Orpheus: Engineering Multiagent Systems via Communicating Agents

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

[1]Dipartimento di Informatica, Università degli Studi di Torino, Corso Svizzera 185, 10149 Torino, Italy
[2]Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA
[3]School of Computing and Communications, Lancaster University, Lancaster LA1 4WA, UK
baldoni@di.unito.it, schrist@ncsu.edu, mpsingh@ncsu.edu, amit.chopra@lancaster.ac.uk

Abstract

We propose Orpheus, a novel programming model for communicating agents based on information protocols and realized using cognitive programming. Whereas traditional models are focused on reactions to handle incoming messages, Orpheus supports organizing the internal logic of an agent based on its goals. We give an operational semantics for Orpheus and implement this semantics in an adapter to help build agents. We use the adapter to demonstrate how Orpheus simplifies the programming of decentralized multiagent systems compared to the reactive programming model.

Code — https://gitlab.com/masr

1 Introduction

Interaction between autonomous agents is the hallmark of a multiagent system (MAS). An *interaction protocol* models the communication constraints between agents. Engineering MAS based on protocols offers key benefits in domains such as business (Lichtenstein et al. 2024) and the Internet of Things (Singh and Chopra 2017). One, protocols enable realizing MAS in a decentralized manner, i.e., without relying on a distinguished locus of state or control. Two, protocols enable structuring an agent's implementation by cleanly separating the coordination aspects of its communications from its internal logic. Three, protocols streamline coordination, reducing agent complexity and avoiding programming errors pertaining to communication. Four, protocols support loose coupling: changes in one agent's implementation do not affect the implementations of others.

Yet, current approaches fail to realize the above benefits of protocols when engineering flexible MAS. There are two reasons why this is so. **First,** prevalent protocol models are flawed. FIPA interaction protocols (FIPA 2003) are specified in Agent UML (AUML) (Odell, Parunak, and Bauer 2001), a language similar to UML sequence diagrams. An agent plays a role in a protocol and communicates accordingly. However, FIPA and UML provide no formal model of the protocol. Lacking a semantics, we cannot reason about information to (1) provide a principled programming model and (2) check whether each message respects the protocol semantics. Statecharts, as in ASEME (Spanoudakis and Moraitis 2022), are enhanced state machines. ASEME, a communicating state-machine model, doesn't accommodate multi (≥ 3) party protocols over asynchronous communication. ASEME role skeletons move in lockstep with matching send and receive states in different roles. Choreographies (Baldoni et al. 2006, 2009), a class of communicating state-machine model of interactions, share ASEME's limitations.

We adopt *information protocols*, a fully declarative and fully asynchronous model for communication because it yields highly flexible interactions between agents to undergird our contribution. Specifically, we adopt the Blindingly Simple Protocol Language (BSPL) (Singh 2011), for which there have been recent advances in formal semantics and verification tools (Singh and Christie 2021). For a detailed discussion of the advantages of information protocols over alternative models of interaction, such as choreographies, see (Chopra, Christie, and Singh 2020).

Second, prevalent programming models for communication are lacking. They are *reactive*, i.e., structured as separate message handlers for each incoming message (Desai et al. 2005), as in message-oriented middleware (Hohpe and Woolf 2004). However, doing so ignores the structuring provided by a protocol and considers each message independently. But, messages are *rarely* independent, and an agent usually needs to act based on its state, which depends on the messages it has received or sent. Thus, the agent reconstructs the necessary state computation (1) tied up with the requisite internal reasoning and (2) in more than one place, based on what message emissions and receptions can lead to that state. Kiko (Christie, Singh, and Chopra 2023) shares our focus on information protocols, though for Python agents. Kiko computes not the relevant messages queried by agent logic but (somewhat unnecessarily) all messages that are *enabled* (i.e., legal to complete and emit).

Leading cognitive programming models, e.g., Jason (Vieira et al. 2007) and JADEL (Bergenti et al. 2017), do not support protocols. Others, e.g., (Rooney, Collier, and O'Hare 2004) support protocols but not information protocols. JaCaMo (Boissier et al. 2013) includes Jason but notably acknowledges (p. 748, n1) its lack of protocol-based programming abstractions; it deemphasizes messaging in favor of operations on centralized artifacts reminiscent of Web services. JADE (Bellifemine, Caire, and Greenwood 2007) supports implementing MAS based on the rigid, as ex-

plained above, FIPA interaction protocols; moreover, JADE does not support custom protocols. Prosocs (Bracciali et al. 2006) provides a reactive programming model; it handles communication but without reasoning about protocols, similar to Desai et al. (2005). Winikoff (2012) identifies high-level abstractions for flexible interactions as a crucial challenge for agent programming languages.

Contributions and Novelty. We contribute *Orpheus* (in Greek mythology, a poet and a companion of Jason on his adventures), a programming model for multiagent systems. Orpheus unites two aspects of autonomy. One, cognitive autonomy, as reflected in an agent's goals and emphasized by approaches in the cognitive paradigm. Two, social autonomy, as reflected in an agent's dependence upon others and emphasized in the interaction-oriented paradigm (Sichman et al. 1994; Singh 1998; Castelfranchi 1998), here via information protocols. Orpheus structures agent code by providing an "API" that can be applied to advance the agent's goals. Specifically, it applies the protocol semantics to generate an adapter that (1) tracks state and computes the enabled messages at runtime, (2) abstracts away the actual emission and reception of messages, and (3) supports the integrity of communications. We provide Orpheus' formal operational semantics and an implementation in Jason.

