BSPL, the Blindingly Simple Protocol Language

Munindar P. Singh

North Carolina State University

April 2011

Interactions and Protocols

All actions are interactions

Goal: Specify distributed systems of autonomous, heterogeneous agents

- Focus on roles that agents play
- Identify rules of encounter
- Maintain independence from internal reasoning (policies)

Approach: Specify protocols as abstractions over interactions

Traditional Approaches

	Declarative	*Procedural*
Meaning	Commitments and other norms	Hard coded within internal reasoning heuristics
Operation	Temporal logic	State machines; Petri nets; process algebras

- 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

Remark on Control versus Data 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
- Data flow
  - Natural across computational threads
  - Explicitly tied to causality

Properties of Participants

- Autonomy
- Myopia
  - All choices must be local
- Heterogeneity: local $\neq$ internal
  - Local state
    - 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

Interaction Orientation

Interactions as first-class constructs

- Protocol
  - Abstract class (or interface) of interactions
  - Based on roles and parameters
- Roles
  - Local but not internal views of each agent
- Parameters
  - Distinguish different instances of the same protocol
- Enact protocols via LoST: Local State Transfer

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

Motivation and Benefits

- Technical
  - Statelessness
  - Consistency
  - Atomicity
  - Natural composition
- Conceptual
  - Make protocol designer responsible for specifying causality
  - Avoid contortions of traditional approaches when causality is unclear

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 existing 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

Parameter Adornments in BSPL

Capture the essential causal structure of a protocol

- ⌜<u>*in*</u>⌝: 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
- ⌜<u>*out*</u>⌝: Information that is generated by the protocol instances
  - Bindings can be fed into other protocols through their ⌜<u>*in*</u>⌝ parameters, thereby accomplishing composition
  - A standalone protocol must adorn all its public parameters ⌜<u>*out*</u>⌝
- ⌜<u>*nil*</u>⌝: Information that is absent from the protocol instance
  - Bindings must not exist

Ignoring data types of parameters for simplicity: assume strings everywhere

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

The *Hello* Protocol

```
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; could be made implicit

The *Pay* Protocol

```
Pay {
 role Payer, Payee
 parameter in ID key, in amount

 Payer ↦ 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

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 uniquifies 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 *Initiate 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

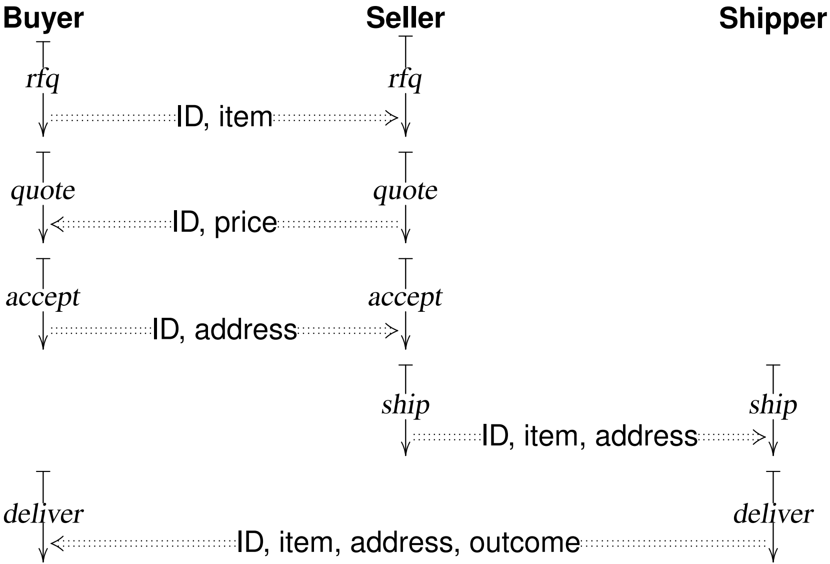
```
Purchase {
 role B, S, Shipper
 parameter out ID key, out item, out price, out outcome

 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]
}
```

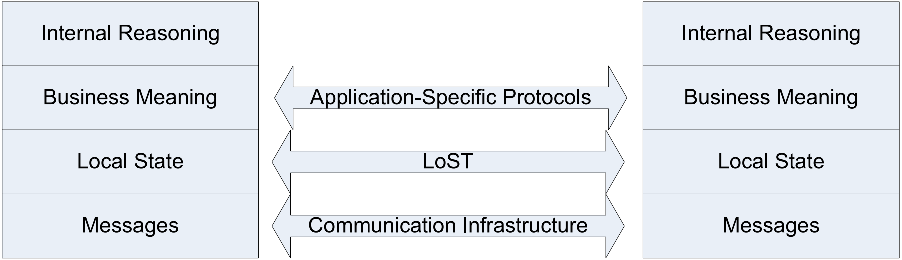
- At most one item, price, and outcome binding per ID
- Enactable standalone
- *reject* conflicts with *accept* on response (a *private* parameter)
- *reject* or *deliver* must occur for completion (to bind outcome)

