Evaluating the Benefits of Orpheus for Iterative and Incremental Development in MAS

Matteo Baldoni[1 [0000−0002−9294−0408]], Cristina Baroglio[1 [0000−0002−2070−0616]], and Roberto Micalizio[1 [0000−0001−9336−0651]]

Università di Torino, 10149 Torino, Italy
{firstname.lastname}@unito.it

Abstract. Traditional agent-oriented programming languages hinder iterative and incremental development in multiagent systems (MAS) because message-centric interaction protocols intertwine coordination logic with internal agent behaviors, causing changes in interaction requirements to cascade across system components. Building on the Orpheus programming model, which uses declarative information protocols in the Blindingly Simple Protocol Language (BSPL) to decouple coordination from agent reasoning, this work analyzes how such abstractions support iterative and incremental development. By localizing change and mitigating the propagation of interaction updates, Orpheus enables more flexible, robust, and maintainable MAS implementations. Tool support that automatically generates protocol-aware interfaces further facilitates evolving communication requirements without pervasive code revisions. The evaluation demonstrates significant advantages for development agility.

Keywords: Iterative and Incremental Development · BSPL · Jason.

1 Introduction

Interaction is a defining characteristic of multiagent systems (MAS), yet providing adequate programming abstractions for flexible and robust interaction remains a longstanding challenge. In this context, Winikoff [23] identified two fundamental shortcomings

of agent-oriented programming languages (AOPLs). First, despite the importance of explicitly modeling interaction in MAS, traditional AOPLs [22, 4, 6, 16, 5] largely provide only low-level primitives for message sending and reception. Such primitives expose communication as a control-transfer mechanism between agents, enabling unstructured interaction patterns that Winikoff critically compared to the use of *goto* statements. Second, *interaction protocols*, typically specified using notations such as AUML [14], are predominantly *message-centric* and impose overly restrictive constraints on agent interactions. While these approaches offer explicit control over message sequencing, they tightly couple communication structure with agent behavior, limiting robustness and flexibility. To

address these limitations, Winikoff advocated the adoption of higher-level communication abstractions. More than a decade later, however, mainstream AOPLs have seen little evolution with respect to these challenges.

From a software engineering perspective, the limitations of traditional AOPLs also hinder the adoption of iterative and incremental development processes [15, 21] in MAS. In message-centric AOPLs, interaction logic, application behavior, and protocol state are often interwoven within agent code. As a consequence, changes to interaction requirements (such as modifying message orderings, adding alternative interaction paths, or introducing new participants) tend to propagate across multiple agents and plans, resulting in costly and error-prone iterations. Effective iteration instead requires programming abstractions that localize change and prevent cascading effects.

At the agent level, BDI-based languages such as Jason [7] already provide a partial solution by clearly separating an agent's goals, beliefs, and plans. This separation supports incremental refinement of agent behavior, enabling new capabilities to be added without disrupting existing reasoning structures. However, these benefits are undermined at the interaction level by protocol encodings that entangle coordination concerns with application logic and require explicit management of protocol execution state. These observations motivate the use of higher-level, declarative interaction models that depart from message-centric protocol specifications. Such models *decouple* coordination from agent-internal reasoning and control flow, allowing interaction constraints to evolve independently of agent behavior. This decoupling *reduces* the cost of change by enabling modifications to interaction policies without requiring pervasive code revisions, and by supporting the composition and refinement of interactions across development increments.

Orpheus is a programming model for MAS [2], built on Jason and characterized by interaction protocols that are expressed in the Blindingly Simple Protocol Language (BSPL) [17]. Unlike traditional agent communication languages, like KQML [12] and FIPA [13], BSPL specifies interaction in terms of information flow constraints rather than message sequences, providing a declarative and asynchronous model that supports flexible coordination among agents [9]. The Orpheus tooling automatically generates plan and query libraries that assist developers by tracking protocol-relevant state and computing, at runtime, the set of *enabled* messages, i.e., messages that an agent is legally permitted to send. Together, these features support the iterative and incremental development of flexible, robust, and decentralized multiagent systems by localizing change and mitigating the impact of evolving interaction requirements.

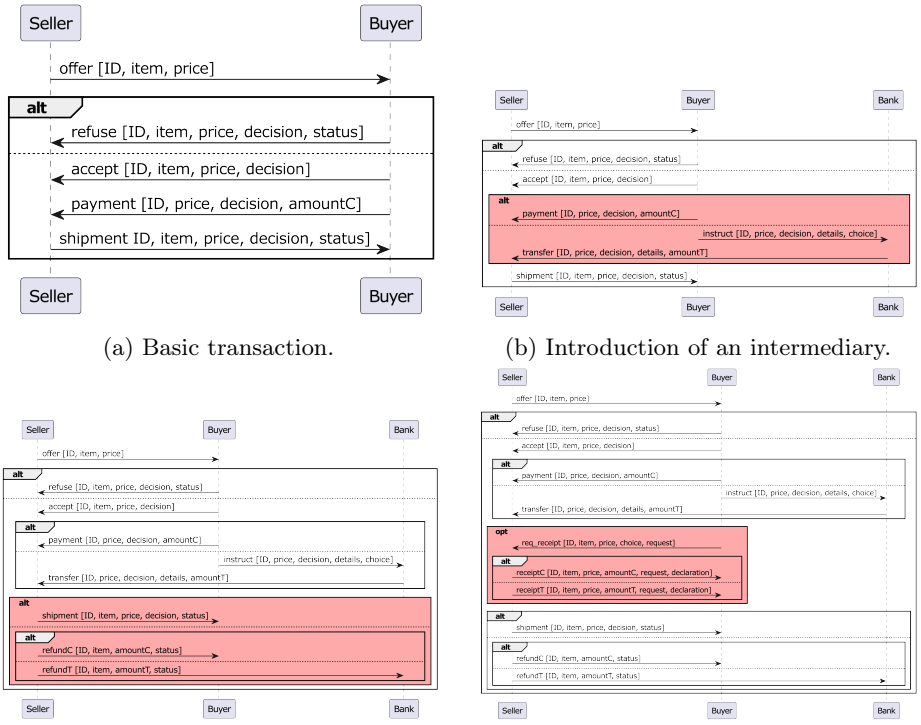
BSPL is equipped with formal semantics and verification techniques [18, 19], as well as programming models and tooling [11, 10]. By adopting BSPL, Orpheus avoids several well-known limitations of message-centric interaction protocols commonly found in traditional AOPLs. In particular, it mitigates agent incompatibilities caused by the entanglement of message schemas with application logic, reduces semantic errors arising from the absence of a formal interaction model, and improves flexibility by eliminating the need for programmers to

