ABSTRACT

CHRISTIE V, SAMUEL HENRY. Information-based Decentralized Systems. (Under the direction of Dr. Munindar Singh.)

As computing becomes ever more important for our economy and daily lives, we interact with more and more devices with varying architectures and responsibilities. Scaling limitations push us beyond many-core to large, multi-site cloud deployments, and fault tolerance requires multiple physical devices to avoid single points of failure. Distributed computing is already here and becomes more important every day.

However, current distributed systems are inflexible and limited in their interactions to fixed architectures or simple APIs. Components are frequently built according to the asymmetric client-server model, perform only a single function as a service, and interact with only a small number of clients. *Decentralized* systems, formed of multiple autonomous entities, are much more flexible but difficult to design correctly.

Although work has been done in the multiagent systems community to develop interaction-level specifications to support the design of decentralized systems, most have worked from basic concepts and models that are easy for programmers to understand but do not reflect the realities of decentralized systems. By assuming reliable, ordered communication, they can continue modeling each process as a sequence of events, similar to classic algorithms. However, physical reality is not reliable nor ordered. Actions at a distance are necessarily asynchronous, and under relativity may appear to occur in different orders to different observers. Autonomy only exacerbates the problem, since agents can delay or withdraw at any time.

Instead of specifying and constraining systems based on some global perspective, which would be unrealizable, we propose focusing on the *information* and meaning in the interactions between parties. Using causal dependencies between the information communicated via messages to derive their ordering gives rise to *information protocols*, which are both more flexible and realizable than sequence-based protocols. Focusing on causality at emission enables information protocols to support both autonomy and asynchrony. The information communicated in such an operational protocol can then be used as a foundation for *norms* that give them meaning in a social context, and these norms can be further composed into commitment protocols or declarative contracts.

In this dissertation, we build on the basic concepts of information protocols and norms to create verification and development techniques necessary for specifying and implementing reliable and effective decentralized systems.

Information-based Decentralized Systems

by
Samuel H. Christie V

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2022

APPROVED BY:

Dr. Amit Chopra External Member	Dr. Jon Doyle
Dr. Rudra Dutta	Dr. John-Paul Ore
Dr. Hong Wan	Dr. Munindar Singh Chair of Advisory Committee

BIOGRAPHY

Samuel H. Christie V was homeschooled by his parents Sam and Martha Christie from third grade through highschool. He began taking classes at NCSU through dual enrollment and later joined as an undergraduate in 2009. He completed his bachelor's degrees in both mathematics and computer science in 2012 and spent a few months doing software development contracting at Emerging Threats.

In Fall 2013 he returned to NCSU for graduate studies in computer science—partly motivated by a desire to continue studying the Japanese language on the side, and partly out of curiosity about what exactly research in computer science entailed. He spent the first few years focused on classes and briefly worked with Dr. Emerson Murphy-Hill in software engineering. He eventually found Dr. Munindar Singh, who shared an interest in decentralized systems, and they began working together through an independent study on blockchains. Dr. Singh then introduced him to information protocols, which became the focus of his research.

Samuel worked at the Lab for Analytic Sciences from 2014-2016, first on recommender systems and then on a protocol-based analytic workflow platform. From 2016-2019 he worked part-time at IBM in the Blockchain division on consent management and decentralized identity. From 2019 until the end of his studies he focused more on research, taking a position at Lancaster University in England as a research associate under Dr. Amit Chopra, where he began working on more practical applications of information protocol theory.

More personally, he was involved extensively with a church on campus, and participated in several summer missions projects with them, including to Orlando and the Dominican Republic. Samuel loves traveling, and he and his brother Stephen spent time backpacking Germany, Austria and the Czech Republic. Now they live on a sailboat, work remotely, and sail to warmer climes in the winter. He enjoys reading web novels, playing computer games with friends, and reinventing the internet.

ACKNOWLEDGMENTS

This work would not have been possible without the support, guidance, and patience of my advsor Dr. Munindar Singh, who always had insightful comments on how to improve or redirect a project, and my unofficial co-advisor Dr. Amit Chopra, who was frequently available for brainstorming and pair editing. Nearly everything I have done has been a direct result of our bi-weekly research meetings.

I would also like to thank my advising committee, Drs. Jon Doyle, Rudra Dutta, John-Paul Ore, and Hong Wan, for putting up with the difficulties of scheduling so many people for a trans-continental exam, and providing helpful critiques of my proposal.

Daria Smirnov was a close collaborator during my time at Lancaster University, and helped significantly on the PoT project. Aditya Parkhi and Bhavana Balraj helped integrate Hercule with Hyperledger Fabric.

I also appreciate my fellow students at the Service-oriented Computing lab at NCSU, especially Nirav Ajmeri and Zhen Guo.

During my extended graduate studies, I received financial support from the NSA Laboratory of Analytic Sciences, NSF, and EPSRC, and was at times employed by the IBM Blockchain Division and the School of Computing and Communications at Lancaster University.

SDG

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Computing has grown from solving equations and performing individual tasks to providing services and forming the substrate of entire business and social systems. The characteristics of a multiorganizational system are very different from a single service, and so the methods and considerations for specifying such a decentralized system must also differ.

1.1 What Decentralization Means

Much work has been done in computing systems focused on *distributed* systems, which are often confused with decentralized systems because they both usually involve multiple, physically separated computers. However, there are important distinctions between the two.

1.1.1 Distributed Systems

Distributed systems are those which involve multiple independently operating computers; distribution effectively occurs at the physical level. Distributed systems can range from multiple cores or CPUs on the same motherboard at one end, to large heterogeneous networks of computers spread across the internet. However, while distributed computers may operate independently (have separate clocks and instruction streams, etc.), they are not necessarily *autonomous*. That is, they could be centrally managed and orchestrated to coordinate their behavior. These centralized, distributed systems are common in large high-performance scientific computing applications.

"Grid computing" is a category of large, homogeneous distributed computing systems where problem subsets are replicated across the network and communicate in strictly patterened ways. Nodes in a grid computing usually only communicate with their neighbors, using a messaging framework like MPI.

1.1.2 Decentralized Systems

Decentralized systems are those involving multiple autonomous parties, and effectively occurs at the semantic level. The autonomous parties need not actually be physically distributed; they could be software agents running on the same hardware (e.g., multiple companies using the same cloud infrastructure). Instead, the distinction is in the authority and control relationships. Agents in decentralized systems can belong to separate organizations, and may have different objectives and motivations for participating.

Decentralized systems suffer from unique problems that arise from the participants' autonomy, such as those involving consensus (Byzantine faults), and alignment, malice, etc. However, many of the classic distributed computing problems also apply to decentralized systems; examples include race conditions and message loss. Decentralization can exacerbate these problems, because centralized solutions are unavailable.

These problems can be solved through the use of information- and interaction-oriented methods. The necessity of such methods comes from the inadequacy of centralized control and synchronous communication, which give mere illusions of a consistent state across multiple computers; discrepancies will still have to be handled in case of an error. The sufficiency of such methods comes from directly specifying information flows to avoid conflicts; centralized restrictions implicitly do the same by e.g. giving one process ownership of a variable. Finally, centralization can significantly reduce performance; blockchains provide a clear example: they achieve a consistent, single history through a synchronously updated data structure, but at the cost of slowing down the entire network to the speed of a single node.

1.2 Important Properties for Decentralized Systems

The following subsections qualititatively describe desireable properties for decentralized systems. We try to explain them clearly, to provide an overview and perspective for motivating and understanding the rest of the work, but they are not formally defined here.

1.2.1 Autonomy

Principals (or participants) in a decentralized system are *autonomous* because they are not controlled by the others, so in one sense autonomy is intrinsic to a decentralized system, and many works in multiagent systems take such autonomy for granted.

However, a system may respect the autonomy of any given agent to varying degrees. Just as a government may constrain the behavior of its residents despite the fundamental autonomy of humans, a decentralized system may only partially respect the autonomy of

its constituents. As such, autonomy may also be a *value* in decentralized systems, just as freedom is in society, which requires explicit motivation and support. Freedom, sadly, is never free.

We distinguish autonomy (freedom from restraints or compulsion) from *independence*, which is the ability to act without dependencies on other agents. Independence is directly useful; fewer dependencies means fewer sources of failure. Independence also reduces the need for coordination, improving scalability. These benefits of independence apply at runtime, but also during design; design coupling increases the need for coordination and reduces scalability. Autonomy enables agents to operate more independently by reducing their dependence on external authorities. Thus we strongly emphasize the value of autonomy to enable independent agents, even though autonomy alone cannot ensure independence.

Of course, some apparent constraints on autonomy are necessary for a functional system; if all agents acted fully independently without any cooperation there would be no system, just independent entities. The question then becomes: How can we maximize both independence and cooperation? Fortunately, cooperation need not come at the expense of autonomy; agents can voluntarily coordinate their behavior.

Voluntary coordination is a basic motivation for interaction specifications such as protocols and norms. Interaction specifications enable discussion and negotiation, just as contracts facilitate business. Without specifications, agents are currently coordinated through orchestration; either implicitly by design coupling, or explicitly by external control; perhaps in the future machine learning will enable more dynamic unspecified interactions. Specifying an interaction does not in itself restrict autonomy, because the agents may choose which specification to enact.

Within a specification, care must be taken not to unnecessarily restrict autonomy, and thus independence. As such, interaction specifications properly address only the interaction, and leave the agent implementations unconstrained; this supports design independence. However, if interaction specifications do not address implementation, they cannot be directly concerned with runtime independence either.

1.2.2 Heterogeneity

Closely related to autonomy is the concept of *heterogeneity*: that not all agents are the same, even those playing the same role.

High performance computing application are often implemented using distributed systems, using multiple processors to split the work into smaller pieces; for example, splitting a fluid simulation into thousands of tiny subvolumes. Such computing systems are homogeneous,

with each processor expected to perform a nearly identical computation, and generally directed by a task scheduler that has complete control over each processor.

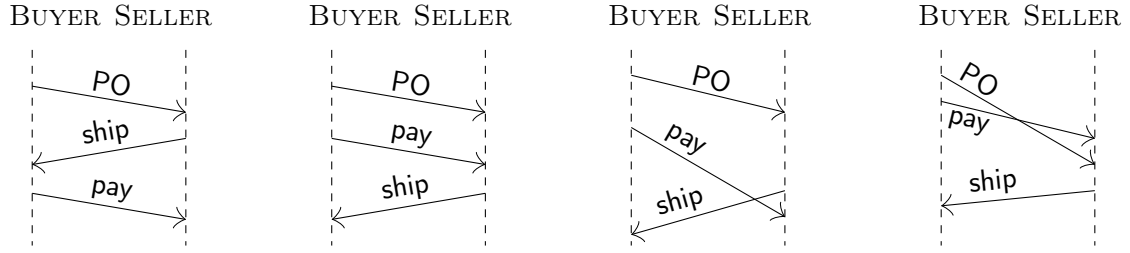
In contrast, decentralized systems should be designed to accomodate the heterogeneity that comes with autonomous agents, especially those belonging to different organizations who likely have different goals. The heterogeneity appears in the contrast between the abstract roles of a protocol and the autonomous agents that play them. Agents may be considered 'the same' in the sense that they play the same role: they are required to handle and emit equivalent messages. However, the logic and resulting values they produce are unlikely to be the same. For example, bidders in an auction will have different budgets and preferences, and therefore bid different amounts; they may also employ different strategies, such as bidding early to intimidate opponents, or bidding late to snipe them.

1.2.3 Asynchrony

In parallel to our justification of autonomy, independence also motivates asynchronous interactions.

Many systems are designed using simplifying assumptions about the communication infrastructure, such as synchronicity or First-In-First-Out (FIFO) ordering of messages. Such assumptions implicitly share state between two endpoints; they can start a conversation and assume that subsequent messages are part of that conversation. Unfortunately, such assumptions limit independence, restricting the agents to handling the messages in a particular order and possibly preventing them from participating in multiple parallel interactions.

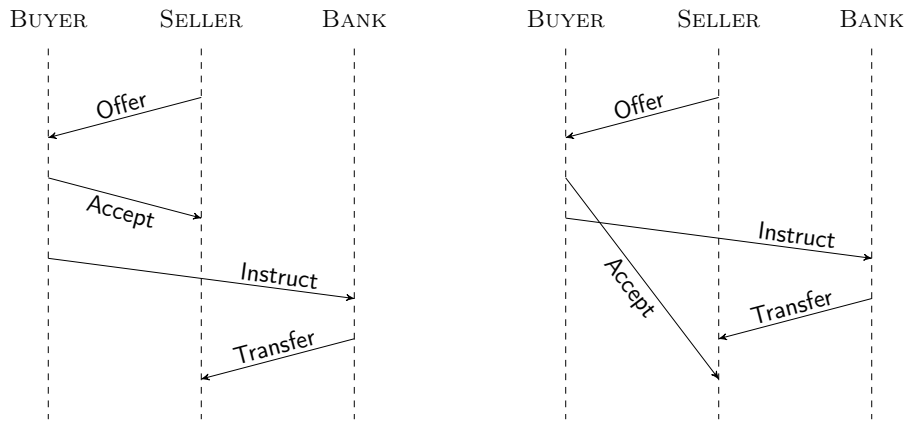
Figure 1.1 shows multiple enactments of a simple purchase protocol that are possible if the ordering is not constrained.



(a) Shipment first (b) Payment first (c) Concurrent (d) non-FIFO

Figure 1.1: Some possible equivalent enactments of *Purchase*.

Any constraints on infrastructure contrary to reality will eventually be violated. Although radio is effectively synchronous over short distances, relativity implies that temporal relationships depend on the frame of reference; over larger distances or slower communication mediums, ordering is unreliable. Any properties relying on assumptions of synchronicity or FIFO ordering will eventually be violated, or preserved at great expense. FIFO ordering, for example, can be preserved by buffering messages and reordering them; however, if (and we argue this is the case) such ordering is not actually essential to correctness, the resulting delays are wasteful. Further, if an interaction involves more than two parties, FIFO is insufficient to guarantee ordering, as shown in Figure 1.2.



(a) In-order delivery.

(b) Out-of-order delivery.

Figure 1.2: FIFO communication does not guarantee consistent ordering across a multiagent system with three or more agents. In this indirect payment protocol each pairwise channel preserves FIFO, but *Accept* can still arrive after *Transfer* as in (b)

Since constraints on infrastructure limit independence and can only be preserved at unnecessary and prohibitive cost, we argue that decentralized systems should be designed to operate on fully asynchronous communication. Note that this is not an argument for the *exclusion* of ordered networking; only against *reliance* on ordering. Systems designed for asynchrony can operate correctly regardless of the infrastructure used, adding another dimension of independence.

1.2.4 Integrity

Besides showing more practical applications of the work, we also use a logistical scenario as an example for *information integrity*, which refers to maintaining the consistency of interaction

information across multiple agents within a system. In a logistical setting, multiple agents are likely processing many different items from multiple manifests (e.g. multiple items in a purchase), and those items need to be properly correlated to pack multiple items in the correct box, or multiple boxes in the correct truck, and deliver them correctly.

A diagram of this scenario is shown in Figure 1.3.

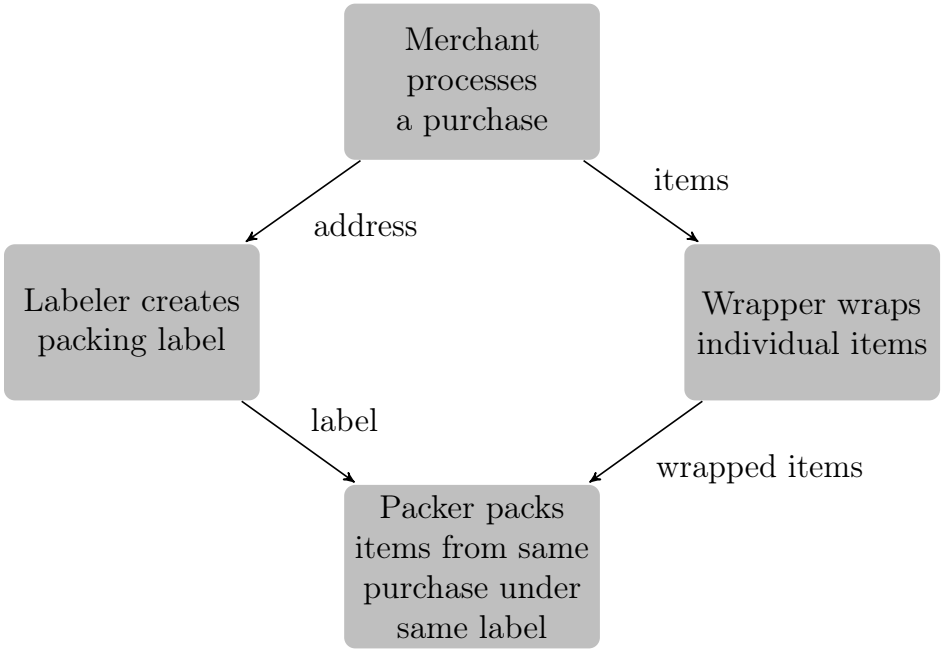


Figure 1.3: Example logistics flow

In this flow, Merchant processes a purchase by sending an address to a Labeler for the creation of a packing label, and item information to a Wrapper to wrap the individual items. Those two agents then send the information about their work to a Packer, who packs the items in the box with the appropriate label, and notifies the Merchant when it is done.

There are two aspects of integrity: consistency and safety. Consistency is the basic runtime property that parameter bindings are preserved throughout an interaction; once an agent observes a given binding in an enactment, it should not observe a different (conflicting) binding. Safety is a property of the interaction itself, which requires that only one agent be responsible for binding a parameter in a given enactment. Safety is necessary for consistency: if two agents can produce bindings for the same parameter (unless they use *exactly* the same logic, and are therefore indistinguishable from being the same agent) they will sometimes produce different bindings, and violate consistency and integrity.

Although all interactions require integrity, the logistics flow demonstrates a particular

application of the concept: correlation. In order to maintain a consistent relationship between the items, address, and purchase ID, the Packer must match the items to the correct label. If integrity is not maintained, then either the label, item, or both would be inconsistent with the original purchase.

Integrity is a locally verified property. Each agent must check all of its incoming and outgoing messages to ensure that their parameters are consistent with its previous observations. Most agents will not have enough information to verify the entire interaction. In this case, each agent can only preserve consistency; e.g. Labeler can make sure that it doesn't change the address, but Packer wouldn't know if it had. Local integrity verification can also be improved through the use of cryptographic signatures; for example, Packer could verify that the address field was signed by Merchant and not fabricated by the Labeler. Ultimately, Merchant can verify the final results, since it does know the original relationships.

Note that the consistency we describe here applies to the interaction level. An agent's observations are local and subjective, and so any correspondence to reality is left to the agent's internal logic. What matters at the interaction level is that once an observation has been declared (as a parameter binding), the subsequent messages consistently reflect that declaration. Thus, consistency applies even to abstract and subjective concepts such as preferences or opinions which may fluctuate wildly over time.

1.2.5 Flexibility

A secondary yet still important characteristic of multiagent systems is *flexibility*, which is related to the choices available to an agent during an enactment. Flexibility is desireable because it supports autonomy; the more flexible a system, the more an agent can act the way it desires.

To illustrate the concept, we will use examples from a commercial setting. Commerce usually involves multiple parties; at the very least a buyer and seller, but often other intermediaries such as a bank, escrow, delivery service, etc. A basic purchase scenario is given as a flow chart in Figure 1.4.

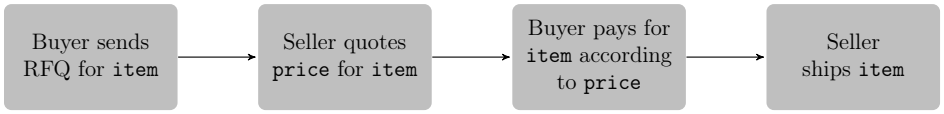


Figure 1.4: Example purchase flow

In this simple purchase scenario, there is only one possible sequence of events: Buyer

requests a quote, Seller gives a quote, Buyer pays for the item, and Seller ships the item, in that order. Because neither of the agents has any choices (besides whether to continue or abandon the interaction), it is completely inflexible.

One way we can give the agents more choices is by giving the agents more possible actions (*affordances*); e.g., allowing the buyer to reject the offered price, instead of always accepting it. However, adding affordances to an interaction changes its meaning, turning it into a different interaction.

Alternatively, we can give the agents more choices about when to exercise the affordances they already have. For example, this scenario may be made more flexible by giving Seller the option to ship the item *before* (or, parallel to) Buyer making payment, ae shown in Figure 1.5.

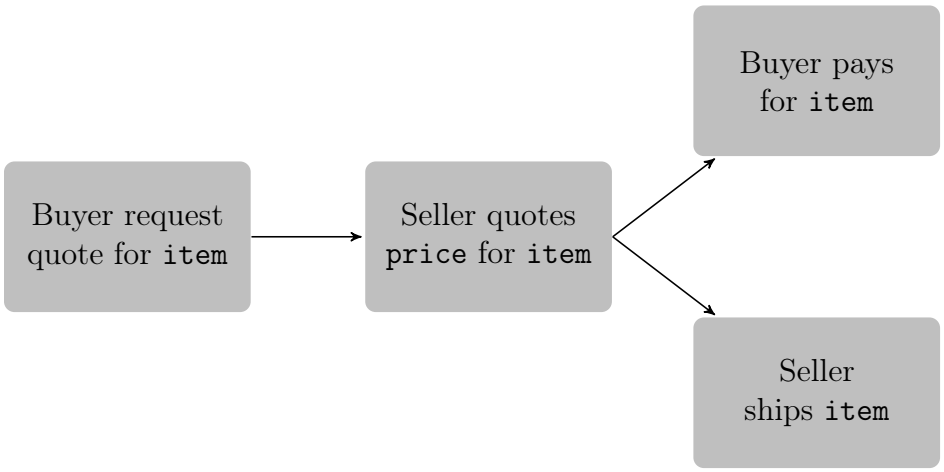


Figure 1.5: Flexible purchase flow; payment and delivery may happen in any order

Even though it does not have any more affordances, this scenario is more flexible because it lets the agents decide when to act.

1.2.6 Reliability

Healthcare also involves multiple distinct parties; primarily the patient and doctor, but also possibly nurses, pharmacists, lab technicians, etc. Each of these roles needs information from the others to perform their tasks, and there is often particular emphasis on accurate and timely results. Also, where commerce is infamous for targeted advertising based on collected personal information, *personal health information* (PHI) is a very private matter with many social norms and regulations against its unnecessary exposure, such as HIPAAChoi et al. [38].

As such, we frequently use the healthcare domain for examples involving reliability or

privacy, such as the following reliability example in Figure 1.6.

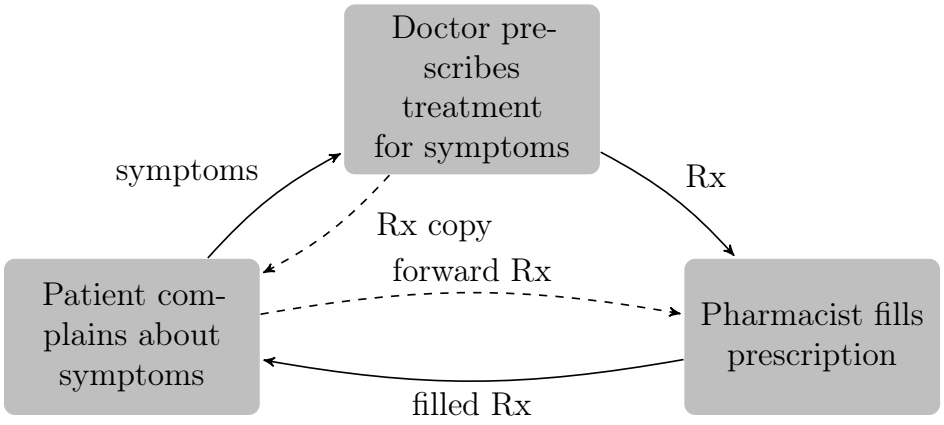


Figure 1.6: Treatment flow; Multiple steps required to fill prescription

The "happy path" is drawn with solid arrows. If patient does not receive a copy of the prescription, they can ask the doctor again; if they receive the prescription, but do not receive the medication within a few days, they can forward the prescription to the pharmacist. The messages used for recovery are drawn with dashed arrows.

This scenario succincly illustrates multiple roles that require information to perform their tasks, and how the other roles might detect a delay or error and resend that information in case it was lost.

1.2.7 Privacy

Another scenario illustrates privacy related norms:

1. A patient visits a doctor, generating new PHI
2. The hospital commits to storing a copy for their records
3. The hospital must share the data with the patient upon request
4. The hospital must also delete the information if requested by the patient
5. The patient can designate people who are authorized to view the data, such as a family member or researcher; the hospital *should* share data with them, but is prohibited from sharing it with anyone else.

6. However, in an emergency a doctor or specialist should be able to view the files, e.g. to avoid any allergic reactions. Such violations of the sharing prohibition should be possible, but also logged and audited afterward.

This scenario shows a few of the privacy-related norms that exist between the members of this simplified healthcare system. When formally represented, these norms can inform the agents how they should behave, and also be used to verify and regulate their behavior.

1.3 Scope and Contributions

The basic concepts of information-based decentralized and multiagent systems are not my own, but strongly based on my advisor's prior work on protocols and norms.

My contributions have been rather to expand and develop what is at the moment still a very new field, and begin to make steps toward practical implementation and application.

The works covered in this dissertation are listed in Table 1.1

Table 1.1: List of works covered in this dissertation

Work	Contribution	Publication
Atomicity	Verifiable correctness of compositions	AAMAS 2018 [47]
Refinement	Fast verification of modular protocols	AAMAS 2020 [46]
PoT	Significantly improved reliability in IoT case study	IEEE Internet Comput. 2020 [48]
Hercule	Efficient, non-destructive, declarative contracts	IEEE Internet Comput. 2021 [50]
Mandrake	Powerful application-level fault-tolerance policies	JAAMAS 2022 [53]
Pippi	Dynamic role binding and flexible composition	AAMAS 2022 [52]
Kiko	A decision-based programming model for cross-enactment reasoning	

1.4 Organization of this Dissertation

Chapter 2 introduces information protocols and BSPL with examples, and provides a formalization for reference.

Chapter 3 describes a simple application of protocols to the Internet of Things, and shows how developing a system using protocols can be simpler and more reliable than traditional methods.

Chapter 4 builds on some ideas started in the PoT project, developing the concept of application-level fault tolerance and supporting it with a basic programming model.

Chapter 5 defines a concept of atomicity for decentralized interactions between autonomous agents, derives algorithms for verifying atomicity, and evaluates them with several example protocols. Protocol atomicity requires that any component protocol must be able to complete once it is started.

Chapter 6 supports efficient verification of modular protocols through the concept of protocol refinement. Refinement is a fairly nuanced concept, capturing that one protocol's enactments are effectively an extended subset of the one it refines. The benefit of verifying refinement is that a composition does not need to be reverified if one of its components is substituted by a refinement; this means a simple composition can be verified, and then its components replaced by more complex refinements that would otherwise make verification of the whole intractable.

Chapter 7 extends the protocol specification language to support practical concerns of MAS instantiation. It introduces *metaprotocols*, whose parameters can refer to other protocols and their roles, to support negotiation and dynamic role binding.

Chapter 8 develops a more advanced programming model that supports cross-enactment decision making, and formalizes the model to show correctness and completeness. The programming model is *decision-based*, presenting available actions to the agent instead of making the developer duplicate the interaction logic in reactive message handlers.

Chapter 9 describes a declarative norm specification language supported by a blockchain-based norm state computation platform. Hercule aims to capture more of the relationships described by a traditional legal contract by representing them as declarative norms (e.g., commitments) rather than the imperative automation of smart contracts.

1.5 Sources

All sources are available under https://gitlab.com/masr. Several chapters have associated projects, but all of the code has been merged into https://gitlab.com/masr/bspl.

Chapter 2

Protocols

From our perspective, the interaction protocol is the fundamental specification of a multi-agent system. Lower-level specifications would be focused on individual agents and their implementations and not relate to the system as a whole. Higher-level specifications, such as norms, organizations, etc. build on information that would be communicated using a protocol.

2.1 Specification

BSPL [147] specifies protocol constraints in terms of causality, as motivated above, and *integrity* based on *key constraints* [75] on the information model. The key constraints capture the idea that in any protocol enactment a role may not send or receive conflicting information. Listing 2.1 illustrates BSPL's main concepts via a simple protocol.

Listing 2.1: Purchase with payment options

```
Purchase {
  roles B, S // Buyer, Seller
  parameters out orderID key, out product
  B ↦ S: PlaceOrder[out orderID]
  B ↦ S: Transfer[in orderID, out payment]
  B ↦ S: Credit[in orderID, out payment]
  S ↦ B: FillOrder[in orderID, in payment, out product]
}
```

The listing declares *Purchase* as the name of the protocol, two public roles B (buyer) and S (seller), and two public parameters order and product. Parameter order is adorned key, meaning that bindings of order uniquely identify enactments of *Purchase* and functionally determine the other parameters. Both parameters are adorned ⌜out⌝, meaning that their bindings are produced by enacting the protocol, that is, enacting the messages declared in it. The public parameters of a protocol serve as its interface and facilitate composition. *Purchase* declares four message schemas (the sequence of their listing is irrelevant). By convention, any key parameter of the protocol is a key parameter for any message in which it appears, though a message may have additional key parameters. Thus, *PlaceOrder* is from the buyer to the seller and its key is order. The ⌜out⌝ adornment means that sending *PlaceOrder* produces a binding for order. In *FillOrder*, order and payment are adorned ⌜in⌝, meaning that a seller may send an instance of *FillOrder* only if it has observed bindings for both. Parameters may in addition be adorned with ⌜nil⌝, indicating that the message cannot be sent if that parameter is bound.

A set of message instances (an *enactment*) completes a BSPL protocol if the messages cover all of its public ⌜out⌝ parameters; that is, each public ⌜out⌝ parameter is included in at least one message instance. For *Purchase*, *PlaceOrder* and *FillOrder* cover order and product. An enactment of *Purchase* is complete when an order has been both placed and filled. However, *FillOrder* has parameter payment adorned ⌜in⌝. Thus the buyer must produce

payment by sending either *Transfer* (representing a bank wire transfer) or *Credit* (representing payment via credit card) before the order can be filled.

Singh [149] formalizes BSPL and properties and gives verification techniques. Informally, a protocol is *enactable* if and only if a valid complete enactment exists. *Request Quote* is enactable, because the enactment consisting of *PlaceOrder* followed by *Transfer* and then *FillOrder* is valid and covers all public parameters of *Purchase*.

2.2 Safety

Informally, safety is the property that *bad things don't happen*. A protocol is *safe* if and only if it is impossible to produce *conflicting* bindings for a parameter in any enactment.

A potential safety violation would be if a buyer sent two instances of *Transfer* for the same order, one with a payment of $10 and one with a payment of $20. Such a violation can be easily avoided by the buyer based solely on its local knowledge, and so is not a flaw in the specification. A real safety violation occurs when two agents may produce conflicting bindings in an enactment. *Purchase* is safe. If the message *Gift* in Listing 2.2 were added to *Purchase*, it would become unsafe. Both seller and buyer could concurrently produce bindings for payment; that is, a nonlocal conflict would exist.

Listing 2.2: An unsafe extension to Purchase.

```
S ↦ B: Gift[in order, out payment, out product]
```

2.3 Liveness

Informally, liveness is the property that *good things do happen*. A protocol is *live* if and only if any enactment can progress to completion. *Purchase* is live: for any value of order, *PlaceOrder* may be sent followed by *Transfer* or *Credit* and then *FillOrder*, which would complete the *Purchase* enactment. An alternative specification without *Transfer* and *Credit* would not be live because without any way to produce a binding for payment, *FillOrder* cannot be sent and the enactment cannot be completed.

2.4 Composition

BSPL supports composition in a natural manner. A single message is an elementary protocol in BSPL. Thus, *PlaceOrder*, *Transfer*, and *Credit*, and *FillOrder* are all elementary protocols. *Purchase* composes these messages by *referring* to them. A BSPL protocol may have references

to one or more protocols. Protocol *RefinedPurchase* (Listing 2.3) replaces *Transfer* and *Credit* with composite protocols *RefinedTransfer* and *RefinedCredit*, respectively.

Listing 2.3: Refined purchase protocol

```
RefinedPurchase {
  roles B, S // Buyer, Seller
  parameters out order key, out payment, out product
  B ↦ S: PlaceOrder[out order]
  RefinedTransfer(B, S, in order, out payment)
  RefinedCredit(B, S, in order, out payment)
  S ↦ B: FillOrder[in order, in payment, out product]
}
```

2.5 Polymorphism

Polymorphism is the ability to select different objects on the basis of their interfaces. Normal programming languages may support polymorphism for functions on the basis of their parameter types, or for objects that inherit from a given base class on the basis of their subclass.

Since BSPL is a specification rather than programming language, there is no dispatching involved; however, polymorphism is a pattern in which multiple messages have the same name and schema but different adornments for some of the parameters. Because each parameter can be bound only once, a role must make a choice between multiple messages that bind the same parameter.

Listing 2.4: Polymorphic RFQ protocol.

```
Polymorphic-RFQ {
  roles B, S
  parameters out ID key, out item, out price
  B ↦ S: greet[out ID key]
  B ↦ S: rfq[out ID key, out item]
  S ↦ B: offer[in ID key, out item, out price]
  S ↦ B: offer[in ID key, in item, out price]
}
```

The *Polymorphic-RFQ* protocol in Listing 2.4 demonstrates simple polymorphism and choice: B has two ways to bind ID, and S has one way to bind price in response to each, both

named *offer* to make the polymorphism clear. This example shows how multiple mutually exclusive choices can be specified, and the power to bind a parameter can be given to different roles based on those choices.

2.6 Formalization

We reproduce here for reference the formal semantics of BSPL from [149], to avoid duplication in several chapters. This semantics is the original foundation for all of our work on BSPL, but some chapters have their own formalization of the same semantics using different definitions to facilitate different perspectives or algorithms.

For convenience, we fix the symbols by which we refer to finite lists of roles ($\vec{t}$), public roles ($\vec{x}$), private roles ($\vec{y}$), public parameters ($\vec{p}$), key parameters ($\vec{k} \subseteq \vec{p}$), ⌜in⌝ parameters ($\vec{p_I} \subseteq \vec{p}$), ⌜out⌝ parameters ($\vec{p_O} \subseteq \vec{p}$), ⌜nil⌝ parameters ($\vec{p_N} \subseteq \vec{p}$), private parameters ($\vec{q}$), and parameter bindings ($\vec{v}$, $\vec{w}$). Here, $\vec{p} = \vec{p_I} \cup \vec{p_O} \cup \vec{p_N}$, $\vec{p_I} \cap \vec{p_O} = \emptyset$, $\vec{p_I} \cap \vec{p_N} = \emptyset$, and $\vec{p_N} \cap \vec{p_O} = \emptyset$. And, t and p refer to an individual role and parameter, respectively. To reduce notation, we rename private roles and parameters to be distinct in each protocol, and the public roles and parameters of a reference to match the declaration in which they occur. Throughout, we use $\downarrow_x$ to project a list to those of its elements that belong to x.

Definition 1 captures BSPL protocols. A protocol may reference another protocol. The references bottom out at message schemas. Above, *Purchase* references *Transfer*. And, if a protocol were to reference *Purchase*, it would be able to reference (from its public or private parameters) only the public parameters of *Purchase*, not payment, which is a private parameter.

Definition 1*:* A *protocol* $\mathcal{P}$ is a tuple $\langle n, \vec{x}, \vec{y}, \vec{p}, \vec{k}, \vec{q}, F \rangle$, where n is a name; $\vec{x}$, $\vec{y}$, $\vec{p}$, $\vec{k}$, and $\vec{q}$ are as above; and F is a finite set of f references, $\{F_1, \ldots, F_f\}$. $(\forall i : 1 \leq i \leq f \Rightarrow F_i = \langle n_i, \vec{x_i}, \vec{p_i}, \vec{k_i} \rangle$, where $\vec{x_i} \subseteq \vec{x} \cup \vec{y}$, $\vec{p_i} \subseteq \vec{p} \cup \vec{q}$), $\vec{k_i} = \vec{p_i} \cap \vec{k}$, and $\langle n_i, \vec{x_i}, \vec{p_i}, \vec{k_i} \rangle$ is the public projection of a protocol $\mathcal{P}_i$ (with roles and parameters renamed).

Definition 2*:* The *public projection* of a protocol $\mathcal{P} = \langle n, \vec{x}, \vec{y}, \vec{p}, \vec{k}, \vec{q}, F \rangle$ is given by the tuple $\langle n, \vec{x}, \vec{p}, \vec{k} \rangle$.

We treat a message schema $\ulcorner s \mapsto r : m\ \vec{p}(\vec{k}) \urcorner$ as an atomic protocol with exactly two roles (sender and receiver) and no references: $\langle m, \{s, r\}, \emptyset, \vec{p}, \vec{k}, \emptyset, \emptyset \rangle$. Here $\vec{k}$ is the set of key parameters of the message schema. Usually, $\vec{k}$ is understood from the protocol in which the schema is referenced: $\vec{k}$ equals the intersection of $\vec{p}$ with the key parameters of the protocol declaration.

Below, let $\text{roles}(\mathcal{P}) = \vec{x} \cup \vec{y} \cup \bigcup_i \text{roles}(F_i)$; $\text{params}(\mathcal{P}) = \vec{p} \cup \vec{q} \cup \bigcup_i \text{params}(F_i)$; $\text{msgs}(\mathcal{P}) =$

$\bigcup_i \text{msgs}(F_i)$ and $\text{msgs}(s \mapsto r : m\ \vec{p}) = \{m\}$. Definition 3 assumes that the message instances are unique up to the key specified in their schema.

Definition 3*:* A *message instance* $m[s, r, \vec{p}, \vec{v}]$ associates a message schema $\ulcorner s \mapsto r : m\ \vec{p}(\vec{k}) \urcorner$ with a list of values, where $|\vec{v}| = |\vec{p}|$, where $\vec{v} \downarrow_p = \ulcorner \mathsf{nil} \urcorner$ if and only if $p \in \vec{p_N}$.

Definition 4 introduces a *universe of discourse (UoD)*. Definition 5 captures the idea of a *history* of a role as a sequence (equivalent to a set in our approach) of all and only the messages the role either emits or receives. Thus H^ρ captures the local view of an agent who might adopt role ρ during the enactment of a protocol.

Definition 4*:* A *UoD* is a pair $\langle \mathcal{R}, \mathcal{M} \rangle$, where $\mathcal{R}$ is a set of roles, $\mathcal{M}$ is a set of message names; each message specifies its parameters along with its sender and receiver from $\mathcal{R}$.

Definition 5*:* A *history* of a role ρ, H^ρ, is given by a sequence of zero or more message instances $m_1 \circ m_2 \circ \ldots$. Each m_i is of the form $m[s, r, \vec{p}, \vec{v}]$ where $\rho = s$ or $\rho = r$, and $\circ$ means sequencing.

Definition 6 captures the idea that what a role knows at a history is exactly given by what the role has seen so far in terms of incoming and outgoing messages. Here, 2(i) ensures that $m[s, r, \vec{p}(\vec{k}), \vec{v}]$, the message under consideration, does not violate the uniqueness of the bindings. And, 2(ii) ensures that ρ knows the binding for each $\ulcorner \mathsf{in} \urcorner$ parameter and not for any $\ulcorner \mathsf{out} \urcorner$ or $\ulcorner \mathsf{nil} \urcorner$ parameter.

Definition 6*:* A message instance $m[s, r, \vec{p}(\vec{k}), \vec{v}]$ is *viable* at role ρ's history H^ρ if and only if (1) $r = \rho$ (reception) or (2) $s = \rho$ (emission) and (i) $(\forall m_i[s_i, r_i, \vec{p_i}, \vec{v_i}] \in H^\rho$ if $\vec{k} \subseteq \vec{p_i}$ and $\vec{v_i} \downarrow_{\vec{k}} = \vec{v} \downarrow_{\vec{k}}$ then $\vec{v_i} \downarrow_{\vec{p} \cap \vec{p_i}} = \vec{v} \downarrow_{\vec{p} \cap \vec{p_i}})$ and (ii) $(\forall p \in \vec{p} : p \in \vec{p_I}$ if and only if $(\exists m_i[s_i, r_i, \vec{p_i}, \vec{v_i}] \in H^\rho$ and $p \in \vec{p_i}$ and $\vec{k} \subseteq \vec{p_i}))$.

Definition 7 captures that a *history vector* for a protocol contains a history for each role, and that all of the histories together are causally sound: a message is received only if it has been previously emitted [104].

Definition 7*:* Let $\langle \mathcal{R}, \mathcal{M} \rangle$ be a UoD. We define a *history vector* for $\langle \mathcal{R}, \mathcal{M} \rangle$ as a vector $[H^1, \ldots, H^{|\mathcal{R}|}]$, such that $(\forall s, r : 1 \leq s, r \leq |\mathcal{R}| : H^s$ is a history and $(\forall m[s, r, \vec{p}, \vec{v}] \in H^r : m \in \mathcal{M}$ and $m[s, r, \vec{p}, \vec{v}] \in H^s))$.

