Decentralized Multiagent Systems

Tutorial

Amit K. Chopra and Munindar P. Singh

Lancaster University
North Carolina State University

2015
Sociotechnical Systems
Principals are humans or organizations

How to specify interaction and provide software support for it?
Interaction Protocols
Specify interaction in distributed systems

- **Low-level**
  - Specify constraints on messaging
  - Many specification languages, e.g., state machines, UML, BSPL (Singh, 2011)

- **High-level**
  - Specify how expectations among principals change as they interact
  - E.g., commitment protocols
Outline

Specification Approaches

BSPL, the Blindingly Simple Protocol Language

Commitments

Summary and Directions
Outline

Specification Approaches
Message Sequence Diagrams
Protocols and Policies

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()

getBalance()

p:Portfolio

total

a:Account

balance

c:Customer

getTotal()

getBalance()

p:Portfolio

total

balance

a:Account
Message Emission and Reception

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

Request for Quote

C: Customer → M: Merchant
The Alternative Block

Nondeterministically choose and execute any fragment whose guard is true

Chopra & Singh (Lancaster & NCSU)  Decentralized Multiagent Systems  2015
The Optional Block
Modeling error here: Showing internal detail (free (spare time)) in a protocol
The Loop Block

Usually bounded in our examples

```
<table>
<thead>
<tr>
<th>c:Customer</th>
<th>m:Merchant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provide Goods</td>
<td></td>
</tr>
<tr>
<td>Pay Charges</td>
<td></td>
</tr>
<tr>
<td>Offer</td>
<td></td>
</tr>
<tr>
<td>Counter Offer</td>
<td></td>
</tr>
</tbody>
</table>
```

loop [5 times]
Purchase (Just the Happy Path)
Notice the hand off pattern, indicative of delegation
The Parallel Block

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

- Provide Goods
- Pay Charges
- Deliver Goods
- Request Payment

Chopra & Singh (Lancaster & NCSU) Decentralized Multiagent Systems 2015 13
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 $d$
Exercise: Which of the Sequence 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

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 $= \text{Protocol} + \text{Policies}$

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

- 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

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 ≠ 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
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

Hello {  
role Self, Other  
parameter out greeting key  

Self \(\rightarrow\) 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 \(\text{in}\)
- 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 $\rightarrow$ Payee: payM[in ID, in amount] }

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

\[
\begin{align*}
\text{Offer} & \{ \\
& \quad \text{role Buyer, Seller} \\
& \quad \text{parameter in ID key, out item, out price} \\
& \quad \text{Buyer} \leftrightarrow \text{Seller}: \text{rfq}[\text{in ID, out item}] \\
& \quad \text{Seller} \leftrightarrow \text{Buyer}: \text{quote}[\text{in ID, in item, out price}] \\
\}
\end{align*}
\]

- The key ID uniquifies instances of *Initiate Offer*, *rfq*, and *quote*
- Not enactable standalone: at least one parameter is \(\text{⌜in\textup{\textregistered}}\)
- 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 → 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 rID]
    B → 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

```plaintext
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]
}
```

- 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)
BSPL, the Blindingly Simple Protocol Language

Possible Enactment as a Vector of Local Histories

Buyer

rfq

quote

accept

deliver

-----------------ID, item----------------->

------------------ID, price------------------

-------------------ID, address----------------

------------------------ID, item, address------------------------>

------------------------ID, item, address, outcome------------------------>

Seller

rfq

quote

accept

ship

------------------ID, price------------------

-------------------ID, address----------------

------------------------ID, item, address------------------------>

------------------------ID, item, address, outcome------------------------>

Shipper

rfq

quote

accept

ship

------------------ID, price------------------

-------------------ID, address----------------

------------------------ID, item, address------------------------>

------------------------ID, item, address, outcome------------------------>
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

