Kiko: Programming Agents to Enact Interaction Protocols

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

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

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

ABSTRACT

Realizing a multiagent system involves implementing member agents who interact based on a protocol while making decisions in a decentralized manner. Current programming models for agents offer poor abstractions for decision making and fail to adequately bridge an agent's internal decision logic with its public decisions.

We present *Kiko*, a protocol-based programming model for agents. To implement an agent, a programmer writes one or more *decision makers*, each of which chooses from among a set of valid decisions and makes mutually compatible decisions on what messages to send. By completely abstracting away the underlying communication service and by supporting practical decision-making patterns, Kiko enables agent developers to focus on business logic. We provide an operational semantics for Kiko and establish that Kiko agents are protocol compliant and able to realize any protocol enactment.

ACM Reference Format:
Samuel H. Christie V, Munindar P. Singh, and Amit K. Chopra. 2023. Kiko: Programming Agents to Enact Interaction Protocols. In *Proc. of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2023), London, United Kingdom, May 29 – June 2, 2023*, IFAAMAS, 10 pages.

1 INTRODUCTION

Enterprise and other applications, e.g., in business and healthcare, involve interactions between social entities such as humans and organizations [25] based on technical resources such as databases. A sociotechnical system (STS) involves social and technical entities [31, 37] and provides a useful abstraction for such applications. Today, an STS is implemented using a conceptually central service through which its entities interact. In contrast, we address the challenges of implementing a decentralized multiagent system (MAS) to realize an STS. Here, each principal maps to an agent; the agents interact with each other via *asynchronous* messaging.

The messages sent by an agent represent its public *decisions*. For example, a *Quote* by Seller for some item for some price represents a decision by Seller; an *Accept* (of some *Quote*) sent by Buyer represents a decision of Buyer; and so on. To coordinate their decisions, the agents rely on an *interaction protocol*. By specifying the constraints on messaging, a protocol specifies the constraints on decision making between the agents in a MAS. For example, a *Purchase* protocol in the above-introduced e-business setting may specify that the price is offered by the seller, and payment is required for delivery.

A protocol is specified abstractly with reference to roles to be adopted by agents in a multiagent system. Implementing an agent according to a role means fleshing out the role with private (internal) decision logic that results in messages being emitted, that is, decisions being made [16]. For example, suppose agents `Bob` and `Sally` play Buyer and Seller, respectively, in *Purchase*. `Sally`'s decision logic may be to send *Quote*s with lower prices to repeat buyers. `Bob`'s decision logic may be to *Accept* a *Quote* if the price fits within its budget. Such decision logic is the essence of an agent.

Supporting the common desire [28, 39] for programming models that separate business logic from other components—and combating complexity in agent communication [10], in general—proves challenging. Traditional protocol languages [3, 18, 24, 29, 42] specify message ordering, which limits flexibility [11]. JADE [5, 6], a programming model for multiagent systems, is noteworthy for its early support for FIPA protocols [19]; however, the FIPA approach is long outdated [34] and the FIPA protocols are limited to a few patterns of interaction specified in terms of message ordering. Agent-oriented programming models such as Jason [8] and JaCaMo [7] provide cognitive abstractions for encoding an agent's internal reasoning but do not support protocols. Existing commitment-based approaches [22, 41] either rely on centralized commitment stores [2] or do not adequately address operationalizing asynchronous communication [17]; some approaches map the problem to protocols [26, 38]—and hence within the scope of this paper. Traditional agent-oriented methodologies [9, 15, 30] emphasize and incorporate protocols as design abstractions. However, the protocol specifications in these approaches are informal (usually UML interaction diagrams), which rules out protocol-based software abstractions for engineering agents. In a nutshell, today we lack a protocol-based programming model for agents that supports flexible, decentralized decision making via asynchronous messaging.

Our contribution, *Kiko*, addresses this gap. Specifically, Kiko advances a novel decision-oriented programming model that enables structuring and implementing agents based on the protocol roles they play. Kiko's fundamental abstraction is that of a *decision maker*, a construct for capturing the decision logic that selects and makes a set of decisions from those currently available. The agent developer's primary task is to write the set of decision makers.

Kiko guarantees an agent's compliance with the roles its plays. Kiko supports practical decision-making patterns that other approaches cannot easily accommodate, including correlation, cross-enactment reasoning, emission sets, and multiprotocol reasoning. Notably, in providing a decision-based interface for programming agents, Kiko abstracts away the communication service that transports messages between agents. In particular, decision making in Kiko avoids having to deal with the order in which messages are received. Actual message emission is also handled transparently in the programming model.

In addition, we contribute a formalization of the programming model and prove its soundness and completeness with respect

Proc. of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2023), A. Ricci, W. Yeoh, N. Agmon, B. An (eds.), May 29 – June 2, 2023, London, United Kingdom.

to possible protocol enactments. We also present an optimized compliance-checking method and establish its validity.

2 INFORMATION PROTOCOLS INTRODUCED

A protocol-based programming model for agents presumes a language in which to specify protocols. We adopt BSPL [36], a declarative protocol language that eschews the specification of message ordering and instead specifies information constraints.

Listing 1: The *Purchase* protocol.

```
Purchase {
  roles Buyer, Seller
  parameters out ID key, out item, out price, out done

  Buyer -> Seller: RFQ[out ID key, out item]
  Seller -> Buyer: Quote[in ID key, in item, out price]
  Buyer -> Seller: Buy[in ID key, in item, in price, out done]
  Buyer -> Seller: Reject[in ID key, in price, out done]
}
```

An information protocol in BSPL specifies the roles, messages between roles, and information constraints that define which message emissions are valid. Information causality captures information dependencies: what information must or must not be known by an agent playing a role to be able to send a message. Information integrity captures consistency in distributed settings: there cannot be two messages sent with conflicting information in the same protocol enactment. Given the local store of an agent (its history of message observations), an agent can send any message that satisfies the specified causality and integrity constraints.

Listing 1 illustrates the main ideas of information protocols. It specifies a purchase protocol to be enacted by agents playing roles Buyer and Seller. *Purchase* composes message schemas, each with its sender and receiver roles and information parameters. For example, *RFQ* is from Buyer to Seller and its parameters are ID and item. A concrete message instance associates the parameter names with value bindings, e.g., binding ID to a UUID and item to "ball."

To support information integrity, some parameters in a message schema are annotated *key*, e.g., ID in all the messages of Listing 1. A tuple of bindings for the key parameters of a message schema uniquely identifies both an instance of the schema and the enactment to which it belongs, in which all nonkey parameters may have at most one binding. For example, say *RFQ* occurs with bindings [ID: 10, item: ball]. Then, a *Quote* with [ID: 10, item: hat, price: 10] would violate integrity because for the same binding of ID there are different bindings of item. Conversely, *Quote* with [ID: 11, item: hat, price: 10] satisfies integrity despite the different binding for item because it has a different binding of the key ID.

