Clouseau: Generating Communication Protocols from Commitments

1798

Abstract

Engineering a decentralized multiagent system (MAS) requires realizing interactions modeled as a communication protocol between autonomous agents. We contribute Clouseau, an approach that takes a commitment-based specification of an interaction and generates a communication protocol amenable to decentralized enactment. We show that the generated protocol is (1) *correct*—realizes all and only the computations the realize the input specification; (2) *safe*—ensures the agents' local views remain consistent; and (3) *live*—ensures the agents can proceed to completion.

1 Introduction

Any application where stakeholders collaborate or compete while retaining their autonomy, such as in finance, healthcare, and the Internet of Things, may be naturally understood as a decentralized multiagent system (MAS). Engineering a MAS presupposes a means to ensure that the agents interoperate even though their internal representations and decision making are hidden from each other.

This paper unifies two core aspects of interoperation. One, **operational** or how the agents realize their interactions. A communication *protocol* specifies what messages an agent may send or receive, and when (Bauer, Müller, and Odell 2000). To reduce coupling, interactions must be asynchronous, i.e., no agent blocks for another except to receive essential information. Recent MAS platforms (Boissier et al. 2019) and protocol approaches (Ferrando et al. 2019; Singh 2011) support asynchrony. Two, **meanings** or the social import of an interaction. We must represent the meanings formally and independently of agent construction, such that the agents can reason about meanings of interactions and agree on the outcomes upon observing the same events.

Social commitments (Fornara and Colombetti 2003; Marengo et al. 2011; Chesani et al. 2013) provide high-level meanings to messages, yielding an operations-independent standard of correctness. For example, Alice (a merchant), may commit to Bob (a customer) to send Bob some goods if he pays. Now if Bob pays, the commitment is discharged only if Alice sends the goods.

Whereas a protocol offers a clear operational interface for agents, commitments capture the meanings of interactions. Current MAS approaches separately specify meanings and protocols, which (1) creates a burden of *maintaining* consistency of two specifications and (2) leads to *inflexible* protocols because manually operationalizing meanings under decentralization is nontrivial.

Clouseau, our approach, provides a new way to operationalize commitments by compiling meanings into flexible protocols that can enacted by agents using asynchronous messaging over unordered channels. This is a significant advance over existing approaches, which lack such a capability. In particular, Clouseau produces correct and flexible protocols, thereby avoiding the above limitations.

Contribution: Operationalizing Commitments How can we automatically generate a communication protocol based on commitments, thereby bridging the gap between meanings and protocols that benefit from asynchronous messaging? A desirable protocol would (1) enable computing commitment states; (2) ensure alignment of the agents with respect to the commitments; (3) be flexible, i.e., allow as many interactions as possible given the commitments and the capabilities of the agents.

Accordingly, we contribute a specification comprising (1) a domain information model; (2) the roles to be played by the interacting agents; (3) commitments between these roles in terms of the information model; (4) the capabilities of the roles; and (5) description of satisfactory completion (an event expression). This input provides the essential knowledge for capturing meanings computationally. This language augments Cupid (Chopra and Singh 2015a) with agent capabilities, which help capture how an agent may enact a protocol. Crucially, we reformulate the semantics of Cupid to a decentralized model.

We adopt as *output* information-based protocols (Singh 2011) that express causality and integrity properties in information. An output protocol captures all legitimate decentralized enactments, thereby yielding flexible agent interaction.

Our overarching contribution is theoretical: a method to generate a protocol from a commitment specification. We show that a generated protocol is (1) correct, that is, it supports exactly the computations of the specification, (2) safe, in the sense that local views of agents remain consistent despite decentralization and asynchronous enactment, and (3) live, ensuring that agents can proceed to completion.

2 NetBill and Commitments

We adopt NetBill (Sirbu 1997), a protocol for trading digital media, as a familiar running example (Yolum and Singh 2002a; Winikoff, Liu, and Harland 2005). Figure 1 shows NetBill as a finite state machine (FSM) whose transitions are labeled with messages (SENDER, RECEIVER: content).

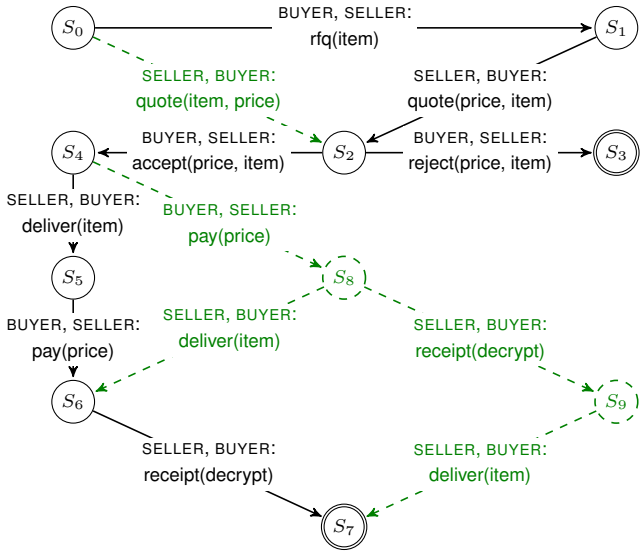

Figure 1: Finite state machine representations of NetBill. The solid lines describe the original NetBill. The dashed lines show some possible generalizations that follow trivially from commitments.

NetBill begins with an rfq (request for quotes) message from BUYER to SELLER for an item (transition from s_0 to s_1), who responds to BUYER with a quote including the price of item (transition from s_1 to s_2). If BUYER sends a reject in response (transition from s_2 to s_3), the protocol ends, indicated by the double circle on s_3. Otherwise, if BUYER sends an accept (transition from s_2 to s_4), SELLER sends a deliver to BUYER, including item in an encrypted format (transition from s_4 to s_5). After receiving deliver, BUYER sends a payment (i.e., an electronic cheque) to SELLER (transition from s_5 to s_6). After receiving payment, SELLER sends a receipt to BUYER including cipher to decrypt the item (transition from s_6 to s_7), and the protocol ends.

3 Specifying Interactions using Clouseau

NetBill includes only two (i.e., accept and reject) branches. We could enhance its flexibility by including, e.g., the dashed edges in Figure 1 whereby SELLER may send an unsolicited quote and BUYER may pay in advance.

But how would we establish that any path, original or added, is legitimate? The idea behind using commitments (Yolum and Singh 2002b) is to express the meaning of an interaction formally and thereby to ensure that all and only the legitimate interactions are realized. Given commitments, paths where the commitments are not violated can be included in the protocol without fear of violating a business requirement—Clouseau supports more general completion requirements. A major benefit is raising the level of abstraction from operations to meanings.

Clouseau provides a way to generate a protocol that retains the flexibility of a meaning-level specification thus obviating the need for manual design of a protocol.