Receiver’s View

- Knows
- Does not know

- Knowledge increases monotonically at each role
- An \(\text{in} \) parameter creates and transmits knowledge
- An \(\text{out} \) 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 in parameters
  - No bindings for out and 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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>collaboratively and (potentially implicitly) compute the definitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>components of the states of other parties from which there is a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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 <code>GET</code></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><code>GET, POST, ...</code></td>
<td><code>⌜in⌟, ⌜out⌟, ⌜nil⌟</code></td>
</tr>
<tr>
<td><strong>Naming</strong></td>
<td>Of resources via URIs</td>
<td>Of interactions via (composite) keys, whose bindings could be URIs</td>
</tr>
</tbody>
</table>
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
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</td>
<td>State machines; Petri nets; process algebras</td>
</tr>
<tr>
<td></td>
<td>BSPL</td>
<td></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 \(\text{⌜in⌟} \) in a declaration must be \(\text{⌜in⌟} \) throughout its body
- A parameter that is adorned \(\text{⌜out⌟} \) in a declaration must be \(\text{⌜out⌟} \) in at least one reference
  - When adorned \(\text{⌜out⌟} \) in zero references, not enactable
  - When adorned \(\text{⌜out⌟} \) in exactly one reference, consistency is guaranteed
  - When adorned \(\text{⌜out⌟} \) in two or more references, no more than one can execute
- A private parameter must be \(\text{⌜out⌟} \) in at least one reference and \(\text{⌜in⌟} \) in at least one reference
BSPL
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
Outline

Specification Approaches

BSPL, the Blindingly Simple Protocol Language
  Advanced Topics
  Bliss

Commitments

Summary and Directions
Safety: *Purchase Unsafe*

Remove conflict between *accept* and *reject*

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

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

  Shipper \rightarrow B: \text{deliver}[\text{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 \( \land \) 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{in} or \texttt{out}

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

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

  S \mapsto B: quote[ID, in item, out price, out qID] 
  S \mapsto B: quote[ID, in item, in price, out qID] 
\}
\end{verbatim}

- The price can be adorned \texttt{in} or \texttt{out} in a reference to this protocol
The \textit{Bilateral Price Discovery} protocol

\begin{verbatim}
BPD {
  role Taker, Maker
  parameter out reqID key, out query, out result

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

\[
\text{GBPД} \{ \\
\quad \text{role } T, \ M \\
\quad \text{parameter } req\text{ID} \text{ key, query, res} \\
\quad T \rightarrow M: \text{priceRequest} [\text{out req\text{ID}, out query}] \\
\quad T \rightarrow M: \text{priceRequest} [\text{in req\text{ID}, in query}] \\
\quad M \rightarrow T: \text{priceResponse} [\text{in req\text{ID}, in query, out res}] \\
\quad M \rightarrow T: \text{priceResponse} [\text{in req\text{ID}, in query, in res}] \\
\} 
\]
The *Multilateral Price Discovery* protocol

\[
\text{MPD } \{
\text{role Taker, Exchange, Maker}
\text{parameter out reqID key, out query, out res}
\text{GBPD(Taker, Exchange, out reqID, out query, in res)}
\text{GBPD(Exchange, Maker, in reqID, in query, out res)}
\}
\]
Standing Order
As in insurance claims processing

Insurance—Claims {
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]
}

- Each claim corresponds to a unique policy and has a unique response
- One policy may have multiple claims
- Could make \{policyNO, reqForClaim\} both key
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

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

B \mapsto S: \text{create}[\text{out ID}] \\
S \mapsto B: \text{quote}[\text{in ID, out lineID, in item, out price}] \\
B \mapsto S: \text{add}[\text{in ID, in lineID, in item, out qty, in price}] \\
B \mapsto S: \text{remove}[\text{in ID, in lineID}] \\
S \mapsto B: \text{total}[\text{in ID, out sum}] \\
B \mapsto S: \text{settle}[\text{in ID, in sum, out finalize}] \\
B \mapsto S: \text{discard}[\text{in ID, out finalize}] \\
\}
\]
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`)

\[
\text{Abruptly Cancel} \{ \\
\text{role } B, S \\
\text{parameter out ID key, out item, out outcome} \\
\}
\]

\[B \leftrightarrow S: \text{order} \ [\text{out ID, out item}]\]

\[B \leftrightarrow S: \text{cancel} \ [\text{in ID, in item, nil outcome}]\]

\[S \leftrightarrow B: \text{goods} \ [\text{in ID, in item, out outcome}]\]

