Decentralized Multiagent Systems

Tutorial

Amit K. Chopra and Munindar P. Singh

University of Lancaster
North Carolina State University

2015
Outline

Specification Approaches

LoST: Local State Transfer

BSPL, the Blindingly Simple Protocol Language

Commitments

Summary and Directions
Outline

Specification Approaches
  Message Sequence Diagrams
  Protocols and Policies

LoST: Local State Transfer

BSPL, the Blindingly Simple Protocol Language

Commitments

Summary and Directions
Sequence Diagrams
Well-known specification approach

- Originally used for object-oriented programming
- Our needs: closest to message sequence charts
- An intuitive way to express interactions
  - Expresses global view consolidating local perspectives
  - Excellent for describing possible interaction instances
  - But beware the pitfalls . . .
- Support (potential) validation checks
  - Formalizing semantics is not obvious: multiple approaches
- Standardized in UML 2.0 as Sequence Diagrams
  - Caveat: Arrowheads and other details of these notes don’t necessarily match UML
Method Invocation in Object-Oriented Programming

Only one thread of control; objects exchange messages

c:Customer

getTotal()

total

p:Portfolio

getBalance()

balance

a:Account
Message Emission and Reception

Independent threads of control; autonomous parties exchange messages, asynchronously sending and receiving

Request for Quote
The Alternative Block

Nondeterministically choose and execute any fragment whose guard is true

- Provide Quote
- Accept Quote
- Reject Quote
The Optional Block

Modeling error here: Showing internal detail \((\text{free (spare time)})\) in a protocol

\[ \text{c:Customer} \quad \text{m:Merchant} \]

\[ \text{Provide Goods} \quad \leftarrow \]

\[ \text{Pay Charges} \quad \rightarrow \]

\[ \text{opt} \quad \text{[free]} \quad \text{Submit Comments} \]
The Loop Block

Usually bounded in our examples

\[
\text{c:Customer} \quad \text{m:Merchant}
\]

- Provide Goods
- Pay Charges
- Offer
- Counter Offer

\[\text{loop} \quad [5 \text{ times}]\]
Purchase (Just the Happy Path)

Notice the hand off pattern, indicative of delegation

- c:Customer
- m:Merchant
- s:Shipper

1. Request for Quotes
2. Quote
3. Accept
4. Ship
5. Deliver
The Parallel Block

- c: Customer
- m: Merchant
- b: Bank

1. Provide Goods
2. Pay Charges
3. Deliver Goods
4. Request Payment
Exercise: Diagramming Precedence

- Four roles: A, B, C, D (could map to the same parties)
- Two messages: \( m_{AB} \) and \( m_{CD} \) (sender to receiver: distinct parties)
- We would like to assert that \( m_{AB} \) precedes \( m_{CD} \)
All Possible Sequence Diagrams

Given messages from $a$ to $b$ and from $c$ to $d$

```
\[ a \neq b \]
\[ c \neq d \]
\[ a = c \]
\[ b = d \]
\[ b \neq d \]
\[ b = c \]
\[ b \neq c \]
\[ b = d \]
\[ a = d \]
\[ b \neq c \]
```
Exercise: Which of the Precedence Diagrams are Compatible with Asynchrony?

Invariant outcomes regardless of relative execution speed, communication delays, and no global clock
Exercise: Diagramming Occurrence and Exclusion

Use guards that refer to message occurrence
If \([m_{AB}]\) occurs then so does \([m_{CD}]\)

- Four roles: \(A, B, C, D\) (could map to the same parties)
- Two messages: \(m_{AB}\) and \(m_{CD}\) (sender to receiver)
- We would like to assert that
  - \(m_{AB}\) excludes \(m_{CD}\)
  - \(m_{AB}\) and \(m_{CD}\) mutually exclude each other
  - \(m_{AB}\) requires \(m_{CD}\)
Properties of a (Point-to-Point) Message Channel

Consider these questions

**Noncreative:** Must a message that is received have been sent before?
- Can we take a system snapshot that violates this property?

**Reliable:** Must a message that is sent be received?
- Can we take a system snapshot that violates this property?

**Ordered:** Must the messages received from the same sender be received in the order in which they were sent?
- In which direction does the information flow?

**Causal:** Must the messages received from different senders be received in the order in which they were sent?
- Can we take a system snapshot that violates this property?
Challenges to Correctness of Protocols

Not specific to sequence diagrams

**Distribution:** different parties observe different messages, i.e., each lacks remote knowledge

**Asynchrony:** different parties observe messages in inconsistent orders
  - Despite FIFO channels

**Intuitions about correctness**
  - If each party interacts correctly, is the overall behavior correct?
  - If not, our sequence diagram is not *realizable* or *enactable*
  - Is the design of each party obvious?
  - Does the design of the parties preclude some legal enactments?
Outline