In a message schema, every message parameter is adorned ⌜in⌝, ⌜out⌝, or ⌜nil⌝. Adornments capture information causality constraints for the emission of an instance of a schema; ⌜in⌝ parameters must be known from prior communications (they are causal dependencies); ⌜out⌝ parameters and ⌜nil⌝ parameters must *not* be known, but ⌜out⌝ parameters are bound in the emission. For example, in Listing 1, Seller must know item before it can send *Quote*, and in doing so produces a binding for price.

Knowledge of a parameter exists in the context of some binding for the associated key. After receiving an *RFQ* with bindings [ID: 10, item: ball], Seller knows that in the enactment ID=10 item is bound to ball, and can produce a binding of price by sending *Quote*.

Integrity and causality apply to protocols generally. In *Purchase* in Listing 1, all protocol parameters are adorned ⌜out⌝ in the protocol parameter line, meaning that each enactment of *Purchase* as identified by the ID generates bindings for all of them. Further, the parameter line enables composition with other protocols.

3 THE KIKO PROGRAMMING MODEL

We introduce the architectural basis for the programming model, followed by examples that illustrate its features.

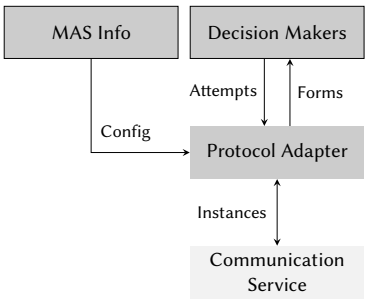

Figure 1: The Kiko agent architecture.

Figure 1 shows the main components of the agent architecture as focused on enacting protocols. The MAS Info and Decision Makers are components provided by the agent programmer (indicated by the border). The Protocol Adapter is a generic component provided by Kiko that understands information protocols and provides an API for plugging in Decision Makers. The adapters of all agents collectively achieve a coordination service and assimilate information received from messages [35]. The Communication Service is anything that provides asynchronous messaging between agents. Our implementation uses UDP, which is unordered and unreliable (lossy).

An information protocol constrains only the *emission* of messages by agents, based on its causal dependencies. This means that ordered delivery, as provided by TCP or a message queue, is not required for correctly enacting a protocol. Further, message reception is idempotent, so messages can be retransmitted to enact a protocol reliably despite message loss [12, 14]. Thus an unordered, lossy transport like UDP is sufficient for enacting BSPL protocols.

MAS Info (Configuration). A protocol specifies a MAS abstractly via reference to roles. A *concrete* MAS for a protocol is identified by a UUID and assigns roles to the agents that will play them. MAS identifiers are essential since an agent may play a role in several MAS. The properties of a (concrete) MAS and the mailboxes of the agents in the MAS are common knowledge to the agents in the MAS. Kiko requires each agent to be configured with such knowledge; Listing 2 gives such a configuration for agent Bob.

Listing 2: Bob's MAS Info Configuration.

```
self = "Bob"
systems = {
  "5feceb66": {
    "protocol": Purchase,
    "roles": {Buyer: self, Seller: "Sally"}}}
agents = {
  self: [("192.168.1.100", 1111)]
  "Sally": [("192.168.1.102", 1111), ("152.1.27.202", 1111)]}
```

In Listing 2, `5feceb66` is an identifier for a MAS that enacts *Purchase* with Bob and Sally as Buyer and Seller, respectively. Bob's and Sally's mailboxes are given as (IP, port) tuples. An agent may have several mailboxes for receiving messages; in Listing 2, Sally has two. Our focus is not on how a MAS is constituted, but on programming abstractions that enable decentralized decision-making. Listing 2 shows the kind of information needed to configure a MAS, and it could be constructed dynamically at runtime.

Formally, we model an agent using a tuple $\langle a, H_a, I_a, O_a \rangle$, where the components are the name of the agent, its history, input channel (its mailbox), and output channel respectively. Channels I_a and O_a are simply sets of message instances being sent and received, respectively, by agent a. Definition 1 defines a MAS.

Definition 1 (MAS). *A multiagent system μ is a tuple $\langle P, A \rangle$, where P is a protocol, and A is a map from roles of P to agents.*

Decision Makers. To write an agent, programmers supply the configuration and write one or more decision makers. A decision maker is invoked upon the occurrence of specified events. When invoked, the adapter supplies it with prototypes of message instances that the agent is enabled to send given the agent's current history of message observations. We refer to these prototypes as *forms*, after documents with fields that need to be filled. A form of a message schema has bindings for the parameters that are adorned ⌜in⌝ in the schema, reflecting that its causal dependencies are satisfied, leaving only the parameters adorned ⌜out⌝ to be bound. The purpose of a decision maker is to flesh out some message instances from the forms by supplying bindings for their ⌜out⌝ parameters; the adapter collects this set of completed instances as an emission *attempt*. The adapter verifies whether the attempt as a whole is consistent with the agent's history and if so, emits the instances in the attempt; else it rejects the attempt.

Suppose Bob's history is empty (it has observed no messages). Then the only form available to Bob is Bob -> Sally: *RFQ*[`5feceb66`, (ID), (item)], with unfilled parameters in parentheses. Since protocol enactments occur within the context of a MAS, each form and any instance produced from it contains a MAS identifier (here, `5feceb66`)—conceptually like the value for an implicit parameter system in every message. Bob's programmer may have written a decision maker that fleshes out the above form into instances such as Bob -> Sally: *RFQ*[`5feceb66`, 1, `bat`] and Bob -> Sally: *RFQ*[`5feceb66`, 2, `ball`] based on some decision logic. These instances are passed on to the adapter for emission. Listing 3 shows a decision maker (in Python) called `start` that is invoked at system initialization, upon `InitEvent`. The argument `enabled` contains the available forms when `start` is invoked and the body of `start` contains code to send two instances of the form, one each for `bat` and `ball`. The instruction to the adapter to emit the instances is implicit—after the decision maker returns, the adapter goes through all forms to see which ones have been fleshed out into instances and emits them (conditional to validation).

Listing 3: Bob's initial decision to send RFQs.

```
@adapter.decision(event=InitEvent)
def start(enabled):
    for item in ["ball", "bat"]:
        ID = str(uuid.uuid4())
        for m in enabled.messages(RFQ):
            m.bind(ID=ID, item=item)
```

Consider another example. Suppose Bob's history contains the above two RFQ instances and Sally -> Bob: *Quote*[`5feceb66`, 1, `bat`, 5]. Then, in addition to the *RFQ* form specified above, the following forms would also be available to Bob: Bob -> Sally: *Buy*[`5feceb66`, 1, `bat`, 5, (done)] and Bob -> Sally: *Reject*[`5feceb66`, 1, `bat`, (done)]. Bob's programmer may have implemented a decision maker (as illustrated in Listing 4) that chooses from one of these two available forms based on how acceptable the price is, fleshes it out by binding done, and instructs the adapter to emit the resulting instance.