Organization The rest of this paper is organized as follows. Section 2 introduces Jason and outlines the typical approach for developing protocol-based agents in Jason. Section 3 highlights the shortcomings of such approaches. Section 4 introduces information protocols. Section 5 introduces our proposed architecture and programming model. Section 6 describes a conceptual evaluation of our approach, showing key distinctions and how it meets important criteria. Section 7 presents a formalization of our approach. Sections 8 present our conclusions and some promising directions engaging with additional relevant literature.

2 MAS Development in a BDI Approach

We adopt Jason as an exemplar BDI approach, but the limitations we identify apply to other such approaches. Though Jason doesn't support protocols, an agent programmer is typically guided by an informal protocol specification, e.g., via a UML sequence diagram. We use NetBill (Sirbu 1997), a simple e-commerce protocol, as our running example.

Listing 1 implements an agent playing the MERCHANT role in NetBill. Each agent has a belief base and a plan library. Agents have two kinds of goals: *achievement* (marked **!**, used here) and *test* (marked **?**). Each plan has a triggering event, either the addition or the deletion of a belief or goal. Listing 1 shows two plans. A Jason plan is specified as *triggering event*: $\langle context \rangle \leftarrow \langle body \rangle$, where *triggering event* denotes the event the plan handles, the *context* specifies the circumstances when the plan could be used, and the *body* is the course of action to be taken. In Listing 1, the nbp_state predicate captures the (NetBill) protocol state. The plan triggered by adding message *request* from CUSTOMER to the beliefs (1) determines the Price via the predicate price, (2) sends the *quote*, and (3) adds an assertion to record the resulting protocol state in the belief base. Here, .send is a Jason builtin, and tell is a Jason message type based on KQML (Finin et al. 1994). The plan for *accept* is similar.

Listing 1: Jason snippet of a MERCHANT agent.

```
1: +request(Id, Item)[source(Customer)]
2:  : price(Item, Price)
3:  <- +nbp_state(Id, quoting);
4:     .send(Customer, tell, quote(Id, Item,
           Price)).
5: +accept(Id,Item,Price)[source(Customer)]
6:  : nbp_state(Id, quoting) &
7:    goods(Item, Goods)
8:  <- -nbp_state(Id, quoting);
9:     +nbp_state(Id, shipping);
10:    .send(Customer, tell, goods(Id, Item,
           Price, Goods)).
```

3 Shortcomings of Traditional Approaches

Listing 1 exemplifies reactive programming: a plan is executed to handle each received message, which depending on the agent's state, may result in the emission of a message. Listing 1 highlights three limitations (which we use to evaluate Orpheus) of the reactive programming model.

Incompatibilities between agents due to the message schemas being blended into the internal logic.

Semantic Errors due to a lack of a formal model—e.g., a CUSTOMER agent may send both *accept* and *reject*.

Inflexibility due to the programmer having to maintain the protocol state via a state machine, where the agents move in lockstep—and with no support for concurrency, asynchrony, messages arriving in unexpected orders, and interactions involving more than two agents.

Elaborating on the last point, as Listing 1 shows, the traditional approach encodes send-receive pairs into the agent code. Specifically, sending *quote* and *goods* are paired with receiving *request* and *accept*, respectively. But what if sending a message depends on the receipt of multiple messages that may arrive in any order? For example, an agent may send a message upon receiving messages from two other agents. Following the traditional approach, the programmer now must write as many plans as the number of messages that must be received, each triggered by the reception of one of those messages and whose context encodes the state where the other messages have already been received. We revisit this point in Section 6.4, where we introduce a protocol with more than two agents.

4 Background: Information Protocols

An information protocol (Singh 2011) specifies communication in a multiagent system and provides a basis for implementing its loosely coupled agents. Listing 2 illustrates BSPL via our running example, NetBill.

A BSPL protocol specifies *roles* and *message schemas*. A message schema has a name, a sender and a receiver role, and one or more parameters, including some designated ⌜key⌝. A *message instance* is a tuple of bindings for

Listing 2: Initial *NetBill* Protocol (goods before pay)

```
1: NetBill {
2:  roles M, C //Merchant, Customer
3:  parameters out ID key, out item, out done
4:  private decision, outcome, price, chit,
        shipped, cc
5:  C -> M: request[out ID key, out item]
6:  M -> C: quote[in ID key, in item, out
        price]
7:  C -> M: accept[in ID key, in item, in
        price, out decision, out outcome]
8:  C -> M: reject[in ID key, in item, in
        price, out decision, out done]
9:  M -> C: goods[in ID key, in item, in
        outcome, out shipped]
10: C -> M: epo[in ID key, in item, in price,
        in shipped, out cc]
11: M -> C: receipt[in ID key, in price, in
        cc, out chit, out done] }
```

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.

A role *knows* bindings for some parameters if it has sent or received messages with bindings for those parameters. Parameters adorned ⌜in⌝ must have bindings known to the sender when emitting a message. Parameters adorned ⌜out⌝ and ⌜nil⌝ (not shown) must not have bindings known to the sender when emitting a message; parameters with ⌜out⌝ become known then, but those with ⌜nil⌝ do not. By uniqueness, no two message instances with the same bindings for overlapping ⌜key⌝ parameters may have distinct bindings for common non-key parameters. Since bindings are introduced through ⌜out⌝ parameters, no two message instances may have overlapping key parameter bindings as well as a binding of the same ⌜out⌝ parameter. BSPL thus captures *causality* and *integrity* through information.

Message emissions are constrained only by causality and locally determinable integrity constraints; there is no separate control flow constraint. ID identifies enactments of *NetBill*. CUSTOMER may send a *request* at any time by generating a new binding for ID and some binding for item. To send an instance of the *quote* message, MERCHANT must know the bindings of ID and the correlated item and not know the binding of the correlated price; however, upon emitting the *quote*, it knows the binding of price. In Listing 2, *epo* (payment) may happen only after *goods* because *epo* has an information dependency on *goods* via shipped.

