Jason+BSPL: Including Communication Protocols in Jason

Jason+BSPL: Including Communication Protocols in Jason

Matteo Baldoni[1][0000−0002−9294−0408],
Samuel H. Christie V[2][0000−0003−1341−0087],
Amit K. Chopra[3][00−0003−4629−7594], and
Munindar P. Singh[2][0000−0003−3599−3893]

[1] Università di Torino, Torino, Italy
[2] North Carolina State University, Raleigh, NC, USA
[3] Lancaster University, Lancaster, United Kingdom

Abstract. Current BDI-based programming models for agents lack support for engineering protocol-based agents. We propose Jason+BSPL, an extension of Jason for communicating agents based on protocols that is compatible with the BDI style of programming agents. Whereas Jason is focused on reactions to handle incoming messages, Jason+BSPL supports organizing the business logic of an agent based on its goals. We describe its implementation and we demonstrate how Jason+BSPL simplifies the programming of decentralized multiagent systems compared to the reactive programming model.

Keywords: Jason · BSPL · Engineering MAS · Interaction Protocols.

1 Introduction

Interaction protocols are crucial to engineering multiagent systems (MAS) comprised of autonomous and heterogeneous agents. Specifically, protocols enable realizing MAS as decentralized software systems, that is, as systems without a distinguished locus of state or control.

Although the value of protocols is widely recognized, established MAS programming models and frameworks lack adequate protocol-based programming abstractions. Cognitive (BDI-based) programming models, as exemplified by Jason [28,7], support sending and receiving messages but not implementing agents based on protocols. JaCaMo [5] includes Jason; however, it deemphasizes messaging in favor of operations on centralized artifacts reminiscent of Web services. Notably, JaCaMo acknowledges the lack of protocol-based programming abstractions. JADE [3] supports implementing MAS based on the FIPA interaction protocols [16]; however, the vast majority of applications would require custom, that is, unsupported, protocols.

Protocols promise several benefits to engineering MAS. One, protocols enable realizing MAS in a decentralized manner, that is, without relying on a distinguished locus of state or control. Two, protocols enable structuring an agent's implementation by cleanly separating the reasoning about its communication

state (local state) from its private internal business logic. Three, protocols help avoid errors as an agent programmer does not have to implement reasoning about the communication state. Fourth, protocols support loose coupling by enabling the implementation of an agent without knowledge of other agents' implementations.

Clearly, any MAS implements some conceptual protocol. What is lacking are protocol-based programming abstractions that make realizing a decentralized MAS (via the implementation of its agents) convenient.

The common programming model for dealing with messages is *reactive*. Here, the idea is to write a message handler for each incoming message. This approach goes back to programming message-oriented middleware [18] and was adopted in early work on programming protocols [15]. However, despite its longevity, the approach has the shortcoming that it largely ignores the structuring provided by a protocol and considers each message independently. However, the messages are generally related to each other and an agent usually needs to act based on its state, which depends on the related messages received or sent. Thus, in the reactive model, the agent code reconstructs the necessary state computation (1) tied up with the requisite business reasoning and (2) in more than one place, based on what message emissions and receptions can lead to that state.

In contrast, Jason+BSPL develops an *enablement-based* programming model that (1) applies the protocol semantics to automate part of the state construction and (2) facilitates an agent program to capture the requisite business reasoning.

1.1 Contributions

We contribute Jason+BSPL, a novel protocol-based programming model for multiagent systems. Jason+BSPL brings together two important 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 [24,9]. Specifically, Jason+BSPL's abstractions enable structuring and implementing an agent as a goal-driven entity that performs communication actions (sends messages) in pursuit of its goals.

As an exemplar of interaction orientation, we adopt the Blindingly Simple Protocol Language (BSPL) [25], a declarative, information-based protocol language that supports decentralized enactments. An advantage of BSPL over alternative languages is that it enables specifying flexible protocols [10]. As an exemplar of cognitive programming models, we adopt Jason. Specifically, we implement a BSPL adapter in Jason that supports selecting and performing the *enabled* communicative acts (given the state of protocol enactments). Jason's rule-based nature accords well with BSPL's declarative information-based approach.

