
- *Information transmission* (sender's view)
  - Any ⌜in⌝ parameter occurs prior to the message
  - Any ⌜out⌝ parameter occurs simultaneously with the message
- *Information reception* (receiver's view)
  - Any ⌜out⌝ or ⌜in⌝ parameter occurs before or simultaneously with the message
- *Information minimality*. If a role observes a parameter, it must be simultaneously with *some* message sent or received
- *Ordering*. If a role sends any two messages, it observes them in some order

Verifying Safety

- Competing messages: those that have the same parameter as out
- *Conflict*. At least two competing messages occur
- *Safety* iff the causal structure $\wedge$ conflict is unsatisfiable

Verifying Liveness

- *Maximality*. If a role is enabled to send a message, it sends at least one such message
- *Reliability*. Any message that is sent is received
- *Incompleteness*. Some public parameter fails to be bound
- *Live* iff the causal structure $\wedge$ the above three is unsatisfiable

Exercises 1: *Abruptly Cancel*

```
Abruptly Cancel {
 role B, S
 parameter 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?
- Is this protocol live?

Exercise 2: *Abruptly Cancel* Modified (with ⌜nil⌝)

```
Abruptly Cancel {
 role B, S
 parameter 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?
- Is this protocol live?

Exercise 3: Goods Priority

- Modify *Abruptly Cancel* so that goods takes priority over cancel
  - If S sends Goods, that is the outcome of the interaction
  - S cannot send Goods after receiving Cancel
  - If S receives Cancel before Goods, cancellation is the outcome
  - B cannot send Cancel after receiving Goods

Solution

```
Abruptly Cancel {
 role B, S
 parameter out ID key, out item, out outcome

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

Outline

Multiagent system = agents + interaction protocol

Power of AI agents: their flexibility

An *interaction protocol* models the communication constraints between agents in a multiagent system

Engineering multiagent system based on protocols offers key benefits

- *Decentralized MAS*; without relying on a distinguished locus of state or control
- *Clear implementation*, separation between the coordination aspects and business logic of an agent
- *Loose coupling*, changes in one agent's implementation do not affect the implementation of others
- *Reducing agent complexity*, avoiding programming errors

Traditional Approach: Informal and Ad Hoc

Prevalent approaches curtail agent flexibility

Unfortunately, leading (cognitive) programming models for MAS

- Jason [Vieira et al., 2007], JADEL [Bergenti et al., 2017], JaCaMo [Boissier et al., 2013], JADE [Bellifemine et al., 2007], SARL [Rodriguez et al., 2014]

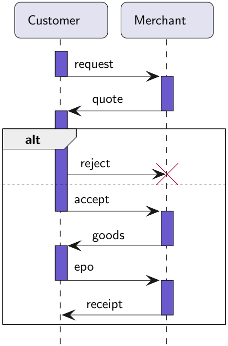
Programming AI agents to enjoy flexibility is difficult

- Methods for interactions and agent programming bypass each other
- Message handlers ignore the protocol
- Agent code is unwieldy and difficult to maintain

Traditional Approach: Informal and Ad Hoc

Informal protocols; no support for meaning; programmer tracks state

The *NetBill* protocol as a UML Sequence Diagram



Listing: Jason snippet of a MERCHANT agent.

```
+request(Id, Item)[source(Customer)]
 : price(Item, Price)
 <- +nbp_state(Id, quoting);
     .send(Customer, tell, quote(Id,
      Item, Price)).
+accept(Id,Item,Price)[source(Customer
 )]
 : nbp_state(Id, quoting) &
   goods(Item, Goods)
 <- -nbp_state(Id, quoting);
    +nbp_state(Id, shipping);
     .send(Customer, tell, goods(Id,
      Item, Price, Goods)).
```

Shortcomings about AOPLs

In 2012, Michael Winikoff [Winikoff, 2012] highlighted two shortcomings about AOPLs

AOPLs supported little more than *primitives* for sending and receiving messages

- GOTOs and labels: Winikoff saw the use of such primitives as transferring control between agents and drew an unflattering analogy with the use of *gotos* in programming

Interaction protocols (typically in AUML) were *message-centric* and *over-constrained* the interaction between agents

- Less flexibility and robustness Interaction protocols (AUML) do not leave the agents room to be autonomous or to exploit their flexibility and robustness when interacting with other agents

Shortcomings about AOPLs

In 2012, Michael Winikoff [Winikoff, 2012] highlighted two shortcomings about AOPLs

AOPLs still suffer from the shortcomings Winikoff highlighted!

As a result:

- Asynchronous message exchanges and multi-party solutions are more difficult to design and to program and they are more prone to errors
- Incompatibilities between agents due to the message schemas being blended into business logic
- Semantic errors due to a lack of a formal model
- Inflexibility due to the programmer having to maintain the protocol state via a state machine

Orpheus

In Greek mythology, a poet and a companion of Jason on his adventures

Di Giovanni Dall'Orto - Opera propria, Pubblico dominio,

https://commons.wikimedia.org/w/index.php?curid=1303452

Orpheus

Orpheus unites two aspects of autonomy

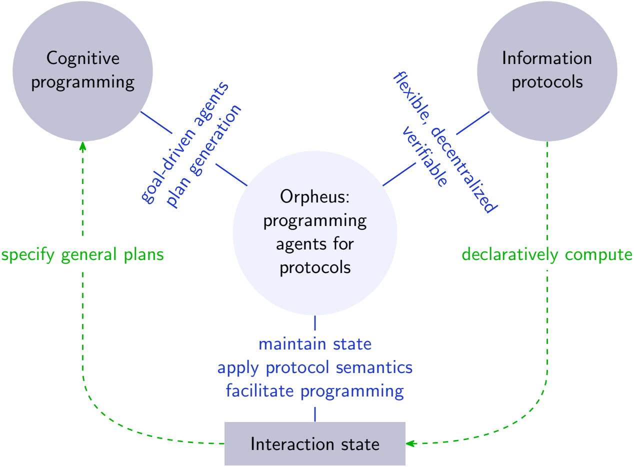
- **Cognitive autonomy**, via Jason [Vieira et al., 2007]
- **Social autonomy**, via information protocols, in particular, *Blindingly Simple Protocol Language* (BSPL) [Singh, 2011]

Orpheus overcomes shortcomings of message-centric interaction protocols

- Offering to *programmers/developers* AOPLs that include **higher level abstractions** that **hide** low-level messaging concerns (as recommended in [Winikoff, 2012])
- its centrepiece is the generation of Jason adapter that supports an agent programming (API) that enables engineering loosely coupled, flexible, and decentralised MAS

Orpheus

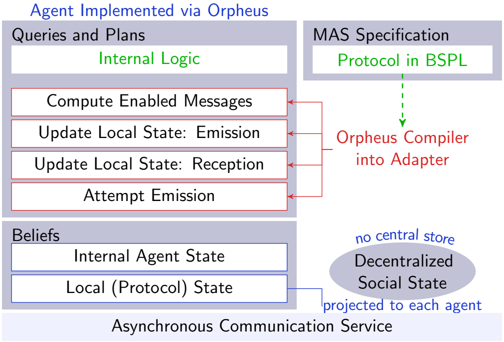
Engineering multiagent systems: Engineering protocols + engineering agents



Orpheus Programming Model

Designing Agents with Orpheus

- Orpheus focuses not on reactions to incoming messages
- Orpheus focuses on computing **messages enabled** to be sent given the protocol semantics and the **information available** to the agent
- Orpheus abstracts out reasoning about the protocol into automatic generated code (through the *Orpheus Tool*)



An example

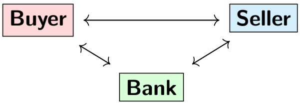
The EBusiness Protocol:
Seller can make an offer; Buyer can instruct Bank to pay; Bank can transfer funds to Seller; Seller can send shipment or refund