Notably, in BSPL, the emission of a message by an agent depends purely upon what information it has. Specifically, unlike traditional state machines, it is decoupled from having sent or received specific messages and their relative ordering. Indeed, a message may be received at any time in any relative order with respect to other messages, obviating the need for ordered-delivery communication services.

Orpheus preserves the flexibility accorded by BSPL—multiple operational paths (each a sequence of sends and receives) in a protocol may produce the same information. Orpheus leverages this information-based abstraction to save programmers from implementing low-level state machines and to address **Inflexibility**.

Orpheus leverages BSPL and uses a protocol (with a formal syntax and semantics due to BSPL) that specifies message schemas separately from agent code, thus addressing **Incompatibility** (Section 3).

5 The Orpheus Programming Model

An Orpheus agent is a Jason agent structured as Figure 1 shows. The agent's beliefs capture its state. The *local state* is the part of the state concerning sent and received messages. 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.

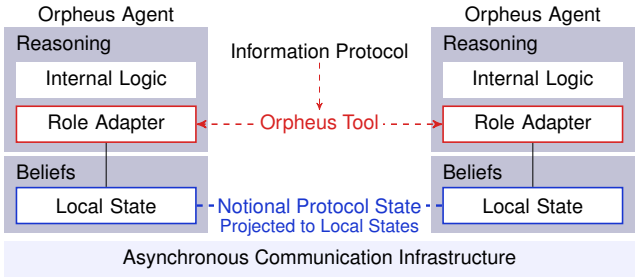

Figure 1: Orpheus maintains the agent's local state (the protocol state projected to the messages sent or received by the agent) as a set of beliefs in Jason. The Orpheus tooling generates an adapter for the role being played by the agent based on the information protocol. The agent's internal logic uses the adapter to send messages.

The Orpheus adapter applies BSPL semantics to verify if an incoming or outgoing message is *consistent* with the local state—no distinct binding is known for any of its parameters relative to the key—and adds it to the local state. Suppose Alice plays CUSTOMER and Bob plays MERCHANT. Then, if Alice's message *request[1, "fig"]* is consistent with her local state, the term *request("Alice", "Bob", 1, "fig")* is added to her local state. And, likewise, upon reception by Bob.

Orpheus focuses not on reactions to incoming messages but on computing messages *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 listings below highlight the Orpheus primitives for emphasis.

5.1 Enablement

A (full) instance of a message schema is a tuple of bindings for the ⌜in⌝ and ⌜out⌝ parameters (⌜nil⌝ parameters of the schema must have no bindings in an instance). Valid instances may be emitted. An *enabled* instance is partial: its ⌜in⌝ parameters are bound (their bindings must be known) and its ⌜out⌝ parameters are not bound (they must not be known). That is, given a local state, one can compute the enabled instances, which capture the potential emissions of the agent in that local state. For example, let MERCHANT's local

Listing 3: Plan pattern and Orpheus primitives.

```
+!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
```

Listing 4: Generated Jason code for computing enabled *quote*s; the code generated for other messages is similar.

```
enabled(quote(Id, Item, out)[receiver(
    Customer)])
 :- table_ID(Id) & table_item(Id, Item)
     & not (table_price(Id, Price))
     & table_Customer(Id, Customer).
```

state contain *request[1, "fig"]* meaning that it has received that *request* from the CUSTOMER. Then, the MERCHANT has one enabled instance in that state: *quote*[*1*, *"fig"*, price].

5.2 Enablement-Based Programming Model

Orpheus supports a novel programming model based on message *enablement*, in which the developer specifies plans for emitting enabled messages. In contrast to the reactive model, the plans are decoupled from message receptions. Listing 3 captures the main points. To achieve some agent-specific goal g, i.e., to meet its principal's requirements, 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. 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. In the listing, Orpheus primitives are in blue, except !complete, shown in red. Besides identifying g, the programmer is responsible only for the plan for !complete.

Listing 4 shows a plan generated for computing enabled messages for MERCHANT in *NetBill*. We maintain each parameter in its own "table" (relative to the key); the bindings for the ⌜in⌝ parameters of an enabled instance exist in the corresponding tables and those for the ⌜out⌝ parameters are missing. The out stands in for missing ⌜out⌝ parameters.

Listing 5 applies the pattern of Listing 3 in implementing MERCHANT from Listing 2. It shows a plan for sending *quote*s that is triggered whenever the goal of sending a *quote* is asserted. The plan checks if there is an enabled *quote*; if so, the plan completes it by computing a binding for the ⌜out⌝ parameter price in *quote* and attempts to send it.

Listing 5: The programmer identifies MERCHANT agent's goals and what messages these goals involve sending, and also specifies complete (not shown).

```
@quote-plan[atomic]
+!send_quote(Id, Item, Customer)
  : enabled(quote(Id, Item, out)[receiver(
      Customer)])
 <- !complete(quote(Id, Item, Price)[
      receiver(Customer)]);
    !attempt(quote(Id, Item, Price)[
       receiver(Customer)]).
@goods-plan[atomic]
+!send_goods(Id, Item, Customer)
  : enabled(goods(Id, Item, Outcome, out)[
      receiver(Customer)])
 <- !complete(goods(Id, Item, Outcome,
      Shipped)[receiver(Customer)]);
    !attempt(goods(Id, Item, Outcome,
       Shipped)[receiver(Customer)]).
```

The messages in an attempt are consistent if no two correlated messages (i.e., with the same key) share an ⌜out⌝ parameter. As Section 4 explains, such messages violate the integrity constraint of the BSPL semantics. So messages are emitted via the Jason *.send* action only if they are consistent. For example, in the protocol in Listing 2, in any enactment, *accept* and *reject* are simultaneously enabled. Since they share decision, that means both feature a binding for decision—therefore sending them would violate the protocol. Therefore, any attempt to send both fails. Together, enabled and attempt support compliance with the protocol, thus addressing **Semantic Errors** (Section 3).