1.2 Novelty

We demonstrate a new protocol-based programming model that builds on Jason and provide an implementation that realizes the model in Jason. This approach demonstrates

- Responsiveness to arbitrary events, upon which we check the enablement of potential actions.
- Structuring code so that the programmer focuses on the business logic to take actions in a decentralized setting.
- Loose coupling between the protocol and the business logic by modularly and reusably capturing the reasoning about protocol state.
- Generating rules corresponding to message semantics based solely on message specifications (protocol).
- Ability to handle composite keys to support automatically identifying and correlating information. By using "semantic" keys (i.e., from the application domain), Jason+BSPL makes it easier to apply the logic programming model underlying Jason to capture business logic.

Organization The rest of this paper is organized as follows. Sections 2 and 3 introduce information protocols and Jason. Section 4 introduces our proposed architecture and programming model. Section 5 describes a conceptual evaluation of our approach, showing key distinctions and how it meets important criteria. Sections 6 present a review of the related work, our conclusions, and a discussion of the future work.

2 Background: Information Protocols

An information protocol, as in the Blindingly Simple Protocol Language (BSPL) [25], specifies communication in a multiagent system and provides a basis for implementing its agents in a loosely coupled manner. Listing 1 illustrates the main features of BSPL via our running example.

Listing 1. The initial *NetBill* Protocol (goods before pay)

```
1  NetBill {
2   role (M)erchant, (C)ustomer
3   parameter out ID key, out item, out outcome
4
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 shipped, out cc]
11  M -> C: receipt[in ID key, in price, in cc, out receipt, out done]
```

A protocol specifies *roles* and *message schemas* exchanged by them. A message schema has a name, a sender, a receiver, and one or more parameters, some of which are designated ⌜key⌝. A *message instance* is a tuple of bindings of that schema. The ⌜key⌝ parameters of a schema form a composite key and uniquify its instances.

A role *knows* bindings for some parameters if it has observed (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 solely by their information constraints, not by any arbitrary control flow constraints. 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 1, *epo* (payment) may only happen after *goods* because *epo* has an information dependency on *goods* via shipped.

Notably, 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.

3 Background: Agent Programming in Jason

Jason implements in Java, and extends, the agent programming language AgentSpeak(L) [7]. Jason agents have a BDI architecture. Each has a belief base, and a plan library. It is possible to specify achievement (operator '!') and test (operator '?') goals. Each plan has a triggering event (causing its activation), which can be either the addition or the deletion of some belief or goal. The syntax is declarative.

A Jason plan is specified as *triggering event* : ⟨ *context* ⟩ ← ⟨ *body* ⟩, where *triggering event* denotes the event the plan handles, the *context* specifies the circumstances when the plan could be used, the *body* is the course of action that should be taken. The context and the body can be *true* or omitted when necessary. Jason extends AgentSpeak(L) by introducing annotations for beliefs, goals, and plans, strong negations, personalization for selection functions, and other functionalities useful to programmers [7]. Listing 2 reports a snippet of NetBill protocol implementation in Jason.

Listing 2. Snippet of NetBill implementation in Jason.

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

```
8           goods(Item, Goods)
9     <-    -nbp_state(Id, quoting);
10          +nbp_state(Id, shipping);
11          .send(Customer, tell, goods(Id, Item, Price, Goods)).
```

There are two plans, the triggering events are the adding of the beliefs *request* and *accept*, respectively, when the source is a message from CUSTOMER. In the case of adding a *request*, the offer Price is determined by the predicate price and, if this succeeds, the *quote* is sent. The progressed state of the protocol is recorded in the belief base by adding the predicate nbp_state.

Listing 2 exemplifies the reactive programming model: a plan is executed to handle each received message, resulting typically in the emission of a message (unless the received message is the last in a protocol enactment).

The reactive programming model couples messages with the agent implementation, forcing the developer to remember their causal relationships. Notably, the protocol itself does not feature any message-to-message coupling; when a message may be emitted depends purely upon information.

4 Proposed Programming Model

Moving away from the reactive model, we propose an information-based programming model in which the protocol-related state of an agent is captured via the set of messages it is *enabled* to send in that state. The enablement-based programming model supports programming a Jason agent by focusing the reasoning on what is needed to emit messages while avoiding recapitulating the protocol in each agent's internal implementation. Below, we describe the main ideas of the programming model.

4.1 System Architecture

We describe the main elements of the Jason+BSPL architecture.

Figure 1 shows a Jason agent under Jason+BSPL. The local state comprises beliefs about observed messages and represents the protocol state. It is used for computing enabled messages and validating messages before emission or reception. The internal state comprises beliefs about whatever is relevant to the agent's reasoning and not included in the protocol. There are no other beliefs. The adapter applies the protocol specification to validate messages and update the local state. Each agent has plans and a reasoner who executes its plans. Its control state is given by its current intentions and associated objects.

