Outline

Preamble

Computing with Services

Challenges of Electronic Business

Specification Approaches

Commitments

BSPL, the Blindingly Simple Protocol Language

LoST: Local State Transfer

Advanced Topics in BSPL
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
BSPL, the Blindingly Simple Protocol Language

Parameter Adornments in BSPL

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

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

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

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

\begin{align*}
\text{Initiate} - \text{Order} \{ \\
\text{role } B, S \\
\text{parameter out ID key, out item, out price, out rID} \\
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 rID]} \\
B \mapsto S: \text{reject [in ID, in item, in price, out rID]} \\
\}
\end{align*}

- 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

```
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)
Outline

Preamble

Computing with Services

Challenges of Electronic Business

Specification Approaches

Commitments

BSPL, the Blindingly Simple Protocol Language

LoST: Local State Transfer

Advanced Topics in BSPL
Possible Enactment as a Vector of Local Histories

**Buyer**
- \( \text{rfq} \)
  - ID, item
- \( \text{quote} \)
  - ID, price
- \( \text{accept} \)
  - ID, address
- \( \text{deliver} \)
  - ID, item, address, outcome

**Seller**
- \( \text{rfq} \)
  - ID, item
- \( \text{quote} \)
  - ID, price
- \( \text{accept} \)
  - ID, address
  - ID, item, address

**Shipper**
- \( \text{ship} \)
  - ID, item, address
- \( \text{deliver} \)
  - ID, item, address, 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 \texttt{out} parameter \textbf{creates} and transmits knowledge
- An \texttt{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>Send nil first</td>
</tr>
<tr>
<td>Sends nil</td>
<td>Receive at least one instance before send</td>
<td>Receive may occur after send</td>
<td>Unconstrained</td>
</tr>
<tr>
<td>Receives in</td>
<td></td>
<td>Impossible</td>
<td>Send before receive</td>
</tr>
<tr>
<td>Receives out</td>
<td></td>
<td>Unconstrained</td>
<td>Unconstrained</td>
</tr>
<tr>
<td>Receives nil</td>
<td></td>
<td>Unconstrained</td>
<td>Unconstrained</td>
</tr>
</tbody>
</table>
Summarizing Approaches for Interaction

<table>
<thead>
<tr>
<th>Declarative (Explicit)</th>
<th>Procedural (Implicit)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Meaning</strong></td>
<td><strong>Operation</strong></td>
</tr>
<tr>
<td>Commitments and other norms</td>
<td>Temporal logic BSPL</td>
</tr>
<tr>
<td>Hard coded within internal reasoning heuristics</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 \[\text{in}\] 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.
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
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

Preamble

Computing with Services

Challenges of Electronic Business

Specification Approaches

Commitments

BSPL, the Blindingly Simple Protocol Language

LoST: Local State Transfer

Advanced Topics in BSPL
Safety: *Purchase Unsafe*

Remove conflict between *accept* and *reject*

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

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

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

- $B$ 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
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 \textit{out}
- \textit{Conflict}. At least two competing messages occur
- \textit{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 \textit{in} or \textit{out}

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

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

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

\end{verbatim}

\begin{itemize}
  \item The price can be adorned \textit{in} or \textit{out} in a reference to this protocol
\end{itemize}
The *Bilateral Price Discovery* protocol

\[
\text{BPD \{ role Taker, Maker parameter out reqID key, out query, out result}
\]

\[
\text{Taker } \xrightarrow{} \text{ Maker: priceRequest[out reqID, out query]}
\]

\[
\text{Maker } \xrightarrow{} \text{ Taker: priceResponse[in reqID, in query, out result]}
\]
The Generalized Bilateral Price Discovery protocol

GBPD {  
  role T, M  
  parameter reqID, key, query, res  

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

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

\[
\text{MPD} \begin{cases}
\text{role Taker, Exchange, Maker} \\
\text{parameter out reqID key, out query, out res}
\end{cases}
\]

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

Insurance—Claims {
role Vendor, Subscriber
parameter out policyNO key, out reqForClaim key, out claimResponse

Vendor $\mapsto$ Subscriber: createPolicy [out policyNO]
Subscriber $\mapsto$ Vendor: serviceReq [in policyNO, out reqForClaim]
Vendor $\mapsto$ 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

\[
\text{Buyer-or-Seller-Offer} \quad \{ \\
\text{role Buyer, Seller} \\
\text{parameter in ID key, out item, out price, out confirmed} \\
\}
\]

\[
\text{Buyer } \leftrightarrow \text{ Seller: rfq} \{ \text{in ID, out item, nil price} \} \\
\text{Buyer } \leftrightarrow \text{ Seller: rfq} \{ \text{in ID, out item, out price} \}
\]

\[
\text{Seller } \leftrightarrow \text{ Buyer: quote} \{ \text{in ID, in item, out price, out confirmed} \} \\
\text{Seller } \leftrightarrow \text{ Buyer: quote} \{ \text{in ID, in item, in price, out confirmed} \}
\]

- The \textbf{BUYER} or the \textbf{SELLER} may determine the binding
- The \textbf{BUYER} has first dibs in this example
Shopping Cart

Shopping Cart 

\begin{verbatim}
role B, S

parameter out ID key, out lineID key, out item, out qty, out price, out finalize

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

S \mapsto B: total[in ID, out sum]
B \mapsto S: settle[in ID, in sum, out finalize]
B \mapsto S: discard[in ID, out finalize]
\end{verbatim}
Exercise 1: *Abruptly Cancel*

Solution added