Listing 4: A simple Buy or Reject decision maker for Bob.

```
@adapter.decision
def start(enabled):
    for m in enabled.messages(Buy):
        if(m["price"] < 20)
            m.bind(done="cool")
        else
            reject = next(enabled.messages(Reject, ID=m["ID"]))
            reject.bind(done="rejected")
```

We now give an example where a decision maker's emission attempt fails because it erroneously contains incompatible instances. Specifically, Listing 5 is erroneous because Bob creates instances for both *Buy* and *Reject* in the same enactment. This emission attempt fails because *Buy* and *Reject* are mutually exclusive according to Listing 1 (because both bind ⌜out⌝ done); neither will be emitted.

Listing 5: Decision maker attempting to send *Buy* and *Reject*.

```
@adapter.decision
def indecisive(enabled):
  buy = next(enabled.messages(Buy))
  reject = next(enabled.messages(Reject, system=buy.system,
        ID=buy["ID"]))
  buy.bind(done="accepted")
  reject.bind(done="rejected")
```

Listing 5's error brings out a remarkable aspect of Kiko. Kiko enables decision makers (programmers) to choose sets of instances to emit. Whereas each of the instances in the set (e.g., *Buy*) would be individually consistent and compatible with the history when the decision maker was invoked and therefore could be emitted by the adapter, collectively, the set of instances chosen by the decision maker could be internally incompatible (*Buy* and *Reject*) and therefore fail emission by the adapter. By rejecting incompatible emission sets, the adapter guarantees that an agent will not make noncompliant emissions.

An alternative would be to limit a decision maker to work on at most one form at a time. Then, its emission by the adapter would be guaranteed. Such a decision maker is a special case for Kiko.

A specific triggering event may be specified for a decision maker (e.g., `InitEvent` in Listing 3). If such a triggering event is not specified (e.g., as in Listing 4), the adapter automatically invokes the decision maker whenever a communication event occurs. Event-based invocation enables some optimizations: First, the agent need not poll to wait for enough information to make a decision; not polling may be seen as an extension of the pub/sub pattern because a decision can depend on multiple pieces of information from multiple sources. Second, because all constraints are relative to an enactment, and communication events contain keys identifying their enactment, the enactment can be directly looked up, thus avoiding linear scans or joins across an entire database for validation.

However, there are cases where an agent may want to emit messages outside of reacting to a message observation (whether sent or received). For example, if the agent needs to make business decisions only once per day, then waiting and making them all as a batch could be more efficient and accurate. To support a wider variety of behavioral patterns, Kiko uses an internal event queue on which the developer can signal custom events, and decision makers can be registered with custom filters to select which events should trigger them.

We now formalize the concepts introduced in the above section.

An *association* binds values to some subset of the parameters of a message schema.

Definition 2 (Association). *If m is a schema in protocol P, then $\mathcal{M}_m$ is a relation with attributes $payload(m) = \langle \mu, \mathfrak{s}_m, \mathfrak{r}_m, \vec{\imath}_m, \vec{o}_m \rangle$, and $\mathcal{M}$ is the union of all such relations. The parameter name μ refers to a multiagent system. A tuple $\widehat{m}$ is an* association *of schema m if and only if it is a tuple of parameter bindings $\langle b_p | p \in payload(m)] \rangle$ in $\mathcal{M}_m$.*

We use $\widehat{m}[\ldots]$ for projecting parameters to their bindings in the message instance; e.g., $\widehat{m}[\mathfrak{s}_m]$ is the sender of $\widehat{m}$, and $\widehat{m}[\vec{k}_m]$ is the projection of $\widehat{m}$'s key parameters.

A *message instance* is an association where all parameters are bound.

Definition 3 (Message Instance). *An association $\widehat{m} \in \mathcal{M}_m$ is a* message instance *and instance($\widehat{m}$) holds if and only if all of its parameters are bound: $p \in payload(\widehat{m}), \widehat{m}[p] \neq \varnothing$.*

$\mathcal{I} \subset \mathcal{M}$ is the set of all instances.

A *form* is an association where the ⌜out⌝ parameters are unbound.

Definition 4 (Form). *An association $\widehat{m} \in \mathcal{M}_m$ is a* form *(referring to a document with empty fields that need to be filled) if some ⌜out⌝ parameter has a null value. That is, $\forall p \in payload(m) \setminus \vec{o}_m : p \neq \varnothing$ and $\exists p \in \vec{o}_m : \widehat{m}[p] = \varnothing$*

$\mathcal{F} \subset \mathcal{M}$ is the set of all forms.

We introduce the notion of context to capture enactments within a specific MAS.

Definition 5 (Context). *The* context *of an association is its MAS and its keys: $\widehat{m}[\mu, \vec{k}_m]$.*

Associations *share context* if their MAS and any of their keys have the same bindings. A form is enabled when all of its ⌜in⌝ parameter bindings match those from observed instances that share context (consistency), and its ⌜out⌝ and ⌜nil⌝ parameters do not conflict with any observed instances (compatibility), as given by Definitions 6–10.

Definition 6 (Consistent). *Let $M, N \subseteq \mathcal{M}$ be sets of associations; then N is* consistent *with M (and consistent(N, M) holds) if and only if the ⌜in⌝ bindings in N are the same as bindings from associations that share context in M:*

$\forall \widehat{m} \in M, \widehat{n} \in N : \widehat{m}[\mu, \vec{k}_m \cap \vec{k}_n] = \widehat{n}[\mu, \vec{k}_m \cap \vec{k}_n] \implies \widehat{m}[payload(m) \cap \vec{\imath}_n] = \widehat{n}[payload(m) \cap \vec{\imath}_n]$.

Definition 7 (Out-Compatible). *Let $M, N \subseteq \mathcal{M}$ be sets of associations; then N is* out-compatible *with M (and compatible$_{\vec{o}}(N, M)$ holds) if and only if no ⌜out⌝ bindings in N are in payloads of associations that share context in M:*

$\forall \widehat{m} \in M, \widehat{n} \in N : \widehat{m}[\mu, \vec{k}_m \cap \vec{k}_n] = \widehat{n}[\mu, \vec{k}_m \cap \vec{k}_n] \implies payload(m) \cap \vec{o}_{\widehat{n}} = \varnothing$

Definition 8 (Nil-Compatible). *Let $M, N \subseteq \mathcal{M}$ be sets of associations; then N is* nil-compatible *with M (and compatible$_{\vec{n}}(N, M)$ holds) if and only if no ⌜nil⌝ bindings in N are in payloads of associations that share context in M:*

$\forall \widehat{m} \in M, \widehat{n} \in N : \widehat{m}[\mu, \vec{k}_m \cap \vec{k}_n] = \widehat{n}[\mu, \vec{k}_m \cap \vec{k}_n] \implies payload(m) \cap \vec{n}_{\widehat{n}} = \varnothing$