```
EBusiness {
    role Buyer, Seller, Bank
    parameter out ID key, out item, out price, out status
    Seller -> Buyer : offer [out ID key, out item, out price]
    Buyer -> Seller : accept [in ID key, in item, in price, out decision]
    Buyer -> Bank : instruct [in ID key, in price, out details]
    Bank -> Seller : transfer [in ID key, in price, in details, out payment]
    Seller -> Buyer : shipment [in ID key, in item, in price, out status]
    Seller -> Bank : refund [in ID key, in item, in payment, out amount, out status]
}
```

An example

The EBusiness Protocol

item



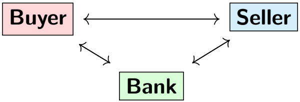
Buyer → Seller : rfq [out ID key, out item]

An example

The EBusiness Protocol

item

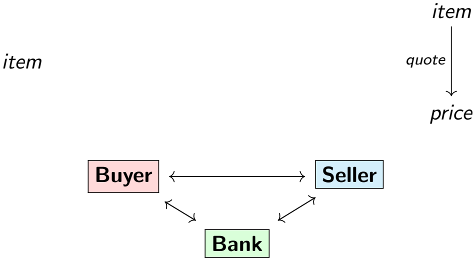
item



Buyer → Seller : rfq [out ID key, out item] SENT!

An example

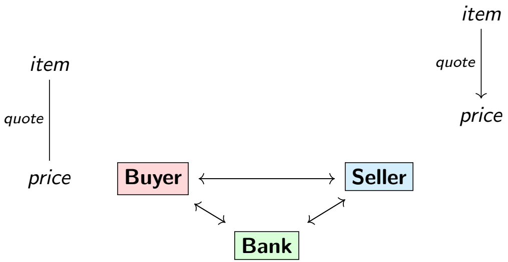
The EBusiness Protocol



Seller $\rightarrow$ Buyer : quote [in ID key, in item, out price]

An example

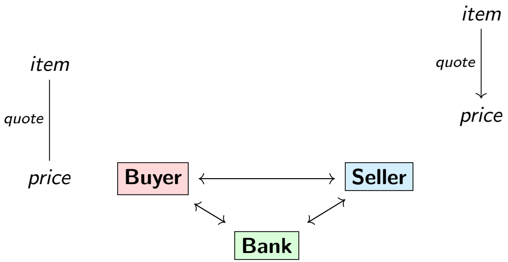
The EBusiness Protocol



Seller $\rightarrow$ Buyer : quote [in ID key, in item, out price] SENT!

An example

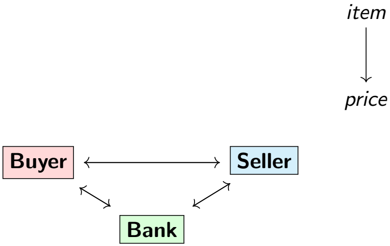
The EBusiness Protocol



Seller → Buyer : quote [in ID key, out price] SENT!

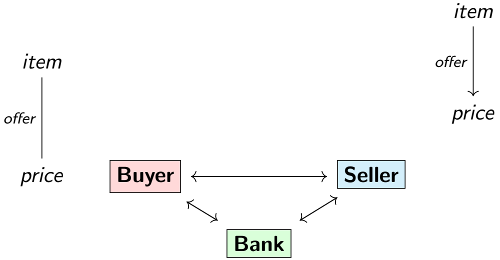
An example

The EBusiness Protocol



An example

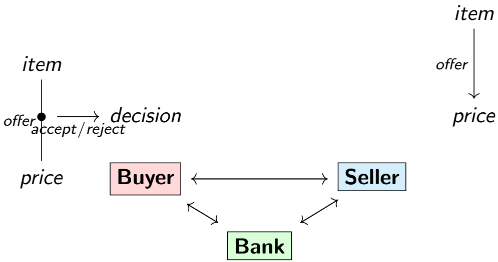
The EBusiness Protocol



Seller → Buyer : offer [out ID key, out item, out price] SENT!

An example

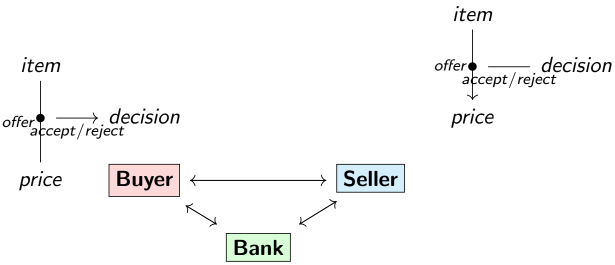
The EBusiness Protocol



Buyer → Seller : accept [in ID key, in item, in price, out decision]

An example

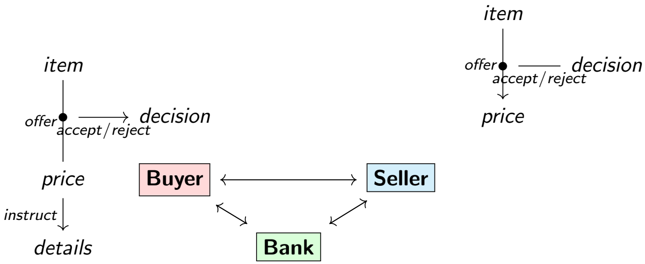
The EBusiness Protocol



Buyer → Seller : accept [in ID key, in item, in price, out decision] SENT!

An example

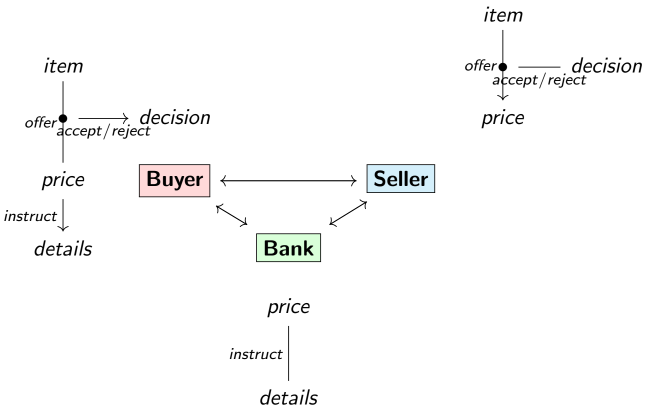
The EBusiness Protocol



Buyer → Bank : instruct [in ID key, in price, out details]

An example

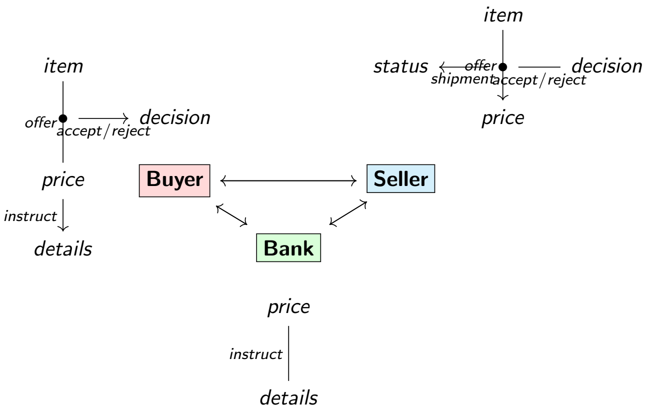
The EBusiness Protocol



Buyer → Bank : instruct [in ID key, in price, out details] SENT!

An example

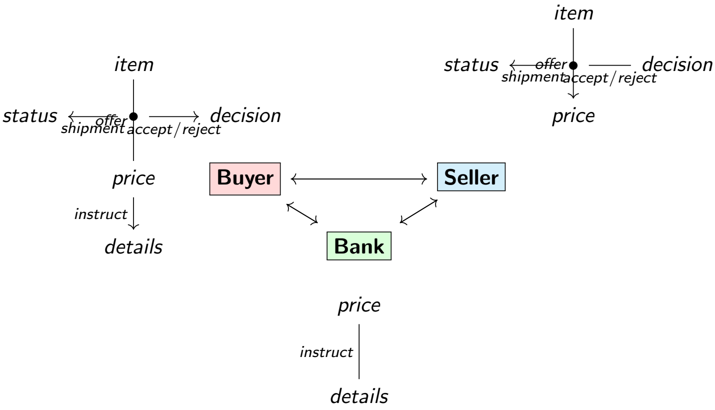
The EBusiness Protocol



Seller → Buyer : shipment [in ID key, in item, in price, out status]

An example

The EBusiness Protocol

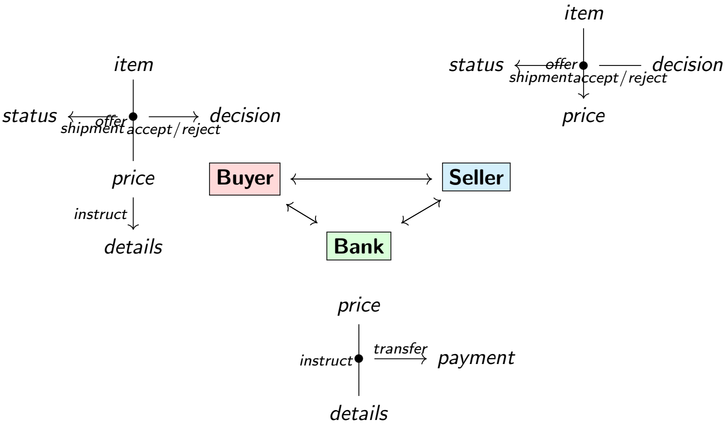


Seller → Buyer : shipment [in ID key, in item, in price, out status]

SENT!

An example

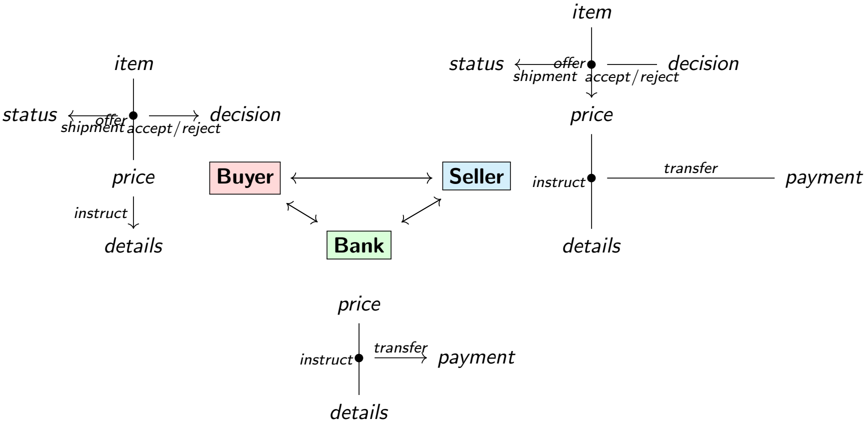
The EBusiness Protocol



Bank → Seller : transfer [in ID key, in price, in details, out status]

An example

The EBusiness Protocol

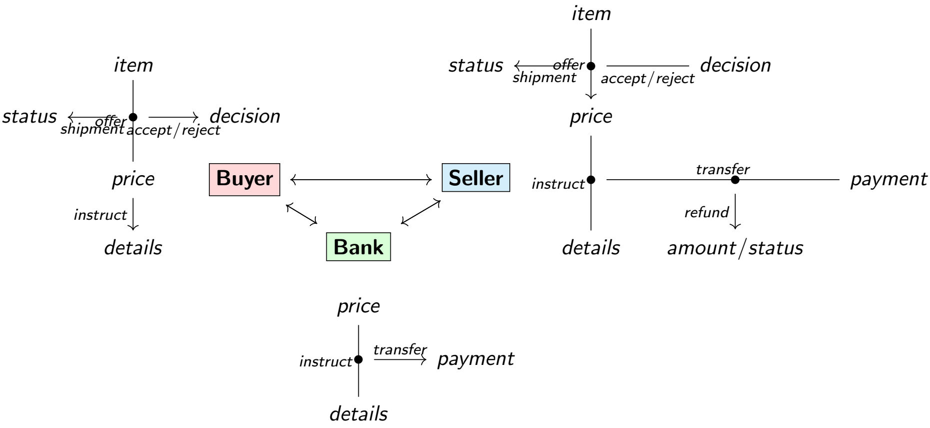


Bank → Seller : transfer [in ID key, in price, in details, out status]

SENT!

An example

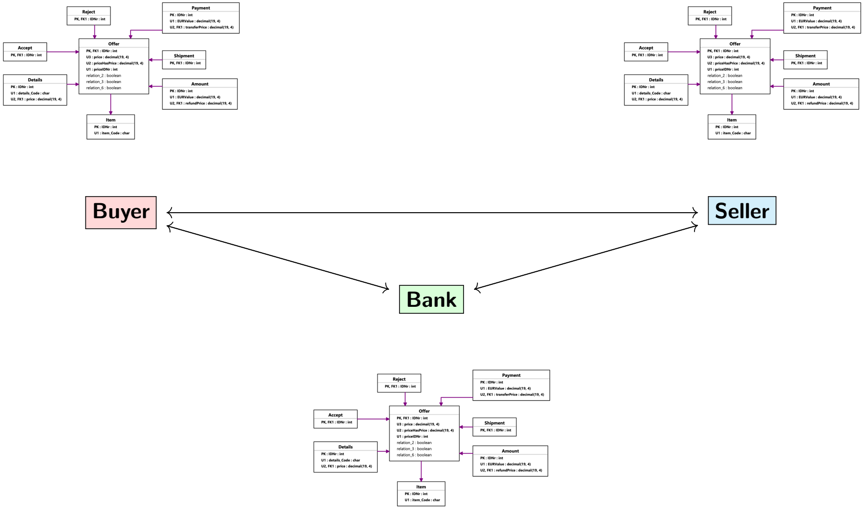
The EBusiness Protocol



Seller → Bank : refund [in ID key, in item, in payment, out amount, out status]

An example

The EBusiness Protocol



Orpheus Programming Model

Designing Agents with Orpheus

An incoming message is added to the local state if it is consistent with the local state

- I.e., if no other binding is already known for any its parameters (relative to the key)

Outgoing messages

- An enabled instance is a partial instance in that:
  - its IN parameters are bound because their bindings are known, and
  - its OUT parameters are not bounded because they are not known
- An attempt is successful if the completed messages are mutually consistent in their bindings; the sent messages are added to the local state

Orpheus Programming Model

Enabled-Based Programming Model

Orpheus supports a novel programming model based on message enablement, in which the developer specifies plans for emitting enabled messages

To achieve some goal, the agent

- **queries** if there are enabled instances corresponding to the message it wants to send,
- **completes** them by producing bindings for their OUT parameters, and
- **attempts** to send them in one shot

Plan Pattern and Orpheus Primitives

Listing: Identify a goal and which messages to send for it