```
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, out outcome]
    S \rightarrow B: goods [in ID, in item, out outcome]
}
```

- Is this protocol safe? **No**
- Is this protocol live? **Yes**
Exercise 2: *Abruptly Cancel* Modified (with \(\text{nil}\))

Solution added

\[
\text{Abruptly Cancel} \quad \{ \\
\text{role B, S} \\
\text{parameter out ID key, out item, out outcome} \\
B \mapsto S: \text{order [out ID, out item]} \\
B \mapsto S: \text{cancel [in ID, in item, nil outcome]} \\
S \mapsto B: \text{goods [in ID, in item, out outcome]} \\
\}
\]

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

Munindar P. Singh (NCSU)  Service-Oriented Computing  Spring 2015
The Generalized Bilateral Price Discovery protocol

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

- May further remove superfluous messages, such as
  - The two messages with all "in" parameters
GBPD Restricted
To parameter adornments of \texttt{\textasciitilde out\textasciitilde}, \texttt{\textasciitilde out\textasciitilde}, and \texttt{\textasciitilde in\textasciitilde}, 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

Preamble

Computing with Services

Challenges of Electronic Business

Specification Approaches

Commitments

BSPL, the Blindingly Simple Protocol Language

LoST: Local State Transfer

Advanced Topics in BSPL
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 $\leftrightarrow$ M: rfq [out ID, out item]
M $\leftrightarrow$ C: offer [in ID, in item, out price]
C $\leftrightarrow$ M: accept [in ID, in item, in price, out confirmation]
M $\leftrightarrow$ C: goods [in ID, in item, in confirmation, out document]
C $\leftrightarrow$ M: pay [in ID, in price, in document, out payment]
M $\leftrightarrow$ 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
  - ID, item, price, done
- NetBill
  - cID
  - mID
  - ID
- Merchant
  - mID
- 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 \rightarrow M: \text{rfq} [\text{out ID, out item}]

M \rightarrow C: \text{offer} [\text{in ID, in item, out price}]
M \rightarrow C: \text{offer} [\text{out ID, out item, out price}]

C \rightarrow M: \text{accept} [\text{in ID, in item, in price, out confirmation}]
C \rightarrow M: \text{accept} [\text{out ID, out item, out price, out confirmation}]

M \rightarrow C: \text{goods} [\text{in ID, in item, in confirmation, out document}]
M \rightarrow C: \text{goods} [\text{in ID, in item, nil confirmation, out document}]
C \rightarrow M: \text{pay} [\text{in ID, in price, in document, out payment}]
C \rightarrow M: \text{pay} [\text{in ID, in price, nil document, out payment}]
M \rightarrow C: \text{receipt} [\text{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

Service Request
- mID
- ID

Provider
- pID

Actions:
- request
- accept
- reject
- forgetIt
- answer
- fail
- released
The \textit{Service Request} Protocol Via Bliss, Now Corrected

\begin{verbatim}
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]
}
\end{verbatim}