Azorus: Commitments over Protocols for BDI Agents

Anonymous Author(s)
Submission Id: 47

Abstract

Commitments support flexible interactions between agents by capturing the meaning of their interactions. However, commitment-based reasoning is not adequately supported in agent programming models. We contribute Azorus, a programming model based on declarative specifications centered on commitments and aligned with information protocols. Azorus supports reasoning about goals and commitments and combines modeling of commitments and protocols, thereby uniting three leading declarative approaches to engineering decentralized multiagent systems. Specifically, we realize Azorus over three existing technology suites: (1) Jason, a popular BDI-based programming model; (2) Cupid, a formal language and query-based model for commitments; and (3) BSPL, a language and its associated tools for information protocols, including Jason programming. We implement Azorus and demonstrate how it enables capturing interesting patterns of business logic.

CCS Concepts

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

Keywords

Decentralization, Decision making, Asynchrony, Causality

ACM Reference Format:

Anonymous Author(s). 2025. Azorus: Commitments over Protocols for BDI Agents. In *Proc. of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), Detroit, Michigan, USA, May 19 – 23, 2025*, IFAAMAS, 9 pages.

1 Introduction

Important domains such as business and healthcare that involve autonomous principals lend themselves to the application of decentralized multiagent systems (MAS). Engineering flexible MAS calls upon programming abstractions for social meaning, operational interactions, and agent reasoning.

Commitments are a high-level abstraction that capture the social *meaning* of a communicative act [26]. For example, an *offer* from a seller to a buyer for some Item and Price may be modeled as a commitment from the seller to the buyer that if *payment* of the Price happens, then the *shipment* of Item will happen. Commitments model autonomy by both enabling flexible engagements between agents and yielding a standard for compliance [19, 33, 37]. There has been work on expressive languages for commitments [9, 13].

Commitments, however, need to be layered on flexible interaction protocols that minimally constrain when agents may perform communicative acts in decentralized settings. For example, *refund* without a prior *payment* would be meaningless; and *accept* and *reject* should be mutually exclusive to be meaningful; however, *shipment* and *payment* may happen in any order. Because of their emphasis on message ordering, traditional protocol specification approaches [2, 4, 16, 35] are not suited to specifying flexible protocols. For this purpose, we turn toward information protocols [27], a declarative approach for specifying flexible protocols. Indeed a motivation for information protocols was a suitable operational layer for commitments [27, p. 498].

Commitments are not adequately supported in programming models for multiagent systems. Popular approaches such as JADE [5], Jason [7], JaCaMo [6], and SARL [20] provide diverse, useful abstractions for engineering multiagent systems. However, the abstractions for communication in these approaches are either low-level (e.g., messaging in JADE and Jason and *event spaces* in SARL), limited in repertoire, inflexible (support for FIPA Interaction Protocols [18] in JADE), or promote centralization (via *artifacts* in JaCaMo). MOISE (the 'Mo' in JaCaMo) [22] supports a notion of commitments but tightly couples them to agent goals; moreover, the commitments are undirected and a distinguished organizational entity tracks and enforces them, betraying an underlying centralized mindset. Baldoni et al. [1] model communicative acts and their effects on commitments via JaCaMo artifacts. Kiko [14], an information protocol-based programming model supports creating flexible, decentralized MAS but does not support commitments.

We contribute *Azorus* (named after the helmsman of Jason's ship, the Argo), a commitment-based programming model that enables implementing flexible MAS via BDI agents. We bring together for the first time three declarative paradigms: commitments, information protocols, and cognitive agents. For the latter, we adopt BDI (belief-desire-intention) agents, which have beliefs and goals, and execute plans in response to changes in beliefs and goals. Jason [7] is a prominent exemplar of the paradigm (and the 'Ja' in JaCaMo). The synthesis makes conceptual sense because in a multiagent system, agents depend on others for the satisfaction of their goals [25]. Commitments capture such dependencies between agents [21], and, as described above, motivate information protocols. Winikoff [34] notes the lack of support for implementing flexible interactions in agent programming approaches. Our synthesis addresses this gap.

- We contribute a novel formalization of Cupid [13], an expressive commitment language, in terms of abstract Jason rules. We provide a compiler from Cupid to Jason that enables a declarative, high-level abstraction for including commitment events in Jason plans.
- We contribute a novel Jason communication adapter that supports an agent's internal reasoning by maintaining the mapping between commitments and enactments of information protocols and providing abstractions for querying and reacting to commitment events and performing valid communicative acts.
- We demonstrate interesting agent reasoning patterns for exploiting Azorus and implementing flexible agents.

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

Organization. The rest of the paper is organized as follows. Section 2 introduces the necessary background on Cupid, information protocols, and Jason. Section 3 introduces the Azorus programming model via its architectural elements, including a Jason semantics for inferring commitment events from communicative acts. Section 4 demonstrates some patterns for implementing flexible agents. Section 5 evaluates our contributions conceptually. Section 6 summarizes our contributions and raises some issues and future directions.

2 Background

We introduce Cupid, a language to specify commitments, information protocols; and finally Jason.

2.1 Specifying Commitments in Cupid

Cupid is an approach for specifying commitments over databases of business events [13].

Listing 1: Commitment specification in Cupid.

```
schema
 offer(Seller, Buyer, Id, Item, Price)
 key Id time Otime
 accept(Buyer, Seller, Id, Item, Price)
 key Id time Atime
 instruct(Buyer, Bank, Id, Price)
 key Id time Itime
 transfer(Bank, Seller, Id, Price, Payment)
 key Id time Ttime
 shipment(Seller, Buyer, Id, Item, Price)
 key Id time Stime
 refund(Seller, Bank, Id, Item, Payment, Amount)
 key Id time Rtime

commitment OfferCom Seller to Buyer
  create offer
  detach transfer[, created OfferCom + 5]
        where "Payment>=Price"
  discharge shipment [, detached OfferCom + 5]

commitment AcceptCom Buyer to Seller
  create accept
  detach shipment[, created AcceptCom + 5]
  discharge transfer[, detached AcceptCom + 5]
        where "Payment>=Price"

commitment RefundCom Seller to Buyer
  create offer
  detach violated OfferCom
  discharge refund[, detached RefundCom + 2]
      where "Amount >= Payment"

commitment TransferCom Bank to Buyer
  create instruct
  discharge transfer[, created TransferCom + 2]
        where "Payment=Price"
```

Listing 1 gives a Cupid specification. It first specifies the base events along with their keys and timestamp attributes. The commitment `OfferCom` specifies that *offer creates* a commitment (instance) from SELLER to BUYER. This commitment is *detached* if *transfer* happens within 5 time units (for purposes of this paper, seconds) of the creation and Payment in the *transfer* is at least as much as Price in the *offer*. The commitment *expires* (fails to be detached) if either of these conditions is not met. The commitment is *discharged* if *shipment* happens within 5 time units of being detached. The commitment is *violated* if it fails to be discharged, that is, if *shipment* fails to occur within the stipulated time.