The progression of a history vector records the progression of an enactment of a multiagent system. Under the above causality restriction, a vector that includes a reception must have progressed from a vector that includes the corresponding emission.

Definition 8*:* A history vector over $\langle \mathcal{R}, \mathcal{M} \rangle$, $[H^1, \ldots, H^{|\mathcal{R}|}]$, is *viable* if and only if either (1) each of its element histories is empty or (2) it arises from the progression of a viable history vector through the emission or the reception of a viable message by one of the roles, that is,

$(\exists i, m_j : H^i = H'^i \circ m_j$ and $[H^1, \ldots, H'^i, H^{|\mathcal{R}|}]$ is viable).

The heart of our formal semantics is the *intension* of a protocol, defined relative to a UoD, and given by the set of viable history vectors, each corresponding to its successful enactment. Given a UoD, Definition 9 specifies a universe of enactments, based on which we express the intension of a protocol. We restrict attention to viable vectors because those are the only ones that can be realized. We include private roles and parameters in the intension so that compositionality works out. In the last stage of the semantics, we project the intension to the public roles and parameters.

Definition 9*:* Given a UoD $\langle \mathcal{R}, \mathcal{M} \rangle$, the *universe of enactments* for that UoD, $\mathcal{U}_{\mathcal{R},\mathcal{M}}$, is the set of viable history vectors, each of which has exactly $|\mathcal{R}|$ dimensions and each of whose messages instantiates a schema in $\mathcal{M}$.

Definition 10 states that the intension of a message schema is given by the set of viable history vectors on which that schema is instantiated, that is, an appropriate message instance occurs in the histories of both its sender and its receiver.

Definition 10*:* The intension of a message schema is given by:
$[[m(s, r, \vec{p})]]_{\mathcal{R},\mathcal{M}} = \{H | H \in \mathcal{U}_{\mathcal{R},\mathcal{M}}$ and $(\exists \vec{v}, i, j : H^s_i = m[s, r, \vec{p}, \vec{v}]$ and $H^r_j = m[s, r, \vec{p}, \vec{v}])\}$.

A (composite) protocol completes if one or more of subsets of its references completes. For example, *Purchase* yields two such subsets, namely, {*PlaceOrder*, *Transfer*, *FillOrder*} and {*PlaceOrder*, *Credit*, *FillOrder*}. Informally, each such subset contributes all the viable interleavings of the enactments of its members, that is, the intersection of their intensions. Definition 11 captures the *cover* as an adequate subset of references of a protocol, and states that the intension of a protocol equals the union of the contributions of each of its covers.

Definition 11*:* Let $\mathcal{P} = \langle n, \vec{x}, \vec{y}, \vec{p}, \vec{k}, \vec{q}, F \rangle$ be a protocol.
Let $cover(\mathcal{P}, G) \equiv \ G \subseteq F \,|\, (\forall p \in \vec{p_O} : (\exists G_i \in G : G_i = \langle n_i, x_i, p_i \rangle$ and $p \in \vec{p_i}))$ and $\mathcal{P}$'s intension, $[[\mathcal{P}]]_{\mathcal{R},\mathcal{M}} = (\bigcup_{\text{cover}(\mathcal{P},G)} (\bigcap_{G_i \in G} [[G_i]]_{\mathcal{R},\mathcal{M}}))\downarrow_{\vec{x}}$.

The UoD of protocol $\mathcal{P}$ consists of $\mathcal{P}$'s roles and messages including its references recursively. For example, *Purchase*'s UoD $U = \langle \{\text{B}, \text{S}\}, \{PlaceOrder, Transfer, Credit, FillOrder\} \rangle$.

Definition 12*:* The *UoD of a protocol* $\mathcal{P}$, $\text{UoD}(\mathcal{P}) = \langle \text{roles}(\mathcal{P}), \text{msgs}(\mathcal{P}) \rangle$.

Definition 13*:* A protocol $\mathcal{P}$ is *enactable* if and only if $[[\mathcal{P}]]_{\text{UoD}(\mathcal{P})} \neq \emptyset$.

Definition 14*:* A protocol $\mathcal{P}$ is *safe* if and only if each history vector in $[[\mathcal{P}]]_{\text{UoD}(\mathcal{P})}$ is safe. A history vector is *safe* if and only if all key uniqueness constraints apply across all its histories.

Definition 15*:* A protocol $\mathcal{P}$ is *live* if and only if each history vector in universe of enactments $\mathcal{U}_{\text{UoD}(\mathcal{P})}$ can be extended through a finite number of message emissions and receptions to a history vector in $\mathcal{U}_{\text{UoD}(\mathcal{P})}$ that is complete.

Chapter 3

Application: PoT

Current programming models for developing Internet of Things (IoT) applications conceive of an application as an orchestration. An orchestration even if physically distributed is logically centralized and thus ill-suited to the most interesting IoT applications, which involve multiple autonomous parties.

We contribute *Protocols over Things (PoT)*, a decentralized programming model for IoT applications that represents an IoT application via a *protocol* between the parties involved. Notably, PoT works over unordered delivery mechanisms such as UDP and supports application-level reliability mechanisms such as resending messages.

We realize PoT using Node-RED, a popular IoT framework, to show how PoT simplifies implementation and avoids errors. Further, we empirically demonstrate that by supporting application-level retry policies, PoT provides improved performance over network-level delivery guarantees.

This chapter is adapted from the paper *Protocols Over Things: A Decentralized Programming Model for the Internet of Things*, published in IEEE Computer in November 2020 [48]

3.1 Introduction

The Internet of Things (IoT) enables new applications that leverage the capabilities of *things*—Internet-accessible devices that sense or control their environment. Many such applications involve autonomous parties who share information from and control over things to effectively cooperate in achieving their respective goals. For example, a patient could share information with a healthcare provider to receive automatically dispensed medication. Patients should have control over their health information, and doctors over how they prescribe medications. More mundane IoT applications exhibit multiple parties too. For example, a smart home could involve an owner, one or more tenants, a power utility, and a security service. Likewise, in an enterprise, different suborganizations have different responsibilities and powers. That is, real-life applications involve multiple logical loci of control.

However, current programming models are geared toward a single locus of control: they violate the autonomy of all but one of the parties and don't jibe with things often being small and numerous. For instance, popular frameworks such as Node-RED (https://nodered.org/) and Eclipse Kura (https://www.eclipse.org/kura/) model an IoT application as an *orchestration* [128] that receives information from sensors, processes it in a workflow, and effects actions in the environment.

In contrast, we advocate a programming model called *Protocols over Things (PoT)* that is not only distributed physically but also *decentralized* [153]: It reflects everyone's autonomy and decouples their reasoning through asynchronous communication. Specifically, PoT captures a decentralized application via a *protocol* that specifies the communication constraints between the parties, abstracted as *roles*. PoT can employ IoT communication standards such as MQTT [121] and CoAP [140] as well as UDP [131].

Using a logistics scenario that we implement in Node-RED according to PoT, we show how a protocol specification enables the use of code generation tools and a generic communication adapter to simplify development. We focus specifically on how the adapter correlates messages according to the protocol, and compare PoT to existing approaches for message correlation in Node-RED.

We further demonstrate how PoT embodies the end-to-end principle [137] by supporting deployment over unreliable asynchronous communication mechanisms instead of relying on transport-layer delivery guarantees.

We experimentally demonstrate that a PoT implementation over UDP with application-level retry strategies compares favorably with the same implementation over MQTT.

3.2 Example IoT Scenario: Logistics

We adopt a warehouse logistics scenario [141], simplified to focus on the aspects relevant to decentralization: parties, their communications, and their decision making.

Figure 3.1 illustrates this scenario conceptually. Here, MERCHANT, WRAPPER, LABELER, and PACKER are autonomous parties. The arrows indicate information flows: each arrow corresponds to a message, with the label indicating the new information communicated by that message.

The interaction is initiated by the MERCHANT, who has internal knowledge of one or more items to deliver to an address. The items and address are correlated using a purchase order (PO) ID, though the ID is not shown in the diagram for simplicity. Although the address and items could originate from a customer, from the perspective of the other agents MERCHANT is the source of the information.

MERCHANT sends the address to LABELER, who generates the appropriate shipment label to be affixed to the shipping box. LABELER sends the generated label information to PACKER. MERCHANT sends information about individual items to the WRAPPER, who selects an appropriate wrapping for shipping the item (e.g., paper for durable items and bubble wrap for fragile ones) and notifies PACKER. PACKER affixes the shipment label to a box, packs each item with the same PO ID into the box, and notifies MERCHANT of the packing status of each item.

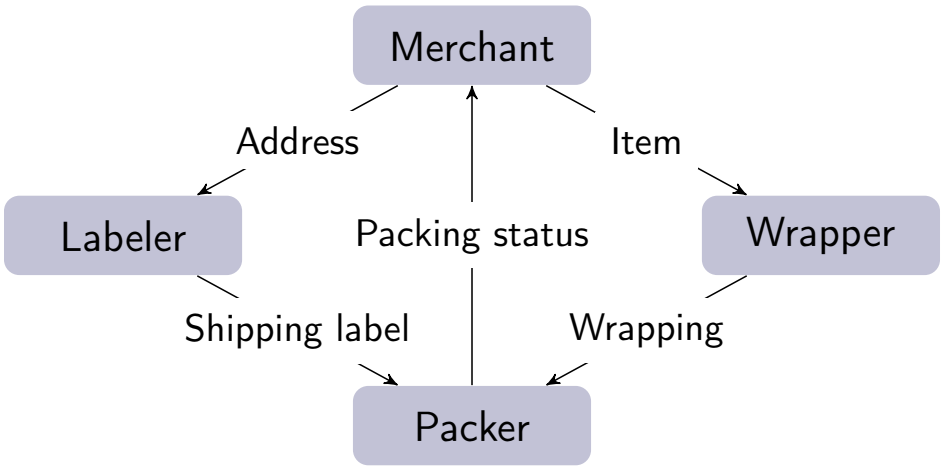


Figure 3.1: Conceptual model of information flow in the logistics scenario: each arrow indicates a message labeled by the new information it communicates. Merchant, Wrapper, and Packer send separate messages for each item, but Labeler only creates one label per PO.

An important nuance is that the messages only communicate *information*. Any physical

actions, such as printing a label or wrapping an item, can only be requested or described by a message, not actually performed. Thus, for example, the item shown in the information flow diagram only refers to the information about the item (e.g. its stock keeping unit), not the transfer of the physical item itself.

Each party uses its own internal logic to decide when and what information to communicate. For example, WRAPPER may select a wrapping based on current inventory and cost and LABELER may select a shipping label based on the speed and cost of the shipping service. Further, a party may choose to delay or not to send a message; e.g., WRAPPER may hold on to an item until it has the appropriate wrapping, and LABELER may discard invalid addresses. In general, the parties work concurrently and asynchronously based on information available to them.

3.3 Node-RED

Node-RED is an interactive programming and execution environment for IoT applications. A Web interface presents a palette of function blocks called *nodes*. A user can construct a *flow*—technically an orchestration—by connecting nodes via virtual *wires* that run from an output port of one node to the input port of another. A flow typically starts with one or more nodes that sense information from the environment and ends with some action in the environment.

Figure 3.2 illustrates a flow. The type of each node is indicated by its color and icon. Each node has an informal name.

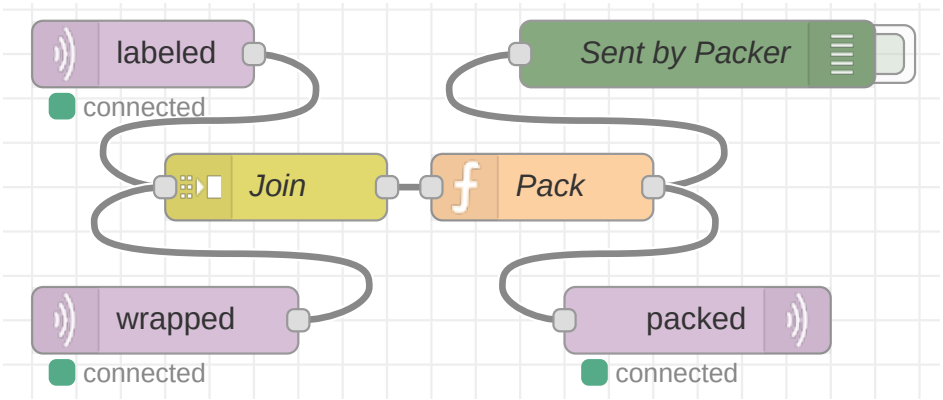


Figure 3.2: A simple Node-RED flow using `Join` to combine messages

In Figure 3.2, the nodes named `labeled` and `wrapped` are MQTT subscriptions to the

topics *Labeled* and *Wrapped*, respectively. The output ports of both `labeled` and `wrapped` are connected to the input port of the `Join` node, meaning that received messages are passed on to the `Join` node. A join node aggregates multiple messages together; in this example, one message from each of the MQTT topics. `Pack` is a function node that applies custom JavaScript code to the aggregates produced by `Join`. The output port of `Pack` is connected to two other nodes, which means that each of its outputs is copied and passed to those nodes separately. The node named `Sent by Packer` logs its inputs to the developer console, to facilitate debugging. The node named `packed` is an MQTT node that publishes any input it receives to the *Packed* topic.

3.4 Challenges in Programming IoT Applications as Orchestrations

Node-RED (see Sidebar) epitomizes the orchestration approach to programming IoT applications. How might one implement a decentralized IoT application using flows? Each party's computations may be captured in a separate flow called the party's *endpoint* that communicates with the endpoints of other parties via messages over e.g. MQTT.

Figure 3.3 illustrates the resulting architecture schematically. Notably, there is no representation of the application that captures its decentralized nature except as endpoints implemented in Node-RED.

Figure 3.3: Endpoint-oriented architecture for IoT applications.

There is another shortcoming in the architecture in Figure 3.3. To interoperate, the parties need a specification of the structures of the messages they communicate. Such a specification lies strictly outside Node-RED but is crucial to decentralization. Lacking such a specification, the endpoints would become tightly coupled.

Suppose such a specification is available, as is common in practice. For example, Listing 3.1 specifies the structures of the messages alluded to in Figure 3.1. Sadly, the message

Listing 3.1: Structures of messages exchanged between the endpoints in the logistics scenario.

```
//Sender to Receiver: MessageName(parameter 1,...,parameter n)

Merchant to Labeler: RequestLabel(orderID, address)

Merchant to Wrapper: RequestWrapping(orderID, itemID, item)

Wrapper to Packer: Wrapped(orderID, itemID, item, wrapping)

Labeler to Packer: Labeled(orderID, address, label)

Packer to Merchant: Packed(orderID, itemID, wrapping, label, status)
```

structures in Listing 3.1 are inadequate for capturing the scenario; particularly its correlation requirements. PACKER needs application-level knowledge to correlate the items and shipping label for the same PO and pack them correctly, since it receives them separately.

Since Node-RED as a general-purpose platform provides only low-level programming facilities unaware of the application semantics, an implementation of correlation using standard Node-RED features will at best be ad hoc and difficult to maintain. Below, we describe three increasingly sophisticated approaches for implementing the correlation necessary for PACKER using standard Node-RED facilities, to explain their advantages and limitations.

3.4.1 Join-Based

Figure 3.2 in fact shows an endpoint for PACKER. Recall that `Join` in the figure produces an aggregate consisting of one message from each of the *labeled* and *wrapped* MQTT subscriptions.

Unfortunately, `Join` aggregates the messages solely based on their sequence in each topic—and ignores their content—producing incorrect aggregations if the label and item do not actually correlate.

For example, if PACKER receives *Labeled* with orderID O_1 and *Wrapped* with orderID O_2, `Join` would mistakenly aggregate them and direct the wrapped item to the address from O_1.

Further, `Join` "consumes" each incoming message so it can appear in at most one output. Hence, only a single item can be associated correctly with a label. If any PO contains multiple items, `Join` matches the remaining items with labels from subsequent POs, causing all subsequent items to correlate with the wrong label.

3.4.2 Wait-Paths-Based

Unlike `Join`, `wait-paths` (a community plugin node) supports a correlation field, here orderID. An explicit correlation field provides some support for correct matching.

However, `wait-paths` also "consumes" what it correlates and associates at most one item in a PO with the PO's label. Thus, excess items are silently held (and eventually dropped) waiting for another matching label.

3.4.3 Custom

The Node-RED `function` node enables custom implementations. There are no limitations on the correctness or efficiency of a custom implementation, but such implementations would be low-level and complex.

3.4.4 Shortcomings Identified

First, although integrity constraints on correlation are explicit in the informal description of the scenario in Section 3.2, Listing 3.1 omits them. Representing constraints would enable reasoning about them computationally.

Second, traditional implementations entangle decision making internal to an endpoint (e.g., PACKER deciding whether, when, and what item to pack) with public communication constraints, e.g., to correlate items and labels.

Third, the endpoints may become inadvertently coupled by accounting for each other's implementation idiosyncrasies, e.g., by accommodating one box per order but failing when the order is split.

3.5 The PoT Programming Model

The foregoing shortcomings motivate a new programming model. Whereas traditional approaches focus on endpoints (as Section 3.4 shows), PoT begins with a *protocol*: a specification that captures the interactions in an application via constraints on communications between the application's endpoints.

Figure 3.4 shows our protocol-based IoT application architecture. Here, an *agent* is an endpoint that adopts a role in a protocol and sends and receives messages to and from other agents in accordance with the protocol.

The *Decision Making* component implements an agent's private reasoning, and relies upon the *Protocol Adapter* to handle incoming and outgoing messages. The *Adapter* verifies

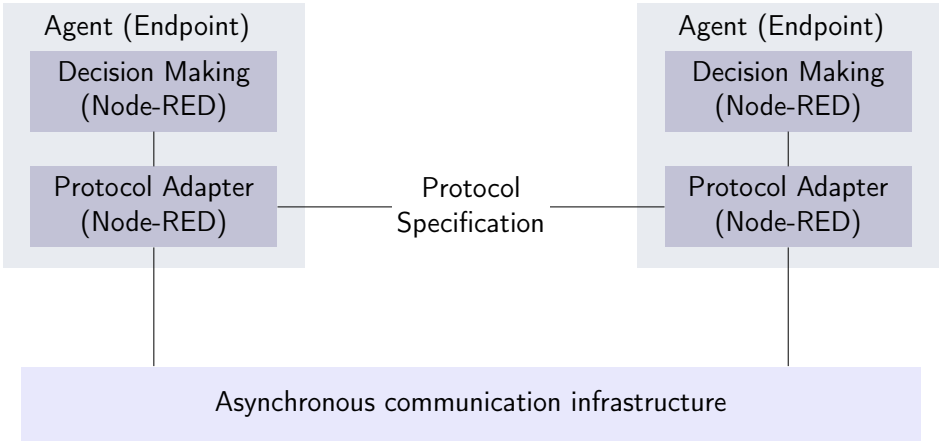


Figure 3.4: The PoT architecture for IoT applications.

whether messages respect the relevant causality and integrity constraints of the supplied protocol specification. If any messages are noncompliant, they may be logged (if debugging is enabled) but are otherwise discarded.

Note that although discarding noncompliant messages may seem to reduce agent autonomy by denying the opportunity to handle those messages, the restrictions imposed are exactly those imposed by adopting the protocol: only properly formed, enabled messages may be sent or received. It would not be unreasonable an implementation to offer optional error handling so that the agent could, for example, automatically notify someone to resolve the issue. However, noncompliant messages should not be added to the agent's history (integrity violations would introduce contradictory bindings). Nor should the agent be crashed by raising an exception, or it be very vulnerable to denial-of-service attacks by sending malformed messages.

3.5.1 Specifying Protocols

PoT adopts *BSPL*, the Blindingly Simple Protocol Language [147]. Listing 3.2 specifies our logistics scenario in BSPL.

Formalizing protocols yields three benefits. First, protocols separate decision making from interaction. Second, with BSPL specifically, integrity constraints capture correlation requirements declaratively. For example, the bindings of wrapping and label in a *Packed* message must be consistent with the binding of ⟨orderID, itemID⟩ (*Packed*'s key). Finally, as we discuss further in the next section, a clear specification of the communication constraints supports the implementation of agents by enabling the automatic generation of agent skeletons

Listing 3.2: The *Logistics* Protocol

```
Logistics {
 role Merchant, Wrapper, Labeler, Packer

 parameter out orderID key, out itemID key, out item, out status

 Merchant -> Labeler: RequestLabel[out orderID key, out address]
 Merchant -> Wrapper: RequestWrapping[in orderID key, out itemID
    key, out item]

 Wrapper -> Packer: Wrapped[in orderID key, in itemID key, in item,
    out wrapping]

 Labeler -> Packer: Labeled[in orderID key, in address, out label]

 Packer -> Merchant: Packed[in orderID key, in itemID key, in
    wrapping, in label, out status]
}
```

that enforce those constraints with the help of a generic protocol adapter.

3.6 Implementing PoT agents

Figure 3.5 shows the implementation of a PACKER agent according to the PoT model. The flow is separated into four layers: Incoming, Message Reasoning, Retry Policy, and Outgoing. The Incoming and Outgoing layers jointly constitute the protocol adapter component of the architecture, but are separated because of the one-way flow programming style of Node-RED. Message Reasoning and Retry Policy together constitute the decision-making component; they handle new messages and duplicate messages, respectively.

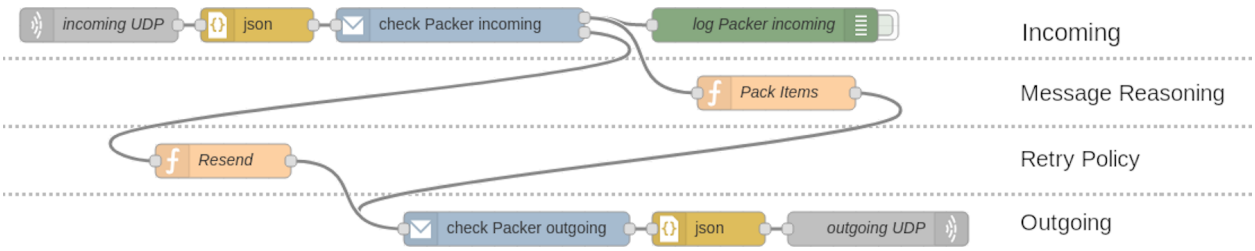


Figure 3.5: PoT version of Packer flow, showing the architectural layers.

Given a protocol, our tooling generates a *skeleton* consisting of the Incoming and Outgoing layers for every role in it. For every agent playing a role, the Message Reasoning and Retry Policy parts are added by the agent developer. We describe each layer below.

Incoming

This layer implements the reception half of an agent's protocol adapter.

Nodes `incoming UDP` and `json` receive messages over UDP and decode them from JSON, respectively. Node `check Packer incoming` has access to the agent's *history*, which is a collection of messages that the agent has already observed. If an incoming message is not a duplicate and satisfies integrity, `check Packer incoming` adds it to the history and outputs it via the top output port. If the message is a duplicate, the node outputs it via the bottom output port without adding it to the history again. Integrity checking on reception is a defense against agents who may send messages in violation of integrity.

Node `log Packer incoming` optionally logs received messages to the developer console.

Message Reasoning

This layer captures an agent's reasoning, as the developer deems fit, as it reacts to received messages and potentially generates outgoing messages. In Figure 3.5, the PACKER implementation uses a single function node `Pack Items` to send *Packed* messages in reaction to received *RequestLabel* and *RequestWrapping* messages. Outgoing messages may also be produced in response to other events; for example, MERCHANT could initiate *Logistics* based on POs in a database.

Retry Policy

PoT also enables support for agent-specific *retry policies* for handling failure cases such as lost messages, and reacting to duplicate incoming messages. If an agent expects to receive a future message in response to one that it sends, such as MERCHANT expecting a *Packed* message for each item, it can detect possible message loss when those expectations are not met within a specified time. Our MERCHANT implementation repeatedly resends the *RequestLabel* and *RequestWrapping* messages for an item every second until the corresponding *Packed* message is received.

However, resending a message only addresses the loss of that message; the loss of *Wrapped*, *Labeled* or *Packed* would also prevent MERCHANT from receiving *Packed*. To further support recovery, the other agents can resend relevant messages when they receive a duplicate. Our Node-RED implementation supports this pattern by providing a second output port on

the reception checking node for handling duplicate messages. In our implementation of PACKER, the `Resend` node handles duplicate *Labeled* or *Wrapped* messages and resends the corresponding *Packed* message if available.

PoT's support for agent retry policies reflects the end-to-end principle [137], providing application-specific assurance of correctness, without the overhead of redundant protections in lower layers of the infrastructure.

Outgoing

This layer implements the emission half of an agent's protocol adapter. The layer checks every outgoing message, adding to the agent's history and sending over the network those that satisfy the relevant causality and integrity constraints, and dropping those that do not. In Figure 3.5, the `check Packer outgoing` node performs the checking. To support retry policies, the node allows sending duplicates; however, they are not added to the agent's history.

Node `json` encodes the message in JSON and `outgoing UDP` sends it via UDP to the specified recipient.

3.7 Evaluation

To demonstrate the advantages of PoT and its support for application-level retry policies, we evaluated four variations of our PoT-based implementation of the *Logistics* scenario:

Plain UDP. PoT over UDP, without retries.
UDP+Retry. PoT over UDP, with the above-mentioned retry policies.
MQTT0. PoT over MQTT at QoS level 0.
MQTT2. PoT over MQTT at QoS level 2.

MQTT runs over TCP. QoS level 0 and level 2 mean that MQTT guarantees a message will be delivered at most once and exactly once, respectively.

We set up our experiment to compare the variations by their average *throughput*, defined as the rate at which enactments complete. We ran the experiment on a single Linux computer, using tc-netem to simulate random packet loss. For each variation and each level of packet loss, we ran ten iterations of one minute each. In each iteration, MERCHANT generates a new PO every 5 milliseconds (simulating external customers), with the POs having a uniformly distributed number of items between 1 and 4. Enactment timeouts were disabled, and the merchant's retry policy was configured to resend its messages every second until it received the corresponding *Packed* message.

Hypothesis *Under random packet loss, UDP+Retry completes more enactments over a given duration than both MQTT variations and Plain UDP.*

Figure 3.6 shows the results of our experiment as a graph of the average throughput (enactments completed per second) over packet loss probability. Each plot shows the mean of ten iterations, with error bars showing the sample standard deviation.

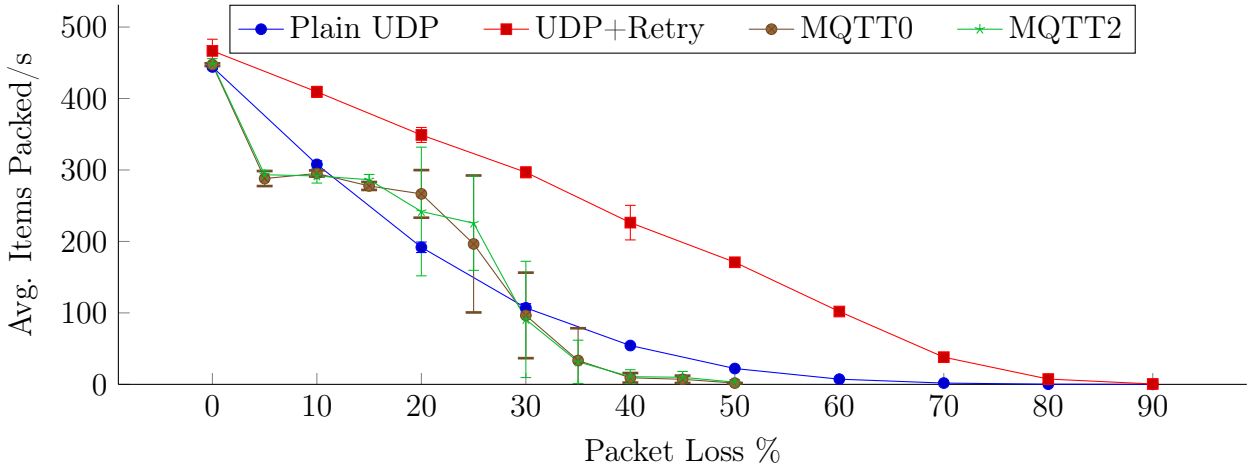


Figure 3.6: A comparison of different retry policies under different packet loss rates with respect to transaction completion.

The results support our hypothesis. Plain UDP drops rapidly and consistently in throughput, since any message loss prevents an enactment from completing. In contrast, UDP+Retry has a consistently higher throughput under packet loss. Since Plain UDP has less overhead and both initiate enactments at the same rate, the difference is in their reliability: UDP+Retry recovers some enactments that would otherwise be lost.

Both MQTT variations suffer an initial drop in throughput at low packet loss, possibly due to TCP backoff. Their throughput changes very little up to 20% loss, demonstrating reliability superior to Plain UDP and the benefits of TCP's ability to batch multiple messages in one packet. However, both MQTT variations drop precipitously in throughput starting at 20% and fail to complete any enactments after 50% loss. At every packet loss rate, UDP+Retry has a higher throughput than either MQTT variation. Since TCP on Linux defaults to retrying packet transmissions for longer than the one-minute sample duration, and UDP+Retry never gives up, the difference in throughput is not due to lost enactments but efficiency. UDP+Retry is more efficient than MQTT's infrastructure-level retry solutions because it can detect losses without using acknowledgments and no enactment blocks another.

3.8 Conclusion

PoT differs from existing work on protocol languages and deriving endpoint representations, e.g., [14, 70], in its use of an information-based representation for protocols. The PoT representation is uniquely compatible with asynchrony and naturally addresses correlation problems. PoT is also of practical value as it enables the generation of agent (endpoint) skeletons in Node-RED and, through them, the enforcement of protocol constraints.

PoT accommodates heterogeneity; an application may involve both PoT agents (implemented as described in Section 3.6) and non-PoT agents. PoT agents are guaranteed to be compliant with the protocol, but non-PoT agents can also be compliant. If a non-PoT agent is not compliant, however, the PoT agent's adapter provides some protection by recognizing and rejecting incoming messages that fail integrity.

PoT's basis in information protocols supports application-level retransmission of messages—a fault tolerance mechanism. PoT could be extended to support a wider and more sophisticated variety of such mechanisms. For example, retransmission could be adaptive: an agent could learn expected arrival times for messages from other agents relative to other messages and use that knowledge to determine when to resend a message. Future extensions should improve ease of use, flexibility, and robustness against various faults or malicious behavior until decentralized programming is no longer considered difficult but properly understood as the best way to improve fault tolerance and create scalable dynamic systems.

PoT opens up the possibility of expressing deeper expectations between parties, such as *norms* [41]. Just as PoT tackles expectations about the information exchanged, norms would capture the meaning of that information, such as whether one party committed to doing something for another or prohibited the other party from doing something. We could evaluate whether such a commitment or prohibition was satisfied or violated and whom to hold to account for it. Such models can enable greater social intelligence, paving the path to superior programming models for autonomy, flexibility, and reusability.

Chapter 4

Fault Tolerance

We conceptualize a *decentralized* software application as one constituted from *autonomous* agents that communicate via *asynchronous* messaging. Modern software paradigms such as microservices and settings such as the Internet of Things evidence a growing interest in decentralized applications. Constructing a decentralized application involves designing agents as independent local computations that coordinate successfully to realize the application's requirements. Moreover, a decentralized application is susceptible to faults manifested as message loss, delay, and reordering.

We contribute *Mandrake*, a programming model for decentralized applications that tackles these challenges without relying on infrastructure guarantees. Specifically, we adopt the construct of an *information protocol* that specifies messaging between agents purely in causal terms and can be correctly enacted by agents with nothing more than unreliable, unordered transport. Mandrake facilitates (1) implementing protocol-compliant agents by introducing a programming model; (2) transforming protocols into fault-tolerant ones with simple annotations; and (3) a policy language that makes it easy to implement fault-tolerance in agents based on the capabilities in protocols.

Mandrake's significance lies in demonstrating a straightforward approach for constructing decentralized applications without relying on coordination mechanisms in the infrastructure, thus achieving some of the goals of the founders of networked computing from the 1970s.

This chapter is adapted from the paper *Mandrake: multiagent systems as a basis for programming fault-tolerant decentralized applications*, published in the Journal of Autonomous Agents and Multi-Agent Systems in February 2022 [53]

4.1 Introduction

We conceptualize a *decentralized* software application as one that involves two or more *autonomous* endpoints, or *agents*. We assume a communication infrastructure based on *asynchronous* messaging (where messages can be received at any time after sending, with no ordering guarantees) since that reduces coupling between the agents. Although our approach could be realized on more restrictive infrastructures, for generality, we focus on asynchrony without ordering guarantees.

We observe two motivations for decentralization. First, decentralization reflects autonomy in the overarching social architecture of an application [42]. Applications in domains such as e-business, finance, and healthcare span multiple autonomous real-world parties. Second, decentralization reflects loose coupling in the technical architecture. The *microservices* paradigm [118] supports developing, deploying, and scaling microservices independently of each other. The Internet of Things (IoT) motivates decentralization in both social and technical terms [98, 153] by bringing forth interactions between devices owned by two or more parties and by technologies such as fog computing that distribute information processing and storage [126].

However, getting decentralized applications right is extremely difficult. Asynchrony and faults make coordinating the computations of an application challenging. This challenge is exacerbated when, as in open applications, agents represent autonomous real-world parties and are independently constructed. Further, autonomy motivates flexibility in interactions [173]; however, flexibility itself is in tension with ease of coordination [13, 167].

In conventional approaches for building distributed systems, coordination, including consistency and fault tolerance, is addressed in an application's communication infrastructure via guarantees for reliable and ordered delivery of messages. TCP, e.g., implements complex reliability and ordering mechanisms. Already there is a move away from TCP in several domains for performance reasons. E.g., in the IoT, lightweight communication services such as CoAP [140] are preferable to TCP-based protocols such as HTTP and AMQP [5]. Performance though is not the only reason move away from complex communication services. There is a fundamental systems principle, the end-to-end model [137], that argues against reliability and ordering guarantees fixed in the infrastructure. The argument is twofold. One, such guarantees turn out to be redundant considering what must be implemented at the application level to make an application robust. Two, and worse, they interfere with *application meaning* [36], which lies in the application domain and refers to how users make distributed decisions in practice. Specifically, infrastructure guarantees hamper flexibility by preventing agents from observing some sequences of events (and thus computations) that are

legitimate from the application perspective and that agents could benefit from in practice. Traditional fault tolerance, in particular, is not meaningful, because it is transparent to the application and neither influences nor considers agent decisions. Naturally, a communication service that has redundant features and interferes with application reasoning is likely to result in poorer performance than if concerns of consistency and fault tolerance were left to the application.

4.1.1 Contributions

This paper addresses the challenges of building robust decentralized applications in a manner compatible with the end-to-end model. In particular, we contribute *Mandrake*, a set of techniques, that demonstrates how a fault-tolerant decentralized application can be realized as a multiagent system sitting on top of an infrastructure that provides neither ordering nor reliability guarantees. Mandrake is based on the insight that to support meaning, a decentralized application must be modeled via a declarative information protocol [147]. An information protocol captures the interactive part of application meaning by constraining an agent's emission of messages based purely on its local information state, which is essentially the agent's history of communications. Receptions, by contrast, are unconstrained. Specifically, an agent may receive messages sent by others in any order. An agent's decision making (which would generally rely both on its local state and internal state) is up to the agent's implementation. Taken together, the possibility of an agent receiving messages in any order and processing them in accordance with its own decision making are crucial to realizing application meaning.

Such an application architecture provides an opportunity to introduce meaningful fault tolerance—that is, fault tolerance based on information available to agents and incorporated in their decision making. In particular, it enables understanding a fault as the violation of an *expectation* of an agent to receive some information from others and fault tolerance as what agents do to prevent or handle the violations.

Concretely, Mandrake makes the following technical contributions.

- A transformation language for extending protocols with common communication patterns that enable fault tolerance, such as forwarding information.
- A high-level programming model that enables realizing agents that enact information protocols. The programming model motivates a specific agent architecture and is realized via a concrete API for implementing agents.
- A declarative language for specifying fault tolerance policies, to simplify the development

of robust, application-specific agents.

Mandrake builds upon recent contributions that take advantage of information protocols to enable application-level fault tolerance. Fault tolerance based on information protocols was first proposed in [45]. Christie et al. [48] demonstrate that information protocols enable constructing agents that can recover from message loss; however, the programming model support for constructing such agents was limited to message validation. It provides neither an API nor a language for specifying fault tolerance policies. Bungie [49] informally presents fault-tolerance patterns that can be applied to information protocols to obtain more robust protocols. Mandrake formalizes those intuitions in a transformation language.

4.1.2 Organization of the Paper

Section 4.2 describes a health care scenario, and gives a protocol specification. Section 4.3 describes the need for fault tolerance, specifically at the application level, and describes the annotations we introduce for transforming protocols to support fault tolerance. Section 4.4 describes our proposed programming model, including architecture, agent programming API, and a declarative fault tolerance policy specification language. Section 4.5 gives a conceptual evaluation and comparison with the Jason agent programming framework. Section 4.6 contrasts Mandrake with related work in the areas of fault tolerance, services, and MAS development. Section 4.7 summarizes our conclusions and identifies several directions for future work.

4.2 Scenario Specification

Our running example is a simple medical treatment scenario: The patient may send a complaint to the doctor describing the symptoms they have been experiencing. In response, the doctor may send a prescription to a pharmacist, who fills the prescription for the patient.

Listing 4.1 gives a specification of a basic prescription process at the interaction level written in the information-based protocol language BSPL [147].

Listing 4.1: Treatment Protocol.

```
Treatment {
  roles Patient, Doctor, Pharmacist
  parameters out sID key, out symptom, out done
  private Rx

  Patient -> Doctor: Complaint[out sID key, out symptom]
  Doctor -> Patient: Reassurance[in sID key, in symptom, nil Rx, out
     done]
  Doctor -> Pharmacist: Prescription[in sID key, in symptom, nil
     done, out Rx]
  Pharmacist -> Patient: FilledRx[in sID key, in Rx, out done]
}
```

An information protocol specifies an interaction in terms of messages between roles, the information those messages carry, and causality and integrity constraints on those information parameters.

The *Treatment* protocol involves three roles (PATIENT, DOCTOR, and PHARMACIST) and consists of four messages between these roles. Each message has a payload of parameters that can be adorned ⌜in⌝, ⌜out⌝, or ⌜nil⌝, and of which at least one must be a key. These parameter adornments define causality constraints on the basis of whether or not a binding for that parameter is known. Parameter bindings occur within the context of an enactment identified by the bindings of key parameters; within an enactment, each parameter may be bound only once for integrity. All constraints are on the emission of messages; message reception is unconstrained except that a message must be emitted to be received.

Parameters adorned ⌜out⌝ create new bindings and so must not already have a binding when the message is sent; for example, PATIENT creates a new binding for symptom when they send *Complaint*. Parameters adorned ⌜in⌝ identify dependencies, and their binding must be known before the message can be sent; for example, DOCTOR cannot send *Prescription* until they know the binding of symptom. Finally, ⌜nil⌝ parameters *prevent* a message from being sent if a binding is known; for example, the ⌜nil⌝ Rx in *Reassurance* prevents DOCTOR from sending it if they have already sent *Prescription* (which binds Rx).

PATIENT can begin an enactment of *Treatment* by sending *Complaint*, which has no ⌜in⌝ parameters and thus no dependencies. Sending *Complaint* introduces new bindings for the sID and symptom parameters because they are adorned ⌜out⌝. Once DOCTOR observes the binding for sID and symptom, they have enough information to either send *Reassurance* or *Prescription*. DOCTOR chooses between *Reassurance* and *Prescription*: *Reassurance* binds

done which blocks *Prescription*, and *Prescription* binds Rx which blocks *Reassurance*. A protocol enactment is complete when all its public ⌜out⌝ parameters are bound, so *Reassurance* completes the enactment of *Treatment* when it binds done. If DOCTOR chooses to send *Prescription* instead, the enactment is completed when PHARMACIST sends *FilledRx* and binds done.

Correlation based on key parameters makes duplicated messages idempotent; only the first reception results in a new observation. Correlation is the basis for handling multiple complaints concurrently; each *Complaint* message has its own sID, and is thus either a duplicate (and ignored) or the beginning of a distinct interaction. Importantly, key-based correlation makes information protocols robust against message reordering, since messages are interpreted by their contents instead of their order.

Key parameters model semantic identifiers, that is, meaningful identifiers for correlation that are part of the application domain. The use of semantic identifiers for correlation and idempotence is not novel, but it is commonplace to rely on arbitrary syntactic identifiers instead—for example, AMQP's correlation ID is a single parameter external to the message (and thus not meaningful within the interaction) for correlating request/response pairs.[44]. De Graauw [60], e.g., notes the limitations of using reliable messaging protocols for Web services and advocates using semantic identifiers for idempotence and fault tolerance. BSPL provides language features for modeling semantic identifiers, enabling programming models such as Mandrake that can exploit them for correlation.

4.3 Fault Tolerance

Under ideal conditions, the above protocol sufficiently specifies the system: each participant is enabled to perform the actions necessary for filling the prescription, yet constrained from doing things they shouldn't, such as producing inconsistent bindings.

