Multiparty Session Types Examples

Contents

1 Preamble

This note presents examples of multiparty session types taken from existing business protocols in a notation developed by Gary Brown and one of the authors of [1], for description, validation and execution of business protocols.

The correspondence between the notation in [1] and the notation here is that (1) participants and channels are given as literals, which are more readable and are declared before they are used; (2) selection and branching are comibined with value passing (as in methods and procedures); and (3) we use the familiar curly brace notation (prefixed with the keyword protocol followed by the name of the type), instead of abstract syntax. For (2) we use the notation

```
channel chSeller @ Seller;
```

[numbers=none] which means a channel `chSeller` is used for sending messages to a participant `Seller` (that is, the notation fixes the receiver of a channel).

Each section treats one example. Each example is preceded, when necessary, by a brief note; and is followed by illustration and comments.

2 Example: Buyer-Broker-Seller Protocol

```
1  import Order, Invoice, OutOfStock;
2  protocol BuyerBrokerSeller {
3    participant Buyer, Broker, Seller;
4    channel chBuyer @ Buyer, chBroker @ Broker, chSeller @
         Seller;
5
6    chBroker.Order from Buyer;
7    choice @ Broker {
8       chSeller.Order from Broker;
9       choice @ Seller {
10         chBroker.Invoice(void) from Seller;
11         chBuyer.Invoice(void) from Broker;
12      } or {
13         chBroker.OutOfStock(void) from Seller;
14         chBuyer.OutOfStock(void) from Broker;
15      }
16   } or {
17      chSeller.OutOfStock(void) from Broker;
18   }
19 }
```

Comment 2.1 (Buyer-Broker-Seller Protocol)
We illustrate the basic scenario of the protocol above.

1. `Buyer` orders goods to `Broker` (Line 6).
2. `Broker` may choose to order the same goods to `Seller` (Line 8), or `Broker` may choose to send instantly an out-of-stock message to `Buyer` (Line 17).
3. If the former is the case then there are two possibilities:
4. First, `Seller` may (for example if it has the goods) send an invoice to `Broker` (Line 10) which in turn sends an invoice to `Buyer` (Line 11).
5. Or alternatively `Seller` may send an out-of-stock message to `Broker` (Line 13) which in turn sends the same message to `Buyer` (Line 14).

Note that the distinction between operator names (e.g. `Order` and `OutOfStock`) is used for making the choice meaningful: this means these names play an essential role both for static validation and at runtime, just like method names (selectors) in objects.

3 Example: Recursion

We first inroduce a notation for recursion.

```
1  import Order, Invoice; // import data types
2  protocol BuyerSellerWithRecursion {
3    participant Buyer, Seller;
4    channel chSeller @ Seller, chBuyer @ Buyer;
5
6    recur @ Buyer Transaction: {
7      chSeller.order(Order) from Buyer;
8      choice @ Seller {
9        chBuyer.invoice(Invoice) from Seller to Buyer;
10     } or {
11       chBuyer.outOfStock(void) from Seller to Buyer;
12     };
13     choice @ Buyer {
14       chSeller.doItAgain(void) from Buyer to Seller;
15       Transaction;
16     } or {
17       chSeller.letsEndIt(void) from Buyer to Seller;
18     }
19   }
20 }
```

Comment 3.1 (Recursion)
In Line 6, the recursion (prefixed by the keyword recur) is introduced, located at `Buyer` (hence the decision to recur lies in `Buyer`) and labelling the recursion block (Lines 6-19) as `Transaction`. In Line 15, this label is used for recursion, by simply citing it. As a whole the protocol reads:

> `Buyer` sends an order to `Seller` (L7); `Seller` then sends back either an invoice (L11) or an out-of-stock message (L14). Then `Buyer` decides whether it wishes to repeat or not: if yes it sends `doItAgain` (L14) and the conversation recurs (L15 then to L6): if no it sends `letsEnd` (L17). Since no recursion is done the protocol terminates.

The difference between repeat and recur is that, in the latter, recurrence is no longer implicit: it now needs to be explicitly specified (as in Line 15) or else the block will exit the recursion block. The use of the recursion label should always be local to (inside) the labelled block.

4 Example: Buyer with Two Sellers (1)

We first treat a simple example where one buyer interacts with two sellers.

```
1  import Product, Quote, Invoice;
2  protocol BuyerAndTwoSellers {
3    participant Buyer, SellerA, SellerB;
4    channel chBuyerA@Buyer, chBuyerB@Buyer, chSellerA@SellerA,
         chSellerB@SellerB;
5
6    chSellerA.quoteRequest(Product) from Buyer to SellerA;
7    chSellerB.quoteRequest(Product) from Buyer to SellerB;
8    parallel {
9      choice @ SellerA {
10       chBuyerA.quote(Quote) from SellerA to Buyer;
11       chSellerA.order(Order) from Buyer to SellerA;
12       chBuyerA.Invoice from SellerA to Buyer;
13     } or {
14       chBuyerA.outOfStock(void) from SellerA to Buyer;
15     }
16   } and {
17     choice @ SellerB {
18       chBuyerB.quote(Quote) from SellerB to Buyer;
19       chSellerB.order(Order) from Buyer to SellerB;
20       chBuyerB.Invoice from SellerB to Buyer;
21     } or {
22       chBuyerB.outOfStock(void) from SellerB to Buyer;
23     }
24   }
25 }
```

Comment 4.1 (Parallel)
In Line 4, two channels (`chBuyerA` and `chBuyerB`) are located at `Buyer`. In Lines 6 and 7, `Buyer` sends a message to `SellerA` and another to `SellerB` sequentially; then from Line 8, there is the parallel construct which describe two parallel conversations, one between `Buyer` and `SellerA` using channels `chSellerA` and `chBuyerA` (from Line 9 to Line 14), and the other isomorphic one between `Buyer` and `SellerB` using channels `chSellerB` and `chBuyerB` (from Line 16 to Line 21). Note that two parallel conversations use two disjoint sets of channels for their conversation: in particular messages to `Buyer` are handled by two distinct channels. This disjointness prevents communicated messages from getting mixed up, which is why Buyer is using two channels.