`AcceptCom` specifies that *accept* creates a commitment from BUYER to SELLER that if *shipment* happens within 5 time units of its creation, then *transfer* will occur within 5 time units of its being detached. `RefundCom` specifies that *offer* creates a commitment from SELLER to BUYER if `OfferCom` is violated, then *Refund* of at least the amount paid will be done with 2 time units of the violation (else, obviously, the `RefundCom` will be violated). `Refund` demonstrates the use of nested commitments, which may be used to capture patterns such as compensation. `TransferCom` captures the bank’s commitment to the buyer to do *transfer* upon *instruct*.

Table 1 defines the formal syntax of Cupid. Below, $\mathcal{A}$ and $\mathcal{T}$ are the sets of agent names and time instants, respectively; in particular, $\mathcal{T} = \mathbb{N} \cup \{\infty\}$, where $\mathbb{N}$ is the set of natural numbers and ∞ is an infinitely distant time instant.

Listing 1 uses a surface syntax for readability. We write and, or, and except for $\sqcap$, $\sqcup$, and $\ominus$ respectively. In time intervals, we omit lower and upper instants when they are 0 and ∞, respectively. An omitted detach clause means the commitment is unconditional. We label commitments to simplify referring to commitment events.

Table 1: Syntax of Cupid [13].

Event	$\longrightarrow$	Base | LifeEvent
LifeEvent	$\longrightarrow$	created($\mathcal{A}$, $\mathcal{A}$, Expr, Expr, Expr) |
		detached($\mathcal{A}$, $\mathcal{A}$, Expr, Expr, Expr) |
		discharged($\mathcal{A}$, $\mathcal{A}$, Expr, Expr, Expr) |
		expired($\mathcal{A}$, $\mathcal{A}$, Expr, Expr, Expr) |
		violated($\mathcal{A}$, $\mathcal{A}$, Expr, Expr, Expr)
Expr	$\longrightarrow$	Event[Time, Time] | Expr $\sqcap$Expr | Expr $\sqcup$Expr |
		Expr $\ominus$ Expr| Expr where φ
Time	$\longrightarrow$	Event + $\mathcal{T}$ | $\mathcal{T}$
ComSpec	$\longrightarrow$	commitment($\mathcal{A}$, $\mathcal{A}$, Expr, Expr, Expr)

Cupid specifies five *life events* for every commitment: *created*, *detached*, *expired*, *discharged*, and *violated*. The semantics of Cupid gives a query for each life event for a commitment. The idea is to infer the life events (including their timestamps) from the base events. Time intervals for an event ([Time, Time] in Table 1) are interpreted strictly: the event is required to occur after (including at) the initial moment but before the final moment of the interval.

In [13], Cupid’s semantics is given in relation algebra; its existing implementation compiles each life event of a commitment into an SQL query. Azorus provides a new implementation of Cupid into Jason to enable BDI programming using commitments.

2.2 Specifying Protocols

Information protocols are *declarative* interaction specifications [27, 28]. In this approach, an interaction is specified as a composition of protocols—messages being the special case of atomic protocols—in terms of the the information dependencies between them. The idea is that an agent can emit any message whose information dependencies are satisfied given its *local state*, that is, its communication history. We adopt information protocols because they support flexible and asynchronous multiparty enactments better than traditional message ordering-oriented representations of protocols [11].

We explain the main ideas via Listing 2, which gives an information protocol named *Ebusiness*. It specifies several messages, along with senders, receivers, and their information parameters. The parameter Id is annotated key, meaning it serves to identify enactments (and correlate messages). Adornments ⌜in⌝, ⌜out⌝, and ⌜nil⌝ for parameters capture information dependencies and are interpreted relative to enactments. A message in some enactment is *viable* (legal) for purposes of emission if it has bindings for all parameters except those adorned ⌜nil⌝; the agent already knows the bindings of all the ⌜in⌝ parameters (that is, the bindings must exist in the agent's local state); and it does not know the bindings for any ⌜out⌝ parameter or ⌜nil⌝ parameters (they must not exist in the local state). Sending the message adds it to the agent's local state (along with the bindings for the ⌜out⌝ parameters, thus making them known). Receiving a message adds it to the receiver's local state (along with the bindings for all its parameters, thus making them known). Notably, information protocols do not specify message reception order.

Listing 2: An information protocol.

```
Ebusiness {
  roles Buyer, Seller, Bank
  parameters out Id key, out Item, out Price, out
        Status

  Seller -> Buyer: offer[out Id key, out Item,
        out Price]
  Buyer  -> Seller: accept[in Id key, in Item, in
        Price, out Decision]
  Buyer  -> Bank: instruct[in Id key, in Price,
        out Details]
  Bank   -> Seller: transfer[in Id key, in Price,
        in Details, out Payment]
  Seller -> Buyer: shipment[in Id key, in Item,
        in Price, out Status]
  Seller -> Bank:  refund[in Id key, in Item, in
        Payment, out Amount, out Status]
}
```

Thus, in any enactment of *Ebusiness*, SELLER may send *offer* anytime since all its parameters are ⌜out⌝. Once SELLER has sent *offer*, it would know the bindings for Id, Item, and Price, which means it may send *shipment* provided it does not already know the binding for Status. By analogous reasoning, BUYER may send *accept* or *instruct* anytime after receiving *offer*; BANK may send a *transfer* anytime after receiving *instruct*; and SELLER may send *refund* anytime after sending *offer* and receiving *transfer*. Messages *shipment* and *refund* are mutually exclusive since they both bind Status (it is ⌜out⌝ in both).

To get a sense of how flexible *Ebusiness* is, consider the fact that it has 658 distinct maximal enactments (each a causally valid permutation of sends and receives of its messages extended until no agent is left with any viable message), including the enactment depicted in Figure 1, which is notable because *accept* and *transfer* are "reordered" in the communication infrastructure and SELLER sends *shipment* even though it has not received *accept*.

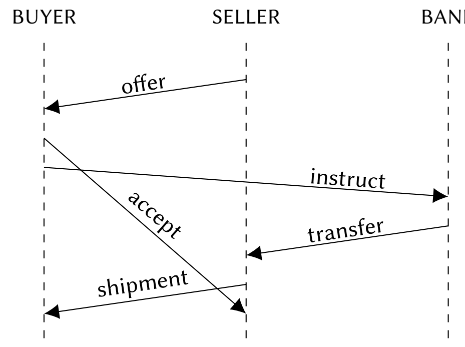

Figure 1: *Ebusiness* enactment in which *shipment* is sent by SELLER even as *accept* was in transit, based on [11, p. 1380].