Definition 9 (Derived). *Let H_a be an agent history and $\widehat{m}$ be a form whose sender is a; then $\widehat{m}$ is* derived from H_a *(and derived($\widehat{m}, H_a$) holds) if and only if all of $\widehat{m}$'s ⌜in⌝ parameters are drawn from instances that share context in the history:*

$\forall p \in \vec{\imath}_m, \exists \widehat{n} \in H_a : \widehat{n}[\mu, \vec{k}_m \cap \vec{k}_n] = \widehat{m}[\mu, \vec{k}_m \cap \vec{k}_n] \wedge p \in \vec{\imath}_n \wedge \widehat{m}[p] = \widehat{n}[p]$

Definition 10 (Enabled). *A message form $\widehat{m}$ is* enabled *and enabled($\widehat{m}, a, H_a$) holds if and only if:*

(1) $\widehat{m}$ is sent by a: $\widehat{m}[\mathfrak{s}_m] = a$
(2) consistent($\{\widehat{m}\}, H_a$)
(3) compatible$_{\vec{o}}(\{\widehat{m}\}, H_a) \wedge$ compatible$_{\vec{n}}(\{\widehat{m}\}, H_a)$
(4) derived($\widehat{m}, H_a$)

We also say that enabled(a, H_a) $\subset \mathcal{F}$ is the set of message forms that a is enabled to send.

Definition 11 says a decision maker constructs only instances that preserve the bindings from message forms.

Definition 11 (Decision Maker). *Let Q be a set of message forms; a* decision maker *is a function $d : \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{I})$ such that $\widehat{m}' \in d(Q) \implies instance(\widehat{m}') \wedge \exists \widehat{m} \in Q : \widehat{m}'[\mathfrak{s}_m, \mathfrak{r}_m, \vec{\imath}_m] = \widehat{m}[\mathfrak{s}_m, \mathfrak{r}_m, \vec{\imath}_m]$.*

3.1 Decision-Making Challenges and Solutions

We highlight select decision making patterns supported by Kiko.

3.1.1 Correlation. An agent may simultaneously be involved in several enactments of a protocol. For example, Buyer may be concurrently engaged with Seller in several distinct enactments, each for some item at some price. The programming model should enable correlating communications by enactment.

Kiko supports correlation through the automatic derivation of correlated forms by the adapter (as described above). The adapter computes forms based on all information available, potentially from the observation of multiple correlated instances. Kiko also makes it convenient to find correlated forms where the decision logic requires it. For example, in Listing 4, correlated Reject forms are found by the id of the Buy forms.

3.1.2 Cross-Enactment Decisions. Agents should be able to use information across enactments in their decision making.

Kiko enables cross-enactment reasoning by providing forms from *all* currently active *contexts*, that is, enactments in all systems, to the decision makers together. Thus, the decision maker can select forms from multiple contexts and flesh them out for emission. For example, Bob could participate in multiple systems, all enacting *Purchase*, to

request quotes for the same item from multiple sellers. Then, Bob can send a *Buy* for the *Quote* with the lowest price (Listing 6).

Listing 6: Selecting cheapest *Buy* across multiple contexts.

```
@adapter.decision
def cheapest(enabled):
  buys = enabled.messages(Buy)
  cheapest = min(buys, key=lambda b: b["price"])
  cheapest.bind(done=True)
```

3.1.3 Multiple Protocols. An agent will often play roles in multiple unrelated protocols, using information from one to make decisions in another.

Kiko enables implementing agents that play roles in multiple unrelated protocols. For example, we specify *Approval* in Listing 7. By enacting *Approval* concurrently with *Purchase*, Bob can seek Alice's approval on any purchases. To do so, Bob must map between the protocols inside its decision makers, which is supported by the enabled set containing forms from *all* the protocols Bob is enacting.

Listing 7: The *Approval* protocol.

```
Approval {
  roles Requester, Approver
  parameters out aID key, out request, out approved

  Requester -> Approver: Ask[out aID key, out request]
  Approver -> Requester: Approve[in aID, in request, out approved]
}
```

Listing 8 shows Bob's decision maker for constructing an *Ask* (approval) for each *Buy* as it becomes available as a form, copying *Buy*'s payload into request.

Listing 8: Requesting approval for a purchase across protocols.

```
@adapter.enabled(Buy)
def request_approval(buy):
    ask = next(adapter.enabled_messages.messages(Ask), None)
    return ask.bind(ID=str(uuid.uuid4()), request=buy.payload)
```

3.1.4 Emission sets. For additional flexibility, Kiko enables a decision maker to emit multiple instances atomically: if the instances are mutually compatible, then they are all emitted, else none are emitted. Thus, e.g., if an emission set contained *Buy* and *Reject* instances for the same enactment, no instance in the set would be emitted. Such atomicity of emission ensures correctness and gives full authority to the decision maker to choose its intended messages; multiple attempts can be made if needed. Selecting some consistent subset of the emission set for emission, by contrast, would be arbitrary and could lead to unintended enactments.

Listing 9 shows a decision maker, where Bob figures out the best combination of items it can buy (as computed by some optimization, whose details are not relevant for our purposes), sending *Buy*s for all those items and *Reject*s for the others.

Listing 9: A decision maker that sends *Buy* in some contexts and *Reject*s in the others.

```
@adapter.decision
def select_gifts(enabled):
    best, rest = best_combo(enabled)
    for b in best:  # buy the best items
        b.bind(done=True)
    for r in rest:  # reject the rest
        r.bind(done=True)
```

Another variety of decision logic where emission sets are valuable is a combination of "front-end" and "back-end" reasoning. For example, imagine Sally has a supplier with whom it engages via some protocol. Suppose Sally wants to order an item from its supplier whenever it delivers an item to a buyer. To accomplish this, it may have a decision maker which puts *Deliver* (to the buyer) and *Reorder* (from supplier) in the same emission set.

3.1.5 Reception-Order Freedom. Requiring agents to receive messages in a particular order can only delay the reception of information, which in turn would limit the agent's ability to respond flexibly to events.

Kiko takes advantage of the fact that BSPL doesn't rely on message ordering for correctness, and abstracts away message reception entirely from decision making. An agent's adapter receives messages as they arrive and depending on the information in them, makes forms available to decision makers. By doing so, Kiko enables agents to respond flexibly to events.

Listing 10: Rescind Quote.

```
  Seller -> Buyer: Rescind[in ID key, in item, in price, out
      rescinded]
  Buyer -> Seller: Buy[in ID key, ..., nil rescinded]
```

For example, Listing 10 extends *Purchase* by allowing Seller to *Rescind* a quote. Because it depends on price, *Rescind* must be sent after *Quote*, but could reach Bob first. Because reception is not constrained except by integrity (inconsistent messages are rejected), *Rescind* will be received, checked, and added to the history when it arrives. As such, the matching *Buy* will be disabled, and Bob need not waste any effort considering it (e.g., by requesting approval).