```
+!g
  :   enabled(m1) &...& enabled(mq)
  <- !complete(m1,...,mq);
     !attempt(m1,...,mq).

+!attempt(m1,...,mq)
  :   consistent(m1,...,mq)
  <- for (.member(m[receiver(R)], [m1,...,mq])) {
        .send(R, tell, m);
        +sent(m)
     }.

enabled(m(...)) :- ... //BSPL semantics

consistent(m1...mq) :-... //BSPL semantics

+sent(m) <- ... // BSPL semantics

+m : consistent(m, local) <- ... // BSPL semantics
```

The Orpheus Compiler

https://www.di.unito.it/~baldoni/argonauts/

The download page of Orpheus

The Orpheus Compiler

https://www.di.unito.it/~baldoni/argonauts/

A zip file with Orpheus and some examples

https://www.di.unito.it/~baldoni/argonauts/AAMAS2025.zip

A brief introduction to Jason

https://www.di.unito.it/~baldoni/argonauts/SlidesJason.pdf

The Orpheus Compiler

https://www.di.unito.it/~baldoni/argonauts/

The Orpheus compiler automatically generates agent adapters to manage the local state and query it.

```
java -jar argonauts.jar --orpheus <file.xml>
```

Listing: Generated Jason code for computing enabled *accepts*

```
enabled(accept(Id, Item, Price, out)[receiver(Seller)]) :-
  table_ID(Id) &
  table_item(Id, Item) &
  table_price(Id, Price) &
  not (table_decision(Id, Decision)) &
  table_Seller(Id, Seller).
```

The Orpheus Compiler

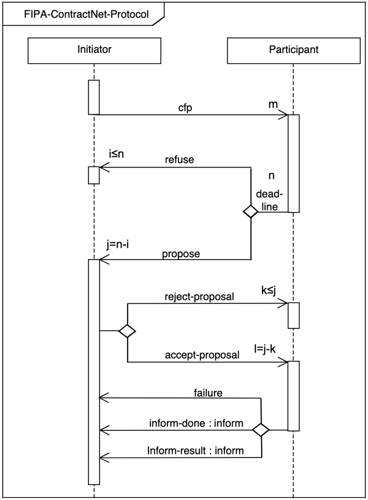
https://www.di.unito.it/~baldoni/argonauts/

Listing: Generated Jason code for computing updating local state

```
+sent(accept(Id, Item, Price, Decision)[receiver(Seller)])
  <- +table_decision(Id, Decision).