2.3 Implementing Agents in Jason

Jason is an extended implementation of the AgentSpeak logic-programming language for specifying agent behavior [7]. In Jason, an agent is modeled as having *beliefs*, which capture the state of the world; *goals*, which capture its objectives; and *plans*, which are methods for realizing its goals. To facilitate building multiagent systems, Jason adopts communication primitives based on the Knowledge Query and Manipulation Language, better known as KQML [8].

To illustrate Jason's programming model, especially, how it weaves together communication and reasoning in an agent, Listing 3 give a snippet of how an agent Bob who plays SELLER in *Ebusiness* might be implemented in Jason without any special support for protocols.

Listing 3: Jason snippet of a SELLER agent Bob.

```
buyer(alice).
in_stock(figs).
goes_for(figs, 10).

!start.
+!start <-
        ?buyer(Buyer);
        ?goes_for(Item, Price);
        .random(Id);
        .send(Buyer, tell, offer(Id, Item, Price)).

+accept(Id, Item, Price, Decision)[source(Sender)]
```

```
: in_stock(Item) & buyer(Sender)
<- .send(Sender, tell, shipment(Id, Item, Price,
    yum)).
```

The first few lines of Listing 3 add beliefs that buyer is `alice`, `figs` are in stock and that they go for the price of `10`. Then the goal *start* is asserted. The following lines show two plans. The first is for the goal *start* and is executed whenever it is asserted. This plan executes two queries to bind variables Buyer and Item and Price, respectively. It then uses a library function to bind variable Id to a random identifier. Finally, it uses the built-in function for sending an *offer* to Buyer using the KQML speech act *tell*.

The following plan is for handling a received *accept*, which is recorded as a belief when it is received by Bob and is executed whenever such a belief is added. The plan checks (via guards) that the Item is in stock and that the Sender is the buyer and if so sends back a *shipment* message, again using a *tell*.

3 Programming Model, Architecturally

Figure 2 describes the Azorus architecture and programming model. A MAS is specified in terms of commitments and an information protocol. To implement an agent to play a role in the MAS, a programmer must supply the requisite internal logic. The Azorus tooling generates an adapter (comprising the red-bordered components) for the role being played by the agent based on the specifications. These components maintain the agent's local state (the protocol state projected to the messages sent or received by the agent) as a set of beliefs and provide primitives for commitment reasoning. The agent's internal logic uses the Azorus adapter to reason about its commitments and perform communicative acts.

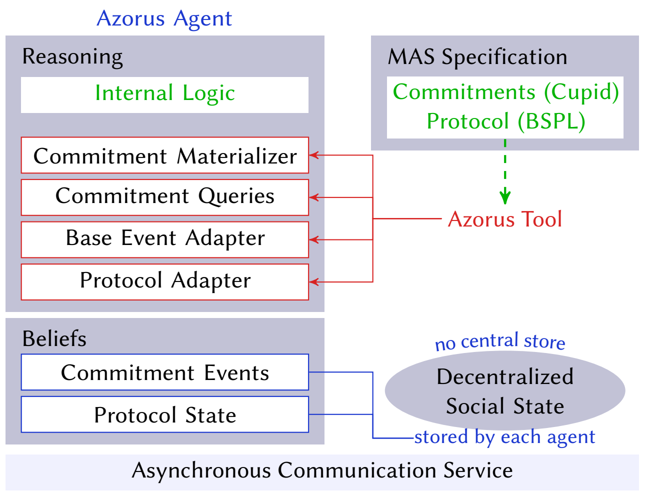

Figure 2: Azorus architecture and programming model.

In the figure, state is represented in blue and computational components in red. Each agent sits atop an *Asynchronous Communication Service* which it uses to send and receive message and has the following components. *Local State* is a set of beliefs corresponding to the messages sent and received by the agent. The *Protocol Adapter* is a representation of protocol corresponding to the role played by the agent in the protocol. It relies upon *Local State* to compute the set of *enabled* communicative acts (explained shortly). As messages are added to the *Local State*, the *Base Event Adapter* adds *Base Events* as timestamped beliefs. *Commitment Queries* are computed on top of *Base Events*. The queries may be used in an agent's *Internal Logic*, as illustrated later.

Materializing the commitment events would accommodate a reactive programming style where *Internal Logic* is expressed as plans that respond to their occurrence. *Commitment Event Materializer* serves precisely this purpose. For every update of *Local State*, it runs the *Commitment Queries* to figure out the commitment events resulting from the update and asserts them as beliefs.

The value of Azorus arises from generating the *Protocol Adapter*, *Base Event Adapter*, *Commitment Queries*, and *Materializer* from commitments and protocols and packaging them as the Azorus adapter. Specifically, the agent programmer may focus on writing the *Internal Logic* based on the interface afforded by Azorus adapter: local state (the communicative acts that have occurred), enabled acts (the acts that may be performed), and commitment queries and materialized commitment events (as capturing meaning).

Below we describe each computational component, including how they update the stateful ones.

3.1 Protocol Adapter

Baldoni *et al.* [3] present a programming model for implementing protocol-based Jason agents. Given an information protocol, the Jason+BSPL protocol adapter enables the implementation of Jason agents that play roles in the protocol. Specifically, an agent's protocol adapter maintains its local state. Based on the state and the protocol specification, it keeps track of information-enabled *forms*. The forms are necessarily partial message instances that would be legal to send if completed. Specifically, a form's ⌜in⌝ parameters have bindings from the local state, whereas the ⌜out⌝ parameters are unbound because their bindings don't exist in the local state; ⌜nil⌝ parameters are omitted from the form because they are neither bound in the local state nor can be bound.

Listing 4 gives a possible local state for a SELLER agent and Listing 5 shows the forms available to it in that state.

Listing 4: A possible local state for a SELLER agent. It contains instances of messages in the *Ebusiness* protocol.

```
offer(1, fig, 10)
offer(2, jam, 100)
accept(2, jam, 100, yes)
transfer(1, fig, done, 10)
```

Listing 5: Enabled forms, showing parameters to be bound.

```
offer(Id, Item, Price)
shipment(1, fig, 10, Status)
shipment(2, jam, 100, Status)
```

To write a Jason+BSPL agent, a programmer writes a set of *plans*. Each plan is an event-triggered piece of code that gets some *enabled* forms; *completes* them via some logic; and then *attempts* to send them. If the attempt passes the required integrity checks, the adapter turns the completed forms into messages on the wire and records them in the local state.