- Is this protocol safe?
- Is this protocol live?
  - But it lacks business realism because the SELLER may send *goods* even after receiving *cancel*
Exercise 3: Goods Priority

- Modify *Abruptly Cancel* so that goods takes priority over cancel, i.e., the SELLER can ignore the cancel message but the BUYER cannot ignore the goods message.
The *Bid Offer* protocol

Bid Offer {
  role Coordinator uni, Bidder ⊑ Winner uni
  parameter out ID key, out request, out response, out decision

  Coordinator ↦ Bidder: CfB[out ID, out request]

  Bidder ↦ Coordinator: bid[in ID, in request, out response]

  Coordinator ↦ Winner: offer[in ID, in request, in response, out decision]
}
The *Generalized Bilateral Price Discovery* protocol

- Like *Bilateral Price Discovery* but supports both \( \preceq \text{in} \preceq \) and \( \preceq \text{out} \preceq \) adornments on parameters
May further remove superfluous messages, such as
- The two messages with all \( \text{in} \) parameters
GBPD Restricted

To parameter adornments of \textsc{out}, \textsc{out}, and \textsc{in}, respectively

▶ Removing the reference whose adornments are incompatible with those stated
**GBPD 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
Outline

Specification Approaches

BSPL, the Blindingly Simple Protocol Language
  Advanced Topics
  Bliss

Commitments

Summary and Directions
The Original NetBill Protocol

Rigid: supports only one sequential enactment

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

  C → M: rfq [out ID, out item]
  M → C: offer [in ID, in item, out price]
  C → M: accept [in ID, in item, in price, out confirmation]
  M → C: goods [in ID, in item, in confirmation, out document]
  C → M: pay [in ID, in price, in document, out payment]
  M → 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

- **Customer**
  - cID
- **NetBill**
  - ID, item, price, done
  - cID
  - mID
- **Merchant**
  - mID

Transactions:
- rfq
- offer
- accept
- goods
- pay
- receipt
NetBill Via Bliss (Partial)

Multiple enactments

NetBill Bliss Simple {
  role C, M
  parameter out ID key, out item, out price, out done
  private confirmation, document, payment

  C ⇔ M: rfq [out ID, out item]

  M ⇔ C: offer [in ID, in item, out price]
  M ⇔ C: offer [out ID, out item, out price]

  C ⇔ M: accept [in ID, in item, in price, out confirmation]
  C ⇔ M: accept [out ID, out item, out price, out confirmation]

  M ⇔ C: goods [in ID, in item, in confirmation, out document]
  M ⇔ C: goods [in ID, in item, nil confirmation, out document]
  C ⇔ M: pay [in ID, in price, in document, out payment]
  C ⇔ M: pay [in ID, in price, nil document, out payment]
  M ⇔ C: receipt [in ID, in item, in payment, out done]
}
Schema for Cyberinfrastructure Resource Sharing
Maps to four protocols, naturally composed
Service Request Protocol
(Erroneous: Unsafe)
BSPL Reconstruction of Unsafe Service Request
Combining some parameters to reduce clutter

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
- cID
- pID
- mID

Provider
- pID
- cID
- mID

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

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

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

Specification Approaches

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 \( \text{goods} \land \ C(\text{Buyer, Seller, goods, pay}) \) Then
  - Active and detached (or unconditional or base)
  - \( C(\text{Buyer, Seller, T, pay}) \)
- If \( C(\text{Buyer, Seller, T, pay}) \) Then
  - If \( \text{pay} \) Then Satisfied
  - If never \( \text{pay} \) Then Violated
- If \( C(\text{Buyer, Seller, goods, pay}) \) Then
  - If \( \text{pay} \) Then Satisfied
  - If never \( \text{pay} \) and never \( \text{goods} \) Then Expired

