

Engineering Information Protocols into Multi-Agent Platforms: a JADE-based Realization

Matteo Baldoni¹, Cristina Baroglio¹, Elisa Marengo¹, Roberto Micalizio¹ and Stefano Tedeschi^{2,*}

¹Università degli Studi di Torino, Dipartimento di Informatica, Torino, Italy

²Università della Valle d'Aosta – Université de la Vallée d'Aoste, Dipartimento di Scienze Economiche e Politiche, Aosta, Italy

Abstract

Widely established agent platforms such as JADE offer a solid infrastructure for developing distributed multi-agent systems. However, their reliance on message-oriented, sequence-based, communication often forces developers to embed interaction logic within agent behaviors. This design choice reduces modularity and hinders incremental development. In this paper, we demonstrate a way to break such a strong reliance on message-oriented communication by integrating an information-centric protocol, such as BSPL, into JADE systems via dedicated adapter components. These adapters realize BSPL's information-centric semantics within JADE's execution model, allowing agents to seamlessly operate in compliance with protocol constraints. In the proposed solution, a relational database is used to manage protocol knowledge, capturing the information produced during protocol enactment and supporting efficient reasoning over interaction states. The resulting architecture combines the declarative advantages of BSPL with the practical benefits of JADE, offering a modular foundation for engineering flexible multi-agent interactions.

Keywords

JADE, BSPL, Information protocols, Engineering MAS.

1. Introduction

Interaction lies at the core of multi-agent systems (MAS), however, many well-known agent-oriented programming languages (AOPLs) and platforms [1, 2, 3, 4], such as JADE [5, 6] and SPADE [7, 8], still approach interaction primarily through low-level message passing. While this paradigm provides flexibility, it places the burden of coordinating interactions on the developer, who must explicitly manage protocol progression, message ordering, and consistency across agents. As systems evolve, this tight coupling between interaction logic and agent behavior becomes a major obstacle to maintainability and extensibility. A key limitation of message-oriented approaches is that interaction protocols are typically encoded procedurally within agent behaviors. This results in implementations where coordination is intertwined with application logic, making even small changes to interaction requirements costly and error-prone. In particular, handling asynchronous communication, alternative interaction paths, and multi-party coordination often leads to increasingly complex control structures that are difficult to adapt over time. From a software engineering perspective, the limitations of traditional AOPLs also hinder the adoption of iterative and incremental development processes [9, 10] in MAS: indeed, effective iteration calls for programming abstractions that localize change and prevent cascading effects.

Recent research has explored *information-centric* approaches to interaction, where protocols are defined in terms of data dependencies rather than message sequences [11]. Among these, the Blindingly Simple Protocol Language (BSPL) [12] provides a declarative framework in which interaction correctness emerges from constraints on information production and usage. BSPL is equipped with formal semantics and verification techniques [13, 14], as well as programming models and tooling [15, 16]. Building

WOA 2026: 27th Workshop From Objects to Agents, June 15-17, Salerno, Italy

*Corresponding author.

✉ matteo.baldoni@unito.it (M. Baldoni); cristina.baroglio@unito.it (C. Baroglio); elisa.marengo@unito.it (E. Marengo); roberto.micalizio@unito.it (R. Micalizio); s.tedeschi@univda.it (S. Tedeschi)

ORCID 0000-0002-9294-0408 (M. Baldoni); 0000-0002-2070-0616 (C. Baroglio); 0000-0003-1879-2088 (E. Marengo); 0000-0001-9336-0651 (R. Micalizio); 0000-0002-9861-390X (S. Tedeschi)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

on this perspective, the Orpheus model demonstrated how protocol reasoning can be decoupled from agent code in the well-known Jason agent programming language, enabling more modular and flexible implementations [17, 18, 19]. Despite these advances, the practical adoption of information protocols in mainstream agent platforms remains limited.

A significant challenge concerns the engineering complexity of *maintaining and handling the information state* required by such protocols: each agent must keep track of the information it has produced or received along all of the protocol instances, ensure consistency of bindings in spite of asynchrony and concurrency, and reason about which interactions are enabled at any given time. As an example, JADE—one of the best known frameworks for MAS—continues to rely on FIPA-style messaging and behavior-based protocol implementations, making the integration of higher-level interaction abstractions, without disrupting its programming model, a real challenge.

We address this challenge by introducing BSPL-compliant adapters for JADE-based MAS, following the architectural intuition of the Orpheus programming model while adapting it to JADE’s execution model. Our approach brings the benefits of declarative interaction into JADE by encapsulating protocol semantics within dedicated adapter components. These adapters mediate communication, enforce protocol constraints, and compute enabled interactions, allowing agents to focus solely on their domain logic. A central aspect of our design is the transparent management of protocol-related knowledge. Rather than embedding this knowledge within the internal state of agents, we adopt a relational database to store and query the information generated during protocol enactment. This choice enables efficient reasoning over interaction states. By combining BSPL’s information-driven model with JADE’s mature infrastructure, our approach provides a practical pathway towards more modular and evolvable multi-agent systems.

The aim of this paper is primarily architectural: we investigate how BSPL-style information-centric interaction can be realized within JADE while preserving JADE’s programming model and runtime infrastructure. The EBusiness protocol (see below) is used as a running example to illustrate the feasibility of the approach and the separation of responsibilities between agent behavior, protocol adapters, and protocol-state management. The contribution of the paper is threefold. First, it shows how information-centric BSPL protocols can be integrated into JADE without replacing JADE’s execution model, agent lifecycle, behavior abstraction, or ACL-based communication infrastructure. Second, it introduces a protocol-aware adapter architecture that separates protocol-state management from application-level agent behavior. Third, it proposes a relational representation of BSPL information states, where message enablement and consistency checking are implemented through database queries and transactional updates.

2. Background

2.1. BSPL

A BSPL protocol [12] (see Listing 1 for 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 $\lceil \text{in} \rceil$ *must be known* to the sender at the time of message emission, whereas parameters marked as $\lceil \text{out} \rceil$ *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 $\lceil \text{in} \rceil$ parameters are bound and known to the agent, and (ii) all its $\lceil \text{out} \rceil$ 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 $\lceil \text{in} \rceil$ 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 same 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 messages to be sent (e.g., by querying for enabled messages), rather than react to message arrivals. This will be a key aspect in facing the benchmark.