+offer(Id, Item, Price)[source(Seller)] :
  not ( table_ID(Id) &
  table_item(Id, Item_other) &
  table_price(Id, Price_other) &
  table_Seller(Id, Seller_other) &
  (Item \== Item_other | Price \== Price_other |
  Seller \== Seller_other) )
  <- +table_ID(Id);
     +table_item(Id, Item);
     +table_price(Id, Price);
     +table_Seller(Id, Seller).
```

Exercise

FIPA Contract Net Interaction Protocol Specification



Exercise

BSPL Contract Net Protocol

```
ContractNet {
        role Contractor, Participant
        parameter out IDt key, out task, out outcome
        private pdecision, offer, outcome
        Contractor -> Participant : cfp [out IDp key, out IDt key, out task]
        Participant -> Contractor : propose [in IDp key, in IDt key, in task, out offer, out pdecision]
        Participant -> Contractor : refuse [in IDp key, in IDt key, in task, out outcome, out pdecision]
        Contractor -> Participant : accept_prop [in IDp key, in IDt key, in offer, out accept, out x]
        Contractor -> Participant : reject_prop [in IDp key, in IDt key, in offer, out outcome, out x]
        Participant -> Contractor : done [in IDp key, in IDt key, in accept, out outcome]
        Participant -> Contractor : failure [in IDp key, in IDt key, in accept, out outcome]
}
```

Exercise

An implementation of CNP in Jason with Orpheus

Listing: An excerpt of the contractor agent code

```
+!startCNP(Idt,Task)
  <-  !find_participants;
      !cfp(Idt, Task);
      .wait(all_enabled_proposal(Idt), 10000, _);
      !contract(Idt).

+!cfp(Idt, Task)
  :   true
  <-  for ( participant(Participant) ) {
        // WHAT IS MISSING HERE ?
        !get_new_counter(Idp);
        !attempt(cfp(Idp, Idt, Task)[receiver(Participant)]);
      }.
```

Exercise

An implementation of CNP in Jason with Orpheus

Listing: An excerpt of the contractor agent code

```
+!startCNP(Idt,Task)
  <-  !find_participants;
      !cfp(Idt, Task);
      .wait(all_enabled_proposal(Idt), 10000, _);
      !contract(Idt).

+!cfp(Idt, Task)
  :  true
  <- for ( participant(Participant) ) {
       ?enabled(cfp(out, out, out)[receiver(out)]);
       !get_new_counter(Idp);
       !attempt(cfp(Idp, Idt, Task)[receiver(Participant)]);
     }.
```

Exercise

An implementation of CNP in Jason with Orpheus

Listing: An excerpt of the contractor agent code

```
+!contract(Idt)
  :   true
  <-  .findall(
         offer(Offer, Idp, P),
         // WHAT IS MISSING HERE ?
         L
     );
     L \== []; // constraint the plan execution to at least one offer
     .min(L, offer(WOffer, WIdp, WAg)); // sort offers, the first is the best
     !announce_result(Idt, L, WIdp, WAg).
```

Exercise

An implementation of CNP in Jason with Orpheus

Listing: An excerpt of the contractor agent code

```
+!contract(Idt)
  :  true
  <- .findall(
       offer(Offer, Idp, P),
       enabled(accept_prop(Idp, Idt, Offer, out, out)[receiver(P)]),
       L
     );
     L \== []; // constraint the plan execution to at least one offer
     .min(L, offer(WOffer, WIdp, WAg)); // sort offers, the first is the best
     !announce_result(Idt, L, WIdp, WAg).
```

Exercise

An implementation of CNP in Jason with Orpheus

Listing: An excerpt of the contractor agent code

```
+!announce_result(_, [], _, _).

// announce to the winner
+!announce_result(Idt, [offer(Offer, WIdp, WAg) | T], WIdp, WAg)
  :  // WHAT IS MISSING HERE ?
  <- !complete(accept_prop(WIdp, Idt, Offer, Accept, X)[receiver(WAg)]);
     // WHAT IS MISSING HERE ?
     !announce_result(Idt, T, WIdp, WAg).

// announce to others
+!announce_result(Idt, [offer(Offer, LIdp, LAg) | T], WIdp, WAg)
  :  LAg \== WAg & LIdp \== WIdp &
     // WHAT IS MISSING HERE ?
  <- !complete(reject_prop(LIdp, Idt, Offer, Outcome, X)[receiver(LAg)]);
     // WHAT IS MISSING HERE ?
     !announce_result(Idt, T, WIdp, WAg).
```

Exercise

An implementation of CNP in Jason with Orpheus

Listing: An excerpt of the contractor agent code

```
+!announce_result(_, [], _, _).
// announce to the winner
+!announce_result(Idt, [offer(Offer, WIdp, WAg) | T], WIdp, WAg)
  :  enabled(accept_prop(WIdp, Idt, Offer, out, out)[receiver(WAg)])
  <- !complete(accept_prop(WIdp, Idt, Offer, Accept, X)[receiver(WAg)]);
     !attempt(accept_prop(WIdp, Idt, Offer, Accept, X)[receiver(WAg)]);
     !announce_result(Idt, T, WIdp, WAg).
// announce to others
+!announce_result(Idt, [offer(Offer, LIdp, LAg) | T], WIdp, WAg)
  :  LAg \== WAg & LIdp \== WIdp &
     enabled(reject_prop(LIdp, Idt, Offer, out, out)[receiver(LAg)])
  <- !complete(reject_prop(LIdp, Idt, Offer, Outcome, X)[receiver(LAg)]);
     !attempt(reject_prop(LIdp, Idt, Offer, Outcome, X)[receiver(LAg)]);
     !announce_result(Idt, T, WIdp, WAg).
```

Azorus

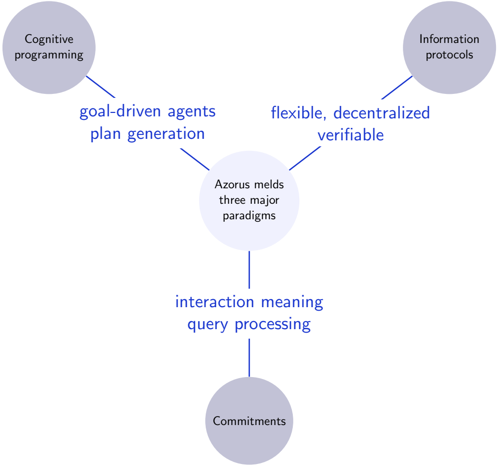
In Greek mythology, the helmsman of Jason's ship, the Argo

By Konstantinos Volanakis, Public Domain,

https://commons.wikimedia.org/w/index.php?curid=9080257

Azorus

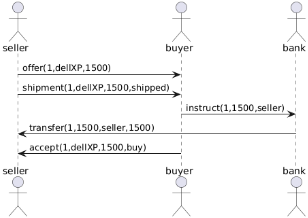
Yet, leading methods for interactions and agent programming bypass each other

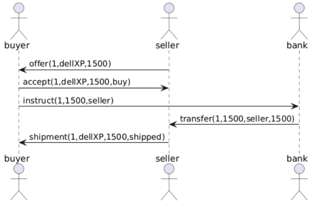


An example

EBusiness Protocol: Many Ways to Execute It

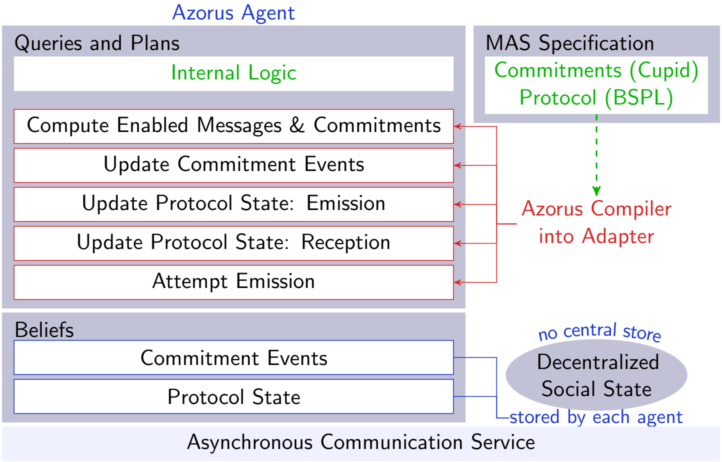
```
EBusiness {
    role Buyer, Seller, Bank
    parameter out ID key, out item, out price, out status
    Seller -> Buyer : offer [out ID key, out item, out price]
    Buyer -> Seller : accept [in ID key, in item, in price, out decision]
    Buyer -> Bank : instruct [in ID key, in price, out details]
    Bank -> Seller : transfer [in ID key, in price, in details, out payment]
    Seller -> Buyer : shipment [in ID key, in item, in price, out status]
    Seller -> Bank : refund [in ID key, in item, in payment, out amount, out status]
}
```





Azorus Architecture and Framework

Green: what developers provide; blue: state; red: tool & generated components



Azorus

Specify operations via an information protocol and meanings via commitments

