Orpheus: Programming Protocol-Based BDI Agents

Demonstration Track

Matteo Baldoni
University of Turin
Turin, Italy
matteo.baldoni@unito.it

Samuel H. Christie V
North Carolina State University
Raleigh, NC, USA
schrist@ncsu.edu

Munindar P. Singh
North Carolina State University
Raleigh, NC, USA
mpsingh@ncsu.edu

Amit K. Chopra
Lancaster University
Lancaster, United Kingdom
amit.chopra@lancaster.ac.uk

ABSTRACT

We demonstrate *Orpheus*, a novel programming model for engineering BDI agents that communicate on the basis of protocols. In Orpheus, protocols are specified in BSPL and agents are implemented in Jason. Given a protocol, Orpheus tooling generates a Jason adapter that exposes a simple interface for sending messages based on protocol state. Orpheus shines in the implementation of flexible, loosely-coupled agents, long a challenge for BDI-based agent programming approaches.

Demonstration video: https://di.unito.it/orpheusvid

CCS CONCEPTS

• **Computing methodologies → Multi-agent systems**.

KEYWORDS

Information Protocols; BDI; Agent Programming

ACM Reference Format:

Matteo Baldoni, Samuel H. Christie V, Munindar P. Singh, and Amit K. Chopra. 2025. Orpheus: Programming Protocol-Based BDI Agents : Demonstration Track. In *Proc. of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), Detroit, Michigan, USA, May 19 – 23, 2025*, IFAAMAS, 3 pages.

1 INTRODUCTION

Michael Winikoff [14] highlighted two shortcomings in agent-oriented programming languages (AOPLs). One, despite the importance of modeling interactions in multiagent systems (MAS), AOPLs supported little more than primitives for sending and receiving messages. He saw such primitives as transferring control between agents in an unstructured manner and drew an unflattering analogy with *gotos*. Two, *interaction protocols*, typically expressed in notations such as AUML [8], were *message-centric* and overconstrained the interactions between agents. With the aim of supporting robustness and flexibility in interactions, Winikoff advocated higher-level communication abstractions. More than a decade later, AOPLs have hardly changed.

Proc. of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), Y. Vorobeychik, S. Das, A. Nowé (eds.), May 19 – 23, 2025, Detroit, Michigan, USA. © 2025 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org).

We recently developed Orpheus, a programming model for multiagent systems [1] that shows how to overcome the limitations pointed out by Winikoff. Orpheus adopts information protocols expressed in the Blindingly Simple Protocol Language (BSPL) [9], which boasts formal semantics and verification tools [10, 11] and programming models [6, 7]. BSPL is a declarative and asynchronous model for flexible interactions between agents [3].

Orpheus overcomes shortcomings of message-centric interaction protocols, such as *incompatibilities* between agents due to the message schemas being blended into business logic; *semantic errors* due to a lack of a formal model; and *inflexibility* due to the programmer having to maintain the protocol state via a state machine. Orpheus is grounded on Jason [12], to fully exploit the *agents' cognitive autonomy* through the agent's goals, beliefs, and intentions. Moreover, it fully exploits the *agent's social autonomy* through the adoption of information protocols. Orpheus tooling generates plan and query libraries that facilitate implementing agents by tracking state and for computing *enabled* (legal to emit) messages at runtime.

2 TOOLING

In Orpheus, agent logic is organized according to what messages are *enabled* to be sent. A 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 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. An message instance is enabled when its ⌜in⌝ parameters are bound (their bindings are known); and its ⌜out⌝ parameters are unbound (they are not known). A role *knows* bindings for some parameters if it has sent or received messages with bindings for those parameters.

An agent uses beliefs to encode its view of the *protocol state*, called *local state* in Orpheus. An *incoming message* is added to the local state if it is consistent with the state, i.e., if no other binding is already known for any its parameters (relative to the key). For *outgoing messages* an attempt to send is successful if the *completed messages* are mutually consistent in their bindings; the sent messages are added to the local state. Moreover, an agent has a set of role-specific queries and plans that are automatically generated by the Orpheus Tool and they constitute the *role adapter* in Orpheus. The queries are used for computing *enabled messages*. The plans validate messages before emission and upon reception.

Role adapters encapsulate all uses of send and receive primitives and in that constitute a higher-level communication abstraction. An agent programmer uses the adapter to define the internal logic of the agent, as Winikoff wished for AOPLs. Listing 1 illustrates BSPL via our running example.