Listing 1: *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]
9 }

```

Concerning the example in Listing 1, the protocol involves two roles, Buyer and Seller, whose enactment is identified by parameter key, and five message schemas can possibly be instantiated during the interaction: offer, accept, refuse, payment, and shipment. The protocol execution ends when all protocol parameters are bound (key, item, price, status). Message 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 as and when issuing an offer. Its execution, however, binds its $\lceil \text{out} \rceil$ parameters. Instead, accept and refuse are in mutual exclusion: this is captured by both having decision as an $\lceil \text{out} \rceil$ 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.

2.2. JADE

JADE [5, 6] is a multi-agent framework designed to facilitate the development of MAS compliant with the FIPA standards [20]. It is implemented in Java and provides a middleware that simplifies the creation, deployment, and management of autonomous agents that can communicate and cooperate within distributed environments. JADE abstracts many of the complexities associated with agent-based programming, such as message passing, lifecycle management, and network communication. Communication is based on a FIPA-compliant ACL, which follows standardized performatives (e.g., request, inform, propose) to ensure interoperability between agents. The platform's architecture is structured around *containers*, which host agents and provide the runtime environment for their execution. A typical JADE MAS consists of a main container – responsible for managing the platform, including agent registration and directory services – and one or more additional containers that can be distributed across different machines. JADE also includes built-in services such as the *directory facilitator*, which acts as a “yellow pages” service where agents can publish and discover services and other agents, as well as the *agent management system*, which supervises agent lifecycle operations.

Each agent runs as an independent thread within a container and has a unique identifier that allows addressing it within the platform. Agents are designed to operate continuously: once instantiated, they execute a lifecycle that includes initialization, active execution, and termination. The internal logic of a JADE agent is structured around *behaviors*. Rather than implementing a single monolithic

control loop, JADE promotes a modular approach where tasks are decomposed into reusable behavior objects. A behavior represents a specific unit of execution, such as handling a request, performing a computation, or waiting for a condition. The JADE scheduler manages these behaviors cooperatively within each agent. Communication between agents is handled through an asynchronous message queue mechanism. Each agent maintains a private queue where incoming ACL messages are stored. Messages are exchanged via the JADE runtime by methods `send()` and `receive()`.

3. Building BSPL-compliant Adapters for JADE

We now present a general architecture for building BSPL role adapters for JADE agents. By integrating BSPL adapters directly into the JADE agents lifecycle, the proposed architecture replaces the traditional message queue mechanism with a protocol-aware counterpart that intercepts incoming messages and enforces BSPL consistency constraints transparently. For illustration purposes, we keep using the EBusiness protocol described in Section 2.1 as a running example¹. The example is intended to demonstrate the feasibility of the proposed integration, and illustrate the implementation pattern; it is not meant to provide a complete performance evaluation of the approach.

Figure 1 shows the Java class diagram that summarizes the main elements of the architecture, organized into two main layers, respectively, the green and yellow classes. The core adapter infrastructure, in green, defines the abstract classes and interfaces that every BSPL-compliant adapter must implement, together with the components needed for JADE integration. The protocol specific layer, in yellow, includes concrete classes specific for a particular BSPL protocol instantiation. These two layers are described in detail in the following sections.

3.1. Core Adapter Infrastructure

The central concept in our model is the *adapter*, a protocol-specific object that sits between the agent's application logic and the JADE communication layer. The adapter maintains, on behalf of one role player, i.e., an agent enacting a protocol, its complete local state, that is, the information about all ongoing interactions: which parameter values have been produced so far and for which keys. Given this state, the adapter can at any moment compute the set of messages that the given role player is currently permitted to send, i.e., the *enabled messages*. The adapter, therefore, acts as a sort of "protocol oracle": instead of having to explicitly keep track of the current step of each ongoing interaction, one agent simply asks its role adapter "what messages may I send right now?".

The agent can, then, select one message from the answer, complete it with its application-level content (i.e., binding the "out" parameters), and hand it back to the adapter for transmission. The adapter takes care of recording the new bindings, updating the information state, and delivering the message to the JADE transport layer.

On the receiving side, the receiver's adapter is notified of every incoming message. It transparently extracts the parameter bindings carried by the message and incorporates them into the receiver's local state, so that when the receiver queries the adapter about the enabled messages, the result already reflects the latest communication from all peers.

It is worth noting that the agent is *autonomous* in the process of *message selection*: the adapter does not influence or constrain in any way the possible choices of the agent, it only shows all the messages that can be sent at that point of the interaction.

A key design choice in the proposed architecture is the use of a *relational representation* of the protocol state. In the current prototype, this representation is implemented through an H2 in-process database, that is local to each role adapter². H2 is used as a lightweight backend for demonstrating the feasibility of the approach, and for making the mapping between BSPL parameters, enablement

¹A preliminary implementation of the proposed architecture, together with the EBusiness example, is available at <https://gitlab.di.unito.it/alice/bspl-middleware>.

²<https://www.h2database.com/html/main.html>

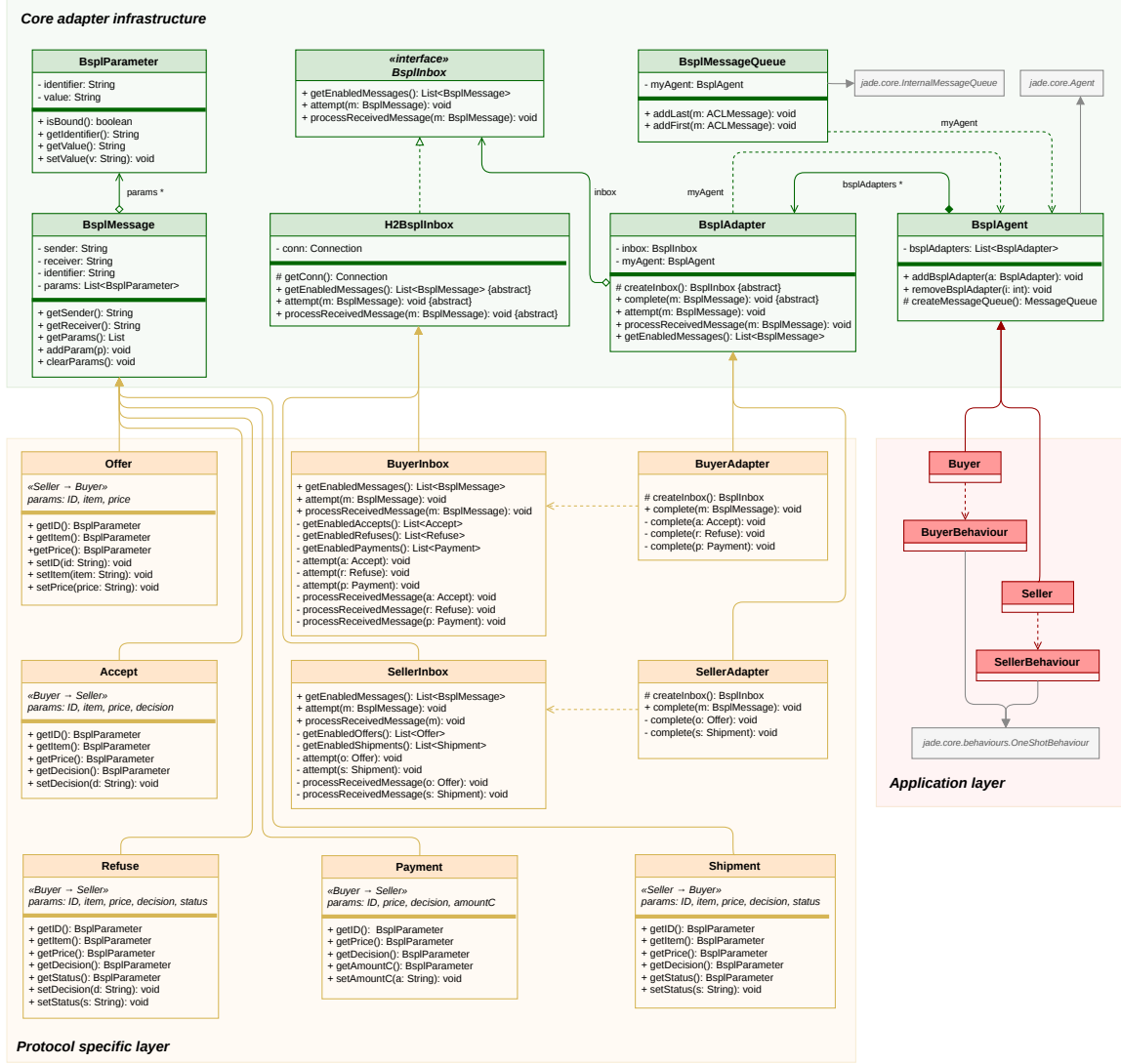


Figure 1: Class diagram of the proposed architecture.

conditions, and database queries explicit. It should not be understood as the only intended deployment solution. The architecture only assumes that the inbox provides transactional storage and query support for parameter bindings; in realistic distributed deployments, the same abstraction could be backed by a persistent or distributed DBMS.

Each protocol parameter is tracked as a dedicated table, indexed by the corresponding protocol key(s). This relational representation makes the BSPL information-causality constraints directly expressible as SQL queries. Message enablement is computed by querying the local database looking for bindings that satisfy the $\lceil \text{in} \rceil$ parameters requirements, and verifying the lack of bindings for $\lceil \text{out} \rceil$ parameters. On the reception of a message, information consistency is enforced through transactional insertions that respect the integrity constraints of the protocol specification: in case a received message contains a different binding for a parameter already known, the message is discarded and the database rollback mechanisms are exploited to restore the local state. The result is a clean separation between the agent's application logic and its protocol execution state, with the latter fully managed by the adapters.

The core adapter infrastructure, the green classes in Figure 1, provides the required classes and interfaces to realize the above picture in general terms, i.e., independently of any specific BSPL protocol. Of course, these classes and interfaces will need to be extended and made concrete for implementing a