Listing 6 shows a Jason code snippet that represents a SELLER agent's internal reasoning. The first plan concerns communicating *offers*. If there is an *enabled offer* form, then it *completes* the form by checking if it has something to offer, and then *attempts* to send it. The listing also contains a plan for completing and attempting *shipment* forms. The *enabled* predicate and *attempt* are adapter abstractions. The programmer uses them and also writes the plan for completing the form. Notably, the programmer never writes code to receive messages.

Listing 6: Some Jason+BSPL snippets.

```
@offer_plan[atomic]
+!send_offer
  :  enabled(offer(out, out, out)[receiver(out)])
  <- !complete(offer(Id, Item,
       Price)[receiver(Buyer)]);
       !attempt(offer(Id, Item,
           Price)[receiver(Buyer)]).

@shipment_plan[atomic]
+!send_shipment(Id, Item, Price, Buyer)
  :  enabled(shipment(Id, Item, Price,
       out)[receiver(Buyer)])
  <- !complete(shipment(Id, Item, Price,
       Status)[receiver(Buyer)]);
     !attempt(shipment(Id, Item, Price,
         Status)[receiver(Buyer)]).

+!complete(offer(Id, Item,
     Price)[receiver(Buyer)])
  :  on_offer(Id, Item, Price) & buyer(Buyer)
  <- -on_offer(Id, Item, Price).

+!complete(shipment(Id, Item, Price,
     Status)[receiver(Buyer)])
  :  in_stock(Item) & condition(Status) &
     buyer(Buyer)
  <- -in_stock(Item).
```

Jason+BSPL abstracts away the maintenance of the local state and presents an interface to the programmer that supplies the enabled communicative acts. However, it does not support meaning-based reasoning—the programmer must encode when messages should be sent using low-level reasoning.

3.2 High-Level Commitment Queries

To support commitment queries, we give abstract Jason rules of the form *head* :- *body*. The rules are substantially more modular than in [13], which facilitates comprehension and enhances confidence that they capture intuitions correctly.

We treat all expressions of type Expr in Table 1, e.g., $X \sqcap Y$, $X \sqcup Y$, and so on, uniformly as events. $[\![X]\!]$ refers to the predicate for event X. For a base event E with attributes $\vec{a}$ and timestamp t, $[\![E]\!]$ is simply $E(\vec{a}, t)$ and its instances are asserted beliefs. For example, the predicate for *offer* is offer(Seller, Buyer, Id, Item, Price, Otime). The rules below lift $[\![]\!]$ to all events.

Below, E, F, and G are either base or commitment life events; L is a life event; more generally, X and Y are events; $\vec{a}_X$ and t_X refer to the attributes and timestamp of X, respectively; t_p stands for a globally unique timestamp name in every application of the rules in which it appears. $[\![X]\!]^{\vec{a}}_{t}$ means that $[\![X]\!]$'s attributes and timestamp are $\vec{a}$ and t, respectively. Where obvious from the rule, we omit them.

C_1 says that an instance of $[\![E[c,\infty]]\!]$ is an instance of of E that has occurred at or after c. C_2 is similar.

C_1 $[\![E[c,\infty]]\!]$:- $[\![E]\!]$ & $c \leqslant t_E$.

C_2 $[\![E[0,d]]\!]$:- $[\![E]\!]$ & $t_E < d$.

A compiler uses the abstract Jason to produce actual Jason. Thus, for example, when the compiler encounters the expression offer$[0,5]$, it will map it to a unique name such as offerPred1 and spit out the Jason rule in Listing 7.

Listing 7: Compiler-generated Jason from applying C_1.

```
offerPred1(Seller, Buyer, Id, Item, Price, Otime)
    :- offer(Seller, Buyer, Id, Item, Price,
    Otime) & 5 <= Otime .
```

C_3 says that an instance of $X \sqcap Y$ represents correlated instances of X and Y and whose timestamp value is the max of their timestamps. Further, the set of attributes of the instance is the union of the attributes in the X and the Y instances.

C_3 $[\![X \sqcap Y]\!]^{\vec{a}_X \cup \vec{a}_Y}_{t_p}$:- $[\![X]\!]$ & $[\![Y]\!]$ & $.max([t_X, t_Y], t_p)$.

Suppose the compiler encountered the expression offer[0,5] $\sqcap$ accept[0,6]. Listing 8 gives the kind of actual Jason code generated.

Listing 8: Compiler-generated Jason from applying C_3.

```
andPred3(Seller, Buyer, Id, Item, Price, T1) :-
  //offerPred1 as described in Listing 7
  offerPred1(Seller,Buyer,Id,Item,Price,Otime) &
  //Assume a rule for accept[0,6] from applying C1
  acceptPred2(Seller,Buyer,Id,Item,Price,Atime) &
  .max([Otime,Atime],T1).
```

C_6 says that an instance of $E[F+c,\infty]$ is an instance of E that has occurred no earlier than c time units after the correlated F instance. C_7 says that an instance $E[0, G+d]$ is an instance of E such that if the correlated G instance has occurred, then the E should have occurred before d units after the G's occurrence. The rest of the rules in C_4–C_9 are straightforward applications of C_3.

C_4 $[\![E[c,d]]\!]$:- $[\![E[c,\infty] \sqcap E[0,d]]\!]$.

C_5 $[\![E[F+c,\infty]]\!]^{\vec{a}_E}_{t_E}$:- $[\![E]\!]$ & $[\![F]\!]$ & $t_F + c \leqslant t_E$.

C_6 $[\![E[F+c,d]]\!]$:- $[\![E[F+c,\infty] \sqcap E[0,d]]\!]$.

C_7 $[\![E[0,G+d]]\!]^{\vec{a}_E}_{t_E}$:- $[\![E]\!]$ & (*not* $[\![G]\!]$ | ($[\![G]\!]$ & $t_E < t_G + d$)).

C_8 $[\![E[c,G+d]]\!]$:- $[\![E[c,\infty] \sqcap E[0,G+d]]\!]$.

C_9 $[\![E[F+c,G+d]]\!]$:- $[\![E[F+c,\infty] \sqcap E[0,G+d]]\!]$.

C_{10} says that an instance of $X \sqcup Y$ is either an X instance or a Y instance. If correlated X and Y instances have both occurred, then the timestamp is the min of the two. The set of attributes of the $X \sqcup Y$ instance is the intersection of the attributes of the X instance and the Y instance. Taking the intersection guarantees the absence of unbound attributes. C_{11} is straightforward.