However, if any of the messages is lost, or if an agent does not perform an expected task (e.g., DOCTOR fails to send the prescription), then the prescription will not be filled. We want to *ensure* that the patient will eventually get their medication, even if some things go wrong.

This concept of maintaining correctness and progress despite faults is *fault tolerance*. However, most work on fault tolerance, which we shall call *infrastructure-level fault tolerance*, has been focused on addressing the sources of faults rather than achieving the desired outcome. This can be seen from the cause-based perspective of the most common fault tolerance taxonomies [10]. The resulting methods are independent of the application, and either focused on specific errors (e.g., memory faults) or generic solutions (e.g., redundant components with automatic failover). Instead, we focus on *application-level fault tolerance*,

using higher-level concepts to work directly toward the desired outcome.

Take PATIENT's submission of a complaint for example. An infrastructure-level approach might be to use TCP, which automatically resends packets until they are acknowledged to recover from packet loss during transmission. However, infrastructure-level acknowledgments are basically meaningless, because they are not produced intentionally by the agent. Such acknowledgments cover the smallest possible amount of progress, mere transmission, without verifying that the agent has received, accepted, or processed the messages.

4.3.1 Application-Level Fault Tolerance

An application-level approach should both reassure that progress is being made and help the agent recover from problems. For example, the DOCTOR could forward a copy of the *Prescription* to the PATIENT. The forwarded prescription both proves that progress has been made, and enables new recovery options for the PATIENT, as the PATIENT can now forward the *Prescription* directly to the PHARMACIST. If all goes well the agents directly interact with each other for maximum efficiency, but if something goes wrong the PATIENT knows where and how to resume the enactment.

Meaningful communication is helpful for fault tolerance, because it supports and informs agent expectations based on the protocol. After submitting a complaint, PATIENT expects to receive either reassurance or a filled prescription. If this expectation is not met within a reasonable time (as determined by PATIENT), PATIENT may assume that something has failed and attempt recovery, perhaps by resending their *Complaint* to the DOCTOR, or by more directly pursuing the fulfillment of their prescription and contacting the PHARMACIST.

4.3.2 Approach Overview

After identifying the completion requirements and agent expectations from the protocol, we propose using the following communication patterns to add robustness to a MAS and support fault tolerance:

1. (*Remind*) Retry sending messages to recover from failed expectations;

   e.g., remind DOCTOR of *Complaint*.

2. (*Checkpoint*) Forward intermediate information to an agent with expectations, so they are aware of progress;

   e.g., forward *Prescription* to PATIENT.

3. (*Continue*) Forward intermediate information to continue from checkpoint, instead of redoing work;

   e.g., forward *Prescription* from PATIENT to PHARMACIST.

These patterns are implemented by adding messages to the protocol, to capture the additional actions that can be taken. Although these changes could be made directly to the protocol, we propose the use of syntactic transformations applied to the protocol. Such transformations enable capturing the higher-level semantics of the change being made, and reduce the effort required for implementing these patterns by abstracting them. Thus, transformations represent a middle-ground compromise between directly modifying the existing protocol (fully backwards-compatible, but loses information about the transformation made), and developing a new protocol language that incorporates concepts such as retries or forwarding directly (which sacrifices backwards compatibility, both at the implementation and theoretical levels).

The following subsections show how to transform the protocol to support these patterns, and implement agent policies to use them.

4.3.3 Transforming Protocols via Annotations

To support fault tolerance, we can extend the protocol to include additional messages that enable the agents to make progress despite faults. These additional messages are working toward existing objectives and thus ultimately communicate the same information, so they can be derived from existing messages according to common patterns.

Although we could take advantage of the idempotence of information-based protocols and resend the original messages, we instead construct new messages for two reasons: First, a new message conveys the information that there is an attempted recovery. The message may introduce a new key parameter to permit multiple distinguishable recovery attempts. Second, new messages can add completely new communication patterns, such as having the PATIENT forward a copy of their prescription directly to PHARMACIST instead of asking DOCTOR to resend the prescription.

To reduce the effort of adding messages to a protocol, we introduce the concept of tooling-supported protocol *transformations*. A transformation takes a protocol and adds messages according to the designer's specifications. In the following listings, we give examples of transformations applied to messages from *Treatment*, followed by the output each transformation produces.

Our primary protocol transformation pattern is the *forward*, which resends some information from one agent to another; possibly though not necessarily between the original

agents. The forward transformation is the basis for all three of our patterns mentioned in Section 4.3.2. For example, the PATIENT can remind DOCTOR about their symptoms by forwarding the *Complaint* if they don't receive any treatment. We forward messages in our protocol using another message containing all the same parameters plus a new key to identify the instances of forwarding.

Listing 4.2: Forward Transformation.

```
@forward to Doctor Reminder rID
Patient -> Doctor: Complaint[out sID key, out symptom]

# produces:
Patient -> Doctor: Reminder[in sID key, in symptom, out rID key]
```

Listing 4.2 shows the `@forward` transformation. The annotation has three arguments: the recipient to forward the message to, the name to use for the new message, and a new key parameter. To support multiple forwards relaying the same message, the annotation can also take a `from` argument. Expanding this reminder transformation produces the *Reminder* message, which contains the same parameters as the original *Complaint* message, plus the new key.

For some patterns we provide additional shorthand. For example, as Listing 4.3 shows, to support the *Remind* pattern, `@remind` is shorthand for forwarding a message from the original sender to the original recipient.

Listing 4.3: Remind Shorthand.

```
@remind
Patient -> Doctor: Complaint[out sID key, out symptom]

// produces:
Patient -> Doctor: ComplaintReminder[in sID key, in symptom, out rID
   key]
```

Note that transformation can generate message and key names if they are not provided.

The examples of forwarding in Listings 4.2 and 4.3 are direct, between the original sender and recipient. In larger multiparty protocols, however, indirect reminders can greatly increase the range of possible fault recovery actions. By *indirect* reminders, we mean reminders received from an agent other than the original sender. Indirect reminders are used in the *Checkpoint* and *Continue* patterns. For example, the DOCTOR may give PATIENT a copy of the prescription (*Checkpoint*), to enable PATIENT to remind PHARMACIST about the prescription directly (*Continue*). Indirect reminders are simple but powerful concepts for

application-specific fault tolerance, endowing parties with necessary local information to directly respond to faults or follow up on incomplete tasks.

We further support indirect reminders through two additional transformations, `route` and `gossip`:

Listing 4.4: Routing Shorthand.

```
@route via Patient
Doctor -> Pharmacist: Prescription[in ID key, in symptom, out Rx]

// equivalent to
@forward to Patient
@forward from Patient to Pharmacist
Doctor -> Pharmacist: Prescription[in ID key, in symptom, out Rx]

// produces
Doctor -> Patient: FwdPrescription[in ID key, in symptom, in Rx, out
   fwdID]
Patient -> Pharmacist: FwdPrescription2[in ID key, in symptom, in
   Rx, out fwdID2]
```

The `route` annotation takes as parameters a sequence of intermediary agents through which to route the forwarding of a message, and expands to a sequence of forwards. In the example in Listing 4.4, `@route` is used to generate messages enabling Doctor to forward the prescription to Patient, who can then forward it again to Pharmacist.

Listing 4.5: Gossip Shorthand.

```
@gossip
Doctor -> Pharmacist: Prescription[in ID key, in symptom, out Rx]

// produces
Doctor -> Patient: FwdPrescription[in ID key, in symptom, in Rx, out
   fwdID]
Patient -> Pharmacist: FwdPrescription2[in ID key, in symptom, in
   Rx, out fwdID2]
Pharmacist -> Patient: FwdPrescription3[in ID key, in symptom, in
   Rx, out fwdID3]
```

Similarly, the `@gossip` transformation generates a complete transmission graph among a set of roles (defaulting to all roles), so each peer can forward a message to its neighbors to eventually reach everyone. Gossip is distinct from the concept of broadcast, which has

the source directly transmit to all recipients. Listing 4.5 shows how applying the `@gossip` transformation to the *Prescription* message adds new forwarding messages to the protocol so that any of the agents can forward it to the others. The gossip transformation enables the agents to enact any propagation policy they choose, and can be an effective means of transmitting messages if the topology of the network changes over time (as it would with agent failure) [138].

4.4 Programming Model

Figure 4.1 shows the architecture of a Mandrake agent. Each role in a protocol has a logical *skeleton* consisting of the messages it is enabled to send and receive; this skeleton is provided by the adapter components below the center line. An operational agent fleshes out this skeleton with internal business logic that drive its interactions with other agents by determining when to send a message and what to do with received messages. Such logic may be either proactive or reactive, forming the respective architectural components framed in dashed red lines. The internal logic would typically rely on private internal state, e.g., as encoded in internal databases.

The Adapter serves as the interface through which each agent interacts with other agents, enforcing compliance by checking incoming and outgoing messages against what the agent knows about the protocol and previously observed messages. The Adapter consists of the following components:

- Local State, which is responsible for storing the interaction history as observed by the agent.
- Checker, which processes incoming and outgoing messages to update the Local State after verifying that they satisfy protocol constraints.
- Emitter and Receiver, which handle the physical transmission of messages, forwarding parsed messages to the Checker.

The implementation and configuration are up to the system designers, but could be made using components from a standard library such as our reference implementation.

The remaining components constitute the agent internals:

- Internal State, independent of the agent's interactions
- Policy components: Proactors and Reactors, which emit messages based on internal triggers (such as a schedule) and message reception events respectively. These components can access but not modify the Local State.

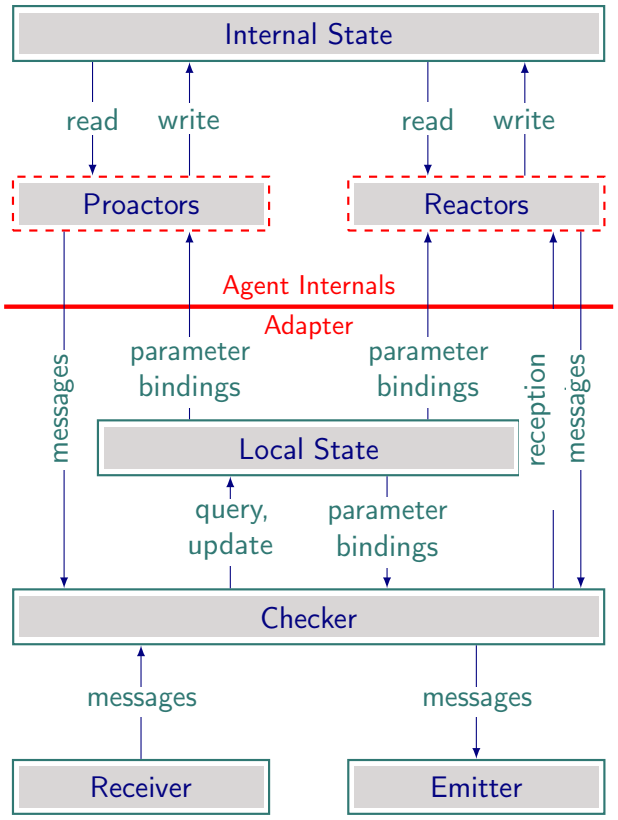


Figure 4.1: Agent architecture.

Our reference implementation realizes the programming model primarily as API functions provided by an adapter library in Python. These API functions, listed in Table 4.1, are all that is required to implement an agent.

Our example listings are also written in Python using the reference API, and many of them are taken from the example treatment scenario implementation in the above repository.

4.4.1 Reactors

Reactors are policy components that are registered with the Checker and invoked by the Checker in response to protocol events such as sending or receiving a message.

Listing 4.6: Reactor for Complaint.

```
# Doctor
@adapter.reaction(Complaint)
async def ComplaintHandler(message):
  logger.info(f"Received Complaint: {message}")
```

Listing 4.6 shows a reactor declaration for the *Complaint* message in the PATIENT agent.

Table 4.1: The Mandrake API for programming agents.

Function	Purpose
adapter(⟨role⟩, ⟨protocol⟩)	Initialize an adapter with a protocol specification and role, so it can check incoming and outgoing messages according to that role's perspective
adapter.register_reactor(⟨schema⟩,⟨handler⟩)	Register a function as a reactor for a schema; whenever that schema is observed, handler is invoked
@adapter.reaction(⟨schema⟩)	Decorator syntax for registering a function definition as a reactor
⟨schema⟩(⟨parameter⟩...)	Construct a message instance according to a schema from the protocol, using keyword arguments to bind parameters
⟨schema⟩.match(⟨parameter⟩...)	Get all enabled instances for a message that match the provided parameters, which are usually usually keys
⟨instance⟩.bind(⟨parameter⟩=⟨value⟩...)	Bind multiple parameters at once; an individual parameter can be bound via property assignment
adapter.send(⟨instance⟩)	Send a message instance to the recipient specified in the protocol
adapter.start(⟨task⟩...)	Start the adapter, listening for incoming messages and running asynchronous tasks such as proactors

Potentially, the role skeleton (describing its reactors) could be automatically generated by tooling from a protocol specification.

Listing 4.7: Complaint Reactor Implementation.

```
# Doctor ComplaintHandler
  for p in Prescription.match(complaint):
    # loop because match returns a list
    p.bind(Rx="aspirin")
    adapter.send(p)
```

Listing 4.7 shows a possible complaint reactor implementation, that gets an instance of the *Prescription* message corresponding to the same enactment, binds its Rx parameter to `aspirin`, and sends it. Requesting a message instance from the enactment automatically fills the known ⌜in⌝ parameters; in this case ID and symptom.

4.4.2 Proactors

Proactors are policy components that are not invoked by the Checker; they could be run according to a schedule or some other internal trigger. Proactive policies are necessary for initiating an enactment, because there are no prior events to react to, but other messages in a protocol may also be sent proactively.

For example, to generate the initial *Complaint* in the *Treatment* protocol, PATIENT could use the following proactive policy function:

Listing 4.8: Proactor for Complaint.

```
# Patient
def Complain(description):
  c = Complaint(ID=uuid.uuid4(),
                symptom=description)
  adapter.send(c)
```

The `Complain` function is not automatically invoked in reaction to an event, but must be proactively invoked by the agent to initiate the interaction. Perhaps PATIENT invokes `Complain` through a suitable user interface, such as a mobile app. Within `Complain`, a *Complaint* message is constructed, passing in the ID and symptom. Because a *Complaint* message initiates a new enactment, it can be constructed directly from parameters instead of derived from previous observations. Finally, the adapter sends the message.

4.4.3 Utilizing Transformations via Agent Policies

Once a protocol has been extended either manually or via transformations to support information forwarding and thus fault recovery, the agents need policies to send the messages at the right times.

Listing 4.9: Hand-written Forwarding Policy.

```
@adapter.reaction(Prescription)
def forward_prescription(message):
  fwd = FwdPrescription.match(message)
  fwd.bind(fwdID=guid())
  adapter.send(fwd)
```

Listing 4.9 implements a simple reactor in the DOCTOR for forwarding the *Prescription* message. When DOCTOR sends *Prescription*, a copy is made with an additional fwdID key, and forwarded to PATIENT.

Reminder policies are more complicated and challenging to write by hand, since reminders are normally only sent for known messages that have not yet been responded to, often according to a schedule.

Listing 4.10: Hand-written Reminder Policy.

```
async def complaint_reminder():
  while True:
    # noon every day
    await aiocrontab.crontab("0 12 * * *").next()
    # loop through all Complaint messages
    for s in adapter.history.by_msg(Complaint):
      # get matching ComplaintReminder instance
      for c in ComplaintReminder.match(s):
        # don't remind about complaints that have treatments
        if any(m.schema in [Reassure, FilledRx] for m in
           adapter.history.find_context(s).messages):
          continue
        else:
          # send message
          c.bind(rID=guid())
          adapter.send(c)
```

Listing 4.10 shows how a reminder policy might be implemented in the PATIENT. The function loops indefinitely, but sleeps until noon each day. At the appropriate time, it scans the history for all sent *Complaint* messages, and sends a reminder for those that do not have a treatment, as evidenced by the observation of a Reassurance or FilledRx message.

Policies for the `@forward`, `@route`, and `@gossip` transformations can be implemented similarly. Fundamentally, every policy either reacts to a message reception (and can use the relevant enactment history) or proactively generates messages according to a schedule.

4.4.4 Declarative Policy Specification

Because some fault tolerance policies are common and potentially complex to write directly in low-level proactor and reactor code (e.g., sending reminders, as illustrated in Listing 4.10), we have implemented a declarative domain-specific language for specifying a class of fault tolerance policies. Instead of implementing the corresponding proactors and reactors, developers can instead specify rules in the language.

Syntax

In the syntax description below, ↩⊃ indicates a line break, and brackets indicate optionality. Monospace `tokens` and quotations are parsed literally, and italic *Tokens* are terms. `+` indicates the object should be matched one or more times.

$\mathcal{L}_1$ Each agent is given a list of policies, with each entry at the same indentation level prefixed by a hyphen and space

$$\textit{Specification} \longrightarrow (\texttt{`- '}\ \textit{Policy}\ \hookleftarrow\supset)\texttt{+}$$

$\mathcal{L}_2$ Each policy consists of several clauses: an 'action' clause, and additional condition clauses. The clauses are given by a keyword followed by a colon and space (': '), and then the body of the clause, usually one clause per line.

$$\begin{aligned}\textit{Policy} \longrightarrow\ & \texttt{action:}\ \textit{Action}\ \hookleftarrow\supset \\ & [\texttt{when:}\ \textit{When}]\ \hookleftarrow\supset \\ & [\texttt{max times:}\ \textit{number}]\ \hookleftarrow\supset\end{aligned}$$

$\mathcal{L}_3$ The body of an action clause itself consists of parts: an action to perform, a list of messages, and optional destination, delay, and preposition subclauses.

$$\begin{aligned}\textit{Action} \longrightarrow\ & (\textit{remind}|\textit{forward}|\textit{broadcast}) \\ & [\texttt{after}\ n\ \texttt{seconds}] \\ & [\texttt{until}|\texttt{upon}\ \textit{events}]\end{aligned}$$

$\mathcal{L}_4$ The various actions have slightly different structures

$$\begin{aligned} \textit{remind} &\longrightarrow \texttt{remind}\ \textit{role}\ \texttt{of}\ \textit{messages} \\ \textit{forward} &\longrightarrow \texttt{forward}\ \textit{messages}\ \texttt{to}\ \textit{role} \\ \textit{broadcast} &\longrightarrow \texttt{broadcast}\ \textit{messages} \end{aligned}$$

$\mathcal{L}_5$ The event subclause of an action identifies an event to react to or wait for

$$\begin{aligned} \textit{events} &\longrightarrow \textit{event}\ (\texttt{(or|and)}\ \textit{event})\texttt{+} \\ \textit{event} &\longrightarrow \texttt{(received|duplicate)}\ \textit{messages} \end{aligned}$$

$\mathcal{L}_6$ A *When* clause describes the schedule for invoking a proactive policy, in traditional cron notation or some frequency in seconds; this clause is left off for reactive policies.

$$\textit{When} \longrightarrow \textit{cron string} \mid \texttt{(every}\ n\ \texttt{seconds)}$$

$\mathcal{L}_7$ A cron string specifies a period using 5 fields: minute, hour, day, month, weekday. Each field is either a number, or an asterisk (*), meaning 'every'. Field may also be restricted with $*/n$, meaning 'every nth'. Multiple values can be provided for each field, separated by commas.

For example, `'30 12 */2 * *'` means 'every other day at 12:30'.

Examples

Listing 4.11: Reminder Policy Specification.

```
- action: forward Complaint to Doctor
  when: 0 12 * * * // noon every day
```

Listing 4.11 shows a policy specification for Patient to remind Doctor about the complaint by forwarding Doctor a new copy of the message on the first of each month. When loaded into Patient's adapter, this policy specification creates a new Scheduler set to run every day, and registers a generated handler that sends the appropriate reminder.

The agent implementation needs to know which message to use for sending a reminder or forward. To avoid coupling the agent implementations, we use an explicit, declarative mapping specification instead of an implicit naming convention:

Listing 4.12: Message Mapping.

```
map:
  forwards:
    Prescription: FwdPrescription
```

These mapping relationships can be automatically generated by tooling as part of the protocol transformation process.

Besides scheduled actions, the language can specify reactions, such as automatically forwarding information immediately when it becomes available:

Listing 4.13: Reactive Forwarding Policy Specification.

```
// Doctor
- action: forward Prescription
  // implicitly reactive

// Patient
- action: remind Doctor of Complaint until Reassurance or
   Prescription
  when: 0 12 * * * // daily at noon
```

The reactive policy in Listing 4.13 automatically forwards the *Prescription* message to PATIENT when it is observed; that is, right after sending it to PHARMACIST. The policy registers a reactor for the *Prescription* message that generates a corresponding *FwdPrescription* according to the map and sends it.

The second specification uses the keyword `remind` as an alias for the generic forward action, and generates two policy components for PATIENT: a proactor for reminding DOCTOR of the complaint, and a reactor for *Prescription* and *Reassurance* that deactivates the reminder for the corresponding complaint after receiving one of those two messages. Thus, this policy will stop sending DOCTOR reminders after the expectation for a treatment is fulfilled, where Listing 4.11 would continue indefinitely.

These simple policies can be used to construct flexible fault recovery strategies by sending reminders until receiving evidence of progress. For simplicity, forwarding sends the message to all potential recipients by default if no recipients are explicitly specified. Similarly, if a policy does not explicitly specify a condition, it implicitly reacts to the concerned messages. For example, `forward Prescription` automatically forwards *Prescription* (e.g., by sending *FwdPrescription*) to all potential recipients immediately after *Prescription* is observed (sent).

Listing 4.14: Robust Prescription Policies, building on Listing 4.13

```
// Doctor
- action: forward Prescription
  // implicitly reactive

// Patient
- action: remind Doctor of Complaint until Reassurance or
   Prescription
  when: 0 12 * * * // daily at noon
- action: remind Pharmacist of Prescription after 2 days until
   FilledRx
  when: 0 0 * * * // daily
  max tries: 5

// Pharmacist
- action: remind Patient of FilledRx upon Prescription reminder
```

Listing 4.14 shows how remind and forward can be combined to enable more effective recovery, adding new policies for Patient and Pharmacist to those in Listing 4.13. Because Patient is the stakeholder desiring FilledRx, it is the best party to judge whether the task has been completed. Forwarding *Prescription* to Patient gives Patient direct recourse to address a delayed or lost prescription by reminding Pharmacist directly. If *Prescription* is lost, the reminder gives Pharmacist the information necessary to send *FilledRx*. If FilledRx is lost instead, the reminder prompts Pharmacist to resend it.

4.5 Evaluation

We evaluate our work conceptually, by explaining how our approach supports application meaning; comparatively, by contrasting it with what would be possible in another MAS framework; and experimentally, by demonstrating how application-level fault tolerance can be effective, practical approach.

4.5.1 Conceptual Evaluation

Our primary contribution is an interaction-oriented programming model designed to support application-level fault tolerance, whose primary competitors are infrastructure-level techniques and ad hoc solutions. Although performance comparisons could be made between our work and examples of competing approaches, such a comparison would at best risk comparing

poor versions of each, but would also make a category error. As such, we further develop our argument here regarding the necessity of application meaning and interaction specifications, and show the necessity for an approach like Mandrake.

Application meaning

To understand our contribution, it helps to take a quick look at the history of networked applications. The creators of the IP standard, for example, recognized the criticality of application meaning [55]. They observed, correctly, that application meaning varies across applications. Thus, it appeared to them that there would be no systematic way to support application meaning through network abstractions. However, they also saw the need for abstractions. Accordingly, they chose to provide a connection abstraction with reliability and ordering. TCP realizes this abstraction. It supports client-server programming, which relieves programmers of having to model complex interactions and creates the illusion of a distributed finite state machine. The solution was known to be suboptimal even at that time because it violates the end-to-end principle that some of the same researchers had recognized and articulated [137]. In particular, client-server programming is ill-suited to decentralization because it is at odds with the idea of the endpoints carrying out their respective computations largely independently of one another with coupling only where necessary.

In a nutshell, then, computing has faced the dilemma of a design choice between (1) accommodating application meaning but at the cost of working directly on low-level abstractions and (2) using programming abstractions that hide network behavior but at the cost of a more restrictive communication layer that largely subverts decentralization.

We respond to the above dilemma by adopting the abstraction of an *information protocol*. A protocol captures the part of an application's meaning concerned with how its endpoints interact, leaving their internal details of each endpoint completely hidden. Moreover, an information protocol captures only the essential causal and integrity constraints to achieve a successful interaction—i.e., judged successful according to application meaning. By avoiding spurious constraints, a protocol provides the most flexible computations with respect to application meaning. An abstraction based on protocols is thus both generic and avoids spurious restrictions.

Potential causality

Research on implementations based on the dominant paradigm of *potential* causality [104] seek to provide even stronger communication guarantees, such as the ordering observed by distinct endpoints being totally ordered [27]. However, potential causality imagines

internal causal connections in an application: a message being sent is assumed to be causally subsequent to any previously received message. Thus, it inherently overestimates the causal relationships. Overestimating the casual relationships leads to well-known problems as brought forth by Cheriton and Skeen [36]; Birman originally resisted this point [26] but has recently acknowledged it in essence [25].

In contrast, information protocols as adopted here capture explicit causal dependencies between message parameters. If one message has an ⌜in⌝ parameter, that message is causally dependent on the observation of another message that communicates a binding for that parameter. In this way, information protocols take the opposite stance on causality and avoid the problems of potential causality.

4.5.2 Comparison: Jason

A significant part of Mandrake is the programming model, which is intended to simplify agent development, with a special focus on supporting application-level fault tolerance. To evaluate the programming model, we compare it to Jason [32], the agent programming portion of the JaCaMo framework [30]. We compare our work with Jason for these reasons: to show that a popular existing approach does not support application-level fault tolerance (novelty), yet would benefit (significance); and to describe how our work might be applied to other platforms than our own reference implementation (generality).

Jason is an implementation and extension of the AgentSpeak [31] programming language that provides logic programming techniques for building agents according to the Belief-Desire-Intention (BDI) model [64]. Mandrake's programming model focuses on supporting the interactive aspects of an agent, leaving the internal decision-making to be plugged in; a BDI model could be used to provide decision-making for a Mandrake agent, perhaps using the python-agentspeak library. We have not implemented a BDI reasoner for any Mandrake agents, but parameters learned through message observations match well with terms in a belief base; and Mandrake's support for both proactive and reactive logic should be able to accommodate any reasoner.

Model

Jason is a high-level system for agent development, organizing actions into plans and goals, and using communication abstractions to directly exchange beliefs with other agents and select an action based on the current beliefs. Since communication is abstract, multiple infrastructures can be used for communication, such as a centralized mode for local development, or the JADE agent development framework. JADE in turn can be configured to use multiple

transport protocols for sending messages, including HTTP, XMPP[81], and JMS[57]. Each of these protocols aims to provide reliable communication at the infrastructure level; HTTP for instance is based on TCP, which resends packets until they are acknowledged. This approach is effective at dealing with uncommon short-term and seemingly random packet loss.

Protocols are not a first-class abstraction in Jason; instead, communication patterns are coded as part of the agent logic, coupling the agent implementations. Nor are protocols often used as an external specification to guide development, so interaction correctness cannot be verified.

However, Jason's declarative logic programming style is compatible with an information-based implementation, and provides flexibility that can help with asynchronous messages. If they are properly used, Jason's rule-based intentions can match the corresponding events regardless of the order they occur, and multiple permutations can be handled with only a few lines.

Fault Tolerance

Neither Jason nor JADE has specific support for application-level fault tolerance. As such, any fault tolerance will be ad hoc and must be replicated for each message. The code to implement timeouts and retries can easily begin to outweigh the primary interaction logic. However, even if fault tolerance is not supported by the language, the concept of application-level fault tolerance is still important, and a variant of the Mandrake methodology can still be useful.

Plans should be written to progress according to available knowledge instead of simple sequences. For example, instead of sequentially handling the steps of a protocol as in the Contract Net example included with Jason, make each step a separate intention triggered by a belief, such as the arrival of all bids. This decouples the steps and makes it easier to enter them from multiple paths, such as a recovery path, by simply setting the correct beliefs. Then plans should be augmented to track expectations, triggering a recovery plan if the expectation is not met.

This methodology should enable application-level fault tolerance, even if the implementation is manual and tedious. Jason's support for Java extensions could enable the implementation of a library for Mandrake similar to our Python version.

4.5.3 Experimental Results

We have implemented the system, and performed lightweight experimental evaluations to demonstrate its existence, and possible strengths and weaknesses of the proposed recovery patterns. These experiments were performed using a Linux VM on a laptop, and are not

expected to demonstrate best-in-class performance.

The experiment features two parameters: the recovery policies, and the loss rate. The results are displayed in multiple subfigures grouped by the statistic being analyzed: the total enactments completed within the 30 second timeout, the number of messages emitted by PATIENT, the number of packets sent by PATIENT, and the rate at which each policy completed its enactments.

The policies examined are (1) a simple reminder policy (referred to as Retry), where PATIENT reminds DOCTOR of *Complaint* if their expectation for *FilledRx* is unmet, and DOCTOR and PHARMACIST correspondingly resend any matching messages when reminded to do so, and (2) a checkpoint policy. For the checkpoint policy, PATIENT reminds DOCTOR of their complaint until they receive either a copy of their prescription or its fulfillment. Once PATIENT has received a copy of the prescription, it no longer reminds DOCTOR but forwards the prescription directly to PHARMACIST; the idea being that if the DOCTOR is busy or unreliable, sending directly will be faster. Once PATIENT has forwarded the prescription to PHARMACIST, they continue reminding PHARMACIST of the prescription until it is filled.

These two policies represent conservative and augmented information flows, respectively. The conservative approach, implemented as a simple reminder policy, only sends information along the original pathways provided by the protocol; e.g. the patient only interacts with the doctor. The conservative retry policy is still an application-level policy, but it is somewhat restricted. The checkpoint policy is more proactive, augmenting the protocol with alternative information flows: the patient can now communicate with the pharmacist directly. One objective of this experiment is to show that when implemented correctly, the augmented flows can be at least as good as a basic reminder flow.

The loss rate is the probability that any message sent by a given agent is lost. We simulated losses of 1%, 5%, and 25% (for a consistent factor of 5 between tests), to see how the statistics scale under simulated conditions representing UDP packet loss during transmission, or overwhelmed agents. UDP loss in a network rarely more than a few percent, but more extreme loss rates were included to make the effects of greater loss more apparent. Instead of simulating all combinations of agent loss rates, we focused on several cases: no loss, one lossy agent with the others unaffected, both non-PATIENT agents being lossy (since it's the one doing the recovery) and all emissions equally lossy. Each of these cases is represented by a separate subfigure, with the label indicating which agents are lossy for that experiment. During a given step, all lossy agents are set to have the same loss rate.

The experiment was run for five iterations for each combination of loss configuration and recovery policy, to average the results. The sample standard deviations were calculated and included in the graphed data, but are too small to see. The PATIENT was configured to send

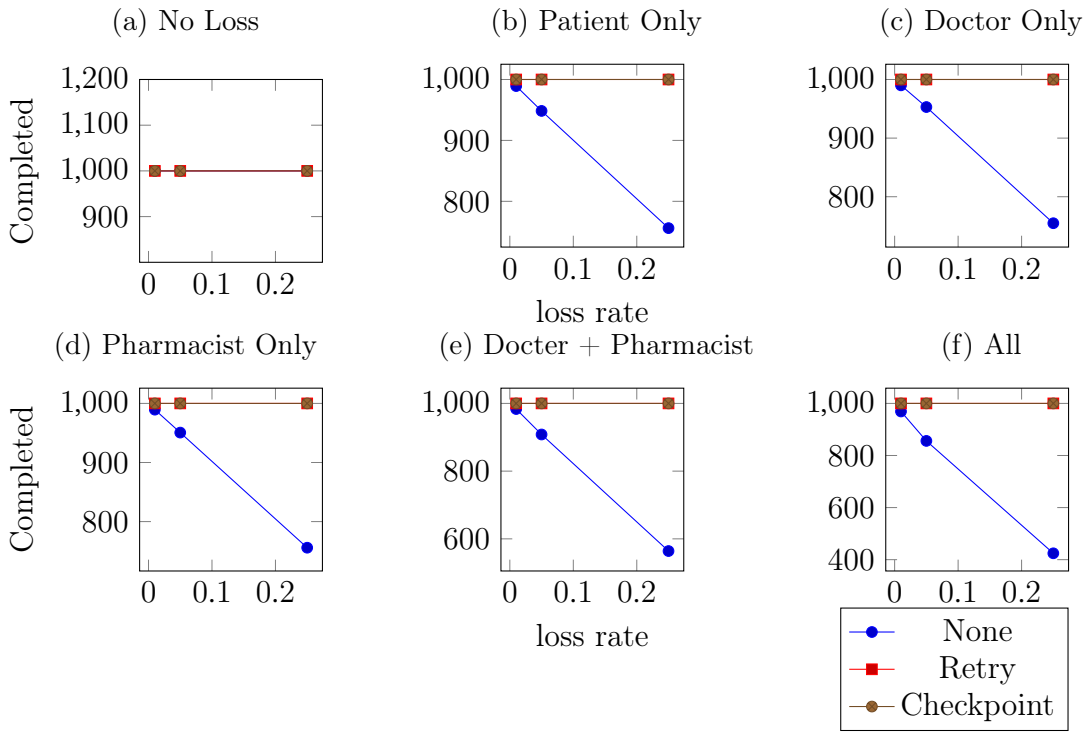


Figure 4.2: Total enactments completed. Each subfigure represents a different loss configuration, with the lines representing the udieecovery policies. In this figure, a line is also given for the absence of a recovery policy, to show the cumulative effect of the loss rate. Subfigure (a) has no loss, (b-d) have one lossy agent, (e) has both doctor and pharmacist lossy, and in (f) all agents are equally lossy. The Y-axes show the number of enactments completed by the timeout, which is set to be 30 seconds. The X-axes are the three different loss rates tested: 0.01, 0.05, and 0.25.

1000 complaints, and expect responses within 1 second (that is, run its recovery policies every second, and wait at least one second before resending a message). Clearly unrealistic for even the most impatient hypochondriac, but still useful as a simulation because our focus is on message loss and recovery. An iteration of the experiment would end as soon as all the enactments were complete, or it had no recovery policy and had not completed any enactments for more than one second. There was also a 30 second timeout that was rarely reached.

Figure 4.2 shows the total number of enactments completed by each policy for each loss configuration. Both recovery policies were effective at recovering from loss in all cases shown here. We include the results for the cases where the patient does not employ a retry policy to show that completion decreases proportionally to cumulative loss, as expected.

Figure 4.3 shows the number of message emissions PATIENT made for each policy and loss configuration. Without any loss, exactly 1,000 messages are emitted. Here we see a slight

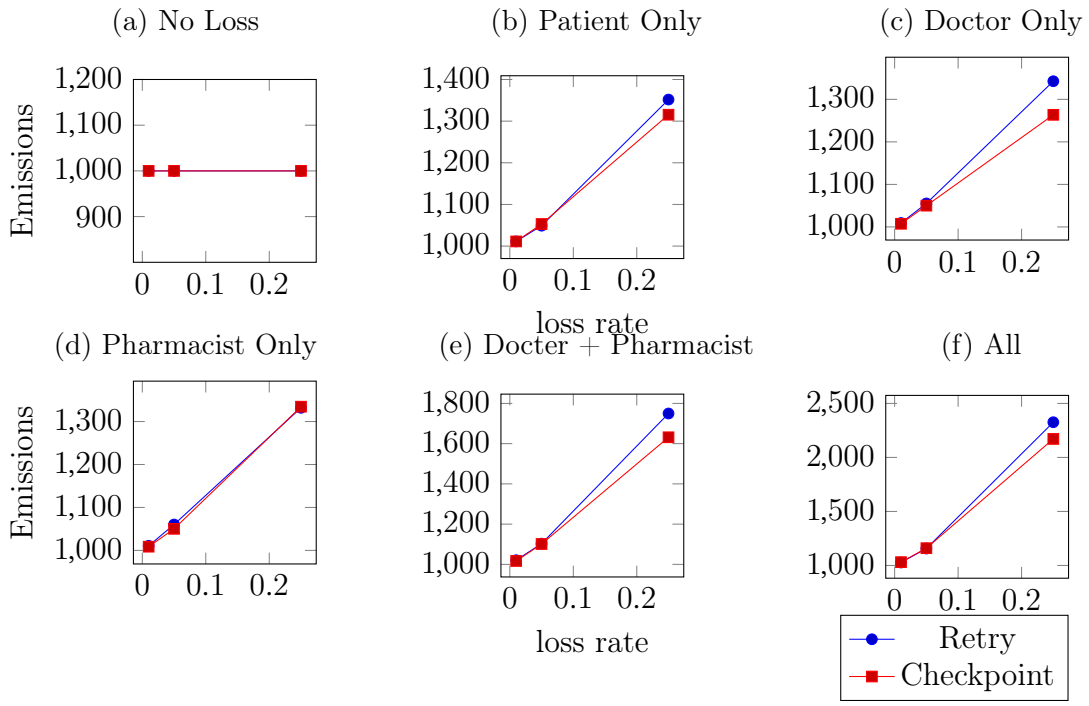


Figure 4.3: Messages emitted. Each subfigure represents a different loss configuration, with the lines representing the two recovery policies. Subfigure (a) has no loss, (b-d) have one lossy agent, (e) has both doctor and pharmacist lossy, and in (f) all agents are equally lossy. The Y-axes show the number of messages emitted by PATIENT. The X-axes are the three different loss rates tested: 0.01, 0.05, and 0.25.

difference in performance between the two recovery policies. Both have to send additional messages to recover from loss, but the basic retry policy is sensitive to the loss rates of both DOCTOR and PHARMACIST, whereas the checkpoint policy is much less sensitive to DOCTOR's loss rate. The checkpoint policy never performs worse than the retry policy, and outperforms it to the greatest extent when only DOCTOR is lossy. This is because DOCTOR will sometimes succeed in sending a copy of the prescription to PATIENT even when failing to send it to PHARMACIST, but during the subsequent recovery the patient can send the message to PHARMACIST directly via a much more reliable connection.

Note that there is additional overhead incurred by the checkpoint approach; namely that DOCTOR must send extra copies of the prescription to PATIENT, doubling the number of messages DOCTOR must send under normal conditions (in failure conditions some of the reminder work is taken over by PATIENT). Thus, checkpoint-style policies are likely best used in applications where replicating the information to multiple agents is either already performed as part of the existing requirements (such as monitoring), or of negligible marginal cost.

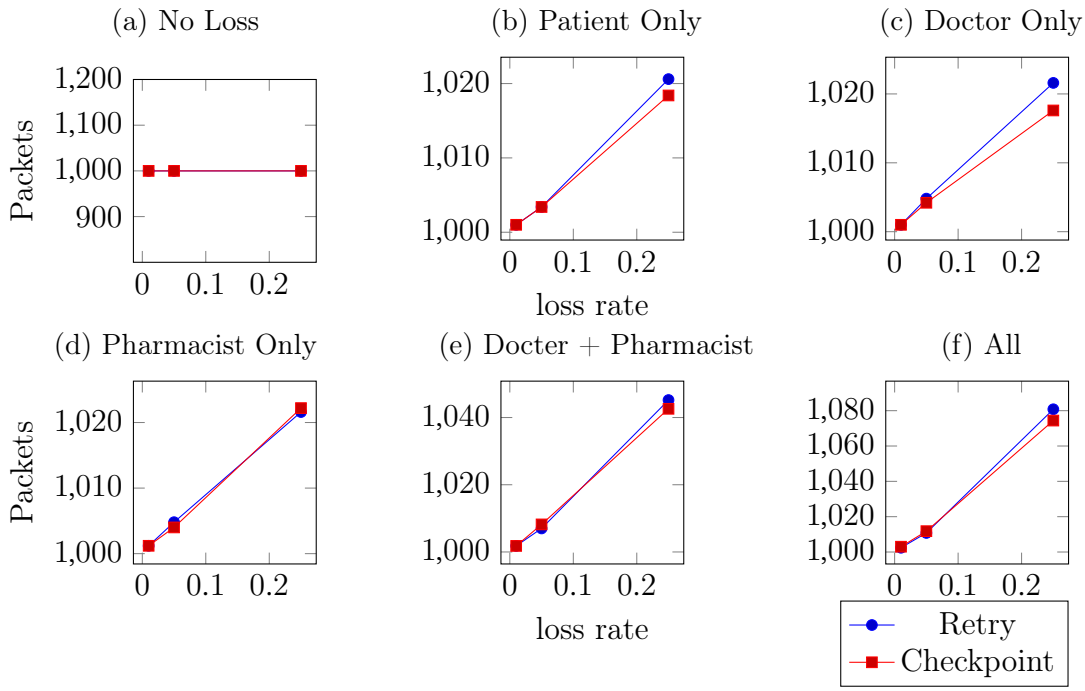


Figure 4.4: Total packets sent. Each subfigure represents a different loss configuration, with the lines representing the two recovery policies. Subfigure (a) has no loss, (b-d) have one lossy agent, (e) has both doctor and pharmacist lossy, and in (f) all agents are equally lossy. The Y-axes show the number of UDP packets sent by PATIENT. The X-axes are the three different loss rates tested: 0.01, 0.05, and 0.25.