specific BSPL protocol (see next section). Let us now quickly describe the main classes and interfaces of this core layer.

BsplMessage and BsplParameter classes `BsplMessage` is a serializable class that represents a single protocol message at runtime. It carries the message identifier, the name of the sender, the name of the intended receiver, and an ordered list of `BsplParameter` objects. Each parameter is a serializable name–value pair; a parameter whose value field is null is considered unbound. Message objects are transmitted as the serialized content of standard JADE ACL messages. Concrete message classes, coded per-protocol in the protocol specific layer (see below), extend `BsplMessage` and expose typed getters and setters for each named parameter, providing a type-safe programming interface over a generic parameter list.

BsplInbox interface `BsplInbox` is the central abstraction of the framework. It is responsible for maintaining the information state of all active protocol instances for a given role and enforcing the BSPL enablement and consistency conditions. It defines four methods:

- `getEnabledMessages()` Returns all messages currently enabled for the corresponding role;
- `getEnabledMessages(Filter)` Returns enabled messages matching some conditions expressed by a filter object;
- `attempt(BsplMessage)` Sends the message and updates the local state with the new bindings of out parameters, provided that the new bindings respect the BSPL protocol specification constraints;
- `processReceivedMessage(BsplMessage)` Records a received message by updating the internal state (if consistent).

Of course, at this point, these methods lack the implementation, but the interface enforces the services and tasks that a concrete inbox has to implement to guarantee compliance with a BSPL protocol specification. In the following we explain how this is achieved concretely.

H2BsplInbox class In principle, the proposed architecture allows for alternative ways to implement a `BsplInbox`, here we suggest the adoption of a database for scalability purposes. `H2BsplInbox` is the abstract base class (implementing `BsplInbox`) for inbox implementations that use a relational database for state management. On construction, `H2BsplInbox` opens an in-memory H2 database connection that persists for the lifetime of the JVM process, accessed through JDBC in the prototype implementation. The relational representation maps tables directly onto the BSPL parameter model. For a protocol like the EBusiness with key parameter `ID` and additional parameters `item`, `price`, `decision`, and `status`, the inbox creates one table per parameter (e.g., `table_item(ID, item)`, `table_decision(ID, decision)`, etc.), with the key parameter as the primary key. This schema allows the enablement conditions of BSPL messages to be expressed as SQL queries. A message with `in` parameters $p_1, \dots, p_i$ and `out` parameters $q_1, \dots, q_j$ is enabled for all enactment keys for which rows exist in tables `table_p1`, ..., `table_pi`, but not yet in tables `table_q1`, ..., `table_qj`, expressed as an SQL JOIN over the `in` parameter tables combined with a set of NOT EXISTS sub-queries over the `out` parameter tables.

The `attempt()` method, invoked when the agent decides to send a message, inserts the newly bound `out` parameters into the appropriate tables within a transactional block, ensuring that state updates are atomic and consistent. The `processReceivedMessage()` method, invoked on receipt, similarly inserts the received parameter bindings, if consistent.

BsplAdapter class The `BsplAdapter` abstract class constitutes the primary integration point between the protocol infrastructure and the application-level agent behavior. Each adapter manages a single `BsplInbox` instance and maintains a reference to the owning agent, enacting a protocol role.

The adapter delegates enabled-message computation entirely to its inbox, and exposes two additional methods:

- `attempt(BsplMessage)` passes an outgoing message to the inbox for updating the information state and takes care of the actual transmission;
- `complete(BsplMessage)` needs to be implemented at the application level to populate domain-specific `out` parameters of each enabled message type.

More in detail, `attempt(BsplMessage)` is a concrete method that (a) delegates to `inbox.attempt(BsplMessage)` to record the send in the relational state, (b) populates the sender field with the agent's local name, and (c) constructs an actual JADE `ACLMessage`, serializes the `BsplMessage` as its content object, and dispatches it through the JADE middleware. In turn, `complete(BsplMessage)` amounts to an abstract method to be overridden by the protocol specific-level adapter. It is called once an enabled message has been selected; the implementation must bind all the `out` parameters of the message (e.g., by consulting the agent's protocol specific-level knowledge).

The rationale at the basis of `BsplAdapter` is a two-step process handling the enabled message selected by the agent for sending: first, the message `out` parameters need to be bound, then the message sending is attempted, and if successful, the agent's local state is updated. Concrete adapter classes (protocol-specific) implement the `complete()` method that acts as a dispatcher: based on the concrete type of the incoming message, it selects and invokes a corresponding strongly typed private method responsible for completing that specific message type.

BsplAgent class This class is the main entry point for building BSPL-compliant JADE agents. Agent developers who would like to leverage the presented architecture should build their agents so that they extend the `BsplAgent` class. In detail, `BsplAgent` extends JADE's `Agent` class and introduces two important mechanisms:

- *Adapter registry.* Each `BsplAgent` maintains a list of `BsplAdapter` instances. Adapters are registered at runtime via `addBsplAdapter(BsplAdapter)` and may be removed via `removeBsplAdapter(BsplAdapter)`. An agent participating in multiple concurrent protocols registers one adapter per each played role.
- *Message queue interception.* `BsplAgent` overrides the `createMessageQueue()` factory method inherited from JADE's `Agent` class to return a `BsplMessageQueue` instead of the default queue. This is the integration point between JADE's transport infrastructure and the BSPL protocol layers. When JADE delivers an incoming ACL message to the queue, this dedicated message queue implementation deserializes the `BsplMessage` from the ACL content object and delivers it to all the registered adapters – each of which will update its inbox accordingly – before forwarding the raw ACL message to the standard queue. In this way, the infrastructure ensures that the local state is always up to date whenever the agent calls `getEnabledMessages()` on an adapter.

BsplMessageQueue `BsplMessageQueue` extends JADE's standard `InternalMessageQueue`. It overrides the `addLast()` and `addFirst()` methods to intercept each incoming message and trigger inbox update whenever an instance of `BsplMessage` is received, without altering the standard delivery semantics observed by JADE behaviors.

3.2. Protocol Specific Layer

All the abstract classes in the core adapter infrastructure must, then, be instantiated in accordance with a given protocol specification. This applies to messages, inboxes and adapters.

Concrete message classes For each BSPL message defined in the protocol specification, a concrete subclass of `BsplMessage` is defined. Each subclass hard-codes the message identifier and the ordered parameter list that matches the protocol specification, and exposes named getters and setters for parameters. For example, see Listing 2, the `Accept` message is constructed with pre-bound `in` parameters (`ID`, `item`, `price`) and an unbound `out` parameter (`decision`). It is worth noting that setters are available for `out` parameters only; `in` parameter values (plus the message receiver) are instead initialized through the constructor upon creation.

Listing 2: `Accept` message class.