The enabled predicates effectively capture the state of the interaction, saving programmers the effort of writing code to track it. Moreover, as we shall see below, by encapsulating state as such, Orpheus helps structure agent code, cleanly separating the internal logic (when to send a *quote* and with what *price*) from computations related to protocol state.

6 Evaluation

We evaluate the claim that Orpheus facilitates a variety of changes to a MAS by reducing code complexity.

6.1 Changes to Protocol

Listing 2 shows a NetBill protocol in which *goods* must happen before *epo* (payment). A more flexible protocol can be obtained by deleting some non-key ⌜in⌝ parameters from some messages, in effect, removing the requirement that those parameters must be known to be able to send those messages. Consider *Flexible-NetBill* in Listing 6, which is obtained by deleting the parameter outcome (adorned ⌜in⌝) from *goods*. This protocol is more flexible because sending *goods* no longer requires receiving *accept* (which produces the binding for outcome) first; receiving *request* is enough.

Let's consider what changes must be made to Listing 5, which shows a MERCHANT's plan for !send_goods, to take advantage of the enhanced flexibility of *Flexible-NetBill*. The enhanced flexibility is actually neatly encapsulated in

Listing 6: Flexible *NetBill*.

```
Flexible-NetBill {
 roles M, C //Merchant, Customer
 parameters out ID key, out item, out done
 private decision, outcome, price, chit,
     shipped, cc
 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, 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]  }
```

Listing 7: Modified plan for !send_goods.

```
@goods-plan[atomic]
+!send_goods(Id, Item, Customer)
 : enabled(goods(Id, Item, out)[receiver(
     Customer)])
 <- !complete(goods(Id, Item, Shipped)[
     receiver(Customer)]);
    !attempt(goods(Id, Item, Shipped)[
        receiver(Customer)]).
```

how the MERCHANT adapter of *Flexible-NetBill* computes enabled *goods* messages. Therefore, the changes to Listing 5 are minimal: we need to delete only the attribute Outcome from the predicate *goods*. Listing 7 gives the resulting code.

By contrast, in the reactive model, changes are extensive. Listing 1 shows a plan for sending *goods* upon *accept*. We must add plans to send *goods* upon receiving *request*, sending *quote*, or receiving *reject*. The state machine encoding is more complex to accommodate new states resulting from the emission of *goods* in those various states.

Flexible-NetBill is not *safe* though: since *goods* is not dependent on outcome, *reject* and *receipt* can be sent concurrently, producing an integrity violation since done is ⌜out⌝ in both. BSPL tooling (Singh and Christie 2021) catches this error and removing done from *reject* fixes it.

The resulting protocol, *Safe-Flexible-NetBill*, can be made even more flexible by deleting shipped from *epo*. Let's refer to this protocol as *Super-Flexible-NetBill*. This modification would mean that *epo* could be sent anytime after *quote* was sent. Again, in the Orpheus approach, the changes to accommodate this enhanced flexibility would be limited to deleting the attribute Shipped from the agent code. Alternatively, one could have implemented a MAS based on *Super-Flexible-NetBill* and switched to *Safe-Flexible-NetBill*, With Orpheus, the changes would have been limited to introducing the corresponding attribute in the agent code. In a nutshell, by abstracting the low-level details of state via enablement, Orpheus addresses **Inflexibility** (Section 3).

6.2 Changes to Agent Decision Making

With Orpheus, changes to an agent's internal logic tend to be highly localized. For example, assuming *Safe-Flexible-NetBill*, the plan !send_goods in Listing 7 could be modified to ship only to *friendly* customers by simply introducing the appropriate query. By contrast, in the reactive model, because there would be several plans for sending goods (as explained in Section 6.1), the query would have to be introduced in each of them.

6.3 Changes to Communication Infrastructure

Orpheus accommodates changes to the communication infrastructure. A fairly drastic change would be switching from an ordered communication service to an unordered service. The motivation to do so could be high performance or settings such as the IoT. Orpheus agents can handle messages in whatever order they arrive: incoming messages are simply added to the agent's local state and may contribute to the enablement or disablement of other messages. However, all this is abstracted away by the adapter. So even without ordering guarantees, Orpheus agents work without needing any change. By contrast, agents implemented using the traditional approach rely on message ordering (e.g., typically FIFO) for correctness. Switching to an unordered communication service would therefore lead to errors.

6.4 Correlating Information

An agent may wish to correlate information from several messages from several agents to decide its action. We adopt a logistics scenario (Sicari, Rizzardi, and Coen-Porisini 2019) to highlight such correlation in handling *purchase orders* (POs) placed (by customers, not modeled). The roles involved are MERCHANT, WRAPPER, LABELER, and PACKER. A PO may have several items, each with a wrapping requirement. MERCHANT sends each item in a PO separately to WRAPPER, who sends wrapped items to PACKER. MERCHANT sends the PO's shipping address to LABELER, who sends the corresponding shipping label to PACKER. If PACKER has received a PO's label, then for every wrapped item it has received for that PO, it may send a notification to the MERCHANT that it is packed. Figure 2 shows how the roles communicate and Listing 8 shows PACKER's part of the protocol in BSPL. Notice the composite key ⟨oID, iID⟩: oID identifies POs and iID identifies items within a PO.

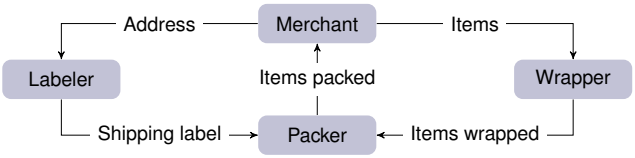

Figure 2: Logistics scenario relative to a PO.