Figure 4.4 shows similarly shaped results for the number of packets that PATIENT emits, though the total numbers are much lower. This is because the policies are batch processes, and may find multiple enactments that require action (resending, forwarding, etc.) at the same time; when multiple messages are generated at the same time, our implementation groups the messages by recipient and encodes them into as few packets as it can. When the messages are small, as in this experiment, the savings can be significant—here, the number of messages ranged from 1000 to 3000, but the number of packets only ranged from 1000 to 1080.

The final graph, Figure 4.5, shows the rate at which each policy completed enactments, in enactments per second. The first configuration, where none of the agents are lossy, shows that the simple retry policy is slightly more efficient to execute than the checkpoint policy. In all other cases, their performance seems similar, with the only notable difference being again the DOCTOR-only case, where the checkpoint policy's throughput is much less effected by loss than the retry policy.

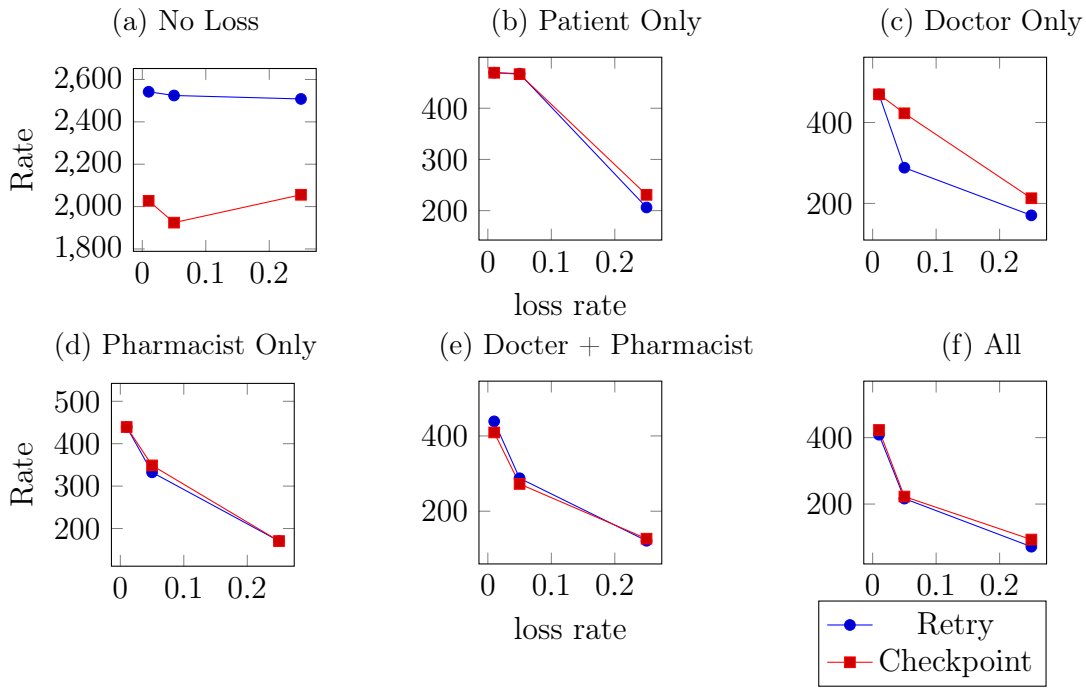


Figure 4.5: Rate of completion (enactments/second). Each subfigure represents a different loss configuration, with the lines representing the two recovery policies. Subfigure (a) has no loss, (b-d) have one lossy agent, (e) has both doctor and pharmacist lossy, and in (f) all agents are equally lossy. The Y-axes show the number of enactments completed (by observing *FilledRx* or one of its reminders) divided by the duration of the iteration. The X-axes are the three different loss rates tested: 0.01, 0.05, and 0.25.

4.6 Related Work

Fault handling may be thought as a kind of exception handling. However, exception in the context of programming languages, including process languages such as BPMN [123], are handled by a centralized language runtime. Plus, exceptions and exception handling are implemented via out of band signaling, which adds complexity to the program. By contrast, Mandrake addresses faults in decentralized MAS, specifically, faults understood as the violation of expectations in the enactment of protocols. Agents with expectations may initiate recovery, in alignment with the protocol. Fault tolerance in Mandrake merely involves the addition of alternative paths to completion, such as retries and forwarding.

Software systems built using the actor model, where processes are encapsulated in actors that interact with their environment through simple messages, have long been used to build fault-tolerant distributed systems [9]. The actor model [1, 2] is conducive to fault tolerance both because the simple messaging interface of actors minimizes coupling and enables testing, and because actors easily model physical distribution, which is necessary for eliminating

single points of failure.

Actor-based systems commonly address fault tolerance through supervision hierarchies. If an actor throws an error, its supervisor receives the error and can respond by restarting the actor, killing dependent actors, or possibly dying and throwing an error itself. These supervision hierarchies enable the developer to design fault tolerance at the application level, and can be supported by libraries such as Erlang's OTP [35] to minimize development effort. However, Erlang does not have specific support for programming application level message retries, relying on TCP for automatic packet-level retries and supervisor-triggered restarts when the connection breaks. While supervision could conceptually be used to implement application-level retries, supervisor hierarchies in Erlang/OTP are specifically for detecting process failures, and restarting failed processes and any processes that depend on them. This mechanism for restarting crashed agents is complementary with Mandrake, which focuses on recovering lost information between live agents. Actor system fault tolerance also usually focuses on handling signaled errors—errors that do not immediately crash the actor, but are instead reported via some status message for another actor to handle. Such signals are helpful for handling domain-specific problems that prevent progress (Mandrake's retries will not help if the Pharmacist is waiting for a delivery from their supplier), or which need to be resolved by another party (perhaps the Pharmacist needs to notify their supplier when they run out of stock).

To avoid hidden coupling, all communications between agents should be explicitly specified; thus the status and error messages should be added to the protocol specification. Signaled errors are thus compatible with Mandrake; we merely focused on the aspects that Mandrake specifically addresses. However, signaled errors cannot handle cases where information was lost (e.g. the Pharmacist could not signal an error for a prescription it never received), or when an unexpected crash occurs (e.g. a hardware failure, preventing any signals from being sent); Mandrake provides a solution for these situations through expectations. Finally, most actor systems rely on infrastructure for reliable messaging and process messages sequentially, expecting FIFO ordering. Relying on FIFO enables the developers to avoid explicitly modeling actor state, and TCP signals when a connection breaks simplifying supervision. In contrast, Mandrake's explicit information model relieves the dependence on FIFO for state alignment, and uses expectations to detect agent failures so communication can survive temporary network disruptions.

Microservices are an increasingly common architecture and means of deployment. Each microservice is intended to encapsulate a single application concern, exposed as a web service for easy composition, and deployed in an isolated environment to reduce coupling and interference from other components [118]. Traditional web-based interactions are asymmetric,

with the client merely reflecting state held by the server. However, microservices are often clients themselves, and depend on other services to produce their outputs, exposing them to communication faults. To some extent fault tolerance is transparently added to microservices via *sidecars* or *service meshes* such as Istio [95], proxies that manage requests and retries on behalf of the microservice. In contrast, Mandrake assumes a decentralized system where each agent manages its own state and messages, enabling more flexible and robust communication patterns between agents.

Fisher et al. [72] discuss a framework for certifying reliable autonomous entities, but consider the problem at the social and regulatory levels. Such an approach is useful for identifying the methods and processes can be used to certify that an agent will behave as it ought despite its autonomy, but does not support designing a system that can complete objectives despite faults.

Much work has been done to support programming models for agent development, such as the JaCaMo framework and extensions thereto [17, 19, 134, 135] and JADE [12]. However, most existing work on programming models is focused on implementing the agent logic rather than supporting interactions, and little effort has focused on supporting fault tolerance. IODA is an approach for multi-agent simulations that does have a model of interactions [102], though it does not specifically support application-level fault tolerance.

There is MAS work focused on exception handling in multiagent contexts: Klein and Dellarocas [101] propose a shared exception handling service that other agents turn to for help recovering from a problem, such as when a plan fails or a garbled message is received. Platon et al. [130] surveyed the challenges for exception handling in MAS, and Platon's thesis is on exception handling and a robust framework for executing agent plans [130]. Lam et al. [103] describe a workflow management system that uses norms and semantic web techniques to handle exceptions that arise during enactment. Mandrake takes a more agent-local and interaction-oriented approach, with each agent responding to any violations of its own expectations using communication patterns supported by an interaction protocol.

Outside of programming models and exception handling, there is also much work on protocols, modeling, verification, and monitoring of MAS behavior.

Several approaches model protocols by specifying an ordering of messages via some control-flow expression, e.g., via a state machine or trace expression. Baldoni et al. [13] present a state machine-based view of interaction from the agent standpoint and show how interoperability of agents can be ensured provided they conform with the endpoint specification based on the role they adopt in a protocol, the main challenge there being to guarantee that the agents don't make incompatible choices. Ferrando et al. [70] specify protocols as trace expressions, with recent work on enactability. Winikoff et al. [170] proposes a graphical protocol notation

based on hierarchical state machines combined with logical predicates for improved usability and expressiveness without sacrificing formal verification.

The control flow-based languages typically assume reliable, FIFO delivery of messages between pairs of agents. However, as we saw earlier, ordering assumptions interfere with agent autonomy by limiting the choices available to them. And reliability assumptions are generally insufficient for purposes of fault tolerance since what counts as a fault and recovery are application-level considerations. One could argue that by specifying message ordering, the control flow-based languages lift the ills of ordering to the application level.

By contrast, BSPL, by specifying information causality and integrity, enables implementing protocols with unordered, unreliable communication. Not relying on ordering in fact helps to easily implement fault tolerance at the application level. For example, to handle message loss, agents have the luxury of being able to retry a transmission whenever they want. And agents can receive such a retransmission at any time. Further, receiving a duplicate has no effect in BSPL because it simply brings already known information.

Dastani et al. [59] present an approach on monitoring agent behavior against specified norms. Günay et al. [84] study commitment protocols, and how to dynamically adapt them to changes in the environment or in the agents to ensure goals could still be reached. Baldoni et al. [17] use type checking for roles in a commitment protocol. In the information protocol framework underlying Mandrake, the compatibility of the choices allowed to agents (via roles) by a protocol are checked through verifying safety and liveness of a protocol [149, 156]. The Mandrake adapter uses runtime protocol monitoring to ensure compliance and support fault recovery policies.

Agent-based approaches are now being employed for applications such as the Internet of Things [54] to improve flexibility (because agents can use multiple means to achieve a goal) and responsiveness (because agents act independently without relying on orchestration) in such a decentralized setting. These objectives would benefit from an approach such as ours, and show how our work could be useful for existing web and IoT applications.

Today, most developers rely on existing protocols such as TCP for communication between services or agents. A few build a new infrastructure-level protocol for their needs that may later become standardized, such as uTP [28] and QUIC [97]. Rarely is application-level information used to inform network-level fault recovery; the closest examples might be job-queuing systems that distinguish between message reception and job completion.

Many network protocols such as TCP use acknowledgments to confirm packet delivery, and thus quickly detect and recover from packet loss [4]. Acknowledgments can also be used at the application level, but we do not focus on application-level acknowledgments because their similarity to packet acknowledgments distracts from the broader significance of application-

level fault tolerance; we discuss them briefly here instead. Network-level acknowledgments are not meaningful: they only confirm packet delivery, not whether the information has been received or processed by the application. Conversely, application-level acknowledgments are less efficient for large data transfers because decision-making and autonomy, necessary for meaning, increase latency and overhead. However, application-level acknowledgments convey the meaning that the information was observed and acknowledged by the agent; they provide a basis for commitments, and indicate that the information is being processed—likely an acknowledged message has been logged, and could be restored even if the agent crashes.

4.7 Discussion: Conclusions and Future Directions

In this paper we have raised the topic of meaning-based application-level fault tolerance. Every application has its own objectives and criteria for success, and fault tolerance must work toward those goals, not just ensure individual steps. Accomplishing application objectives requires awareness and utilization of application meaning; infrastructure-level techniques do not have access to application-level information and so cannot solve application-level problems. Infrastructure-level techniques are used because they are well understood, optimized, and generic—any problem that uses the infrastructure can theoretically benefit from them. To close this gap between easy-to-use infrastructure-level techniques and the application-level solutions we need, specifically in the MAS domain, we propose an interaction-oriented agent programming model with first-class support for fault tolerance. However, our work is just the beginning. We now discuss several possible directions for future work:

Multicast errors Although many common system designs use a pairwise interaction model compatible with our current work, there are other communication patterns for which the compatibility is less clear. One important direction for extending our work might be to determine Mandrake's limits for handling multicast communication and then address any revealed problem, possibly by extending our framework to include patterns for consensus protocols or other aggregation.

Role replacement One possible way to restore progress in a decentralized application despite unavailability is to identify new business partners. That is, the agents enacting a protocol could give up on a counterparty that is nonresponsive, and find another agent to fill the role. For example, Patient could choose a different Doctor or Pharmacist if Patient is unsatisfied with their service.

Porting to other platforms Mandrake is currently implemented for BSPL on a Python-based agent development framework. However, the concepts proposed by Mandrake should be applicable to other agent development frameworks. For example, a Mandrake-style protocol adapter could be implemented for JADE or JaCaMo on top of their existing speech-act communication systems, which would enable the enactment of BSPL protocols on those platforms. Or, a BDI-based system for expectations and recovery policies could be built, integrating Mandrake concepts more natively. It would also be interesting to generate Jason plan skeletons using our declarative policy specifications.

Dynamically adjusting transmitted information Although a message's dependencies may be useful to track for provenance, not all information dependencies are useful for the recipient, especially if they are already known. Redacting unused parameters may save bandwidth and decoding time.

To support such an optimization, we conceptually distinguish between four representations of a message: the schema, instance, bun, and packet. The schema is the abstract specification of a message, and is not associated with a particular payload or enactment. An instance consists of a message schema plus information bindings, and is associated with an enactment. A *bun* is an instance of a message that has been serialized for transmission; it may have more or fewer parameters than the message schema, so long as the recipient can reconstruct the complete instance upon reception. Finally, the packet is the physical message (e.g., UDP) transmitted by the infrastructure, though some transports (such as TCP) may not have application-visible packets. We distinguish between bun and packet because a bun is the representation of a complete message; one bun may be split across multiple packets, or multiple buns may be batched into a single packet.

Congestion control Enabling the switch from TCP to UDP may gain in latency and overhead but risks problems such as congestion collapse. Such problems may not be apparent in simple simulations but can make real-life implementations difficult.

By sharing information with the Receiver, the Emitter can be aware of statistics for each channel. By aggregating timestamp observations, the Emitter can estimate delay, which is an indicator for traffic congestion.

Using information about observed delays and retries, a range of congestion policies can be implemented, such as TCP-style backoff, or adjusting transmission rate based on observed delay, as in Low Extra Delay Background Transport (LEDBAT) [139].

In our estimation, based on the end-to-end principle, all decisions that could affect the application should be made at the application level. That does not mean that the developer

must make such decisions, just that they should be able to. Separating the system into layers makes those decisions inaccessible to higher layers of the system. Thus, we propose and demonstrate an architecture that uses encapsulation instead of layering to hide complexity. The adapter can provide default implementations of e.g., congestion control (possibly identical to TCP), but the developer can choose to override them at any time. In this way, flexibility can be gained without also increasing complexity for the developer.

Mandrake is applicable to any MAS since it makes no assumptions regarding the coupling between agents. That is, it addresses the most general setting where the agents are loosely coupled and can behave in any way they like. Given a protocol, Mandrake provides means to have the agents interact in a robust manner. Mandrake does impose some requirements on the interaction, however, through its adoption of BSPL; namely, BSPL requires a key-based information structure and so may not be compatible with existing systems. BSPL is also limited from doing certain kinds of multicast and streaming interactions. In future work Mandrake could be implemented to support BSPL extensions for multicast and streaming interactions.

Chapter 5

Atomicity

An interaction protocol specifies the constraints on communication between agents in a multiagent system. Ideally, we would like to be able to treat protocols as modules and compose them in a declarative manner to systematically build more complex protocols. Supporting composition correctly requires taking into account information-based causality relationships between protocols. One important problem that may arise from inadequate consideration of such relationships is that the enactment of a composite protocol may violate *atomicity*; that is, some components may be initiated but prevented from completing. We use the well-known *all or nothing* principle as the basis for formalizing atomicity as a novel correctness property for protocols.

Our contributions are the following. One, we motivate and formalize atomicity and highlight its distinctiveness from related correctness notions. Two, we give a decision procedure for verifying atomicity and report results from an implementation. For concreteness of exposition and technical development, we adopt BSPL as an exemplar of information causality approaches.

This chapter is adapted from the paper *Compositional Correctness in Multiagent Interactions*, published in the AAMAS conference proceedings of July 2018 [47]

5.1 Introduction

An interaction protocol specifies the rules of encounter between autonomous agents in a multiagent system. Two kinds of protocols have been studied extensively in the multiagent systems literature: protocols that specify constraints on message ordering and occurrence, and protocols that specify the meanings of interactions. We refer to the former as *operational* and the latter as *meaning-based*. RASA [112], HAPN [170], and UML interaction diagrams and its variants such as AUML [122] are among several languages developed to specify operational protocols. Meaning-based protocols are exemplified by work on commitment protocols [15, 73, 163, 172].

We are primarily concerned here with the *composition* of operational protocols. Ideally, we would like to be able to treat a protocol as a module, analogous to the way a program may be treated as a module, and *compose* it with other protocols to obtain more complex protocols. Further, we would like to be able to compose protocols *declaratively* on the basis of information-based causality constraints [148, 151]. Specifically, such constraints would capture what an agent must know (or not know) to produce a new piece of information. For example, in a typical sales transaction, one constraint would capture that a seller cannot fill an order until it knows what the order is. For concreteness, say there were a message schema *FillOrder* with a parameter order whose value reflected the details of the items ordered in the transaction. An instance of *FillOrder* could not be sent by the seller until it knew the details of the items ordered through a prior interaction in the transaction, say, through the receipt of an instance of the message schema *PlaceOrder* (sent by the buyer) that produces the value for order. Ignoring other details, the two specifications *PlaceOrder* and *FillOrder* may be composed into a protocol named, say *Purchase*, on the basis that *PlaceOrder*, by producing a binding for order, satisfies the causal constraint on *FillOrder* that the binding of order be known. Notice how the dependency induces a message ordering: in any instance of *Purchase*, *PlaceOrder* must be received by seller before it can send *FillOrder*.

Although the foregoing example composes message specifications, note that a message specification is an elementary protocol and the idea of causality-based composition applies to protocols in general. The Blindingly Simple Protocol Language (BSPL) [147] and its extensions such as Splee [43] are exemplars of the information-based causality approaches.

Compositions based on causality yield declarative specifications and support flexible enactments in fully decentralized asynchronous settings [148]. However, not all compositions would be desirable, as the following example demonstrates. Say the aforementioned protocol *Purchase* were extended with *Transfer* (pay by wire) and *Credit* (pay by credit card) protocols intended to offer a mutually exclusive choice between two payment methods. The specification

of *Purchase* would clearly be incorrect if both *Transfer* and *Credit* could be enacted in the same instance of *Purchase*. *Purchase* would also be incorrect, specifically, *nonatomic*, if there were enactments wherein *Transfer* had been initiated but was blocked from completion even though *Credit* had completed. Interestingly, it may be the completion of *Credit* itself that *blocks* the completion of *Transfer* (as a way of ensuring that only one payment method goes through). Such cases, where the enactment of *Transfer* is initiated but blocked, in effect, left *dangling*, may be indicative of semantic errors. For instance, the errors may the take the form of active commitments that cannot possibly be discharged.

To avoid undesirable compositions that could result in dangling enactments, we propose *atomicity* as a correctness criterion for protocols. The basis for atomicity is the observation that a message schema is *de facto* atomic: either its enactment occurs completely or not at all—there is nothing in between. For a composed protocol, the situation is not as straightforward, as we saw above for *Purchase*. Some constituents may complete, others may block. Lifting the idea of atomicity to protocols generally, we say that a composed protocol is atomic if and only if each constituent is atomic and completion of a constituent implies that the composed protocol can complete. Informally, our notion of atomic protocols is analogous to the notion of atomic programs [87] and transactions [75] in that it captures a notion of *all or nothing*. However, atomicity for programs and transactions is typically formulated in shared memory settings whereas we address decentralized settings.

Clearly, the problem in the foregoing example could be avoided if the buyer prudently enacts either *Transfer* or *Credit* but not both. Further, we would like to guarantee atomicity from the protocol specification alone, without resort to agent specifications. This motivation reflects the essential doctrine for protocols: capturing the *interaction logic* and presenting it in a reusable form [8, 113, 142].

It is important to note that we can only verify atomicity for a fully known composition. That is, a protocol verified to be atomic may cease to be so if used in composition with an incompatible protocol. For example, consider any protocol that has an ⌜out⌝ parameter (e.g. done) that is not bound by its first message. Then, using that protocol in composition with a simultaneously enabled protocol that binds done will block its completion, and violate atomicity. Our inability to assume atomicity will be preserved in an arbitrary composition is not a unique disadvantage; approaches for achieving atomicity in other domains (e.g. locks or transactions) are also useless for protecting atomicity if they are composed with an incompatible component that ignores them.

Also, our concept of atomicity can only be verified for the *protocol specification*, not the *system* that enacts it. Because we are dealing with both asynchrony and autonomy, there is no guarantee that any component will actually complete; an autonomous agent could abandon

an enactment partway and leave it hanging, or a message may be delayed indefinitely.

Nevertheless, verifying protocol atomicity still has value, for detecting conflicts between protocols in a composition. In this way,

We adopt BSPL to convey our ideas concretely and formally. Our contributions are the following:

- We motivate atomicity for information protocols and provide examples and patterns of atomicity violations and their corrections (Section 5.2).
- We formalize atomicity for information-based protocols (Section 5.3).
- We give a decision procedure for verifying atomicity and describe an implementation of the decision procedures (Section 5.4). We report results from running the implementation on examples in the present chapter.

Finally, Section 5.5 discusses related work and future directions.

5.2 Atomicity Concepts

Consider the protocol *Purchase* in Listing 5.1. In *Purchase*, a buyer places an order and then tenders payment via either wire transfer or credit. *Transfer* and *Credit* conflict because both produce a binding for payment. The conflict makes them mutually exclusive: either B can send *Transfer* or *Credit* but not both. *Purchase* is both live and safe.

Listing 5.1: Purchase with payment options

```
Purchase {
  roles B, S // Buyer, Seller
  parameters out orderID key, out product
  B ↦ S: PlaceOrder[out orderID]
  B ↦ S: Transfer[in orderID, out payment]
  B ↦ S: Credit[in orderID, out payment]
  S ↦ B: FillOrder[in orderID, in payment, out product]
}
```

Purchase is also atomic in the given (isolated) composition; to see why, recall the intuition behind atomicity from Section 5.1: (1) a protocol is atomic if each of its constituent protocols is atomic and (2) the completion of a component implies the composition can complete. Let's apply these criteria to *Purchase*. In *Purchase*, all of its references *PlaceOrder*, *Transfer*,

Credit, and *FillOrder* are atomic by virtue of being message schemas, thus satisfying (1) above. Further, any enactment where any of them occurs can be completed, thus satisfying (2) above. To see this, consider that even though *Transfer* and *Credit* are mutually exclusive (and block the other from being sent), they both enable *FillOrder* and thus completion of a given enactment.

However, some conflicts violate atomicity if they occur between protocols *after* they have both been initiated and prevent one of them from completing. Consider *RefinedPurchase* in Listing 5.2, in conjunction with the definitions for *RefinedTransfer* and *RefinedCredit* in Listing 5.3.

Listing 5.2: Refined purchase protocol

```
RefinedPurchase {
  roles B, S // Buyer, Seller
  parameters out order key, out payment, out product
  B ↦ S: PlaceOrder[out order]
  RefinedTransfer(B, S, in order, out payment)
  RefinedCredit(B, S, in order, out payment)
  S ↦ B: FillOrder[in order, in payment, out product]
}
```

Listing 5.3: Refined payment protocols

```
RefinedTransfer {
  roles B, S
  parameters in order key, out payment
  B ↦ S: OfferTransfer[in order, out transferOffer]
  S ↦ B: AcceptTransfer[in order, in transferOffer, out
     transferAccepted]
  B ↦ S: InitiateTransfer[in order, in transferAccepted, out
     payment]
}
RefinedCredit {
  roles B, S
  parameters in order key, out payment
  B ↦ S: OfferCredit[in order, out creditOffer]
  S ↦ B: AcceptCredit[in order, in creditOffer, out accept]
  B ↦ S: PayCredit[in order, in accept, out payment]
}
```

In Listing 5.3, the *OfferTransfer* and *OfferCredit* messages do not conflict with each other, as neither binds any parameters that would prevent the other from being sent. This lack of conflict means that both *RefinedTransfer* and *RefinedCredit* can be initiated in the same enactment. The seller may additionally send *AcceptTransfer* and *AcceptCredit* without conflict. However, the buyer is prevented from sending both *InitiateTransfer* and *PayCredit* because both produce bindings for payment. Thus both protocols can be initiated but only one can be completed, violating atomicity.

This violation of atomicity indicates that the specification is flawed. Perhaps the *RefinedTransfer* and *RefinedCredit* protocols should be mutually exclusive as in the original *Purchase* protocol, requiring the buyer to initiate only one of them. Alternatively, some mechanism for canceling one of the two should be added, or the protocols should be modified so they do not conflict with each other, as would be the case if the buyer is allowed to split the payment across multiple methods.

For example, the *FixedPurchase* variant of *Purchase* adds the offer parameter to *OfferTransfer* and *OfferCredit* messages as in Listing 5.4, so that the buyer may initiate only one of the two payment protocols. Since only one of the protocols can be initiated the other cannot prevent it from completing, so atomicity is restored.

Listing 5.4: *FixedPurchase* with explicit choice

```
B ↦ S: OfferTransfer[in order, out offer, out transferOffer]
B ↦ S: OfferCredit[in order, out offer, out creditOffer]
```

Not all atomicity violations require mutual exclusion. Some can be resolved by an alternative path to completion that avoids the conflict. For example, consider *ShareHealthData* in Listing 5.5.

In *ShareHealthData*, a patient can *Authorize* a clinic to share their data with researchers, until they subsequently *Revoke* access. While access is granted a researcher may *Request* the data, which the clinic sends them via *Provide*. However, because of the ⌜nil⌝ adornment, *Provide* cannot be sent after revoke is bound. Since data must be bound to complete *AccessData*, atomicity is violated in enactments where the patient sends *Revoke* after *Request* occurs but before *Provide*.

The conflict between ⌜out⌝ and ⌜nil⌝ parameters causes the protocols to be partially ordered rather than mutually exclusive: the *AccessData* component can be enacted completely as long as actions are performed in the correct sequence.

The patient should be able to revoke access even if there is a pending request. To enable this possibility without violating atomicity, the clinic should be able to complete *AccessData* without sending *Provide*, such as by rejecting the request. Listing 5.6 adds a reject message

Listing 5.5: Sharing private health records

```
ShareHealthData {
  roles P, R, C // patient, researcher, clinic
  parameters out ID key, out granted, out revoked
  P ↦ C: Authorize[out ID, out granted]
  AccessData(R, C, in ID, in granted, nil revoked, out data)
  P ↦ C: Revoke[in ID, in granted, out revoked]
}
AccessData {
  roles R, C // researcher, clinic
  parameters in ID key, in granted, nil revoked, out data
  R ↦ C: Request[in ID, in granted, out req]
  C ↦ R: Provide[in ID, in req, out data, nil revoked]
}
```

to *AccessData* to restore atomicity. (data in this case would be empty or a rejection message)

Listing 5.6: Alternative path to complete *AccessData*

```
  C ↦ R: Reject[in ID, in req, out data, in revoked]
```

Based on the kinds of conflicts that are possible with simple causal relationships as expressed in BSPL, we have identified several kinds of atomicity violations, illustrated by the above examples and summarized in Table 5.1. Exclusion conflicts involving either an ⌜in⌝ or ⌜out⌝ conflicting with an ⌜out⌝, such as that in *RefinedPurchase*, can be resolved by only enabling one of the conflicting protocols. Ordering conflicts involving an ⌜in⌝ or ⌜out⌝ conflicting with a ⌜nil⌝, such as that in *ShareHealthData*, can additionally be resolved by specifying an alternative path to completion. Indirect conflicts involving an ⌜in⌝ parameter and an ⌜out⌝ or ⌜nil⌝ parameter are equivalent to direct conflicts involving an ⌜out⌝ and either ⌜out⌝ or ⌜nil⌝ respectively; the information simply passes through at least one intermediary before the conflict becomes apparent. Because these are the only combinations of BSPL parameter types that produce conflicts, this list is exhaustive.

Table 5.1: Atomicity violations and their resolutions.

Violation	Cause	Resolution
Mutual Exclusion	⌜out⌝&⌜out⌝	Enable only one
Indirect Exclusion	⌜in⌝&⌜out⌝	
Partial Ordering	⌜out⌝&⌜nil⌝	Provide other means of completion
Indirect Ordering	⌜in⌝&⌜nil⌝	

5.3 Atomicity Formalization

We define $ref(\mathcal{P})$ as the set of references of $\mathcal{P}$.

Additionally, we use $\tau \preceq \tau'$ to mean that the history vector τ' is an extension of τ obtained by appending at most a finite number emissions and receptions.

$[[\mathcal{R}]] \sqsubseteq [[\mathcal{Q}]]$ means $\forall \tau \in [[\mathcal{R}]],\ \exists \tau' \in [[\mathcal{Q}]]$ such that $\tau \preceq \tau'$.

Definition 16*:* A protocol $\mathcal{Q}$ is atomic in universe of discourse U if and only if $\forall \mathcal{R} \in ref(\mathcal{Q})$,

1. $\mathcal{R}$ is atomic in U, and
2. $[[\mathcal{R}]]_U \sqsubseteq [[\mathcal{Q}]]_U$

"$\mathcal{P}$ is atomic" or "the atomicity of $\mathcal{P}$" are shorthand for the atomicity of $\mathcal{P}$ in its own universe of discourse.

Although the definition considers only direct references, its recursive nature means that if any message is sent, every composition that includes it must eventually complete. This definition captures our intuition that initiating a component protocol should result in its eventual completion, all the way from the individual messages to the highest level composition.

The intension $[[\mathcal{Q}]]$ of protocol $\mathcal{Q}$ is the set of enactments that complete $\mathcal{Q}$ by the emission of at least one message from the cover of each of its ⌜out⌝ parameters. The universe of discourse specifies which roles and messages are involved in the enactments. Using the universe of discourse of a composition $\mathcal{P}$ that includes $\mathcal{Q}$ means that conflicts can occur between messages anywhere in the composition, rather than just within the one component protocol.

For example, $[[\mathit{Transfer}]]_{UoD(\mathit{Purchase})}$ projected to role *B* is:

{[*PlaceOrder*, *Transfer*], [*PlaceOrder*, *Transfer*, *FillOrder*]} For this intension, each history vector in $[[\mathit{Transfer}]]_{UoD(\mathit{Purchase})}$ can be extended by a message reception to a history vector that completes *Purchase*, and the same is true for the other roles and components, so *Purchase* is atomic.

Conversely, $[[\mathit{OfferTransfer}]]_{UoD(\mathit{RefinedPurchase})}$ contains the enactment [*PlaceOrder*, *OfferCredit*, *OfferTransfer*, *AcceptCredit*, *PayCredit*] which cannot be extended to an enactment that completes *Transfer*, so *RefinedPurchase* is not atomic.

Because the cover of a protocol contains only messages within the protocol, each component protocol must be completed by its own messages to be atomic. Even if an enactment produces the same parameters via messages from another component $\mathcal{Q}'$, it is not in the intension of $\mathcal{Q}$ because its cover is not complete. Thus *Credit* is not completed by the binding of payment produced by *InitiateTransfer* because that message is not in the cover of *Credit*.

5.3.1 Relationships with Existing Properties

Although atomicity is orthogonal to the other properties, there are some connections. The following theorem shows that a protocol is atomic if it is not only nonlive, but none of its messages is enactable.

Theorem 1*:* A protocol is atomic if its universe of enactments is empty.

Proof. If the universe of enactments of protocol $\mathcal{P}$ is empty, then $\forall m \in \text{msgs}(\mathcal{P}), [\![m]\!]_{UoD(\mathcal{P})} = \emptyset$. Similarly, $[\![\mathcal{P}]\!]_{UoD(\mathcal{P})} = \emptyset$

Let a protocol of height 0 be a message schema which is an elementary protocol and therefore atomic.

Let a protocol of height $n+1$ be a protocol which references protocols of height n or less.

Suppose protocols of height n or less with an empty universe of enactments are atomic. Let $\mathcal{Q}$ be a protocol of height $n+1$ with an empty universe of enactments. Then all references $\mathcal{R} \in ref(\mathcal{Q})$ are of height n or less, and $\mathcal{U}_{UoD(\mathcal{R})} \subseteq \mathcal{U}_{UoD(\mathcal{Q})} = \emptyset$, and so have empty universes of enactments. Thus $\forall \mathcal{R} \in ref(\mathcal{Q}), \mathcal{R}$ is atomic by assumption and $[\![\mathcal{R}]\!]_{UoD(\mathcal{Q})} \subseteq \mathcal{U}_{UoD(\mathcal{Q})} = \emptyset$, so $[\![\mathcal{R}]\!]_{UoD(\mathcal{Q})} \sqsubseteq [\![\mathcal{Q}]\!]_{UoD(\mathcal{Q})}$, and $\mathcal{Q}$ is atomic.

By induction, any protocol with an empty universe of enactments is atomic. □ □

Conversely, if a protocol is both enactable and atomic then it must be live. Liveness concerns the protocol as a whole and guarantees it can always complete, while atomicity concerns compositions of distinct component protocols, recursively ensuring that each will complete if initiated.

Theorem 2*:* Any protocol that is enactable and atomic is live.

Proof. A protocol $\mathcal{P}$ is live if for each $\tau \in \mathcal{U}_{\text{UoD}(\mathcal{P})}$, $\exists \tau' \in [\![\mathcal{P}]\!]_{UoD(\mathcal{P})}$ such that $\tau \preceq \tau'$.

Suppose protocol $\mathcal{P}$ is enactable. Then $\exists \tau \in [\![\mathcal{P}]\!]_{UoD(\mathcal{P})}$.

Suppose υ is a history vector in $\mathcal{U}_{\text{UoD}(\mathcal{P})}$. Then either υ is empty, or $\upsilon \in [\![m]\!]_{UoD(\mathcal{P})}$ for some message $m \in \text{msgs}(\mathcal{P})$.

If υ is empty, then $\upsilon \preceq \tau \in [\![\mathcal{P}]\!]_{UoD(\mathcal{P})}$.

If $m \in ref(\mathcal{P})$, then by the definition of atomicity $\exists \upsilon' \in [\![\mathcal{P}]\!]_{UoD(\mathcal{P})}$ such that $\upsilon \preceq \upsilon'$.

If $m \notin ref(\mathcal{P})$, then $\exists \mathcal{Q} \in ref(\mathcal{P})$ such that $m \in \text{msgs}(\mathcal{Q})$ and by atomicity $[\![\mathcal{Q}]\!]_{UoD(\mathcal{P})} \sqsubseteq [\![\mathcal{P}]\!]_{UoD(\mathcal{P})}$. By induction $[\![m]\!]_{UoD(\mathcal{P})} \sqsubseteq [\![\mathcal{P}]\!]_{UoD(\mathcal{P})}$. Then by the definition of atomicity $\exists \upsilon' \in [\![\mathcal{P}]\!]_{UoD(\mathcal{P})}$ such that $\upsilon \preceq \upsilon'$.

Thus in each case υ can be extended to a history vector in $[\![\mathcal{P}]\!]_{UoD(\mathcal{P})}$, so $\mathcal{P}$ is live. □ □

5.4 Verification

We have built a tool named for automatically checking whether or not a protocol specification is atomic, to demonstrate that protocol atomicity is not just a theoretical property of protocols.

The verification process is similar to the method used by Singh [149]. We used a simple temporal logic to represent the definitions and constraints required for atomicity. The temporal logic itself was then implemented on top of a boolean logic solving library.

In this approach, each event is represented as a boolean variable. These events are then combined into expressions representing the integrity constraints and desired properties of the protocol. Finally, the expressions are evaluated using the *boolexpr* SAT solving library for Python.

5.4.1 Logic: Syntax and Semantics

The temporal logic language we adopt, Precedence, was used by Singh to verify the BSPL correctness properties of enactability, safety, and liveness [146, 149].

The atoms of Precedence are events. Below, e and f are events. If e is an event, its complement $\overline{e}$ is also an event. $\overline{e}$ is not the simple negation or non-occurrence of e, but an event indicating that e can never occur in the future. The terms $e \cdot f$ and $e \star f$, respectively, mean that e occurs prior to f and e and f occur simultaneously. The Boolean operators: '$\vee$' and '$\wedge$' have the usual meanings. The syntax follows conjunctive normal form:

L_1. $I \longrightarrow \ clause \mid clause \wedge I$

L_2. $clause \longrightarrow \ term \mid term \vee clause$

L_3. $term \longrightarrow \ event \mid event \cdot event \mid event \star event$

The semantics of Precedence is given by pseudolinear runs of events (instances): "pseudo" because several events may occur together though there is no branching. Let Γ be a set of events where $e \in \Gamma$ if and only if $\overline{e} \in \Gamma$. A run is a function from natural numbers to the power set of Γ, that is, $\tau : \mathbb{N} \mapsto 2^{\Gamma}$. The i^{th} index of τ, $\tau_i = \tau(i)$. The length of τ is the first index i at which $\tau(i) = \emptyset$ (after which all indices are empty sets). We say τ is empty if $|\tau| = 0$. The subrun from i to j of τ is notated $\tau_{[i,j]}$. Its first $j - i + 1$ values are extracted from τ and the rest are empty, that is, $\tau_{[i,j]} = \langle \tau_i, \tau_{i+1} \ldots \tau_{j-i+1} \ldots \emptyset \ldots \rangle$. On any run, e or $\overline{e}$ may not both occur. Events are nonrepeating.

$\tau \models_i E$ means that τ satisfies E at i or later. We say τ is a *model* of expression E if and only if $\tau \models_0 E$. E is *satisfiable* if and only if it has a model.

M_1. $\tau \models_i e$ if and only if $(\exists j \geq i : e \in \tau_j)$

M_2. $\tau \models_i e \star f$ if and only if $(\exists j \geq i : \{e, f\} \subseteq \tau_j)$

M_3. $\tau \models_i r \vee u$ if and only if $\tau \models_i r$ or $\tau \models_i u$

M_4. $\tau \models_i r \wedge u$ if and only if $\tau \models_i r$ and $\tau \models_i u$

M_5. $\tau \models_i e \cdot f$ if and only if $(\exists j \geq i : \tau_{[i,j]} \models_0 e$ and $\tau_{[j+1,|\tau|]} \models_0 f)$

5.4.2 Causality

We first define a set of clauses, $\mathcal{C}_P$, which can be automatically derived from a protocol specification to represent the fundamental causal semantics of BSPL enactments. Let $\mathcal{C}_P$ be the conjunction of all clauses of the following types, illustrated with examples from *Purchase* in Listing 5.1.

1. Transmission: Each message must be sent to be received. (4 clauses)

   $\overline{\mathsf{S{:}PlaceOrder}} \vee \mathsf{B{:}PlaceOrder}$

2. Emission: A message cannot be sent if its ⌜out⌝ or ⌜nil⌝ parameters have already been observed or if its ⌜in⌝ parameters are not observed. (4 clauses)

   $\overline{\mathsf{S{:}FillOrder}} \vee (\mathsf{S{:}order} \wedge \mathsf{S{:}payment} \wedge \overline{\mathsf{S{:}product}})$

3. Reception: Either a message is not received or its ⌜out⌝ and ∈ parameters are observed no later than the message. (4 clauses)

   $\overline{\mathsf{S{:}PlaceOrder}} \vee \mathsf{S{:}order} \cdot \mathsf{S{:}PlaceOrder} \vee \mathsf{S{:}order} \star \mathsf{S{:}PlaceOrder}$

4. Minimality: For any role, if a parameter occurs, it occurs simultaneously with some message emitted or received. No role observes a parameter noncausally. (6 clauses)

   $\overline{\mathsf{S{:}product}} \vee \mathsf{S{:}product} \star \mathsf{S{:}FillOrder}$

5. Nonsimultaneity: A role cannot emit messages simultaneously; they are sent in some order. (4 clauses)

   $\overline{\mathsf{B{:}Transfer}} \vee \overline{\mathsf{B{:}Credit}} \vee \mathsf{B{:}Transfer} \cdot \mathsf{B{:}Credit} \vee \mathsf{B{:}Credit} \cdot \mathsf{B{:}Transfer}$

Based on the semantics of BSPL, an enactment of a protocol is valid if and only if it satisfies $\mathcal{C}_P$. According to Singh [149], given a well-formed protocol $\mathcal{P}$, for every viable history vector, there is a model of $\mathcal{C}_P$ and vice versa.