```

1 // Buyer -> in Seller : accept [in ID key, in item, in price, out decision]
2 public class Accept extends BsplMessage {
3
4     public Accept(String receiver, String ID, String item, String price) {
5         super("accept", receiver,
6             new BsplParameter("ID", ID),
7             new BsplParameter("item", item),
8             new BsplParameter("price", price),
9             new BsplParameter("decision")
10        );
11    }
12
13    public BsplParameter getID() { return this.getParams().get(0); }
14    public BsplParameter getItem() { return this.getParams().get(1); }
15    public BsplParameter getPrice() { return this.getParams().get(2); }
16    public BsplParameter getDecision() { return this.getParams().get(3); }
17
18    public void setDecision(String decision) { this.getParams().get(3).setValue(decision); }
19 }

```

Concrete inbox classes Each agent role defined in the protocol – Buyer and Seller in our example – is associated with a dedicated pair of classes: a concrete inbox and a concrete adapter. Together, these components intercept all message traffic and maintain track of the information state of the running protocol instances.

Each inbox class (`BuyerInbox` and `SellerInbox` in our example) extends `H2BsplInbox` and is responsible for maintaining the information state of all active protocol instances for the given role and enforcing the BSPL enablement and consistency conditions. Upon construction, each inbox initializes the embedded relational database with one table per protocol parameter (`table_ID`, `table_item`, `table_price`, `table_decision`, `table_status`, `table_amountC`), plus a table recording the identity of the other role players (`table_seller` or `table_buyer`, respectively). In our example, all tables use `ID` as a primary key. Listing 3 shows the `BuyerInbox`'s constructor.

Listing 3: `BuyerInbox`'s constructor.