4.2 Representing the Local State

Jason+BSPL preserves agent autonomy by separating each agent's local state (shared between agents who communicate with each other) from its internal state (not shared).

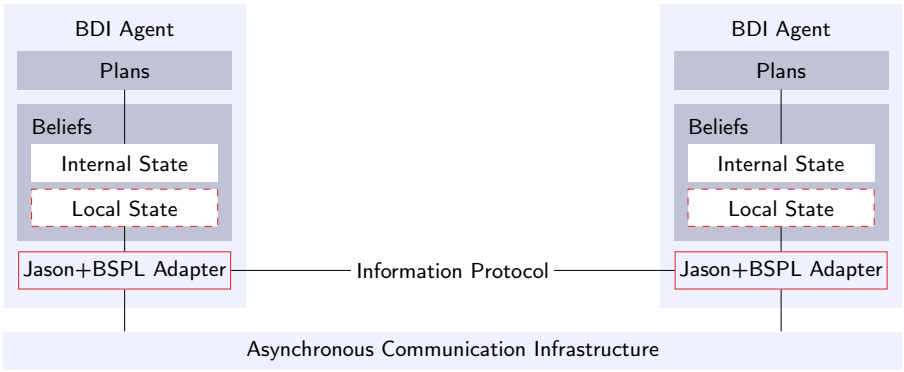

Fig. 1. Each agent has an Jasons+BSPL adapter generated for the roles it plays in the information protocol. The adapter computes enabled messages and validates incoming and outgoing messages against the protocol and updates the local state. The local state is maintained as a set of beliefs in Jason.

In Jason+BSPL, a valid message observation (emission or reception) is represented as a belief and added to the local state. For example, assuming agents Alice and Bob as CUSTOMER and MERCHANT, respectively, the message *request[1, "fig"]*, if it passes validation, is constructed as the Jason term *request("Alice", "Bob", 1, "fig")* and is added to the local state.

Each message observation affects the agent's local state. Given key parameters, additional parameters are deemed known, if the bindings of these parameters exist in the belief base. And, parameters are deemed unknown if their bindings are not already known. An incoming message is compatible with the local state if it is consistent (with respect to its key parameters) with the bindings already present in the local state.

4.3 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 full instances may be emitted. An enabled instance is a partial instance in that its ⌜in⌝ parameters are bound because their bindings are known and its ⌜out⌝ parameters are not bound because they are not known. The idea is given a local state, one can compute the enabled instances, which capture the possible emissions of the agent in that local state.

For example, let MERCHANT's local state contain *request[1, "fig"]* meaning that it has received that *request* from the CUSTOMER. Then, the MERCHANT has two enabled instances in that state: *request*[ID, item] and *quote*[*1*, *"fig"*, price].

4.4 Programming Model

To avoid the message-to-message coupling induced by the traditional reactive programming model, Jason+BSPL supports a novel programming model based

on message *enablement*, in which the developer specifies plans for each of the messages being *emitted* instead of those received. The basic idea of the programming model is captured by the plan pattern in Listing 3: To achieve some agent-specific goal g, the agent queries if there are enabled instances corresponding to the messages it wants to send, and if there are, it *completes* them by producing bindings for their ⌜out⌝ parameters, and then *attempts* to send them all in one fell swoop. The goals !g and !*complete* are agent-specific; in particular, the agent programmer must define the plan corresponding to the business logic of completing enabled instances into full instances. However, the agent programmer does not have to implement the enabled queries or write the plan to achieve !attempt; they are supported by a generic protocol adapter corresponding to the MERCHANT role (Listing 5).

Listing 3. Agent plan pattern in Jason+BSPL.

```
+!g
    :    enabled(m_1) &...& enabled(m_q) & context
    <-   !complete m_1(o_1),...,m_q(o_q);
         !attempt(m_1,...,m_q).
```

Listing 4 shows an application of the pattern in Listing 3 in an implementation of a NetBill MERCHANT from Listing 1. It shows a plan for sending *quote*s; it is triggered whenever the goal of sending a *quote* is asserted; if there is an enabled *quote*; it computes a binding for the ⌜out⌝ parameter price in *quote*; and sends it (via the attempt).