explicitly manage protocol execution through state machines. Interaction constraints are, instead, captured declaratively, allowing agents and protocols to be composed and refined independently across development increments.

In this work, we practically show that Orpheus allows overcoming the limitations identified by Winikoff, fostering the adoption of iterative and incremental development processes [15, 21] in MAS, localizing changes and preventing cascading effects.

2 The Benchmark

In order to show the limits underlined in the introduction, we develop in an iterative manner the EBusiness protocol [8]. Here, a Seller and a Buyer engage



(a) Basic transaction.

(b) Introduction of an intermediary.

(c) Refund mechanism.

(d) Proof of receipt.

Fig. 1: EBusiness interaction protocol developed in fourth iterations (in red the difference with the previous iteration).

in commercial transactions. The Seller may submit an offer, which the Buyer may either accept or refuse. Upon acceptance, the Buyer proceeds with payment, after

which the Seller ships the item. The Buyer may either pay the Seller directly or instruct the Bank to execute the transfer. If, for any reason, the Seller is unable to ship the item, the Seller shall issue a refund to the Buyer (or to the Bank, as applicable). After completing the payment, the Buyer may request a receipt from the Seller. We model this interaction through four successive iterations (Fig. 1):

1. *Basic transaction.* The Seller submits an offer, which the Buyer may accept or decline. If the offer is accepted, the Buyer pays the Seller directly and the Seller ships the item. See Figure 1a.
2. *Introduction of an intermediary.* A new role, the Bank, is introduced. The Buyer may choose either to pay the Seller directly or to instruct the Bank to execute the transfer on their behalf. See Figure 1b.
3. *Refund mechanism.* The possibility of a refund is introduced. If the Seller is unable to ship the item, the Seller shall issue a refund to the Buyer or, when applicable, to the Bank. See Figure 1c.
4. *Proof of receipt.* After completing the payment, the Buyer may request a declaration of receipt from the Seller. See Figure 1d.

We will compare two implementations: one in which the agents playing the roles of the EBusiness protocol are implemented in Jason and agent communication is realized by KQML [7, Chapter 6]), and one in which the agents are implemented according the Orpheus programming model and communication is realized in BSPL. Note that also Orpheus uses Jason as agent programing language, but the programming model and communication support are different. *The comparison focuses on the programmer's effort in updating each iteration onto the previous one.* For this reason, we leave out the code that is generated in an automatic way in Orpheus, which does not modify the programmers' effort because it is transparent to their eyes. For the sake of comparability, the two programs need to behave "in the same way", that is, they are developed according to a minimality principle by which a message can be sent (or handled) only when such action is correct with respect to the state of the protocol specification, at which the execution arrived. Moreover, messages must respect the same schema in both implementations, and the goals for sending or handling messages must have the same parameters.

The features that practically allow the comparison are: 1) the number of code lines that are added or modified at each iteration, 2) the number of new or modified plans, 3) the kind of applied modifications. Performance, in this context, is not an issue. For this reason we do not compare the execution times.[1]

3 EBusiness in Jason

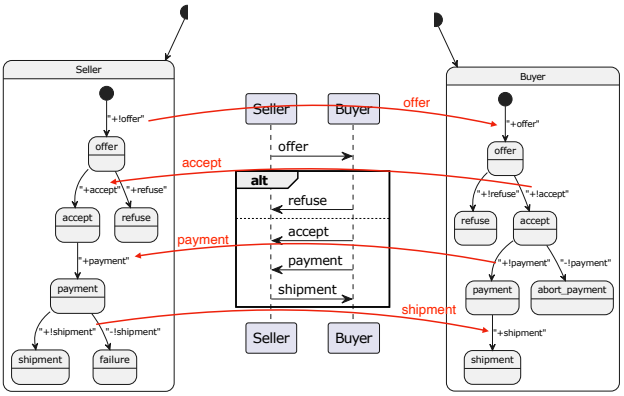
We developed the Jason implementation as it is described in [7, Sec. 6.3, page 134]. In particular, the belief *step* is added to remember the current state of

[1] All the source codes are available at https://gitlab.di.unito.it/baldoni/argonauts in the folder EBusiness Incremental Approach inside Examples.

