Programming Multiagent Systems via Information Protocols: the case of Jason+BSPL

APM Workshop 2024

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

October 3, 2024 - The 6th International Asynchronous Programming Models Workshop 02-04.10.2024, in Turin, Italy

[1]Università di Torino, Torino, Italy
[2]North Carolina State University, Raleigh, NC, USA
[3]Lancaster University, Lancaster, United Kingdom

Motivation

- An *interaction protocol* models the communication constraints between agents in a multiagent system
- Engineering multiagent system based on protocols offers key benefits:
  - *Decentralized MAS*; without relying on a distinguished locus of state or control
  - *Clear implementation*, separation between the coordination aspects and business logic of an agent
  - *Loose coupling*, changes in one agent's implementation do not affect the implementation of others
  - *Reducing agent complexity*, avoiding programming errors

Motivation

- Unfortunately, leading (cognitive) programming models for MAS *have a limited* or *do not support* protocols
  - Jason [Vieira et al., 2007]
  - JADEL [Bergenti et al., 2017]
  - JaCaMo [Boissier et al., 2013]
  - JADE [Bellifemine et al., 2007]
  - SARL [Rodriguez et al., 2014]

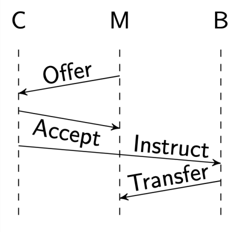
Motivation

- Prevalent programming models are lacking, the common approach for dealing with messages is reactive via a *message handler* for each incoming message, as in message-oriented middleware [Hohpe and Woolf, 2004], they considers each message *independently*
- Messages are generally related to each other and an agent usually needs to act based on its state, which depends on messages received or sent, the agent code reconstructs the necessary state of computation:
  - tied up with the requisite business reasoning, and
  - in more than one place, based on what message emissions and receptions can lead to that state

Motivation

Synchronous

A common approach of programming languages paradigm: sends and receives must interlock as in a zip (Hoare), assume FIFO messaging and agents selectively receive messages



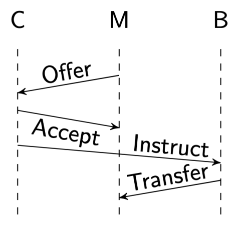
Motivation

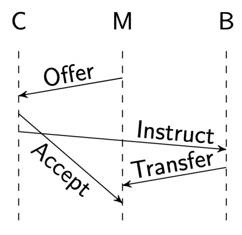
Synchronous

A common approach of programming languages paradigm: sends and receives must interlock as in a zip (Hoare), assume FIFO messaging and agents selectively receive messages

Violated despite FIFO messaging.

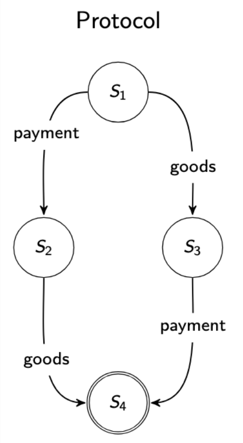
Choose to receive Accept before Transfer?





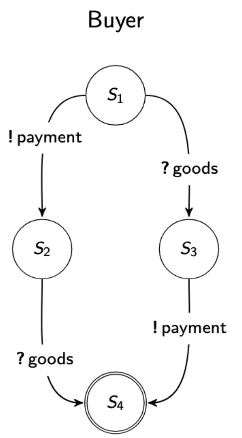
Motivation

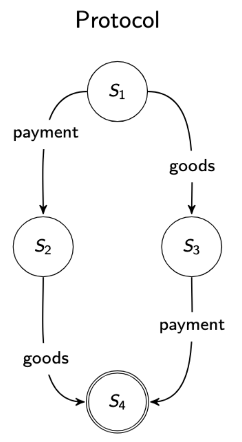
Is receiving a message an agent decision?

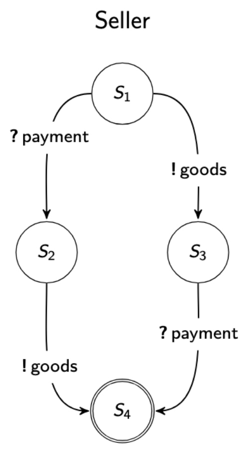


Motivation

Is receiving a message an agent decision?