Specification Approaches
  Message Sequence Diagrams
  Protocols and Policies

LoST: Local State Transfer

BSPL, the Blindingly Simple Protocol Language

Commitments

Summary and Directions
Business Protocols

Interactions among autonomous parties, understood at the business level

▶ Conversation: An instance of a protocol
▶ Operational representations: steps taken
  ▶ Procedural
    ▶ Sequence diagrams
    ▶ State diagrams
    ▶ Activity diagrams
    ▶ Petri Nets
  ▶ Declarative
    ▶ Temporal logic
    ▶ Dynamic logic
    ▶ Information-based specifications
▶ Meaning-based representations: underlying business transaction
  ▶ Declarative, if captured formally at all
    ▶ Commitment machines
    ▶ Constitutive specifications
Exercise: Identify the Public and Private Components

Process = Protocol + Policies

- Request for Quotes
- Quote
- Accept
- Ship
- Deliver
Exercise: How Might we Modularize Protocols?
Consider Purchase
Modular Business Protocols

- Identify small, well-defined interactions with clear business meanings
- Improve flexibility and concurrency
- Possibly lead to invalid executions
- How can we ensure good properties despite modularity?
  - Begin from a constraint language
  - Standardize modular fragments as patterns, e.g., RosettaNet
Sequence Diagrams for Business Modeling
No!

- No internal reasoning
  - No private predicates in guards
- No method calls
  - No self calls
- No synchronous messages
  - No business puts itself on indefinite hold waiting for its partner to proceed
- No causally invalid expectations
  - No nonlocal choice
    - No nonlocal choice that matters
  - No control of incoming message occurrence or ordering
  - No dependence on occurrence or ordering of remote message emission or reception
  - No reliance on ordering across channels
    - No reliance on ordering within a channel unless warranted
Outline

Specification Approaches

LoST: Local State Transfer

BSPL, the Blindingly Simple Protocol Language

Commitments

Summary and Directions
Outline

Specification Approaches

LoST: Local State Transfer

BSPL, the Blindingly Simple Protocol Language

Commitments

Summary and Directions
Interactions and Protocols
Distributed systems of autonomous, heterogeneous agents

- Enable interoperation
- Maintain independence from internal reasoning (policies)
- Support composition of distributed systems
Properties of Participants

- Autonomy
- Myopia
  - All choices must be local
  - Correctness should not rely on future interactions
- Heterogeneity: local $\neq$ internal
  - Local state (projection of global, 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
Traditional Specifications

Low-level, procedural approaches leading to over-specified protocols

- 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
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)

- ▶ \(\text{in}\): 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

- ▶ \(\text{out}\): Information that is generated by the protocol instances
  - Bindings can be fed into other protocols through their \(\text{in}\) parameters, thereby accomplishing composition
  - A standalone protocol must adorn all its public parameters \(\text{out}\)

- ▶ \(\text{nil}\): Information that is absent from the protocol instance
  - Bindings must not exist
The **Hello Protocol**

```plaintext
Hello { 
  role Self, Other 
  parameter out greeting key

  Self ↦ Other: hi[out greeting key]
}
```

- At most one instance of `Hello` for each greeting
- At most one `hi` message for each greeting
- Enactable standalone: no parameter is ⌜ in ⌝
- The key of `hi` is explicit; often left implicit on messages
The *Pay* Protocol

Pay \{  
  role Payer, Payee  
  parameter in ID key, in amount  
  
  Payer \mapsto Payee: \text{payM}[\text{in ID, in amount}]  
\}

- At most one \text{payM} for each ID
- Not enactable standalone: \textbf{why}?
- The key of \text{payM} is implicit; could be made explicit
- Eliding `means` clauses in this paper
The Offer Protocol

Offer {
role Buyer, Seller
parameter in ID key, out item, out price

Buyer \rightarrow Seller: rfq [in ID, out item]
Seller \rightarrow Buyer: quote [in ID, in item, out price]
}

- The key ID uniques instances of Initiate Offer, rfq, and quote
- Not enactable standalone: at least one parameter is \( \lnot \text{in} \)
- An instance of rfq must precede any instance of quote with the same ID: why?
- No message need occur: why?
- quote must occur for Offer to complete: why?
The *Initiate Order* Protocol

Initiate—Order 

role B, S

parameter out ID key, out item, out price, out rID

B $\mapsto$ S: rfq [out ID, out item]
S $\mapsto$ B: quote [in ID, in item, out price]

B $\mapsto$ S: accept [in ID, in item, in price, out rID]
B $\mapsto$ S: reject [in ID, in item, in price, out rID]

- The key ID uniquifies instances of *Order* and each of its messages
- Enactable standalone
- An *rfq* must precede a *quote* with the same ID
- A *quote* must precede an *accept* with the same ID
- A *quote* must precede a *reject* with the same ID
- An *accept* and a *reject* with the same ID cannot both occur: *why*?
The *Purchase* Protocol

\[
\text{Purchase } \{ \\
\quad \text{role } B, S, \text{ Shipper} \\
\quad \text{parameter out ID key, out item, out price, out outcome} \\
\quad \text{private address, resp} \\
\]

\[
B \mapsto S: \text{rfq [out ID, out item]} \\
S \mapsto B: \text{quote [in ID, in item, out price]} \\
B \mapsto S: \text{accept [in ID, in item, in price, out address, out resp]} \\
B \mapsto S: \text{reject [in ID, in item, in price, out outcome, out resp]} \\
\]

\[
S \mapsto \text{Shipper: ship [in ID, in item, in address]} \\
\text{Shipper} \mapsto B: \text{deliver [in ID, in item, in address, out outcome]} \\
\}