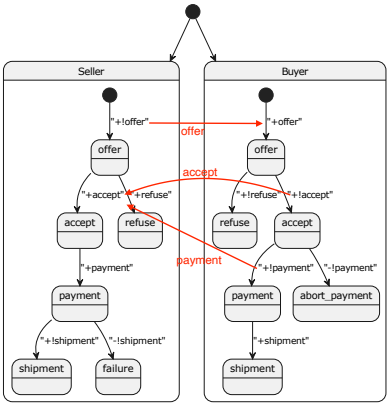
the protocol and it constrains the receiving of the different messages. This is needed to implement the minimality of the implementation, that is, an agent should send a message (or consider a received message) only when the protocol state allows it. Goals for sending (tackling) messages that are not foreseen by the protocol specification for that state are avoided. The following code (from Buyer) shows the use of the *step* belief:

```
1 +!accept(Id, Item, Price)[receiver(Seller)] : step(Id, offer) <-
2   -step(Id, offer); +step(Id, accept); ...
3 +!accept(Id, Item, Price)[source(Seller)] : not step(Id, offer) <- ...
```

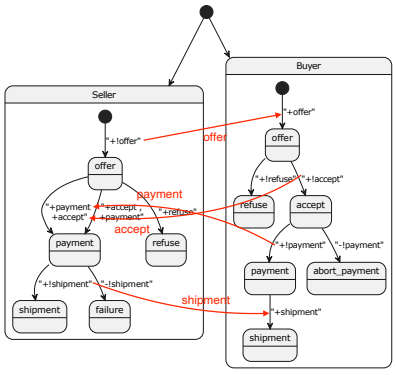
The first plan says that accept is executed at the step "offer". Its execution changes the step, passing from "offer" to "accept". The second plan specifies that accept can be executed only at step "offer".



(a) Protocol state for Basic transaction iteration.



(b) Protocol state for Basic transaction iteration: a problem.



(c) Protocol state for Basic transaction iteration: a better solution.

Fig. 2: Protocol state for the Basic transaction iteration of EBusiness protocol.

Figure 2a shows the protocol states for Seller and Buyer for the *Basic transaction* iteration. Specifically, the protocol specification is shown in the middle. On the left and right there are the state automata for the Buyer and the Seller roles, derived from the protocol specification in Figure 1a. The red arrows capture the message exchanges.

Dealing with asynchronicity. If the programmer realizes a trivial implementation of the state automata, a problem arises (see Figure 2b). Message exchange is asynchronous, that is, the sender does not wait for the receiver to receive a message before sending the next one, to guarantee the agents' autonomy. Thus, it may happen that the Buyer (sender) is faster than the Seller (receiver) and sends, one after the other, both *accept* and *payment* before the Seller processes the *accept*. However, the protocol specifies that *accept* is allowed by protocol state *offer*, while *payment* is allowed by protocol state *accept*. *Payment* is received in the wrong protocol state and, for minimality, it is discarded. This is not the only problem. It may, for instance, also happen that *payment* is received before *accept*, even though the two are sent in the right order because of the transport mechanisms. A better solution is to implement Seller in a way that avoids possible intermediate states (`accept` in our case, see figure 2b) in the reception of the two sequential messages from Buyer, keeping only the state before the sequence and that at the end (`offer` and `payment in our case`). The solution is shown in Figure 2c. Here, we account for both possible sequences *first payment then accept* and *first accept then payment* as single transitions from state *offer* to state *payment*. The implementation consists of two plans, one causes the transition from `offer` to *payment*, when *payment* is received and *accept* had been received earlier. The second plan causes the same transition when the message order is inverted. Here an excerpt of the Seller's code:

```
1 +accept(Id, Item, Price, Decision)[source(Buyer)] :
2   step(Id, offer) & payment(Id, Price, Decision, AmountC, Choice)[source(
        Buyer)]
3   <- ...
4      -step(Id, offer); +step(Id, payment);
5      !shipment(Id, Price, Decision)[receiver(Buyer)].
6 ...
7 +payment(Id, Price, Decision, AmountC)[source(Buyer)] :
8   step(Id, offer) & accept(Id, Item, Price, Decision)[source(Buyer)]
9   <- ...
10     -step(Id, offer); +step(Id, payment);
11     !shipment(Id, Price, Decision)[receiver(Buyer)].
```

The solution works but the Jason plans will be qualitatively more complex. In fact, as shown above, the plan context not only considers the protocol state (belief *step*), but also information about other messages making the agents' implementations strongly coupled and the agents less autonomous. Should the protocol be enriched with a new message, all of the plan contexts would be updated. The problem we have explained generalizes to the many parties case when one of the roles waits for messages that come from many other roles.

Of course, the problem could be solved by making the message exchange synchronous, but this would have the consequence of making the agent executions strongly coupled.

Coupling between history and control flow. In general, as the degree of asynchronicity increases, developers are forced to rely more heavily on the history of previously received messages (i.e., the order in which they were received) to reconstruct the current context of interaction. As a consequence, the logic governing message emission often depends on extracting information from messages that have already arrived. This tight coupling between message history and control flow significantly complicates the implementation and makes the code increasingly difficult to maintain across subsequent iterations. Let us start with iteration 2, where a third role is introduced: an intermediary for the payment (Bank), see Figure 1b. Here, payment can either be from the Buyer to the Seller or from the Bank to the Seller. The involved, alternative messages are *payment*, when the Buyer direct pays the Seller, and *transfer*, when the Bank is instructed to perform the payment. From the Seller's perspective a new problem arises: besides the messages *accept* and *payment*, also *transfer* is to be tackled but *transfer* and *payment* are mutually exclusive. Specifically, the allowed sequences are *first accept then payment*, *first accept then transfer*, *first payment then accept*, *first transfer then accept*. From a programmer's perspective, the plan to tackle the case *first accept then payment* can be re-used but the plan for tackling *first payment then accept* must be modified to include also the transfer option (*first payment or transfer then accept*), while a third plan must be added to tackle the last case (*first accept then transfer*). Here is an excerpt of the code:

```
+accept(Id, Item, Price, Decision)[source(Buyer)] :
  step(Id, offer) &
  ( payment(Id, Price, Decision, AmountC, Choice)[source(Buyer)] |
    transfer(Id, Price, Decision, Details, AmountT)[source(Bank)] ) <-
  -step(Id, offer); +step(Id, payment);
  !shipment(Id, Price, Decision)[receiver(Buyer)].
...
+payment(Id, Price, Decision, AmountC, Choice)[source(Buyer)] :
  step(Id, offer) &
  accept(Id, Item, Price, Decision)[source(Buyer)] <-
  -step(Id, offer); +step(Id, payment);
  !shipment(Id, Price, Decision)[receiver(Buyer)].
...
+transfer(Id, Price, Decision, Details, AmountT)[source(Bank)] :
  step(Id, offer) &
  accept(Id, Item, Price, Decision)[source(Buyer)] <-
  -step(Id, offer); +step(Id, payment);
  !shipment(Id, Price, Decision)[receiver(Buyer)].
```