3.1 Syntax of Clouseau Specifications

A Clouseau specification comprises (1) a set of roles; (2) an information schema of base events over lists of attributes; (3) commitments between the roles over the events; (4) each role's capabilities, i.e., which events a role can bring about and attribute bindings it can set; and (5) completion. Cupid (Chopra and Singh 2015a) has (2) and (3); Clouseau adds (1), (4), and (5).

Table 1 shows Clouseau's surface syntax. Here, $\longrightarrow$ indicates a production, and | indicates choice. A $+$ indicates one or more repetitions. $\lfloor\ \rfloor$ indicate optionality. $\mathscr{A}$, $\mathscr{B}$, $\mathscr{C}$, $\mathscr{R}$, and $\mathscr{S}$ are sets of (terminal) attributes, base events, commitments, roles, and specification names. $\mathcal{T} = \mathbb{N} \cup \infty$ is a set of time instances. ([ ] are terminals that identify time intervals.) We write $\sqcap$, $\sqcup$, and $\ominus$ for and, or, and except, respectively.

Table 1: Syntax of Clouseau's input specification.

Spec	$\longrightarrow$	$\mathscr{S}$ {roles $\mathscr{R}\mathscr{R}^+$ Base$^+$ Commit$^+$ Cap$^+$ Comp}
Base	$\longrightarrow$	event $\mathscr{B}(\lfloor\mathscr{A}\lfloor$opt$\rfloor\rfloor^+)$ key $\mathscr{A}^+$
Commit	$\longrightarrow$	commitment $\mathscr{C}$ $\mathscr{R}$ to $\mathscr{R}$ create Ev $\lfloor$detach Ev$\rfloor$ discharge Ev $\lfloor$release Ev$\rfloor$
Cap	$\longrightarrow$	capability $\mathscr{R}$ $\mathscr{B}$ $\lfloor$with $\lfloor\lfloor$fresh$\rfloor$ $\mathscr{A}\rfloor^+\rfloor$
Comp	$\longrightarrow$	completion Ev
Ev	$\longrightarrow$	$\mathscr{B}\lfloor[\lfloor$Time, $\rfloor$ Time$]\rfloor$ | Life $\mathscr{C}\lfloor[\lfloor$Time, $\rfloor$ Time$]\rfloor$ | Ev and Ev | Ev or Ev | Ev except Ev
Time	$\longrightarrow$	$\mathcal{T}$ | $\mathscr{B} + \mathcal{T}$
Life	$\longrightarrow$	created | detached | discharged | released | expired | violated| done

Listing 1 specifies NetBill based on Yolum and Singh (2002b). Lines 4–10 show base event schemas, each a relation over its attributes. Some attributes form a key. A timestamp attribute is implicit. For example, rfq has attributes nID (its key) and item. Each instance of rfq is a tuple of values for its attributes that is unique for its value of nID.

Listing 1: Specification of NetBill in Clouseau.

```
1  NetBill-Flexible {
2   roles Seller, Buyer
3
4   event rfq(nID, item) key nID
5   event quote(nID, item, price) key nID
6   event accept(nID, item, price, decision) key nID
7   event reject(nID, item, price, decision) key nID
8   event payment(nID, price, decision, cheque) key nID
9   event deliver(nID, decision, item) key nID
10  event receipt(nID, item, cheque, cipher) key nID
11
12  commitment PromiseGoods Seller to Buyer
13   create quote
14   detach created AcceptQuote
15   discharge deliver[accept + 3]
16   release reject
17
18  commitment PromiseReceipt Seller to Buyer
19   create quote
20   detach created AcceptQuote and payment[accept + 3]
21   discharge receipt[payment + 1]
```

```
22   release reject
23
24  commitment AcceptQuote Buyer to Seller
25   create accept[quote + 5]
26   detach deliver[accept + 3]
27   discharge payment[accept + 5]
28
29  capability Seller quote with nID, item, price
30  capability Seller deliver
31  capability Seller receipt with cipher
32  capability Buyer rfq with nID, item
33  capability Buyer accept with fresh decision
34  capability Buyer reject with fresh decision
35  capability Buyer payment with cheque
36
37  completion done PromiseGoods and done PromiseReceipt and
    done AcceptGoods
38 }
```

Only a role who has the *capability* for a base event (along with any its attributes) can instantiate it. Line 29 states that SELLER can bring about quote and may produce bindings for nID, item, and price—indicated by with—that respect quote's key. Lines 33–34 state that BUYER must bind a fresh value—indicated by fresh—for decision in accept and reject. Therefore, since accept and reject have the same key, nID, at most one of them can occur for any nID value.

A commitment involves four event expressions: create and discharge are required but detach and release are optional. AcceptQuote begins on Line 24: BUYER (its debtor) commits to SELLER (its creditor) upon accept (its create event) that if deliver occurs within three time units of accept (its detach event) then payment occurs within five time units of accept (its discharge event). A created commitment expires when its detach fails to occur; is violated if it is detached but fails to discharge; and released (frees the debtor) if its release event (shown for PromiseReceipt) occurs.

An event expression (Ev in Table 1) is either a base event or a lifecycle event restricted by time constraints, or a complex event built up from simpler events using and, or, except. Lifecycle events refer to a commitment's being created, detached, discharged, expired, violated, or released (the last four mean the commitment is terminated). The form done abbreviates created except detached. A commitment's lifecycle events are derived entirely from its specification.

Line 37 means that NetBill completes when for each of its three commitments, any instance that is created terminates.

Definition 1 states Clouseau's deep syntax.

Definition 1. *A* specification S*, is a tuple* $\langle R, B, A, C, Q\rangle$*, where* R *is a set of two or more roles;* B *is a nonempty set of base events;* A *is a set of capabilities over* R *and* B*;* C *is a set of commitments over* R *and* B*;* Q *is a completion event.*

A base event*,* e *or* $e(\vec{a}, \vec{\kappa}, \vec{q})$*, associates an event named* e *with sets of attributes* $\vec{a} \subseteq \mathcal{A}$*, key attributes* $\vec{\kappa} \subseteq \vec{a}$*, and optional attributes* $\vec{q} \subseteq (\vec{a} \setminus \vec{\kappa})$*.*

$\mathsf{C}(x, y, c, r, u, l)$ *is a* commitment *from debtor* $x \in R$ *to creditor* $y \in R$ *with create, detach, discharge, and release events* c*,* r*,* u*, and* l*, respectively.*

$\mathsf{A}(x, e, \vec{w}, \vec{n})$ *denotes a* capability *of role* $x \in R$ *for base event* $e(\vec{a}, \vec{\kappa}, \vec{q}) \in B$*. It means that role* x *can bring about instances of* e *by supplying values for* e*'s nonoptional attributes. The attributes* $\vec{w}$ *are those for which* r may *generate a value if not already known. The attributes* $\vec{n}$ *(*$\vec{n} \subseteq \vec{w}$*) are those for which* r must *produce a fresh value (meaning those attributes cannot be already known).*

3.2 Semantics of Clouseau Specifications

Although we adopt Cupid's syntax, we reformulate its semantics based on roles. This section provides a synchronous model of observations by roles. Section 4 provides an asynchronous model based on messages. Section 5 shows how to generate a asynchronous model (embodied in a protocol) from the synchronous model. (Below, V is the domain of values for all attributes.)