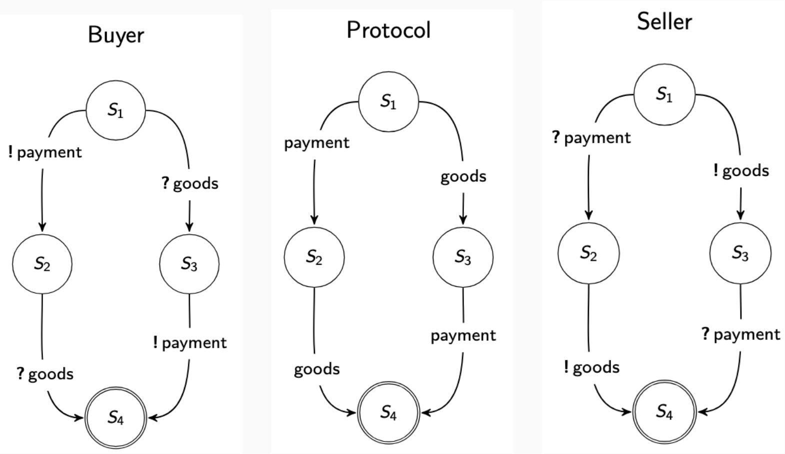




Motivation

Is receiving a message an agent decision?

Results in deadlock!



Motivation

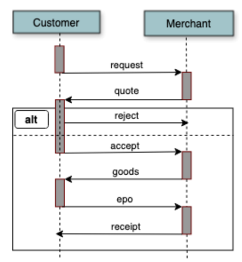
Common approach

Causality via control flow (state machine, AUML, process algebra, session types).

Jason+BSPL: Our goal

Overcome shortcomings of traditional approaches, such as:

- **Incompatibilities** between agents due to the message schemas being blended into business logic
- **Semantic errors** due to a lack of a formal model
- **Inflexibility** due to the programmer having to maintain the protocol state via a state machine




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

Jason+BSPL: Our proposal

- We adopt information protocols, in particular, *Blindingly Simple Protocol Language* (BSPL) [Singh, 2011], a fully **declarative** and fully **asynchronous** model for communication
- **Jason+BSPL** unites two aspects of autonomy:
  - **Cognitive autonomy**, via Jason
  - **Social autonomy**, via information protocols

Information protocols

Causality via information dependencies

Information protocols, BSPL [Singh, 2011]

- Roles
- Message schemas
  - A name
  - A sender role
  - A receiver role
  - One or more parameters
- A message may be received at *any time* in any relative order with respect to other messages
- The emission of a message *depends upon* what information the agent has

```
NetBill {
 roles (M)erchant, (C)ustomer
 parameters out ID key, out item, out done

 C -> M: request[out ID key, out item]
 M -> C: quote[in ID key, in item, out price]
 C -> M: accept[in ID key, in item, in price, out decision, out outcome]
 C -> M: reject[in ID key, in item, in price, out decision, out done]
 M -> C: goods[in ID key, in item, in outcome, out shipped]
 C -> M: epo[in ID key, in item, in price, in shipped, out cc]
 M -> C: receipt[in ID key, in price, in cc, out chit, out done] }
```

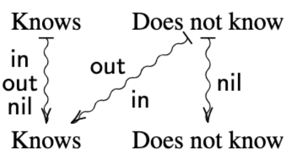
Information protocols, BSPL [Singh, 2011]

- A *message instance* is a tuple of bindings for the parameters of that schema that are adorned either ⌜in⌝ or ⌜out⌝
- The ⌜key⌝ parameters of a schema form a composite key and uniquify its instances

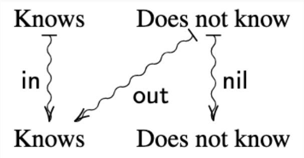
```
NetBill {
 roles (M)erchant, (C)ustomer
 parameters out ID key, out item, out
     done

 C -> M: request[out ID key, out item]
 M -> C: quote[in ID key, in item, out
     price]
 C -> M: accept[in ID key, in item, in
     price, out decision, out outcome]
 C -> M: reject[in ID key, in item, in
     price, out decision, out done]
 M -> C: goods[in ID key, in item, in
     outcome, out shipped]
 C -> M: epo[in ID key, in item, in price
     , in shipped, out cc]
 M -> C: receipt[in ID key, in price, in
     cc, out chit, out done] }
```

Receiver:

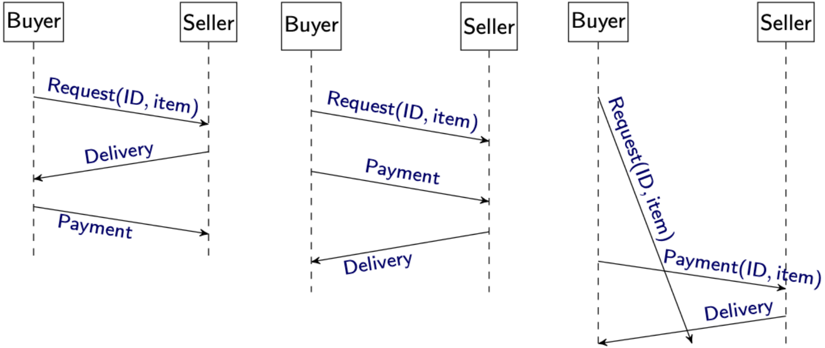


Sender:



Information protocols, BSPL [Singh, 2011]

```
B ↦ S: Request[out ID key, out item]
S ↦ B: Delivery[in ID key, in item, out status]
B ↦ S: Payment[in ID key, in item, out token]
```



Information protocols, BSPL [Singh, 2011]

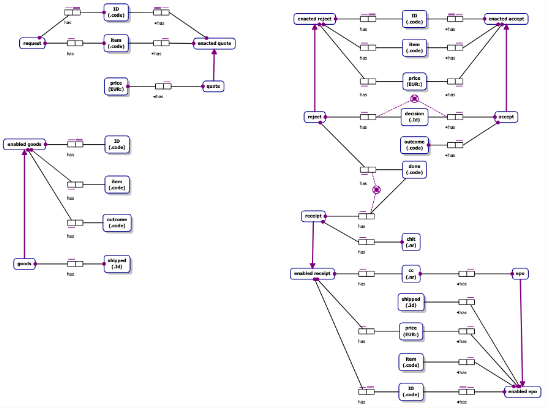
- No two message instances with the same bindings for overlapping ⌜key⌝ parameters may have distinct bindings for common non-key parameters
- No two message instances may have overlapping key parameter bindings as well as a binding of the same ⌜out⌝ parameter
- The key parameters of a protocol provides a basis for the uniqueness of its enactments

```
NetBill {
 roles (M)erchant , (C)ustomer
 parameters out ID key , out item , out done

 C -> M: request[out ID key , out item]
 M -> C: quote[in ID key , in item , out price]
 C -> M: accept[in ID key , in item , in price , out decision , out outcome]
 C -> M: reject[in ID key , in item , in price , out decision , out done]
 M -> C: goods[in ID key , in item , in outcome , out shipped]
 C -> M: epo[in ID key , in item , in price , in shipped , out cc]
 M -> C: receipt[in ID key , in price , in cc , out chit , out done] }
```

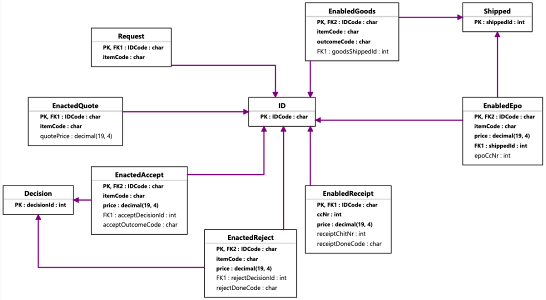
Information protocols, BSPL [Singh, 2011]

- No two message instances with the same bindings for overlapping ⌜key⌝ parameters may have distinct bindings for common non-key parameters
- No two message instances may have overlapping key parameter bindings as well as a binding of the same ⌜out⌝ parameter
- The key parameters of a protocol provides a basis for the uniqueness of its enactments



Information protocols, BSPL [Singh, 2011]

- No two message instances with the same bindings for overlapping ⌜key⌝ parameters may have distinct bindings for common non-key parameters
- No two message instances may have overlapping key parameter bindings as well as a binding of the same ⌜out⌝ parameter
- The key parameters of a protocol provides a basis for the uniqueness of its enactments



An example

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

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

An example

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

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

Initiate (virtual)

ID	item

B:rfq

ID	item

S:rfq

ID	item

An example

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

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

Initiate (virtual)	
ID	item
1	fig

B:rfq	
ID	item
1	fig

S:rfq	
ID	item

An example

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

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

Initiate (virtual)

ID	item
1	fig
5	jam

B:rfq

ID	item
1	fig
5	jam

S:rfq

ID	item

An example

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

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

Initiate (virtual)

ID	item
1	fig
5	jam

B:rfq

ID	item
1	fig
5	jam

S:rfq

ID	item
5	jam

An example

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

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

Initiate (virtual)

ID	item
1	fig
5	jam

B:rfq

ID	item
1	fig
5	jam
×1	apple

S:rfq

ID	item
5	jam

An example

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

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

Initiate (virtual)

ID	item
1	fig
5	jam
8	fig

B:rfq

ID	item
1	fig
5	jam
8	fig

S:rfq

ID	item
5	jam

An example

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

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

An example

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

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

Offer (virtual)