5.4.3 Maximality

To support unbounded enactments and exclude failures caused by noncompliant agent behavior or transmission errors, we assume that each enactment is *maximal*. That is, every message will be sent and received unless it is prevented by an unmet precondition, such as an unavailable ⌜in⌝, or an observed ⌜out⌝ or ⌜nil⌝. The clause generated for the Transfer message is (B:Transfer ∨ $\overline{\text{B:order}}$ ∨ B:payment); either *Transfer* is transmitted, the prerequisite order is not observed, or a binding for payment already exists. We label the conjunction of these clauses for each message in protocol $\mathcal{P}$ as $\mathcal{M}_P$. By definition, an enactment is maximal if and only if it satisfies $\mathcal{M}_P$.

If an enactment satisfies maximality yet is still incomplete, then it truly cannot be completed; there must be something other than intransigent agents or network failure preventing completion, namely, the protocol specification. This intuition is the basis of the liveness definition and verification technique used by Singh [149].

5.4.4 Enactability

We additionally construct clauses representing the enactability of a protocol $\mathcal{P}$, labeled $\mathcal{E}_P$.

An enactment satisfies $\mathcal{E}_P$ if and only if it completes protocol $\mathcal{P}$. A protocol is complete when each of its ⌜out⌝ parameters is produced by one of its messages, as outlined in Figure 1. For Purchase, the set of messages covering order is {*PlaceOrder*}, and the cover of product is {*FillOrder*}. Thus the resulting clause $\mathcal{E}_{\text{Purchase}}$ is (S:PlaceOrder ∧ B:FillOrder).

An algorithm generating $\mathcal{E}_P$ is given in Figure 1, though it is written using Q as the protocol name for readability. The algorithm iterates over each ⌜out⌝ parameter p of a protocol Q. The occurrence of any message m in Q that has p as an ⌜out⌝ parameter produces a binding for p, so the disjunction of all such occurrences (here labeled cover_p) accounts for all ways to bind p. As an enactment of Q is complete when all of its ⌜out⌝ parameters is bound, we return the conjunction of all cover_p as $\mathcal{E}_Q$.

Algorithm 1: Algorithm generating $\mathcal{E}_Q$.

```
1 Function E(Q)
2     forall p ∈ out(Q) do
3         cover_p ← ⋁{m ∈ Q | p ∈ out(m)}
4     return ⋀_{p∈out(Q)} cover_p
```

5.4.5 Atomicity

Suppose $\mathcal{P}$ is a composition that includes (or may be identical to) some protocol $\mathcal{Q}$ with reference $\mathcal{R}$. Atomicity of a protocol requires both that each of its references be atomic, and that if the reference completes the protocol can also complete. Using the definition for $\mathcal{E}_P$ given above, the statement that the completion of $\mathcal{R}$ implies the completion of $\mathcal{Q}$ can be written as $\mathcal{E}_R \Rightarrow \mathcal{E}_Q$. Thus if $\mathcal{Q}$ is atomic in $UoD(\mathcal{P})$, any valid enactment of $\mathcal{Q}$ will satisfy the formula $\mathcal{C}_P \wedge \mathcal{M}_P \wedge (\neg\mathcal{E}_R \vee \mathcal{E}_Q)$ for all $\mathcal{R} \in ref(\mathcal{Q})$.

To prove that a composition is atomic, we verify that there are no valid nonatomic enactments by inverting the atomicity clause. Doing so yields the following formula: $\mathcal{C}_P \wedge \mathcal{M}_P \wedge \mathcal{E}_R \wedge \neg\mathcal{E}_Q$. If this formula cannot be satisfied for any R, then there are no valid, maximal, nonatomic enactments, and $\mathcal{Q}$ must be atomic in $UoD(\mathcal{P})$.

With this formula, we can recursively verify the atomicity of every component in composition $\mathcal{P}$, as shown by the algorithm in Figure 2.

In the base case, messages are always atomic and do not have any components for further recursion. The algorithm then iterates across each reference $\mathcal{R}$ in $\mathcal{Q}$, recursively testing for atomicity and checking that there are no enactments in which $\mathcal{R}$ completes but $\mathcal{Q}$ does not.

Algorithm 2: Atomicity verification process.

```
1 Function atomic(Q, P)
2     if Q is a message then
3         return True
4     forall R ∈ ref(Q) do
5         if ¬atomic(R, P) then
6             return False
7         if SAT(C_P ∧ M_P ∧ E_R ∧ ¬E_Q) then
8             return False
9     return True
```

Having designed a verification procedure, we now prove it correctly verifies that a protocol $\mathcal{Q}$ is atomic in $UoD(\mathcal{P})$ assuming that the clauses $\mathcal{C}_P$, $\mathcal{M}_P$, and $\mathcal{E}_P$ correctly capture correctness, maximality, and enactability.

Theorem 3: A protocol $\mathcal{Q}$ is *atomic* in $UoD(\mathcal{P})$ if and only if, for all $\mathcal{R}$ in $ref(\mathcal{Q})$, $\mathcal{R}$ is atomic and there is no enactment τ which satisfies $\mathcal{C}_P \wedge \mathcal{M}_P \wedge \mathcal{E}_R \wedge \neg\mathcal{E}_Q$.

Proof. Let the atomicity of $\mathcal{Q}$ be denoted A_Q. We desire to prove $\forall\mathcal{R} \in ref(\mathcal{Q}) : \text{atomic}(\mathcal{R}, \mathcal{P}) \Rightarrow A_Q \Leftrightarrow \neg(\mathcal{C}_P \wedge \mathcal{M}_P \wedge \mathcal{E}_R \wedge \neg\mathcal{E}_Q)$. First, we assume from the definition that $\mathcal{R}$ is atomic, and

consider only enactments that satisfy $C_P \wedge \mathcal{M}_P$ (that is, are correct and maximal), leaving $A_Q \Leftrightarrow \neg(\mathcal{E}_R \wedge \neg\mathcal{E}_Q)$. Distributing the negation reduces the statement to $\forall \mathcal{R} \in \mathit{ref}(\mathcal{Q}) : A_Q \Leftrightarrow \neg\mathcal{E}_R \vee \mathcal{E}_Q$.

Suppose protocol $\mathcal{Q}$ is atomic in $UoD(\mathcal{P})$. By the definition of atomicity we know that for all $\mathcal{R} \in \mathit{ref}(\mathcal{Q})$, any enactment τ in intension $[[\mathcal{R}]]_{UoD(\mathcal{P})}$ can be extended with a finite number of message transmissions to some enactment τ' in intension $[[\mathcal{Q}]]_{UoD(\mathcal{P})}$. That is, by assuming maximality, if it completes $\mathcal{R}$ it also completes $\mathcal{Q}$. So enactment τ' satisfies $\mathcal{E}_\mathcal{R} \wedge \mathcal{E}_\mathcal{Q}$, and $A_\mathcal{Q} \Rightarrow \neg\mathcal{E}_R \vee \mathcal{E}_\mathcal{Q}$.

Conversely, suppose for protocol $\mathcal{Q}$ and all $\mathcal{R} \in \mathit{ref}(\mathcal{Q})$, enactment τ satisfies $\neg\mathcal{E}_\mathcal{R} \vee \mathcal{E}_\mathcal{Q}$. By the definition of $\mathcal{E}_\mathcal{P}$, either τ completes $\mathcal{Q}$ or it does not complete $\mathcal{R}$. Thus, it is in the intension $[[\mathcal{Q}]]_{UoD(\mathcal{P})}$ or it is not in the intension $[[\mathcal{R}]]_{UoD(\mathcal{P})}$. Furthermore, by the assumption of maximality $\nexists \tau' \in [[\mathcal{R}]] \ni \tau \preceq \tau'$, the definition of atomicity is vacuously satisfied, and $\neg\mathcal{E}_\mathcal{R} \vee \mathcal{E}_\mathcal{Q} \Rightarrow A_\mathcal{Q}$. □

□

5.4.6 Results

Table 5.2: Verification results for example protocols.

Protocol (Listing)	Atomic?	Clauses
Purchase (Listing 5.1)	True	92
RefinedPurchase (Listing 5.2)	False	156
FixedPurchase (Listing 5.4)	True	390
ShareHealthData (Listing 5.5)	False	102
CreateOrder	False	690

Table 5.2 displays an overview of the results of running our implementation on each of the example protocols in this chapter. The second column shows whether the protocol was verified as atomic or not. The third column shows how many clauses were generated according to the clause definitions given above. Note that the number of clauses varies because of the short-circuit nature of the recursive algorithm; *FixedPurchase* recurses through all of its components, while *RefinedPurchase* exits at the first violation.

The last entry, *CreateOrder*, is a specification of the Create Laboratory Order workflow defined by HL7 [94]. The workflow describes the process of a patient visiting a doctor and getting a sample collected, and a lab testing the sample and returning the results. A naïve

specification leaves several choices enabled when they should be mutually exclusive, as in the *RefinedPurchase* example. The first such choice is when the physician has the sample collected: they can collect it themselves, have a third party collect the sample before sending it to the lab, or have the lab collect the sample directly from the patient. A naïve composition of these three alternatives violates atomicity, because the physician can initiate more than one of them but ultimately only one sample should be collected. When translating the workflow into a protocol, this naïve approach was initially used to verify that our tool caught the violation. However, this protocol proved to be a better demonstration than first thought. After fixing the violation by making the collection protocols mutually exclusive, the tool still reported that atomicity was violated. After looking more closely, it was discovered that a second violation had been overlooked until caught by the tool. Near the end of the workflow, the lab can either directly send the results to the physician, or simply send a notification that the results are available. The atomicity violation would occur if the physician queried the results, but was then sent them directly instead of as a response, preventing the query protocol from completing. This incident shows that atomicity is a useful property for verifying the design of composite protocols.

5.5 Discussion

The idea of atomic action has a long history in multiagent systems; see, for example, Boissier et al. [29], Omicini and Zambonelli [124], and Alechina et al. [3]. However, unlike the present work, existing works either assume a shared memory or address a single-agent setting.

Notably, several diverse approaches for specifying protocols, including AUML [122], message sequence charts (MSCs) [96], choreography languages such as WS-CDL [171], and process calculi-inspired languages [6, 90] ignore information altogether and instead use control flow-based abstractions such as sequence, choice, and so on, to compose and constrain the enactment of protocols. However, specifying protocols in terms of control flow results in regimented enactments. Such lack of flexibility is true even of approaches such as RASA [112] and HAPN [170] that support declarative constraints on information values. Because the protocol enactments are guided by control flow constructs rather than information causality no conflicts can occur between components in a composition, just as the addition of states and transitions to a state machine does not affect the operation of other portions. For this reason we focus on the application of atomicity to the information-based approach exemplified by BSPL.

A direction for future work is to incorporate support for *relative atomicity* [107], so that a protocol is only required to complete after some critical subset is completed. Identifying

the commitments created or discharged by each message could allow the discharging of all commitments to be used as a more precise correctness criterion. Similarly, identifying relationships between message effects should enable more sophisticated conflict resolutions such as reversion.

In contrast to the early work on commitment protocols [168, 172], newer work on commitments and more generally meanings [40, 41] leaves out altogether concerns such as message ordering and occurence. The motivation is that meanings (of information) and causality are separate concerns. They are not disconnected however; as demonstrated in [100], the meanings are layered on top of information protocols. A rich direction is to explore further the connection between meaning-based and information protocols.

Chapter 6

Refinement

An interaction protocol specifies a decentralized multiagent system operationally by specifying constraints on messages exchanged by its member agents. Engineering with protocols requires support for a notion of *refinement*, whereby a protocol may be substituted without loss of correctness by one that refines it. We identify two desiderata for refinement. One, *generality*: refinement should not restrict enactments by limiting protocols or infrastructures under consideration. Two, *preservation*: to facilitate modular verification, refinement should preserve liveness and safety.

We contribute a novel formal notion of protocol refinement based on enactments. We demonstrate generality by tackling the declarative framework of information protocols. We demonstrate preservation by formally establishing that our notion of refinement is safety and liveness preserving. We show the practical benefits of refinement by implementing a checker. We demonstrate that it is less time-intensive to check refinement (and thereby gain safety and liveness) than to recheck safety and liveness of a composition.

This chapter is adapted from the paper *Refinement for Multiagent Protocols*, published in the AAMAS conference proceedings of May 2020 [46]

6.1 Introduction

We are concerned with multiagent systems (MAS) where each agent represents an autonomous real-world principal. Such systems are decentralized: agents exercise independent decision making and engage in arms-length communications with each other via asynchronous messaging. An interaction protocol enables agents in a MAS to coordinate their computations by specifying constraints on messaging, including the format, content, and conditions under which an agent may send and receive a message. A protocol being the primary operational specification for a MAS makes protocols crucial in engineering MAS. Notable approaches for specifying protocols include AUML [122], trace expressions (Trace) [6, 70], state machines [11], hierarchical state machines (HAPN) [170], session types [90, 91], and information protocols (BSPL) [147].

Protocols are doubly modular [157]. Along one dimension of modularity, each role of a protocol is a module. Role-as-module raises the question of when a role specification, derived from a protocol, may be substituted by another specification. A role R' *conforms* to role R if and only if R' can interoperate with any set of roles with which R can interoperate [11, 13, 67]. However, existing works don't handle interactions between three or more parties and several impose FIFO message delivery. Works that tackle multiparty settings are restricted to protocols in which only one agent can make an autonomous (internal) choice, thereby limiting concurrency.

Along the other dimension of modularity, a protocol is a module that may be composed with other protocols [112, 164]. Protocol-as-module raises the question of when a protocol P may be substituted in a composition by a protocol Q: Q must be a specialization of P.

Refinement captures a notion of specialization that legitimizes substitution. Suppose *Purchase* is a protocol that facilitates goods for payment transactions between buyers and sellers in which either buyer or seller may initiate a transaction. A refinement of *Purchase* could be *Purchase-Escrow*, where instead of paying the seller directly, the buyer pays an escrow service, which pays the seller when appropriate. Another refinement could be *Buyer-Initiates*, where only the buyer may initiate the transaction.

We identify two desirable features of protocol refinement: One, *generality*: refinement doesn't restrict protocols or the infrastructure. Two, *correctness preservation*: substitution by a refinement preserves correctness properties, specifically, liveness and safety, of the composition. That is, if the original composition satisfies liveness (or safety), then a composition resulting from the substitution must also satisfy liveness (or safety). Practically, checking refinement once saves the effort of checking every composition post substitution. Suppose we established that a composition using *Purchase* were safe. Then, substituting *Purchase-Escrow*

for *Purchase* in that composition would yield a safe composition.

Our contribution is two-fold. One, we contribute a notion of protocol refinement that satisfies both of the foregoing desiderata. We satisfy generality by formalizing refinement for *information protocols* [147, 149], which are declaratively described and whose semantics supports asynchronous messaging without ordering guarantees. We satisfy preservation by proving that substituting a constituent of a composition by the constituent's refinement ensures safety and liveness. We prove that the composition obtained from substituting a constituent by its refinement is itself a refinement of the original composition.

Two, we contribute an algorithm for refinement checking and its implementation in a tool. We show that checking refinement is no more time-intensive than checking liveness and safety, the properties preserved. The relative performance makes it worthwhile to check refinement once rather than checking the liveness and safety of each individual composition post substitution.

Section 6.2 motivates refinement with examples. Section 6.3 formalizes our definition. Section 6.4 presents important theorems. Section 6.5 presents an empirical evaluation of our tool. Section 6.6 discusses related work. Section 6.7 concludes with a discussion.

6.2 Motivating Protocol Refinement

Intuitively, we consider protocol Q to refine protocol P if its enactments are at least as elaborate with at most the same flexibility.

An enactment q of Q is *elaborates on* enactment p of P if q binds the parameters in p in the same order. By elaborating on p, q supports any actions or extensions that p does. A protocol is less flexible if there are fewer decisions or *alternative paths* available during an enactment. For example, in a flexible protocol either the buyer or seller may name the price, whereas a less flexible refinement may specify that only the seller may do so.

6.2.1 Enactment Paths

To frame the motivations of refinement using examples and later formalize the concepts, we first describe our intuitions and a notation for illustrating them.

A crucial intuition is that if Q refines P, then every enactment of Q *maps* to an enactment of P [108]. Q may have fewer enactments and its enactments may have more details, but each enactment binds the same parameters as some enactment of P.

Under BSPL's semantics, an enactment of a protocol is a vector of role histories, which can be denoted by $[r_1{:}h_1, r_2{:}h_2, \ldots, r_n{:}h_n]$, where n is the number of roles in the protocol, r_i is

the name of the ith role, and h_i is the history for role r_i. Each history is a sequence of message instances. Each message instance is a tuple of name, sender, receiver, and parameter-value pairs. A complete enactment is one in which all of the protocol's public parameters appear.

To more explicitly and concisely capture the choices made during an enactment, we project an enactment to a sequential view of events called a *path*. A path represents a possible perspective on the relative ordering of events: if two events A and B are independent then both [A, B] and [B, A] are valid paths. A path is a sequence of message instances, each with sender s, recipient r, key parameters $\vec{k}$, parameter list $\vec{p}$, and optional transmission offset o, written as $\langle s{\mapsto}r, \vec{k}|\vec{p}\rangle_o$. We omit the offset parameter when it is 0.

The only use of paths is to represent asynchronous enactments (history vectors) compactly. Suppose a path is $[m_0, m_1, \ldots]$. For a role R, order its emissions in the same order as in the path. If m_j with offset k is directed at R, the m_j arrives any time after all of R's emissions from m_0 to m_{j+k} have been made and before any of R's emissions from m_{j+k+1} onward is made.

To be a valid path, each message instance must be enabled by the information known to the sender from the subpath preceding it. A complete path is a path that represents a complete enactment; not all valid paths are complete. The set of all paths extending a path is its set of *branches*; that is, paths with that path as a prefix. For brevity, we do not duplicate the keys in the parameter list, and show only the parameters introduced by (that is, observed by the sender or recipient for the first time in) a message.

Listing 6.1 shows an example path for *Purchase*. Three of the messages have a offset of 0, meaning they are received before the next message is sent. The third message, introducing a binding for the parameter payment, has a offset of 1 to indicate that it is not received until *after* the following message emission occurs, capturing the case where *pay* and *ship* are being transmitted concurrently, since they are received at the same point in the path.

Listing 6.1: An example path in Purchase.

$$
\begin{aligned}
&[\langle b{\mapsto}s, \mathsf{ID} \mid \mathsf{item}\rangle_0\ ,\\
&\ \langle s{\mapsto}b, \mathsf{ID} \mid \mathsf{price}\rangle_0\ ,\\
&\ \langle b{\mapsto}s, \mathsf{ID} \mid \mathsf{payment}\rangle_1\ ,\\
&\ \langle s{\mapsto}b, \mathsf{ID} \mid \mathsf{deliver}\rangle_0]
\end{aligned}
$$

The following sections give examples of how a refinement may be less flexible or more elaborate than the protocol it refines.

6.2.2 Running Example

We illustrate several kinds of refinement via variants of the following protocol, the overall composition of which is given in Listing 6.2.

Listing 6.2: Composition for developing further examples.

```
Commerce {
  roles B, S, C // Buyer, Seller, Catalog
  parameters out ID key, out item, out shipped
  private price, payment

  Either-Starts(B, S, out ID key, out item)
  Lookup-Prices(S, C, in ID key, out query key, out price)
  S ↦ B: quote[in ID key, in item, in price]
  Flexible-Payment(B, S, in ID key, in item, in price, out payment,
     out shipped)
}
```

This composition *references*, i.e., includes as constituents, three protocols. *Either-Starts* specifies the beginning of the enactment, where either the buyer or the seller can request a quote or recommend an item, respectively. Then, the seller queries the catalog for the price information regarding the item being sold in *Lookup-Prices*. The *quote* message forwards the price information to the buyer, so it can proceed with payment. And, *Flexible-Payment* describes the conclusion of the transaction, in which the buyer can pay before or after the seller ships the product.

6.2.3 Polymorphism Reduction

The basic RFQ protocol in Listing 6.3 is a refinement of the polymorphic RFQ protocol given in Listing 2.4.

Listing 6.3: Simple RFQ protocol.

```
RFQ {
  roles B, S
  parameters out ID key, out item, out price
  B ↦ S: rfq[out ID key, out item]
  S ↦ B: offer[in ID key, in item, out price]
}
```

The complete paths of *Polymorphic-RFQ* are:

$[\langle B{\mapsto}S, \mathsf{ID} \mid \rangle, \langle S{\mapsto}B, \mathsf{ID} \mid \mathsf{item}, \mathsf{price}\rangle]$,
$[\langle B{\mapsto}S, \mathsf{ID} \mid \mathsf{item}\rangle, \langle S{\mapsto}B, \mathsf{ID} \mid \mathsf{price}\rangle]$

The only complete path of *RFQ* is:

$[\langle B{\mapsto}S, \mathsf{ID} \mid \mathsf{item}\rangle, \langle S{\mapsto}B, \mathsf{ID} \mid \mathsf{price}\rangle]$

Thus the paths of *RFQ* are a subset of the paths of *Polymorphic-RFQ*, so *RFQ* is a refinement.

6.2.4 Initiation Reduction

Initiation reduction removes alternative initiating messages that may be sent by different roles. Consequently, the resulting protocol has a subset of otherwise identical paths.

Listing 6.4: Either-Starts protocol.

```
Either-Starts {
  roles B, S
  parameters out ID key, out item
  B ↦ S: rfq(out ID key, out item)
  S ↦ B: recommend(out ID key, out item)
}
//complete paths: [⟨B↦S, ID | item⟩], [⟨S↦B, ID | item⟩]
```

In *Either-Starts* in Listing 6.4, both B and S have the option to send the initiating message, since both *rfq* and *recommend* produce a binding for the key parameter ID.

Listing 6.5: Buyer-Starts refinement of Either-Starts.

```
Buyer-Starts {
  roles B, S
  parameters out ID key, out item
  B ↦ S: rfq(out ID key, out item)
}
//complete path: [⟨B↦S, ID | item⟩]
```

Buyer-Starts in Listing 6.5 is a refinement of *Either-Starts*, because it selects only one of the two possible enactments.

6.2.5 Key Parameter Reduction

Demoting a key parameter, i.e., turning it into a non-key parameter, strengthens the original key constraints. Thus demoting a key parameter reduces flexibility, and is a valid refinement. (A valid protocol must have at least one key.)

Consider Listing 6.6, which has two key parameters.

Listing 6.6: Multiple lookups via two key parameters.

```
Lookup-Prices {
  roles S, C
  parameters in ID key, out query key, out price
  S -> C: lookup[in ID key, out query key]
  C -> S: result[in ID key, in query key, out price]
}
// possible paths
[..., ⟨S↦C, [ID, query] | query⟩, ⟨C↦S, [ID, query] | price⟩]
[..., ⟨S↦C, [ID, query] | query⟩, ⟨S↦C, [ID, query] | query⟩,
 ⟨C↦S, [ID, query] | price⟩, ⟨C↦S, [ID, query] | price⟩]
...
```

The paths are shown with an elided prefix, because there are ⌜in⌝ parameters that need to be bound by other messages to enable *lookup*. Because *Lookup-Prices* has a composite key, each parameter need only have a unique binding for each pair of bindings of ID and query. Thus, we explicitly include some of the normally elided key bindings to show that the second path contains multiple bindings for query, and can have multiple corresponding prices.

Listing 6.7: Protocol with reduced keys.

```
Single-Lookup {
  roles S, C
  parameters in ID key, in item, out price
  S -> C: lookup[in ID key, in item]
  C -> S: result[in ID key, in item, out price]
}
// only one complete path
[..., ⟨S↦C, ID | item⟩, ⟨C↦S, ID | price⟩]
```

Single-Lookup in Listing 6.7 is a refinement of *Lookup-Prices*, because the one path of *Single-Lookup* corresponds to a path of *Lookup-Prices*, and its keys are a subset of the keys of *Lookup-Prices*.

6.2.6 Concurrency Elimination

Listing 6.8 specifies *Flexible-Purchase*, in which the order that the buyer and seller respectively pay and deliver is flexible.

Listing 6.8: Purchase with concurrent payment and delivery.

```
Flexible-Purchase {
 roles B, S
 parameters in ID key, in item, in price, out payment, out shipped

 B ↦ S: pay[in ID key, in item, in price, out payment]
 S ↦ B: ship[in ID key, in item, in price, out shipped]
}
```

Flexible-Purchase does not specify a dependency relationship between *pay* and *ship*, so they are not ordered and any enactment may complete in one of three ways.

1. B sends *pay* after receiving *ship*, yielding the path:
   $[\ldots, \langle S \mapsto B, \mathsf{ID} \mid \mathsf{shipped} \rangle_0, \langle B \mapsto S, \mathsf{ID} \mid \mathsf{payment} \rangle_0]$

2. S sends *ship* after receiving *pay*. This enactment has the path:
   $[\ldots, \langle B \mapsto S, \mathsf{ID} \mid \mathsf{payment} \rangle_0, \langle B \mapsto S, \mathsf{ID} \mid \mathsf{shipped} \rangle_0]$

3. B and S concurrently send *pay* and *ship*, respectively, yielding either of the following two paths:
   $[\ldots, \langle B \mapsto S, \mathsf{ID} \mid \mathsf{payment} \rangle_1, \langle S \mapsto B, \mathsf{ID} \mid \mathsf{shipped} \rangle_0]$
   $[\ldots, \langle S \mapsto B, \mathsf{ID} \mid \mathsf{shipped} \rangle_1, \langle B \mapsto S, \mathsf{ID} \mid \mathsf{payment} \rangle_0]$

Having a nonzero offset captures asynchrony, by allowing another message to be sent before the previous message is received. In the first path, for example, *pay* is sent before *ship* but has a offset of 1, so they are both received at the same point in the path. That an enactment involving concurrency can be represented in two ways shows its flexibility, and distinguishes a protocol that enables concurrency from one that enables only a single sequence.

The *Pay-First* protocol, given in Listing 6.9 entertains exactly one complete enactment, where S sends *ship* after receiving *payment*.

Listing 6.9: Eliminating concurrency leads to refinement.

```
Pay-First {
  roles B, S
  parameters in ID key, in item, in price, out payment, out shipped

  B ↦ S: pay[in ID key, in item, in price, out payment]
  S ↦ B: ship[in ID key, in item, in payment, out shipped]
}
```

The only complete path for this protocol is:

```
[..., ⟨S↦B, ID | payment⟩, ⟨B↦S, ID | shipped⟩]
```

This path is identical to the second path of *Flexible-Purchase* in Listing 6.8. Further, all other paths of *Pay-First* are prefixes of this complete path, which means that they are paths of *Flexible-Purchase* as well. Therefore, we claim that *Pay-First* refines *Flexible-Purchase*.

6.2.7 Adding an Intermediary

Adding roles to a protocol yields a valid refinement if care is taken to ensure that all of the original roles observe the same information in the same sequences. Parameters observed together in the original protocol must be observed together in a refinement.

Listing 6.10 gives *Direct-Purchase*, in which the buyer and seller interact directly, while Listing 6.11 gives *Indirect-Purchase*, where the purchase is made through an intermediary.

Listing 6.10: Direct purchase protocol.

```
Direct-Purchase {
  roles B, S // Buyer, Seller
  parameters out ID key, out item, out deliver
  private payment
  S ↦ B: greet[out ID key]
  B ↦ S: order[in ID key, out item]
  S ↦ B: ship[in ID key, in item, out deliver]
}
// one complete path
[⟨S↦B, ID |⟩, ⟨B↦S, ID | item⟩, ⟨S↦B, ID | deliver⟩]
```

Listing 6.11: Indirect purchase protocol.

```
Indirect-Purchase {
  roles B, S, I // Buyer, Seller, Intermediary
  parameters out ID key, out item, out deliver
  private payment
  S ↦ B: greet[out ID key]
  B ↦ I: order[in ID key, out item]
  I ↦ S: confirm[in ID key, in item]
  S ↦ B: ship[in ID key, in item, out deliver]
}
// one complete path
[⟨S↦B, ID |⟩, ⟨B↦I, ID | item⟩, ⟨I↦S, ID |⟩, ⟨S↦B, ID | deliver⟩]
```

In *Indirect-Purchase*, B does not send *order* directly to S, but through an intermediary I instead. Communicating through an intermediary does not affect the observations of the original roles, because it merely decouples the information emission and reception in the same way as asynchrony.

6.2.8 Private Parameters

A refinement may introduce new private parameters, though care must be taken to avoid private safety conflicts that are not caught by refinement. Listing 6.12 shows how using private parameters enables splitting information across multiple messages.

Listing 6.12: Purchase with Escrow.

```
Escrowed-Purchase {
  roles B, S, I // Buyer, Seller, Intermediary
  parameters out ID key, out item, out deliver
  private payment, transfer // new private parameters

  S ↦ B: greet[out ID key]
  B ↦ S: order[in ID key, out item]
  B ↦ I: pay[in ID key, out payment]
  I ↦ S: transfer[in ID key, in payment, out transfer]
  S ↦ B: ship[in ID key, in item, in payment, out deliver]
}
// one complete path, projected to public parameters only
[⟨S↦B, ID |⟩, ⟨B↦S, ID | item⟩, ⟨B↦I, ID |⟩,
 ⟨I↦S, ID |⟩, ⟨S↦B, ID | deliver⟩]
```

Listing 6.12 shows an extended refinement of *Direct-Purchase*, in which the buyer sends the order directly to the seller, but sends additional payment information through an intermediary that acts as an escrow. Since the payment parameter is private, this change does not violate refinement. And, *transfer* does not have any ⌜out⌝ parameters, so it does not introduce any new bindings, but is necessary to communicate the binding of payment to S. The fact that *Escrowed-Purchase* is a refinement of *Direct-Purchase* can be seen from the paths, because the B and S roles observe the same information in the same sequence. The additional messages are ignored.

6.3 Formalization

We now formalize a path-based semantics for protocols. This semantics is intended to capture all of the same properties as the original semantics, but in terms of linear sequences of events to simplify statements about relationships between two enactments, such as when one extends another. Using linear sequences does not impose a global order, since all possible orderings are considered.

We use $\downarrow_x$ to project a list to those of its elements that belong to x.

The basic element of a protocol, and thus of a path, is a message.

Definition 17*:* A *message schema* $\ulcorner s \mapsto r : m\ \vec{p}(\vec{k})\urcorner$ associates sender s, recipient r, message name m, parameter list $\vec{p}$, and keys $\vec{k}$.

When message schemas are enacted, they produce message instances that contain additional details unique to the enactment, namely parameter values and a reception offset.

Definition 18*:* A *message instance* associates a schema with a list of values $\vec{v}$ and an integer reception offset o. An instance has the form $m[s, r, \vec{p}(\vec{k}, \vec{v}), o]$. Below, s_i, p_i, etc. refer to the fields of a message instance m_i. And, $\mathsf{ins}(m)$, $\mathsf{outs}(m)$, and $\mathsf{nils}(m)$ refer to parameters of m that are adorned $\ulcorner\mathsf{in}\urcorner$, $\ulcorner\mathsf{out}\urcorner$, and $\ulcorner\mathsf{nil}\urcorner$, respectively.

Definition 19*:* The *offset* of a message instance (o_i for message m_i) is the number of events that occur before the message is received. Thus, a message with offset 0 is effectively received immediately, before any other events occur, whereas a message of offset 1 is received after the next event, and so on.

A *protocol* is either a message schema or a bag of protocols (references). Like messages, protocols also have roles, parameters, and keys. Parameters that are not public are renamed uniquely, to provide encapsulation when used in composition.

Definition 20*:* A *protocol* P is a tuple $\langle n, \vec{x}, \vec{y}, \vec{p}, \vec{k}, \vec{q}, F\rangle$, where n is a name; $\vec{x}$ and $\vec{y}$ are the public and private roles, respectively; $\vec{p}$, $\vec{k}$, and $\vec{q}$ are the public, key, and private parameters, respectively; and F is a finite set of f references to other protocols, $\{F_1, \ldots, F_f\}$. $(\forall i : 1 \leq i \leq f \implies F_i = \langle n_i, \vec{x}_i, \vec{p}_i, \vec{k}_i\rangle$, where $\vec{x}_i \subseteq \vec{x} \cup \vec{y}$, $\vec{p}_i \subseteq \vec{p} \cup \vec{q})$, $\vec{k}_i = \vec{p}_i \cap \vec{k}$, and $\langle n_i, \vec{x}_i, \vec{p}_i, \vec{k}_i\rangle$ projects protocol P_i to its public components (with roles and parameters renamed for uniqueness).

Below, C_P is a *composition* (i.e., a composite protocol) that references P—that is, P is a *constituent* of C_P. $C_{P/Q}$ is a composition where the reference to P is replaced by a reference to Q.

Definition 21*:* If C_P has references $F = F_1, \ldots, F_f$, then Q *substitutes* P in $C_{P/Q}$ if and only if $C_{P/Q}$ has references $F' = F'_1, \ldots, F'_f$ such that for every $i \leq f$, $F_i = P$ and $F'_i = Q$ or $F_i = F'_i$.

A *universe of discourse* (UoD) consists of a set of roles and a set of messages they can enact. Generally a UoD is taken from a protocol specification, but multiple protocols can be composed together, or enacted in a context that includes other protocols.

We refer to Definition 4: A *UoD* is a pair $\mathcal{U} = \langle \mathcal{R}, \mathcal{M} \rangle$, where $\mathcal{R}$ is a set of roles, $\mathcal{M}$ is a set of message names; each message specifies its parameters along with its sender and receiver from $\mathcal{R}$.

The universe of discourse for the roles and messages of protocol P is denoted $\mathcal{U}_P$, and the union of UoDs $\mathcal{U}_1 \cup \mathcal{U}_2$ is $\langle \mathcal{R}_1 \cup \mathcal{R}_2, \mathcal{M}_1 \cup \mathcal{M}_2 \rangle$.

A *path* is a sequence of events, namely message instances, corresponding to an enactment. Each path induces a history vector, which can be derived by appending each message instance to the sender's history in the order the instances occur, and then to the recipient's after the corresponding offset. Conversely, multiple paths may induce a history vector, since role histories are independent.

Definition 22*:* A *path* is a list of instances, $\tau = (m_1, m_2, \ldots, m_n)$, with length $\|\tau\|$.

An *extension* of a path appends one or more messages to it.

Definition 23*:* (*extension*) If $\tau = (m_1, m_2, \ldots m_n)$, then

$\tau \circ m' = (m_1, m_2, \ldots m_n, m')$

and $\tau \circ [m_{n+1}, m_{n+2}, \ldots, m_{n+j}] = (m_1, m_2, \ldots m_{n+j})$

A message instance m_i is *received* on a path if its offset is less than or equal to the number of events after m_i, else it is in transit. That is, the position of m_i describes when it was sent, and the offset describes how many events occur before it is received.

Definition 24*:* $\mathsf{received}(\tau) = \{m_i \in \mathsf{P} \mid i + o_i \leq \|\tau\|\}$

Definition 25 captures the information that a role observes after a sequence of messages has been sent. Each parameter in the message is known by role R if the parameter values match the bindings in the enactment, and either R is the sender, or R is the recipient and the message has been received.

Definition 25*:* $\mathsf{known}(\tau, \vec{k}, \vec{v}, R) = \{p \mid \exists m_i \in \tau \colon p \in \vec{p}_i \Rightarrow \vec{v}_i \downarrow_{\vec{k}} = \vec{v} \downarrow_{\vec{k}}$ and ($s_i = R$ or $r_i = R$ and $m_i \in \mathsf{received}(\tau)$)$\}$

A message instance is *viable* on a path if the sender knows the $\ulcorner\mathsf{in}\urcorner$ parameters, but not the $\ulcorner\mathsf{out}\urcorner$ or $\ulcorner\mathsf{nil}\urcorner$ parameters.

Definition 26*:* Message instance $m[s, r, \vec{p}(\vec{k}), \vec{v}]$ is *viable* on path τ if and only if $p \in \mathsf{ins}(m)$ implies $p \in \mathsf{known}(\tau, \vec{k}, \vec{v}, s)$, and $p \in \mathsf{known}(\tau, \vec{k}, \vec{v}, s)$ implies $p \notin \mathsf{outs}(m)$ and $p \notin \mathsf{nils}(m)$

A path is *valid* if and only if it is either empty or the extension of a valid path by a viable message. A path is *extensible* in a universe of discourse if there are viable messages that can

be appended to it. The set of all valid paths in $\mathcal{U}$ is denoted $\mathsf{paths}(\mathcal{U})$. The set of paths produced by a protocol must cover all possible orderings of independent events. That is, if a and b are independent events, then both paths $(\ldots, a, \ldots, b, \ldots)$ and $(\ldots, b, \ldots, a, \ldots)$ are possible.

Definition 27*:* $\tau \in \mathsf{paths}(\mathcal{U})$ if and only if $\tau = \emptyset$, or there is path $\tau' \in \mathsf{paths}(\mathcal{U})$, and message $m \in \mathsf{viable}(\tau')$ such that $\tau = \tau' \circ m$.

The *branches* at path τ in universe of discourse $\mathcal{U}$ is the set of all paths in $\mathcal{U}$ with prefix τ.

Definition 28*:* $\mathit{branches}(\mathcal{U}, \tau) = \{\tau' \mid \tau \sqsubseteq \tau' \text{ and } \tau' \in \mathsf{paths}(\mathcal{U})\}$, where $\tau \sqsubseteq \tau'$ denotes that τ is a prefix of τ'.

The *sources* of a parameter in a path are the roles that send messages binding that parameter in the path. Safe protocols have at most one source for each parameter in any path.

Definition 29*:* $\mathit{sources}(\tau, p) = \{s_i \mid p \in \mathsf{outs}(m_i \in \tau)\}$

The *keys* of a path are the combinations of key parameters that appear in message instances on the path.

Definition 30*:* $\mathit{keys}(\tau) = \{\vec{k}_i \mid m_i \in \tau\}$, where k_i is the list of key parameters of message m_i.

Subsumption captures the idea that one path induces the same knowledge as another, such that the same combinations of relevant parameters must be bound in the same order and by the same roles.

Definition 31*:* Path τ' *subsumes* path τ for parameter list $\vec{p}$, denoted $\tau' \triangleright_{\vec{p}} \tau$, if and only if

1. $\forall p \in \vec{p} : \mathsf{sources}(\tau', \mathsf{p}) = \mathsf{sources}(\tau, \mathsf{p})$
2. $\forall R \in \mathcal{R}, \forall \vec{k} \in \mathsf{keys}(\tau), \forall \vec{v} :$
   $\mathsf{known}(\tau', \vec{k}, \vec{v}, R) \cap \vec{p} = \mathsf{known}(\tau, \vec{k}, \vec{v}, R) \cap \vec{p}$
3. $\tau_2 \sqsubseteq \tau \Rightarrow \exists \tau_2'$ such that $\tau_2' \sqsubseteq \tau'$ and $\tau_2' \triangleright_{\vec{p}} \tau_2$

where $\mathcal{R}$ is the set of roles that appear in τ.

Definition 32 captures refinement in terms of paths. Informally, Q refines P if and only if every path in Q subsumes some path in P, and every path in Q that subsumes some extensible path in P is extensible or has unreceived messages. That is, if P has branches, then Q must also have branches or unreceived messages.

Definition 32*:* Protocol Q *refines* protocol P for parameters $\vec{p}$, denoted $Q \preceq_{\vec{p}} P$, if and only if for every $\mathcal{U}$ and path $\tau_Q \in \mathsf{paths}(\mathcal{U} \cup \mathcal{U}_Q)$, there is a path $\tau_P \in \mathsf{paths}(\mathcal{U} \cup \mathcal{U}_P)$ such

that (1) $\tau_Q \triangleright_{\vec{p}} \tau_P$, and (2) $\mathsf{branches}(\mathcal{U} \cup \mathcal{U}_P, \tau_P) \neq \emptyset$ implies ($\mathsf{branches}(\mathcal{U} \cup \mathcal{U}_Q, \tau_Q) \neq \emptyset$ or $\|\mathsf{received}(\tau_Q)\| < \|\tau_Q\|$).

$Q \preceq P$ denotes the common case of refinement with respect to the public parameters of P.

Safety is the correctness property that no parameter takes on more than one value for a given set of keys in any path.

Definition 33*:* Protocol P is *safe* if every path in $\mathcal{U}_P$ is safe. A path τ is safe if and only if, for every $m_i \in \tau$, there is no $m_{j \neq i}$ such that $\vec{k}_i = \vec{k}_j \wedge \vec{v}_i \downarrow_{\vec{k}_i} \neq \vec{v}_j \downarrow_{\vec{k}_j}$

Liveness is a correctness property requiring that every enactment of a protocol be extensible by a finite sequence of emissions to an enactment in which all public $\ulcorner\mathsf{out}\urcorner$ parameters are bound.

Definition 34*:* A protocol P with public $\ulcorner\mathsf{out}\urcorner$ parameters $\vec{p}$ is *live* in $\mathcal{U}_P$ if and only if every path in $\mathcal{U}_P$ is a prefix of some path in $\mathcal{U}_P$ such that $\vec{p} \subseteq \bigcup_{\vec{k},\vec{v},R} \mathsf{known}(\tau, \vec{k}, \vec{v}, R)$

6.4 Theoretical Results

We say that the protocols are safe or live *up to* $\vec{p}$, meaning that there are no violations involving those parameters. In simpler terms, introducing unused public parameters can artificially violate liveness and introducing conflicts on private parameters can artificially violate safety. Such conditions are readily checked.