```
commitment OfferCom Seller to Buyer   // if transfer, then shipment
  create offer
  detach transfer[, created OfferCom + 5] where "Payment>=Price"
  discharge shipment [, detached OfferCom + 5]

commitment AcceptCom Buyer to Seller // if shipment, then transfer
  create accept
  detach shipment[, created AcceptCom + 5]
  discharge transfer[, detached AcceptCom + 5] where "Payment>=Price"

commitment RefundCom Seller to Buyer // if violated OfferCom, then refund
  create offer
  detach violated OfferCom
  discharge refund[, detached RefundCom + 2] where "Amount >= Payment"

commitment TransferCom Bank to Buyer // if instructed, then transfer
  create instruct
  discharge transfer[, created TransferCom + 2] where "Payment=Price"
```

Azorus

Snippets of generated agent code

```
now_detached(OfferCom, Seller, Buyer, Id, Item, Price, Bank,
  Payment, Timestamp0) :-
  detached_OfferCom(Seller, Buyer, Id, Item, Price, Bank, Payment
    , Timestamp0) &
  myLib.second_time(Now) &
 Timestamp0<=Now.

enabled(shipment(Id, Item, Price, out)[receiver(Buyer)]) :-
  table_ID(Id) &
  table_item(Id, Item) &
  table_price(Id, Price) &
  not (table_status(Id, Status)) &
  table_Buyer(Id, Buyer).

+!attempt(Message[receiver(R)]) <-
  .send(R, tell, Message);
   +sent(Message[receiver(R)]).
```

Exercise

EBusiness Protocol: Seller agent

Listing: An excerpt of the Seller agent code

```
+!select_and_handle_message([shipment(Id, Item, Price, out)[receiver(Buyer)] | _])
  : enabled(shipment(Id, Item, Price, out)[receiver(Buyer)]) &
    // WHAT IS MISSING HERE??
    in_stock(Item) & .my_name(Seller)
  <- !complete(shipment(Id, Item, Price, Status)[receiver(Buyer)]);
    !attempt(shipment(Id, Item, Price, Status)[receiver(Buyer)]).
```

Listing: An excerpt of the Seller agent code

```
+!select_and_handle_message([refund(Id, Item, Payment, out, out)[receiver(Bank)] | _])
  : enabled(refund(Id, Item, Payment, out, out)[receiver(Bank)]) &
    // WHAT IS MISSING HERE??
  <- !complete(refund(Id, Item, Payment, Amount, Status)[receiver(Bank)]);
    !attempt(refund(Id, Item, Payment, Amount, Status)[receiver(Bank)]).
```

Exercise

EBusiness Protocol: Seller agent

Listing: An excerpt of the Seller agent code

```
+!select_and_handle_message([shipment(Id, Item, Price, out)[receiver(Buyer)] | _])
  :   enabled(shipment(Id, Item, Price, out)[receiver(Buyer)]) &
      now_detached_OfferCom(Seller, Buyer, Id, Item, Price, Bank, Details, Payment, Time) &
      in_stock(Item) & .my_name(Seller)
  <-  !complete(shipment(Id, Item, Price, Status)[receiver(Buyer)]);
      !attempt(shipment(Id, Item, Price, Status)[receiver(Buyer)]).
```

Listing: An excerpt of the Seller agent code

```
+!select_and_handle_message([refund(Id, Item, Payment, out, out)[receiver(Bank)] | _])
  : enabled(refund(Id, Item, Payment, out, out)[receiver(Bank)]) &
    now_detached_RefundCom(Seller, Buyer, Id, Item, Price, Bank, Details, Payment, Time)
  <-  !complete(refund(Id, Item, Payment, Amount, Status)[receiver(Bank)]);
      !attempt(refund(Id, Item, Payment, Amount, Status)[receiver(Bank)]).
```

Azorus

Snippets of generated agent code

```
+!updateComEvents(Id) <-
  if(now_created_RefundCom(Seller, Buyer, Id, Item, Price, Time0) &
  not ev_now_created_RefundCom(Seller, Buyer, Id, Item, Price, Time0))
  { +ev_now_created_RefundCom(Seller, Buyer, Id, Item, Price, Time0);}

  if(now_detached_RefundCom(Seller, Buyer, Id, Item, Price, Bank, Details, Payment, Timestamp14) &
  not ev_now_detached_RefundCom(Seller, Buyer, Id, Item, Price, Bank, Details, Payment, Timestamp14))
  { +ev_now_detached_RefundCom(Seller, Buyer, Id, Item, Price, Bank, Details, Payment, Timestamp14);}

  if(now_discharged_RefundCom(Seller, Buyer, Id, Item, Price, Bank, Payment, Amount, Status, Timestamp15) &
  not ev_now_discharged_RefundCom(Seller, Buyer, Id, Item, Price, Bank, Payment, Amount, Status, Timestamp15))
  { +ev_now_discharged_RefundCom(Seller, Buyer, Id, Item, Price, Bank, Payment, Amount, Status, Timestamp15);}
  ...
```

Azorus

Reacting to commitment event state change

Listing: An excerpt of the Seller agent code

```
+ev_now_detached_OfferCom(Seller, Buyer, Id, Item, Price, Bank, Details, Payment, Time)
  :   enabled(shipment(Id, Item, Price, out)[receiver(Buyer)]) &
      in_stock(Item) &
      .my_name(Seller)
  <-  !complete(shipment(Id, Item, Price, Status)[receiver(Buyer)]);
      !attempt(shipment(Id, Item, Price, Status)[receiver(Buyer)]).

+ev_now_detached_RefundCom(Seller, Buyer, Id, Item, Price, Bank, Details, Payment, Time)
  :   enabled(refund(Id, Item, Payment, out, out)[receiver(Bank)]) &
      .my_name(Seller)
  <-  !complete(refund(Id, Item, Payment, Amount, Status)[receiver(Bank)]);
     !attempt(refund(Id, Item, Payment, Amount, Status)[receiver(Bank)]).
```

The Azorus Compiler

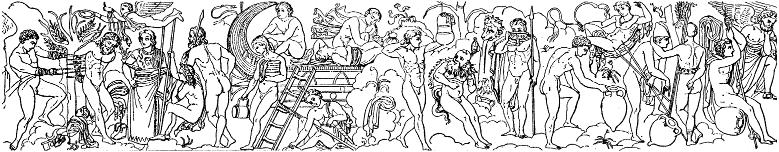
https://www.di.unito.it/~baldoni/argonauts/

The Azorus compiler automatically generates agent adapters to manage the local state and query it.

```
java -jar argonauts.jar --orpheus <file.xml> --azorus
<file.cupid>
```

The Argonauts

https://www.di.unito.it/~baldoni/argonauts/



By Nordisk familjebok - https://runeberg.org/nfba/0792.html,PublicDomain

Argonauts tools.