Listing 9 shows a snippet of a PACKER agent. Correlation is handled transparently by the adapter in computing enabled *packed* messages; the programmer has to merely write code to send them. Without Orpheus, a Jason programmer has to tackle each order separately, which increases complexity and exacerbates code maintenance.

Listing 8: Fragment of *Logistics* relevant to PACKER.

```
W -> P: wrapped[in oID key, in iID key,
    in item, out wrapping]
L -> P: labeled[in oID key, in address,
    out label]
P -> M: packed[in oID key, in iID key, in
     wrapping, in label, out status]
```

Listing 9: Plan in Orpheus for sending *packed* messages. The correlation of label with item is automatic.

```
@quote_plan[atomic]
+!send_packed
  : enabled(packed(OID, IID, Item, Wrapping
      , Label, out)[receiver(M)])
  <- !complete(packed(_,_,_,_,_, Status)[
      receiver(M)]);
     !attempt(packed(OID, IID, Item,
         Wrapping, Label, Status)[receiver(
         M)]).
+!complete(packed(_,_,_,_,_, Status)[
    receiver(_)])
  <- Status = true.
```

7 Formalization

Figure 3 gives a transition system semantics for Orpheus. Let $\lambda(m, x, y, \vec{p}, \vec{k}, \vec{i}, \vec{o}, \vec{n})$ be a message schema, where m is the name of the message; x and y are sender and receiver, respectively; $\vec{k}$, $\vec{i}$, $\vec{o}$, and $\vec{n}$ are the sets of key, ⌜in⌝, ⌜out⌝, and ⌜nil⌝ parameters, respectively. Further, $\vec{i} \cup \vec{o} \cup \vec{n} = \vec{p}$; $\vec{i}$, $\vec{o}$, and $\vec{n}$ are pairwise disjoint; and $\vec{k} \subseteq \vec{i} \cup \vec{o}$. A *message instance* of the schema has bindings (values) for each parameter in $\vec{i}$ and $\vec{o}$ and is represented as $m(x, y, \vec{k}, \vec{i}, \vec{o})$ (or $m'(x, y, \vec{k'}, \vec{i'}, \vec{o'})$). An *enabled instance* of the schema has bindings for only the parameters in $\vec{i}$ and is represented by $\mathsf{enabled}(m, x, y, \vec{i}, \vec{o}, \vec{n})$. And, $m[\vec{a}]$ gives the bindings of parameters $\vec{a}$ in instance m. We omit the components of an instance where they are not needed.

Two instances correlate if they share some key parameters with the same bindings. Notice that non-key parameters such as item make sense only in conjunction with key parameters such as ID. We therefore impose the well-formedness condition that if two messages didn't share a key parameter, then they couldn't share any non-key parameter. Let m and m' be instances (message or enabled); $m \circ m'$ (m and m' are correlated) means $m[\vec{k} \cap \vec{k'}] = m'[\vec{k} \cap \vec{k'}]$. And, $m \odot m'$ is $m \circ m'$ where m' is an instance in the agent's local state L.

Rule RECV says that upon reception of a message, if no instance in L conflicts with the received message on its bindings, then it can be added to the local state.

Rule ENABLED defines enabled instances. It says that the bindings for each (of the ⌜in⌝) parameters of an enabled instance must already be present in some correlated instances in the agent's local state, and bindings for none of its ⌜out⌝ or ⌜nil⌝ parameters must already be present in the local state.

Rule ATTEMPT says that given a set of enabled instances, an agent can attempt them if they are completed. As Listing 3 indicates abstractly, the agent must have a plan for the complete goal that would transform a set of enabled instances into a set of message instances by providing bindings for the ⌜out⌝ parameters of each enabled instance in the set.

Rule SEND uses attempt to send messages. If the message instances in attempt are consistent in the sense that no pair of correlated instances disagrees on any parameter binding, then the instances in attempt are all sent.

In this manner, the semantics demonstrates how Orpheus plugs into Jason, enabling benefits from information protocols while respecting Jason's constructs.

$$\textsc{Recv}\ \frac{+m \qquad \mathsf{local}(L) \qquad m \odot m' \rightarrow (p \in \vec{p} \cap \vec{p'} \rightarrow m[p] = m'[p])}{\mathsf{local}(L \cup \{m\})}$$

$$\textsc{Enabled}\ \frac{p \in \vec{i} \rightarrow (m \odot m' \wedge m[p] = m'[p]) \qquad p \in \vec{o} \cup \vec{n} \rightarrow (m \odot m' \rightarrow p \notin \vec{o'})}{\mathsf{enabled}(m, \vec{i}, \vec{o}, \vec{n})}$$

$$\textsc{Attempt}\ \frac{\mathsf{enabled}(m_1), \ldots, \mathsf{enabled}(m_q) \qquad !\mathsf{complete}(\{m_1, \ldots, m_q\})}{!\mathsf{attempt}(\{m_1, \ldots, m_q\})}$$

$$\textsc{Send}\ \frac{!\mathsf{attempt}(\{m_1, \ldots, m_q\}) \qquad \mathsf{local}(L) \qquad m_i \circ m_j \rightarrow (p \in \vec{o_i} \rightarrow p \notin \vec{o_j})}{.\mathsf{send}(m_1, \ldots, m_q) \qquad \mathsf{local}(L \cup \{m_1, \ldots, m_q\})}$$

Figure 3: Operational semantics of Orpheus, as inference rules for an agent's state-transition system. Dotted line: inference within a state; solid line: transition from one state to the next. This semantics highlights the modularity of Orpheus. Rules RECV, ENABLED, and SEND are based on the BSPL semantics rendered into Jason, and implemented in our tool. The programmer is responsible only for complete to apply Rule ATTEMPT.