The problem becomes even bigger when in iteration 4 we enrich the protocol by allowing the Buyer to optionally require a receipt after the payment (or transfer), see Figures 1d (sequence diagram) and 3 (protocol states). In this case, it is not just a question of sequencing the messages but some message might be present as well as not. Since the agent cannot foresee whether optional messages will arrive, it must be ready to tackle a possible reception even after many state transitions. Concerning the example, in practice we have three messages (one optional) that may arrive in any order from Buyer to Seller. The Seller needs plans to tackle the cases in which the optional *receipt-request* is not received (that allow to move to *shipment* when both *payment* and *accept* are arrived in whatever order), and plans to tackle also *receipt-request*, multiplying all the previous plans to make new versions that handle the optional message. Note, however, that

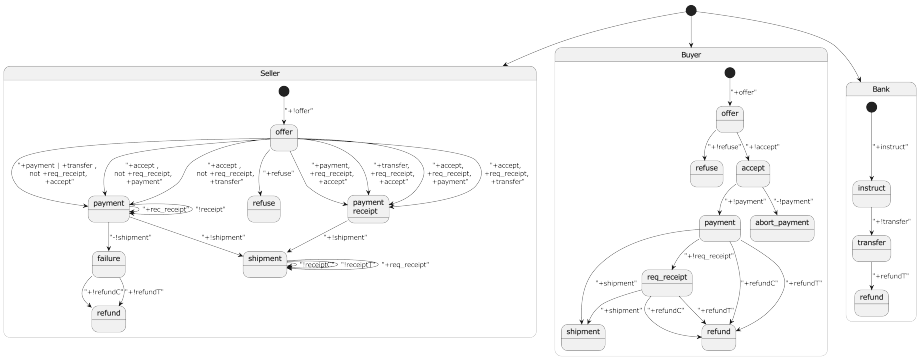


Fig. 3: Protocol states at iteration 4.

Seller cannot be sure that *receipt-request* will never arrive, thus, also the plans for tackling *accept* and *payment* "only" must be complicated because of the possible future arrival of such message. The result is the protocol in Figure 3.

Dealing with concurrent instances. Another problem concerns the possible presence of many concurrent instances of a same protocol. To support this case, a common engineering solution, used also in [7, Sec. 6.3, page 134], is to associate a session identifier with each exchanged message. However, this approach introduces additional complications when multiple messages from the same sender are permitted, or when identical messages originating from different senders are allowed. Addressing these cases typically requires the introduction of increasingly sophisticated and ad hoc identification mechanisms, further increasing system complexity and maintenance costs. As an example, the *id* parameter in the previous code samples serves this purpose.

Summary. Asynchronous message exchanges and multi-party interactions are inherently more difficult to design and implement, and they are more prone to subtle errors. In practice, this often leads to incompatibilities between agents, as message schemas become entangled with business logic. Moreover, the need for programmers to explicitly maintain the protocol state (often via manually engineered state machines) results in rigid implementations that are difficult to extend or adapt. Such limitations hinder the incremental and iterative development process discussed earlier, as even minor extensions to an interaction often require invasive changes to existing implementations and compromise previously validated behavior. This difficulty stems from the implicit encoding of protocol state, message ordering assumptions, and information dependencies within application-level control logic. As a consequence, evolving a protocol becomes tightly coupled with reengineering its implementation. The following code shows how information about the protocol state (lines 2 and 10 in particular) are used together with information about the agent's control logic (lines 3, 4 and 11, 12) in deciding how to behave:

Listing 1: Shipment in Jason.

```
+!shipment(Id, Price, Decision)[receiver(Buyer)] :
  ( step(Id, payment) | step(Id, payment_receipt) ) &
  accept(Id, Item, Price, Decision)[source(Buyer)] &
  in_stock(Item) <- ...
  .send(Buyer, tell, shipment(Id, Item, Price, Decision, Status));
  -step(Id, payment); +step(Id, shipment);
  +shipment(Id, Item, Price, Decision, Status)[receiver(Buyer)].
...
+!shipment(Id, Price, Decision)[receiver(Buyer)] :
  ( step(Id, payment) | step(Id, payment_receipt) ) &
  accept(Id, Item, Price, Decision)[source(Buyer)] &
  not in_stock(Item) <-
  -step(Id, payment); +step(Id, failure);
  !refund(Id, Item, Price)[receiver(Buyer)].
```

As will be shown in the remainder of this paper, Orpheus addresses these challenges by adopting information protocols based on BSPL, in which roles, information causality, and interaction constraints are made explicit at the protocol level. By externalizing these concerns from procedural code, Orpheus supports principled protocol evolution and facilitates incremental refinement, alleviating many of the issues highlighted in asynchronous and multi-party settings.

4 EBusiness in Orpheus

This section provides a little background on the way Orpheus operationalizes the information-centric view of interaction of BSPL [17], where interaction is defined in terms of *information flow* rather than message sequences. Afterwards, it introduces Orpheus programming model. Orpheus focuses not on reactions to incoming messages but on computing which messages are *enabled* (to be sent), given the protocol semantics and the local state. Doing so yields flexibility in agent decisions while abstracting out reasoning about the protocol into automatically generated code. The protocol states, that we have seen when talking about Jason, are substituted by the "states" of message enablements. The advantage is the seamless integration into the agents life-cycle (percept-deliberate-act): enabled messages will amount to new options among which the agent will deliberate how to act in order to achieve its goal. The set of enabled options captures the state of the protocol.