- At most one item, price, and outcome binding per ID
- Enactable standalone
- *reject* conflicts with *accept* on response (a *private* parameter)
- *reject* or *deliver* must occur for completion (to bind outcome)
Possible Enactment as a Vector of Local Histories

Buyer

rfq

↓

ID, item

quote

↓

ID, price

accept

↓

ID, address

Shipper

rfq

quote

accept

ship

ID, item, address

deliver

↓

ID, item, address, outcome

Chopra & Singh (Lancaster & NCSU) Decentralized Multiagent Systems 2015 37
Knowledge and Viability

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

- **Sender’s View**
  - Knows
  - Does not know

  ![Diagram of Sender's View]

- **Receiver’s View**
  - Knows
  - Does not know

  ![Diagram of Receiver's View]

- 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
Realizing BSPL via LoST

LoST = Local State Transfer

- Does not assume FIFO or reliable messaging
- Provides
  - 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
- Send any unique message provided
  - Parameter bindings agree with previous bindings for the same keys for \(\text{in}\) parameters
  - No bindings for \(\text{out}\) and \(\text{nil}\) parameters exist
Comparing LoST and WS-CDL

- **Similarity:** both emphasize interaction
- **Differences:** WS-CDL is
  - **Procedural**
    - Explicit constructs for ordering
    - Sequential message ordering by default
  - **Agent-oriented**
    - Includes agents’ internal reasoning within choreography (specify what service an agent executes upon receiving a message)
    - Relies on agents’ internal decision-making to achieve composition (take a value from Chor A and send it in Chor B)
  - No semantic notion of completeness
## Comparing LoST and ReST

<table>
<thead>
<tr>
<th></th>
<th>ReST</th>
<th>LoST</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Modality</strong></td>
<td>Two-party; client-server; synchronous</td>
<td>Multiparty interactions; peer-to-peer; asynchronous</td>
</tr>
<tr>
<td><strong>Computation</strong></td>
<td>Server computes definitive resource state</td>
<td>Each party computes its definitive local state and the parties collaboratively and (potentially implicitly) compute the definitive interaction state</td>
</tr>
<tr>
<td><strong>State</strong></td>
<td>Server maintains no client state</td>
<td>Each party maintains its local state and, implicitly, the relevant components of the states of other parties from which there is a chain of messages to this party</td>
</tr>
</tbody>
</table>
## Comparing LoST and ReST

<table>
<thead>
<tr>
<th></th>
<th>ReST</th>
<th>LoST</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Transfer</strong></td>
<td>State of a resource, suitably represented</td>
<td>Local state of an interaction via parameter bindings, suitably represented</td>
</tr>
<tr>
<td><strong>Idempotent</strong></td>
<td>For some verbs, especially GET</td>
<td>Always; repetitions are guaranteed harmless</td>
</tr>
<tr>
<td><strong>Caching</strong></td>
<td>Programmer can specify if cacheable</td>
<td>Always cacheable</td>
</tr>
<tr>
<td><strong>Uniform interface</strong></td>
<td>GET, POST, ...</td>
<td>⌜in⌝, ⌜out⌝, ⌜nil⌝</td>
</tr>
<tr>
<td><strong>Naming</strong></td>
<td>Of resources via URLs</td>
<td>Of interactions via (composite) keys, whose bindings could be URLs</td>
</tr>
</tbody>
</table>
Benefits

- Technical
  - Statelessness
  - Consistency
  - Atomicity
  - Natural composition

- Conceptual
  - Make protocol designer responsible for specifying causality
  - Avoid contortions of traditional approaches when causality is unclear
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
Send-Receive and Send-Send Constraints on a Role

Considering two or more schemas with the same parameter

<table>
<thead>
<tr>
<th></th>
<th>Sends in</th>
<th>Sends out</th>
<th>Sends nil</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sends in</td>
<td>Unconstrained</td>
<td>Send out first</td>
<td>Send nil first</td>
</tr>
<tr>
<td>Sends out</td>
<td></td>
<td>Send at most one</td>
<td></td>
</tr>
<tr>
<td>Send nil</td>
<td>Receive at least</td>
<td>Receive may occur after send</td>
<td>Unconstrained</td>
</tr>
<tr>
<td>Receives in</td>
<td>one instance before send</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receives out</td>
<td>Receive at least</td>
<td>Impossible</td>
<td>Send before receive</td>
</tr>
<tr>
<td></td>
<td>one instance before send</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receives nil</td>
<td>Unconstrained</td>
<td>Unconstrained</td>
<td>Unconstrained</td>
</tr>
</tbody>
</table>
### Summarizing Approaches for Interaction