```
usage:  java -jar argonauts.jar [ ( -p | -orpheus ) <file> ] [ ( -z |
 -azorus) <file> ] [ -t ] [ -s ] [ -u ] [ -a ] [ -r ]
 -a,--azint set azorus integration for orpheus
 -m,--smp Adapters contain a plan for sending messages
 -p,--orpheus <orpheus> use orpheus compiler
 -r,--checkrole set check role out
 -s,--symboltable set bspl symbol table text output
 -t,--text set bspl text output
 -u,--plantuml set bspl plantuml output
 -z,--azorus <azorus> use azorus compiler
```

Programming Abstractions for Decentralized Decision Making

- Interactions traditionally viewed in terms of message ordering
- Properly, interactions are about decentralized decision making
  - An agent's communications represents its decisions, driven by internal (business) logic

Kiko

Enables programming agents on the basis of decision making abstractions that combine internal logic and communications

Noteworthy Benefits

- Enables programming an agent as a set of decision makers
- Empowers programmers by enabling them to focus on the business logic
  - Lower-level aspects of communication abstracted away
- Works over unordered, asynchronous communication channels
- Supports complex decision-making patterns leading to sets of communications, belonging possibly to multiple multiagent systems enacting different protocols

Basis: Declarative Information Protocols

Specify causality, not message ordering

```
Purchase {
  roles (B)uyer, (S)eller
  parameters out ID key, out item, out price, out done

  B -> S: RFQ[out ID key, out item]
  S -> B: Quote[in ID key, in item, out price]
  B -> S: Buy[in ID key, in item, in price, out done]
  B -> S: Reject[in ID key, in price, out done]
}
```

- B can send RFQ anytime by generating ID and item
- S can send Quote if it knows ID and item its from local state; it can generate any binding for price
- Buy and Reject are mutually exclusive since they both conflict on done

Forms as Causally-Enabled Communications

Message schemas with ⌜in⌝ parameters filled

```
B -> S: RFQ[out ID key, out item]
S -> B: Quote[in ID key, in item, out price]
B -> S: Buy[in ID key, in item, in price, out done]
B -> S: Reject[in ID key, in price, out done]
```

RFQ(1, fig)
RFQ(2, jam)
Quote(1, fig, $10)

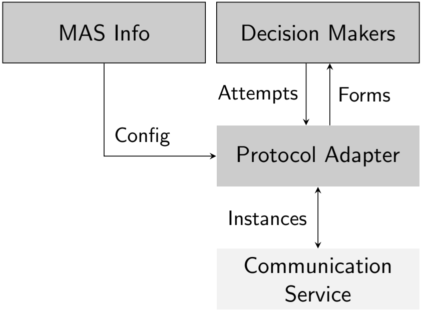
Buyer's Local state

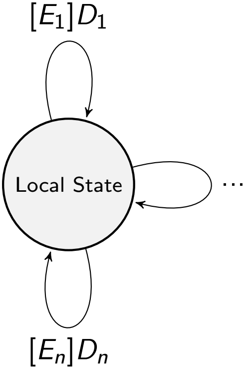
RFQ(ID, item)
Buy(1, fig, $10, done)
Reject(1, $10, done)

Buyer's Forms

Adapter-Supported Programming Model for Agents

Write agent as a set of event-triggered decision makers





- Adapter provides decision makers with forms
- *Attempts* are completed forms by the business logic
- Upon validation, attempts emitted as message instances

Buyer Bob's Decision Makers

```
@adapter.decision(event=InitEvent) // To send RFQs
def start(forms):
    for item in ["ball", "bat"]:
        ID = str(uuid.uuid4())
        for m in forms.messages(RFQ):
            m.bind(ID=ID, item=item)
```

```
@adapter.decision(event = 9AM) //To send Buy or Reject
def buy-reject(forms):
  for m in forms.messages(Buy):
    if(m["price"] < 20)
        m.bind(done="cool")
    else reject = next(forms.messages(Reject, ID=m["ID"]))
           reject.bind(done="rejected")
```

Multienactment Reasoning and Emitting Sets of Messages

```
//To buy cheapest and reject the rest
@adapter.decision(event = 9AM)
def cheapest(forms):
  buys = forms.messages(Buy)
  cheapest = min(buys, key=lambda b: b["price"])
  cheapest.bind(done=True)
```

```
/*Maximize number of Buys given budget; reject the others*/
@adapter.decision(event=birthday)
def select_gifts(forms):
    best, rest = best_combo(forms)
    for b in best: # buy the best items
        b.bind(done=True)
    for r in rest: # reject the rest
        r.bind(done=True)
```

Multiprotocol Business Logic

Asking for approval for each Buy

```
Approval {
  roles (R)equester, (A)pprover

  R -> A: Ask[out aID key, out request]
  A -> R: Approve[in aID, in request, out approved]
}
```

```

@adapter.enabled(Buy)
def request_approval(buy):
    ask = next(adapter.enabled_messages.messages(Ask), None)
    return ask.bind(ID=str(uuid.uuid4()), request=buy.payload)
```

Attempts with Contradictions Blocked From Emission

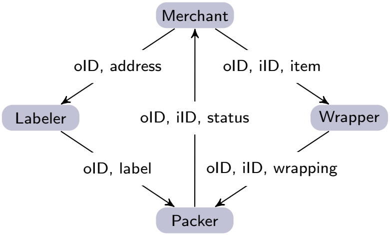
Attempts of a decision maker either wholly succeed or fail

```
@adapter.decision
def indecisive(forms):
  buy = next(forms.messages(Buy))
  reject = next(forms.messages(Reject, system=buy.system, ID=buy
    ["ID"]))
  buy.bind(done="accepted")
  reject.bind(done="rejected")
```

Direction

Leverage appropriate typing notions to enable treating a parameter as a resource that can be bound at most once in a decision maker.

Complex Correlation; Abstraction over Message Events




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

```
   @adapter.decision
  def decision_packing(forms):

      packeds = forms.messages(Packed)

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

Conclusions

Kiko: Information-based distributed programming

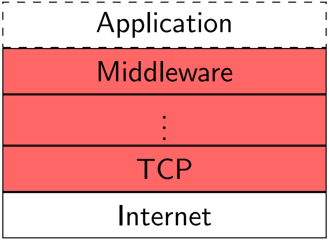
- Supports complex decision making patterns that involve atomically emitting sets of messages
- Avoids architectural complexity and inefficiency by not requiring ordered communication
- Takes maxim of letting programmers focus on the business logic to new heights

More Directions

- Enable decision making based on norms
- Make fault tolerance convenient (Mandrake coming up)

Fault Tolerance in Distributed Systems

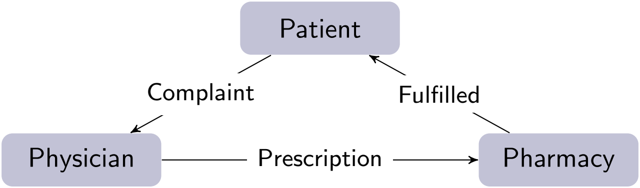
Current paradigm focuses on implementing reliability in communication services



- TCP guarantees in-order delivery of TCP segments within a connection
- Message queues, typically layered on top of TCP, do this at the level of messages

Delivery Guarantees in Communication Services

Inadequate because sender wants to know if message has been *processed* by receiver

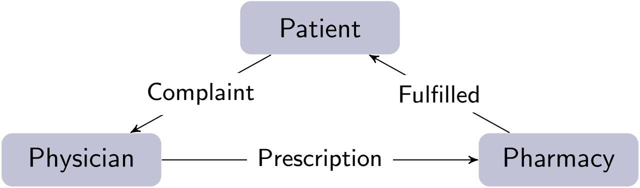


Fault

Despite delivery guarantees, Patient times out waiting for Fulfilled from Pharmacy. Why?