Note that by programming in terms of enabled forms, a decision maker such as the one in Listing 4 that emits *Buy*s need not change at all; the disabled *Buy*s are simply not provided to the decision maker for consideration.

3.1.6 Loose Coupling. Clearly, protocols support the independent development of agents by capturing the constraints relevant to interoperation between them. In general, if a protocol changes, then one would expect that the agents' decision making would have to change as well. Because Kiko is based on information though, it is not necessarily the case that protocol changes lead to changes in an agent's decision making, thus supporting loose coupling even better.

For example, suppose (as illustrated in Listing 11) *Purchase* included a *Deliver* message from Seller that depended on payment provided by *Buy*:

Listing 11: Delivery.

```
  Buyer -> Seller: Buy[in ID key, in item, in price, out payment]
  Seller -> Buyer: Deliver[in ID key, in payment, out delivery]
```

Then, suppose *Purchase* were extended so that Buyer could pay indirectly via bank transfer (as illustrated in Listing 12) . Because the messages in Listing 12 do not change the messages emitted by Seller, only how it receives the necessary information, Seller's decision logic need not be changed to support indirect payment. Seller's adapter will automatically derive the *Deliver* form when the indirect payment has been received, demonstrating loose coupling between the agents.

Listing 12: Bank Transfer.

```
Buyer -> Seller: Accept[in ID key, in price, out acceptance]
Buyer -> Bank: RequestTransfer[in ID key, in price, out txinfo]
Bank -> Seller: Transfer[in ID key, in txinfo, out payment]
```

3.1.7 Single Form Decision Makers. The general decision making pattern of supporting the emission of sets of instances is highly flexible, but for cases in which an agent need emit only instance at a time, Kiko supports the convenient abstraction of *single form* decision makers.

Such decision makers are functions invoked with a single message form; its return value is either a message instance for emission (binding its ⌜out⌝ parameters), or a null value canceling the emission. Listing 13 shows an example where an enabled form of *Quote* is fleshed out.

Listing 13: Single Form Decision Maker for *Quote*.

```
@adapter.enabled(Quote)
def send_quote(msg):
    msg["price"] = random.randint(20, 100)
    return msg
```

3.2 Adapter Implementation

Figure 2 blows up the adapter from Figure 1 to highlight its internal components (highlighted in green).

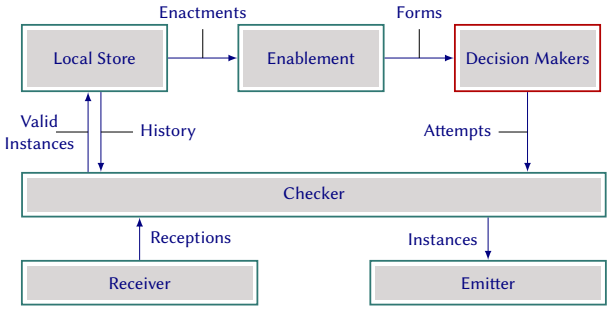

Figure 2: Adapter implementation.

The Emitter and Receiver interface with the communication service, putting messages on and receiving them from the wire, respectively. The Local Store records the agent's history of emissions and receptions. The Checker validates (checking for satisfaction of causality and integrity constraints in the protocol specifications) any attempt (by a decision maker) to emit a set of messages (Definition 12). If an attempt is validated, then the instances in it are added to the Local Store and passed on the Emitter for emission; else, the attempt is discarded.

Definition 12 (Send-Check). *If $H_a \subseteq \mathcal{M}$ is a history for agent a, and $T \subseteq \mathcal{M}$ is a set of message instances, $check_s(T, H_a)$ holds if and only if:*

(1) a is enabled to send every $\widehat{m}$ in T:
$\forall \widehat{m} \in T, enabled(\widehat{m}, a, H_a)$
(2) T is out-, and nil-compatible with T:
$compatible_{\vec{o}}(T, T) \wedge compatible_{\vec{n}}(T, T)$

If $check_s(T, H_a)$ holds, then T is a valid set of emissions for a and thus a valid extension of H_a.

The Checker also validates received messages for integrity; if they pass, they are added to the Local Store, else they are discarded (Definition 13).

Definition 13 (Receive-Check). *If $H_a \subseteq \mathcal{M}$ is a history for agent a, and $\widehat{m} \in \mathcal{M}$ is a message instance, $check_r(\widehat{m}, H_a)$ holds if and only if:*

(1) $\widehat{m}$ is receivable by a: $a = \widehat{m}[r_m]$
(2) $\widehat{m}$ is consistent and out-compatible with the history:
$consistent(\{\widehat{m}\}, H_a) \wedge compatible_{\vec{o}}(\{\widehat{m}\}, H_a)$

If $check_r(T, H_a)$ holds, it is valid for a to receive every instance in T and T is a valid extension of H_a.

The Local Store is used by Enablement to compute the forms that the agent is enabled to send. Algorithm 1 describes how enabled forms are computed for each context. We use an incremental method, so that only those contexts that have new information are updated. First, on Line 1, every context that shares key bindings with the observed instance $\widehat{o}$ is checked to see if it enables any instances of m. Lines 2 and 3 check that the ⌜out⌝ and ⌜nil⌝ parameters of the schema, respectively, are not already bound in the context. Line 4 copies the bindings of the ⌜in⌝ parameters from the context, Line 6 copies the system ID, and Line 7 adds the form to the result set for processing by decision makers.

Input: Message schema m, Message instance $\widehat{o}$
$Q \leftarrow \{\}$;
1 **foreach** $c \in matching\_contexts(\widehat{o})$ **do**
2 $\quad o \leftarrow \nexists p : p \in \vec{o}_m \wedge p \in c.\text{bindings}$;
3 $\quad n \leftarrow \nexists p : p \in \vec{n}_m \wedge p \in c.\text{bindings}$;
4 $\quad i \leftarrow \forall p : p \in \vec{i}_m \implies p \in c.\text{bindings}$;
$\quad$ **if** $o \wedge i \wedge n$ **then**
5 $\quad\quad \widehat{m}[\vec{i}_m] \leftarrow c.\text{bindings}[\vec{i}_m]$;
6 $\quad\quad \widehat{m}[\mu] \leftarrow \widehat{o}[\mu]$;
7 $\quad\quad Q \leftarrow Q \cup \widehat{m}$;
$\quad$ **end**
$\quad$ **return** Q;
end

Algorithm 1: Derive instance of schema from observation.

4 OPERATIONAL SEMANTICS

Protocols are formalized in an online Appendix. Here, we formalize an agent and MAS computations via a transition semantics.

Figure 3 gives the transition semantics.

The Recv rule specifies how messages are received. For agent a to receive a message instance $\widehat{m}$ there are three conditions: (1) $\widehat{m}$ must be in the agent's input channel I_a, (2) $\widehat{m}$ must not already be in the agent's history H_a, and (3) $\widehat{m}$ must be a valid extension of H_a. If these three conditions are met, then $\widehat{m}$ is added to H_a.