Listing 4. Snippet of NetBill implementation in Jason+BSPL.

```
@quote-plan[atomic]
+!send_quote(Id, Item, Customer)
    :    enabled(quote(id(Id), item(Item), price(out))) &
         price(Item, Price)
    <-   !attempt(quote(id(Id), item(Item),
         price(Price))[receiver(Customer)]).

@quote-plan[atomic]
+!send_goods(Id, Item, Customer)
    :    enabled(goods(id(Id), item(Item), outcome(Outcome),
         shipped(out)) &
         shipped(Item, Shipped)
    <-   !attempt(goods(id(Id), item(Item), outcome(Outcome),
         shipped(Shipped))[receiver(Customer)]).
```

Listing 5 shows a snippet from the NetBill protocol adapter. It receives messages and adds them to the local state after checking consistency, computes enabled predicates, and translates attempts into actual emissions if they pass consistency. This is the code the agent programmer doesn't have to write since it can be generated from Listing 1.

Listing 5. Protocol adapter for MERCHANT role in NetBill. Agent programmer does not have to write the adapter. Any MERCHANT agent can use the adapter in its plans.

```
+m  // If message m passes consistency, add it to local state
    : consistent(m)
    <- +m[Istate].

enabled(quote(id(Key), item(Item), price(out))[receiver(Customer)])
    :-   request(id(Key), item(Item))[source(C), Istate] &
         not sent(quote(id(Key), _, _))[source(myself), Istate].

```

```
enabled(goods...)  //Analogous to enabled(quote) above
  :- ...
enabled(receipt...) //Analogous to enabled(quote) above
  :-...

+!attempt(Messages)
   : compatible(Messages)
       <- for ( .member(Message[receiver(R)], Messages) {
            .send(R, tell, Message);
       }.
```

Messages in an attempt are compatible if there are no two messages that are correlated (are in the same enactment) and share the same ⌜out⌝ parameter. If there were two such messages, that would mean that in one enactment, two bindings for an ⌜out⌝ parameter have been generated, which as explained in Section 2 is not allowed by BSPL semantics. So messages are only emitted via the Jason *.send* action if they are compatible. For example, by the protocol in Listing 1, in any enactment, *accept* and *reject* will be simultaneously enabled. Therefore, an agent programmer may write a plan that attempts to send both. Since they share decision, that means both feature a binding for decision, and therefore actually sending them would violate the protocol. Therefore, the attempt to send both fails. Together, the ideas of enablement and attempt support compliance with the protocol.

5 Evaluation

We now describe our approach in additional detail, weaving in an evaluation with respect to our motivating criteria. Our overarching claim is that Jason+BSPL to better structured agent code that contributes to more loosely coupled systems, meaning that changes to various system components can be made more easily without changing other components. We support this claim by demonstrating several examples.

Listing 2 shows the typical features of a Jason implementation of a NetBill Merchant from Listing 1.

Listing 2 contains two different coordination patterns: subjective and objective [22]. *Subjective* coordination concerns how the merchant drives its interactions with others in view of its own goals. In Listing 2, this is represented by the computation of the price of an item (the predicate price(Item, Price)) and the computation of the goods starting from the item (the predicate goods(Item, Goods)). In Figure 2, this is represented by the process activity in the lifeline (in blue color). *Objective* coordination concerns how the interaction between an agent and its environment (in our case, other agents) is governed so that the MAS has desirable global properties. In the listing and in the Figure 2, objective coordination is represented by the sequence of message exchange, the protocol specification. For example, the reception of *accept* can be processed in the protocol state where merchant has sent *quote* and results in the sending of *goods* and the corresponding state change. *request* can be received in any state (due to the lack of state guard) and results in the sending of a *quote* and the corresponding state change.

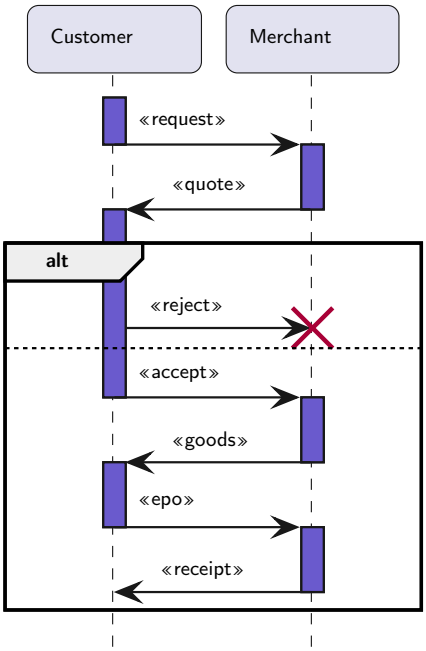

Fig. 2. The UML Sequence Diagram of the NetBill Protocol.