Theorem 1 establishes that refinement preserves liveness. Hence, if a protocol is live, it is sufficient to check that a substitute protocol is a refinement—and we verify its liveness for free.

Theorem 1. *If P is live and $Q \preceq_{\vec{p}} P$, then Q is live up to $\vec{p}$.*

Proof Sketch. Since P is live, for every path τ in $\mathcal{U}_P$, one of the following holds: (1) $\mathsf{branches}(\mathcal{U}_P, \tau) \neq \emptyset$; (2) $\|\mathsf{received}(\tau)\| < \|\tau\|$; or (3) $\vec{p} \subseteq \bigcup_{\vec{k},\vec{v},R} \mathsf{known}(\tau, \vec{k}, \vec{v}, R)$. Since $Q \preceq P$, for every path $\tau_Q \in \mathcal{U}_Q$ there is a path $\tau_P \in \mathsf{paths}(\mathcal{U}_P) : \tau_Q \triangleright_{\vec{p}} \tau_P$, and $\mathsf{branches}(\mathcal{U}_P, \tau_P) \neq \emptyset \Rightarrow$ $\mathsf{branches}(\mathcal{U}_Q, \tau_Q) \neq \emptyset$ or $\|\mathsf{received}(\tau_Q)\| < \|\tau_Q\|$. From subsumption, $\bigcup_{\vec{k},\vec{v},R} \mathsf{known}(\tau_Q, \vec{k}, \vec{v}, R) = \bigcup_{\vec{k},\vec{v},R} \mathsf{known}(\tau_P, \vec{k}, \vec{v}, R)$. Thus we have one of: (1) $\mathsf{branches}(\mathcal{U}_Q, \tau) \neq \emptyset$; (2) $\|\mathsf{received}(\tau_Q)\| < \|\tau_Q\|$; or (3) $\vec{p} \subseteq \bigcup_{\vec{k},\vec{v},R} \mathsf{known}(\tau_Q, \vec{k}, \vec{v}, R)$. So Q is live up to $\vec{p}$. □

Theorem 2 establishes that refinement preserves safety.

Theorem 2. *If P is safe and $Q \preceq_{\vec{p}} P$ then Q is safe up to $\vec{p}$.*

Proof Sketch. Suppose P is safe, $Q \preceq_{\vec{p}} P$ and $\vec{p}_i \subseteq \vec{p}$ for all messages in Q. Because $Q \preceq_{\vec{p}} P$, the sources of any p in $\vec{p}$ on any path in Q are the same as the sources of p on some path in

P. Thus, for every parameter in $\vec{p}$, there is at most one source for each parameter of Q in any path in Q, and Q is safe up to $\vec{p}$. □

Theorem 3 establishes that refinement applies to substitution of a constituent in a composition. That is, if C_P is a composition in which P is a constituent, and Q is a refinement of P, then a composition $C_{P/Q}$ with Q substituted for P is a refinement of C_P.

Theorem 3. *If $Q \preceq P$, then $C_{P/Q} \preceq C_P$.*

Proof Sketch. Suppose $Q \preceq P$, with P having public parameters $\vec{p}$. Let $\mathcal{U}_C$ be the universe of discourse of C_P without the messages in P, such that $\mathcal{U}_{C_P} = \mathcal{U}_C \cup \mathcal{U}_P$, and $\mathcal{U}_{C_{P/Q}} = \mathcal{U}_C \cup \mathcal{U}_Q$.

(0) Suppose it is not true that $C_{P/Q} \preceq C_P$. By the definition of refinement, (1) there must be some path τ_Q in $\mathit{paths}(\mathcal{U}_{C_{P/Q}})$ that does not subsume any path τ_P in $\mathcal{U}_{C_P}$, or (2) τ_P has branches but τ_Q does not and τ_Q does not have any unreceived messages.

(1) Suppose there is some path $\tau_Q \in \mathsf{paths}(\mathcal{U}_{C_{P/Q}})$ that subsumes a path $\tau_P \in \mathsf{paths}(\mathcal{U}_{C_P})$, but for some message m in $\mathcal{U}_{C_{P/Q}}$, $\tau_Q \circ m$ does not subsume any path in $\mathsf{paths}(\mathcal{U}_{C_P})$. Since τ_Q subsumes τ_P, they must induce the same knowledge at each role, and thus would enable the same messages. Thus for $\tau_Q \circ m$ to not subsume any path, m must not exist in $\mathcal{U}_{C_P}$, which means it must be in Q. But by the definition of refinement, for any $\mathcal{U}$, $\forall \tau_Q \in \mathsf{paths}(\mathcal{U} \cup \mathcal{U}_Q), \exists \tau_P \in \mathsf{paths}(\mathcal{U} \cup \mathcal{U}_P) : \tau_Q \triangleright_{\vec{p}} \tau_P$, contradicting (1).

(2) Suppose a path $\tau_Q \in \mathsf{paths}(\mathcal{U}_{C_{P/Q}})$ does not have branches or any unreceived messages, but subsumes a path $\tau_P \in \mathsf{paths}(\mathcal{U}_{C_P})$ that has branches. Since τ_Q subsumes τ_P, they must induce the same knowledge at each role, and thus would enable the same messages; thus any messages enabled on τ_P but not τ_Q must be in P, since all the messages in $\mathcal{U}_C$ are in both $\mathsf{paths}(\mathcal{U}_{C_{P/Q}})$ and $\mathsf{paths}(\mathcal{U}_{C_P})$. But $Q \preceq P$ and by the definition of refinement $\forall \mathcal{U} \mathsf{branches}(\mathcal{U} \cup \mathcal{U}_P, \tau_P) \neq \emptyset \Rightarrow (\mathsf{branches}(\mathcal{U} \cup \mathcal{U}_Q, \tau_Q) \neq \emptyset \vee \|\mathsf{received}(\tau_Q)\| < \|\tau_Q\|)$. Thus τ_Q has branches in contradiction of supposition (2).

Thus, (1) and (2) are false, contradicting (0), so $C_{P/Q} \preceq C_P$. □

Theorems 4 and 5 establish that the safety and liveness of a composition are preserved when substituting a constituent by a refinement. This result yields assurance of the liveness and safety of a composition without full reverification; only the replacement protocol needs to be checked for safety and refinement.

Theorem 4. *If C_P is live and $Q \preceq P$, then $C_{P/Q}$ is live.*

Proof Sketch. Suppose C_P is live and $Q \preceq P$. Assume that Q does not pathologically have public parameters not in P that block messages in C. By Theorem 3, $C_{P/Q} \preceq C_P$. Since C_P is live, every path in $\mathit{paths}(\mathcal{U}_{C_P})$ either has branches or binds every $\ulcorner\mathsf{out}\urcorner$ parameter of C_P.

By refinement, every path in $\mathcal{U}_{C_{P/Q}}$ subsumes some path in $\mathcal{U}_{C_P}$. Thus, every path in $\mathcal{U}_{C_{P/Q}}$ either has branches, or binds every ⌜out⌝ parameter of C_P. Since C_P and $C_{P/Q}$ have the same public parameters, this means $C_{P/Q}$ is live. □

Theorem 5. *If C_P is safe and $Q \preceq_{\vec{p}} P$, then $C_{P/Q}$ is safe up to $\vec{p}$.*

Proof Sketch. Let C_P be safe, and $Q \preceq_{\vec{p}} P$. By Theorem 3, $C_{P/Q} \preceq C_P$ and Q enables only what P did. C_P is safe, so any safety violations are internal to Q. By Theorem 2 Q is safe up to $\vec{p}$, so $C_{P/Q}$ is safe up to $\vec{p}$. □

6.5 Evaluation

Algorithm 3 verifies refinement based on Definition 32. It enumerates all possible paths in both P and Q, and checks that each path q in Q subsumes some path p in P such that if p has branches then q does as well, otherwise it is not a refinement.

Algorithm 3: Refinement algorithm

```
1  foreach τ_Q in paths(U_Q) do
2      match ← ∅ ;
3      for τ_P in paths(U_P) do
4          match ← τ_P ;
5          if There are no branches of τ_P, or τ_Q has branches or unreceived messages
            then
6              break;                                  // found match; stop looking
7      if no match then
8          return False;                                  // failed Definition 32.1
9      if match has branches but τ_Q does not have branches or unreceived messages then
10         return False;                                  // failed Definition 32.2
11     return True;
```

We have implemented the refinement checking algorithm in Python 3.6.5 enhancing our toolchain for BSPL. Testing was done via pytest, executing the relevant functions in a for loop, skipping the first iteration so that any loading or caching occurs before our measurements. All tests were performed on a laptop running Gentoo Linux, kernel version 4.19.27, with an Intel i7-6600U cpu, 16GB of DDR3 memory, and 1TB SSD. All timings are in milliseconds;

and the minimum, mean, maximum, and sample standard deviation (σ) are computed over 10 runs.

Table 6.1 shows the time required to verify three of the example refinements, indicating that the approach is tractable.

Table 6.1: Time to verify refinement for example protocols.

Protocol	Min	Mean	Max	σ
Buyer-Starts $\preceq$ Either-Starts	1	1	2	0.6
Single-Lookup $\preceq$ Lookup-Prices	103	138	169	21.3
Pay-First $\preceq$ Flexible-Purchase	53	70	87	9.6

For comparison, Table 6.2 lists the time it takes to verify liveness and safety for the *Commerce* protocol resulting from the substitution of a single protocol by its refinement (e.g., *Either-Starts* with *Buyer-Starts*). The last row gives the time for verifying liveness and safety after all three constituents are substituted; this protocol is the simplest since all three constituents have been simplified, and so takes the least time to verify. As these measurements demonstrate, the time to check refinement is less than the time to check liveness for the *Commerce* protocol post substitution, thus establishing the practical benefit of checking refinement.

The time to check safety post substitution is also higher than the time to check refinement except when *Buyer-Starts* is substituted for *Either-Starts*, as in the first and last rows, where the safety checking tool statically detects that there are no messages with the same `⌜out⌝` parameter but different senders, ruling out the violation of safety in time linear in the size of the protocol specification.

Table 6.2: Times to verify properties of *Commerce* (Listing 6.2) with select constituent replaced by its refinement, as given in Table 6.1.

Substitution	Property	Min	Mean	Max	σ
sub. *Buyer-Starts*	Liveness	14 419	14 745	15 099	222
	Safety	0	0	0	0
sub. *Single-Lookup*	Liveness	18 483	18 568	18 714	68
	Safety	18 296	18 401	18 541	74
sub. *Pay-First*	Liveness	18 408	18 512	18 601	78
	Safety	18 246	18 347	18 517	78
substitute all	Liveness	11 624	11 868	12 279	197
	Safety	0	0	0	0.03

We also tested preexisting protocol specifications, checking that the NetBill protocol [56] is a refinement of the Bliss variant [151], and that a variant of HL7's CreateLaboratoryOrder flow [89] is a refinement of the original. Table 6.3 shows that checking refinement is faster than re-verifying liveness in both cases.

Table 6.3: Time to verify liveness and refinement for the *NetBill* and *CreateLabOrder* protocols.

Protocol	Property	Min	Mean	Max	σ
NetBill	Refinement	1 256	1 294	1 323	21
	Liveness	13 615	13 926	14 141	174
CreateLabOrder	Refinement	5 879	6 003	6 162	83
	Liveness	45 257	46 469	50 878	1 648

6.6 Related Work

Refinement, a preorder relation on protocols, is somewhat related to *simulation* [127], which is a preorder relation on processes. A process P simulates a process Q if P can match all of Q's moves; that is, if a transition is available at a state in Q, the same transition is available at a matching state in P. A simulating process can do everything the simulated process can do, and therefore substitute for (impersonate) it. Refinement is concerned with substitution in compositions where additional states may introduce conflicts, such as two roles binding the same parameter, and so is more like specialization than extension. Each path in a refinement must have the same observations in the same order as the paths it subsumes, but may have fewer branches.

Process testing [62] defines notions of refinement for communicating processes, in which one process (e.g., a server) refines another if it passes the same tests (e..g., supports the same clients). Bernardi and Hennessy [24] extend testing to mutual testing such as peer-to-peer relationships. However, testing applies to processes, not interactions. Even more recent work on multiparty testing De Nicola and Melgratti [63] addresses only multiple clients interacting with the same server, relaxing the preorder to accommodate flexible ordering in the server's responses to uncoordinated clients.

The Liskov substitution principle [105] states that subtypes of a type satisfy properties of the type. Refinement is a kind of subtyping and preserves safety and liveness. However, Liskov's behavioral substitution applies not to protocols, but to roles, which describe behavior.

Gay [76] discusses channel-oriented and process-oriented subtyping for session types [90]. Session types work that tackles multiparty asynchronous protocols [91] requires FIFO channels.

We distinguish conformance from *compliance*, the problem of determining at runtime whether an agent satisfies a role [78, 144, 152]. In this terminology, Ancona et al. [7] deals with compliance rather than conformance.

Mazouzi et al. [109] describe a design methodology for communication protocols involving translation of abstract protocol specification diagrams in AUML to recursive colored Petri nets (RCPNs) for formal verified at various levels of abstraction. However, although RCPNs support concurrency, they are synchronous across the various roles, and do not account for the possible risks to information integrity introduced by asynchrony.

Refinement relates to prior work on atomicity of protocols [47], which requires that if any constituent is partially enacted it must be possible to complete, in that both address correctness of protocols used together in a composition. Although atomicity entails liveness for the overall composition, it is primarily concerned with detecting conflicts that would prevent a constituent from completing and leave it "dangling." Atomicity can be verified only by using the composition as a whole; it cannot be determined from the specification of a constituent protocol in isolation. By contrast, refinement verifies that particular substitutes for a constituent protocol do not affect the liveness of the composition, without needing to consider the other constituents. Thus the two concepts complement each other, with atomicity proving that a constituent will be live in a composition, and refinement verifying that a substitute protocol will behave the same way.

Information protocols enable layering meaning protocols, e.g., based on commitment and norms [14, 99, 117, 154, 169, 172]. Refinement has been studied for commitment protocols [39, 77, 108]. Our intuitions regarding refinement—that Q refines P if every run of Q embeds some run of P—agree with Gerard and Singh's [77]. We additionally require that every enactment in Q be able to progress if the corresponding enactment in P can progress. The present framework accommodates asynchronous communications, which the above works do not address.

In ongoing work [44, p. 15], we show that *Flexible-Payment* (Listing 6.8), which supports payment and shipment in any order or concurrently, cannot be expressed in protocol languages based on trace expressions or session types. These languages can express an analog of *Pay-First* (Listing 6.9)—a refinement of *Flexible-Payment*.

6.7 Conclusions and Future Work

We have motivated and formalized a notion of refinement of information protocols. We established important results about this notion of refinement. Specifically, we showed that (1) a refinement of a safe and live protocol is, respectively, safe and live; (2) safety and liveness of a composition are preserved when substituting a constituent protocol with its refinement; and (3) a composition with a refinement substituted for a constituent protocol is itself a refinement of the original composition.

We described an algorithm for checking refinement and provide an empirical demonstration of its performance that establishes that checking refinement once is worthwhile compared to checking the liveness and safety of compositions post substitution. Future work will consider optimizations—such as taking advantage of path symmetry, memoization, and lazy generation—that should make the refinement-checking algorithm even more efficient.

Note that although refinement preserves the safety and liveness of a protocol, sometimes a refined protocol may be safe or live when the protocol it refines is not. Indeed, that would be a common occurrence during engineering. Engineers may specify a protocol that meets their application requirements and upon determining (through a checker tool) that the protocol is not safe or live, may proceed to refine it to obtain one that is safe and live—and therefore appropriate for developing agents to play roles in it.

We could support richer kinds of refinement if we exploited semantic mappings between the parameters of protocols. For example, suppose a purchase protocol supports separate delivery addresses for each item. It should be reasonable to refine this protocol to one in which the same address is used for all of the items. Such a protocol should be a refinement, since it reduces the number of choices and does not produce any less information: the customer could provide the same address multiple times in the unrefined protocol. However, such a refinement goes beyond this paper, as this paper does not support mappings between parameters.

Our notion of protocol refinement shares with works on conformance the intuition that refinement means a reduction of emission choices relative to the original protocol. A future direction is to investigate the relationship between refinement and conformance. We would expect that if protocol Q refines protocol P and if a role ρ features in both, Q projected to ρ would conform with P projected to ρ.

An interesting line of work would be to consider the refinement of commitment specifications along with the refinement of information protocols. For example, introducing an intermediary for purposes of escrow (Listing 6.12) is a refinement of the direct purchase protocol (Listing 6.3) in the current work; however, this does not consider that the relevant

commitment specifications may have to be refined as well, e.g., to set up commitments between escrow and the other agents. It would be necessary to ensure that protocols support enactments that enable agents to comply with their commitments, along the lines of the Clouseau approach [154].

Chapter 7

Pippi – Practical Protocol Instantiation

A protocol specifies interactions between roles, which together constitute a multiagent system (MAS). Enacting a protocol presupposes that agents are bound to the its roles. Existing protocol-based approaches, however, do not adequately treat the practical aspects of how roles bindings come about.

Pippi addresses this problem of MAS instantiation. It proposes the notion of a metaprotocol, enacting which instantiates a MAS suitable for enacting a given protocol. Pippi demonstrates the subtleties involved in instantiating MAS arising from protocol composition, correlation, and decentralization. To address these subtleties and further support practical application patterns, we introduce an enhanced protocol language, with support for parameter types (including role and protocol typed parameters, for metaprotocols), interface flexibility, and binding constraints. We discuss the realization of our approach through an extended agent architecture, including the novel concept of a MAS adapter for contact management. We evaluate Pippi's expressiveness by demonstrating common patterns for agent discovery.

This chapter is adapted from the paper *Pippi: Practical Protocol Instantiation*, published in the AAMAS conference proceedings of May 2022 [52]

7.1 Introduction

Multiagent systems (MASs) can be specified from multiple perspectives, including organizations, commitments, and interactions. Yet, from a practical standpoint, how do agents come together to instantiate a MAS?

We focus on multiagent systems specified through interaction protocols [66], which are characterized by the messages that the agents send to and receive from one another and the constraints on the ordering and occurrence of those messages. For such a specification, the agents take on *roles* in a protocol, which specifies the messages each role may send and receive. Most protocol specification languages focus on the interaction itself and the order in which messages may be sent and received, with little consideration of how the roles are bound to agents. From a theoretical standpoint, this is a reasonable assumption; role binding can be left as an implementation detail. But for practical applications of interaction protocols, it is important to consider how each agent learns of its role and how each agent learns the bindings of the other roles it must interact with.

Composition is a longstanding and challenging theme for practical applications [93]. Protocol composition presents crucial challenges for both modeling and verification [47] because protocols focus on interaction and interaction in a decentralized setting is nontrivial to deal with. Traditional protocol languages rigidly specify the sequences of events that agents can observe, making composition difficult. Information-based protocol languages such as Splee [43] and BSPL [149] support composition but have their own limitations. BSPL protocols have fixed conditions for completion, meaning that protocols only have two states, complete and incomplete, and must therefore hide any nuances. BSPL and Splee protocol key constraints require a composition to have a common key, imposing on their constituent protocols that would otherwise have their own keys. Finally, current implementations propagate meaning from the outermost composition into each constituent protocol by substituting the parameter names; from a conceptual point of view, this is correct (the meaning of each constituent does change according to its use in the composition), but for modularity each agent should only need to know the protocols it supports, not the compositions they are used in.

We adopt the name Pippi (after the famous character in children's literature) for our approach because of its similarity with PPI (for Practical Protocol Instantiation).

7.1.1 Scenario: Wedding

To illustrate some of these concepts, we adopt a marriage scenario. The scenario starts with one agent, the proposer, who would like to get married to another. The proposer is initialized as part of a broader MAS context and can select potential partners from that context. Once

the proposer has selected a partner, they then propose marriage; that is, they propose that the two agents enact a ceremony (protocol) together that will produce a social result of marriage. When the proposee has agreed to go through the ceremony (possibly selecting which form of ceremony they will enact), they can begin; note that this agreement to enact the ceremony is *not* the same thing as committing to the marriage.

As they enact the marriage ceremony, they will need to bring in other agents to participate; for example, in a court wedding, they need a judge to officiate and a witness to sign the license. It should be possible to ask someone to witness at the last second, and the witness shouldn't need to know any details about the rest of the ceremony.

7.1.2 Objectives and Novelty

The scenario above presents challenges, which we address:

Dynamic Role Binding

How do agents come together at runtime to enact a protocol?

In the wedding scenario, the proposer starts by inviting their prospective spouse, a judge, and a witness. The proposer must communicate not only each invitee's role in the protocol but also the roles of the other agents they must interact with. The proposer may prefer to use *late binding* to invite those agents only when their role becomes necessary, to protect privacy for all parties, and avoid overhead in case the enactment is terminated early.

Composition

How can we ensure that exactly the right amount of information is shared between the constituent protocols in a composition?

The wedding scenario can be decomposed into subprotocols, e.g., a specific wedding ceremony and a generic witness protocol. How do we ensure *independence*, that is, that each agent need only be aware of the protocol it is enacting? A witness participating in a wedding ceremony need only be aware of the generic witness protocol and should not need to implement support for (or even be aware of) the broader ceremony.

7.1.3 Contribution: Our Solution Conceptually

To negotiate which protocol to enact and which agents to play the roles, agents need the ability to discuss these objects. We introduce the concept of *metaprotocols*, which are protocols whose parameters refer to the elements of another protocol.

Table 7.1: Pippi Syntax

L_4	Spec	$\longrightarrow$	$\lfloor$*Name*:*Value* $\rfloor^*$ $\lfloor$*Protocol*$\mid$*Message*$\rfloor^+$
L_5	Protocol	$\longrightarrow$	*Name*(*ParamExpr*)
			{ $\lfloor$private *Parameter*$^+\rfloor$ *Reference*$^+$ }
L_6	ParamExpr	$\longrightarrow$	*ParamClause* $\lfloor$, *ParamClause*$\rfloor^*$ | (*ParamExpr*)
L_7	ParamClause	$\longrightarrow$	*Parameter* | *ParamExpr* $\lfloor$or *ParamExpr*$\rfloor^*$
L_8	Parameter	$\longrightarrow$	$\lfloor\lfloor$*Qualifier*$\rfloor$ *Name*$^+$ $\lfloor$: *Constraint*$^+\rfloor\rfloor^+$
L_9	Reference	$\longrightarrow$	*Name*(*ParamExpr*) | *Message*
L_{10}	Message	$\longrightarrow$	$\lfloor$*Ad*$\rfloor$ *Name* $\mapsto$ $\lfloor$*Ad*$\rfloor$ *Name*$^+$: *Name*[*ParamExpr*]
L_{11}	Ad	$\longrightarrow$	in | nil | out| any | opt
L_{12}	Qualifier	$\longrightarrow$	$\lfloor$key$\rfloor\lfloor$local$\rfloor\lfloor$set$\rfloor$
L_{13}	Name	$\longrightarrow$	$\lfloor$*Prefix*:$\rfloor$(*S*)*tring*
L_{14}	Constraints	$\longrightarrow$	$\lfloor$*Type*$\rfloor\lfloor\subseteq\mid\in\mid=\mid<\mid>$ *Name*$\mid$*Value*$\rfloor$
L_{15}	Type	$\longrightarrow$	role | protocol | *Name*

Although metaprotocols can be implemented using normal protocols with special interpretation for some of the parameters, we introduce protocol language extensions to make those meanings explicit. We borrow role parameters from Splee [43], and add support for parameter type declarations. Parameters with the type `protocol` may be used as variables for more flexible composition.

7.2 The Protocol Language

BSPL [147], the foundational information protocol language, did not address role binding, assuming that roles were bound prior to protocol instantiation. Splee [43] supported treating roles as parameters for dynamic binding but did not address practical instantiation challenges.

We now introduce the core aspects of our protocol language, illustrated through specifications of the scenario.

7.2.1 Syntax

The formal syntax of our language, derived from Splee and BSPL, is given in Table 7.1, and described in detail below. A superscript of + indicates one or more repetitions, superscript $*$ indicates zero or more, and $\lfloor$ and $\rfloor$ delimit expressions, which are optional when without a superscript.

L_4. A specification document consists of a preamble and one or more protocols (which may be individual messages). The preamble contains key/value pairs used to declare common information (such as author or default prefix) or terms for substitution in the document

(such as namespaces and IRI abbreviations).

L_5. A protocol declaration consists of a name, a public parameter expression, optional private parameters, and references to constituent protocols or messages. The public parameters with the key qualifier form this declaration's key. The private parameters can be surrounded by parentheses to spread them across multiple lines.

L_6. A parameter expression is a conjunction of parameter clauses, written as a (possibly parenthesized) comma-separated list.

L_7. A parameter clause is a disjunction of parameter expressions or an adorned parameter. If an adornment is not provided, it should be inferred in a reference or default to ⌜in⌝.

L_8. A parameter has a name and optional qualifiers and constraints. Multiple names can be included, separated by commas, as shorthand for repeating the same adornments, qualifiers, and constraints, provided all the names are on the same line or parenthesized.

L_9. A reference to a protocol may consist of a name appended by a parameter expression matching the protocol's declaration. The name must be the name of a protocol or the name of a parameter whose type is protocol.

L_{10}. A message schema consists of a name, sending role and receiving role(s), and a parameter expression.

L_{11}. An adornment is usually either ⌜in⌝ or ⌜out⌝; ⌜nil⌝ indicates an unknown parameter, ⌜any⌝ means either ⌜in⌝ or ⌜out⌝, and ⌜opt⌝ means the parameter is optional.

L_{12}. Qualifiers indicate if a parameter is any combination of key, local, and set. The order of the qualifiers is irrelevant.

L_{13}. Names are strings with an optional namespace prefix separated by a colon and an optional abbreviation in parentheses (the location of the abbreviation in the string does not matter). Disallowed characters include: [(){}[]:;,] (whitespace, brackets, and separators)

L_{14}. Constraints describe the possible values that a parameter binding may have, expressed as a semicolon-separated list. Example constraints include type, enumerated values, relationships with other parameters, and so on.

L_{15}. The type of a parameter may be role, protocol, or some name that can be resolved using the context and prefixes declared in the specification.

7.2.2 Illustration

Listing 7.1: Generic Witness Protocol

```
Witness(in J, W: role, in key cID, out sig or out objection) {
  private req
  J -> W: RequestApproval[in cID, out req]
  W -> J: Approve[in cID, in req, out sig]
  W -> J: Object[in cID, in req, out objection]
}
```

The *Witness* protocol, given in Listing 7.1, shows a simple protocol involving five parameters: J (Judge) and W (Witness) are roles, cID is the key that uniquely identifies each enactment (which is adorned ⌜in⌝ and therefore bound externally), and sig (signature) and objection are two parameters that may be bound during an enactment. The public parameters declare the completion requirements for an enactment: when all public parameters are bound, the enactment is complete. In *Witness*, we demonstrate syntax for complex Boolean completion formulas; only one of sig or objection needs to be bound to complete the enactment.

The body of the protocol lists the subprotocols it is composed of; a message is an elementary protocol. The *RequestApproval* message on line 3 is sent by J (the role before the arrow) to W. Its payload contains two parameters, cID and req; these parameters represent the information conveyed by the message. The parameter cID is adorned ⌜in⌝, and so must be bound before *RequestApproval* can be sent. Conversely, req is ⌜out⌝, and so must *not* already be bound; it is bound when sending *RequestApproval*. Parameter bindings are functionally dependent on their keys; req may have only one binding for each value of cID.

Messages may be sent when they are enabled by the satisfaction of their causality constraints as specified by their parameter adornments. Only emissions are constrained; a message may be received any time after it has been sent.

Listing 7.2: Court Wedding

```
CourtWedding(in Propose(R), Propose(E), (J)udge, (W)itness: role,
  any key cID,
  (out vowR, out vowE, out license) or out objection) {
  private signature
  J -> R, E: Ask[any cID, out questions]
  E -> J, R: EVow[in cID, in questions, out vowE]
  R -> J, E: RVow[in cID, in questions, out vowR]
  Witness(J, W, cID, signature or objection)
  J -> E, R: Marry[in cID, in vowE, in vowR, in signature, out
     license]
}
```

Listing 7.2 gives a protocol describing a court wedding. Note that all of the role parameters are adorned ⌜in⌝; they must be bound outside the context of the protocol. The key cID is adorned ⌜any⌝, which means it can be either ⌜in⌝ or ⌜out⌝. In addition to the features used in *Witness*, *CourtWedding* references *Witness*, and handles the two outcomes separately. If the witness provides a signature, then the judge can perform the marriage; otherwise, the protocol terminates with an objection. *CourtWedding* exhibits the role abbreviation syntax, e.g., identifying R as the abbreviation for Proposer.

7.3 Dynamic Role Binding

All that is required to enact a protocol is a group of agents willing and able to play all its roles. In most existing work on protocols, the role bindings are assumed; perhaps the MAS was hard-coded or statically configured with the agents' roles and endpoints.

For greater flexibility and to properly model real-life systems, agents need to discover new peers and negotiate protocols with them. Such dynamic role binding is useful even for simple systems, such as where one seller enacts the same purchase protocol with multiple customers. Explicit semantics for how the seller is introduced to each customer and how those role bindings relate to the enactment information is necessary for correct implementation.

Furthermore, we would like to support not only explicit role binding but also *late* role binding, where some agents begin to play their roles after the protocol has already been initiated. For example, in a rideshare setting, riders might want to request a different driver if the first is not coming without having to resubmit their destination. In addition, it is reasonable to delay the involvement of additional parties for efficiency and privacy reasons. If a customer's bank is notified every time the customer considers a purchase regardless of

their final choice, it unnecessarily exposes their actions and wastes the bank's resources.

To address the above concerns, we introduce two patterns of protocol design: metaprotocols and self-contained protocols.

7.3.1 Metaprotocols

Metaprotocols are protocols that specify communication about the elements of another protocol, such as its structure and role bindings. Roles in a protocol may be represented by parameters in a metaprotocol without any special semantics.

Late role binding is enabled in a metaprotocol by interleaving its enactment with that of its target protocol. Then, messages binding specific roles need not be sent until they are relevant in the target protocol, though, of course, they may be sent earlier.

A metaprotocol capturing the proposal phase of the marriage scenario might be implemented as follows:

Listing 7.3: Proposal Metaprotocol

```
Proposal(out r0, r1: role, out key mID, out acceptance or out
   rejection) {
  private (r2, r3: role,
    cID, R, E, J, W,
    ceremony: protocol)
  out r0 -> out r1: Propose[out mID, out ceremony, out R, out E]
  r1 -> r0: Accept[in mID, in ceremony, out acceptance]
  r0 -> r1: Plan[in mID, in ceremony, in accept, out cID]
  ceremony(R, E, J, W, cID, out vowR, out vowE, out result)
  r0 -> out r2: Schedule[in mID, in cID, in E, out J, out date]
  r0 -> out r3: Invite[in mID, in cID, in J, out W, out invitation]
  r1 -> r0: Reject[in mID, in ceremony, out rejection]
}
```

The *Proposal* metaprotocol, given in Listing 7.3, sets up the roles necessary for enacting a wedding ceremony such as *CourtWedding*. First, the roles in this protocol are numbered because they have no meaning in themselves; they are placeholders for the purpose of inviting agents to perform the ceremony. Thus, r0 sends *Propose* to r1, suggesting that they take on the roles R and E in some ceremony—not necessarily in that order. The ceremony is specified by binding it to the ceremony parameter. If r1 accepts, r0 may proceed to invite other agents. Note that *all* of the roles from *CourtWedding* are bound in this protocol, yet late binding is possible because r0 may delay inviting r3 to perform the W role until it becomes relevant.

Metaprotocols can be used to implement protocol negotiation, as *Proposal* demonstrates

by using a parameter to reference a protocol: r0 can propose the ceremony to enact, provided it matches the same interface. However, not all metaprotocols involve negotiation; a simple metaprotocol could simply communicate the relevant role bindings to each participant.

Metaprotocols cleanly separate discussion about the protocol from discussion of its enactment. They are fully compatible with BSPL even without our syntax enhancements and may be easily written for existing protocols. Metaprotocols do not reduce flexibility but enable it. Without a metaprotocol, role binding is not flexible but merely unspecified, leading to tight coupling and hidden complexity in a multiagent system.

7.3.2 Metaprotocol Generation

A simple algorithm for generating a metaprotocol from an existing protocol is as follows:

1. Compute a contact graph among the roles, with directed edges following the message transmissions.
2. Identify the initiating role, possibly with user input.
3. Using the contact graph compute the rank of each role (distance of each role from the initiator).
4. Generate a message from the initiator that binds a metaprotocol ID, and another binding the protocol ID.
5. Generate messages from each agent to their contacts of higher rank (flowing from the initiator), inviting them to the protocol and naming any mutual contacts they have invited.

The above algorithm generates introduction messages for each role according to the contact graph of the original protocol; in this way, agents are responsible for inviting only agents they contact. The rank-based invitation approach produces a minimal metaprotocol; that is, the smallest number of single-recipient messages that cover all of the invitations. As such, the complexity of the metaprotocol is no worse than the original protocol; it has approximately one message per role.

The resulting metaprotocol can be customized to support domain requirements. For example, the metaprotocol for a purchase protocol can be customized so that the seller chooses the transaction ID, even though the customer initiates the interaction. Or, in the wedding example, the metaprotocol can specify that the proposer will invite the witness even though the judge is the only agent who contacts the witness during the ceremony.

Using the above algorithm on *CourtWedding* and arbitrarily selecting R as the initiator, we can derive the following:

1. R sends messages to J and E; J sends to R, E, and W; E sends messages to R; W sends to J

2. R has rank 0 (as the selected initiator), J and E have rank 1, and W has rank 2

In general, the initiating agent could be identified from the protocol by looking at which messages declare the primary keys of the protocol or depend only on external parameters. However, in this particular example, the initiator for the metaprotocol (R) is not the same as the first agent to communicate in the CourtWedding protocol (J), so it must be explicitly selected as the initiator.

With the above contact graph and rank information, we can derive the comprehensive metaprotocol shown in Listing 7.4. The *Generated* protocol in Listing 7.4 is simple because the rank sorting prevents J from inviting E and vice-versa. E is arbitrarily invited first in, e.g., alphabetical order. Our example algorithm is far from the only possibility; alternative algorithms could generate a multicast message that simultaneously invites multiple agents or generate a more complex metaprotocol to invite them in either order.

Listing 7.4: Generated Metaprotocol for CourtWedding

```
Generated(out r0, r1, r2, r3: role,
  out R, E, J, W: role,
  out key mID, out cID,
  out vowR, vowE, out license or out objection) {
  out r0 -> out r1: InviteE[out mID, out cID, out R: role=r0, out E:
     role=r1]
  r0 -> out r2: InviteJ[mID, cID, R, E, out J: role=r2]
  r2 -> out r3: inviteW[mID, cID, R, E, out W: role=r3]
}
```

We used type and value constraints (e.g., '`role=r2`', where '`role`' is a type constraint and '`=r2`' is a value constraint) for the ⌜`out`⌝ role parameters; both are optional but make the protocol clearer.

7.3.3 Self-Contained Protocols

Although the previous approach to protocol configuration is conceptually simple and compatible with existing BSPL semantics, it can be somewhat impractical because every nontrivial

protocol must have a corresponding and cumbersome metaprotocol. Those metaprotocols can add many messages as overhead solely for communicating contact information.

We propose further extensions to BSPL to support more practical, progressive protocol instantiation at the cost of some complexity.

Listing 7.5: Self-Contained Wedding Protocol

```
Self-Contained-Wedding(out Propose(R), Propose(E): role
  out key cID, out rejection or
    (out vowR, out vowE, out license or out objection)) {
  private (J)udge: role, (W)itness: role, signature

  out R -> out E: Propose[out cID]
  E -> R: Accept[in cID, out acceptance]
  E -> R: Reject[in cID, out rejection]
  R -> out J: Schedule[in cID, in E, in acceptance, out date]
  J -> R, E: Ask[in cID, out questions]
  E -> J, R: EVow[in cID, in questions, out vowE]
  R -> J, E: RVow[in cID, in questions, out vowR]
  R -> out W: Invite[in mID, in cID, in J, out invitation]
  Witness(J, W, cID, out signature or out objection)
  J -> E, R: Marry[in cID, in vowE, in vowR, in signature, out
     license]
}
```

The *Self-Contained-Wedding* protocol in Listing 7.5 does not rely on a separate metaprotocol but instead specifies the process and propagation of role binding internally.

Self-contained protocols are not superior to metaprotocols, but may be an optimization if they can, e.g., communicate role bindings alongside other information in the protocol.

7.4 Composition

Composition is another important aspect of information protocols. Implicitly, agents can participate in multiple protocols at the same time and must keep them separate. Protocols can be explicitly composed into a larger protocol specification. Provided all the agents know the composite protocol, composition adds no complexity to an implementation. However, the following concerns arise with respect to instantiation: 1. enabling interface flexibility, so that protocols may share or hide information as necessary with the broader composition 2. agents should not be required to know all compositions containing the protocols they

support 3. enabling composition of protocols without needing a common key.

7.4.1 Interface Flexibility

Existing information protocol languages BSPL and Splee have rigid protocol interfaces; all public parameters must be bound for an enactment to complete. Hence, protocol designers must encode multiple termination states—such as accepting or rejecting a marriage proposal—in a single result parameter.

Listing 7.6: Information Hiding Example

```
OpaqueOffer(in B,S: role, in key oID, out item, amount, result) {
  private acceptance, rejection
  S -> B: Offer[oID, out item, out amount]
  B -> S: Accept[oID, item, amount, out result]
  B -> S: Reject[oID, item, amount, out result]
}

Purchase(out B, S: role, out key pID, out transaction) {
  private request, item, amount
  out B -> out S: RFQ[out pID, out request]
  OpaqueOffer(B, S, pID, item, result)
  S -> B: Ship[pID, result, out transaction]
  // no way to distinguish outcomes in composition
  S -> B: Cancel[pID, result, out transaction]
}
```

OpaqueOffer in Listing 7.6 has only a single result parameter in its public parameter line, hiding the actual response of B. If used in a composition such as *Purchase* in the same listing, there is no way to distinguish the outcomes except through internal agent logic applied to the parameter bindings.

However, when protocols are used in a composition the limitation becomes clearer; what if some subsequent actions should be available only after acceptance, and others only after rejection? For example, the wedding ceremony should be enacted only if the proposee accepts the proposal. In this case, the ceremony has an explicit dependency on the outcome of the proposal, so it should not be hidden.

Our proposed solution is to enhance protocol interfaces to support more complex Boolean expressions over the protocol's parameters. Where before the interface was a conjunction over the public parameter bindings, now parameters can be grouped into disjoint clauses.

Listing 7.7: Flexible Interface

```
TransparentOffer(in B, S: role, in key oID, out item, amount, out
   acceptance or rejection) {
  private acceptance, rejection
  S -> B: Offer[oID, out item, out amount]
  B -> S: Accept[oID, item, amount, out acceptance]
  B -> S: Reject[oID, item, amount, out rejection]
}

Purchase(out B,S:role,out key pID,out package or out canceled) {
  private request, item, amount, acceptance, rejection
  out B -> out S: RFQ[out pID, out request]
  TransparentOffer(B, S, pID, item, acceptance or rejection)
  S -> B: Ship[pID, acceptance, out package]
  S -> B: Cancel[pID, rejection, out canceled]
}
```

TransparentOffer in Listing 7.7 shows how Boolean parameter expressions enable more transparent interfaces that expose multiple termination states, which can then be handled separately in a composition. The parameters acceptance and rejection are normal parameters joined into a disjunctive clause by the `or` keyword; only one of the two bindings is necessary for completion. Thus, an enactment of *TransparentOffer* is complete when the parameters oID, item, amount, and either acceptance or rejection are bound.

Note that because messages are elementary protocols, they may use Boolean parameter expressions. However, this flexibility can obscure the meaning of the messages and so should be used with care. For example, in a wedding or purchase protocol, sending an explicit *Accept* message forms a commitment unambiguously and can be interpreted at the event level without inspection. By contrast, interpreting a more abstract `Decide[ID, out acceptance or out rejection]` requires examining the payload.

7.4.2 Constituent Protocol Independence

Semantically speaking, when a protocol is referenced in a composition, its parameters are substituted by the parameters passed in the reference. Thus, a bank transfer protocol may involve a parameter named amount that is used to transfer payment in a purchase protocol or refund in a protocol for processing returns. Proper encapsulation and modularity require that the bank is able to enact its role in the transfer without being aware of the broader purpose. This requirement is violated by current implementations because the bank would need to

interpret the substituted parameter names instead of the expected amount.

Listing 7.8: Dependent Representation

```
// (D)ebtor, (C)reditor, (B)ank
// original: D -> B: Transfer[in ID, out amount, C]
{"ID": <uuid>, "amount": 100, "C": "Creditor"}
// (B)uyer, (S)eller, Ban(K)
//composed: B -> K: Transfer[in pID, out payment, S]
{"pID": <uuid>, "payment": 50, "S": "Seller"}
```

Listing 7.8 shows two versions of a message schema. The first schema (line 2) is the generic transfer expected by the bank with the parameter amount specifying the amount to be transferred. Its representation in JSON is given on the following line, with the expected parameter names as dictionary keys. Using the transfer protocol in a purchase composition would result in substituting the parameters, as shown on line 5. The subsequent JSON representation uses the parameter names from the composition, which may not be understood by the bank.