8 Discussion

A protocol captures stakeholder requirements modularly (Chopra et al. 2014). Whereas a protocol constrains the decisions an agent (playing a role) may make, the agent decides to respect its principal's requirements (Chopra and Singh 2016). Orpheus facilitates specifying agent decision making.

Orpheus addresses a crucial gap in engineering MAS by combining BDI-based and protocol-based abstractions. Orpheus relieves the programmer of reasoning about the nitty gritty of an ongoing interaction and leads to improved code structure by encapsulating and automating the handling of protocol logic. It thus reduces code complexity and facilitates changes to a multiagent system. Orpheus reflects the intuition that an agent sends a message because it advances

its goals based on its beliefs, not because it received a message, as would be the case in object-oriented programming.

Orpheus facilitates reasoning about the protocol state via its *enabled* and *attempt* primitives. As Section 6 shows, Orpheus shines as the underlying protocol becomes more flexible. The number of computations of a flexible information protocol may be exponential in the size of its specification. Without Orpheus, a programmer would have to figure out the possible computations to produce a correct implementation. When multiple computations involving different sets of messages may produce the same information, the programmer would have to produce multiple plans with complex triggers and ensure they produced decisions in accordance with the requirements. The resulting agent would be challenging to maintain, even for small changes to a protocol. Orpheus, by contrast, abstracts away the operations and exposes a higher-level API in which for each message schema, there is one enabled predicate. Consequently, as Section 6 shows, Orpheus elegantly handles changes to protocols.

Coordination in MAS (Omicini and Ossowski 2003) may be *subjective*—how each agent interacts in view of its goals, e.g., computing the price of an item or dealing with a message—or *objective*—how an agent's interactions are governed to satisfy constraints, e.g., on the ordering and occurrence of messages. Artifacts (Weyns, Omicini, and Odell 2007; Ricci et al. 2009), which encode the objective coordination, are not suitable for realizing decentralized MAS (Tolksdorf 2000). For decentralized settings, Omicini and Ossowski (2003) motivate agent platforms to engineer MAS. Orpheus is such an agent platform. The traditional approach outlined in Section 2 mixes the objective and subjective coordination in the agent implementation. Orpheus modularizes them via the adapter. A broader question is how interaction structures affect how the properties of MAS interplay with those of their constituents (Singh 1991).

Basing Orpheus on protocols brings forth additional opportunities for flexibility in engineering MAS. One direction is to determine the conditions for interoperability with others when an agent implements a refined protocol (Baldoni et al. 2009). Another direction is to enable code reuse for agent implementations as the underlying protocol is modified to adapt to different contexts (Chopra and Singh 2006).

8.1 Directions in Addressing Limitations

Orpheus shares challenges with rule-based programming. Errors involving patterns are easy to make and hard to find: an enabled predicate must exactly match the message schema. Any difference in parameter ordering would bind parameters to the wrong values, and missing or extra parameters would silently prevent the plan from matching at all. These problems can be alleviated by tools that generate agent stubs and warn about schema mismatches. Ideally, the programming model should make it clear—via typing—what parameters of an enabled message a plan needs to bind.

8.2 Directions in Addressing Opportunities

MAS software methodologies, e.g., Prometheus (Padgham and Winikoff 2005), typically support deriving informal UML interaction diagrams from requirements. Some works address the anatomy of information protocols (Singh 2014). A crucial direction is methodologies for obtaining information protocols and agents from requirements. The reduction in complexity due to Orpheus can help manage the complexity of testing BDI programs (Winikoff and Cranefield 2014).

The Orpheus adapter uses KQML's tell primitive for communication, though only for transport. KQML and the FIPA ACL (FIPA 2002) have well-known limitations that make them unsuited to multiagent systems (Singh 1998). An interesting direction would be to make information protocols native to cognitive agent languages, obviating their use of KQML or ACL and improving MAS quality by systematically modeling interactions (Chopra and Christie V 2023).

JaCaMo (Boissier et al. 2019) is a framework for programming MAS that combines Jason, CArtAgO (Ricci et al. 2009), and MOISE (Hübner, Sichman, and Boissier 2007). Orpheus will enable improvements through the JaCaMo value chain, including by taking fuller advantage of CArtAgO and MOISE for improved programming and support for commitment-based reasoning (Baldoni et al. 2018).

Kiss, Madden, and Logan (2010) propose enhancements to agent programming to tackle plan failures atomically. These are interesting improvements but not directly related to our present research questions. Yet, extending Orpheus to support protocol-based fault handling, as shown by Christie, Chopra, and Singh (2022), would be valuable.

Imperative cognitive languages, e.g., ASTRA (Dhaon and Collier 2014) and SARL (Galland, Rodriguez, and Gaud 2020), can also support protocols. Though our Kiko programming model and adapter (Christie, Singh, and Chopra 2023) doesn't tackle cognitive programming, it can guide the development of Orpheus-like adapters for imperative cognitive languages since it is built on Python. Comparing Orpheus implementations with Kiko in terms of code quality and performance could yield insights into the tradeoffs between various agent programming paradigms for protocols.

Information protocols fit well with declarative business process modeling based on data (Montali, Calvanese, and Giacomo 2014; Lichtenstein et al. 2024). Orpheus can help relate data-driven and BDI approaches, combining flexible decision making with routine computations.

Günay, Winikoff, and Yolum (2015) and Meneguzzi et al. (2018) generate commitments based on goals. Singh and Chopra (2020) generate protocols from commitments. What we need is a comprehensive model of goals, commitments, and protocols. Our recent work, Azorus, recasts commitment semantics into Jason to realize a programming model in the style of Orpheus (Chopra et al. 2025). One direction is to incorporate joint formal reasoning about commitments and protocols. Another is to support protocols based *not* on messages, but on communicative actions, as in Langshaw (Singh, Christie, and Chopra 2024). Langshaw improves the compactness and flexibility of protocols and is well-suited to modeling commitments. How can we develop effective cognitive agents for such languages?