<table>
<thead>
<tr>
<th></th>
<th>Declarative (Explicit)</th>
<th>Procedural (Implicit)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Meaning</strong></td>
<td>Commitments and other norms</td>
<td>Hard coded within internal reasoning heuristics</td>
</tr>
<tr>
<td><strong>Operation</strong></td>
<td>Temporal logic BSPL</td>
<td>State machines; Petri nets; process algebras</td>
</tr>
</tbody>
</table>

- Declarative approaches for meaning
  - Improve flexibility
  - Under-specify enactment: potential of interoperability failures
- Procedural or declarative approaches for operations
  - Operationally clear, but
    - Tend to emphasize control flow
    - Tend to over-specify operational constraints
    - Yield nontrivial interoperability and endpoint projections
Well-Formedness Conditions

- A parameter that is adorned \texttt{in} in a declaration must be \texttt{in} throughout its body
- A parameter that is adorned \texttt{out} in a declaration must be \texttt{out} in at least one reference
  - When adorned \texttt{out} in zero references, not enactable
  - When adorned \texttt{out} in exactly one reference, consistency is guaranteed
  - When adorned \texttt{out} in two or more references, no more than one can execute
- A private parameter must be \texttt{out} in at least one reference and \texttt{in} in at least one reference
New Contributions
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
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
  - Must have bindings for some parameters
  - Analogous to NOT NULL constraints

- Immutability
  - Durability
  - Robustness: insensitivity to
    - Reordering by infrastructure
    - Retransmission: one delivery is all it needs
Advanced Topics
Safety: **Purchase Unsafe**

Remove conflict between *accept* and *reject*

\[
\text{Purchase Unsafe } \{
\begin{align*}
\text{role } & \text{ B, S, Shipper} \\
\text{parameter} & \text{ out ID key, out item, out price, out outcome} \\
\text{private address, resp} \\
\text{B} & \mapsto \text{ S: rfq [out ID, out item]} \\
\text{S} & \mapsto \text{ B: quote [in ID, in item, out price]} \\
\text{B} & \mapsto \text{ S: accept [in ID, in item, in price, out address]} \\
\text{B} & \mapsto \text{ S: reject [in ID, in item, in price, out outcome]} \\
\text{S} & \mapsto \text{ Shipper: ship [in ID, in item, in address]} \\
\text{Shipper} & \mapsto \text{ B: deliver [in ID, in item, in address, out outcome]} \\
\end{align*}
\]

- B can send both *accept* and *reject*
- Thus outcome can be bound twice in the same enactment
Liveness: *Purchase No Ship*

Omit *ship*

\[ \text{Purchase No Ship} \{
\text{role B, S, Shipper}
\text{parameter out ID key, out item, out price, out outcome}
\text{private address, resp}
\]

\[
\text{B } \rightarrow \text{ S: rfq [out ID, out item]}
\]

\[
\text{S } \rightarrow \text{ B: quote [in ID, in item, out price]}
\]

\[
\text{B } \rightarrow \text{ S: accept [in ID, in item, in price, out address, out resp]}
\]

\[
\text{B } \rightarrow \text{ S: reject [in ID, in item, in price, out outcome, out resp]}
\]

\[
\text{Shipper } \rightarrow \text{ B: deliver [in ID, in item, in address, out outcome]}
\]

- If B sends *reject*, the enactment completes
- If B sends *accept*, the enactment deadlocks
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 ∧ 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 $\land$ the occurrence is unsatisfiable
in-out Polymorphism

price could be \texttt{\textless in\textgreater} or \texttt{\textless out\textgreater}

\begin{verbatim}
Flexible-Offer { 
role B, S
parameter in ID key, out item, price, out qID

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

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

- The price can be adorned \texttt{\textless in\textgreater} or \texttt{\textless out\textgreater} in a reference to this protocol
\end{verbatim}
The *Bilateral Price Discovery* protocol

BPD {
    role Taker, Maker
    parameter out reqID key, out query, out result

    Taker \rightarrow Maker: priceRequest[out reqID, out query]
    Maker \rightarrow Taker: priceResponse[in reqID, in query, out result]
}
The *Generalized Bilateral Price Discovery* protocol

GBPD  

role T, M  

parameter reqID, key, query, res

\[ T \rightarrow M: \text{priceRequest}[\text{out reqID, out query}] \]
\[ T \rightarrow M: \text{priceRequest}[\text{in reqID, in query}] \]

\[ M \rightarrow T: \text{priceResponse}[\text{in reqID, in query, out res}] \]
\[ M \rightarrow T: \text{priceResponse}[\text{in reqID, in query, in res}] \]
The **Multilateral Price Discovery** protocol

**MPD**

- **role** Taker, Exchange, Maker
- **parameter** out reqID key, out query, out res