C_{10} $\llbracket X \sqcup Y \rrbracket^{\vec{a}_X \cap \vec{a}_Y}_{t_p}$:- $(\llbracket X \rrbracket \ \&\ \llbracket Y \rrbracket \ \&\ .min([t_X, t_Y], t_p) \mid$
$(\llbracket X \rrbracket \ \&\ not\ Y \ \&\ t_p = t_X) \mid$
$(\llbracket Y \rrbracket \ \&\ not\ X \ \&\ t_p = t_Y)$.

C_{11} $\llbracket X$ where $\varphi \rrbracket$:- $\llbracket X \rrbracket \ \&\ \varphi$.

Let commitment(x, y, c, r, u) represent a commitment specification with debtor x, creditor y, and create, detach, and discharge expressions c, r, and u, respectively. For brevity, in the rules below, we write commitment(c, r, u) instead of commitment(x, y, c, r, u) since the debtor and creditor are the same throughout.

C_{12}–C_{14} give the rules for some of the commitment life events of interest. For commitment(c, r, u), the created instances are the c instances; detached instances represent correlated created and r instances; and discharged instances represent correlated created and u instances.

C_{12} $\llbracket created(c, r, u) \rrbracket$:- $\llbracket c \rrbracket$.

C_{13} $\llbracket detached(c, r, u) \rrbracket$:- $\llbracket created(c, r, u) \sqcap r \rrbracket$.

C_{14} $\llbracket discharged(c, r, u) \rrbracket$:- $\llbracket created(c, r, u) \sqcap u \rrbracket$.

Jason rules for computing expired and violated instances of commitments require the notion of failed events. C_{15} says that an instance of E fails to occur at or after c if it occurs before c. C_{16} says that an instance of E fails to occur before d either if it occurs at or after d or it does not occur at all. In both cases, the timestamp of failure is d. C_{20} says that an instance of E fails to occur before $t_G + d$ if either E occurs at or after $t_G + d$ or E does not occur at all. In both cases, the timestamp of failure is $t_G + d$. The rest of the rules in C_{15}–C_{22} are straightforward.

C_{15} $\llbracket \overline{E[c, \infty]} \rrbracket$:- $\llbracket E[0, c] \rrbracket$.

C_{16} $\llbracket \overline{E[0, d]} \rrbracket_{t_p}$:- $\llbracket E[d, \infty] \rrbracket \mid not\ \llbracket E \rrbracket) \ \&\ t_p = d$.

C_{17} $\llbracket \overline{E[c, d]} \rrbracket$:- $\llbracket \overline{E[c, \infty]} \sqcup \overline{E[0, d]} \rrbracket$.

C_{18} $\llbracket \overline{E[F + c, \infty]} \rrbracket$:- $\llbracket E[0, F + c] \rrbracket$.

C_{19} $\llbracket \overline{E[F + c, d]} \rrbracket$:- $\llbracket \overline{E[F + c, \infty]} \sqcup \overline{E[0, d]} \rrbracket$.

C_{20} $\llbracket \overline{E[0, G + d]} \rrbracket^{\vec{a}_E}_{t_p}$:- $\llbracket G \rrbracket \ \&\ (\llbracket E[G + d, \infty] \rrbracket \mid not\ \llbracket E \rrbracket) \ \&$
$t_p = t_G + d$.

C_{21} $\llbracket \overline{E[c, G + d]} \rrbracket$:- $\llbracket \overline{E[c, \infty]} \sqcup \overline{E[0, G + d]} \rrbracket$.

C_{22} $\llbracket \overline{E[F + c, G + d]} \rrbracket$:- $\llbracket \overline{E[F + c, \infty]} \sqcup \overline{E[0, G + d]} \rrbracket$.

C_{23}—C_{25} extend failure to some more expressions following De Morgan's laws.

C_{23} $\llbracket \overline{X \sqcap Y} \rrbracket$:- $\llbracket \overline{X} \sqcup \overline{Y} \rrbracket$.

C_{24} $\llbracket \overline{X \sqcup Y} \rrbracket$:- $\llbracket \overline{X} \sqcap \overline{Y} \rrbracket$.

C_{25} $\llbracket \overline{X \text{ where } \varphi} \rrbracket$:- $\llbracket \overline{X} \sqcup (X$ where $not\ \varphi) \rrbracket$.

C_{26} says that an instance of $X \ominus Y$ is an instance of X such that the correlated Y has failed to occur. Its timestamp is the max of the two.

C_{26} $\llbracket X \ominus Y \rrbracket^{\vec{a}_X}_{t_p}$:- $\llbracket X \rrbracket \ \&\ \llbracket \overline{Y} \rrbracket \ \&\ .max([t_X, t_{\overline{Y}}], t_p)$.

C_{27} $\llbracket \overline{X \ominus Y} \rrbracket$:- $\llbracket \overline{X} \sqcup Y \rrbracket$.

C_{28}–C_{29} give the rules of computing expired and violated instances. An expired instance is one that has failed to detach; a violated instance is one that has failed to discharge.

C_{28} $\llbracket expired(c, r, u) \rrbracket$:- $\llbracket created(c, r, u) \ominus r \rrbracket$.

C_{29} $\llbracket violated(c, r, u) \rrbracket$:- $\llbracket detached(c, r, u) \ominus u \rrbracket$.

Often, we are interested in life events that have actually occurred, that is, their timestamp is no later than the current time, as C_{30} captures.

C_{30} $\llbracket nowL \rrbracket$:- $\llbracket L \rrbracket \ \&\ t_L \leqslant Now \ \&\ system\_time(Now)$.

3.3 Base Event Adapter

Base event schemas correspond to but may be different from message schemas. For example, in Listing 2, message *accept* has a parameter decision whereas in Listing 1, the corresponding base event schema has no such attribute.

The difference arises from the idea that Cupid specifications concern purely meaning whereas information protocols concern both meaning and coordination [29]. Specifically, every base event schema corresponds to some message schema; however, the message schema may feature additional parameters whose purpose is to enable or disable the occurrence of other messages.

Moreover, each base event schema has an additional timestamp attribute. Every time a message is sent or received, an instance of the corresponding base event schema (if one exists) is asserted where the value of its timestamp is the current system time. C_{31} gives the corresponding rule pattern, whose instance the tooling generates for every base event schema, corresponding message pair $(b(\vec{a}, t), m(\vec{p}))$.

3.4 Commitment Event Materializer

To materialize commitment events as beliefs, we assert an *update* commitment events goal every time an agent asserts a base event (as described above). Any base event affects commitments that are relevant to some subset of enactments, as identified by the bindings of the key attributes. Therefore, for efficiency, the update goal is parameterized by key attributes that are common to the base event schemas and are therefore guaranteed to occur in every life event predicate. C_{31} triggers the update ($\vec{k} \subseteq \vec{a}$).

C_{31} $+m(\vec{p})$: $system\_time(Now)$ <- $+b(\vec{a}, Now)$; $!update(\vec{k})$.