Delivery Guarantees in Communication Services

Inadequate because sender wants to know if message has been *processed* by receiver



Fault

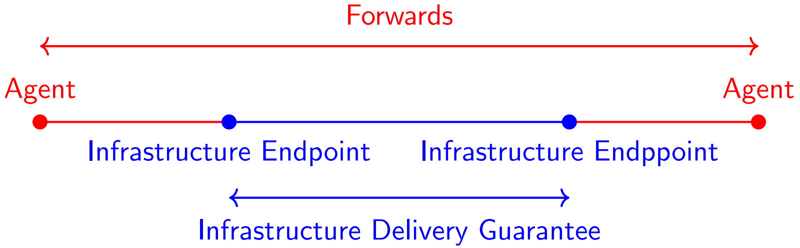
Despite delivery guarantees, Patient times out waiting for Fulfilled from Pharmacy. Why?

- Pharmacy sat on the Prescription!

Need Application-Level Mechanisms

Obviate communication service's delivery guarantees

- Patient forwards Complaint to Physician as a reminder
- Physician forwards Prescription to Pharmacy as a reminder and to Patient to show progress
- Patient forwards Prescription to Pharmacy



Contributions

- Idea of fault as failure of communication expectation as defined in application-specific interaction protocol
- Programming model and annotation-based patterns for engineering fault-tolerant multiagent systems without relying on communication service guarantees
  - Our stuff works over UDP
- A path to realizing decentralized systems in accordance with the end-to-end principle

The *Prescription* Protocol

```
Prescription {
  roles Patient, Physician, Pharmacist
  parameters out sID key, out symptom, out done

  Patient ↦ Physician: Complaint[out sID key, out symptom]
  Physician ↦ Patient: Reassurance[in sID key, in symptom, nil Rx, out done]
  Physician ↦ Pharmacist: Prescription[in sID key, in symptom, nil done, out Rx]
  Pharmacist ↦ Patient: Fulfilled[in sID key, in Rx, out done]
}
```

Sending Reminders (“Retransmission”)

Application-level: Regardless of whether the original message is lost or simply because the other agent hasn’t yet responded to it

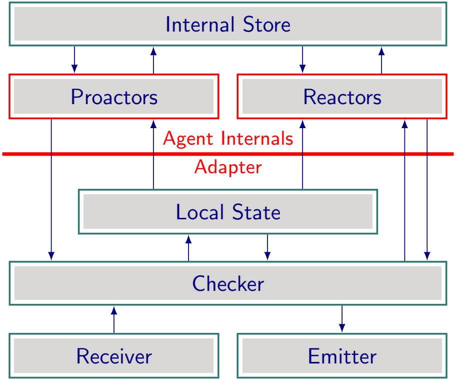
```
@remind
Patient ↦ Physician: Complaint[in sID key, out symptom]

# Annotation means the following message is added to the protocol
Patient ↦ Physician: ComplaintReminder[in sID key, in symptom, out remID key]
```

Forwarding Messages

Sending information a different way to route around failures or accelerate recovery

```
@route via Patient
Physician ↦ Pharmacist: Prescription[in ID key, in symptom, out Rx]

// equivalent to
@forward to Patient
@forward from Patient to Pharmacist
Physician ↦ Pharmacist: Prescription[in ID key, in symptom, out Rx]

// produces
Physician ↦ Patient: FwdPrescription1[in ID key, in symptom, in Rx, out fwID1]
Patient ↦ Pharmacist: FwdPrescription2[in ID key, in symptom, in Rx, out fwID2]
```

Agent Architecture



Recovery Policy

Declarative specification of recovery policies

```
// Patient
- action: remind Physician of Complaint until Reassurance or Prescription or Filled
  when: 0 0 * * * // daily
  max tries: 5
- action: remind Pharmacist of Prescription after 2 days until Filled
  when: 0 0 * * * // daily
  max tries: 5
```

Experimental Results

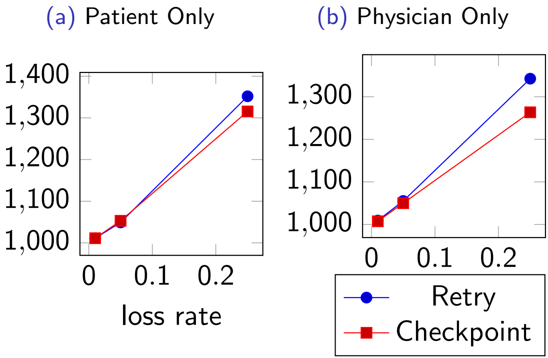


Figure: Messages emitted. Each subfigure represents a different loss configuration, with the lines representing the two recovery policies. Subfigure (a) has no loss, (b-c) have one lossy agent. The Y-axes show the number of messages emitted by PATIENT. The X-axes are the three different loss rates tested: 0.01, 0.05, and 0.25.

Conclusion

- Don't rely on infrastructure alone
- Application-level fault-tolerance is necessary
- Mandrake makes application-level fault tolerance easier to think about and implement

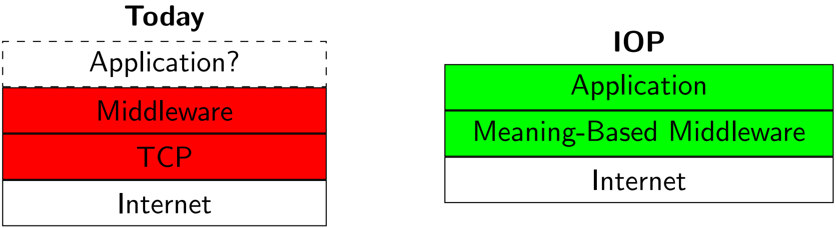
Outline

Decentralization and Norms

- Any application that spans autonomous parties is decentralized
- Decentralization requires modeling interactions
- Norms capture meaning of interaction
- Norms must be represented
  - Crucial to modeling agreements
  - Support compliance checking, trust, and accountability
- Norms are operationalized over protocols

Let's Raise Our Game...to the Application-Level

And Simplify Decentralized Systems!



Flexibility means specify, verify, and implement protocols

- Model a mutliagent system via a protocol
- Focus on message meaning; don't rely on message ordering
- Do application-level fault tolerance (not optional)
- Use programming models to implement protocols

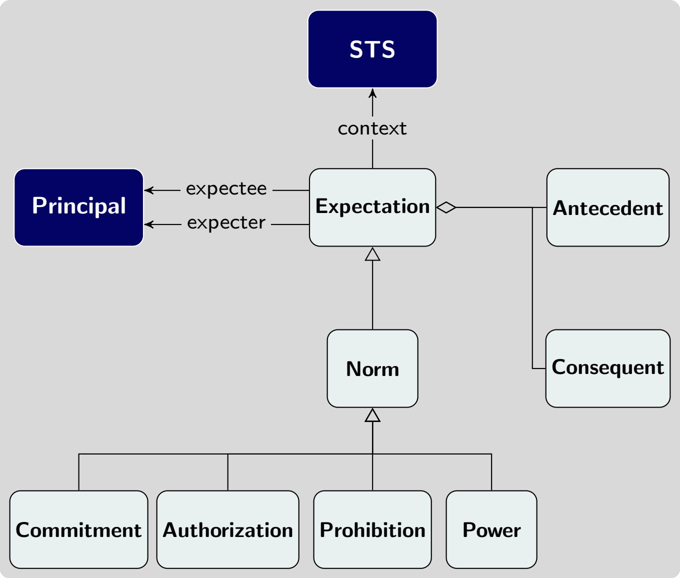
Exercise: Collective Concept Map

- What theme do you remember most from today?
- What connections did you make with stuff you already knew?
- What additional high-level themes should we consider within
  - Software engineering?
  - Programming languages?
  - Artificial intelligence?
  - Distributed computing?
- What directions are worth pursuing with the aim of promoting a deeper understanding between Interaction Orientation, Distributed Computing, and Agentic?

Thanks!