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

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

How about this?

\[
\text{create}(d, c, p, q)
\]

\[q\]

\[\text{opt} \quad \text{true}\]

\[\text{d:Debtor} \quad \text{c:Creditor}\]

\[p\]
Operationalizing Commitments: Detach First; Optional Discharge

How about this?
Operationalizing Commitments: Creation by Creditor

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

\[ C(\text{debtor}, \text{creditor}, \text{antecedent}, \text{consequent}) \]
Commitment Life Cycle (and Patterns)
\[ C(\text{debtor, creditor, antecedent, consequent}) \]

(a) Commit

(b) Relieve
Commitment Operations

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

Specification Approaches

BSPL, the Blindingly Simple Protocol Language

Commitments
  Cupid: Information-Based Commitments
  Calypso: Commitment Alignment

Summary and Directions
Motivation
Commitment schemas (specifications) and instances?

- Relational schema and its instances
- Message schema and its instances
- Protocol specification and its instances (enactments)
- Commitment specifications and instances?
  - Technically, from propositional to first-order settings
Commitment Identifiers
Adequate treatment crucial

- **Existing approaches**: From $C(x, y, r, u)$ to $C(i, x, y, r, u)$
  - $C(0, \text{placeOrder}, \text{deliver})$, $C(1, \text{placeOrder}, \text{deliver})$, $C(2, \text{placeOrder}, \text{receipt})$
  - **Shortcoming**: syntactic, arbitrary, not amenable to reasoning
    - $C(\text{placeOrder}, \text{deliver}) \land C(\text{placeOrder}, \text{receipt}) \vdash C(\text{placeOrder}, \text{deliver} \land \text{receipt})$
    - However, $C(0, \text{placeOrder}, \text{deliver}) \land C(2, \text{placeOrder}, \text{receipt}) \vdash ?$
Commitment Identifiers: Initial Intuitions

- Commitments should be identified based upon their content
  - \( C(\text{placeOrder}(0), \text{deliver}(0)) \land C(\text{placeOrder}(0), \text{receipt}(0)) \vdash C(\text{placeOrder}(0), \text{deliver}(0) \land \text{receipt}(0)) \)
  - \( C(\text{placeOrder}(1), \text{deliver}(0)) \land C(\text{placeOrder}(0), \text{receipt}(0)) \nvdash C(\text{placeOrder}(0), \text{deliver}(0) \land \text{receipt}(0)) \)
- Cupid contribution: Formalizing identifier-related intuitions
Motivation (cont.)

Tracking commitments in databases

- Agents want to track the commitments they are involved in.
  - Many Key Performance Indicators (KPIs) based on commitments.
- Currently, each agent does so in an ad hoc manner.
Cupid: Canonical Commitment Queries

- An agent executes canonical commitment queries that map to its information store.
Canonical Queries

Commitment Instance Lifecycle

- Null (N)
  - create

- Active
  - Conditional
    - antecedent
      - antecedent_fail
        - Expired
  - consequent
    - consequent_fail
      - Violated

- Detached
  - Satisfied
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 ten 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)
Canonical Queries for DiscountQuote

- `query-name(create-clause, detach-clause, discharge-clause)`
- `created(Quote, Order and Payment[, Quote + 10] where pPrice ≥ 0.9 * uPrice * qty, Shipment[, Payment + 5])`
- Queries for detached, expired, discharged, violated are analogous
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>

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>

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></td>
<td>Feb 9</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>
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.
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 \(\sqcap\) Expr | Expr \(\sqcup\) Expr
                | Expr \(\oplus\) Expr | Expr where \(\varphi\)
T → Event + \(T\) | \(T\)
Semantics in Relational Algebra

With an eye on practical implementations

- Defines each event expression $X$’s extension $\llbracket X \rrbracket$ as a set of its instances.
- For a base event $E$, $\llbracket E \rrbracket$ equals its materialized relation.
- Semantics lifts $\llbracket \cdot \rrbracket$ to all expressions.