GBPD(Taker, Exchange, out reqID, out query, in res)
GBPD(Exchange, Maker, in reqID, in query, out res)


Standing Order

As in insurance claims processing

\textbf{Insurance - Claims} {} \\
\begin{verbatim}
role Vendor, Subscriber 
parameter out policyNO key, out reqForClaim key, out claimResponse 

Vendor \rightarrow Subscriber: createPolicy [out policyNO] 
Subscriber \rightarrow Vendor: serviceReq [in policyNO, out reqForClaim] 
Vendor \rightarrow Subscriber: claimService [in policyNO, in reqForClaim, out claimResponse] 
\end{verbatim}

\begin{itemize}
\item Each claim corresponds to a unique policy and has a unique response
\item One policy may have multiple claims
\item Could make \{policyNO, reqForClaim\} both key
\end{itemize}
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
Shopping Cart

Shopping Cart {
role B, S
parameter out ID key, out lineID key, out item, out qty, out price, out finalize

B → S: create[out ID]
S → B: quote[in ID, out lineID, in item, out price]
B → S: add[in ID, in lineID, in item, out qty, in price]
B → S: remove[in ID, in lineID]

S → B: total[in ID, out sum]
B → S: settle[in ID, in sum, out finalize]
B → S: discard[in ID, out finalize]
}
Exercise 1: Abruptly Cancel

Solution added

```
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? **No**
▶ Is this protocol live? **Yes**
Exercise 2: *Abruptly Cancel Modified (with \texttt{nil})*

Solution added

\begin{verbatim}
Abruptly Cancel {
role B, S
parameter out ID key, out item, out outcome

B \rightarrow S: order [out ID, out item]
B \rightarrow S: cancel [in ID, in item, nil outcome]
S \rightarrow B: goods [in ID, in item, out outcome]
}
\end{verbatim}

- Is this protocol safe? **Yes**
- Is this protocol live? **Yes**
  - But it lacks business realism because the \texttt{SELLER} may send \textit{goods} even after receiving \textit{cancel}
The *Bid Offer* protocol

**Bid Offer** 

role Coordinator uni, Bidder ⇋ Winner uni

parameter out ID key, out request, out response, out decision

Coordinator ↦ Bidder: \( \text{CfB}[\text{out ID, out request}] \)

Bidder ↦ Coordinator: \( \text{bid}[\text{in ID, in request, out response}] \)

Coordinator ↦ Winner: \( \text{offer}[\text{in ID, in request, in response, out decision}] \)

}
The **Generalized Bilateral Price Discovery** protocol

- Like **Bilateral Price Discovery** but supports both \( \text{in} \) and \( \text{out} \) adornments on parameters
Obtaining *BPD* from *GBPD*

- May further remove superfluous messages, such as
  - The two messages with all \(\text{in}\) parameters
**GBPDR Restricted**

To parameter adornments of `⌜out⌝`, `⌜out⌝`, and `⌜in⌝`, respectively.

<table>
<thead>
<tr>
<th>GBPDR-out-out-in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taker</td>
</tr>
<tr>
<td>priceRequest</td>
</tr>
<tr>
<td>Taker</td>
</tr>
<tr>
<td>req ID</td>
</tr>
<tr>
<td>req ID</td>
</tr>
<tr>
<td>priceResponse</td>
</tr>
<tr>
<td>Maker</td>
</tr>
<tr>
<td>req ID</td>
</tr>
</tbody>
</table>

- Removing the reference whose adornments are incompatible with those stated.
**GBPDR Restricted**

To parameter adornments of ⌜in⌝, ⌜in⌝, and ⌜out⌝ respectively

- Removing the reference whose adornments are incompatible with those stated
Multilateral Bilateral Price Discovery from GBPD

- For specification, does not violate encapsulation
- For enactment, treats each copy of GBPD as a macro
The Original *NetBill* Protocol

Rigid: supports only one sequential enactment

```plaintext
NetBill Original {
role C, M
parameter out ID key, out item, out price, out done
private confirmation, document, payment