9 Reproducibility

The entire codebase (including tooling) and all examples are available at the Code URL shown below the abstract.

Acknowledgments

Thanks to the reviewers for their helpful comments. MPS thanks the NSF (grant IIS-1908374) for partial support.

References

Baldoni, M.; Baroglio, C.; Capuzzimati, F.; and Micalizio, R. 2018. Commitment-based Agent Interaction in JaCaMo+. *Fundamenta Informaticae*, 159(1-2): 1–33.

Baldoni, M.; Baroglio, C.; Chopra, A. K.; Desai, N.; Patti, V.; and Singh, M. P. 2009. Choice, Interoperability, and Conformance in Interaction Protocols and Service Choreographies. In *Proceedings of the 8th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, 843–850. Budapest: IFAAMAS.

Baldoni, M.; Baroglio, C.; Martelli, A.; and Patti, V. 2006. A Priori Conformance Verification for Guaranteeing Interoperability in Open Environments. In *Proceedings of the 4th International Conference on Service-Oriented Computing (ICSOC)*, LNCS 4294, 339–351. Chicago: Springer.

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.; and Ricci, A. 2019. Dimensions in Programming Multi-Agent Systems. *Knowledge Engineering Review (KER)*, 34: e2.

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.

Bracciali, A.; Endriss, U.; Demetriou, N.; Kakas, A. C.; Lu, W.; and Stathis, K. 2006. Crafting the Mind of PROSOCS Agents. *Applied Artificial Intelligence*, 20(2–4): 105–131.

Castelfranchi, C. 1998. Modelling Social Action for AI Agents. *Artificial Intelligence*, 103(1–2): 157–182.

Chopra, A. K.; Baldoni, M.; Christie V, S. H.; and Singh, M. P. 2025. Azorus: Commitments over Protocols for BDI Agents. In *Proceedings of the 24th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. Detroit: IFAAMAS.

Chopra, A. K.; and Christie V, S. H. 2023. Communication Meaning: Foundations and Directions for Systems Research. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, 1786–1791. London: ACM.

Chopra, A. K.; Christie V, S. H.; and Singh, M. P. 2020. An Evaluation of Communication Protocol Languages for Engineering Multiagent Systems. *Journal of Artificial Intelligence Research (JAIR)*, 69: 1351–1393.

Chopra, A. K.; Dalpiaz, F.; Aydemir, F. B.; Giorgini, P.; Mylopoulos, J.; and Singh, M. P. 2014. Protos: Foundations for Engineering Innovative Sociotechnical Systems. In *Proceedings of the 22nd IEEE International Requirements Engineering Conference (RE)*, 53–62. Karlskrona, Sweden: IEEE Computer Society.

Chopra, A. K.; and Singh, M. P. 2006. Contextualizing Commitment Protocols. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems*, 1345–1352. Hakodate, Japan: ACM Press.

Chopra, A. K.; and Singh, M. P. 2016. From Social Machines to Social Protocols: Software Engineering Foundations for Sociotechnical Systems. In *Proceedings of the 25th International World Wide Web Conference*, 903–914. Montréal: ACM.

Christie V, S. H.; Chopra, A. K.; and Singh, M. P. 2022. 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.

Christie V, S. H.; Singh, M. P.; and Chopra, A. K. 2023. Kiko: Programming Agents to Enact Interaction Protocols. In *Proceedings of the 22nd International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, 1154–1163. London: IFAAMAS.

Desai, N.; Mallya, A. U.; Chopra, A. K.; and Singh, M. P. 2005. Interaction Protocols as Design Abstractions for Business Processes. *IEEE Transactions on Software Engineering*, 31(12): 1015–1027.

Dhaon, A.; and Collier, R. W. 2014. Multiple Inheritance in AgentSpeak(L)-Style Programming Languages. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control, (AGERE!)*, 109–120. Portland, Oregon: ACM.

Finin, T.; Fritzson, R.; McKay, D.; and McEntire, R. 1994. KQML as an Agent Communication Language. In *Proceedings of the 3rd International Conference on Information and Knowledge Management*, 456–463. Gaithersburg, Maryland: ACM Press.

FIPA. 2002. FIPA Agent Communication Language Specifications. http://www.fipa.org/repository/aclspecs.html. FIPA: The Foundation for Intelligent Physical Agents. Accessed 2025-01-20.

FIPA. 2003. FIPA Interaction Protocol Specifications. http://www.fipa.org/repository/ips.html. FIPA: The Foundation for Intelligent Physical Agents. Accessed 2024-11-24.

Galland, S.; Rodriguez, S.; and Gaud, N. 2020. Run-time Environment for the SARL Agent-Programming Language: The Example of the Janus platform. *Future Generation Computer Systems*, 107: 1105–1115.

Günay, A.; Winikoff, M.; and Yolum, P. 2015. Dynamically Generated Commitment Protocols in Open Systems. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 29(2): 192–229.

Hohpe, G.; and Woolf, B. 2004. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Signature Series. Boston: Addison-Wesley.

Hübner, J. F.; Sichman, J. S.; and Boissier, O. 2007. Developing Organised Multiagent Systems using the MOISE$^+$ Model: Programming Issues at the System and Agent Levels. *International Journal of Agent-Oriented Software Engineering*, 1(3/4): 370–395.

Kiss, D.; Madden, N.; and Logan, B. 2010. Atomic Intentions in Jason+. In *Proceedings of the 8th International Workshop on Programming Multiagent Systems (ProMAS)*, LNCS 6599, 79–95. Toronto: Springer.