Listing 1: Initial *Contract Net* Protocol.

```
ContractNet {
  role C, P
  parameter out IDt key, out task, out outcome
  private pdecision, offer, outcome
  C -> P : cfp [out IDp key, out IDt key, out task]
  P -> C : propose [in IDp key, in IDt key, in task,
       out offer, out pdecision]
  P -> C : refuse [in IDp key, in IDt key, in task, out
        outcome, out pdecision]
  C -> P : accept_prop [in IDp key, in IDt key, in
       offer, out accept, out x]
  C -> P : reject_prop [in IDp key, in IDt key, in
       offer, out outcome, out x]
  P -> C : done [in IDp key, in IDt key, in accept, out
        outcome]
  P -> C : failure [in IDp key, in IDt key, in accept,
       out outcome]  }
```

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. So, for instance, there will not be two *cfp* instances with the same value for IDp but with different bindings for task. Moreover, there will not be an instance of *accept_prop* and one of *reject_prop* for the same pair IDp and IDt, because they should both bind x.

BSPL thus captures *causality* and *integrity* through information. Instead of reacting to message receptions, to achieve its goal, the agent queries for enabled instances of the messages it may send. These enabled instances are incomplete and the agent must provide bindings for their ⌜out⌝ parameters so they can be sent. It may choose to complete some of the enabled messages by producing those bindings and attempting to send them in one shot. The sent messages are added to the local state. A message may be received at any time, obviating the need for ordered-delivery infrastructure.

The Orpheus Tool is implemented in Java. It takes as input a BSPL protocol and produces a role adapter for each role in the protocol, as Figure 1 shows. The video shows how agents are implemented.

3 IMPLEMENTING AGENTS

Using Orpheus, programmers of Jason agents focus *not* on reactions to incoming messages, but the enabled messages the agent may send, abstracting out reasoning about the protocol into automatic generated code. To achieve some goal, the agent:

(1) *queries* if there are enabled instances corresponding to the message it wants to send (Listing 2, lines 2 and 7;
(2) *completes* them by producing bindings for their ⌜out⌝ parameters (Listing 2, lines 4 and 11; and
(3) *attempts* to send them in one shot (Listing 2, lines 5 and 12.

For instance, in Listing 2, first, the agent checks if *cfp* is enabled, then, it completes the message by binding the out parameters, finally, it attempts to send the message. For assigning the contract, the agent checks for all the enabled *accept_prop* messages. It looks

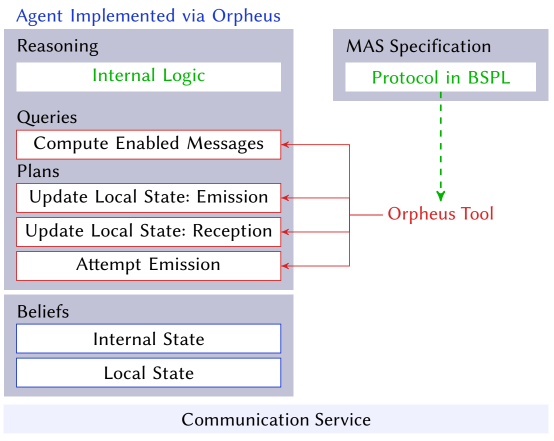

Figure 1: The Orpheus Tool.

for the best one, then, by *announce_result*, it selects the accept to be sent accordingly and sends rejects of all other proposals.

Listing 2: An excerpt of the contractor in the CNP.

```
1  +!cfp(Idt, Task)
2   : enabled(cfp(out, out, out)[receiver(out)])
3   <- for ( participant(P) ) {
4    !complete(cfp(Idp, Idt, Task)[receiver(P)]);
5    !attempt(cfp(Idp, Idt, Task)[receiver(P)]);  }.
6  +!contract(Idt)
7   <- .findall(offer(Offer, Idp, P), enabled(accept_prop(
        Idp, Idt, Offer, out, out)[receiver(P)]), L);
8    L \== []; .min(L, offer(WOffer, WIdp, WAg));
9    !announce_result(Idt, L, WIdp, WAg).
10 +!announce_result(Idt, [offer(Offer, WIdp, WAg) | T],
      WIdp, WAg)
11  <- !complete(accept_prop(WIdp, Idt, Offer, Accept, X)[
        receiver(WAg)]);
12   !attempt(accept_prop(WIdp, Idt, Offer, Accept, X)[
         receiver(WAg)]);
13   !announce_result(Idt, T, WIdp, WAg).
```

4 CONCLUSIONS

We demonstrate Orpheus, whose value proposition to engineering MAS lies in enabling the implementation of loosely-coupled, flexible agents via high-level communication abstractions. Orpheus simplifies changes to agent decision making and to protocols. By using protocols, it supports the implementation of MAS on fully asynchronous communication services, multiparty (more than two) interactions, and multiple concurrent instances of a protocol. In [2], we discuss Azorus, a programming model for multiagent systems that combines cognitive abstractions with BSPL. In particular, we capture the meaning of interaction via a specification of commitments [3–5, 13, 15] and the operational constraints on interaction via BSPL [6, 7, 9–11].