```

1 public BuyerInbox() {
2     try {
3         Statement stmt = getConn().createStatement();
4         stmt.execute("CREATE TABLE table_ID (ID VARCHAR(255) PRIMARY KEY)");
5         stmt.execute("CREATE TABLE table_item (ID VARCHAR(255) PRIMARY KEY, item VARCHAR(255))");
6         stmt.execute("CREATE TABLE table_price (ID VARCHAR(255) PRIMARY KEY, price VARCHAR(255))");
7         stmt.execute("CREATE TABLE table_decision (ID VARCHAR(255) PRIMARY KEY, decision VARCHAR(255))");
8         stmt.execute("CREATE TABLE table_status (ID VARCHAR(255) PRIMARY KEY, status VARCHAR(255))");
9         stmt.execute("CREATE TABLE table_amountC (ID VARCHAR(255) PRIMARY KEY, amountC VARCHAR(255))");
10        stmt.execute("CREATE TABLE table_seller (ID VARCHAR(255) PRIMARY KEY, seller VARCHAR(255))");
11    } catch (SQLException e) { ... }
12 }

```

The `getEnabledMessages()` methods execute SQL queries that express the BSPL enablement conditions for each message type the role may send. For the Buyer role, this amounts to retrieving all the enabled `Accept`, `Refuse` and `Payment` messages. An `Accept` message (Listing 4), for instance, is enabled when a key (`ID`) is present with bound `item`, `price`, and `seller` parameters, but no `decision` parameter binding yet exists. For each enabled message, a corresponding `BsplMessage` of the corresponding type is created and returned.

Listing 4: Excerpt of the BuyerInbox's getEnabledMessages() method implementation.

```

1 @Override
2 public List<BsplMessage> getEnabledMessages() {
3     List<BsplMessage> messages = new ArrayList<BsplMessage>();
4     messages.addAll(getEnabledAccepts());
5     messages.addAll(getEnabledRefuses());
6     messages.addAll(getEnabledPayments());
7     return messages;
8 }
9
10 // Buyer -> in Seller : accept [in ID key, in item, in price, out decision]
11 private List<Accept> getEnabledAccepts() {
12     List<Accept> res = new ArrayList<>();
13     try {
14         Statement stmt = getConn().createStatement();
15         ResultSet rs = stmt.executeQuery("SELECT t.ID, ti.item, tp.price, ts.seller\n" +
16                                         "FROM table_ID AS t\n" +
17                                         "JOIN table_item AS ti ON t.ID = ti.ID\n" +
18                                         "JOIN table_price AS tp ON t.ID = tp.ID\n" +
19                                         "JOIN table_seller AS ts ON t.ID = ts.ID\n" +
20                                         "WHERE NOT EXISTS (\n" +
21                                         "    SELECT *\n" +
22                                         "    FROM table_decision AS td\n" +
23                                         "    WHERE td.ID = t.ID\n" +
24                                         ");");
25     };
26     while (rs.next()) {
27         String id = rs.getString("id");
28         String item = rs.getString("item");
29         String price = rs.getString("price");
30         String receiver = rs.getString("seller");
31         Accept a = new Accept(receiver, id, item, price);
32         res.add(a);
33     }
34 }
35 } catch (SQLException e) { ... }
36 return res;
37 }
38
39 private List<Refuse> getEnabledRefuses() { ... }
40 private List<Payment> getEnabledPayments() { ... }

```

The processReceivedMessage(BsplMessage) method, in turn, inserts the parameter bindings of each received message into the appropriate tables. Listing 5 shows an excerpt of the method implementation for the BuyerInbox. It is worth noting that parameter bindings are stored only if no other binding is already present for the same parameter and the same key(s). Otherwise, the whole message is discarded. For example, for an Offer message (see below) a query is executed to check if another binding for item and price (and sender) already exists for the same ID in the corresponding tables. New bindings are stored transactionally only if does not hold.

Listing 5: Excerpt of the BuyerInbox's processReceivedMessage(BsplMessage) method implementation.

```

1 @Override
2 public void processReceivedMessage(BsplMessage msg) {
3     if (msg instanceof Offer) {
4         processReceivedMessage((Offer) msg);
5     }
6     else if (msg instanceof Shipment) {
7         processReceivedMessage((Shipment) msg);
8     }
9 }
10
11 // Seller -> out Buyer : offer [out ID key, out item, out price]
12 private void processReceivedMessage(Offer offer) {
13     String ID = offer.getID().getValue();
14     String item = offer.getItem().getValue();
15     String price = offer.getPrice().getValue();
16     String sender = offer.getSender();
17
18     Connection conn = getConn();

```

```

19
20     synchronized(conn) {
21         try {
22             conn.setAutoCommit(false);
23             Statement stmt = getConn().createStatement();
24
25             ResultSet rs = stmt.executeQuery("SELECT *\n" +
26                 "FROM table_item\n" +
27                 "WHERE ID = '" + ID + "' AND item != '" + item + "';"
28             );
29             if (rs.next()) { // The result set is not empty, skipping the message
30                 return;
31             }
32             ...
33
34             stmt.execute("INSERT INTO table_ID (ID)\n" +
35                 "SELECT '" + ID + "'\n" +
36                 "WHERE NOT EXISTS(\n" +
37                 "SELECT '" + ID + "' FROM table_ID WHERE ID = '" + ID + "'\n" +
38                 ")")
39         );
40         stmt.execute("INSERT INTO table_item (ID, item)\n" +
41             "SELECT '" + ID + "', '" + item + "'\n" +
42             "WHERE NOT EXISTS(\n" +
43             "SELECT '" + ID + "' FROM table_item WHERE ID = '" + ID + "'\n" +
44             ")")
45         );
46         ...
47
48         conn.commit();
49     } catch (SQLException e) {
50         conn.rollback();
51     } finally {
52         conn.setAutoCommit(true);
53     }
54 }
55 }
56
57 private void processReceivedMessage(Shipment shipment) { ... }

```

Similarly, the attempt (BsplMessage) method inserts the newly produced $\lceil$ out $\rceil$ bindings transactionally. As before, bindings are stored only if another binding for the same key(s) does not already exist. Listing 6 sketches the BuyerInbox's method implementation.

Listing 6: Excerpt of the BuyerInbox's attempt (BsplMessage) method implementation.