The Tx rule models message delivery by copying messages from an output channel to the appropriate input channel; unreliability is modeled by not exercising the rule.

Finally, Decide specifies how messages are instantiated for emission: First, a set Q of message forms is computed based on the

Message Schema	$m \in S_P$
Message Instance	$\widehat{m} \in \mathcal{M}$
History	$H \in \mathcal{H} \subseteq \mathcal{M}$
Input	$I \subseteq \mathcal{M}$
Output	$O \subseteq \mathcal{M}$
Agent	$a \coloneqq \langle H_a, I_a, O_a \rangle \in \mathcal{A}$
Check	$\mathsf{check}_r \in \mathcal{M} \times \mathcal{H} \to \{\mathbf{T}, \mathbf{F}\}$
	$\mathsf{check}_s \in \mathcal{P}(\mathcal{M}) \times \mathcal{H} \to \{\mathbf{T}, \mathbf{F}\}$
Enabled	$\mathsf{enabled} \in \mathcal{A} \times \mathcal{H} \to \mathcal{P}(\mathcal{F})$
Decision maker	$d \in \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{M})$
Consistent	$\mathsf{consistent} \in \mathcal{P}(\mathcal{M}) \to \{\mathbf{T}, \mathbf{F}\}$

$$\textsc{Recv}\ \frac{\widehat{m} \in I_a \qquad \widehat{m} \notin H_a \qquad \mathsf{check}_r(\widehat{m}, H_a)}{a\langle H_a, I_a, O_a\rangle \longrightarrow a\langle H_a \cup \{\widehat{m}\}, I_a, O_a\rangle}$$

$$\textsc{Tx}\ \frac{\widehat{m} \in O_x \qquad \widehat{m}[r] = y}{I_y \longrightarrow I_y \cup \{\widehat{m}\}}$$

$$\textsc{Decide}\ \frac{Q \coloneqq \mathsf{enabled}(a, H_a) \qquad T \coloneqq d(Q) \qquad \mathsf{check}_s(T, H_a)}{a\langle H_a, I_a, O_a\rangle \longrightarrow a\langle H_a \cup T, I_a, O_a \cup T\rangle}$$

Figure 3: Notation and core semantics.

$$\textsc{Decide}_2\ \frac{\begin{array}{c}Q \coloneqq \mathsf{enabled}(a, H_a) \quad T \coloneqq d(Q) \\ \mathsf{compatible}_{\vec{o}}(T, T) \quad \mathsf{compatible}_{\vec{n}}(T, T)\end{array}}{a\langle H_a, I_a, O_a\rangle \longrightarrow a\langle H_a \cup T, I_a, O_a \cup T\rangle}$$

Figure 4: Optimized decision that checks for internal consistency instead of full validity.

agent's history. Next, a set of instances are derived from the message forms by applying a decision maker d to the enabled form set. If this set of instances is valid, then it is added to both the agent's history and output channel. Otherwise, the rule cannot be applied and no messages are sent.

No rules are required for cases where the messages fail a validity check; there is simply no transition in those cases. A transition for a MAS is simply a transition for one of its agents.

Figure 4 shows an alternative version of the Decide rule, $\textsc{Decide}_2$. Because transitions are atomic, the forms will not be disabled before the transition completes, so they do not need to be rechecked for validity; checking internal compatibility is sufficient (e.g., not selecting both an *Accept* and *Reject* in the same enactment). Checking only internal compatibility of a small set of emissions should be faster than a full send-check, which requires both internal compatibility *and* that the instance is consistent and compatible with the rest of the agent's history.

Our goal is to show that a MAS developed using our operational semantics to implement a protocol will be both correct (that is, reach only valid states) and complete (it is possible to implement a system that can reach any valid state). As such, we formalize the state of a MAS, which states are reachable according to the operational semantics, and which states *match* a protocol enactment.

Definition 14 (MAS State). *The* state *of a MAS μ is the set of its agent histories: $\{H_a | a \in A_\mu\}$*

Definition 15 (Reachable State). *Given MAS μ and transition semantics $\mathcal{T}$, state s of MAS μ is* reachable *and an element of $\mathcal{S}_{\mu,\mathcal{T}}$ if and only if there is a sequence of transitions $t_i \in \mathbb{N} \to \mathcal{T}$ that results in state s.*

$\mathcal{E}_P$ (formally defined in the Appendix) is the set of reachable enactments of protocol P, where a reachable enactment $E \in \mathcal{E}_P$ is a set of role histories each constructed by a sequence of viable events according to P's specification.

Definition 16 (Matching State). *If μ is a MAS implementing protocol P, then state s of μ* matches *$E \in \mathcal{E}_P$, written $s \equiv E$, if and only if, for every agent history H_a in s and instance $\widehat{m} \in H_a$:*

(1) if a plays $\flat_m$ in μ then m is sent in the corresponding role history $H_{\flat_m} \in E$ (that is, $a = \widehat{m}[\flat_m] \implies \langle sent, m\rangle \in H_{r_m}$)
(2) if a plays the receiver of m in μ then m is received in the corresponding role history $H_{r_m} \in E$ (that is, $a = \widehat{m}[r_m] \implies \langle received, m\rangle \in H_{r_m}$)

Simulation is the idea that transitions in the MAS should match the reachable enactments in its protocol; each transition may be equivalent to a set of *multiple* viable extensions because the Decide rule can produce a *set* of message instances, where viable extensions cover only one instance at a time.

Definition 17 (Simulation). *If μ is a MAS implementing protocol P, then state $s \in \mathcal{S}_\mu$* simulates *$E \in \mathcal{E}_P$, written $s \sim E$, if and only if, for every agent history H_a in s and instance $\widehat{m} \in H_a$:*

(1) s matches E
(2) for every transition t, the state $s' : s \xrightarrow{t} s'$ matches some enactment E' reachable from E in a finite number of viable extensions.

We can now state the following theorems, whose proofs are in the Appendix.

Theorem 1. *Given a MAS μ implementing protocol P, every reachable state $s \in \mathcal{S}_\mu$ simulates some enactment $E \in \mathcal{E}_P$.*

Theorem 1 gives the *correctness* of our operational semantics by showing that compliant MAS can only reach states that match reachable enactments of a protocol. Even though the states reached by the MAS will depend on the decision makers, they can only select subsets of the enabled forms, and therefore cannot reach an invalid state (that is, one that does not match an enactment that is reachable under the protocol semantics).

Theorem 2. *$\textsc{Decide}_2$ is equivalent to Decide.*

Theorem 2 shows that the conditions for Decide are redundant, given that the forms are drawn from $\mathsf{enabled}(a, H_a)$ and decision makers preserve their bindings (and thus consistency and compatibility with history); all that needs to be checked for the selected emissions T is that they are compatible with each other.