To better support protocol independence, we propose a simplification of message representations, where each message is encoded as a (name, payload) pair, and the payload is simply an array of unnamed parameters. The name must uniquely identify the message in the MAS; it could be an IRI or a more compact encoding defined by the MAS for efficiency. The payload is interpreted by referring to the message schema.

Replacing named parameters with a uniquely identified message schema minimizes the context sensitivity of the message and simplifies interpretation. That is, referring to the protocol that specifies the message provides the information necessary for decoding the message: which parameters are included in which order. However, all semantic information is left out of the encoding, so each agent can interpret the message according to its understanding of the composition. This is similar to message encoding libraries such as Protocol Buffers [79], which use an external schema to parse the messages, enabling compact encoding and backward compatibility across schema versions (since names can change without changing the encoding).

Listing 7.9: Independent Representation

```
// original: D -> B: Transfer[in ID, out amount, C]
[<uuid>,100,"Creditor"]
// composed: B -> K: Transfer[in pID, out payment, S]
[<uuid>,50,"Seller"]
```

In Listing 7.9, leaving out the parameter names does not make the schema ambiguous

but does make it independent of its use in a composition. The bank receives the information necessary to execute a transfer without being confused by the buyer's intent to use the transfer as payment for a purchase.

7.4.3 Global and Local Keys

We have added syntax for distinguishing global and local keys. The purpose of the distinction is to enable *invertible* relationships between protocol keys, without which protocols in composition must share a common key to share information.

Listing 7.10: Noninvertible Key Composition

```
Noninvertible(B, S: role, key ID, oID, dID,
  item, price, package) {
  out B -> out S: Start[out ID]
  Order(B, S, ID, oID, item, price)
  Deliver(B, S, ID, dID, item, package)
}
```

As shown in Listing 7.10, the *Order* and *Deliver* protocols must share an *ID* to correlate the information, which intrusively imposes awareness of the composition on those protocols. If written independently, each would need only its own key.

An invertible relationship exists between keys if they are uniquely associated with each other; that is, an agent receiving a message containing one key can correlate it to an enactment of another protocol using the other key. If a relational data model is used, the agent's adapter can construct queries for enactment information using an association table to join information from the two protocols. Or, the agent's adapter can use a dictionary to map the unique keys to a single shared enactment object.

Listing 7.11 shows a composition where the keys are universally unique. Because both oID and dID are unique, the *TrackingInfo* message declares an invertible relationship between them, enabling both agents to correlate the messages inside *Deliver* to the broader enactment without needing to propagate a common key.

Listing 7.11: Invertible Key Composition

```
Invertible(B, S: role,
  key oID, dID // new default of global keys
  item, price, package) {
  Order(B, S, oID, item, price)
  S -> B: TrackingInfo[in oID, out dID]
  Deliver(B, S, dID, item, package)
}
```

Global keys are the default. They are identified by the sole qualifier `key`, and must be guaranteed unique within the MAS. We include the `local` keyword to cover all other kinds of keys (e.g., sequence IDs or timestamps). Such keys are required to be unique only within the context of the global keys. Thus, a local key must be used in the context of a global key, possibly scoped further by other ⌜in⌝ local keys. If multiple local keys are ⌜out⌝ in the same message, they form a hierarchy based on the order in which they are given.

Listing 7.12 gives an example of how local keys might be used in a protocol. It specifies a support request protocol, where B opens a ticket, specifying a topic and generating a new unique ID. Then, B and S can continue replying with an indefinite number of messages, each uniquely identified with reply IDs (possibly sequential or timestamped), bound to brID and srID, respectively. These response IDs need only be unique within the context of the support request because they are local keys.

Listing 7.12: Local Key Example

```
Support (B, S: role,
  key ID, // global key
  local key brID, srID, // local keys
  topic, closed) {
  B -> S: OpenTicket[out ID, out topic]
  S -> B: SReply[in ID, in topic, out srID, out content]
  B -> S: BReply[in ID, in topic, out brID, out content]
  S -> B: CloseTicket[in ID, out closed]
}
```

7.5 Realization

Programming agents to support dynamic role binding and multiple parallel interactions need not be complicated but is more involved than supporting a single protocol.

In previous programming models for information protocols (PoT [48], Bungie [49], Deserv

[51]), agents adopt a single role in a single protocol. Architecturally, each agent was instantiated using a protocol adapter configured with the role and protocol that the agent supported. The adapter was configured with static role bindings that were bound outside of the protocol enactment. However, it may be possible to change these bindings; the adapter supported only a single binding for each role and protocol. Supporting additional protocols would require the agent to run multiple adapters with separate endpoints.

7.5.1 Protocol Adapter

The protocol adapter must support multiple roles across several protocols and metaprotocols. Each protocol is loaded in the context of a specification, which may define terms used in the protocols.

The adapter would support a collection of initial protocols (or metaprotocols), along with the roles it is willing to play in those protocols and handlers for those messages. These initial protocols could be implemented according to patterns such as peer-to-peer introductions or a central registry, among others, which we examine further in Section 7.6.

During the enactment of the initial protocols, the agent will either propose enactments of protocols as specific roles or receive invitations to take on a role. Fortunately, the agent need not specify special-case handlers for these invitations any different from other messages; the information structure of the protocols is enough to guide further enactment.

Roles have become parameters bound within the context of a protocol or metaprotocol instance key. No longer are roles implicitly part of the message schema. A role must either be explicitly bound in sending a message to a new role or provided by the enactment history. Provided the protocol (or metaprotocol) is verified to be live and enacted without violation or error, the agents will receive the contacts they need to complete their role. How an agent selects or approves the specific role bindings is left to their internal logic.

7.5.2 MAS Adapter

To better support agent discovery and selection, we propose a new component, the *MAS Adapter*. The MAS adapter is optional; simple agents (such as web services) may handle each enactment independently. Provided each role binding is used only within the context of an enactment, the information management provided by the protocol adapter is sufficient. The MAS adapter is responsible for remembering the contacts and the history of interactions with them beyond a single enactment.

The MAS adapter implements a registry of known agents. New entries are automatically added as agents are introduced. Each entry contains information about the protocols and

roles that the agent is believed to support, based on the introduction. Each entry contains information about the roles that the agent has played and the corresponding enactment histories. Additional information can be added according to domain requirements, such as quality of service ratings. An agent can query its MAS adapter to find agents to enact protocols with.

7.6 Evaluation

We now show how the above concepts can be applied toward capturing canonical role-binding patterns in practical decentralized applications.

7.6.1 Preconfigured Contacts

In peer-to-peer applications, it is common for the peers to be preconfigured with the knowledge of some bootstrap nodes to get the application going. Network applications are typically preconfigured with a DNS server for purposes of resolving domain names.

In our context, preconfigured contacts are those an agent is given before it is initiated. Preconfigured contacts are implemented in the agent internals and used to bind the initial ⌜out⌝ roles of the first protocols the agent enacts. This is a simple pattern, but it is a necessary component of all more complex patterns.

Listing 7.13: Preconfigured Contacts Example (JSON)

```
{"Seller":["http://storeA.com/agent","http://storeB.com/agent"],
 "Bank": ["http://bank.com/agent"]}
```

Listing 7.13 shows how contacts might be declared in a JSON file. Preconfigured contacts are not hard coded bindings; they are candidates that are bound to roles dynamically according to the protocol structure and agent logic.

7.6.2 Central Registry

The next level of complexity from a collection of preconfigured contacts is a central registry provided as a service by another agent. Discovering agents from a central registry is common in services. The Uber application, e.g., uses Uber's (the organization's) central registry of potential drivers to bind as the pickup driver in a particular transaction with a customer.

A central registry is simple in that it means that an agent configuration requires only a single registry connection. In addition, a registry enables more dynamic and scalable peer discovery than preconfiguration. As each agent comes online, they can enact a registration

protocol, notifying the registry of their existence and willingness to perform specific roles. To enact a protocol, an agent can first query the registry to discover potential peers. As the example in Listing 7.14 shows, the pattern could be implemented as two protocols, where a single well-known agent uses its MAS adapter to remember and recommend contacts to other agents.

Listing 7.14: Central Registry

```
Registration(out A, R: role,
  out key ID, out endpoint, confirmation
  out set protocols: protocol) {
  out A -> out R: Register[out ID, out endpoint, out protocols]
  R -> A: Confirm[in ID, in endpoint, in protocols, out confirmation]
}

Discovery(out Q, R: role,
  out key ID, out protocol: protocol, out set agents: role) {
  out Q -> out R: Query[out ID, out protocol]
  R -> Q: Introduce[in ID, in protocol, out agents]
}
```

7.6.3 Peer Sharing

Peer sharing is characterized by the absence of any distinguished system nodes that support discovery. Peer-to-peer discovery and binding are common in MAS and distributed systems. For example, it is used in referral networks [174], in the Contract Net [159]), and in leader election protocols. Further, mesh networks and IoT-based systems typically invoke the peer-to-peer pattern.

To find a desired peer, each agent checks its own MAS adapter and queries its neighbors. The exact nature of the peer selection and query process is application-specific, but generally, each query will return more peers; either the desired peer will be among them, or they can be queried in turn.

Listing 7.15: Peer Discovery

```
Discover(out P1, P2: role, out key ID,
  out set protocols: protocol, out set neighbors: role) {
  out P1 -> out P2: Query[out ID, out protocols]
  P1 -> P2: Introduce[in ID, in protocol, out neighbors]
}
```

In the peer discovery protocol in Listing 7.15, there is no need for registration; each agent simply needs an initial bootstrap peer to connect to. Each time an agent asks a peer for neighbors, it naturally introduces itself and reveals the protocol(s) it is interested in.

7.7 Discussion

We conceptualized the problem of instantiating a MAS in terms of binding the roles of the protocol that models the MAS. Although multiagent systems are typically thought of as being open in the sense of agents dynamically joining and leaving the system, MAS approaches have not paid sufficient attention to the operational aspects of the problem. Pippi demonstrates how MAS can be instantiated dynamically by showing how a protocol's roles are bound. Our approach supports dynamic (not hard-coded) and late (just-in-time) role bindings: not all roles in a protocol need to be bound before enacting the protocol. We discussed a possible realization of our approach and demonstrated how commonly used patterns can be captured in our approach.

Dastani et al. [58] consider compatibility between agent and role specifications as a basis for an agent's decision whether or not to play a role. Such internal decision making, although practically relevant, is outside the scope of Pippi. Pippi is concerned with public aspects of decision making as reflected in protocol enactments. HAPN [170] supports dynamic role binding; e.g., an auction's winner is dynamically bound. It does not give a general metaprotocol-based approach that enables discovery and bindings of roles. Grenna et al. [82] extended JADE to enable agents to enact roles, but within the context of an organization rather than a protocol.

Günay et al. [85] propose a metaprotocol by which agents can reach agreement on the commitment protocol: All agents should accept that they will create the commitments involved. They assume role bindings (e.g., that there are customer and merchant agents). Thus, Pippi complements their work. McGinnis and Robertson [110] propose protocols as first-class abstractions for composing open MAS. Pippi agrees with their intuition and demonstrates how role bindings generated as information in the messages of one protocol (the metaprotocol) can be used as roles (the senders and receivers of messages) in another protocol via composition.

Multiagent systems are conceived of as open systems in the sense that agents can join and leave the MAS. Mazouzi et al. [109] give an early example of an agent wanting to join several groups and show that abstract specifications can be refined depending on requirements. Role binding, as we formalize here, is akin to "joining" a MAS. However, because parameter bindings and hence role bindings are immutable, "leaving" a MAS is not unbinding the role.

Leaving would be captured by communicating that the agent has left the MAS. Immutability of information gives the immutability of events, which is necessary for realism. For example, once bound as a partner in some wedding, an agent is always a partner for that wedding. Divorce is not an unbinding of the agent to the partner role but a change in the normative relationships [42] between the partners, perhaps via the creation of a new MAS. For example, two divorced people may have joint custody of their children and may thus function together in a MAS, albeit not the same MAS as when they were married. Such ideas merit further study.

Minsky and Murata [115] discuss the robustness and manageability of a MAS modeled as a law-based society [114]. In considering robustness, Minsky and Murata bring up both static and dynamic role binding in the law, which is realized as a set of Prolog-like rules. The static bindings are given as facts, and the dynamic bindings are established via events referred to in the rules. Our approach is compatible with rule-based specifications (e.g., commitments or other norms [42]) but captures the general operational aspects of role binding in a decentralized system via protocols.

Ferrando et al. [70] consider the enactability [65] of protocols specified in a language for specifying execution traces, where the constraints specify message ordering. They evaluate the enactability of protocols under different infrastructure assumptions, e.g., with or without FIFO delivery. Ferrando et al. do not consider the problem of role binding. Other early work on formally specifying [146] and enacting [143] multiagent interactions also doesn't address role binding as an explicit concern.

Pippi's protocols can be enacted with the minimal set of infrastructure assumptions, namely that only sent messages be delivered. Further, Pippi's protocols can be interleaved without requiring to be composed (this is a property of information protocols [44]), which is not a possibility with trace-based approaches. Role binding for a protocol can be interleaved with the enactment of the protocol, as shown in the *Proposal* metaprotocol in Listing 7.3.

Chocron and Schorlemmer [37] consider the problem in open MAS where agents know they are participating in the same interaction (specified in temporal logic) but have different message vocabularies. They propose a method by which agents can dynamically align their vocabularies, that is, learn the mappings between their vocabularies. Pippi currently assumes a shared vocabulary (whatever is in the shared information protocols) but would benefit from alignment techniques in scenarios where the assumption does not hold. Arguably, aligning the "same" information protocol with different vocabularies is a more natural and challenging problem to address since information protocols can be more flexible than message ordering constraints specified in temporal logic.

Rocha and Brandão [136] apply multiagent systems to model dynamism in Internet of

Things applications. They model devices via agents that enter and leave the system. Pippi enables realizing such dynamism in a general way with clean, modular representations.

JADE [23] supports the discovery of agents via a directory facilitator that provides a yellow pages service. JADE supports FIPA interaction protocols by providing endpoint implementations (classes and methods) that agents can use to implement interactions with others. However, JADE is not equipped to enable building multiagent systems off protocol specifications as Pippi's adapter and programming model enable. Further, discovery, although important, differs from role binding. In the wedding scenario, an agent can discover other agents looking for prospective partners via a directory service but would need to *bind* one of them to Proposee to enact Proposal. Briola et al. [33] demonstrate how JADE can be used to discover agents and implement a peer-to-peer system. As the foregoing examples demonstrate, Pippi supports both discovery and role binding and thus enables the realization of arbitrary applications as a peer-peer system.

Carriero and Gelernter's [34] tuple space approach for coordinating processes has been influential in multiagent systems. It features as the underlying coordination mechanism in CArtAgO [133], which itself is part of JaCaMo [29], which Baldoni et al. [18] use to support the implementation of commitment-based business processes. Tuple spaces (like logic programming) are attractive for their information-based abstractions (one works with tuples of information). However, they represent a shared-memory approach, which is not suitable for building decentralized MAS. It is widely argued that a tuple space decouples the readers and writers of information, but that argument holds only where the writers don't care who reads the information (as is the case in the classic "readers and writers" synchronization problem). In MAS, agents communicate with particular agents, e.g., the Proposer wants to communicate different information to the Proposee and to the Judge. Such coupling between agents is specified in the protocol. It would, however, be interesting to investigate if tuple spaces can be used to implement an agent's local state since it provides information-based abstractions.

Chapter 8

Kiko – Decision-Oriented Programming for Decentralized Multiagent Systems

In a multiagent system, an agent's communications represent its decisions. An interaction protocol is therefore a specification of decentralized decision making by agents. However, current programming models offer poor decision making-oriented abstractions for implementing agents. In particular, they lack abstractions that bridge an agent's internal decision logic with its public decisions.

We adopt the framework of declarative information protocols and present *Kiko*, a protocol-based programming model. Kiko's main abstraction is that of a *decision maker*. 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 some mutually compatible decisions (by sending the corresponding messages). Kiko agents are guaranteed protocol-compliant—they never make an incorrect decision. Moreover, by completely abstracting away the underlying communication service and by supporting practical decision-making patterns, Kiko enables agent developers to focus on the business logic. We provide an operational semantics for Kiko agents and establish the correctness of Kiko agents and that they can realize any protocol enactment.

This chapter is adapted from a draft of *Kiko – Decision-Oriented Programming for Decentralized Multiagent Systems* written in collaboration with Amit Chopra and Munindar Singh for submission to AAMAS.

8.1 Introduction

A sociotechnical system (STS) is characterized by interactions between real-world principals such as humans and organizations for some purpose, e.g., ebusiness, healthcare, and so on [129, 150]. Today, it is common to realize an STS as a conceptually central service, e.g., a Web service. A multiagent system (MAS) offers a decentralized alternative for realizing an STS. In a decentralized MAS, each principal would be represented by an autonomous agent that encodes the principal's decision logic and interacts with other agents 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 protocol such as *Purchase* may specify when agents may send specific messages in the above-introduced e-business setting.

Protocols are specified abstractly with reference to roles that agents may play. The value of a protocol is that for each role, it provides a guide for independently implementing agents playing that role. Broadly, implementing an agent according to some role means fleshing out the role with private decision logic (internal reasoning) that results in messages being emitted, that is, decisions being made. For example, let's say 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.

Implementing protocol-based agents is, however, nontrivial, especially when the involved protocols support flexible decision making by agents via asynchronous communication. Traditional approaches for specifying protocols, e.g., [70, 90, 122, 170] specify message ordering, which limits flexibility [44]. JADE [22, 23], a programming model for multiagent systems, is noteworthy for its early support for FIPA protocols [71]; however, the FIPA approach is long outdated [144] 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 [31] and JaCaMo [29] provide valuable cognitive abstractions for encoding an agent's internal reasoning; however, they do not support the protocol-based construction of agents. Commitment-based approaches for programming agents [85, 167], while promising, either often rely on centralized commitment stores [18] or do not adequately dwell upon the operational aspects related to asynchronous communication [66], thus sidestepping the practical problem of decentralized

decision making. Agent-oriented software engineering methodologies, e.g., Prometheus [125], have traditionally incorporated protocols as design abstractions; however, they offer little in terms of a practical programming model for 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* address 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 would be to write the set of decision makers.

Kiko guarantees an agent's compliance with the roles its plays and supports a variety of practical and interesting decision making patterns that would be difficult to accommodate in alternative approaches. These patterns include correlation, cross-enactment reasoning, emission sets, and multiprotocol reasoning. The common refrain one often hears is that programming models should support programmers in focusing on the business logic [106, 158]; Kiko takes this support to new heights. Notably, in providing a decision-based interface for programming agents, the programming model abstracts away the communication service that transports messages between agents. In particular, decision making is oblivious to the order in which messages are received by an agent. Actual message emission is also handled transparently in the programming model.

In addition to demonstrating Kiko's support for decision making, we contribute a formalization of the programming model and prove its soundness and completeness with respect to possible protocol enactments. We also present an optimized compliance checking method and establish its validity.

8.2 Information Protocols Introduced

A protocol-based programming model for agents presumes a language in which to specify protocols. We adopt BSPL [147], a declarative protocol language that eschews the specification of message ordering and instead specifies information constraints.

Listing 8.1: 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 state of an agent (its history of message observations), an agent can send any message that satisfies the specified causality and integrity constraints.

We illustrate the main ideas of information protocols via Listing 8.1, which 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 8.1. 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.

Integrity constraints can inform the behavior of an agent through automatic *correlation* of information. A Buyer enacting multiple instances of *Purchase* from Listing 8.1 with the same Seller can correlate the received *Quote* messages with the appropriate *RFQ* message based on the key ID.

In a message schema, every message parameter is adorned ⌜in⌝, ⌜out⌝, or ⌜nil⌝. Adorn-

ments 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 8.1, SELLER must know item before it can send *Quote*, and in doing so produces a binding for price.

Note that 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*.

The notion of integrity and causality applies to protocols generally. In *Purchase* in Listing 8.1, 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.

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 [48, 49]. Thus an unordered, lossy transport like UDP is sufficient for enacting BSPL protocols.

8.3 The Kiko Programming Model

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

Figure 8.1 shows the main components of the agent architecture under Kiko relevant to enacting protocols. The MAS-Agent and Decision Makers are components provided by the agent programmer (indicated by the border). The Protocol Adapter is a generic component (indicated by the lack of border) provided by Kiko that understands information protocols and provides an API for plugging in Decision Makers. The Communication Service is anything that provides asynchronous messaging between agents. Our current implementation uses UDP, which is unordered and unreliable (lossy).

MAS-Agent Configuration. A protocol specifies a MAS abstractly via reference to roles. A *concrete* MAS for a protocol is identified by a UUID and gives the agents that play specific roles. 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

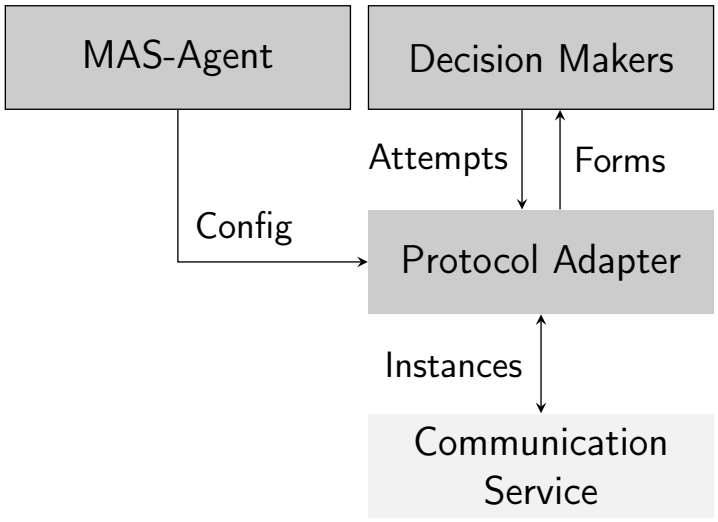


Figure 8.1: The Kiko agent architecture.

knowledge; Listing 8.2 gives such a configuration for agent `Bob`.

Listing 8.2: Bob's MAS-Agent 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 8.2, `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 8.2, SALLY has two.

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.

Let's consider an example to illustrate the foregoing ideas. Consider that `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, forms and instances produced from them contain a MAS identifier (here, `5feceb66`); think of it as the value for the reserved and implicit parameter system in every message schema. Bob's programmer may have written a decision maker that fleshes out the above form into instances such as `Bob` -> `Sally`: *RFQ*[`5feceb66`, `bat`, `5`] and `Bob` -> `Sally`: *RFQ*[`5feceb66`, `ball`, `6`] based on some decision logic. These instances are passed on to the adapter for emission. Listing 8.3 shows a decision maker (in Python, the language of our implementation) 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 8.3: 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`, `bat`, `5`]. Then, in addition to the *RFQ* form specified above, the following forms would also be available to `Bob`: `Bob` -> `Sally`: *Buy*[`5feceb66`,

bat, 5, (done)] and Bob -> Sally: *Reject*[5feceb66, bat, 5, (done)]. Bob's programmer may have implemented a decision maker that chooses from one of these two available forms (e.g., based on how acceptable the price is), fleshes it out by binding done and instructs the adapter to emit the resulting instance.

Listing 8.4: 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 contains incompatible instances. Such a decision maker would be buggy. Specifically, Listing 8.5 is buggy because Bob creates instances for both *Buy* and *Reject* in the same enactment. This emission attempt fails because *Buy* and *Reject* in the same enactment are mutually exclusive according to Listing 8.1 (because both bind ⌜out⌝ done); neither will be emitted.

Listing 8.5: 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")
```

The bug in Listing 8.5 brings out a remarkable aspect of Kiko. Kiko allows decision makers (programmers) to choose sets of instances to emit. Whereas each of the instances in the set (e.g., the *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 (the *Buy* and the *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 allow a decision maker to work on at most one form at a time. Then, its emission by the adapter would be guaranteed. We support such a decision maker as a special case; however, as we see below, not allowing the emission of sets of instances

would be too inflexible from the point of view of decision making.

By default, decision makers are invoked automatically by the adapter whenever a communication event occurs (e.g., the decision maker in Listing 8.4, which, unlike the decision maker in Listing 8.3, lacks an `event` specification). Event-based invocation enables some optimization: first, the agent is not polling to wait for enough information to make a decision; and second, the event can help focus on the specific enactment(s) affected. 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 $\mathsf{payload}(m) = \langle \mu, \imath_m, \rightsquigarrow_m, \vec{i}_m, \vec{o}_m \rangle$, 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 $\langle b_p | p \in \mathsf{payload}(m)] \rangle$ in $\mathcal{M}_m$.

We use $\widehat{m}[...]$ for projecting parameters to their bindings in the message instance; e.g., $\widehat{m}[\imath_m]$ 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 $\mathsf{instance}(\widehat{m})$ holds if and only if all of its parameters are bound: $p \in \mathsf{payload}(\widehat{m}), \widehat{m}[p] \neq \varnothing$.

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

A *form* is an association where the $\ulcorner\mathsf{out}\urcorner$ 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 $\ulcorner\mathsf{out}\urcorner$ parameter has a null value. That is, $\forall p \in \mathsf{payload}(m) \setminus \vec{o}_m : p \neq \varnothing$ and $\exists p \in \vec{o}_m \colon \widehat{m}[p] = \varnothing$

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

We introduce the notion of context to capture enactments in 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 $\ulcorner\mathsf{in}\urcorner$ parameter bindings match those from observed instances that share context (consistency), and its $\ulcorner\mathsf{out}\urcorner$ and $\ulcorner\mathsf{nil}\urcorner$ 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 $\mathsf{consistent}(N, M)$ holds) if and only if the $\ulcorner\mathsf{in}\urcorner$ bindings in N are the same as bindings from associations that share context in M:

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

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

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

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

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

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 $\mathsf{derived}(\widehat{m}, H_a)$ holds) if and only if all of $\widehat{m}$'s $\ulcorner\mathsf{in}\urcorner$ parameters are drawn from instances that share context in the history:

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

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

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

We also say that $\mathsf{enabled}(a, H_a) \subset \mathcal{F}$ 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 $\widehat{m} \in d(Q) \implies \mathsf{instance}(\widehat{m}) \wedge \exists \widehat{m} \in Q : \widehat{m}[\imath_m, \varkappa_m, \vec{i}_m] = \widehat{m}[\imath_m, \varkappa_m, \vec{i}_m]$.

8.3.1 Decision-Making Challenges and Solutions

We highlight select decision making patterns supported by Kiko.

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 8.4, correlated Reject forms are found by the ID of the Buy forms.

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 8.6).

Listing 8.6: 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)
```

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 8.7. 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 8.7: 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 8.8 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 8.8: 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)
```

Emission sets

Some kinds of decision logic may require the emission of a set of messages instances in one go. For such cases, Kiko enables expressing the decision logic as a single decision maker even though it may result in the emission of several instances.

Listing 8.9 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 8.9: A decision maker that sends *Buy* in some contexts and *Reject*s in the others.

```
1 @adapter.decision
2 def select_gifts(forms):
3     best, rest = best_combo(forms)
4     for b in best:  # buy the best items
5         b.bind(done=True)
6     for r in rest:  # reject the rest
7         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.

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 8.10: 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 8.10 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 8.4 that emits *Buy*s need not change at all; the disabled *Buy*s are simply not provided to the decision maker for consideration.

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 *Purchase* included a *Deliver* message from Seller that depended on payment provided by *Buy*:

Listing 8.11: 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. Because the messages in Listing 8.12 do not change SELLER's messages, 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 8.12: 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]
```

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 8.13 shows an example where an enabled form of *Quote* is fleshed out.

Listing 8.13: Single Form Decision Maker for *Quote*.

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

8.3.2 Adapter Implementation

Figure 8.2 blows up the adapter from Figure 8.1 to highlight its internal components (highlighted in green).

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

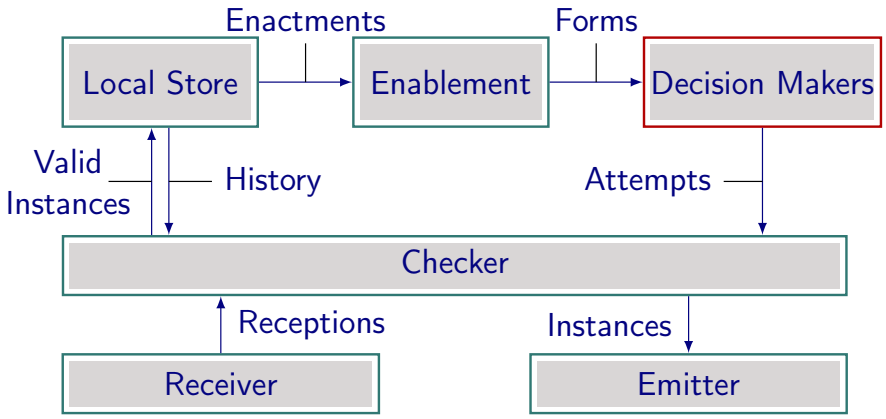


Figure 8.2: Adapter implementation.

emit a set of messages (Definition 12). If an attempt is validated, then the instances in it are added to the Local State 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, $\mathsf{check}_s(T, H_a)$ holds if and only if:

1. a is enabled to send every $\widehat{m}$ in T:

   $\forall \widehat{m} \in T, \mathsf{enabled}(\widehat{m}, a, H_a)$

2. T is out-, and nil-compatible with T:

   $\mathsf{compatible}_{\vec{o}}(T, T) \wedge \mathsf{compatible}_{\vec{n}}(T, T)$

If $\mathsf{check}_s(T, H_a)$ 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 State, 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, $\mathsf{check}_r(\widehat{m}, H_a)$ holds if and only if:

1. $\widehat{m}$ is receivable by a: $a = \widehat{m}[\nearrow_m]$

2. $\widehat{m}$ is consistent and out-compatible with the history:

   $\mathsf{consistent}(\{\widehat{m}\}, H_a) \wedge \mathsf{compatible}_{\vec{o}}(\{\widehat{m}\}, H_a)$

If $\mathsf{check}_r(T, H_a)$ 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 4 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 3, every context that shares key bindings with the observed instance $\widehat{o}$ is checked to see if it enables any instances of m. Lines 5 and 7 check that the ⌜out⌝ and ⌜nil⌝ parameters of the schema, respectively, are not already bound in the context. Line 9 copies the bindings of the ⌜in⌝ parameters from the context, Line 14 copies the system ID, and Line 16 adds the form to the result set for processing by decision makers.

Algorithm 4: Derive instance of schema from observation.

Input : Message schema m, Message instance $\widehat{o}$