Definition 2 states that a base event's *instance* provides bindings for its attributes and a timestamp. An *observation* associates an instance with observers, including one *doer*.

Definition 2. *An* instance*,* (e, b, t)*, of event* $e(\vec{a}, \vec{\kappa}, \vec{q})$ *associates a partial function* $b\colon \vec{a} \to V$ *with* $\vec{a} \setminus \vec{q} \subseteq dom(b)$ *(binds all nonoptional attributes of* e*), and a timestamp* t*.*

An observation, $\mathsf{O}(x, \vec{y}, e, b, t)$*, associates an instance* (e, b, t) *with observer roles* $\vec{y}$*, where* $|\vec{y}| \geq 2$*, and a distinguished* doer $x \in \vec{y}$ *who brings about the instance.*

Below, $b(a)$ is the result of evaluating b on a and $b(\vec{\kappa})$ is a vector mapping b over each element of $\vec{\kappa}$.

For brevity, we omit the obvious components of $\mathsf{O}(x, \vec{y}, e, b, t)$ and abbreviate it as o, o_i, and so on.

Definition 3 states that in a run for role r, (1) r makes each observation and (2) the sequence respects timestamp order.

Definition 3. *A* run *for* r*,* τ^r*, is a sequence of observations* $o_1 o_2 \ldots$*, where* $\forall o_i, o_j\colon$ *(1)* $r \in \vec{y_i}$ *and (2)* $i < j$ *iff* $t_i < t_j$*.*

At most one instance occurs at a time: if $\mathsf{O}(x_i, \vec{y_i}, e_i, b_i, t)$ and $\mathsf{O}(x_j, \vec{y_j}, e_j, b_j, t)$, then and $x_i = x_j$, $\vec{y_i} = \vec{y_j}$, $e_i = e_j$, and $b_i = b_j$. Definition 4 captures synchrony: if a role makes an observation in a run vector, all roles mentioned in that observation also make that observation.

Definition 4. *A* run vector*,* $\Gamma = [\tau^1 \ldots \tau^{|R|}]$*, is one where* $\forall r \in R$*, (1)* τ^r *is a run, and (2)* $\forall o_i \in \tau^r, \forall y \in \vec{y_i}\colon o_i \in \tau^y$*.*

An observation $o \in \Gamma$ *iff it appears in some run in* Γ*.*

Definition 5 states that a run for role r is viable iff for any observation of which r is the doer: (1) r has the capability for the observed event; (2) r previously observes each attribute that is not part of the capability; and (3) r does not previously observe any of e's fresh attributes.

Definition 5. *A run of role* r*,* τ^r*, is* viable *iff* $\forall \mathsf{O}(r, \vec{y}, e, b, t) \in \tau^r$*: (1)* $\exists \mathsf{A}(r, e, \vec{w}, \vec{n}) \in A$*; (2)* $\forall a \in \vec{a} \setminus \vec{w}\colon (\exists o_i\colon t_i < t,\ a \in \vec{a_i},\ \vec{\kappa_i} \subseteq \vec{\kappa},\ b(\vec{\kappa_i}) = b_i(\vec{\kappa_i})$*, and* $b(a) = b_i(a))$*; (3)* $\forall n \in \vec{n}\colon \nexists o_i\colon t_i < t, n \in \vec{a_i}$*, and* $\vec{\kappa_i} \subseteq \vec{\kappa}$*. A run vector is* viable *iff each of its runs is viable.*

$[\![S]\!]$ *denotes the set of viable runs in* S*.*

Consistency A run vector is *consistent* iff any two observations in it respect key integrity, i.e., if they agree on their common key attributes, they agree on all common attributes.

Definition 6. *Let* o_i *and* o_j *be observations. Let* $\vec{\kappa} = \vec{\kappa_i} \cap \vec{\kappa_j}$ *and* $\vec{a} = \vec{a_i} \cap \vec{a_j}$*. Then,* o_i *and* o_j *are* consistent *iff* $b_i(\vec{\kappa}) = b_j(\vec{\kappa}) \Rightarrow b_i(\vec{a}) = b_j(\vec{a})$*.*

A run vector Γ *is* consistent *iff for any roles* i, j, *any* $o_i \in \tau^i$ *and* $o_j \in \tau^j$ *are consistent. Specification* S *is* consistent *iff every run vector in* $[\![S]\!]$ *is consistent.*

Inconsistency may occur only if the agents have overlapping capabilities. Listing 2 modifies *NetBill-Flexible*: deliver includes cheque and SELLER can do deliver with cheque. Upon accept, SELLER and BUYER can bring about deliver and payment, respectively. Since both SELLER and BUYER control cheque, they can set different bindings for it for the same nID, producing an inconsistency.

Listing 2: An inconsistent variant of NetBill (Listing 1).

```
1 NetBill-Inconsistent { //Modifying NetBill-Flexible
2
3  event accept(nID, item, price, decision)
4  event payment(nID, decision, item, price, cheque)
5  event deliver(nID, decision, item, cheque) key nID
6
7  capability Seller deliver with cheque
8  capability Buyer payment with cheque
9 }
```

Overlapping capabilities need not cause inconsistency. As Listing 1 shows, SELLER and BUYER can bind item but at most one of them can do so for the same binding of nID.

In the rest of this paper, all specifications are consistent.

Intensions The intension of an event gives all the run vectors in which instances of the event, including commitment events, occur. We develop this idea below.

Two instances cohere iff they bind their common key attributes to the same values.

Definition 7. *Two instances,* $\langle e_i, b_i, t_i\rangle$ *and* $\langle e_j, b_j, t_j\rangle$, cohere *iff* $b_i(\vec{\kappa}) = b_j(\vec{\kappa})$, *where* $\vec{\kappa} = \vec{\kappa}_i \cup \vec{\kappa}_j$. *Two sets of instances* cohere *iff each pair of their members coheres.*

A *chain* is a set (ordered by time) of instances drawn from one consistent run vector that are (pairwise) coherent and implicitly ordered by timestamp. The *mix* of two chains ignores coherence so the result is not a chain. The *merge* of two chains is their mix restricted by coherence.

Definition 8. *A* chain *in a run vector* Γ *is a set of instances* $\{\langle(e_0, b_0, t_0)(e_1, b_1, t_1)\ldots\rangle\}$ *iff* $\forall i, j\colon (e_i, b_i, t_i) \in \Gamma$ *and* $\langle e_i, b_i, t_i\rangle$ *and* $\langle e_j, b_j, t_j\rangle$ *are coherent.*

The mix *of two sets of chains is their element-wise union:* $\sigma_1 \Cup \sigma_2 = \{u_1 \cup u_2 | u_1 \in \sigma_1, u_2 \in \sigma_2\}$

The merge *of two sets of chains is their mix restricted by coherence:* $\sigma_1 \diamond \sigma_2 = \{u | u \in \sigma_1 \Cup \sigma_2$ *and* u *is coherent*$\}$.