Lichtenstein, T.; Chopra, A. K.; Singh, M. P.; and Weske, M. 2024. From Visual Choreographies to Flexible Information Protocols. In *Proceedings of the 22nd International Conference on Service-Oriented Computing (ICSOC)*, number 15404 in Lecture Notes in Computer Science, 354–369. Tunis, Tunisia: Springer.

Meneguzzi, F.; Magnaguagno, M. C.; Singh, M. P.; Telang, P. R.; and Yorke-Smith, N. 2018. GoCo: Planning Expressive Commitment Protocols. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 32(4): 459–502.

Montali, M.; Calvanese, D.; and Giacomo, G. D. 2014. Verification of data-aware commitment-based multiagent system. In *Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems*, 157–164. Paris: IFAAMAS.

Odell, J.; Parunak, H. V. D.; and Bauer, B. 2001. Representing Agent Interaction Protocols in UML. In *Proceedings of the 1st International Workshop on Agent-Oriented Software Engineering (AOSE 2000)*, LNCS 1957, 121–140. Toronto: Springer.

Omicini, A.; and Ossowski, S. 2003. Objective versus Subjective Coordination in the Engineering of Agent Systems. In Klusch, M.; Bergamaschi, S.; Edwards, P.; and Petta, P., eds., *Intelligent Information Agents - The AgentLink Perspective*, LNCS 2586, 179–202. Springer.

Padgham, L.; and Winikoff, M. 2005. Prometheus: A Practical Agent-Oriented Methodology. In Henderson-Sellers, B.; and Giorgini, P., eds., *Agent-Oriented Methodologies*, chapter 5, 107–135. Hershey, Pennsylvania: Idea Group.

Ricci, A.; Piunti, M.; Viroli, M.; and Omicini, A. 2009. Environment Programming in CArtAgO. In Bordini, R. H.; Dastani, M.; Dix, J.; and Seghrouchni, A. E. F., eds., *Multi-Agent Programming, Languages, Tools and Applications*, chapter 8, 259–288. Dordrecht, Netherlands: Springer.

Rooney, C.; Collier, R. W.; and O'Hare, G. M. P. 2004. VIPER: A VIsual Protocol EditoR. In *Proceedings of the 6th International Conference on Coordination Models and Languages COORDINATION*, LNCS 2949, 279–293. Pisa: Springer.

Sicari, S.; Rizzardi, A.; and Coen-Porisini, A. 2019. Smart Transport And Logistics: A Node-RED Implementation. *Internet Technology Letters*, 2(2): e88.

Sichman, J. S.; Conte, R.; Demazeau, Y.; and Castelfranchi, C. 1994. A Social Reasoning Mechanism Based on Dependence Networks. In *Proceedings of the 11th European Conference on Artificial Intelligence*, 188–192. Amsterdam: John Wiley and Sons.

Singh, M. P. 1991. Group Ability and Structure. In Demazeau, Y.; and Müller, J.-P., eds., *Decentralized Artificial Intelligence, Volume 2*, 127–145. Amsterdam: Elsevier/North-Holland. ISBN 9780444597380. Revised proceedings of the *2nd European Workshop on Modeling Autonomous Agents in a Multi Agent World (MAAMAW)*, St. Quentin en Yvelines, France, August 1990.

Singh, M. P. 1998. Agent Communication Languages: Rethinking the Principles. *IEEE Computer*, 31(12): 40–47.

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)*, 491–498. Taipei: IFAAMAS.

Singh, M. P. 2014. Bliss: Specifying Declarative Service Protocols. In *Proceedings of the 11th IEEE International Conference on Services Computing (SCC)*, 235–242. Anchorage, Alaska: IEEE Computer Society.

Singh, M. P.; and Chopra, A. K. 2017. The Internet of Things and Multiagent Systems: Decentralized Intelligence in Distributed Computing. In *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 1738–1747. Atlanta: IEEE. Blue Sky Thinking Track.

Singh, M. P.; and Chopra, A. K. 2020. Clouseau: Generating Communication Protocols from Commitments. In *Proceedings of the 34th Conference on Artificial Intelligence (AAAI)*, 7244–7252. New York: AAAI Press.

Singh, M. P.; and Christie V, S. H. 2021. Tango: Declarative Semantics for Multiagent Communication Protocols. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI)*, 391–397. Online: IJCAI.

Singh, M. P.; Christie V, S. H.; and Chopra, A. K. 2024. Langshaw: Declarative Interaction Protocols Based on Sayso and Conflict. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI)*, 202–210. Jeju, Korea: IJCAI.

Sirbu, M. A. 1997. Credits and Debits on the Internet. *IEEE Spectrum*, 34(2): 23–29.

Spanoudakis, N. I.; and Moraitis, P. 2022. The ASEME Methodology. *International Journal of Agent-Oriented Software Engineering*, 7(2): 79–107.

Tolksdorf, R. 2000. Models of Coordination. In Omicini, A.; Tolksdorf, R.; and Zambonelli, F., eds., *Engineering Societies in the Agent World, First International Workshop, ESAW 2000, Berlin, Germany, August 21, 2000, Revised Papers*, LNCS 1972, 78–92. Springer.

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.

Weyns, D.; Omicini, A.; and Odell, J. 2007. Environment as a First Class Abstraction in Multiagent Systems. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 14(1): 5–30.

Winikoff, M. 2012. Challenges and Directions for Engineering Multi-Agent Systems. *CoRR*, abs/1209.1428.

Winikoff, M.; and Cranefield, S. 2014. On the Testability of BDI Agent Systems. *Journal of Artificial Intelligence Research (JAIR)*, 51: 71–131.