C_{32} gives the abstract Jason plan for materializing commitment events; $\llbracket bel\_nowL \rrbracket$ is a predicate with the same attributes and timestamp as $\llbracket nowL \rrbracket$. The plan for the update goal consists of asserting a belief corresponding to a life event if it is an instance of the life event predicate but not yet asserted. Assume that the life event predicates are $\llbracket L_1 \rrbracket, \ldots, \llbracket L_n \rrbracket$.

C_{32} $+!update(\vec{k})$ <- if $(\llbracket nowL_1 \rrbracket \ \&\ not\ \llbracket bel\_nowL_1 \rrbracket)$
$\{+\llbracket bel\_nowL_1 \rrbracket; \}$
$\ldots$
if $(\llbracket nowL_n \rrbracket \ \&\ not\ \llbracket bel\_nowL_n \rrbracket)$
$\{+\llbracket bel\_nowL_n \rrbracket; \}$.

As explained above, $\vec{k} \subseteq \vec{a}$ for every $\llbracket L_i \rrbracket^{\vec{a}}$.

4 Implementing Flexible Agents

We now give examples of how Azorus agents can reason about commitments to flexibly enact protocols.

4.1 With Commitments as Queries

Azorus offers a set of queries for each commitment as a module (see Figure 2). These queries can be used for driving the choices of the enabled messages computed by the protocol adapter module.

Listing 9: Commitments as queries in Azorus.

```
+!handle_form([shipment(Id, Item, Price,
    out)[receiver(Buyer)]|_])
 :  in_stock(Item) &
     now_detached_OfferCom(Seller, Buyer, Id,
          Item, Price, Bank, Payment, Timestamp)
 <- !send_shipment(Id, Item, Price, Buyer).

+!handle_form([shipment(Id, Item, Price,
    out)[receiver(Buyer)]|_])
 :  not in_stock(Item) &
     now_detached_OfferCom(Seller, Buyer, Id,
          Item, Price, Bank, Payment, Timestamp)
 <- !send_refund(Id, Item, Payment, Bank).

+!handle_form([refund(Id, Item, Payment, out,
    out)[receiver(Bank)]|_])
 : now_detached_RefundCom(Seller, Buyer, Id,
     Item, Price, Bank, Payment, Timestamp)
 <- !send_refund(Id, Item, Payment, Bank).
```

A common reasoning pattern is to discharge a commitment if it is detached. The first plan in Listing 9 embodies this pattern. The seller executes the goal send_shipment if the Item is in stock and the commitment `OfferCom` is detached, that is, the *shipment* occurs if the *transfer* has been done in a timely manner.

Otherwise, by the second plan, if the Item is not in stock but `OfferCom` is detached, the goal send_refund is executed. The plan for send_shipment is as in Listing 6 and the plan for send_refund is analogous. The last plan is for when the commitment `OfferCom` is violated (because shipping does not occur by the deadline); again, the goal send_refund is executed. Both plans intend *refund*; however, the second does it simply on the basis the detach of `OfferCom` whereas the last plan does it upon the violation of `OfferCom`.

4.2 With Commitments as Events

Besides the set of queries for each commitment, an agent program can exploit the commitment event materializer. The commitment event materializer module (see Figure 2) produces an event for each commitment state change in the form of a belief adding event. These events can be exploited to support reasoning.

Listing 10: Commitments as events in Azorus.

```
+!offer : on_offer(Id, Item, Price)
 <- !send_offer.
+ev_now_detached_OfferCom(Seller, Buyer, Id,
     Item, Price, Bank, Payment, Timestamp)
 :  in_stock(Item)
 <- !send_shipment(Id, Item, Price, Buyer).
+ev_now_detached_RefundCom(Seller, Buyer, Id,
     Item, Price, Bank, Payment, Timestamp)
  <- !send_refund(Id, Item, Payment, Bank).
```

For example, in Listing 10, the agent SELLER sends an offer to a potential BUYER. Upon a timely *transfer*, the commitment `OfferCom` is detached and, by exploiting the rule C_{32}, the event +ev_now_detached_OfferCom is produced by adding the corresponding belief to the SELLER agent's belief base. This triggers the plan for dealing with such an event: the agent performs the *shipment*. Analogously, in the case the event +ev_now_detached_RefundCom is generated (the *shipment* does not occur within the deadline) the agent performs the *refund*.

4.3 Timestamp-Based Reasoning

Recall that for a life event L, an instance of $[[nowL]]$ is an $[[L]]$ instance that has actually occurred (that is, with current time as the reference point). In general, any time instant, in the past or the future, could be the point of reference.

Suppose the SELLER agent, as a matter of managing its commitments, wanted to discharge the `OfferCom` commitments that will be violated within 10 time units from now (unless, of course, *shipment* is sent). Listing 11 shows how to accomplish this using a future time instant as the point of reference.

Listing 11: Deadline-based reasoning.

```
+!handle_form([shipment(Id, Item, Price,
    out)[receiver(Buyer)]|_])
 :  in_stock(Item) & violated_OfferCom(Id,..., T)
    & system_time(Now) & T <= Now + 10
 <- !send_shipment(Id, Item, Price, Buyer).
```

5 Conceptual Evaluation

Let's summarize what must be manually specified or coded and what Azorus provides as abstractions. The commitment specification, the protocol, and an agent's internal reasoning must be manually specified. Azorus supports the coding of internal reasoning by providing abstractions that enable reasoning about commitments and performing communicative acts that are legal from the standpoint of the protocol.

In virtually any multiparty application, commitments and protocols are domain objects; there is no avoiding reasoning about them. Specifying them cleanly opens up the possibility of building a tool-supported methodology around them, including verification [15, 30, 31, 36] and programming abstractions (as we do in Azorus), and other productivity tools such as IDEs. Not specifying them means architects and programmers must figure out the possible enactments and encode the reasoning using low-level abstractions. Naturally, such code is likely to be ad hoc, complex, and error-prone even for simple MAS involving rigid interactions between two parties, let alone more than two party-MAS with flexible engagements (such as the *Ebusiness* protocol, which, recall, has 658 enactments).

If Jason had just protocol support (as Jason+BSPL) provides, the programmer would still have to encode reasoning about commitments manually. Consider Listing 12, which shows a SELLER's code snippet. It says that the agent sends an enabled *shipment* if *transfer* has occurred. Since *transfer* is required for the detach of `OfferCom`, this seems to capture the intent behind the first plan in Listing 9. It does not though because it misses the time-related reasoning.