Theorem 3. *Given a MAS μ implementing protocol P, there is some set of decision makers D that can simulate any reachable enactment in $\mathcal{E}_P$, assuming that all sent message instances are received.*

Theorem 3 shows *completeness* for our operational semantics: the operational semantics do not restrict a MAS from simulating any reachable enactment of the protocol. Or, given a reachable enactment of a protocol, it is possible to construct decision makers for the agents that would reach that enactment. This is not to say that every implementation is complete; proving completeness for a given implementation would require formalizing its decision makers as transition rules.

5 DISCUSSION

Kiko bridges business logic and communications: an agent provides business decisions and the underlying adapter applies the protocol semantics to determine which messages are viable. The underlying causal information semantics captures the information flow and avoids having to generate guards [33]. An agent makes and communicates a set of decisions (as reflected in the forms provided by the adapter) based on some evaluation of the state of the world. The decision making is conceptualized declaratively and suits rule-based programming languages such as Jason.

An interesting direction is to extend Kiko's notion of forms to support norms-based decision making. For example, the discharge of a commitment by an agent could be made available as a form to be picked and instantiated by the agent. Baldoni et al. [4] present a model for accountability that is implemented in JaCaMo via obligations and relates to both (the giving of) accounts and recovery strategies when things go wrong. Kiko's adapter could incorporate standard protocols for demanding accounts from other agents when norm violations occur and incorporate them into further decision making, e.g., to decide from which agent to buy items.

Variants of programming models based on information protocols have been proposed in recent years. The idea of enabled message forms was first introduced in Stellar [21]; however, Stellar lacked support for emission sets and relied on the abstraction of message handlers as opposed to decision makers. Thus, Bob's implementation in Stellar would be a set of message handlers, one for each type of message it could observe. Within a message handler, one could retrieve a form and instantiate it. Message handling-based abstractions are lower level compared to Kiko's decision makers, which are information-based. To see this, suppose an agent needed information from two instances, say i_1 and i_2, which it may receive in any order, to be able to send a third instance i_3 (e.g., a shipper may need the address from the buyer and the item from the seller to be able to deliver). Then, in the message handling approach of Stellar, one would write separate message handlers for i_1 and i_2 and in each one check whether the form for i_3 is available. By contrast, in Kiko, one would simply write a single decision maker that completes the form for i_3. The Mandrake [13] and PoT [14] programming models share Stellar's limitations; however, they both also address application-level fault tolerance, a theme that is a direction for Kiko.

Like Stellar (and Mandrake and PoT), Kiko enables building applications directly over an unordered, unreliable communication service such as UDP for message transport. Kiko is therefore compatible with the influential end-to-end argument [32], which advocates building applications over simple communication services, both for reasons of enabling application-level flexibility and performance. By contrast, message ordering-based protocol approaches would be incompatible with the end-to-end argument. Establishing the performance of Kiko-based agents and MAS compared to traditional application architectures that rely on complex communication services and middleware is a crucial direction. Preliminary evidence from Mandrake and PoT indicates high performance.

Kiko's features such as support for correlation, cross-enactment reasoning, and multiple protocols are not readily supported in programming models for message ordering-based protocol approaches. This is because all of the above features have to do with querying information, which is inadequately represented in ordering-based protocols. Emission sets are unique to Kiko and are a powerful feature that enables emitting a set of message instances (possibly from different protocols and to different agents) atomically.

In the current semantics, decision makers execute atomically with respect to the history. This makes possible the optimization of simply checking the internal compatibility of the emission set before emitting all its instances. However, an alternative semantics is possible where decision makers execute concurrently from the same history. Concurrent execution would enable taking advantage of multicore and cloud architectures. Implementation-wise, decision makers could be spawned off as actors [1, 23]. The tradeoff is that the emission sets produced by concurrent decision makers may be in conflict with each other (e.g., one set contains *Buy* whereas another contains *Reject* for the same enactment) and therefore an internal compatibility check would no longer suffice. Each emission set would have to be checked for validity against the history, which could be more expensive.

IoT-based paradigms such as edge and fog computing and the industry paradigm of realizing applications via microservices are conceptually decentralized. In the case of microservices especially, decentralization is driven by the scalability afforded by the containerization of application components. Current microservices development approaches tend to avoid distributed database transactions in favor of loose coupling [27]. However, this raises the question: On what basis should microservices coordinate their computations? Information protocols could be thought of as a model for *business transactions*. Therefore, approaches like Kiko, suitably adapted to microservices, can help.

SARL [20] is an agent programming language that supports communication using events in spaces that are akin to environments [40]. SARL would benefit from a protocol-based programming model. Kiko would benefit from a more general treatment of events. Currently, in Kiko, messages model events. However, some domain events don't map to messages. For example, while a *Quote* may reasonably be modeled as a message, *Shipment* may actually correspond to a package traveling in the back of a truck. Receiving a shipment, therefore, requires sensing the arrival of the package. Extending Kiko's adapter to incorporate observation of events from the environment would be valuable.

Supplementary Material. The online Appendix and the Kiko software are available at: https://gitlab.com/masr/bspl/-/tree/kiko.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. We thank the EPSRC (grant EP/N027965/1) and the US National Science Foundation (grant IIS-1908374) for partial support.

REFERENCES

[1] Gul A. Agha. 1986. *Actors.* MIT Press, Cambridge, Massachusetts. https://doi.org/10.7551/mitpress/1086.001.0001

[2] Matteo Baldoni, Cristina Baroglio, Federico Capuzzimati, and Roberto Micalizio. 2019. Process Coordination with Business Artifacts and Multiagent Technologies. *Journal on Data Semantics* 8, 2 (June 2019), 99–112. https://doi.org/10.1007/s13740-019-00100-8

[3] Matteo Baldoni, Cristina Baroglio, Amit K. Chopra, Nirmit Desai, Viviana Patti, and Munindar P. Singh. 2009. Choice, Interoperability, and Conformance in Interaction Protocols and Service Choreographies. In *Proceedings of the 8th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS).* IFAAMAS, Budapest, 843–850. https://doi.org/10.5555/1558109.1558129

[4] Matteo Baldoni, Cristina Baroglio, Roberto Micalizio, and Stefano Tedeschi. 2021. Robustness Based on Accountability in Multiagent Organizations. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS).* IFAAMAS, Online, 142–150. https://doi.org/10.5555/3461017.3461040

[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] Federico Bergenti, Giovanni Caire, Stefania Monica, and Agostino Poggi. 2020. The First Twenty Years of Agent-Based Software Development with JADE. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 34, 2 (2020), 36. https://doi.org/10.1007/s10458-020-09460-z

[7] 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

[8] 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

[9] Luca Cernuzzi, Thomas Juan, Leon Sterling, and Franco Zambonelli. 2004. The Gaia Methodology. In *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*, Federico Bergenti, Marie-Pierre Gleizes, and Franco Zambonelli (Eds.). Multiagent Systems, Artificial Societies, and Simulated Organizations, Vol. 11. Kluwer, Dordrecht, Netherlands, Chapter 4, 69–88. https://doi.org/10.1007/1-4020-8058-1_6