Background on BSPL. A BSPL protocol, Listing 2 reports an example, specifies the *information* that may be exchanged among agents and the constraints governing its production and use, without prescribing a specific ordering of messages. This declarative view decouples interaction semantics from control flow, enabling flexible and asynchronous coordination. It consists of a set of *roles* and a set of *message schemas*. Each message schema is defined by a name, a sender role, a receiver role, and a set of parameters. Some parameters are designated as *key parameters*, which together form a composite key that uniquely identifies protocol enactments. The remaining parameters represent information that may be exchanged during interaction. Parameters are annotated to capture how information flows. Parameters marked as ⌜in⌝ *must be known* to the sender at the

time of message emission, whereas parameters marked as ⌜out⌝ *must be unknown and become known as a result of sending the message.*

A *message instance* is a tuple of parameter bindings associated with a message schema. A message instance is said to be *enabled* for an agent when: (i) all its ⌜in⌝ parameters are bound and known to the agent, and (ii) all its ⌜out⌝ parameters are unbound. An agent *knows* a binding for a parameter if it has previously sent or received a message containing that binding.

Interaction constraints are expressed in terms of information-based *causality* and *integrity*. Causality is enforced by requiring that all ⌜in⌝ parameters of a message instance be bound before the message can be sent. Integrity is enforced by constraining parameter bindings across message instances. Specifically, no two message instances that share overlapping key parameter bindings may introduce different bindings for the same non-key parameter. As a result, once a piece of information is produced, it cannot be contradicted later in the interaction. These constraints ensure that interaction correctness emerges from information consistency rather than from adherence to predefined message orders. Crucially, these constraints are locally determinable. An agent can decide whether a message may be sent or accepted based solely on the information it currently knows, without requiring global synchronization or knowledge of the full interaction history. This property supports asynchronous communication and reduces coupling between agents. Decoupling allows agents to reason proactively about possible actions, querying for enabled messages rather than reacting to message arrivals. This will be a key aspect in facing the benchmark.

Listing 2: *EBusiness* protocol: basic transaction in BSPL.

```
1 EBusiness {
2   role Buyer, Seller
3   parameter out ID key, out item, out price, out status
4   Seller -> out Buyer : offer [out ID key, out item, out price]
5   Buyer -> in Seller : refuse [in ID key, in item, in price, out decision,
        out status]
6   Buyer -> in Seller : accept [in ID key, in item, in price, out decision]
7   Buyer -> in Seller : payment [in ID key, in price, in decision, out
        amountC]
8   Seller -> in Buyer : shipment [in ID key, in item, in price, in decision,
        out status]  }
```

Concerning the example in Listing 2, the protocol is made of two roles, Buyer and Seller, `key`, is the identifier, and five actions are involved, `offer`, `accept`, `refuse`, `payment`, and `shipment`. The protocol execution ends when all protocol parameters are bound (`key`, `item`, `price`, `status`). Action `offer`, from Seller to Buyer, is the only one that produces information without requiring any: in other terms, such information is not subject to causal dependencies, and the agent can freely decide whether doing an offer. Its execution, however, binds its parameters. Instead, `accept` and `refuse` are in mutual exclusion: this is captured by both having `decision` as an ⌜out⌝ parameter, the first that binds it blocks the other. `refuse` terminates the execution because it binds `status` when all other protocol parameters have been bound. The same holds for `shipment`. Instead, `payment` is enabled after `accept` because it needs not only `key` and `price` but also `decision`, which is produced by `accept`.

Orpheus Programmng Model. In Orpheus, an agent's beliefs encode its view of the protocol execution, referred to as the agent's *local state*. The local state records all parameter bindings known to the agent as a result of message exchanges. When an incoming message is received, it is incorporated into the local state only if its bindings are consistent with the existing state. That is, no other binding is already known for any of its parameters relative to the corresponding key. Similarly, when an agent attempts to send one or more messages, the attempt succeeds only if the completed message instances are mutually consistent in their bindings. Upon successful emission, the sent messages are added to the local state. By enforcing consistency locally and incrementally, Orpheus eliminates the need for explicit protocol state machines and ordered message delivery. Messages may arrive at any time, and agents can safely process them as long as information constraints are satisfied. This design directly addresses the rigidity and fragility of traditional message-centric AOPLs.

Role adapters are a key abstraction introduced by Orpheus to avoid providing programmers with send and receive primitives, which would undermine the benefits of the high-level declarative specification of interaction of BSPL:

- Given a BSPL protocol, the Orpheus Tool automatically generates, for each role, a set of role-specific queries and plans that constitute the role adapter. The adapter encapsulates all interaction-related concerns, including message validation, enablement checking, and consistency enforcement.
- Queries generated by the adapter are used to compute enabled message instances and implement the local state.
- Plans generated by the adapter validate messages before emission and upon reception, ensuring conformance with protocol constraints.

As a result, all low-level communication logic is isolated within generated code. This design enforces a clear separation of concerns between: (i) agent-internal reasoning and application logic, and (ii) coordination and interaction constraints. By localizing interaction logic within role adapters, Orpheus mitigates cascading effects caused by changes to interaction requirements, supporting iterative and incremental development.