C \rightarrow M: rfq [out ID, out item]
M \rightarrow C: offer [in ID, in item, out price]
C \rightarrow M: accept [in ID, in item, in price, out confirmation]
M \rightarrow C: goods [in ID, in item, in confirmation, out document]
C \rightarrow M: pay [in ID, in price, in document, out payment]
M \rightarrow C: receipt [in ID, in item, in payment, out done]
}
```
Bliss Conceptual Model: Functions of Parameters

- **Key**
  - For interaction instantiation and uniqueness
- **Payload**
  - For interaction meaning
- **Completion**
  - To help determine when the interaction is over
- **Integrity**
  - For interaction integrity
- **Control**
  - To force certain preferred orders of enactment
Bliss Methodology
Iterate over the following steps

1. Identify the roles needed in a protocol
2. Identify the conceptual social object computed
3. Identify the messages (or, recursively, subprotocols) to compute the social object
4. Identify each message as a component of the social object and any additional constraints
5. Introduce polymorphism of messages to support flexible sourcing of parameter bindings
Conceptual Schema for NetBill
**NetBill Via Bliss (Partial)**

Multiple enactments

\[
\text{NetBill Bliss Simple \{ \\
\text{role C, M} \\
\text{parameter out ID key, out item, out price, out done} \\
\text{private confirmation, document, payment} \\
\text{C} \leftrightarrow M: \text{rfq [out ID, out item] } \\
\text{M} \leftrightarrow C: \text{offer [in ID, in item, out price]} \\
\text{M} \leftrightarrow C: \text{offer [out ID, out item, out price]} \\
\text{C} \leftrightarrow M: \text{accept [in ID, in item, in price, out confirmation]} \\
\text{C} \leftrightarrow M: \text{accept [out ID, out item, out price, out confirmation]} \\
\text{M} \leftrightarrow C: \text{goods [in ID, in item, in confirmation, out document]} \\
\text{M} \leftrightarrow C: \text{goods [in ID, in item, nil confirmation, out document]} \\
\text{C} \leftrightarrow M: \text{pay [in ID, in price, in document, out payment]} \\
\text{C} \leftrightarrow M: \text{pay [in ID, in price, nil document, out payment]} \\
\text{M} \leftrightarrow C: \text{receipt [in ID, in item, in payment, out done]} \\
\}}
\]
Schema for Cyberinfrastructure Resource Sharing

Maps to four protocols, naturally composed

BSPL, the Blindingly Simple Protocol Language

Chopra & Singh (Lancaster & NCSU)  Decentralized Multiagent Systems
Service Request Protocol
(Erroneous: Unsafe)
BSPL Reconstruction of Unsafe Service Request

Combining some parameters to reduce clutter

```plaintext
protocol OOI Service Request Unsafe {
role R, P
parameter out ID key, out operation, out result
private confirmation

R → P: request [out ID, out operation]
P → R: accept [in ID, out confirmation]
P → R: reject [in ID, out confirmation, out result]
R → P: cancel [in ID, out result]
P → R: fail [in ID, out result]
P → R: answer [in ID, out result]
}
```
A Conceptual Schema for Service Request

- Requester
  - rID
  - ID, operation, done
  - clID
  - mID

- Provider
  - pID

Options for Requester:
- request
- accept
- reject
- forgetIt
- answer
- fail
- released
The *Service Request* Protocol Via Bliss, Now Corrected

```
protocol OOI Service Request Corrected {
    role R, P
    parameter out ID key, out operation, out result
    private confirmation, releaseToken

    R \rightarrow P: request[out ID, out operation]
    P \rightarrow R: accept[in ID, in operation, out confirmation]
    P \rightarrow R: reject[in ID, in operation, out confirmation, out result]
    R \rightarrow P: forgetIt[in ID, in operation, in confirmation, out releaseToken]
    P \rightarrow R: answer[in ID, in operation, in confirmation, nil releaseToken, out result]
    P \rightarrow R: fail[in ID, in operation, in confirmation, nil releaseToken, out result]
    P \rightarrow R: released[in ID, in operation, in releaseToken, out result]
}
```
Outline

Specification Approaches

LoST: Local State Transfer

BSPL, the Blindingly Simple Protocol Language

Commitments

Summary and Directions
Commitments as Elements of a Contract

A kind of normative relationship: Express meanings of interactions

- Are atoms of contractual relationships
- Enable correctness checking of contracts
- Yield precise meanings and verifiability
Example: Commitment Progression
Via explicit operations or because of logical properties

C(Buyer, Seller, goods, pay): Active and conditional
  ▶ If goods ∧ C(Buyer, Seller, goods, pay) Then
    ▶ Active and detached (or unconditional or base)
    ▶ C(Buyer, Seller, T, pay)
  ▶ If C(Buyer, Seller, T, pay) Then
    ▶ If pay Then Satisfied
    ▶ If never pay Then Violated
  ▶ If C(Buyer, Seller, goods, pay) Then
    ▶ If pay Then Satisfied
    ▶ If never pay and never goods Then Expired

Can be nested:
C(Seller, Buyer, pay, C(Shipper, Buyer, T, deliverGoods))
Operationalizing Commitments: Detach then Discharge

$C(\text{debtor}, \text{creditor}, \text{antecedent}, \text{consequent})$
Operationalizing Commitments: Discharge First; Optional Detach

How about this?

d: Debtor

create(d, c, p, q)

q

opt
true

p

c: Creditor
Operationalizing Commitments: Detach First; Optional Discharge

How about this?