An explicit environment model [29,23,8] that encodes the objective coordination is not suitable for decentralized MAS. The agents must interact directly with each other and agent programmers must ensure that they will behave as expected to ensure global properties (e.g., protocols) [27]. For decentralized settings, [22] motivate agent platforms that provide abstractions that support conveniently engineering MAS in a manner that supports global properties. In particular, they argue for *high cohesion* and *low coupling* of the software in order to facilitate maintenance and reuse. Cohesion captures the unity of purpose of a component, and coupling captures interdependence between components.

Existing programming models, such as Jason [6] and JADE [2], don't provide adequate support for objective coordination. As Listing 2 shows, using Jason, the logic corresponding to objective and subjective coordination is entangled and must be implemented by the agent programmer in an ad hoc manner. This leads to tighter coupling between components and lower cohesion within.

As Listing 2 shows, sending *quote* and *goods* are tightly coupled to receiving *request* and *accept*, respectively. Such tight coupling is the hallmark of the reactive way of implementing protocols. But what if sending a message depends on the receipt of multiple messages? A possible way is to add constraints on the plan (for example, by some conditions in the context). However, doing so mixes the subjective and objective coordination even further, causing tighter coupling and lower cohesion. We discuss this in Section 5.4.

Jason+BSPL offers a way to introduce such abstractions and mechanisms into agent platforms with the ambition of realizing fully decentralized MAS. It

separates the subjective coordination from the objective coordination into different modules. Whereas subjective coordination is specific to the agent, objective coordination, being based on a protocol, is neatly encapsulated in an automatically generated protocol adapter that provides enabled messages. From the agent (programmer's) point of view, the set of enabled messages at any point captures its protocol state, decoupling its history of emissions and receptions from future emissions, and thus avoiding the limitations of the reactive paradigm. The subjective coordination lies in agent-specific goals and plans that pick some enabled messages to flesh out (by binding their ⌜out⌝ parameters) and send them. In a nutshell, sending a message is not necessarily a reaction to the reception of a message. Instead, it is a consequence of the agent pursuing its own goals by taking the enabled messages into account.

In Listing 4, the enabled predicate and the attempt goal neatly encapsulate the objective coordination and separative it from the subjective aspects (when to send a *quote* and with what *price*). This contributes to conveniently engineering agents in a manner that supports low coupling and high cohesion.

Below, we show in more detail how Jason+BSPL supports loose coupling and high cohesion by focusing on how it accommodates changes to the three main conceptual artifacts in a decentralized MAS: (1) the protocol, (2) agents, and (3) the communication service that agents use to exchange messages.

5.1 Changes to Protocol

Listing 1 shows a NetBill protocol in which *goods* must happen before *epo* (payment). Listing 6 shows a NetBill-Mod protocol in which *epo* must happen before *goods*.

Listing 6. The *NetBill* Protocol

```
1  NetBill-Mod {
2   role (M)erchant, (C)ustomer
3   parameter out ID key, out item, out outcome
4
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 cc, out shipped]
10  C -> M: epo[in ID key, in item, in outcome, out cc]
11  M -> C: receipt[in ID key, in price, in cc, out receipt, out done]
```

Listing 4 shows a plan for !send_goods. Given *NetBill-Mod*, the only modification we would have to make to Listing 4 to make it work would be in the signatures of the enabled predicate and the *attempt* goal. Specifically, we need to only replace outcome(Outcome) by shipped(Shipped). The change of the ordering between *goods* and *epo* is neatly encapsulated in *NetBill-Mod*'s adapter and enablement. Thus the impact of the changes in the specification of the protocol is fairly limited on the agent code. By contrast, under the reactive model, the changes are more extensive. For the original *NetBill*, one would have written a plan to process *accept* by sending *goods*. For *NetBill-Mod*, that plan would have to be replaced by a plan that processes *epo* by sending *goods*.

5.2 Changes to Agent Logic

If the protocol remains the same, under Jason+BSPL, changes to an agent's business logic will tend to be highly localized. For example, plan !*send_goods* in Listing 4 could be modified to ship only to *friendly* customers (via an appropriate query). Such changes would also be relatively straightforward in a reactive model except for the fact that the protocol logic and the agent's business logic would be mixed.