Listing 12: No support for commitment reasoning can lead to errors by underspecification.

```
+!handle_form([shipment(Id, Item, Price,
    out)[receiver(Buyer)]|_])
  : in_stock(Item) &
    transfer(Id, Price, _, Payment)
  <- !send_shipment(Id, Item, Price, Buyer).
```

With just commitment support, things could go wrong. Listing 13 gives an example that allows both *shipment* and *refund*, mutually exclusive acts, to occur if the plan for reacting to *transfer* takes so long to execute that shipment happens too late.

Listing 13: No support for protocols can lead to erroneous communication.

```
+transfer(Id, Price, Payment)
  : in_stock(Item) &
     now_detached_OfferCom(Seller, Buyer, Id,
          Item, Price, Bank, Payment, Timestamp)
  <- .send(Buyer, tell, shipment(Id, Item, Price,
     yum)).

+now_detached_RefundCom(Seller, Buyer, Id, Item,
    Price, Bank, Payment, Timestamp)
  : Amount=Payment
  <- .send(Bank, tell, refund(Id, Item, Payment,
     Amount, done)).
```

Without protocol support, in Jason, programmers typically end up using the *tell* for sending every message. We might as well just drop KQML support (and FIPA ACL [17] support from JADE) and instead consider the protocol messages themselves as first-class communicative acts and express their meaning via social abstractions such as commitments (see Singh's essay in [10]), as Azorus does.

6 Discussion

Azorus' novelty is twofold. One, it shows how protocols as operational abstractions and commitments as high-level abstractions can be leveraged in a multiagent programming model. Two, it extends Jason, a popular BDI-based programming model with higher-level communication abstractions. Azorus exploits practical, expressive languages for commitments and protocols and the Azorus adapter is the first careful working out of the interplay between protocol enactment and commitment reasoning. Its significance is also two-fold. One, Azorus simplifies the engineering of flexible, decentralized MAS. Two, it brings goals, commitments, and protocols—all of which represent autonomy—in a single programming model.

We now discuss some issues that require further investigation.

Specifying Commitments. Different commitment specifications could be overlaid on the same protocol. Listing 14 shows an alternative to the specification in Listing 1 (assume the same base event schemas and we omit the alternative `Alt-RefundCom`). The specification in Listing 1 is "direct" in that it gives the meaning of both *offer* and *accept* as an exchange of *shipment* and *transfer*. Assuming the buyer trusts the seller to discharge its commitments, the expected enactment would be *offer* followed by *transfer* and then *shipment*. By contrast, the specification in Listing 14 has a "waterfall" flavor. Under the same assumption, the expected enactment would be *offer* followed by *accept*, then *shipment*, and then *transfer*.

Listing 14: Alternative commitment specification.

```
commitment Alt-OfferCom Seller to Buyer
  create offer
  detach accept[, created OfferCom + 5]
        where "Payment>=Price"
  discharge shipment [, detached OfferCom + 5]

commitment Alt-AcceptCom Buyer to Seller
  create accept
  detach shipment[, created AcceptCom + 5]
  discharge transfer[, detached AcceptCom + 5]
        where "Payment>=Price"
```

The possibility of several alternative commitment specifications motivates characterizing the specifications in terms of properties and stakeholder requirements that they satisfy. More generally, we need methodologies for deriving commitment specifications from requirements.

Implementing Agents. Consider buyer and seller agents implemented such that the seller waited for the buyer to detach `OfferCom` by effecting *transfer* and the buyer waited for the seller to detach `AcceptCom` by doing *shipment*. Naturally, in every enactment, the agents end up deadlocked (even though the *Ebusiness* protocol itself is live). It is tempting to take the view that the commitment specification in Listing 1 lends itself to deadlocks. However, notice that deadlocks can happen even with the alternative specification in Listing 14 if neither agent is prepared to detach first. Often, a deadlocked enactment is a result of agents exercising their autonomy by not sending messages.

Some deadlocks may be unintentional, resulting from a narrow reading of commitments and failing to take into other factors such as trust (which is crucial to progress in interactions) and other business requirements. For example, if a buyer trusts the seller or the monetary amount involved is small or the item involved is urgent, the buyer may be willing to detach `OfferCom` from the seller, effectively moving first in the exchange. What we need are novel methodologies for implementing agents that take into account the various contextual assumptions and business requirements.

There has been work on studying commitments from the requirements perspective and the ideas are potentially relevant for both specifying commitment and implementing agents. Marengo et al. [23] and Günay et al. [21] study commitments from the notions of safety and control. Some work has studied relationships between goals (as representation of requirements) and commitments [12, 24, 32]. Studying commitments from the point of view of concession (taking risk by moving first) [38] would also be also interesting.

To summarize, there has been limited work on methodologies for building flexible, decentralized MAS; it is a direction that should yield rich dividends.

Supplementary Material. Contains our tooling and the Ebusiness MAS code.

References

[1] Matteo Baldoni, Cristina Baroglio, Federico Capuzzimati, and Roberto Micalizio. 2018. Commitment-based Agent Interaction in JaCaMo+. *Fundamenta Informaticae* 159, 1-2 (2018), 1–33. https://doi.org/10.3233/FI-2018-1656

[2] Matteo Baldoni, Cristina Baroglio, Alberto Martelli, and Viviana Patti. 2006. A Priori Conformance Verification for Guaranteeing Interoperability in Open Environments. In *Proceedings of the 4th International Conference on Service-Oriented Computing (ICSOC) (Lecture Notes in Computer Science, Vol. 4294)*. Springer, Chicago, 339–351. https://doi.org/10.1007/11948148_28

[3] Matteo Baldoni, Samuel H. Christie V, Amit K. Chopra, and Munindar P. Singh. 2024. Jason+BSPL: Including Communication Protocols in Jason. https://drive.google.com/file/d/18cFBUUykuxdPZ3NB1NM4gzU9JODJnmey/view Agent Toolkits Special Track at the 21st European Conference on Multi-Agent Systems.

[4] Bernhard Bauer, Jörg P. Müller, and James Odell. 2000. An Extension of UML by Protocols for Multiagent Interaction an existing Multi-Agent Planning System. In *Proceedings of the 4th International Conference on Multiagent Systems (ICMAS)*. IEEE Computer Society, Boston, 207–214. https://doi.org/10.1109/ICMAS.2000.858455

[5] Fabio Bellifemine, Giovanni Caire, and Dominic Greenwood. 2007. *Developing Multi-Agent Systems with JADE*. Wiley, Chichester, UK. https://doi.org/10.1002/9780470058411

[6] Olivier Boissier, Rafael H. Bordini, Jomi Fred Hübner, Alessandro Ricci, and Andrea Santi. 2013. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming* 78, 6 (June 2013), 747–761. https://doi.org/10.1016/j.scico.2011.10.004

[7] Rafael H. Bordini and Jomi Fred Hübner. 2010. 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)*. IOS Press, Lisbon, 635–640. https://doi.org/10.3233/978-1-60750-606-5-635

[8] Hans Chalupsky, Tim Finin, Rich Fritzson, Don McKay, Stu Shapiro, and Gio Wiederhold. 1992. *An Overview of KQML: A Knowledge Query and Manipulation Language*. TR. University of Maryland Computer Science Department, Baltimore.