```plaintext
create(d, c, p, q)
```

d:Debtor

opt

[true]

p

c:Creditor

q
Operationalizing Commitments: Creation by Creditor

\[ C(\text{debtor}, \text{creditor}, \text{antecedent}, \text{consequent}) \]
Operationalizing Commitments: Strengthening by Creditor

\[ C(\text{debtor, creditor, antecedent, consequent}) \]

\[ C(d, c, p, q) \]

\[ p \]

\[ q \]

\[ \text{no active commitment} \]
Commitment Life Cycle (and Patterns)

$C(\text{debtor, creditor, antecedent, consequent})$

(a) Commit

(b) Relieve
Commitment Operations

- \texttt{create}(C(d, c, p, q)) establishes the commitment
- \texttt{detach}(C(d, c, p, q)) turns it into a base commitment
- \texttt{discharge}(C(d, c, p, q)) satisfies the commitment
- \texttt{cancel}(C(d, c, p, q)) cancels the commitment
- \texttt{release}(C(d, c, p, q)) releases the debtor from the commitment
- \texttt{delegate}(z, C(d, c, p, q)) replaces \(d\) by \(z\) as the debtor
  - \(d\) remains ultimately responsible (in our work)
- \texttt{assign}(w, C(d, c, p, q)) replaces \(c\) by \(w\) as the creditor
Outline

Specification Approaches

LoST: Local State Transfer

BSPL, the Blindingly Simple Protocol Language

Commitments
  Cupid: Information-Based Commitments
  Calypso: Commitment Alignment

Summary and Directions
Tracking Commitments

- Current approaches (above): Each agent tracks commitments internally via ad hoc reasoning
- Cupid (below): An agent executes canonical commitment queries that map to its information store
Commitment Lifecycle

- Null (N)
  - create
  - antecedent
    - Conditional
      - antecedent_fail
        - Expired
    - consequent
      - Satisfied
  - Detached
    - consequent_fail
      - Violated
Cupid: Language for Specifying Commitments

Based on underlying information model (schema)

- Supports systematic treatment of commitment lifecycle
- Supports a systematic treatment of commitment instances
- Provides features needed for real-world scenarios
  - Deadlines
  - Nested commitments
  - Complex event expressions
An Information Model and Commitment Specification

Quote(mID, cID, qID, itemID, uPrice, t) with key qID
Order(cID, mID, oID, qID, qty, addr, t) with key oID
Payment(cID, mID, pID, oID, pPrice, t) with key pID
Shipment(mID, cID, sID, oID, addr, t) with key sID
Refund(mID, cID, rID, pID, rAmount, t) with key rID
Coupon(cID, mID, uID, oID, rebate, t) with key uID

commitment DiscountQuote mID to cID
create Quote
detach Order and Payment[ , Quote + 10] where pPrice >= 0.9 * uPrice * qty
discharge Shipment[ , Payment + 5]

A DiscountQuote commitment from a merchant to a customer is

- **created** upon Quote;
- **detached** if Order happens and Payment happens within 10 days of Quote and is for at least 90% of quoted amount (else **expires**)
- **discharged** if Shipment happens within five days of Payment (else **violated**)

Example: Compensation

Nested Commitment

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

A Compensation commitment is created upon Quote and says that if DiscountQuote is violated, the merchant will refund the payment within nine days of the violation.
Example: Expedited Shipping Upon Coupon

Absence of information via “except”

commitment ExpeditedWithoutCoupon mID to cID create Quote and Order except Coupon detach Payment[ , Quote + 10] where pPrice >= 0.9 * uPrice * qty discharge Shipment[ , Payment + 1]

An ExpeditedWithoutCoupon commitment is created only if the customer does not introduce a coupon in the transaction; it is discharged if the shipment is expedited (sent within one day of payment instead of five, as in the previous example).
### Table Extensions

Evaluate queries, say, on Feb 15

#### Quote

<table>
<thead>
<tr>
<th>qID</th>
<th>itemID</th>
<th>uPrice</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Posters</td>
<td>10</td>
<td>Jan 27</td>
</tr>
<tr>
<td>Q2</td>
<td>Pens</td>
<td>5</td>
<td>Jan 27</td>
</tr>
</tbody>
</table>

#### Order

<table>
<thead>
<tr>
<th>oID</th>
<th>qID</th>
<th>qty</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>O1</td>
<td>Q1</td>
<td>1</td>
<td>Jan 29</td>
</tr>
<tr>
<td>O2</td>
<td>Q1</td>
<td>1</td>
<td>Jan 29</td>
</tr>
<tr>
<td>O3</td>
<td>Q2</td>
<td>1</td>
<td>Jan 29</td>
</tr>
</tbody>
</table>

#### Payment

<table>
<thead>
<tr>
<th>pID</th>
<th>oID</th>
<th>pPrice</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>O1</td>
<td>10</td>
<td>Feb 2</td>
</tr>
<tr>
<td>P2</td>
<td>O3</td>
<td>5</td>
<td>Feb 2</td>
</tr>
</tbody>
</table>

#### Shipment

<table>
<thead>
<tr>
<th>sID</th>
<th>oID</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>O1</td>
<td>Feb 5</td>
</tr>
<tr>
<td>S2</td>
<td>O3</td>
<td>Feb 9</td>
</tr>
</tbody>
</table>

#### Discharged

<table>
<thead>
<tr>
<th>qID</th>
<th>oID</th>
<th>pID</th>
<th>sID</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>O1</td>
<td>P1</td>
<td>S1</td>
<td>Feb 5</td>
</tr>
</tbody>
</table>

#### Violated

<table>
<thead>
<tr>
<th>qID</th>
<th>oID</th>
<th>pID</th>
<th>sID</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
<td>O3</td>
<td>P2</td>
<td></td>
<td>Feb 8</td>
</tr>
</tbody>
</table>