5.3 Changes to Communication Infrastructure

Jason+BSPL accommodate changes to the communication infrastructure more easily. A fairly drastic change would be switching from an ordered communication service such as TCP to an unordered service such as UDP. The motivation to do so could be high performance or settings such as the IoT, where TCP is undesirable. Jason+BSPL agents can handle messages in whatever order they arrive: they 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 in the local state. So even without ordering guarantees, Jason+BSPL agents work without needing any change.

5.4 Correlation

Correlation refers to the situation where an agent may wish to combine information from multiple messages to decide its action. In BSPL, such interactions are indicated by a protocol (and some of its messages) having a composite key. We adopt a logistics scenario from [11] to highlight the potential for correlation.

We first describe the scenario, which concerns the handling of *purchase orders* (POs) once they have been placed (by customers) with a Merchant. The other roles involved are Wrapper, Labeler, and Packer. A PO may have several items, each of which may have its own wrapping requirements. Merchant sends each item in a PO separately to Wrapper, who sends wrapped items to Packer. Merchant also 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 3 shows the flow of communication between the roles and Listing 7 captures the protocol in BSPL. Notice the composite key $\langle \mathsf{oID}, \mathsf{iID} \rangle$: oID identifies POs and iID identifies items within a PO.

Listing 7. The *Logistics* Protocol [11]

```
1 Logistics
2  role (M)erchant, (W)rapper, (L)abeler, (P)acker
3
4  parameter out oID key, out iID key, out item,
5            out status
6
7  M -> L: labelRequest[out oID key, out address]
8
```

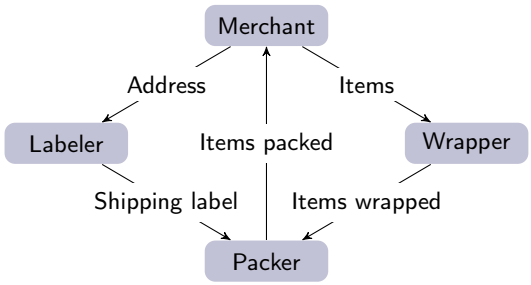

Fig. 3. Logistics scenario, schematically, relative to a PO.

```
M -> W: wrappingRequest[in oID key, out iID key,
        out item]

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 8. An Jason+BSPL plan for sending *packed* messages. Notice the absence of programmer-written logic to correlate label with item.

```
@quote-plan[atomic]
+!send_packed
    : enabled(packed(oID(OID), iID(IID),item(Item),
        wrapping(Wrapping),label(Label),status(out))
    <- !completePacked(_,_,_,_,status(Status));
       !attempt(packed(oID(OID), iID(IID),item(Item),
           wrapping(Wrapping),label(Label), status(Status))).

+!completePacked(_,_,_,_,status(Status))
    <- Status = true.
```

Listing 9 shows, in contrast, the logic a Jason programmer would have to write. Notice that the programmer must implement the correlation via two rules, which leads to complexity.

Listing 9. Jason plans for sending *packed* messages. The programmer would write two plans to capture the orders in which *Labeled* and *Wrapped* messages may arrive. Notice that the programmer must write these plans even if pairwise FIFO messaging is assumed between the agents.

```
+labeled(OID,_,Label)
    :    wrapped(OID, IID,_,Wrapping)
         & packingdone(OID, IID, Status)
    <-   .send(packed(OID, IID, Wrapping, Label, Status))