The *certificate* for an event expression is a chain that makes the expression true. Let $\Gamma \in [\![S]\!]$ for specification S. In Definitions D_1–D_{17}, e is a base event, E, F, and G are base or lifecycle events, and X and Y are event expressions. *last* and *ts* give the last instance of a chain and the timestamp of an instance respectively.

D_1. $[\![e]\!]^\Gamma = \{\langle(e, b, t)\rangle \mid \mathsf{O}(x, \vec{y}, e, b, t) \in \Gamma\}$

D_2. $[\![E[c, d]]\!]^\Gamma = \{\sigma | \sigma \in [\![E]\!]^\Gamma$ and $c \leqslant ts(last(\sigma)) < d\}$

D_3. $[\![E[F + c, d]]\!]^\Gamma = \{\sigma | \sigma \in [\![E]\!]^\Gamma \diamond [\![F]\!]^\Gamma$ and $ts(last(F, \sigma)) + c \leqslant ts(last(E, \sigma)) < d\}$

D_4. $[\![E[c, G + d]]\!]^\Gamma = \{\sigma | \sigma \in [\![E]\!]^\Gamma \diamond [\![G]\!]^\Gamma$ and $c \leqslant ts(last(E, \sigma)) < ts(last(G, \sigma)) + d\}$

D_5. $[\![E[F + c, G + d]]\!]^\Gamma = \{\sigma | \sigma \in [\![E[F + c, \infty]]\!]^\Gamma \diamond [\![E[0, G + d]]\!]^\Gamma\}$

D_6. $[\![X \sqcap Y]\!]^\Gamma = [\![X]\!]^\Gamma \diamond [\![Y]\!]^\Gamma$

D_7. $[\![X \sqcup Y]\!]^\Gamma = [\![X]\!]^\Gamma \cup [\![Y]\!]^\Gamma$

D_8. $[\![X \ominus (Y \sqcap Z)]\!]^\Gamma = [\![(X \ominus Y) \sqcup (X \ominus Z)]\!]^\Gamma$

D_9. $[\![X \ominus (Y \sqcup Z)]\!]^\Gamma = [\![(X \ominus Y) \sqcap (X \ominus Z)]\!]^\Gamma$

D_{10}. $[\![X \ominus (Y \ominus Z)]\!]^\Gamma = [\![(X \ominus Y) \sqcup (X \sqcap Z)]\!]^\Gamma$

D_{11}. $[\![X \ominus ET]\!]^\Gamma = ([\![X]\!]^\Gamma \Cup [\![ET]\!]^\Gamma) \setminus ([\![X]\!]^\Gamma \diamond [\![ET]\!]^\Gamma)$ (T is a time expression)

For brevity, let $k = \mathsf{C}(x, y, c, r, u, l)$ be a commitment.

D_{12}. $[\![created(k)]\!]^\Gamma = [\![c]\!]^\Gamma$

D_{13}. $[\![detached(k)]\!]^\Gamma = [\![created(k) \sqcap r]\!]^\Gamma$

D_{14}. $[\![expired(k)]\!]^\Gamma = [\![created(k) \ominus r]\!]^\Gamma$

D_{15}. $[\![discharged(k)]\!]^\Gamma = [\![created(k) \sqcap u]\!]^\Gamma$

D_{16}. $[\![violated(k)]\!]^\Gamma = [\![detached(k) \ominus (u \sqcup l)]\!]^\Gamma$

D_{17}. $[\![released(k)]\!]^\Gamma = [\![created(k) \sqcap l]\!]^\Gamma$

The *intension* of an event X is $\{\Gamma | \Gamma \in [\![S]\!]$ and $[\![X]\!]^\Gamma \neq \emptyset\}$. The *intension* of an specification is the intension of its completion event (Q), computed via the above semantics.

Definition 9. *The intension of specification* S, $[\![S]\!] = [\![Q]\!]$.

4 Communication Protocols

We express protocols in an variant of BSPL (Singh 2011), which specifies protocols via causality and integrity constraints. Our variant includes (1) multicast messages and (2) multiple protocol parameter *lines*, each a list of parameters needed for one alternative completion. Table 2 shows the syntax using $\mathscr{M}$, $\mathscr{R}$, $\mathscr{S}$, and $\mathscr{X}$ as sets of (terminal) messages, roles, protocols, and parameter names, respectively. Listing 3 shows a protocol generated from Listing 1. Section 5 shows how to do so automatically.

Table 2: Syntax of protocols generated by Clouseau.

Prot	$\longrightarrow \mathscr{S}\{\mathsf{roles}\ \mathscr{R}^+ \lfloor\mathsf{parameters}\ \lfloor\mathrm{Param}\ \lfloor\mathsf{key}\rfloor\rfloor^+\rfloor^+\ \mathrm{Msg}^+\}$
Msg	$\longrightarrow \mathscr{R} \mapsto \mathscr{R}^+ \colon \mathscr{M}[\mathrm{Param}^+]$
Param	$\longrightarrow \mathsf{in}\mathscr{X} \mid \mathsf{out}\mathscr{X} \mid \mathsf{nil}\mathscr{X}$

Listing 3: Protocol generated by Clouseau for NetBill. Clouseau adorns all protocol parameters ⌜out⌝—omitted here for readability.

```
1 NetBill-Flexible-Protocol {
2  roles Seller, Buyer
3  parameters nID key, item, price, quoteP
4  parameters nID key, item, price, rfqP, quoteP
5  parameters nID key, item, price, decision, rfqP,
   quoteP, rejectP
6  parameters nID key, item, price, decision, rfqP,
    quoteP, acceptP
7  parameters nID key, item, price, decision, cheque,
    rfqP, quoteP, acceptP, payP
8  parameters nID key, item, price, decision, rfqP,
    quoteP, acceptP, deliverP
```

```
9  parameters nID key, item, price, decision, cheque,
     rfqP, quoteP, acceptP, payP, deliverP
10 parameters nID key, item, price, decision, cheque,
     rfqP, cipher, quoteP, acceptP, payP, receiptP
11 parameters nID key, item, price, decision, cheque,
     rfqP, cipher, quoteP, acceptP, payP, deliverP,
     receiptP
12
13 Buyer ↦ Seller: rfqM[out nID, out item, out rfqP]
14 Seller ↦ Buyer: quoteM[in nID, in item, out price, out
   quoteP]
15 Seller ↦ Buyer: quoteM[out nID, out item, out price,
   out quoteP]
16 Buyer ↦ Seller: acceptM[in nID, in item, in price, out
     decision, out acceptP]
17 Buyer ↦ Seller: rejectM[in nID, in item, in price, out
     decision, out rejectP]
18 Buyer ↦ Seller: payM[in nID, in price, in decision, out
      cheque, out payP]
19 Seller ↦ Buyer: deliverM[in nID, in item, in decision,
     out deliverP]
20 Seller ↦ Buyer: receiptM[in nID, in item, in cheque,
     out cipher, out receiptP]
21 }
```