The abstractions described above fundamentally shape the agent programming model in Orpheus. Agents are programmed to pursue goals by querying for enabled messages, completing them by producing bindings for output parameters, and attempting to send them. Incoming messages are incorporated opportunistically, subject to consistency checks. This programming model aligns naturally with BDI agent languages such as Jason. Goals determine which interactions are relevant, beliefs encode the local protocol state, and intentions drive the completion and emission of enabled messages. Indeed, an Orpheus agent is a Jason agent, whose beliefs capture its state; the part of the state concerning sent and received messages is the local state. An agent's internal logic is expressed as plans. Based on the protocol, the Orpheus tool generates role-specific adapters (Jason code, i.e., plans) that compute enabled messages and validate messages before emission and upon reception. By abstracting protocol reasoning into gen-

erated role adapters, Orpheus allows agent programmers to focus on cognitive decision-making while preserving both cognitive autonomy and social autonomy.

Orpheus programming model is based on message *enablement*: the developer specifies plans for emitting enabled messages. In contrast to the reactive model, plans are decoupled from message receptions. Listing 3 captures the main aspects [2]. Suppose an agent wants to achieve goal g. It has (among others) a plan that involves sending certain messages. The agent queries if the messages it would like to send amount to enabled instances. If so, it *completes* them by producing bindings for their ⌜out⌝ parameters, and *attempts* to send them all in one shot. An attempt is successful if the completed messages are mutually consistent in their bindings. The sent messages are added to the local state. A received message is added to the local state if it is consistent with the local state.

Listing 3: Plan pattern in Orpheus from [2]. Orpheus supplied primitives are in blue; in red the code the programmer must write.

```
+!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
```

As an example, Listing 4 applies the pattern of Listing 3 in implementing `shipment` (see Listing 1 for comparison).

Listing 4: Shipment in Orpheus.

```
+!select_and_handle_message([shipment(Id, Item, Price, Decision, out)[
    receiver(Buyer)] | _]) :
    enabled(shipment(Id, Item, Price, Decision, out)[receiver(Buyer)]) &
    in_stock(Item)
 <- !complete(shipment(Id, Item, Price, Decision, Status)[receiver(Buyer)])
     ;
    !attempt(shipment(Id, Item, Price, Decision, Status)[receiver(Buyer)]).
...
+!select_and_handle_message([shipment(Id, Item, Price, Decision, out)[
    receiver(Buyer)] | Rest_enabled_messages]) :
    enabled(shipment(Id, Item, Price, Decision, out)[receiver(Buyer)]) &
    not in_stock(Item)
 <- !select_and_handle_message(Rest_enabled_messages).
```

`select_and_handle_message` is a predicate for selecting one of the options among the enabled messages. The agent can use `shipment` if it is enabled and the needed items are in stock. If this doe not happen, another option will be chosen (second plan). If we compare this code with the Jason analogous code, the context not only checked the items were in stock, but also that the execution was in the right state, mingling business logic and protocol control flow. Here, this does not happen. Note that no plan is needed for receiving messages (reception is transparent to the agent).

5 Comparison

Role		Jason				Orpheus			
		1	2	3	4	1	2	3	4
Buyer	Lines	169	191	203	252	155	185	185	212
	New lines		22 (13.02%)	12 (6.28%)	49 (24.14%)		30 (19.35%)	0 (0%)	25 (14.59%)
	Mod. lines		8 (4.73%)	0 (0%)	6 (2.96%)		8 (5.16%)	0 (0%)	0 (0%)
Seller	Lines	144	160	208	311	106	106	154	186
	New lines		15 (10.42%)	48 (30%)	103 (49.52%)		0 (0%)	48 (45.28%)	32 (20.78%)
	Mod. lines		4 (2.78%)	1 (0.63%)	8 (3.85%)		0 (0%)	0 (0%)	0 (0%)
Bank	Lines	–	52	64	64	–	58	58	58
	New lines			12 (23.08%)	0 (0%)			0 (0%)	0 (0%)
	Mod. lines			0 (0%)	0 (0%)			0 (0%)	0 (0%)
Total	Lines	313	403	475	627	261	349	397	456
	New Lines		37 (11.82%)	72 (17.87%)	152 (32%)		30 (11.49%)	48 (13.75%)	59 (14.86%)
	Mod. Lines		12 (3.83%)	1 (0.25%)	14 (2.95%)		8 (3.07%)	0 (0%)	0 (0%)

Tables **??** and **??** report the performance of the two implementations, computed according our reference measures, that are the number of the added/modified code lines and the number of the added/modified plans. Performance is reported in two ways: the tables show both the counts of lines/plans that were added/modified with respect to the previous iteration, and the percentages of lines/plans that were added/modified with respect to the previous iteration.

Iteration 1: Basic transaction. The implementation of the simplest version of the EBusinss protocol amounts to the reference point, with respect to which all counts are computed. There are only two roles, Buyer and Seller, whose implementations count 169 and 144 code lines respectively in Jason, and 155 and 106 in Orpheus. The number of plans are 19 and 18 respectively in the case of Jason, and 18 and 14 in Orpheus. Orpheus is a little bit more compact on this case because there are no plans for dealing with message reception. The Seller is the role that shrinks the most because it is the role that, in the protocol schema, receives more messages.