+wrapped(OID, IID,_,Wrapping)
    :    labeled(OID,_,Label)
         & packingdone(OID, IID, Status)
    <-   .send(packed(OID, IID, Wrapping, Label, Status))
```

6 Related Work and Conclusions

We demonstrated Jason+BSPL, a programming model that combines BDI-based and protocol-based abstractions and as such addresses a crucial gap in multiagent programming models. We demonstrated that Jason+BSPL leads to better code structure by encapsulating the protocol logic in a generic adapter based on the protocol and letting the programmer use it to encode the agent's business logic. We demonstrated that Jason+BSPL leads to less complex code than the traditional reactive model. Further, we demonstrated that Jason+BSPL supports looser coupling and higher cohesion in agents, desirable properties in any software. Finally, whereas traditionally objective coordination has been embodied in the environment, Jason+BSPL shows how it can be embodied in decentralized agents in a modular form separately from subjective coordination.

Jason+BSPL shares some difficulties with rules-based programming with Jason. Errors involving pattern matching are easy to make and hard to find. For example, an enablement handler 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 tooling for generating agent stubs and warning about schema mismatches. Ideally, the programming model should make it clear—via some form of typing—the parameters of an enabled message a plan needs to bind.

[19] propose enhancements to Jason to tackle plan failures atomically. These are interesting improvements but not directly related to our present research questions. Yet, the idea of fault handling in the Jason+BSPL programming model, possibly in conjunction with protocol-based fault handling [12] would be a valuable future direction.

Although some work combines BDI-style agent reasoning with normative abstractions such as commitments [17,20], agent programming languages have generally not kept up with advances in modeling communications. As such, Jason's rule-based representations turn out to be an especially natural fit with protocols.

JaCaMo [4] is a powerful framework for programming MAS that brings together Jason and CArtAgO. We imagine that by improving the communicative foundation of Jason and obviating CArtAgO, Jason+BSPL can enable improvements through the JaCaMo value chain. For instance, [1] show how to extend and apply JaCaMo for commitment reasoning. Jason+BSPL could help place such approaches in decentralized MAS, especially in the light of new results that demonstrate how to enact commitments over protocols [26].

Information protocols provide abstractions that fit well with other data-driven approaches for interaction and business process modeling [14,21]. Jason+BSPL, by providing a connection with Jason, can further help relate data-driven and rule-based BDI approaches. Conceivably, the more flexible parts of agents could be realized in Jason, whereas the more streamlined computations of the internal processes could be carried out in a classical data-driven manner.

A better design of protocols leads to greater decoupling and avoidance of ad hoc functions that operate outside of the BDI structure. The resulting reduction in complexity will help address the difficult task of testing BDI programs [30].

[13] introduce Kiko, an enablement-based programming model for protocols. Unlike Jason+BSPL, Kiko supports Python agents. A big difference is that the Kiko adapter computes all enabled messages and returns them to the agent to select from and complete whereas the Jason+BSPL adapter supports computing solely the enabled messages of interest via queries, thereby avoiding unnecessary computation. A performance evaluation of Jason+BSPL would be valuable.

References

1. Baldoni, M., Baroglio, C., Capuzzimati, F., Micalizio, R.: Commitment-based agent interaction in JaCaMo+. Fundamenta Informaticae **159**(1-2), 1–33 (2018). https://doi.org/10.3233/FI-2018-1656
2. Bellifemine, F., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley, Chichester, UK (2007). https://doi.org/10.1002/9780470058411
3. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley-Blackwell (2007)
4. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A.: Dimensions in programming multi-agent systems. Knowledge Engineering Review (KER) **34**, e2 (Jan 2019). https://doi.org/10.1017/S026988891800005X
5. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with JaCaMo. Science of Computer Programming **78**(6), 747–761 (Jun 2013). https://doi.org/10.1016/j.scico.2011.10.004
6. Bordini, R.H., Hübner, J.F.: Semantics for the Jason variant of AgentSpeak (plan failure and some internal actions). In: Proceedings of the 19th European Conference on Artificial Intelligence (ECAI). Frontiers in Artificial Intelligence and Applications, vol. 215, pp. 635–640. IOS Press, Lisbon (Aug 2010). https://doi.org/10.3233/978-1-60750-606-5-635
7. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason. John Wiley & Sons (2007)
8. CArtAgO: Example 05a – implementing coordination artifacts (2010), http://cartago.sourceforge.net/?page_id=99, common ARTifact infrastructure for AGents Open environments. Accessed: 2021-08-31