Each message *morph* has a sender, a receiver, a name, and a set of parameters (subset of the protocol's parameters), some of which form a key. Morphs of the same name have the same sender and receiver but their parameters and parameter adornments can differ. Each parameter has an adornment, which captures the sender's knowledge of a binding of that parameter: ⌜in⌝ means the sender knows it prior to sending; ⌜out⌝ means the sender doesn't know it prior but produces it when sending; ⌜nil⌝ means the sender doesn't know it prior and doesn't produce it. For example, Line 14 shows quoteM with sender SELLER and receiver BUYER; nID is the key (inherited from the protocol); SELLER must have observed nID and item (for the specified key) before sending; must not know price before; but produce price when sending.

The adornments capture causal dependencies. For example, rfqM of Line 13, which produces nID and item, must precede quoteM of Line 14, which uses those parameters. Modeling causality yields flexibility: the constraints are causally essential; every compatible order is correct. Thus, deliverM and payM may be sent in any mutual order.

Message morphs encode alternative enactments. For example, quoteM has two morphs: Line 14 adorns nID and item ⌜in⌝; and Line 15 has only ⌜out⌝ parameters. We assume the message name, λ, includes an identifier to distinguish different morphs of the same name. Below, $W = \{\ulcorner\mathsf{in}\urcorner, \ulcorner\mathsf{out}\urcorner, \ulcorner\mathsf{nil}\urcorner\}$ is the set of adornments.

Definition 10. *A* message *(morph),* $\mathsf{M}(\lambda, x, \vec{y}, \vec{p}, \alpha, \vec{\kappa})$*, comprises a name* λ*, a sender role* x*, a set of receiver roles* $\vec{y}$*, a function* $\alpha\colon \vec{p} \to W$ *assigning an adornment to each parameter, and a list of key parameters* $\vec{\kappa} \subseteq \vec{p}$*.*

A protocol consists of a set of roles, a set of protocol parameter lines each with the same key (subset of parameters), and a set of messages (morphs). A protocol completes if all parameters on any parameter line are bound. We extend BSPL with multiple parameter lines because each line captures one alternative completion of the protocol while avoiding the need for giving a null binding to a parameter.

Definition 11. *A* protocol*,* $\langle \lambda, \vec{x}, \vec{p}, \vec{\pi}, \vec{\kappa}, F\rangle$*, comprises a name* λ*, a list of roles* $\vec{x}$*, a list of parameters* $\vec{p}$*, a set* $\vec{\pi}$ *of parameter lines (each a pair* $[\vec{q}, \alpha]$*, where* $\vec{q} \subseteq \vec{p}$ *and* $\alpha\colon \vec{q} \to W$*), a set of key parameters* $\vec{\kappa} \subseteq \vec{p}$*, and a set of message morphs* F*.*

An instance of a message binds its parameters. A role's history is a sequence of message instances that a role views. We overload b from attribute to parameter bindings.

Definition 12. $\mathsf{N}(\lambda, b)$ *is an* instance *of a message* $\mathsf{M}(\lambda, x, \vec{y}, \vec{p}, \alpha, \vec{\kappa})$ *iff* $\vec{p} = dom(b)$*. The* history *of role* x*,* h^x*, is a sequence of message instances* $m_0 m_1 \ldots$*, emitted or received by* x*.* $\mathsf{N}(\lambda, b)$ *appears in* h^x *when emitted by* x *and, for all* $y \in \vec{y}$*, in* h^y *when received by* y*.*

A message instance m is viable for emission by role x at its history h^x iff (1) the bindings of m's ⌜in⌝ parameters are set earlier in the history and the bindings of m's ⌜out⌝ or ⌜nil⌝ parameters are not set earlier in the history.

Definition 13. *For integer* i*, let* $\mathsf{N}(\lambda_i, b_i)$ *be an instance of message* $\mathsf{M}(\lambda_i, x, \vec{y_i}, \vec{p_i}, \alpha_i, \vec{\kappa_i})$ *that occurs at index* i *in history* h^x*. Then,* $\mathsf{N}(\lambda_j, b_j)$ *is* viable *in* h^x *iff it is a reception, or (1)* $\forall p \in \vec{p}, \alpha(p) = \ulcorner\mathsf{in}\urcorner\colon \exists i < j\colon \kappa_i \subseteq \kappa_j$*,* $p \in dom(\alpha_i)$*, and* $b_j(p) = b_i(p)$*; and (2)* $\forall p \in \vec{p}, \alpha(p) \in \{\ulcorner\mathsf{out}\urcorner, \ulcorner\mathsf{nil}\urcorner\}\colon \nexists i < j\colon \kappa_i \subseteq \kappa_j$ *and* $p \in dom(\alpha_i)$*.*

A history vector H progresses to H', $H \prec H'$, based on a message emission or reception, capturing asynchrony.

Definition 14. $H = [h^1 \ldots h^{|R|}]$ *is* viable *iff (1) all its histories are empty, or (2) it progresses a viable history vector through the emission or reception of a viable message.* $\langle\!\langle P \rangle\!\rangle$ *is the set of viable history vectors for protocol* P*.*

A *safe* protocol guarantees integrity of information, i.e., no parameter may obtain conflicting bindings.

Definition 15. *Messages* $\mathsf{N}(\lambda_i, b_i)$ *and* $\mathsf{N}(\lambda_j, b_j)$ *are* consistent *iff, for* $\vec{\kappa} = \vec{\kappa}_i \cap \vec{\kappa}_j$ *and* $\vec{a} = \vec{a}_i \cap \vec{a}_j$*,* $b_i(\vec{\kappa}) = b_j(\vec{\kappa}) \Rightarrow b_i(\vec{a}) = b_j(\vec{a})$*. A history vector is* safe *iff all pairs of messages in it are consistent. A protocol* P *is* safe *if and only if each vector in* $\langle\!\langle P \rangle\!\rangle$ *is safe.*

Definition 16. *A history vector* H *is complete for parameters* $\vec{p}$ *iff* $(\forall p \in \vec{p}\colon \exists \mathsf{N}(\lambda, b) \in H$ *and* $p \in dom(b))$*. A protocol* P *is live iff* $\forall H \in \langle\!\langle P \rangle\!\rangle\colon \exists H'\colon H \prec H'$ *where* $H' \in \langle\!\langle P \rangle\!\rangle$ *and* H' *is complete for a parameter line* $\vec{p}$ *of* P*.*

5 Generating a Communication Protocol

The correctness of a generated protocol requires that its set of history vectors matches the *intension* (i.e., the set of run vectors) of the specification. Clouseau uses the commitments and capabilities for determining senders and receivers of messages, and capabilities for determining the operational constraints expressed via adornments. We demonstrate this method by generating Listing 3 from Listing 1.