```

1 @Override
2 public void attempt(BsplMessage message) {
3     if (message instanceof Accept) {
4         attempt((Accept) message);
5     }
6     else if (message instanceof Refuse) {
7         attempt((Refuse) message);
8     }
9     else if (message instanceof Payment) {
10        attempt((Payment) message);
11    }
12 }
13
14 // Buyer -> in Seller : accept [in ID key, in item, in price, out decision]
15 private void attempt(Accept accept) {
16     String ID = accept.getID().getValue();
17     String decision = accept.getDecision().getValue();
18
19     Connection conn = getConn();
20
21     try {
22         synchronized(conn) {
23             conn.setAutoCommit(false);
24             Statement stmt = conn.createStatement();
25             stmt.execute("INSERT INTO table_decision (ID, decision) VALUES ('" + ID + "', '" + decision + "')
26                 ;");

```

```

27     conn.commit();
28     }
29     } catch (SQLException e) {
30         conn.rollback();
31     }
32     finally {
33         conn.setAutoCommit(true);
34     }
35 }
36
37 private void attempt(Refuse refuse) { ... }
38 private void attempt(Payment payment) { ... }

```

Concrete adapter classes Concrete adapter classes (BuyerAdapter and SellerAdapter in our example) extend BsplAdapter. Each adapter's createInbox() factory method returns the corresponding concrete inbox. Additionally, each adapter's complete(BsplMessage) method dispatches on the runtime type of the message and delegates to a typed private method that sets the "out" parameter values from protocol-specific agent-level knowledge (e.g., BuyerAdapter.complete(Accept) may set the decision parameter to "OK"). It is important to note that, because the way messages are completed depends on the agent's internal logic, the implementation of these methods is the responsibility of the agent developer. This is the only part of the entire architecture whose implementation is delegated to agent developers; all the other components, by contrast, can be strictly derived from the BSPL protocol specification.

Listing 7 sketches the BuyerAdapter class.

Listing 7: Excerpt of the BuyerAdapter class.