9. Castelfranchi, C.: Modelling social action for AI agents. Artificial Intelligence **103**(1-2), 157–182 (1998)
10. Chopra, A.K., Christie V, S.H., Singh, M.P.: An evaluation of communication protocol languages for engineering multiagent systems. Journal of Artificial Intelligence Research **69**, 1351–1393 (2020)
11. Christie V, S.H., Smirnova, D., Chopra, A.K., Singh, M.P.: Protocols over Things: A decentralized programming model for the Internet of Things. IEEE Computer **53**(12), 60–68 (2020)
12. Christie V, S.H., Chopra, A.K., Singh, M.P.: Mandrake: Multiagent systems as a basis for programming fault-tolerant decentralized applications. Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS) **36**(1), 16:1–16:30 (Apr 2022). https://doi.org/10.1007/s10458-021-09540-8

13. Christie V, S.H., Singh, M.P., Chopra, A.K.: Kiko: Programming agents to enact interaction protocols. In: Proceedings of the 22nd International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 1154–1163. IFAAMAS, London (May 2023). https://doi.org/10.5555/3545946.3598758
14. De Masellis, R., Francescomarino, C.D., Ghidini, C., Montali, M., Tessaris, S.: Add data into business process verification: Bridging the gap between theory and practice. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence. pp. 1091–1099. San Francisco (2017)
15. Desai, N., Mallya, A.U., Chopra, A.K., Singh, M.P.: Interaction protocols as design abstractions for business processes. IEEE Transactions on Software Engineering **31**(12), 1015–1027 (Dec 2005). https://doi.org/10.1109/TSE.2005.140
16. FIPA: FIPA interaction protocol specifications (2003), http://www.fipa.org/repository/ips.html, FIPA: The Foundation for Intelligent Physical Agents. Accessed 2023-02-27
17. Günay, A., Winikoff, M., Yolum, P.: Dynamically generated commitment protocols in open systems. Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS) **29**(2), 192–229 (Mar 2015). https://doi.org/10.1007/s10458-014-9251-7
18. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Signature Series, Addison-Wesley, Boston (2004)
19. Kiss, D., Madden, N., Logan, B.: Atomic intentions in Jason+. In: Proceedings of the 8th International Workshop on Programming Multiagent Systems (ProMAS). Lecture Notes in Computer Science, vol. 6599, pp. 79–95. Springer, Toronto (May 2010). https://doi.org/10.1007/978-3-642-28939-2_5
20. Meneguzzi, F., Magnaguagno, M.C., Singh, M.P., Telang, P.R., Yorke-Smith, N.: Goco: Planning expressive commitment protocols. Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS) **32**(4), 459–502 (Jul 2018). https://doi.org/10.1007/s10458-018-9385-0
21. Montali, M., Calvanese, D., Giacomo, G.D.: Verification of data-aware commitment-based multiagent system. In: Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems. pp. 157–164. IFAAMAS, Paris (May 2014)
22. Omicini, A., Ossowski, S.: Objective versus Subjective Coordination in the Engineering of Agent Systems. In: Klusch, M., Bergamaschi, S., Edwards, P., Petta, P. (eds.) Intelligent Information Agents - The AgentLink Perspective. Lecture Notes in Computer Science, vol. 2586, pp. 179–202. Springer (2003). https://doi.org/10.1007/3-540-36561-3_9, https://doi.org/10.1007/3-540-36561-3_9
23. Ricci, A., Ciortea, A., Mayer, S., Boissier, O., Bordini, R.H., Hübner, J.F.: Engineering scalable distributed environments and organizations for MAS. In: Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 790–798. IFAAMAS, Montréal (May 2019). https://doi.org/10.5555/3306127.3331770
24. Singh, M.P.: Agent communication languages: Rethinking the principles. IEEE Computer **31**(12), 40–47 (Dec 1998)
25. Singh, M.P.: Information-driven interaction-oriented programming: BSPL, the Blindingly Simple Protocol Language. In: Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 491–498. IFAAMAS, Taipei (May 2011). https://doi.org/10.5555/2031678.2031687
26. Singh, M.P., Chopra, A.K.: Clouseau: Generating communication protocols from commitments. In: Proceedings of the 34th Conference on Artificial Intelligence

(AAAI). pp. 7244–7252. AAAI Press, New York (Feb 2020). https://doi.org/10.1609/aaai.v34i05.6215

27. Tolksdorf, R.: Models of Coordination. In: Omicini, A., Tolksdorf, R., Zambonelli, F. (eds.) Engineering Societies in the Agent World, First International Workshop, ESAW 2000, Berlin, Germany, August 21, 2000, Revised Papers. Lecture Notes in Computer Science, vol. 1972, pp. 78–92. Springer (2000). https://doi.org/10.1007/3-540-44539-0_6, https://doi.org/10.1007/3-540-44539-0_6
28. Vieira, R., Moreira, Á.F., Wooldridge, M.J., Bordini, R.H.: On the formal semantics of speech-act based communication in an agent-oriented programming language. Journal of Artificial Intelligence Research (JAIR) **29**, 221–267 (Jun 2007). https://doi.org/10.1613/jair.2221
29. Weyns, D., Omicini, A., Odell, J.: Environment as a first class abstraction in multiagent systems. Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS) **14**(1), 5–30 (Feb 2007). https://doi.org/10.1007/s10458-006-0012-0
30. Winikoff, M., Cranefield, S.: On the testability of BDI agent systems. Journal of Artificial Intelligence Research (JAIR) **51**, 71–131 (Sep 2014). https://doi.org/10.1613/jair.4458