Information integrity requires that (1) if two messages share a parameter, their keys must overlap and (2) instances determined by the same key bindings must be equal.

Definition 17 captures the intuition that when a parameter is ⌜in⌝, its binding in an instance with respect to its key must

be known before the instance is produced. That is, the parameter must be accompanied by sufficient key parameters to be meaningful. The *determinant* of a parameter p, Δ_p, is the intersection of keys of all message morphs in which p appears. A message is well-determined iff when a parameter is $\ulcorner\mathsf{in}\urcorner$ so is each parameter in its determinant.

Definition 17. $\mathsf{M}(\lambda, x, \vec{y}, \vec{p}, \alpha, \vec{\kappa})$ *is well-determined iff* $(\forall p \in \vec{p}\colon \alpha(p) = \ulcorner\mathsf{in}\urcorner \Rightarrow \forall p' \in \Delta_p \Rightarrow \alpha(p') = \ulcorner\mathsf{in}\urcorner)$.

Generating Message Parameters and Adornments Clouseau generates message morphs to capture role choices. Listing 3 captures two paths in Figure 1 via two morphs of quoteM (Lines 14–15). Line 14 adorns nID and item $\ulcorner\mathsf{in}\urcorner$ to capture the interaction in which SELLER receives an rfqM, which binds nID and item, before sending a quote. Conversely, Line 15 adorns nID and item $\ulcorner\mathsf{out}\urcorner$ to capture that SELLER can send quote without receiving rfqM.

Definition 18 lays out the possible adornments of the parameters in a morph.

Definition 18. *Let* $e(\vec{a}, \vec{\kappa}, \vec{q})$ *be an event,* $\chi = \mathsf{A}(x, e, \vec{w}, \vec{n})$ *be a capability of role* x*. Then, message* $\mathsf{M}(\lambda, x, \vec{y}, \vec{p}, \alpha, \vec{\kappa})$ *corresponds to* e *iff* $\vec{a} \subseteq \vec{p}$ *and* $\forall a \in \vec{a}\colon \alpha(a)$ *is as specified in the* $\boldsymbol{\alpha}(a)$ *column of Table 3 in the matching row.*

$\mathcal{M}_\chi$ *is the set of messages corresponding to capability* χ*. For* $S = (R, B, A, C)$*, the set of messages is* $\bigcup_{\chi \in A} \mathcal{M}_\chi$.

#	$a \in \vec{q}$	$a \in \vec{w}$	$a \in \vec{n}$	$\boldsymbol{\alpha}(a)$ (Possible adornments)
1	No	No	No	$\ulcorner\mathsf{in}\urcorner$
2	No	No	Yes	Ill-formed: no output
3	No	Yes	No	$\ulcorner\mathsf{in}\urcorner$, $\ulcorner\mathsf{out}\urcorner$
4	No	Yes	Yes	$\ulcorner\mathsf{out}\urcorner$
5	Yes	No	No	$\ulcorner\mathsf{in}\urcorner$, $\ulcorner\mathsf{nil}\urcorner$
6	Yes	No	Yes	Ill-formed: no output
7	Yes	Yes	No	$\ulcorner\mathsf{in}\urcorner$, $\ulcorner\mathsf{out}\urcorner$, $\ulcorner\mathsf{nil}\urcorner$
8	Yes	Yes	Yes	Ill-formed: no output

Table 3: The adornments for a parameter based on attribute a depend on whether a is optional ($a \in \vec{q}$), whether the sender can produce a ($a \in \vec{w}$), and whether it must produce a ($a \in \vec{n}$). Rows 2 and 6 are ill-formed because a must be produced by sender, but it cannot be; row 8 because a must be produced but is optional.

Generating Message Senders and Receivers Senders are those roles with a capability for an event from which a morph is generated. In Listing 1, SELLER can bring about quote, so it would be a sender for quoteM.

For receivers, we have some choices. The simplest is to multicast the message to everyone: those to whom it matters will read it. However, we can avoid generating superfluous messages through the following criteria.

First, include receivers to ensure commitment alignment (Chopra and Singh 2015b). In Listing 1, quote features in the creates of PromiseGoods and PromiseReceipt, of which SELLER is debtor and BUYER is creditor. Thus, make SELLER the sender and BUYER the receiver of quoteM.

Second, include receivers to enable a stated capability, i.e., if a role needs an attribute binding to exercise a capability. If so, we make that role a receiver of any message arising from an event that includes that attribute.

Importantly, each message includes an $\ulcorner\mathsf{out}\urcorner$ *autonomy parameter* to make the sender's exercising of its autonomy explicit in the information model. Doing so is essential to reflect the social meaning of communications. A suffix "P" indicates such a parameter, e.g., quoteP for quoteM.

Generating Protocol Parameters Clouseau generates parameter lines. The connection with events is enforced through the inclusion of relevant autonomy parameters in a parameter line. In Listing 3, Line 4 includes rfqP and quoteP and omits acceptP: thus, it is one of the complete vectors when PromiseGoods and PromiseReceipt expire because AcceptQuote is not created.

To select the appropriate parameter lines, we begin from the completion event in the specification. We convert it to a quasi disjunctive normal form (DNF) by applying the semantics of Section 3.2 to eliminate the commitment lifecycle events and to remove any $\sqcup$ from within the scope of $\sqcap$ or $\ominus$. Then, we produce one parameter line for each disjunct, including the parameters of messages derived from events that appear positively in the disjunct and excluding the autonomy parameter of any message derived from an event that appears as the second argument of $\ominus$.

Below, $G(S)$ is a protocol generated from specification S.

6 Correctness of Protocol Generation

To establish correctness of a generated protocol, we first define how the run vectors of a specification *correspond to* the history vectors of a protocol.

We give a series of definitions of correspondence, including (1) of an observation of an event instance to a message instance; (2) a run to a history; (3) a run vector to a history vector; and (4) a specification to a protocol.

Definition 19. *Let* $\mathsf{O}(x, \vec{y}, e, b, t)$ *be an observation and* $\mu = \mathsf{N}(\lambda, b_\mu)$ *be an instance of a morph* $\mathsf{M}(\lambda, x_\mu, \vec{y_\mu}, \vec{p_\mu}, \alpha_\mu, \vec{\kappa_\mu})$*. Then,* e corresponds to λ*,* $e \rightsquigarrow \lambda$*, iff* $\lambda = eM$*,* $x = x_\mu$*,* $\vec{y} = \vec{y}_\mu$*, and* $b = b_\mu$.