- National Science Foundation
- EPSRC
- Science of Security Lablet
- Consortium for Ocean Leadership

Approach: Specify Social Expectations as Norms



Norms: Specify Directed Expectations between Roles

Specify the social architecture of an STS

Kinds	**Accountable (Expectee)**	**Privileged (Expecter)**
Commitment	Debtor	Creditor
Prohibition	Prohibitee	Prohibiter
Authorization	Authorizer	Authorizee
Power	Empowering	Empowered

Norms Kinds: Canonical Lifecycles

Captures evolutions of an *instance* of the kind

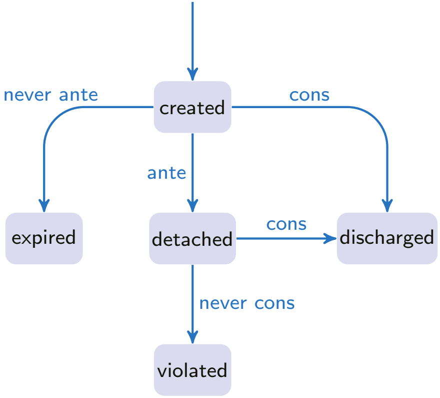
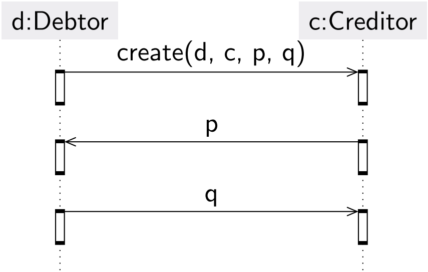


Figure: Commitment lifecycle

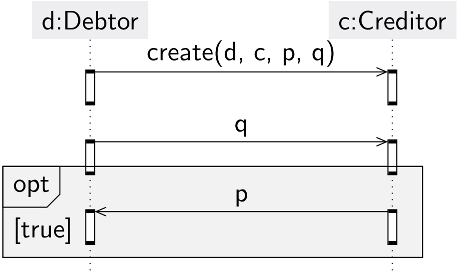
Operationalizing Commitments: Detach then Discharge

C(debtor, creditor, antecedent, consequent)



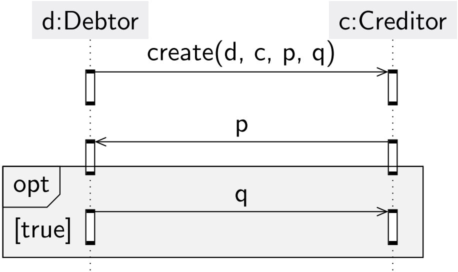
Operationalizing Commitments: Discharge First; Optional Detach

How about this?



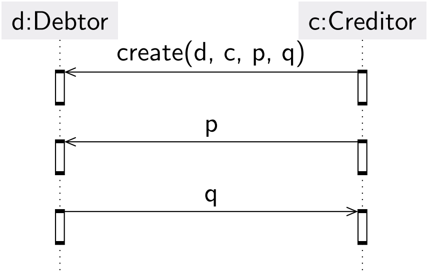
Operationalizing Commitments: Detach First; Optional Discharge

How about this?



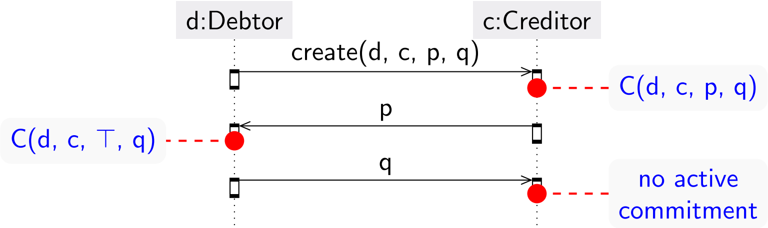
Operationalizing Commitments: Creation by Creditor

C(debtor, creditor, antecedent, consequent)



Operationalizing Commitments: Strengthening by Creditor

C(debtor, creditor, antecedent, consequent)



Norm Specifications as Information Schemas

Technical motivation: Tracking norm instances in information stores

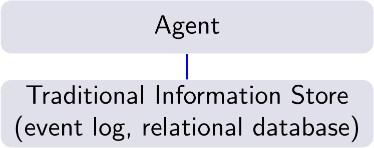


Figure: Existing approaches

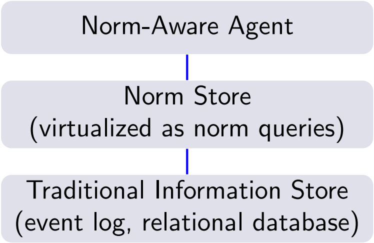


Figure: Cupid

An Information Model and Commitment Specification

E-commerce setting

```
Quote(S, B, ID, item, uPrice, t) with key ID
Accept(B, S, ID, qty, addr, t)  with  key ID
Payment(B, S, ID, pPrice, t)  with  key ID
Shipment(S, B, ID, addr, t) with key ID
Refund(S, B, ID, rAmount, t) with key ID
```

```
commitment DiscountQuote S to B
 create Quote
 detach Accept[, Quote + 4] and Payment[, Quote + 4]
  where pPrice >= 0.9 * uPrice * qty
 discharge Shipment[, detached DiscountQuote + 4]
```

Canonical Queries for DiscountQuote

- *query-name*(*create-clause*, *detach-clause*, *discharge-clause*)
- Queries for created, detached, expired, discharged, and violated commitment instances
- Implementation produces SQL
- Generated SQL long and complicated; near impossible to write manually
  - violated DiscountQuote is 413 lines long
  - The five queries amount to 1060 lines

Query Results

For times up to 16 June 2020

Quote

ID	item	uPrice	t
T1	fig	1	1 June 2020
T2	pear	1	1 June 2020

Accept

ID	qty	addr	t
T1	1	Lancaster	2 June 2020
T2	1	Raleigh	2 June 2020

Payment

ID	pPrice	t
T1	1	2 June 2020
T2	1	2 June 2020

Shipment

ID	addr	t
T1	Lancaster	3 June 2020

discharged

ID	t
T1	3 June 2020

violated

ID	t
T2	7 June 2020

Example: Compensation

Nested Commitment

```
commitment Compensation S to B
  create Quote
  detach violated(DiscountQuote)
  discharge Refund[,violated(DiscountQuote) + 9] where rAmount =
    pPrice
```

Semantics in Relational Algebra

For a base event E, $[\![E]\!]$ equals its materialized relation. The semantics lifts $[\![\ ]\!]$ to all expressions. A few given below to illustrate the style.

D_1. $[\![E[g,h]]\!] = \sigma_{g \leqslant t < h}[\![E]\!]$. Select all events in E that occur after (including at) g but before h.

D_2. $[\![X \sqcap Y]\!] = \sigma_{t \geq t'}([\![X]\!] \bowtie \rho_{t/t'}[\![Y]\!]) \cup \sigma_{t' < t}(\rho_{t/t'}[\![X]\!] \bowtie [\![Y]\!])$. Select (X, Y) pairs where both have occurred; the timestamp of this composite event is the greater of the two.

D_3. $[\![\text{created}(c, r, u)]\!] = [\![c]\!]$. A commitment is created when its create event occurs.

D_4. $[\![\text{violated}(c, r, u)]\!] = [\![(c \sqcap r) \ominus u]\!]$. A commitment is violated when it has been created and detached but not discharged within the specified interval.

Bellifemine, F. L., Caire, G., and Greenwood, D. (2007).
Developing Multi-Agent Systems with JADE.
Wiley-Blackwell.

Bergenti, F., Iotti, E., Monica, S., and Poggi, A. (2017).
Agent-oriented model-driven development for JADE with the JADEL programming language.
Computer Languages, Systems & Structures, 50:142–158.

Boissier, O., Bordini, R. H., Hübner, J. F., Ricci, A., and Santi, A. (2013).
Multi-agent oriented programming with JaCaMo.
Science of Computer Programming, 78(6):747–761.

Rodriguez, S., Gaud, N., and Galland, S. (2014).
Sarl: A general-purpose agent-oriented programming language.
In *2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*, volume 3, pages 103–110.

Singh, M. P. (2011).
Information-driven interaction-oriented programming: BSPL, the Blindingly Simple Protocol Language.
In *Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 491–498, Taipei. IFAAMAS.

Vieira, R., Moreira, Á. F., Wooldridge, M. J., and Bordini, R. H. (2007).
On the formal semantics of speech-act based communication in an agent-oriented programming language.
Journal of Artificial Intelligence Research (JAIR), 29:221–267.

Winikoff, M. (2012).
Challenges and directions for engineering multi-agent systems.
CoRR, abs/1209.1428.