5 RESOURCES

Source code is available at https://di.unito.it/orpheus.

ACKNOWLEDGMENTS

Thanks to the NSF (grant IIS-1908374).

REFERENCES

[1] Matteo Baldoni, Samuel H. Christie V, Munindar P. Singh, and Amit K. Chopra. 2025. Orpheus: Engineering Multiagent Systems via Communicating Agents. In *Proceedings of the 39th AAAI Conference on Artificial Intelligence (AAAI)*. AAAI, Philadelphia, 1–9.

[2] Amit K. Chopra, Matteo Baldoni, Samuel H. Christie V, and Munindar P. Singh. 2025. Azorus: Commitments over Protocols for BDI Agents. In *Proceedings of the 24th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Detroit.

[3] Amit K. Chopra, Samuel H. Christie V, and Munindar P. Singh. 2020. An Evaluation of Communication Protocol Languages for Engineering Multiagent Systems. *Journal of Artificial Intelligence Research (JAIR)* 69 (Dec. 2020), 1351–1393. https://doi.org/10.1613/jair.1.12212

[4] Amit K. Chopra and Munindar P. Singh. 2015. Cupid: Commitments in Relational Algebra. In *Proceedings of the 29th Conference on Artificial Intelligence (AAAI)*. AAAI Press, Austin, Texas, 2052–2059. https://doi.org/10.1609/aaai.v29i1.9443

[5] Amit K. Chopra and Munindar P. Singh. 2016. Custard: Computing Norm States over Information Stores. In *Proceedings of the 15th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Singapore, 1096–1105. https://doi.org/10.5555/2936924.2937085

[6] Samuel H. Christie V, Amit K. Chopra, and Munindar P. Singh. 2022. Mandrake: Multiagent Systems as a Basis for Programming Fault-Tolerant Decentralized Applications. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 36, 1, Article 16 (April 2022), 30 pages. https://doi.org/10.1007/s10458-021-09540-8

[7] Samuel H. Christie V, Munindar P. Singh, and Amit K. Chopra. 2023. Kiko: Programming Agents to Enact Interaction Protocols. In *Proceedings of the 22nd International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, London, 1154–1163. https://doi.org/10.5555/3545946.3598758

[8] Marc-Philippe Huget and James Odell. 2004. 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)*. Springer, New York, 16–30. https://doi.org/10.1007/978-3-540-30578-1_2

[9] Munindar P. Singh. 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)*. IFAAMAS, Taipei, 491–498. https://doi.org/10.5555/2031678.2031687

[10] Munindar P. Singh. 2012. Semantics and Verification of Information-Based Protocols. In *Proceedings of the 11th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Valencia, Spain, 1149–1156. https://doi.org/10.5555/2343776.2343861

[11] Munindar P. Singh and Samuel H. Christie V. 2021. Tango: Declarative Semantics for Multiagent Communication Protocols. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI)*. IJCAI, Online, 391–397. https://doi.org/10.24963/ijcai.2021/55

[12] Renata Vieira, Álvaro F. Moreira, Michael J. Wooldridge, and Rafael H. Bordini. 2007. On the Formal Semantics of Speech-Act Based Communication in an Agent-Oriented Programming Language. *Journal of Artificial Intelligence Research (JAIR)* 29 (June 2007), 221–267. https://doi.org/10.1613/jair.2221

[13] Michael Winikoff. 2007. Implementing Commitment-based Interactions. In *Proceedings of the 6th International Conference on Autonomous Agents and Multiagent Systems*. 1–8.

[14] Michael Winikoff. 2012. Challenges and Directions for Engineering Multi-Agent Systems. *CoRR* abs/1209.1428 (2012), 12 pages.

[15] Pınar Yolum and Munindar P. Singh. 2002. Flexible Protocol Specification and Execution: Applying Event Calculus Planning using Commitments. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. ACM Press, Bologna, 527–534. https://doi.org/10.1145/544862.544867