```

1 public class BuyerAdapter extends BsplAdapter {
2
3     @Override
4     protected BsplInbox createInbox() {
5         return new BuyerInbox();
6     }
7
8     @Override
9     public void complete(BsplMessage message) {
10         if (message instanceof Accept) {
11             complete((Accept)message);
12         } else if (message instanceof Refuse) {
13             complete((Refuse)message);
14         } else if (message instanceof Payment) {
15             complete((Payment)message);
16         }
17     }
18
19     // Buyer -> in Seller : accept [in ID key, in item, in price, out decision]
20     private void complete(Accept message) {
21         message.setDecision("OK");
22     }
23
24     private void complete(Refuse message) { ... }
25     private void complete(Payment message) { ... }
26 }

```

4. Discussion

The architecture presented in this paper follows the line drawn in [17, 18, 19], which tries to address the critiques raised by Winikoff in [21], concerning interaction support in AOPLs. The major critique is that, despite interaction being a defining feature of multi-agent systems, mainstream AOPLs often provide only low-level primitives for message sending and reception, leaving developers to reconstruct higher-level coordination structures manually. Moreover, traditional message-centric protocols tend to impose overly rigid interaction structures, tightly coupling communication patterns with agent behavior.

In this sense, the problem is not simply that agents exchange messages, but that the programming model exposes message exchange as the primary abstraction for implementing interaction.

One of the most significant practical consequences of this proposal, in turn, is the degree to which it simplifies the programming model exposed to agent developers. In a conventional JADE application, writing a protocol-conformant agent requires the developer to maintain, explicitly and manually, a representation of the current interaction state: which messages have been sent and received, in which order, which conditions must hold before the next message may be sent, and which sequences of events are still possible. This logic is typically scattered across guard conditions and branching control flow inside the agent’s behaviors. This makes it difficult to verify that the implementation faithfully reflects the protocol specification, and it is hard to maintain as the protocol evolves. The proposal that we have made simplifies the process, guiding the development based on a sequence of standardized simple steps the agent is expected to perform during a protocol enactment:

1. *Get the enabled messages.* The agent queries the role adapter for the set of messages that it is currently permitted to send, given the information accumulated so far during the interaction.
2. *Select one message according to goals.* The agent applies its own decision logic — goals, preferences, beliefs, or any other application-level criterion — to choose a message from the enabled set.
3. *Complete the message.* The agent binds the output parameters of the chosen message with concrete values drawn from its application-level knowledge.
4. *Attempt to send it.* The agent hands the completed message back to the adapter, which commits the new parameter bindings to the local state and delivers the message to the intended peer.
5. *Repeat.* The execution cycle begins again, with the enabled set recomputed to reflect the new local state.

This approach mirrors the pioneering Orpheus and Azorus programming model outlined in [17, 18, 19, 22] for the Jason language [23], and applies it to JADE by changing the level at which protocol programming takes place. The agent developer is no longer required to directly encode a procedural interaction state machine inside JADE behaviors. Instead, behaviors interact with BSPL-aware adapters that expose interaction states in terms of enabled messages. Thus, the developer can focus on coding the agent’s choices, preferences, and domain decisions, while the adapter manages the agent’s local state resulting from interaction. This realizes, within JADE, the kind of higher-level communication abstraction advocated in [21]. Indeed, Orpheus [17] shows in Jason that, by adopting a message-centric implementation approach, even relatively simple modifications to a protocol specification – such as adding an alternative interaction path – cause changes to propagate across several agents and plans. The reason is not only the number of messages, but the fact that asynchronous delivery forces developers to account for different possible arrival orders, and hence to reconstruct the interaction context from message history. This leads to an undesirable coupling between the interaction state and the agent control flow. Plans do not only express what the agent wants to do, but also encode which messages may have arrived, in which combinations, and at which protocol step.

The architecture proposed in this paper addresses the same problem at the design level in JADE by replacing this message-centric perspective with an “information-centric” one. Thus, the proposal is also closely related to the Kiko model, which also adopts BSPL information protocols as the basis for programming protocol-compliant agents in pure Python [15]. Our work follows the same intuition, but specializes it to the JADE ecosystem. In this sense, our contribution can be seen as a JADE-based realization of the architectural principles underlying both Kiko and Orpheus. The novelty of our work lies in showing how these principles can be realized inside the JADE ecosystem, preserving JADE’s behavior model and ACL transport while introducing BSPL-aware adapters, message-queue interception, and relational state management as middleware-level mechanisms. This is achieved through a layered design that sits upon the JADE infrastructure. Our proposal, in fact, introduces no severe modifications to the JADE runtime, it imposes no restrictions on the JADE behavior model, and creates no additional thread-management concerns for agent developers. Instead, our proposal introduces components to enforce information-flow constraints, transparently. Concretely, the proposal exploits a single, well-defined JADE extension point, namely the message queue factory, to intercept incoming BSPL messages

before they are observed by ordinary behaviors. This design preserves the practical advantages of JADE, i.e., its agent lifecycle, the ACL transport layer, the behavior abstraction, and its deployment infrastructure, while at the same time importing the declarative semantics of BSPL to govern interaction.

The architecture is intended to preserve BSPL semantics by maintaining the following invariant: for each role adapter, the local relational state contains exactly the parameter bindings that the role player has produced or received so far for each key. Message enablement is computed only from this local state, following BSPL's causality requirement that all $\lceil \text{in} \rceil$ parameters are known and all $\lceil \text{out} \rceil$ parameters are still unbound. Message reception and message sending update the local state transactionally, so that integrity violations are rejected before they become visible to the agent. The interception of BSPL messages in JADE's standard message queue, in turn, does not alter the order or delivery semantics of ACL messages observed by ordinary JADE behaviors. It only ensures that BSPL messages are processed by the registered adapters before being exposed to the application-level behaviors. Thus, under the assumption that BSPL messages are exchanged only through adapters, the agent's behaviors involved in the interaction are constrained to be compliant with BSPL sending and receiving semantics. A full formal proof of semantic preservation with respect to JADE's asynchronous execution model is beyond the scope of this paper and could be part of future work.

A first practical consequence of this design is the emergence of a uniform interaction cycle for programming protocol-compliant agents. This cycle also makes explicit a separation of concerns; the adapter determines what the agent is allowed to do according to the BSPL specification and the current information state, whereas the agent's behaviors determine what the agent wants to do according to its own goals, preferences, beliefs, or domain knowledge. No explicit representation of the interaction state is mixed with the agent's application logic: no flags, counters, or state-machine automata tracking which messages have been exchanged are needed. Moreover, there is no guard checking whether a given message has been received before constructing another one. Finally, no explicit `receive()` calls are necessary. All of these concerns are handled by the adapter and inbox infrastructure transparently. In the Ebusiness example, the Buyer agent does not need to test whether an offer has been received before constructing an `Accept` or `Refuse` message. It simply asks the adapter for the enabled messages. Those messages appear only when the required $\lceil \text{in} \rceil$ parameters are present in its information state. What remains in the agent's code is the Buyer's application-level decision, e.g., whether the offered item is desired and whether the price is acceptable.