Run vectors progress in a lockstep manner in their global timestamp order whereas history vectors progress in asynchrony. The message emissions in a role's history anchor the desired correspondence with the events brought about.

Write a role r's run τ as $\mu_1\beta_1\mu_2\ldots$ where each μ_i is a possibly empty list of observations $\mathsf{O}(x, \vec{y}, e, b, t)$, $r \neq x$, and each β_i is an observation $\mathsf{O}(r, \vec{y}, e, b, t)$ whose doer is r.

Write a role r's history h as $\nu_0\delta_0\nu_1\ldots$ where each ν_i is a possibly empty set of message receptions and each δ_i is a single message emission. Then, h is *canonical* for τ iff (1) $|\tau| = |h|$; (2) $\forall \mu_i\colon \mu_i \rightsquigarrow \nu_i$; and (3) $\forall \beta_i\colon \beta_i \rightsquigarrow \delta_i$.

Let $h' = \mu_0\delta_0\mu_1\ldots$. Then, h' *causally permutes* h iff the δ_i are unmoved and each μ_i is a permutation of ν_i.

Definition 20. *A run* τ *and history* h correspond*,* $\tau \rightsquigarrow h$*, iff* $\exists h'$ *where* h' *causally permutes* h*, and* h' *is canonical for* τ.

Run vector Γ *and history vector* H correspond*,* $\Gamma \rightsquigarrow H$*, iff their respective members correspond:* $\forall r \in R\colon \tau^r \rightsquigarrow h^r$.

Definition 21. *Let* S *and* P *be a specification and protocol respectively.* $S \rightsquigarrow P$ *iff* $\forall \Gamma \in [\![S]\!] \exists H \in \langle\!\langle P \rangle\!\rangle\colon \Gamma \rightsquigarrow H$ *and* $\forall H \in \langle\!\langle P \rangle\!\rangle \exists \Gamma \in [\![S]\!]\colon \Gamma \rightsquigarrow H$.

Theorem 1 establishes correctness of protocol generation.

Theorem 1. $S \rightsquigarrow G(S)$.

Proof sketch. First, show by induction on run vectors that for every $\forall \Gamma \in [\![S]\!]\colon \exists H \in \langle\!\langle G(S) \rangle\!\rangle$ such that $\Gamma \rightsquigarrow H$. For the base case, consider a run vector Γ with a single observation $\mathsf{O}(x, \vec{y}, e, b, t)$. This means that $\mathsf{A}(x, e, \vec{w}, \vec{n}) \in A$ for base event $e(\vec{a}, \vec{\kappa}, \vec{q}) \in B$ such that $\vec{a} = \vec{w}$ (all values must be generated in this observation). According to Definition 18 $m = \mathsf{M}(eM, x, \vec{y}, \vec{a}, \alpha, \vec{\kappa})$ where $\forall a \in \vec{a}\colon \alpha(a) = \ulcorner \mathsf{out} \urcorner$ is in $G(S)$. Therefore, there exists a history vector H in which the only message instance sent instantiates m as defined above. Therefore, $\Gamma \rightsquigarrow H$.

For the inductive step, assume for each Γ^i that is a run vector over i observations, $\Gamma^i \rightsquigarrow H^i$. Let $\mathsf{O}(x, \vec{y}, e, b, t)$ be the $(i+1)^{st}$ observation. This means that $\mathsf{A}(x, e, \vec{w}, \vec{n}) \in A$ for $e(\vec{a}, \vec{\kappa}, \vec{q}) \in B$. Let $\vec{a}' \subseteq \vec{a}$ be known from Γ^i. This means that $\vec{a}' \cap \vec{n} = \emptyset$ and $\vec{w} \setminus \vec{a}'$ are the bindings produced in the observation. Let $m' = \mathsf{M}(eM, x, \vec{y}, \vec{p}, \alpha, \vec{\kappa})$ where $\vec{a}' \subseteq \vec{p}$ and $\forall a' \in \vec{a}'\colon \alpha(a') = \ulcorner \mathsf{in} \urcorner$ and $\forall w \in \vec{w} \setminus \vec{a}'\colon \alpha(w) = \ulcorner \mathsf{out} \urcorner$. According to Definition 18, m' is in $G(S)$. Therefore, there exists Γ^{i+1} such that $\Gamma^{i+1} \rightsquigarrow H^{i+1}$.

Now we show the converse, that is, $\forall H \in \langle\!\langle G(S) \rangle\!\rangle\colon \exists \Gamma \in [\![S]\!]$. The argument is by induction on history vectors. The empty history vector corresponds to the empty run vector. Let H be a history vector such that for all strictly shorter H', there exists Γ' such that $\Gamma' \rightsquigarrow H'$. Going from H' to H may have either involved a reception or emission by r of an instance of $\mathsf{M}(eM, x, \vec{y}, \vec{p}, \alpha, \vec{\kappa})$. In either case, there exists capability $\mathsf{A}(x, e, \vec{w}, \vec{n}) \in A$ for base event $e(\vec{a}, \vec{\kappa}, \vec{q}) \in B$, which would guarantee that there exists Γ such that $\Gamma \rightsquigarrow H$.

Theorem 2 relates the consistency of a specification and the safety of a generated protocol.

Theorem 2. *If S is consistent, then protocol $G(S)$ is safe.*

Proof sketch. The empty run vector is in $[\![S]\!]$ and is consistent. The empty run vector corresponds to the empty history vector in $G(S)$, which is safe. Assume S is consistent. Let $\Gamma, \Gamma' \in [\![S]\!]$ such that that Γ' extends Γ by one observation $\mathsf{O}(x, \vec{y}, e, b, t)$ of instance (e, b, t), of event $e(\vec{a}, \vec{\kappa}, \vec{q})$. Further, let the observation be effected by x exercising its capability $\mathsf{A}(x, e, \vec{w}, \vec{n})$. By Theorem 1 $\exists H, H' \in \langle\!\langle P \rangle\!\rangle$ such that $\Gamma \rightsquigarrow H$ and $\Gamma' \rightsquigarrow H'$. Therefore, H' differs from H in having one extra emission—of a message instance m corresponding to observation O. Assume by induction that H is safe. Let $\vec{v} \subseteq \vec{w}$ such that $b(v)$ for each $v \in \vec{v}$ is not bound in Γ but bound in Γ', meaning that O produces the bindings for $\vec{v}$. From $\Gamma \rightsquigarrow H$, we see that no $v \in \vec{v}$ is bound in H. By Definition 18, we know that m is an instance of morph M for A where exactly the $\vec{v}$ are $\ulcorner \mathsf{out} \urcorner$. Hence, H' is safe.