Iteration 2: Introduction of an intermediary. This is the iteration that adds a new role: the Bank. Besides paying directly, a Buyer may choose to instruct the Bank to execute the transfer on their behalf. Besides adding the plans needed by the Bank to carry out its task (7 plans in Jason, 8 in Orpheus), we see a difference. In the case of *Jason*, the Buyer agent's code had to be enriched with 2 new plans and 3 plans had to be modified. Moreover, also 2 of the Seller's plans had to be modified and 2 added. Instead, in the case of *Orpheus*, *only the Buyer had to be modified*, by adding 4 plans and modifying 4. Concerning the number of code lines, the Jason implementation of the Buyer was enriched with 22 new lines and 8 were modified, while that of the Seller was enriched

Role		Jason				Orpheus			
		1	2	3	4	1	2	3	4
Buyer	Plans	19	211	23	30	18	22	22	25
	New Plans		2 (10.53%)	2 (9.52%)	7 (30.43%)		4 (22.22%)	0 (0%)	4 (18.18%)
	Mod. Plans		3 (15.79%)	0 (0%)	0 (2.96%)		4 (22.22%)	0 (0%)	0 (0%)
Seller	Plans	18	20	27	40	14	14	22	28
	New Plans		2 (11.11%)	7 (35%)	13 (48.15%)		0 (0%)	8 (57.14%)	6 (27.27%)
	Mod. Plans		2 (11.11%)	1 (5%)	7 (25.93%)		0 (0%)	0 (0%)	0 (0%)
Bank	Plans	–	7	9	9	–	8	8	8
	New Plans			2 (28.57%)	0 (0%)			0 (0%)	0 (0%)
	Mod. Plans			0 (0%)	0 (0%)			0 (0%)	0 (0%)
Total	Plans	37	48	59	79	32	44	52	61
	New Plans		4 (10.81%)	11 (12.92%)	20 (33.9%)		4 (12.5%)	8 (18.18%)	10 (19.23%)
	Mod. Plans		5 (13.51%)	1 (2.08%)	7 (11.86%)		4 (12.5%)	0 (0%)	0 (0%)

with 15 new lines and 8 were modified; instead, in Orpheus, only the Buyer agent's code was enriched with 30 new lines and other 8 lines were modified. Concerning the kinds of updates that were implemented, in Orpheus additions concern only the plans needed to tackle `instruct`. The modifications concern the addition of the new parameter `choice` to payment. In Jason, the changes to the Buyer are similar (both `instruct` and `choice` had to be added). In this case, however, the Seller had to be modified as well by adding new contextual predicates, aimed at taking care of the entanglement between history and control flow (see Section 3). In particular, this was done to handle the case in which `accept` make the state progress if `payment` or `transfer` was received. Not only in Orpheus this is not needed but, qualitatively, this is a heavier modification (with respect to parameter addition) because it amounts to a change to the control logic of the protocol implementation.

Iteration 3: Refund mechanism. This iteration is built upon iteration 2 by adding a new possibility for the Seller, who may issue a refund in case the shipment cannot be done. The difficulty is that the refund should go either to the Buyer or to the Bank depending on who did the payment. Also in this case, the modification applied in the Orpheus case concern a single agent, the Seller this time, which is enriched by adding 8 plans (for a total of 48 lines). In the case of Jason modifications concerned all three agents: 2 new plans to the Buyer (12 new lines of code), 7 to the Seller (48 new lines), and 2 to the Bank (12 new lines). In Jason one further plan is modified by adding a new line, that amounts to the activation of a `refund` goal, done in case shipment fails. In Orpheus this is not necessary because enablement already offers refund as an alternative to shipment.

Iteration 4: Proof of receipt. The last addition is the optional possibility, for the Buyer, to request a receipt from the Seller after the payment. The difference appears more clearly in the last case. In both cases only the Buyer and the Seller are modified. While in Orpheus the Buyer is enriched by adding 4 plans

(25 lines) and the Seller by adding 6 plans (32 lines) and no other line of code is modified, in Jason the Buyer gets 7 new plans (49 lines) and the Seller 13 new plans (103 lines), but 7 plans of the Buyer (6 lines) and 7 of the Seller (8 lines) are also modified. With reference to Figure 3, in Jason the Buyer modifies some plans because it needs to add, after payment, the decision whether to also ask for a receipt or not. Other plans are modified because the states in which it receives the shipment can either be the state that accounts that the payment has occurred or the one in which, after payment, the Buyer also asked for a receipt. Similarly for refund: the Buyer can receive it after payment or after payment and receipt request. In Orpheus, the only modification to the Buyer is the addition of the plans that are needed for managing the enablement of a receipt request. Requesting a receipt becomes one of the enabled actions, which the agent can deliberate to execute. Concerning the Seller, in Jason previous plans were modified because of the entanglement of history and control flow. The new plans were added in part for managing receipt requests and in part for managing events that were already tackled. The latter are necessary because of the optionality of receipt requests: if the request was not done, an already existing plan can be used, but in case it is, indeed, requested, then a new plan needs to be devised that handles both shipment and receipt production.

Figure 4 shows the commented performances on two plots, the one on the left concerning code lines, that on the right concerning plans. We are aware that a single case is insufficient to draw general conclusions, but it is interesting to observe how lines diverge quickly along the iterations, in particular, for what concerns the number of plans the programmer needs to update. In Orpheus it looks flatter while in Jason it seems to grow faster. This is coherent with the observations of the previous sections, where we explained the need of multiplying plans to take care of asynchronicity, of alternative sequences, and of concurrency in Jason.

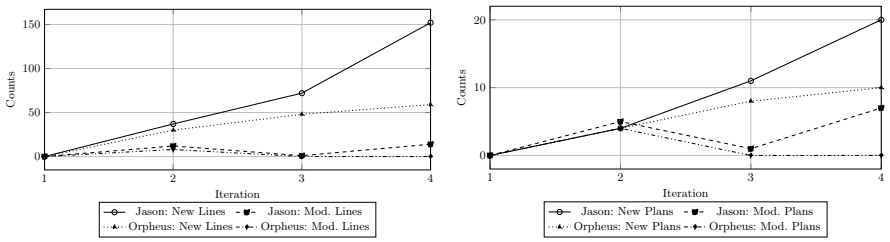


Fig. 4: Left: plot of new lines and modified lines in Jason and Orpheus along the four iterations; Right: plot of new plans and modified plans in Jason and Orpheus along the four iterations

6 Conclusions

In this work we have compared the effort of incrementally modifying interaction protocols in Jason and in Orpheus. The former adopts an agent programming model that is message-centric, while the latter is enablement-centric, practically showing that for what concerns incremental iterative development, Orpheus requires a reduced effort of programmers. This study is limited by considering a single protocol, but we believe that the results are pretty general because they concern pivotal aspects of the two programming models.