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

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

$D_3$. $\llbracket created(c, r, u) \rrbracket = \llbracket c \rrbracket$. A commitment is created when its create event occurs.

$D_4$. $\llbracket violated(c, r, u) \rrbracket = \llbracket (c \sqcap r) \ominus u \rrbracket$. 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

- *Well-identified* specifications capture a notion of adequate correlation among the events that in the specification.

- Instances of a *finitely expirable* specification are guaranteed to expire if not detached within a finite amount of time.

- Instances of a *finitely violable* specification are guaranteed to expire if not discharged within a finite amount of time.
Ongoing Work and Future Directions

- Implementation, optimization, performance evaluation
- Expressiveness (e.g., aggregation)
- Case studies
- Explore alternative formalizations (e.g. 4QL, and others closer to temporal logic)
Outline

Specification Approaches

BSPL, the Blindingly Simple Protocol Language

Commitments
  Cupid: Information-Based Commitments
  Calypso: Commitment Alignment

Summary and Directions
Commitment Progression via Messaging

Two views of the same enactment

- $C$ is $C$(Alice, Bob, paid($12), del(bnw))
- $C_U$ is $C$(Alice, Bob, $\top$, del(bnw))
Protocols Promote Interoperability

Asynchrony makes interoperability harder
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),** and **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
Commitment-Level Interoperability

What is an appropriate such notion?

- Interoperability for low-level protocols
  - Characterized, e.g., by absence of deadlocks
    - Deadlocked: Alice waits for payment before delivery and Bob for delivery before payment
  - Even if noninteroperable at low-level, potentially interoperable at commitment-level
Commitment Alignment
Fundamental notion of interoperability

- Each agent observes messages sequentially
  - Agent’s observation sequence determines agent’s state
- System state at time $t$: Vector of observation sequences (one for each agent) at time $t$
- System state $O$ is aligned w.r.t $C(x, y, r, u)$ iff

$$O_y \vdash C(x, y, r, u) \Rightarrow O_x \vdash C(x, y, r, u)$$
Commitments as Expectations

Asymmetric

- Not necessary

\[ O_x \vdash C(x, y, r, u) \Rightarrow O_y \vdash C(x, y, r, u) \]
Complete State

Definitionally, disregard incomplete states: In (A) and (B), misaligned at dotted line, but not in (C)

▶ Prove: Complete vectors are aligned
Challenge: Guarantee Alignment

By appropriately constraining an agent’s local computations

▶ Formalize updates from observing messages pertaining to the creation, cancelation, release, detach, and discharge of a commitment
▶ Formalize information propagation constraints
  ▶ Notify creditors of discharges
  ▶ Notify debtors of detaches
▶ Taking into account commitment reasoning (from Singh 2008)

\[
\text{Detach. } C(r \land s, u) \land r \rightarrow C(s, u)
\]

\[
\text{L-Disjoin. } C(r, u) \land C(s, u) \rightarrow C(r \lor s, u)
\]

\[
\text{R-Conjoin. } C(r, u) \land C(r, v) \rightarrow C(r, u \land v)
\]

▶ Prove: Computations guarantee complete \( O \) for any incomplete \( O \)
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
Cupid and Calypso
Independent but related

- 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

BSPL, the Blindingly Simple Protocol Language

Commitments

Summary and Directions
Layered Architecture for MAS
Low-level and high-level protocols
Summary and Directions

Exercise: Collective concept map
Directions

- More expressive protocol languages
  - More norm types: Dialectical commitments, authorizations, prohibitions, power, and so on
- Tools for specification, verification, and enactment
- Performance studies and improvement
- Application and case studies with domain partners
- Accountability
Thanks!

- National Science Foundation
- Consortium for Ocean Leadership

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