Theorem 3. *If S is consistent, then protocol $G(S)$ is live.*

Proof sketch. Let $\Gamma \in [\![S]\!]$ be a run vector. Then Γ satisfies S's completion event Q. Viewing Q in DNF, therefore, Γ must satisfy at least one disjunct of Q. By construction of protocol $G(S)$, each parameter line of $G(S)$ corresponds to one disjunct of Q as well. Let $H \in \langle\!\langle G(S) \rangle\!\rangle$ such that $\Gamma \rightsquigarrow H$. Since Γ would have bound each attribute in one disjunct of Q, H would bind the corresponding parameter as well as the autonomy parameter for each message, thereby completing the relevant parameter line.

7 Discussion

Our contribution bridges the gap between meaning-based and operational specifications via a method for generating a protocol given a Clouseau specification. Clouseau is higher level than operational languages since it works from meaning abstractions, especially commitments, events, and capabilities. But whereas events in Clouseau are synchronous (occurring simultaneously for all observers), messaging in BSPL is asynchronous (the emission of a message is decoupled from its emission).

What makes Clouseau significant is that a flexible protocol supporting asynchrony is in general difficult to construct and maintain by hand. By modeling the meanings precisely using Clouseau, a designer can avoid that overhead. Winikoff (2007) and Desai and Singh (2008) highlight the challenges of computing commitments in asynchronous settings. Our approach could be adapted to target other protocol languages, e.g., trace expressions (Ferrando et al. 2019) or HAPN (Winikoff, Yadav, and Padgham 2018).

Yolum and Singh (2002a) introduced the idea of computing directly with commitments, leading to enhanced languages and tooling (Winikoff, Liu, and Harland 2005; Chopra and Singh 2006; Baldoni et al. 2014), and work on reasoning patterns (Yolum and Singh 2002b; Fornara and Colombetti 2003; Chopra and Singh 2015a). This body of work formalizes commitment reasoning in the context of a conceptually unitary machine and does not tackle asynchrony. Our contribution builds upon the core idea of these approaches, namely, that messages carry meanings, by supporting decentralized enactments.

Recent work on monitoring commitments (Chesani et al. 2013; Kafalı and Torroni 2018) or norms (Alechina, Dastani, and Logan 2014; Dastani, Torroni, and Yorke-Smith 2018) assumes a unitary model with synchronous state changes. Conceptually, these works are closer to Cupid, which too is based on a unitary model, and which our contribution extends to decentralized settings. El-Menshawy et al. (2018) address the problem of model checking commitments with real-time constraints. Baldoni et al. (2018) give an approach for verifying agents against commitments. Such techniques would be valuable for Clouseau specifications.

MAS programming frameworks, e.g., JaCaMo (Boissier et al. 2019), address complementary concerns to ours. JaCaMo supports asynchronous messaging, and can support agents that enact protocols generated by Clouseau. Boissier et al. (2019) point out that interaction between agents in JaCaMo could be based on direct messaging between them or via shared artifacts. Indeed, Baldoni et al. (2014) show how to program agents using a shared JaCaMo artifact to coordinate commitment-based interactions.

In brief, Clouseau is unique in that it unifies the meaning-based and operational aspects of interaction with respect to asynchronous communication. It can potentially enhance productivity and quality in producing and maintaining a protocol that is maximally flexible given a meaning specification and correct.

References

Alechina, N.; Dastani, M.; and Logan, B. 2014. Norm approximation for imperfect monitors. In *Proceedings of the 13th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, 117–124. Paris: IFAAMAS.

Baldoni, M.; Baroglio, C.; Marengo, E.; Patti, V.; and Capuzzimati, F. 2014. Engineering commitment-based business protocols with the 2CL methodology. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 28(4):519–557.

Baldoni, M.; Baroglio, C.; Capuzzimati, F.; and Micalizio, R. 2018. Type checking for protocol role enactments via commitments. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 32(3):349–386.

Bauer, B.; Müller, J. P.; and Odell, J. 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)*, 207–214. IEEE Computer Society.

Boissier, O.; Bordini, R. H.; Hübner, J. F.; and Ricci, A. 2019. Dimensions in programming multi-agent systems. *Knowledge Engineering Review* 34:e2.

Chesani, F.; Mello, P.; Montali, M.; and Torroni, P. 2013. Representing and monitoring social commitments using the event calculus. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 27(1):85–130.

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

Chopra, A. K., and Singh, M. P. 2015a. Cupid: Commitments in relational algebra. In *Proceedings of the 29th Conference on Artificial Intelligence (AAAI)*, 2052–2059. Austin, Texas: AAAI Press.

Chopra, A. K., and Singh, M. P. 2015b. Generalized commitment alignment. In *Proceedings of the 14th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, 453–461. Istanbul: IFAAMAS.

Dastani, M.; Torroni, P.; and Yorke-Smith, N. 2018. Monitoring norms: A multi-disciplinary perspective. *Knowledge Engineering Review* 33:e25.

Desai, N., and Singh, M. P. 2008. On the enactability of business protocols. In *Proceedings of the 23rd Conference on Artificial Intelligence (AAAI)*, 1126–1131. Chicago: AAAI Press.

El-Menshawy, M.; Bentahar, J.; Kholy, W. E.; and Laarej, A. 2018. Model checking real-time conditional commitment logic using transformation. *Journal of Systems and Software* 138:189–205.

Ferrando, A.; Winikoff, M.; Cranefield, S.; Dignum, F.; and Mascardi, V. 2019. On the enactability of agent interaction protocols: Toward a unified approach. *CoRR* abs/1902.01131v4.

Fornara, N., and Colombetti, M. 2003. Defining interaction protocols using a commitment-based agent communication language. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, 520–527. Melbourne: ACM Press.

Kafalı, Ö., and Torroni, P. 2018. COMODO: Collaborative monitoring of commitment delegations. *Expert Systems with Applications* 105:144–158.

Marengo, E.; Baldoni, M.; Chopra, A. K.; Baroglio, C.; Patti, V.; and Singh, M. P. 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)*, 467–474. Taipei: IFAAMAS.

Singh, M. P. 2011. Information-driven interaction-oriented programming: BSPL, the Blindingly Simple Protocol Language. In *Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, 491–498. Taipei: IFAAMAS.

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

Winikoff, M.; Liu, W.; and Harland, J. 2005. Enhancing commitment machines. In *Proceedings of the 2nd International Workshop on Declarative Agent Languages and Technologies (DALT)*, volume 3476 of *Lecture Notes in Artificial Intelligence*, 198–220. Berlin: Springer.

Winikoff, M.; Yadav, N.; and Padgham, L. 2018. A new Hierarchical Agent Protocol Notation. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 32(1):59–133.

Winikoff, M. 2007. Implementing commitment-based interactions. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, 868–875. Honolulu: IFAAMAS.

Yolum, P., and Singh, M. P. 2002a. Commitment machines. In *Proceedings of the 8th International Workshop on Agent Theories, Architectures, and Languages (ATAL 2001)*, volume 2333 of *Lecture Notes in Artificial Intelligence*, 235–247. Seattle: Springer.

Yolum, P., and Singh, M. P. 2002b. 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)*, 527–534. Bologna: ACM Press.