Concerning the future, we are working on an extended version of JADE [3] that replaces traditional message queues with BSPL role adapters, making their use fully transparent. JADE was chosen because it is already one of the most used middleware for AOPL. Besides software frameworks, two other challenges need to be tackled to spread the use of BSPL: 1) a methodology for implementing agents that exploit BSPL protocols; 2) a methodology for developing BSPL protocols themselves. About the second point, it is worthwhile to mention two contributions. Langshaw [20] is a declarative language which allows the specification of a protocol in terms of communicative actions. It adopts a centralized and synchronous perspective in order to simplify the modeling task, but a Langshaw model can be compiled into an asynchronous BSPL protocol. We, on our hand, are develping a tool that build BSPL protocols automatically, based on requirements specification [1].

Acknowledgments. The authors would like to thank Amit Chopra for the insightful discussions and advice.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Baldoni, M., Baroglio, C., Micalizo, R.: Castor and Pollux: A First Demonstration of a BSPL Protocol Discovery Tool (2026), submitted
2. Baldoni, M., Christie V, S.H., Singh, M.P., Chopra, A.K.: Orpheus: Engineering multiagent systems via communicating agents. In: Proceedings of the 39th AAAI Conference on Artificial Intelligence (AAAI). pp. 1–9. AAAI, Philadelphia (Feb 2025)
3. Bellifemine, F., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley, Chichester, UK (2007). https://doi.org/10.1002/9780470058411
4. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley-Blackwell (2007)
5. Bergenti, F., Iotti, E., Monica, S., Poggi, A.: Agent-oriented model-driven development for JADE with the JADEL programming language. Computer Languages, Systems & Structures **50**, 142–158 (Dec 2017). https://doi.org/10.1016/j.cl.2017.06.001
6. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with JaCaMo. Science of Computer Programming **78**(6), 747–761 (Jun 2013). https://doi.org/10.1016/j.scico.2011.10.004

7. Bordini, R.H., Hubner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason. Wiley-Interscience (2007)
8. Chopra, A.K., Baldoni, M., Christie V, S.H., Singh, M.P.: Azorus: Commitments over protocols for BDI agents. In: Proceedings of the 24th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). IFAAMAS, Detroit (May 2025)
9. Chopra, A.K., Christie V, S.H., Singh, M.P.: An evaluation of communication protocol languages for engineering multiagent systems. Journal of Artificial Intelligence Research (JAIR) **69**, 1351–1393 (Dec 2020). https://doi.org/10.1613/jair.1.12212
10. Christie V, S.H., Chopra, A.K., Singh, M.P.: Mandrake: Multiagent systems as a basis for programming fault-tolerant decentralized applications. Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS) **36**(1), 16:1–16:30 (Apr 2022). https://doi.org/10.1007/s10458-021-09540-8
11. Christie V, S.H., Singh, M.P., Chopra, A.K.: Kiko: Programming agents to enact interaction protocols. In: Proceedings of the 22nd International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 1154–1163. IFAAMAS, London (May 2023). https://doi.org/10.5555/3545946.3598758
12. Finin, T., Fritzson, R., McKay, D., McEntire, R.: KQML as an agent communication language. In: Proceedings of the 3rd International Conference on Information and Knowledge Management. pp. 456–463. ACM Press, Gaithersburg, Maryland (Dec 1994). https://doi.org/10.1145/191246.191322
13. Foundation for intelligent physical agents (FIPA) specification (1998), www.fipa.org
14. Huget, M.P., Odell, J.: Representing agent interaction protocols with agent UML. In: Proceedings of the 5th International Workshop on Agent-Oriented Software Engineering (AOSE). Lecture Notes in Computer Science, vol. 3382, pp. 16–30. Springer, New York (Jul 2004). https://doi.org/10.1007/978-3-540-30578-1_2
15. Larman, C.: Applying UML And Patterns: An Introduction To Object-Oriented Analysis And Design And Iterative Development. Prentice Hall (2004)
16. Rodriguez, S., Gaud, N., Galland, S.: 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). vol. 3, pp. 103–110 (2014)
17. Singh, M.P.: 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). pp. 491–498. IFAAMAS, Taipei (May 2011). https://doi.org/10.5555/2031678.2031687
18. Singh, M.P.: Semantics and verification of information-based protocols. In: Proceedings of the 11th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS). pp. 1149–1156. IFAAMAS, Valencia, Spain (Jun 2012). https://doi.org/10.5555/2343776.2343861
19. Singh, M.P., Christie V, S.H.: Tango: Declarative semantics for multiagent communication protocols. In: Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI). pp. 391–397. IJCAI, Online (Aug 2021). https://doi.org/10.24963/ijcai.2021/55
20. Singh, M.P., Christie V, S.H., Chopra, A.K.: Langshaw: Declarative interaction protocols based on sayso and conflict. In: Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI). pp. 202–210. IJCAI, Jeju, Korea (Aug 2024). https://doi.org/10.24963/ijcai.2024/23

21. Sommerville, I.: Software engineering. Pearson (2015)
22. Vieira, R., Moreira, Á.F., Wooldridge, M.J., Bordini, R.H.: On the formal semantics of speech-act based communication in an agent-oriented programming language. Journal of Artificial Intelligence Research (JAIR) **29**, 221–267 (Jun 2007). https://doi.org/10.1613/jair.2221
23. Winikoff, M.: Challenges and directions for engineering multi-agent systems. CoRR **abs/1209.1428** (2012)