Comment 4.2 (Parallel and Message Mix-up)
A mix-up of messages which may be asynchronously arriving at a participant, can be prevented in several ways.

(1) Distinct channels, as in Example 4 above.
(2) Distinct operator names.
(3) Distinct values.

Among the three, (3) is *not* desirable since we cannot guarantee the lack of confusion statically, i.e. at a compile time. On the other hand, the solutions (1) and (2) have the merit in that such a guarantee can be done at the signature (protocol) level. Once this is done, any program or model which conforms to the protocol is guaranteed to be confusion-free.

Between (1) and (2), (1) is robust in that the design (including its change) can be done systematically by preparing disjoint channels for conversations which may possibly run in parallel. (2) may be less robust since there may always be the chances that two parallel threads of conversations wish to use the same document types etc., though for smaller protocols this method will also work. The static checking is equally easy in both cases.

Comment 4.3 (simple multi-casting)
Lines 6 and 7 may as well be written:

```
{chSellerA,chSellerB}.quoteRequest(Product) from Buyer;
```

which has the same semantics as Lines 6 and 7 since we are assuming communication is asynchronous. A later version will discuss an example with full-fledged multiparty channel.

5 Example: Buyer with Two Sellers (2)

We consider a variant of the previous example.

```
1  import Product, Quote, Order, Invoice; // import data types
2
3  protocol BuyerAndTwoSellersAsync {
4    participant Buyer, SellerA, SellerB;
5    channel chBuyerA@Buyer, chBuyerB@Buyer, chSellerA@SellerA,
         chSellerB@SellerB;
6
7    parallel {
8      chSellerA.quoteReq(Product) from Buyer to SellerA;
9      choice @ SellerA {
10       chBuyerA.quote(Quote) from SellerA to Buyer;
11       chSellerA.order(Order) from Buyer to SellerA;
12       chBuyerA.invoice(Invoice) from SellerA to Buyer;
13     } or {
14       chBuyerA.outOfStock(void) from SellerA to Buyer;
15     }
16   } and {
17     chSellerB.quoteReq(Product) from Buyer to SellerB;
18     choice @ SellerB {
19       chBuyerB.quote(Quote) from SellerB to Buyer;
20       chSellerB.order(Order) from Buyer to SellerB;
21       chBuyerB.Invoice(Invoice) from SellerB to Buyer;
22     } or {
23       chBuyerB.outOfStock(void) from SellerB to Buyer;
24     }
25 }
```

Comment 5.1 (Parallel and Synchronisation)
In Example 4, we first have two interactions (Lines 6 and 7), followed by two parallel conversations (one from Line 9 to Line 14 and another from Line 16 to Line 21).

In Example 5, the initial two interactions are now merged into respective parallel conversations: so the original Line 6 is now found in Line 8 of Example 5, similarly Line 7 is found in Line 16 of Example 5.

In the present case, these two protocols (as far as we do not mind the order of two initial sending actions by `Buyer`) may in fact be regarded as being logically identical *as far as we regard send actions to be purely asynchronous* (i.e. they demand no synchronous ack — which we presume to be the standard assumption). If however a sending action demands a synchronous ack, Example 4 has an extra sequencing, hence two are logically distinct.

6 Example: Buyer with Two Sellers (3)

We next consider the case when we synchronise *after* the parallel construct, taking a speculative conversation by `Buyer`.

```
1  protocol BuyerAndTwoSellersSpeculative {
2    participant Buyer, SellerA, SellerB;
3    channel chBuyerA@Buyer, chBuyerB@Buyer, chSellerA@SellerA,
         chSellerB@SellerB;
4
5    parallel {
6      chSellerA.quoteReq(Product) from Buyer to SellerA;
7      chBuyerA.quote(Quote) from SellerA to Buyer;
8    } and {
9      chSellerB.quoteReq(Product) from Buyer to SellerB;
10     chBuyerB.quote(Quote) from SellerB to Buyer;
11   };
12   choice @ Buyer {
13     chSellerA.order(Order) from Buyer to SellerA;
14     chBuyerA.Invoice from SellerA to Buyer;
15   } or {
16     chSellerB.order(Order) from Buyer to SellerB;
17     chBuyerB.invoice(Invoice) from SellerA to Buyer;
18   }
19
20 }
```

Comment 6.1 (Parallel and Synchronisation, 3)
Note Lines 12-18 describe a conversation after the parallel construct. We outline the scenario of this protocol:

> `Buyer` asks `SellerA` and `SellerB` the quotes of the same product in parallel; if one gives a better quote then only with that seller `Buyer` will order the product to which the seller will send the invoice.

In this case it makes a good case that `Buyer` waits until *both* parallel conversations complete, which is precisely what the protocol above dictates.

References

1. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. To appear as a technical report, Department of Computing, Imperial College, London, 2007.