Possible Enactment as a History Vector



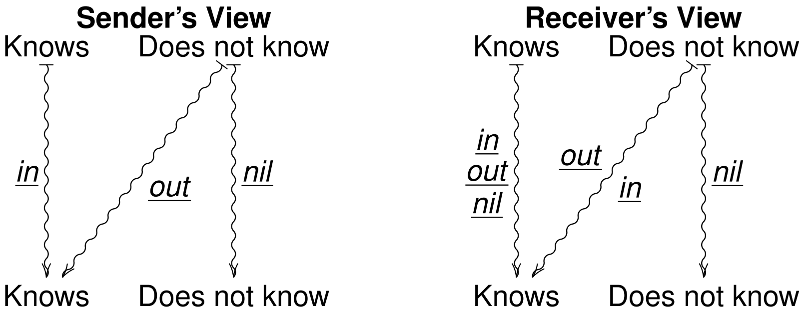
LoST Schematically

Local State Transfer



Knowledge and Viability

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



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

Send-Receive and Send-Send Constraints on a Role

	Sends in	*Sends out*	*Sends nil*
Sends in	Unconstrained	Send *out* first	Send *nil* first
Sends out		Send at most one	Send *nil* first
Sends nil			Unconstrained
Receives in	Receive first copy before send	Receive may occur after send	Send before receive
Receives out	Receive first copy before send	Impossible	Send before receive
Receives nil	Unconstrained	Unconstrained	Unconstrained

Comparing LoST and ReST

	ReST	*LoST*
Modality	Two-party; client-server; synchronous	Multiparty interactions; peer-to-peer; asynchronous
Computation	Server computes definitive resource state	Each party computes its definitive local state and the parties collaboratively and (potentially implicitly) compute the definitive interaction state
State	Server maintains no client state	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

Comparing LoST and ReST

	ReST	*LoST*
Transfer	State of a resource, suitably represented	Local state of an interaction via parameter bindings, suitably represented
Idempotent	For some verbs, especially GET	Always; repetitions are guaranteed harmless
Caching	Programmer can specify if cacheable	Always cacheable
Uniform interface	GET, POST, . . .	⌜*in*⌝, ⌜*out*⌝, ⌜*nil*⌝
Naming	Of resources via URIs	Of interactions via (composite) keys, whose bindings could be URIs

Comparing BSPL and WS-CDL

- Similarity: both emphasize interaction
- Differences: WS-CDL is
  - Operational
    - 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

Well-Formedness Conditions

- A parameter that is adorned ⌜*in*⌝ in a declaration must be ⌜*in*⌝ throughout its body
- A parameter that is adorned ⌜*out*⌝ in a declaration must be ⌜*out*⌝ in at least one reference
  - When adorned ⌜*out*⌝ in zero references, not enactable
  - When adorned ⌜*out*⌝ in exactly one reference, consistency is guaranteed
  - When adorned ⌜*out*⌝ in two or more references, no more than one can execute
- A private parameter must be ⌜*out*⌝ in at least one reference

ACID Properties

With inspiration from database transactions though with modifications

- Atomicity: if a protocol completes, each reference within it that is initiated also completes
  - Ensured by placing one agent in charge of each conflict
- Consistency: at most one of a set of conflicting references occurs
  - Ensured by placing one agent in charge of each conflict
- Isolation: separate enactments do not interfere
  - Ensured by keys
- Durability: any enactment is permanent
  - Ensured by the immutability of bindings

References: Analogous to Macros or Procedures?

- Macro
  - Expanded into the body of a composite protocol: partially enactable
  - Maximize concurrency
- Procedure
  - All or none
  - Enable compositionality
- BSPL treats references as both
  - Enactment is maximally concurrent, at the level of individual messages
  - Atomicity avoids undesirable outcomes

Standing Order

As in insurance claims processing

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

 Vendor ↦ Subscriber: createPolicy[out policyNO]
 Subscriber ↦ Vendor: serviceReq[in policyNO, out reqForClaim]
 Vendor ↦ 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

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

in-out Polymorphism

Flexible Offer

```
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 ⌜*in*⌝ or ⌜*out*⌝ in a reference to this protocol

The *Bilateral Price Discovery* protocol

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

 Taker ↦ Maker: priceRequest[out reqID, out query]
 Maker ↦ Taker: priceResponse[in reqID, in query, out result]
}
```

The *Generalized Bilateral Price Discovery* protocol

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

 T ↦ M: priceRequest[out reqID, out query]
 T ↦ M: priceRequest[in reqID, in query]

 M ↦ T: priceResponse[in reqID, in query, out res]
 M ↦ T: priceResponse[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)
}
```

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]
}
```

Directions

- Implementation of LoST
- Methodology for specifying practical protocols
- Expansions of the language to handle role hierarchies
- Theoretical results
  - Decision procedures for judging *consistent* enactability
  - Treatment of recursive protocols