A further intended implication of the architecture is robustness by construction. An agent programmed by following the proposed architecture cannot send a message whose $\lceil \text{in} \rceil$ parameters are not yet known, because such a message is not enabled yet. Similarly, it cannot deliberately send a message that violates the protocol's safety conditions, because `attempt()` records the new bindings only if they are consistent with the information state. A clean separation also yields between framework-level and application-level responsibilities. The core adapter infrastructure provides reusable abstractions, while the protocol-specific layer instantiates these abstractions for a concrete BSPL specification. The use of a relational database as the backend for maintaining information state is central to this separation. BSPL's parameter model maps naturally to a relational representation. Parameter bindings are stored in tables indexed by protocol keys, and enablement conditions become SQL queries over the known bindings. Similarly, reception and sending are handled through transactional insertions, so that inconsistent updates can be rejected atomically.

5. Conclusion

This paper presented a JADE-based realization of an architecture to support information-centric interaction protocols, showing how BSPL semantics can be integrated into an established multi-agent platform, preserving at the same time its execution model. Our proposal introduces protocol-aware adapters and inboxes that mediate between JADE agents and the underlying communication infrastructure, shifting the responsibility for interaction state management from agent behaviors to reusable middleware components. By representing protocol knowledge in a relational database, the approach makes message

enablement and consistency checking operationally explicit and directly computable, while preserving the asynchronous and distributed nature of JADE communication. This provides a practical bridge between the declarative advantages of BSPL and the maturity of JADE, supporting the development of MAS that are more modular, robust, and easier to evolve.

The current implementation leaves two main tasks to the developer that could be automated. First, adapters and inboxes must be written manually for each protocol role. Nonetheless, it is worth noting that these implementations follow a mechanical pattern derivable from the protocol specification. A natural direction for future work could be to generate concrete adapter and inbox classes directly from the parsed BSPL specification in the line of [17, 18, 19]. Second, the `complete()` method in each concrete adapter requires the developer to consider all the message types that the enacted role may send, and provide a binding strategy for their `⌈out⌋` parameters. This task is inherently application-dependent, because the values to be assigned to these parameters depend on the agent's goals, beliefs, and domain knowledge. Still, for protocols with many message types, code generation could produce a template-based structure with typed completion stubs, leaving the developer to fill in only the domain-specific logic. The present paper focuses on the architectural realization of BSPL-compliant interaction support in JADE, and uses the EBusiness protocol as a running example to illustrate feasibility and clarity of the approach. A broader empirical and quantitative evaluation, involving larger protocols, multi-party settings, and incremental protocol extensions, remains an important direction for future work to assess performance, scalability, and developer effort more systematically. In particular, a comparison with conventional JADE implementations would help quantify the trade-off between the additional middleware layer introduced by the adapters and the expected gains in modularity and maintainability. Such an evaluation could follow the approach adopted in [22].

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT-5.5 and Grammarly in order to: Grammar and spelling check. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

- [1] R. Vieira, Á. F. Moreira, M. J. Wooldridge, R. H. Bordini, On the formal semantics of speech-act based communication in an agent-oriented programming language, *Journal of Artificial Intelligence Research (JAIR)* 29 (2007) 221–267. doi:10.1613/jair.2221.
- [2] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, A. Santi, Multi-agent oriented programming with JaCaMo, *Science of Computer Programming* 78 (2013) 747–761. doi:10.1016/j.scico.2011.10.004.
- [3] S. Galland, S. Rodriguez, N. Gaud, Run-time environment for the SARL agent-programming language: The example of the Janus platform, *Future Generation Computer Systems* 107 (2020) 1105–1115. doi:10.1016/j.future.2017.10.020.
- [4] F. Bergenti, E. Iotti, S. Monica, A. Poggi, Agent-oriented model-driven development for JADE with the JADEL programming language, *Computer Languages, Systems & Structures* 50 (2017) 142–158. doi:10.1016/j.cl.2017.06.001.
- [5] F. Bellifemine, A. Poggi, G. Rimassa, Developing multi-agent systems with JADE, in: *International workshop on agent theories, architectures, and languages*, Springer, 2000, pp. 89–103.
- [6] F. Bellifemine, G. Caire, D. Greenwood, *Developing Multi-Agent Systems with JADE*, Wiley, Chichester, UK, 2007. doi:10.1002/9780470058411.
- [7] M. E. Gregori, J. P. Cámara, G. A. Bada, A jabber-based multi-agent system platform, in: *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, 2006, pp. 1282–1284.

- [8] J. Palanca, A. Terrasa, V. Julian, C. Carrascosa, Spade 3: Supporting the new generation of multi-agent systems, *IEEE Access* 8 (2020) 182537–182549.
- [9] C. Larman, *Applying UML And Patterns: An Introduction To Object-Oriented Analysis And Design And Iterative Development*, Prentice Hall, 2004.
- [10] I. Sommerville, *Software engineering*, Pearson, 2015.
- [11] A. K. Chopra, S. H. Christie V, M. P. Singh, An evaluation of communication protocol languages for engineering multiagent systems, *Journal of Artificial Intelligence Research (JAIR)* 69 (2020) 1351–1393. doi:10.1613/jair.1.12212.
- [12] M. P. Singh, 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, 2011*, pp. 491–498. doi:10.5555/2031678.2031687.
- [13] M. P. Singh, Semantics and verification of information-based protocols, in: *Proceedings of the 11th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS), IFAAMAS, Valencia, Spain, 2012*, pp. 1149–1156. doi:10.5555/2343776.2343861.
- [14] M. P. Singh, S. H. Christie V, Tango: Declarative semantics for multiagent communication protocols, in: *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI), IJCAI, Online, 2021*, pp. 391–397. doi:10.24963/ijcai.2021/55.
- [15] S. H. Christie V, A. K. Chopra, M. P. Singh, Kiko: Programming agents to enact interaction protocols, in: *Proceedings of the 22nd International Conference on Autonomous Agents and MultiAgent Systems (AAMAS), IFAAMAS, London, 2023*, p. to appear.
- [16] S. H. Christie V, A. K. Chopra, M. P. Singh, Mandrake: Multiagent systems as a basis for programming fault-tolerant decentralized applications, *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 36 (2022) 16:1–16:30. doi:10.1007/s10458-021-09540-8.
- [17] M. Baldoni, S. H. Christie V, M. P. Singh, A. K. Chopra, Orpheus: Engineering Multiagent Systems via Communicating Agents, in: T. Walsh, J. Shah, Z. Kolter (Eds.), *Proc. Thirty-Ninth AAAI Conference on Artificial Intelligence, AAAI 2025*, volume 39, AAAI Press, Philadelphia, Pennsylvania, PA, USA, 2025, pp. 23135–23143. URL: <https://doi.org/10.1609/aaai.v39i22.34478>. doi:10.1609/AAAI.V39I22.34478.
- [18] M. Baldoni, S. H. Christie V, M. P. Singh, A. K. Chopra, Orpheus: Programming Protocol-Based BDI Agents, in: S. Das, A. Nowé, Y. Vorobeych (Eds.), *Proc. of 24th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2025, IFAAMAS / ACM, Detroit, Michigan, USA, 2025*, pp. 3006–3008. URL: <https://dl.acm.org/doi/10.5555/3709347.3744074>. doi:10.5555/3709347.3744074.
- [19] A. K. Chopra, M. Baldoni, S. H. Christie V, M. P. Singh, Azorus: Commitments over Protocols for BDI Agents, in: S. Das, A. Nowé, Y. Vorobeych (Eds.), *Proc. of 24th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2025, IFAAMAS / ACM, Detroit, Michigan, USA, 2025*, pp. 490–499. URL: <https://dl.acm.org/doi/10.5555/3709347.3743564>. doi:10.5555/3709347.3743564, finalist Best Paper Award.
- [20] FIPA, *Foundation for intelligent physical agents (FIPA) specification*, 1998. www.fipa.org.
- [21] M. Winikoff, *Challenges and directions for engineering multi-agent systems*, *CoRR* abs/1209.1428 (2012).
- [22] M. Baldoni, C. Baroglio, R. Micalizio, Evaluating the Benefits of Orpheus for Iterative and Incremental Development in MAS, in: *Post-Proceedings of the 14th International Workshop on Engineering Multi-Agent Systems (EMAS 2026) co-located with the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026)*, *Lecture Notes in Artificial Intelligence*, Springer, 2026. To appear.
- [23] R. H. Bordini, J. F. Hubner, M. Wooldridge, *Programming Multi-Agent Systems in AgentSpeak Using Jason*, Wiley-Interscience, 2007.