[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 Technology (TIST)* 42, 2, Article 20 (March 2013), 23 pages. https://doi.org/10.1145/2438653.2438655

[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] Samuel H. Christie V, Amit K. Chopra, and Munindar P. Singh. 2021. Bungie: Improving Fault Tolerance via Extensible Application-Level Protocols. *IEEE Computer* 54, 5 (May 2021), 44–53. https://doi.org/10.1109/MC.2021.3052147

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

[14] Samuel H. Christie V, Daria Smirnova, Amit K. Chopra, and Munindar P. Singh. 2020. Protocols Over Things: A Decentralized Programming Model for the Internet of Things. *IEEE Computer* 53, 12 (Dec. 2020), 60–68. https://doi.org/10.1109/MC.2020.3023887

[15] Massimo Cossentino, Nicolas Gaud, Vincent Hilaire, Stéphane Galland, and Abderrafiaa Koukam. 2010. ASPECS: An Agent-Oriented Software Process for Engineering Complex Systems. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 20, 2 (March 2010), 260–304. https://doi.org/10.1007/s10458-009-9099-4

[16] Nirmit Desai, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh. 2005. Interaction Protocols as Design Abstractions for Business Processes. *IEEE Transactions on Software Engineering* 31, 12 (Dec. 2005), 1015–1027. https://doi.org/10.1109/TSE.2005.140

[17] Nirmit Desai and Munindar P. Singh. 2008. On the Enactability of Business Protocols. In *Proceedings of the 23rd Conference on Artificial Intelligence (AAAI).* AAAI Press, Chicago, 1126–1131. http://www.aaai.org/Library/AAAI/2008/aaai08-178.php

[18] 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

[19] FIPA. 2003. FIPA Interaction Protocol Specifications. http://www.fipa.org/repository/ips.html FIPA: The Foundation for Intelligent Physical Agents. Accessed 2023-02-27.

[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] Akin Günay and Amit K. Chopra. 2018. Stellar: A Programming Model for Developing Protocol-Compliant Agents. In *Proceedings of the 6th International Workshop on Engineering Multi-Agent Systems (EMAS) (Lecture Notes in Computer Science, Vol. 11375).* Springer, Stockholm, 117–136. https://doi.org/10.1007/978-3-030-25693-7_7

[22] 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

[23] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular Actor Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI).* William Kaufmann, Stanford, 235–245. http://ijcai.org/Proceedings/73/Papers/027B.pdf

[24] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* ACM, San Francisco, 273–284. https://doi.org/10.1145/1328438.1328472

[25] Michael N. Huhns, Nigel Jacobs, Tomasz Ksiezyk, Wei-Min Shen, Munindar P. Singh, and Philip E. Cannata. 1992. Enterprise Information Modeling and Model Integration in Carnot. In *Enterprise Integration Modeling: Proceedings of the First International Conference*, Charles J. Petrie, Jr. (Ed.). MIT Press, Hilton Head, South Carolina, 290–299. https://doi.org/10.7551/mitpress/2768.003.0036

[26] Thomas Christopher King, Akın Günay, Amit K. Chopra, and Munindar P. Singh. 2017. Tosca: Operationalizing Commitments over Information Protocols. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI).* IJCAI, Melbourne, 256–264. https://doi.org/10.24963/ijcai.2017/37

[27] Rodrigo N. Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. 2021. Data Management in Microservices: State of the Practice, Challenges, and Research Directions. *Proceedings of the VLDB Endowment* 14, 13 (Sept. 2021), 3348–3361. https://doi.org/10.14778/3484224.3484232

[28] Mark Little. 2017. Virtual Panel: Microservices in Practice. https://www.infoq.com/articles/microservices-in-practice/. Accessed: 1 Mar 2023.

[29] James Odell, H. Van Dyke Parunak, and Bernhard Bauer. 2001. Representing Agent Interaction Protocols in UML. In *Proceedings of the 1st International Workshop on Agent-Oriented Software Engineering (AOSE 2000) (Lecture Notes in Computer Science, Vol. 1957).* Springer, Toronto, 121–140. https://doi.org/10.1007/3-540-44564-1_8

[30] Lin Padgham and Michael Winikoff. 2005. Prometheus: A Practical Agent-Oriented Methodology. In *Agent-Oriented Methodologies*, Brian Henderson-Sellers and Paolo Giorgini (Eds.). Idea Group, Hershey, Pennsylvania, Chapter 5, 107–135. https://doi.org/10.4018/978-1-59140-581-8.ch005

[31] Jeremy Pitt, Julia Schaumeier, and Alexander Artikis. 2012. Axiomatization of Socio-Economic Principles for Self-Organizing Institutions: Concepts, Experiments and Challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 7, 4, Article 39 (Dec. 2012), 39 pages. https://doi.org/10.1145/2382570.2382575

[32] Jerome H. Saltzer, David P. Reed, and David D. Clark. 1984. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems* 2, 4 (Nov. 1984), 277–288. https://doi.org/10.1145/357401.357402

[33] Munindar P. Singh. 1996. Synthesizing Distributed Constrained Events from Transactional Workflow Specifications. In *Proceedings of the 12th International Conference on Data Engineering (ICDE).* IEEE, New Orleans, 616–623. https://doi.org/10.1109/ICDE.1996.492212

[34] 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

[35] Munindar P. Singh. 1998. A Customizable Coordination Service for Autonomous Agents. In *Intelligent Agents IV: Proceedings of the 4th International Workshop on Agent Theories, Architectures, and Languages (ATAL-97) (Lecture Notes in Computer Science, 1365).* Springer, Providence, Rhode Island, 93–106. https://doi.org/10.1007/BFb0026752

[36] 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

[37] Munindar P. Singh. 2013. Norms as a Basis for Governing Sociotechnical Systems. *ACM Transactions on Intelligent Systems and Technology (TIST)* 5, 1, Article 21 (Dec. 2013), 23 pages. https://doi.org/10.1145/2542182.2542203

[38] Munindar P. Singh and Amit K. Chopra. 2020. Clouseau: Generating Communication Protocols from Commitments. In *Proceedings of the 34th Conference on Artificial Intelligence (AAAI).* AAAI Press, New York, 7244–7252. https:

//doi.org/10.1609/aaai.v34i05.6215

[39] Benjamin Smith. 2021. Getting started with serverless for developers: Part 2 - The business logic. https://aws.amazon.com/blogs/compute/getting-started-with-serverless-for-developers-part-2-the-business-logic/. Accessed: 1 Mar 2023.

[40] Danny Weyns, Andrea Omicini, and James Odell. 2007. Environment as a First Class Abstraction in Multiagent Systems. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 14, 1 (Feb. 2007), 5–30. https://doi.org/10.1007/s10458-006-0012-0

[41] 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

[42] 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