ID	item	price
1	fig	

B:rfq

ID	item
1	fig

B:quote

ID	item	price

S:rfq

ID	item
1	fig

S:quote

ID	item	price

An example

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

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

Offer (virtual)

ID	item	price
1	fig	10

B:rfq

ID	item
1	fig

B:quote

ID	item	price

S:rfq

ID	item
1	fig

S:quote

ID	item	price
1	fig	10

An example

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

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

Offer (virtual)

ID	item	price
1	fig	10

B:rfq

ID	item
1	fig

B:quote

ID	item	price
1	fig	10

S:rfq

ID	item
1	fig

S:quote

ID	item	price
1	fig	10

An example

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

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

Offer (virtual)

ID	item	price
1	fig	10

B:rfq

ID	item
1	fig

B:quote

ID	item	price
1	fig	10

S:rfq

ID	item
1	fig

S:quote

ID	item	price
1	fig	10
×4	fig	10

An example

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

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

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

An example

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

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

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

Decide Offer (virtual)

ID	item	price	decision
1	fig	10	

B:rfq

ID	item
1	fig

B:quote

ID	item	price
1	fig	10

B:accept

ID	item	price	decision

B:reject

ID	item	price	decision

An example

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

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

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

Decide Offer (virtual)

ID	item	price	decision
1	fig	10	nice

B:rfq

ID	item
1	fig

B:quote

ID	item	price
1	fig	10

B:accept

ID	item	price	decision
1	fig	10	nice

B:reject

ID	item	price	decision

An example

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

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

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

Decide Offer (virtual)

ID	item	price	decision
1	fig	10	nice

B:rfq

ID	item
1	fig

B:quote

ID	item	price
1	fig	10

B:accept

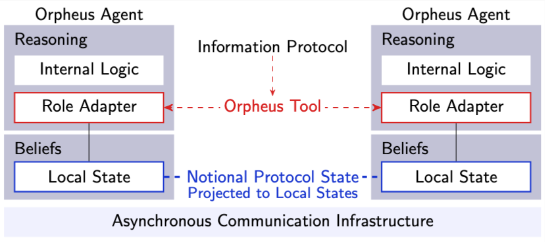
ID	item	price	decision
1	fig	10	nice

B:reject

ID	item	price	decision
×1	fig	10	nice

Jason+BSPL Programming Model

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



Jason+BSPL Programming Model

- **An incoming message is added to the local state if it is consistent with the local state**, i.e., if **no other binding** is already known for any its parameters (relative to the key)
- For **outgoing messages**:
  - An **enabled instance** is a partial instance in that:
    1. its ⌜in⌝ **parameters** are bound because their bindings are **known**, and
    2. its ⌜out⌝ **parameters** are not bounded because they are **not known**
  - An **attempt** is successful if the completed messages are mutually **consistent** in their bindings; **the sent messages are added to the local state**

To achieve some goal, the agent (1) **queries** *if there are enabled instances corresponding to the message it wants to send, (2)* **completes** *them by producing bindings for their ⌜out⌝ parameters, and (3)* **attempts** *to send them in one shot*

Enabled-Based Programming Model

- Jason-BSPL supports a novel programming model based on message enablement, in which the developer specifies plans for emitting enabled messages
- To achieve some agent-specific goal g, the agent queries if there are enabled instances corresponding to the messages it wants to send,
- *Completes* them by producing bindings for their ⌜out⌝ parameters, and
- *Attempts* to send them all in one shot

```
+!g
 : enabled(m_1) &...& enabled(m_q)
 <- !complete(m_1,...,m_q);
      !attempt(m_1,...,m_q).
+!attempt(m_1,...,m_q)
 : consistent(m_1,...,m_q)
 <- for (.member(m[receiver(R)], [m_1,...,m_q]))
        {.send(R, tell , m);
         +sent(m)}.
enabled(m(...)) :- ... //BSPL semantics
consistent(m_1...m_q) :-...//BSPL semantics
+sent(m) <- ... // BSPL semantics
+m : consistent(m, local) <- ... // BSPL semantics
```

Jason+BSPL Programming Model: Advantages

- Changes to protocol
- Changes to Agent Decision Making
- Changes to Communication Infrastructure
- Correlating Information

Jason+BSPL: The Orpheus Tool

- It is available here: `https://di.unito.it/orpheus`

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

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

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

Hohpe, G. and Woolf, B. (2004).
Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.

Signature Series. Addison-Wesley, Boston.

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

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

Vieira, R., Moreira, Á. F., Wooldridge, M. J., and Bordini, R. H. (2007).
On the formal semantics of speech-act based communication in an agent-oriented programming language.

Journal of Artificial Intelligence Research (JAIR), 29:221–267.