#### Expired

<table>
<thead>
<tr>
<th>qID</th>
<th>oID</th>
<th>pID</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>O2</td>
<td>P1</td>
<td>Feb 9</td>
</tr>
</tbody>
</table>
Cupid Syntax

Com → commitment(\(A, A, \text{Expr}, \text{Expr}, \text{Expr}\))
Event → Base | Life(\(A, A, \text{Expr}, \text{Expr}, \text{Expr}\))
Life → created | detached | discharged | expired | violated
Expr → Event[T, T] | Expr \(\cap\) Expr | Expr \(\cup\) Expr
          | Expr \(\ominus\) Expr | Expr where \(\varphi\)
T → Event + \(\mathcal{T}\) | \(\mathcal{T}\)
A protocol designer specifies commitments and event schema. From these specifications, Cupid generates canonical queries to compute commitment instances in each lifecycle state.
Semantics in Relational Algebra

A model $M$ of an information schema specifies each event schema $E$’s extension $[E]$ as a set of its instances. For a base event $E$, $[E]$ equals its materialized relation. The semantics lifts $[ ]$ to all expressions. Some important postulates are below; the rest are in the paper.

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

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

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

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

- All Cupid queries are safe
  - Given any possible model $M$ with finite extensions for base events, the extension of $Q$ relative to $M$, $\llbracket Q \rrbracket$, is finite
- A commitment specification is well-identified iff the key of its detach event expression determines the key of its create event expression and the key of its discharged event expression determines the key of either its detach or its create event expressions. That is, there is adequate correlation between the events that affect a commitment.
  - Specifications that are not well-identified are vacuous
- Instances of a finitely expirable commitment specification are guaranteed to expire if not detached within a finite amount of time.
- Instances of a finitely violable commitment specification are guaranteed to expire if not discharged within a finite amount of time.
Outline

Specification Approaches

LoST: Local State Transfer

BSPL, the Blindingly Simple Protocol Language

Commitments

  Cupid: Information-Based Commitments
  Calypso: Commitment Alignment

Summary and Directions
Commitment Alignment

Fundamental notion of interoperability

- Each agent observes messages sequentially
- System state is a vector of observation sequences, one for each agent
- Invariant: In any system state, if a creditor infers an active commitment from its observations, then debtor must infer that commitment as active from its own observations.
Asynchrony Problems

Scenarios (B), (C), and (D) end in misalignment
Identifiers on Commitments are Ineffective

- (A): Using different identifiers for new offers
- (B): A stronger offer crosses the reject of the previous offer
Race Recast in Terms of Commitments

- Offer($12,BNW), Reject($12,BNW), Cancel($12,BNW), and Pay($12) above map to Create(c_B), Release(c_B), Cancel(c_B), and Declare($12), respectively

- Notice how in each of (B), (C), and (D), Alice infers a commitment (either c_B or c_{UB}) that EBook does not
State Machine With Transitions for (B)

Alice and EBook begin from $\neg c_B$, but end up in different states after A.2 and E.2
Problems Due to NonFIFO Message Delivery

In (B) and (C) messages are not delivered on FIFO basis, which causes misalignment.
Transaction-Related Problems

Aligned but unrealistic outcomes due to the lack of transactions
Completeness

In (A) and (B), misaligned at dotted line, but not in (C)
Technical Result

- Formalized updates from messages and proved that a complete state is also aligned
- Gave algorithms for sending messages that guarantee progress toward complete states
Cupid and Calypso

Done independently, but deep connections

- Modeling of identifiers to enable distinguishing instances crucial to both
  - Keys in Cupid and transaction identifier in Calypso
  - Part of the domain model

- Cupid tells us what is in a database; Calypso ensures multiple databases are sufficiently synced up

- Parts of the same puzzle: a commitment-based middleware
  - Cupid for computing commitment states; Calypso for ensuring information propagation
Outline

Specification Approaches

LoST: Local State Transfer

BSPL, the Blindingly Simple Protocol Language

Commitments

Summary and Directions
Summary and Directions

Exercise: Collective concept map
Business Environments

Theme of this course: How is computer science different for open environments?

- **Autonomy**
  - Messaging, not APIs
  - Markets

- **Heterogeneity**
  - Capturing structure of information
  - Transforming structures

- **Dynamism**
  - Partially addressed through above

Support flexibility and arms-length relationships
Directions

- Tool support for specifying practical protocols
- Expansion of the language to handle role hierarchies
- Treatment of recursive protocols
Thanks!

- National Science Foundation
- Consortium for Ocean Leadership

http://www.csc.ncsu.edu/faculty/mpsingh/