[9] Federico Chesani, Paola Mello, Marco Montali, and Paolo Torroni. 2013. Representing and Monitoring Social Commitments using the Event Calculus. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 27, 1 (July 2013), 85–130. https://doi.org/10.1007/s10458-012-9202-0

[10] Amit K. Chopra, Alexander Artikis, Jamal Bentahar, Marco Colombetti, Frank Dignum, Nicoletta Fornara, Andrew J. I. Jones, Munindar P. Singh, and Pınar Yolum. 2013. Research directions in agent communication. *ACM Transactions on Intelligent Systems and Technologies* 4, 2 (2013), 20:1–20:23.

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

[12] Amit K. Chopra, Fabiano Dalpiaz, F. Başak Aydemir, Paolo Giorgini, John Mylopoulos, and Munindar P. Singh. 2014. Protos: Foundations for Engineering Innovative Sociotechnical Systems. In *Proceedings of the 22nd IEEE International Requirements Engineering Conference (RE)*. IEEE Computer Society, Karlskrona, Sweden, 53–62. https://doi.org/10.1109/RE.2014.6912247

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

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

[15] Mohamed El Menshawy, Jamal Bentahar, Hongyang Qu, and Rachida Dssouli. 2011. On the Verification of Social Commitments and Time. In *Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Taipei, 483–490.

[16] Angelo Ferrando, Michael Winikoff, Stephen Cranefield, Frank Dignum, and Viviana Mascardi. 2019. On Enactability of Agent Interaction Protocols: Towards a Unified Approach. In *Proceedings of the 7th International Workshop on Engineering Multi-Agent Systems (EMAS) (Lecture Notes in Computer Science, Vol. 12058)*. Springer, Montréal, 43–64. https://doi.org/10.1007/978-3-030-51417-4_3

[17] FIPA. 2002. FIPA Agent Communication Language Specifications. FIPA: The Foundation for Intelligent Physical Agents, http://www.fipa.org/repository/aclspecs.html.

[18] FIPA. 2003. FIPA Interaction Protocol Specifications. http://www.fipa.org/repository/ips.html FIPA: The Foundation for Intelligent Physical Agents. Accessed 2024-08-11.

[19] Nicoletta Fornara and Marco Colombetti. 2002. Operational Specification of a Commitment-Based Agent Communication Language. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. ACM Press, Melbourne, 535–542. https://doi.org/10.1145/544862.544868

[20] Stéphane Galland, Sebastian Rodriguez, and Nicolas Gaud. 2020. Run-time Environment for the SARL Agent-Programming Language: The Example of the Janus platform. *Future Generation Computer Systems* 107 (June 2020), 1105–1115. https://doi.org/10.1016/j.future.2017.10.020

[21] Akın Günay, Michael Winikoff, and Pınar Yolum. 2015. Dynamically Generated Commitment Protocols in Open Systems. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 29, 2 (March 2015), 192–229. https://doi.org/10.1007/s10458-014-9251-7

[22] Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. 2007. Developing Organised Multiagent Systems using the MOISE$^+$ Model: Programming Issues at the System and Agent Levels. *International Journal of Agent-Oriented Software Engineering* 1, 3/4 (2007), 370–395. https://doi.org/10.1504/IJAOSE.2007.016266

[23] Elisa Marengo, Matteo Baldoni, Amit K. Chopra, Cristina Baroglio, Viviana Patti, and Munindar P. Singh. 2011. Commitments with Regulations: Reasoning about Safety and Control in Regula. In *Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Taipei, 467–474. https://doi.org/10.5555/2031678.2031684

[24] Felipe Meneguzzi, Mauricio C. Magnaguagno, Munindar P. Singh, Pankaj R. Telang, and Neil Yorke-Smith. 2018. GoCo: Planning Expressive Commitment Protocols. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 32, 4 (July 2018), 459–502. https://doi.org/10.1007/s10458-018-9385-0

[25] Jaime Simão Sichman, Rosaria Conte, Yves Demazeau, and Cristiano Castelfranchi. 1994. A Social Reasoning Mechanism Based on Dependence Networks. In *Proceedings of the 11th European Conference on Artificial Intelligence*. John Wiley and Sons, Amsterdam, 188–192.

[26] Munindar P. Singh. 1998. Agent Communication Languages: Rethinking the Principles. *IEEE Computer* 31, 12 (Dec. 1998), 40–47. https://doi.org/10.1109/2.735849

[27] Munindar P. Singh. 2011. Information-Driven Interaction-Oriented Programming: BSPL, the Blindingly Simple Protocol Language. In *Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Taipei, 491–498. https://doi.org/10.5555/2031678.2031687

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

[29] Munindar P. Singh. 2014. Bliss: Specifying Declarative Service Protocols. In *Proceedings of the 11th IEEE International Conference on Services Computing (SCC)*. IEEE Computer Society, Anchorage, Alaska, 235–242. https://doi.org/10.1109/SCC.2014.39

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

[31] Pankaj R. Telang and Munindar P. Singh. 2012. Specifying and Verifying Cross-Organizational Business Models: An Agent-Oriented Approach. *IEEE Transactions on Services Computing* 5, 3 (July 2012), 305–318. Appendix pages 1–5.

[32] Pankaj R. Telang, Munindar P. Singh, and Neil Yorke-Smith. 2019. A Coupled Operational Semantics for Goals and Commitments. *Journal of Artificial Intelligence Research (JAIR)* 65 (May 2019), 31–85. https://doi.org/10.1613/jair.1.11494

[33] Michael Winikoff. 2007. Implementing Commitment-Based Interactions. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Honolulu, 868–875. https://doi.org/10.1145/1329125.1329283

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

[35] Michael Winikoff, Nitin Yadav, and Lin Padgham. 2018. A New Hierarchical Agent Protocol Notation. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 32, 1 (Jan. 2018), 59–133. https://doi.org/10.1007/s10458-017-9373-9

[36] Pınar Yolum. 2007. Design Time Analysis of Multiagent Protocols. *Data and Knowledge Engineering Journal* 63, 1 (2007), 137–154.

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

[38] Pınar Yolum and Munindar P. Singh. 2007. Enacting Protocols by Commitment Concession. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Honolulu, 116–123. https://doi.org/10.1145/1329125.1329158