```
1  Q ← {};
3  foreach c ∈ matching_contexts(ô) do
5      o ← ∄p : p ∈ ō_m ∧ p ∈ c.bindings;
7      n ← ∄p : p ∈ n̄_m ∧ p ∈ c.bindings;
9      i ← ∀p : p ∈ ī_m ⟹ p ∈ c.bindings;
10     if o ∧ i ∧ n then
12         m̂[ī_m] ← c.bindings[ī_m];
14         m̂[μ] ← ô[μ];
16         Q ← Q ∪ m̂;
17     return Q;
```

8.4 Protocol Semantics

We now formalize protocol semantics, so that we can prove correctness and completeness of the programming model.

Protocol specifications are abstract, consisting of schemas and roles, rather than agents.

Definition 14 (Protocol)*:* A protocol specification P is a tuple $\langle S, R, I, K\rangle$, where S is a set of message schemas, R is a set of roles, I is a set of public parameters (its public interface for composition), and $K \subseteq I$ is a set of key parameters.

Message schemas are sent between a sender and receiver role, and contain parameter names and adornments to specify information dependencies.

Definition 15 (Message Schema)*:* A message schema m is a tuple $\langle \mathfrak{s}, \mathfrak{r}, \vec{k}, \vec{i}, \vec{o}, \vec{n}\rangle$, where the components are sets of *parameters* of m and refer to the sender role, receiver role, keys, ⌜in⌝

parameters, ⌜out⌝ parameters, and ⌜nil⌝ parameters, respectively. The ⌜in⌝ parameters $\vec{i}$ and ⌜out⌝ parameters $\vec{o}$ are disjoint sets: $\vec{i} \cap \vec{o} = \varnothing$. The key parameters $\vec{k}$ are a subset of the ⌜in⌝ and ⌜out⌝ parameters: $\vec{k} \subseteq \vec{i} \cup \vec{o}$, but the ⌜nil⌝ parameters $\vec{n}$ are disjoint with the payload: $\vec{n} \cap (\vec{i} \cup \vec{o}) = \varnothing$. For clarity, we will sometimes write these components with the schema they belong to as a subscript; e.g., $\vec{k}_m$ for the keys of schema m.

We are working toward modeling an enactment of a protocol specification, using the concept of an *event*, which models either the emission or reception of a message.

Definition 16 (Event)*:* An *event* is a tuple $\langle t, m \rangle$, where m is a message schema and t is either sent or received

Events are observed by roles, forming the roles' histories.

Definition 17 (Role History)*:* A role history H_r is the set of events observed (that is, sent or received) by role r: $H_r = \{\langle t, m \rangle \mid t = \mathsf{sent} \implies r = \mathscr{s}_m \wedge t = \mathsf{received} \implies r = \mathscr{r}_m\}$

An enactment is just the sum of the observations of the roles of a protocol.

Definition 18 (Enactment)*:* An enactment E of protocol P is a set of role histories: $E = \{H_r \mid r \in R_P\}$.

Valid enactments are constructed of viable events; that is, emissions that are enabled for the role based on their information dependencies, and receptions that are preceded by the corresponding emission.

Definition 19 (Viable Event)*:* If E is an enactment and $e = \langle t, m \rangle$ is an event, $\mathsf{viable}(e, E)$ holds if and only if:

1. if $t = \mathsf{sent}$:
   (a) m's ⌜in⌝ parameters are known from other messages:
   $p \in \vec{i}_m \implies \exists n \in H_{\mathscr{s}_m} : p \in \vec{i}_n \cup \vec{o}_n$
   (b) m's ⌜out⌝ and ⌜nil⌝ parameters are not ⌜in⌝ or ⌜out⌝ in any other message in $\mathscr{s}_m$'s history:
   $p \in \vec{o}_m \cup \vec{n}_m \implies \forall n \in H_{\mathscr{s}_m}, p \notin \vec{i}_n \cup \vec{o}_n$
2. if $t = \mathsf{received}$:
   (a) m was sent: $\exists H_{r'} \in E : \langle \mathsf{sent}, m \rangle \in H_{r'}$
   (b) m's ⌜out⌝ parameters are not ⌜in⌝ or ⌜out⌝ in any other message in $\mathscr{r}_m$'s history:
   $p \in \vec{o}_m \cup \vec{n}_m \implies \forall n \in H_{\mathscr{r}_m}, p \notin \vec{i}_n \cup \vec{o}_n$

If $\mathsf{viable}(e, E)$ holds, then e is a *viable extension* of E, and $E + e = \{H_1, ..., H_r \cup e, ..., H_k\}$ where $r = \mathscr{s}_m$ if $t = \mathsf{sent}$ or $r = \mathscr{r}_m$ if $t = \mathsf{received}$.

Finally, we can define a *reachable enactment* as one that can be reached by a (possibly empty) sequence of viable events. This definition is the culmination of our protocol semantics, describing the range of flexibility for a protocol specification. Building on this definition, we can say that an operational semantics would be correct if it only realizes reachable enactments, and complete if there are no enactments that it can't realize.

Definition 20 (Reachable Enactment)*:* An enactment E of protocol P is *reachable* and an element of $\mathcal{E}_P$, the set of all reachable enactments of P, if and only if either:

1. E is an empty enactment:

   $\forall H_r \in E, H_r = \varnothing$

2. Or, E is an extension of a reachable enactment by a viable event:

   $\exists E' \in \mathcal{E}_P, \exists e : \mathsf{viable}(e, E') \wedge (E = E' + e)$

8.5 Operational Semantics

Here, we formalize an agent and MAS computations via a transition semantics.

Figure 8.3 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 agent's history. Next, a set of instances are derived from the message forms by applying a decision maker d to the enabled message 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 8.4 shows an alternative version of the DECIDE rule, 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

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	$a := \langle H_a, I_a, O_a \rangle \in \mathcal{A}$
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	$\mathsf{consistent} \in \mathcal{P}(\mathcal{M}) \to \{\mathbf{T}, \mathbf{F}\}$

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

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

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

Figure 8.3: Notation and core semantics.

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

Definition 22 (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.

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

Figure 8.4: Optimized decision that checks for internal consistency instead of full validity.

Definition 23 (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 $\jmath_m$ in μ then m is sent in the corresponding role history $H_{\jmath_m} \in E$ (that is, $a = \widehat{m}[\jmath_m] \implies \langle \mathsf{sent}, m\rangle \in H_{\imath_m}$)

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

Definition 24 (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 $s' \colon s \xrightarrow{t} s'$ matches some enactment E' reachable from E in a finite number of viable extensions.

We can now prove the following theorems:

Lemma 1*:* Given MAS μ implementing protocol P, $s \in \mathcal{S}_\mu$, and $E \in \mathcal{E}_P$ such that $s \sim E$, $\mathsf{check}_s(\{\widehat{m}\}, H_{\widehat{m}[\jmath_m]})$ implies $\mathsf{viable}(\langle \mathsf{sent}, m\rangle, E)$

Proof Sketch. Because the type of $\langle \mathsf{sent}, m\rangle$ is sent, we are considering the first case in Definition 19.

$\mathsf{check}_s(\{\widehat{m}\}, H_{\widehat{m}[\imath_m]})$ implies $\mathsf{enabled}\{\widehat{m}, a, H_a\}$, which in turn implies $\widehat{m}$ is derived from $H_a \in s$; thus m's $\ulcorner\mathsf{in}\urcorner$ parameters are known from other messages in $H_{\jmath_m}$, satisfying Clause 1a.

Further, $\mathsf{enabled}\{\widehat{m}, a, H_a\}$ implies $\widehat{m}$ is out- and nil-compatible with $H_a \in s$, so m does not share any $\ulcorner\mathsf{out}\urcorner$ or $\ulcorner\mathsf{nil}\urcorner$ parameters with other schemas in $H_{\jmath_m}$, satisfying Clause 1b.

Thus, Definition 19 is satisfied and $\mathsf{viable}(\langle \mathsf{sent}, m\rangle, E)$ holds. □

Lemma 2*:* Given MAS μ implementing protocol P, $s \in \mathcal{S}_\mu$, and $E \in \mathcal{E}_P$ such that $s \sim E$, $\langle \mathit{sent}, m\rangle \in H_{\jmath_m}$ and $\mathsf{check}_r(\widehat{m}, H_{\widehat{m}[\imath_m]})$ implies $\mathsf{viable}(\langle \mathsf{received}, m\rangle, E)$

Proof Sketch. Because the type of $\langle \mathsf{received}, m \rangle$ is $\mathsf{received}$, we are considering the second case in Definition 19.

Our assumption that $\langle \mathsf{sent}, m \rangle \in H_{\jmath_m}$ satisfies Clause 2a.

Then, $\mathsf{check}_r(\widehat{m}, H_{\widehat{m}[\imath_m]})$ implies $\widehat{m}$ is out- and nil-compatible with $H_a \in s$, and thus m should be out- and nil-compatible with $H_{\imath_m} \in E$ satisfying Clause 2b.

Thus, Definition 19 is satisfied and $\mathsf{viable}(\langle \mathsf{received}, m \rangle, E)$ holds. □

Theorem 6. *Given a MAS μ implementing protocol P, every reachable state $s \in \mathcal{S}_\mu$ simulates some enactment $E \in \mathcal{E}_P$.*

Theorem 6 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).

Proof Sketch. Suppose $\mu = \langle P, A \rangle$, where A is some map of agents to the roles they play in R_P.

For a base case, the empty state s_0 matches the empty enactment E_0.

Further, suppose $s \in \mathcal{S}_{\mu,\mathcal{T}}$ matches some $E \in \mathcal{E}_P$ under $\mathcal{T}$. We must show that every state s' reachable from s by a transition in $\{\textsc{Recv}, \textsc{Tx}, \textsc{Decide}\}$ matches some enactment E' reachable from E.

We will ignore Tx, because it does not affect the agent histories.

Consider the case where $s \xrightarrow{\textsc{Recv}} s'$. Then, the conditions of Recv must apply, and there is some unreceived message instance $\widehat{m} \in I_a$ such that $\mathsf{check}_r(\{\widehat{m}\}, H_a)$ holds. This means that $\widehat{m}$ was copied into I_a by Tx, and was already sent by another agent (and thus $\widehat{m} \in H_{\widehat{m}[\jmath_m]} \in s$ and $\langle \mathsf{sent}, m \rangle \in H_{\jmath_m} \in E$). Then, by Lemma 2, $e = \langle \mathsf{received}, m \rangle$ is viable in E, and there is a reachable enactment $E' = E + e$. By the consequent of Recv, $\widehat{m}$ is added to the history of $H_{\widehat{m}[\jmath_m]}$ in s', so $s' \equiv E'$.

Next, consider the case where $s \xrightarrow{\textsc{Decide}} s'$. Then, by the conditions of Decide, $\mathsf{valid}(T, H_a)$ holds. Because $\mathsf{check}_s(T, H_a)$ implies that T is internally consistent and compatible, we can without loss of generality consider the instances of T individually in any order; they will not introduce $\ulcorner\mathsf{out}\urcorner$ or $\ulcorner\mathsf{nil}\urcorner$ parameters that block each other. Then, by Lemma 1 for $\widehat{m}_i \in T$, $e_i = \langle \mathsf{sent}, m_i \rangle$ is viable in E, and there is a reachable enactment $E' = E + e_0 + ...e_n$. By the consequent of Decide, T is added to the agent history $H_{\widehat{m}[\jmath_m]}$ in s', so $s' \equiv E'$.

Thus, $s' \in \mathcal{S}_\mu$ simulates some enactment $E' \in \mathcal{E}_P$, and by induction all reachable states simulate some reachable enactment. □

Theorem 7. $\textsc{Decide}_2$ *is equivalent to* Decide.

Theorem 7 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.

Proof Sketch. DECIDE and DECIDE$_2$ both include the clauses $Q := \mathsf{enabled}(a, H_a)$ and $T := d(Q)$; however, DECIDE requires that $\mathsf{valid}(T, H_a)$, where DECIDE$_2$ requires $\mathsf{consistent}(T, T) \wedge \mathsf{compatible}_{\vec{o}}(T, T) \wedge \mathsf{compatible}_{\vec{n}}(T, T)$.

The definition of $\mathsf{enabled}(a, H_a)$ ensures that the resulting message forms are all sent by a, consistent with H_a, out and nil compatible with H_a, and derived from H_a. Additionally, the definition of decision maker d requires that the resulting instances share all of the above properties with a form in the enabled set Q. Thus, the only clauses of $\mathsf{check}_s(T, H_a)$ not covered by the above are those involving the whole set T, since they were only checked individually: $\mathsf{consistent}(T, T)$, $\mathsf{compatible}_{\vec{o}}(T, T)$, and $\mathsf{compatible}_{\vec{n}}(T, T)$—but these are precisely the clauses in DECIDE$_2$.

Thus DECIDE$_2$ is equivalent to DECIDE. □

Theorem 8. *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 8 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.

Proof Sketch. Suppose MAS μ implements protocol P, and that all sent message instances are received. Trivially, μ can reach the empty state s_0 which matches the empty enactment E_0. Further, suppose that μ can reach a state s that matches some enactment E, and that e is some event that is a viable extension of E.

Thus, e must be either $\langle \mathsf{sent}, m \rangle$ or $\langle \mathsf{received}, m, \rangle$ for some m in P. If e is a $\mathsf{received}$ event, then because e is viable m must have been sent in E, and therefore some instance $\widehat{m}$ was sent in s and therefore received by $\widehat{m}[\mathscr{r}_m]$ (that is, $\widehat{m} \in H_{\widehat{m}[\mathscr{r}]}$). Thus, $s' : s \overset{\text{RECV}}{\rightarrow} s'$ would simulate E'.

If e is a sent event, and $a = \widehat{m}[\mathscr{s}_m]$, then $s' : s \overset{\text{DECIDE}}{\rightarrow} s'$ would match E' if:

1. $\widehat{m}$ is enabled in s,

2. $\widehat{n} \in d(Q)$ for some decision maker d, where $\widehat{n}[\mu, \vartheta_m, \varkappa_m, \vec{i}_m] = \widehat{m}[\mu, \vartheta_m, \varkappa_m, \vec{i}_m]$,

3. and $\{n\}$ is a valid extension of H_a

We have Clause 1 from the fact that e is viable in E and $s \equiv E$; thus $\widehat{m} \in Q = \mathsf{enabled}(a, H_a)$. If we suppose there is some d such that $\widehat{n} \in d(Q)$, then we also have Clause 2. Clauses 1 and 2 together imply Clause 3, given the definitions of decision maker, enabled, and check_s. So we have satisfied the conditions and can exercise DECIDE, reaching $s' \equiv E'$.

Thus, $s' \equiv E'$ is reachable for all reachable E', and by induction any reachable enactment can be simulated. □

8.6 Discussion

Kiko is a natural bridge from business logic to communications as decisions. It envisages an agent as a serial decision maker. At any time, an agent can make and communicate a set of decisions (as reflected in the forms provided by the adapter) based on some evaluation of the state the world. Decision making conceptualized as such has a declarative favor and is suited to a rules-based expression in agent programming languages such as Jason.

An interesting direction would be 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. [20] 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.

Several 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 [83]; 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 sent a third instance i_3 (e.g., a shipper may need address from buyer and item from 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 [53] and PoT [48] programming models share Stellar's weaknesses compared to Kiko; 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. The influential end-to-end argument [137] advocates building applications over simple communication services, both for reasons of enabling application-level flexibility and performance. No protocol-based programming model in which protocols specify message ordering can work with an unordered communication service (the two being at cross purposes). In this chapter we focus on Kiko's programming abstractions and the expressions they enable. 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 a powerful feature that enables emitting a set of message instances (possibly from different protocols and to different agents) atomically. Scribble [175], a protocol-based programming model, is based on pairwise FIFO channels and ordered messages, and so differs greatly from Kiko. The Scribble-Java loan example [92] is similar to our *Approval* example, in that it involves confirming eligibility of the applicant before completing the loan. However, the Scribble implementation follows a strict linear flow orchestrated by an entity called ApplicationPortal, and handles only a single enactment.

In the current semantics, decision makers execute atomically with respect to the history. This makes the optimization of simply checking the internal compatibility of the emission set before emitting all its instances possible. 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 [2, 88]. 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 containerization of application components. Current microservices development approaches tend to avoid distributed database transactions in favor of loose coupling. 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.

Galland et al. [74] present SARL, an agent programming language that, like asynchronous microservices, supports communication using events in spaces that are akin to environments [166]. 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 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.

Chapter 9

Hercule – Norms over Unstructured Data

Current blockchain approaches for business contracts are based on smart contracts, software programs placed on a blockchain that are automatically executed to realize a contract. However, smart contracts lack flexibility and interfere with the autonomy of the parties concerned.

We propose Hercule, an approach for declaratively specifying blockchain applications in a manner that reflects business contracts. Hercule represents a contract via regulatory norms that capture the expectations between the involved parties. It computes the states of norms (hence, of contracts) from events in the blockchain. Hercule's novelty and significance lie in that it operationalizes declarative contracts over semistructured databases, the underlying representation for practical blockchain such as Hyperledger Fabric and Ethereum. Specifically, it exploits the map-reduce capabilities of such stores to compute norm states.

We demonstrate that our implementation over Hyperledger Fabric can process thousands of events per second, sufficient for many applications.

This chapter is adapted from the paper *Hercule: Representing and Reasoning About Norms as a Foundation for Declarative Contracts Over Blockchain*, published in IEEE Internet Computing, July-August 2021 [50]

9.1 Introduction

A contract conceptually underpins any application that involves two or more autonomous parties. Blockchain, by providing shared state and event ordering across trust boundaries, can enable shared and automated interpretation and adjudication of contracts.

Today's blockchains support *smart contracts*, software programs meant to automate contracts [160]. However, smart contracts prove unwieldy for assurance; in combination with the immutability of blockchain, they can yield disastrous outcomes, as the DAO incident [111] illustrates. In conceptual terms, smart contracts obstruct the autonomy and flexibility of the parties to a business transaction, which is simply unacceptable in real-world applications [155]. Ricardian contracts (https://iang.org/papers/ricardian_contract.html) tie a natural language description with a computational representation, possibly a smart contract. They face the fundamental problem of confusion about which version is correct: the one a human can read or the one the blockchain executes. Instead, we develop a declarative model to respect autonomy, improve readability, reduce bugs, and avoid computing the effects of each transaction on every validator node.

We build on recent work on a declarative representation for contracts based on *regulative norms* that provides high-level abstractions with a precise semantics [155]. "Norms" here are not mere descriptions of social behavior but carry prescriptive force and serve as elements of legal contracts [145, 165]. Norms have been formalized and mapped to a relational information model of events [41]. However, the relational approach has crucial limitations: (1) it requires a fixed structure of tables and columns and (2) it is often unavailable. For example, Hyperledger Fabric, a leading blockchain architecture, provides only a LevelDB based key-value store, and a CouchDB-based document store.

Therefore, to realize declarative contracts on blockchain platforms, it is important to show how a logic-based contract language can be implemented over a semistructured information model. We consider document stores wherein documents generally lack a schema, do not follow a predefined structure, and do not support the indexes required for efficient join queries common for relational settings. Instead, document stores process queries using map and reduce operations for data parallelism.

9.1.1 Contributions

We contribute Hercule, an approach to realize contracts as norms over event histories that can be securely shared between multiple agents using a blockchain. We show how to automatically generate map-reduce queries from norms.

We empirically evaluate our implementation and show that it is efficient enough to be

used in any situation where Hyperledger Fabric is practical, while providing the requisite expressiveness and flexibility.

9.2 Contracts and Smart Contracts

A *contract* is a document describing a legal relationship between multiple parties stating what each party may expect from the others under what conditions.

A *smart contract* is a programmatic description of a contract along with an architecture for automatic verification and enforcement. For example, a vending machine that automatically provides a product to anyone who pays the requisite amount is a smart contract [160].

Bitcoin, the first blockchain [119], represents financial transactions as simple, limited programs that verify the claims made against them. Subsequent blockchain platforms, such as Ethereum and Hyperledger Fabric, extend the concept of smart contracts to arbitrary executable programs, often called *distributed applications* or Dapps.

However, specifying contracts as arbitrary programs has major drawbacks. First, since the program is the source of truth regarding the meaning of a smart contract, there is no recourse against unintended behaviors, as the infamous DAO incident highlighted [155]. Second, a program omits the portions of a business relationship that cannot be automated, limiting the participants' ability to exercise their autonomy. In general, a party to a contract would exercise its discretion in acting, including deciding what terms to violate and penalties to risk.

Automation is infeasible wherever human insight is needed. For example, even if sharing a patient's health information is normally prohibited, a physician might share the information with a specialist during a medical emergency; an automated system would be unable to violate the prohibition. An auditor can decide afterward whether the physician was justified in violating the prohibition. Automation is also infeasible if external resources are involved. For example, a hospital may borrow supplies from another hospital promising to return them on request. To guarantee automatic return, a Dapp must control the supplies and not let them be used, which would defeat the purpose of borrowing.

9.2.1 Why Blockchain?

After years of blockchains being the hammer for every potentially decentralized nail, even those that least deserve it, a valid question is *why* blockchains are involved in the first place. Hercule implements a norm state engine primarily using CouchDB without any direct requirement for a blockchain, so why use one?

The first reason is that Hercule was intended to bridge the blockchain and multiagent normative system communities, to show how the existing procedural approaches to smart contracts are inadequate for capturing the complete social relationships usually involved in a legal contract. They focus far too heavily on those aspects which can be automated, when (as we endeavor to show) it is possible to capture much more complex and nuanced social relationships without relying on purely programmatic smart contracts. Thus we attempt to show the potential advantages of a norm-based declarative contract system, including flexibility, completeness, and performance; the events are logged to the chain, but all processing can be performed by interested parties offline using database queries.

Secondly, multiagent systems are decentralized—made up of agents representing multiple organizations, each with their own goals and interests—and therefore are not likely to trust each other with sole management of their mutual transaction data. Blockchains (especially development kits like Hyperledger Fabric) provide an off-the-shelf solution for tamper-proof database replication between mutually suspicious partners. Using a blockchain to record interaction events does not solve all points of contention between multiple parties, but it does create a fair system of record that they can all submit events to and observe, without fear that any side is tampering with the log (beyond submitting false events in the first place). This makes a blockchain a good basis for a neutral normative system, which can enable arbitration between multiple parties based on their recorded observations of events.

Most specifically, Hyperledger Fabric was chosen as a basis for development because of its performance, being a permissioned blockchain that can handle hundreds of transactions per second, and because of its integration with CouchDB that directly supports our advanced querying requirements. Any other blockchain (e.g. Ethereum) could be used for event replication but would be more expensive and slower for storing the event data, and would not directly support querying.

9.3 Compacts and Regulatory Norms

Hercule seeks to capture the essence of real-life contracts by specifying the legal relationships between the parties without curtailing their autonomy. We adopt the name *compact* for our formal notion to differentiate it from natural language contracts.

We summarize established terminology and semantics on norms [41]. A norm (instance) is a directed expectation between two agents, the *expector* and *expectee*. A norm is generally conditional, featuring an *antecedent* and a *consequent*, both events (possibly complex: logical expressions over simpler events). Three major norm types, *commitment*, *prohibition*, and *authorization*, are adequate for illustrating Hercule's language and reasoning.

To illustrate these norm types, we adopt a scenario from healthcare as our running example. Healthcare providers (HCPs) possess electronic health record (EHR) data about their patients that could be useful in research, but privacy regulations prevent them from sharing that data to outside organizations, unless specifically authorized by the patient. We motivate four main requirements on HCPs, which we formalize via norms: storing data, destroying data, enabling legitimate access, preserving confidentiality.

In a **commitment** norm, the expectee (*debtor*) commits to the expector (*creditor*) that if the antecedent occurs, then the consequent will occur. If the antecedent occurs but the consequent doesn't, then the commitment is violated; if the consequent occurs, it is satisfied. N_1 and N_2 below specify commitments in our EHR scenario compact.

N_1. (StoreData) An HCP commits to store a patient's data (consequent) after they visit (antecedent). The HCP is the debtor and the patient is the creditor; if a patient visits but the HCP does not store their data, the HCP violates the commitment.

N_2. (DestroyData) The HCP commits to destroying a patient's data (which must not be stored directly on the blockchain!) upon request by the patient.

In a **prohibition** norm, the expectee is prohibited by the expector from bringing about the consequent if the antecedent holds. If the antecedent and the consequent both occur, then the prohibition is violated; if the antecedent occurs but not the consequent, it is satisfied. N_3 in our EHR compact is a prohibition.

N_3. (Access) An HCP is prohibited from sharing patient data with an agent (consequent) unless the patient has granted access (antecedent).

In an **authorization** norm, the expectee authorizes the expector for the consequent when the antecedent occurs. If the antecedent occurs, then if the consequent cannot occur (e.g., due to incorrect access control settings), then the authorization is violated, else it is satisfied. N_4 in our EHR compact is an authorization.

N_4. (Confidentiality) A hospital may authorize a family member for access to a patient's health records (consequent) if the patient submits a release form (antecedent).

Any norm is instantiated (created) as specified events occur, but expires if its antecedent doesn't occur. For example, when the patient sends a specific directive identifying the data item and recipient, the authorization is instantiated for that patient, that item, and that recipient.

9.4 Declarative Contracts Using Norms

We adopt a simplified notion of the components and lifecycle of a compact [162]:

Provisions or the clauses describing what is expected of or offered to each party.

Operational model or how participants may move in reference to a compact and the outcomes of their moves.

Enforcement or how violations are handled.

Hercule implements compacts specified using norms to represent the provisions of a compact, which yield flexible operationalization and enforcement. Although Hercule does not provide tools for specifying the operational model or enforcement (outside of defining additional provisions), these components can be deployed as part of the implementation. This approach to implementing operational constraints is similar to defining the standard procedures of a legal jurisdiction in which a compact is interpreted; not every compact needs to specify the procedures used in the courtroom for adjudication, because they are assumed from the legal context. From this perspective, Hercule is a framework for implementing a jurisdiction or legal context in which compacts can be formed, leaving the exact operational and enforcement models largely out of scope for the compacts themselves.

We propose using regulative norms as a model for higher-level declarative specifications that capture intentions and relationships, instead of low-level procedures. Our model shares an intuition with Bitcoin, in which events are recorded to the blockchain but the interpretations are made by the clients. Specifically, Bitcoin does not use scripts to update balances stored on the blockchain; indeed it does not store user balances at all. Instead, it records individual transactions, and users compute their total balances by aggregating the relevant transactions. In the same way, we use the blockchain as a history of events that can be validated and preserved in themselves, with the interpretation supplied externally by a client. External interpretations are acceptable, because ultimately enforcement will be carried out by some external authority (which may be an automated service); this is the same as traditional contracts, where each party may have their own opinions about whether a violation has occurred, but the ultimate decision belongs to an arbiter.

Unlike most smart contract systems, which require all nodes to redundantly compute results during transaction processing, in Hercule only those clients that care about a given compact state are required to compute it. Also, Hercule separates the enactment of a compact from its enforcement, enabling flexible social enforcement methods and updates to the compact that reinterpret past events without forking the history.

Hercule builds on a concept of histories of events, related by logical propositions. Using this simple foundation, we build up to a normative framework with explicit support for

commitments, authorizations, and prohibitions as well as the flexibility to define custom norm types.

9.4.1 Architecture

Whereas Hercule can be applied in multiple settings, Figure 9.1 illustrates the architecture of a consent management system built with Hercule and deployed on top of a distributed ledger such as Hyperledger Fabric.

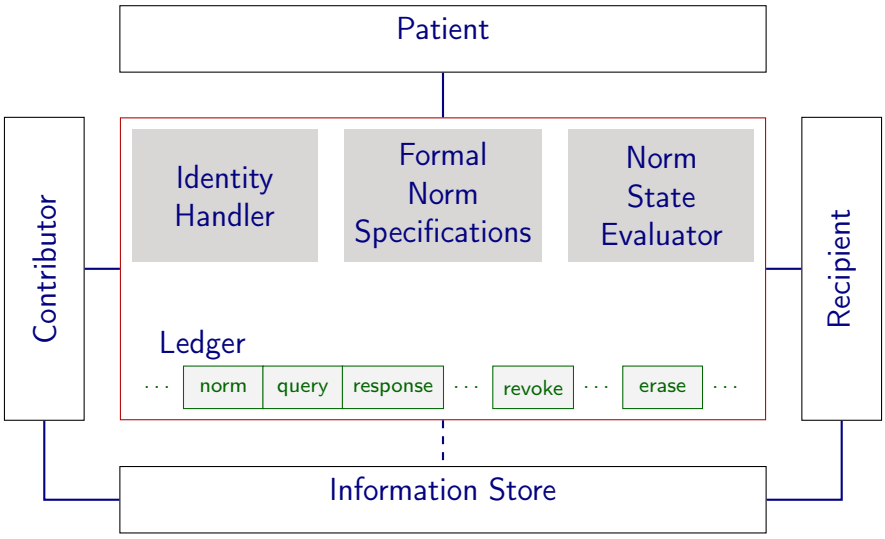


Figure 9.1: A high-level schematic description of a potential Consent Management System built on Hercule and how it interfaces with the main stakeholders and with external information stores. Here, a contributor is a patient's physician or laboratory or clinical trial firm; a recipient is a researcher. Hercule provides norm specifications and a norm state evaluator overlaid on a ledger that captures a consensus view of events. The identity handler maps a patient identity to identifiers in external information stores, and is not emphasized in this chapter.

9.4.2 Events

Events are things that happen in the real world, as subjectively observed by an agent.

Transactions are an example of events on a financial blockchain, in that they reflect changes in reality onto the blockchain. Once a transaction is recorded, the blockchain can be used to compute the balances of each account using only internal information.

Similarly, our events are immutable reflections of reality onto a blockchain. They are

contextualized to a specific observer and time, so it doesn't matter if reality changes and no longer matches the observation; what's important is that an agent claimed to observe the event. Once recorded, the events can be used to compute the state of the relevant norms.

Hercule represents events as JSON documents, as shown in Listing 9.1.

Listing 9.1: Example event object.

```
"Store": {
  "$by": "H",
  "item": "Diagnosis",
  "$time": 3
}
```

Listing 9.1 shows a single instance of the *Store* event. The agent that observed the event is recorded in the $by attribute, and the time the event was reported is recorded in the $time attribute. For simplicity, we represent time as an integer, but a timestamp could easily be used instead.

9.4.3 Histories

A history is a record of the information produced in an interaction between multiple parties. In Hercule, a history is represented as a single JSON document with a unique identifier in which attributes and values are only added and never modified. A single ledger may record multiple histories that are mutually unrelated. All events and norms are interpreted relative to a given history.

New information is added to a history through a transaction on the ledger when an event occurs. When updating a history, both the new event and copies of the new information it produces are added to the history as attributes. Although binding attributes on the history itself adds some overhead, doing so generalizes information away from the events that produce it, so that compact provisions can be written to depend on the information instead of the specific events.

In Hercule, a history is represented as a single JSON document. Listing 9.2 shows an example history, building on the previous *Store* event. The history has a unique ID, and comprises several events each represented by a subdocument. Each subdocument has attributes and values corresponding to information parameters bound by one of the events.

Listing 9.2: An example history document.

```
{
  "_id": "7c4f054dc8740698c93a9452e856cc87",
  "patient": "P",
  "physician": "D",
  "hospital": "H",
  "date": "2018-11-16",
  "item": "Diagnosis",
  "Visit": {
    "$by": "P",
    "date": "2018-11-16",
    "$time": 1,
  },
  "Record": {
    "$by": "D",
    "patient": "P",
    "item": "Diagnosis"
    "$time": 2,
  },
  "Store": {
    "$by": "H",
    "item": "Diagnosis"
    "$time": 3,
  }
}
```

The document representation matches the map-reduce querying facility of the CouchDB database underlying a Hyperledger Fabric ledger. Map and reduce operate only on individual documents in isolation, so any information that needs to be correlated for computing a query result must be stored together.

9.4.4 Norms

Norms are specified as a set of *states*, which are defined by logical formulas over events, usually involving multiple parties; since the formulas need not be exclusive, a norm instance can be in multiple states at once. The events in the formulas are variables that are true if a matching event has occurred in a history based on the name and parameters; extra parameters are ignored, but missing parameters disqualify an event from matching. A history *instantiates* a

norm in a given state (or combination of states) when it satisfies the corresponding formula(s). Generally, the first state of a norm is the *created* state, which determines whether a norm is relevant to a history. All other states of the norm are conjunctions involving the *created* state.

Listing 9.3 specifies the norms described in Section 9.3.

Listing 9.3: Privacy norm specifications in Hercule, with references to the compact clauses specified above

```
*(\ref{norm:StoreData} \textsf{StoreData})*
commitment StoreData(hospital->patient):
 created: patient.Visit{date}
 detached: physician.Record{patient, item}
 discharged: hospital.Store{item}

*(\ref{norm:DestroyData} \textsf{DestroyData})*
commitment DestroyData(hospital->patient):
 created: hospital.Store{item}
 detached: patient.RequestDeletion{item}
 discharged: hospital.Deleted{item}

*(\ref{norm:Access} \textsf{Access})*
authorization Access(patient->recipient, item):
 created:
  patient.GrantAccess{recipient,item} @ t
 detached:
  recipient.RequestAccess{item} @ t2 > t
  except
  patient.RevokeAccess{recipient,item} @ [t,t2]
 discharged:
  hospital.Shared{item, recipient} @ [t2, t2+10]

*(\ref{norm:Confidentiality} \textsf{Confidentiality})*
prohibition Confidentiality(patient->hospital):
 created: hospital.Store{patient, item}
 violated:
  hospital.Shared{item, recipient}
  except Access(patient->recipient, item):detached
```

Listing 9.3 specifies the four norms introduced in Section 9.3.

StoreData is a commitment with three specified states, *created*, *detached*, and *discharged*. Each state contains an *event* expression used to select matching enactments from the database. The event expression in the *created* state specifies an event named *Visit*, which is created by PATIENT and contains an attribute named `date`. Thus, any enactment containing a *Visit* event matching this description is considered as creating an instance of *StoreData*. Expressions can be composed from simpler ones using operators such as *and*, *or*, and *except*. The other norm specifications follow the same pattern.

Event expressions can be extended by time expressions, appended with the `@` symbol, which either label the time at which an event occurs (as in Access.created, which occurs at time `t`) or constrain it. The time expression may perform a simple comparison, as in $t_2 > t_1$, or restrict it to an interval, such as $[t_2, t_2 + 10]$.

We now explain how an instance of a norm may be created and progress through its states. An instance of *StoreData* is created when a patient visits the physician—when the patient's agent reports a *Visit* event containing the patient and the date of the visit. That instance is detached when the physician's agent reports the *Record* event with attributes describing both which patient the record is for, and what item is being recorded. And, that instance is discharged when the hospital permanently stores the item, as indicated when the hospital's agent reports a *Store* event with the item as an attribute.

Hercule provides special handling of commitments, prohibitions, and authorizations, which all have standard states and semantics, and map well to real-life contracts. For example, Hercule automatically derives the violated state of a commitment based on the detached and discharged components of its specification. Hercule could be readily enhanced with additional norm types.

9.5 Implementation

We adopted Hyperledger Fabric as our platform because it provides CouchDB as a document store that supports advanced queries.

Events are recorded by submitting them via transactions to the Hyperledger Fabric network. Fabric interprets transactions using *chaincode*, plugin programs that execute in containers on every node. The Hercule chaincode adds the submitted events to the appropriate history documents, making changes to the ledger and updating the underlying database to match. Hyperledger Fabric handles the consensus process by synchronizing and committing changes across all relevant nodes.

CouchDB supports map-reduce processing via *views* [161]; indexes or collections of derived data that can be queried like a normal collection. Views are computed by JavaScript functions

that are stored in special *design documents*. Hercule processes norm specifications to produce a design document for each norm, with one view for each state. When loaded into the database and queried, each view is applied to the data to produce a separate collection of matching norm instances. Hercule queries these collections to discover the current states of various norm instances and inform agent behavior, e.g., by detecting past violations by an HCP.

When triggered by a query, CouchDB applies the map and reduce functions of a view to all documents in the database (or incrementally to documents created or updated since the last query) to produce the view collection. Views are derived from blockchain data but do not modify it, so they do not need to be synchronized across all of the Fabric nodes. As such, a view needs to be computed not by all nodes in the network, but only those nodes that query it. The view collections are provided by the underlying CouchDB database independent of the ledger, so agents can query the norm states either by invoking a chaincode query or by directly querying the database.

Listing 9.4 shows the StoreData part of the design document generated by Hercule for the norm specifications in Listing 9.3.

Listing 9.4: StoreData design document (generated from Listing 9.3)

```
"StoreData": {
 "language": "javascript",
 "views": {
  ...,
  "violated": {
   "map":
    "function (doc) {
     // created
     doc.Visit
     // detached
     && (doc.Visit && doc.record
     // not discharged
     && !(doc.Visit && doc.Store
          || doc.Visit && doc.Record && doc.Store))
     && emit(doc)
    }"
  }
 }
}
```

As a design document, *StoreData* consists of two keys, language and views. Each view has

the name of the state as its key, and a single map function implementing the query logic. Each map function is applied to every document in the database, via the parameter doc, to produce one or more results via emit.

Each map function contains a single Boolean expression testing whether a given doc matches the specified norm state. For example, the created function in *StoreData* simply emits all documents that contain the *Visit* event. Hercule compiles the various states following Section 9.4.4. For example, the detached function for *StoreData* checks that both *Visit* and *Record* have occurred. Similarly, for the other states.

9.6 Conceptual Evaluation

We now compare our approach to smart contract systems.

9.6.1 Autonomy

Smart contracts operate automatically, precluding autonomy. This automation means smart contracts are inviolable, and is sometimes touted as an advantage, but in practice it means that many useful contracts cannot be adequately represented, as the examples in Section 9.2 show.

Because Hercule does not automatically enforce norms, agents are free to handle violations as they see fit. Moreover, an enforcement clause may itself be expressed as a norm. Sharing data without authorization may normally result in punishment, but a hospital may determine that it was necessary for properly responding to a medical emergency.

9.6.2 Enactment Scope

The Bitcoin blockchain implements a single history, in which each transaction is interpreted according to the same rules and can depend on the results of any previous transactions. Thus, the scope of enactment is broad and could include all the participants and events on the blockchain.

Ethereum includes a similar accounting system, but each Dapp has separate storage [69]: one Dapp may invoke another but cannot read or modify another's information directly. The scope of Dapp history could be restricted to selected participants and events or be a long-running open system that every agent may eventually participate in, as in ERC20 tokens, which are themselves full accounting systems [68].

As in Ethereum, Hercule has separate histories for each instance of a compact, which is sufficient for all compacts involving prespecified participants and events. Thus, the history

model reflects the mutual independence of the compacts; if information across histories is essential, the compacts should be specified as one. Of course, an agent participating in two compacts may copy information from one history to another—but that's purely its choice.

9.6.3 Operational Model

In Bitcoin's operational model each transaction depends on the outputs of prior transactions, and must satisfy the scripts of those prior transactions to be validated and added to the ledger [119].

In Ethereum's operational model, a Dapp when invoked may produce whatever new state its code would output, but the length of the computation is bounded by the amount of *gas* provided by the invoker [69]. If the computation does not complete before the gas runs out, no updates are recorded but the gas is consumed as fees for the miners.

Hercule does not specify an operational model. Events must be consistent with the history they are added to, but there are otherwise no constraints on the content of the events themselves or who may submit them. Norms can be written to identify histories as valid or invalid, but such would only provide warnings to agents (if they looked), not prevent them from making such changes.

Hercule is intended to be customized for specific domains, not operated as a standalone universal system. A domain-specific deployment could embed an operational model, e.g., requiring that only agents bound to a role in a history can submit events, or using agent signatures to demonstrate consent for changes to the norms.

9.7 Performance

Characterizing the performance of a Hyperledger Fabric network is challenging, because of the number of variables and nontrivial interactions between them. However, unlike smart contract systems, which perform validation and computation when a transaction is submitted prior to the consensus process, Hercule minimally checks each event for consistency during submission, with most of the computation occurring later when the state of a norm is queried. Furthermore, CouchDB can be used to replicate the data to separate nodes, so that querying can be performed independently of blockchain operations. Since event submission is trivial in Hercule, and norms can be queried against separate CouchDB instances, Hercule's performance is independent of Hyperledger Fabric's performance. This motivates our experimental design to measure throughput of norm queries—to verify that Hercule is fast enough to handle the maximum throughput of a Hyperledger Fabric system, and therefore suitable for practical

use.

We measured average throughput by creating a new database to prevent caching and data reuse, loading 200,000 randomly generated enactments, and probing the *changes per second* statistic while the database created a view after starting a query. The number of enactments was selected to achieve a reasonable minimum number of performance samples. Each enactment was generated with a uniformly distributed degree of progress—e.g., enactments in which only *Visit* occurred were more common than those that also included *Shared*.

Our experiment was run on a multiprocess, single-node CouchDB installation in Docker. All tests were performed on a laptop running Gentoo Linux, kernel version 4.19.27, with an Intel i7-6600U cpu, 16GB of DDR3 memory, and 1TB SSD. This setup provides a lower bound for performance, since CouchDB can be sharded to run map-reduce operations in parallel across a cluster.

Figure 9.2 shows our results grouped by norm and state. The measured throughput ranges from three to six thousand changes per second across up to eight parallel tasks. Note that *Confidentiality* does not have detached or discharged states.

Within a given norm later states generally have lower throughput than earlier ones because the later states usually depend on and thus subsume the logic of the earlier ones. Thus later states require more processing, approximately linear with the size of the logical formula (excepting possible short-circuiting).

Conversely, the later norms have *higher* throughput in a given state than *StoreData* because fewer enactments satisfy their conditions for creation; rejecting objects is the fastest way to process them.

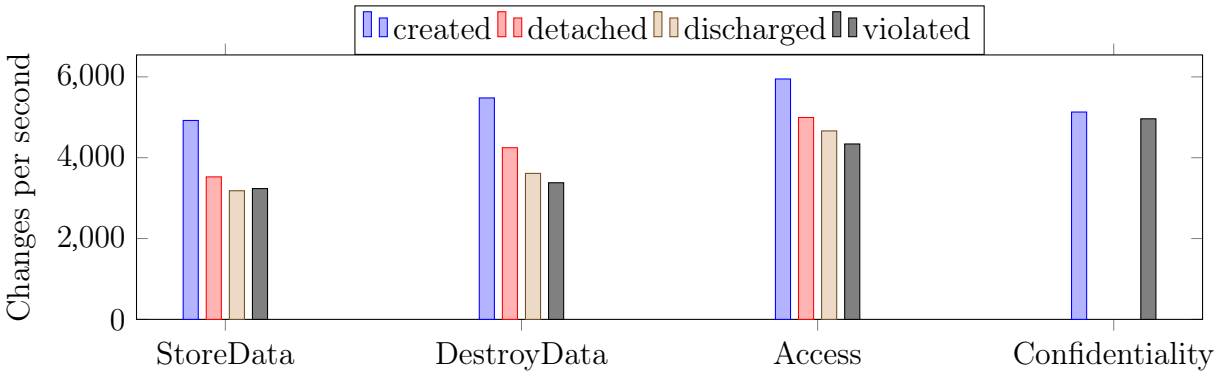


Figure 9.2: View construction performance of the example norms.

These results correspond to an approximate throughput of three thousand norm state changes per second. That could be three thousand norms applied to one updated history,

or a single norm applied to three thousand changed histories. Also, if more than one event is added to a history before a norm state is queried again, they will result in only a single change.

A recent performance analysis of Hyperledger Fabric showed that the maximum throughput of a Fabric network is around 400 transactions per second [120]. Thus, Hercule appears to have sufficient throughput for practical use. Even if Hercule were slower than Fabric on our test laptop, it scales much better than the Fabric consensus process because each node computes norm states only when queried, and the computation itself is a map-reduce operation that can be distributed across a CouchDB cluster.

9.8 Discussion

Hercule demonstrates a possible approach to representing contractual relationships in a way that captures social aspects that cannot be automated, and supports agent autonomy.

A side benefit of Hercule being modeled on norms is that it facilitates incorporating frameworks for intelligent agents that rely on norms to capture social and organizational reasoning capabilities [16] and compliance monitoring [59, 116]. Such approaches typically involve rule-based reasoning about events and often map from norms to cognitive models based on beliefs and goals.

Previous approaches based on logic indicate the viability of a declarative approach. However, they suffer from the common limitation of adopting the automatic enforcement pattern of smart contracts. Governatori et al. [80] discuss imperative and declarative smart contracts and their lifecycles, capabilities, and possible implementations, but do not provide a specific model or implementation. Purnell and Schwitter [132] demonstrate a logic program to implement a will on Ethereum. Unlike Hercule, their approach lacks generality and suffers from inefficiency by requiring a logic programming module to be run on all Ethereum nodes to process a transaction. de Kruijff and Weigand [61] suggest a commitment-based approach for smart contracts using RuleML as the specification language. They neither support other kinds of norms nor provide an implementation.

Because Hercule digitally represents events and norms to model business relationships, a closely related topic is *digital transformation*, which transforms existing business processes to use computational automation. Gurzhii et al. [86] present a systematic literature review of blockchain-based digital transformation. Most of the projects reviewed are basic applications of distributed ledgers: nonrepudiatable, auditable records for tracking provenance, ownership, etc. Hercule could be combined with such systems to add social reasoning and norms related to those recorded events. Battaiola et al. [21] describe a blockchain system that digitizes

a financial factoring market, which is like a payday loan for merchants (factors buy future invoices at a reduced price). The blockchain protects against selling the same invoice to multiple factors (double pledging), using cryptographic commitments. The work constructs an ad-hoc blockchain protocol for factoring from business requirements; Hercule could provide a syntax and mechanism for implementing such protocols.

Hercule can also be deployed without blockchains, since it is primarily built on top of CouchDB. In such a deployment, it is comparable to other norm processing engines such as Cupid[41], which implements norm reasoning over a relational database. However, without the blockchain Hercule would lose its desirable properties of cross-organizational verifiability and immutability. Also, several architectural tradeoffs were made to support Hyperledger Fabric and CouchDB's map-reduce model.

As Section 9.4.3 explains, Hercule maps each compact to a single history. Some blockchain uses do not readily map to a simple compact. For example, a token accounting system like Bitcoin's is better thought of as a specialized environment with a particular operational model in which unboundedly many transactions can be linked. One approach to representing such a system would be to treat a transaction as a single history with copies of the input transactions included, and an operational model that prevents the creation of internally inconsistent histories or histories referencing invalid predecessors. A future direction is to extend the Hercule chaincode interface to enable post-processing on map-reduce queries to handle joins across histories, and produce results agents can use more easily than raw norm states.

Although a typical contract has a lifecycle possibly involving multiple stages of negotiation such as formation, modification, and termination, Hercule focuses on computing the state of active contract instances and leaves the negotiation to external processes. However, it is straightforward to extend Hercule to support compact modification for a specific application. For example, the norm specification could be versioned using the blockchain, possibly with an approval process for all parties to consent to a new version. Then, anyone interested in the compact could evaluate queries according to the most recent version.

Although the design of Hercule supports flexibility in the operational models and enforcement schemes that can be implemented, leaving them out of scope for the core system places additional burdens on the platform implementors and the participating agents.

Chapter 10

Conclusions

This dissertation attempts to present and advance an approach for designing and implementing decentralized systems according to a realistic, information-based *local* perspective—one where each agent makes decisions based on what it knows, rather than some idealized global perspective of the system.

10.1 What has been accomplished?

- In PoT and Mandrake (chapters 3 and 4) we show how information protocols can be used to improve system reliability, first by overcoming correlation challenges, and second by enabling application-level recovery policies.
- In Atomicity and Refinement (chapters 5 and 6), we define new verifiable properties for protocols. Atomicity helps verify that subprotocols will not get stuck in an incomplete state, and refinement helps identify when protocols can substitute for each other. Both of these results proved useful, but also showed how difficult it can be to derive desireable properties and guarantees given the constraints of autonomy and asynchrony.
- Pippi (chapter 7) proposes a metaprotocol method for instantiating protocols – negotiating which protocol to use, and inviting agents to participate only when necessary.
- Kiko (chapter 8) gives a programming model for implementing agents that achieves cross-enactment and cross-protocol reasoning by operating on the entire set of enabled messages at once. The enablement-based approach reduces coupling and automates correlation.
- Finally, Hercule (chapter 9) layers normative reasoning on top of the information parameters each agent records in their interaction events. These event structures are

designed to be compatible with using a blockchain to record and share data reliably and verifiably between multiple organizations.

10.2 Now what?

This is only the beginning of information-based interaction-oriented research into the design and development of decentralized systems. Some possible further directions include:

- Extending the specification and programming models to general system composition. BSPL describes compositions of communication, which need not be limited to the agent level—both higher level organizations, and lower level software components can also be modeled in terms of communication.
- Abstracting the programming model to a general concept of "decisions" that would potentially encompass both internal changes (where currently only messages are considered) and supports emitting multiple messages per decision (e.g. simplifying the selection of accept vs reject into a single decision). Such a model would unify the internal and interaction portions of an agent implementation.
- Creating metaprotocols that specify *how* messages are to be transmitted, e.g. for specifying encryption protocols or message encoding formats.
- Generally handling challenges of malicious or uncooperative agents. We have not focused on incentive alignment, or the possibility of deliberately misleading information (e.g. lying about a forwarded parameter binding).
- Explore the possibilities of machine learning for more flexible interactions. How does ML integrate with discrete, declarative interaction specifications?

BIBLIOGRAPHY

[1] Gul Agha. Concurrent object-oriented programming. *Communications of the ACM (CACM)*, 33(9):125–141, September 1990. doi: 10.1145/83880.84528.

[2] Gul A. Agha. *Actors.* MIT Press, Cambridge, Massachusetts, 1986. ISBN 9780262010924.

[3] Natasha Alechina, Mehdi Dastani, and Brian Logan. Programming norm-aware agents. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1057–1064, Valencia, 2012. IFAAMAS.

[4] Mark Allman. On the generation and use of TCP acknowledgments. *ACM SIGCOMM Computer Communication Review*, 28(5):4–21, 1998.

[5] AMQP. Advanced Message Queuing Protocol, 2014. URL http://www.amqp.org. Accessed 2022-03-28.

[6] Davide Ancona, Daniela Briola, Angelo Ferrando, and Viviana Mascardi. Global protocols as first class entities for self-adaptive agents. In *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems*, pages 1019–1029, Istanbul, 2015. IFAAMAS.

[7] Davide Ancona, Angelo Ferrando, and Viviana Mascardi. Agents interoperability via conformance modulo mapping. In *Proceedings of the 19th Workshop on From Objects to Agents (WOA)*, number 2215 in CEUR, pages 109–115, Palermo, Italy, June 2018. CEUR-WS.org.

[8] Farhad Arbab. Puff, the magic protocol. In Gul Agha, Olivier Danvy, and José Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems*, pages 169–206. Springer, 2011.

[9] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors.* PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.

[10] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[11] Matteo Baldoni, Cristina Baroglio, Alberto Martelli, and Viviana Patti. A priori conformance verification for guaranteeing interoperability in open environments. In

Proceedings of the 4th International Conference on Service-Oriented Computing (ICSOC), volume 4294 of *Lecture Notes in Computer Science*, pages 339–351, Chicago, 2006. Springer.

[12] Matteo Baldoni, Guido Boella, Valerio Genovese, Roberto Grenna, and Leendert van der Torre. How to program organizations and roles in the JADE framework. In *Proceedings of the 6th German Conference on Multiagent System Technologies*, number 5244 in Lecture Notes in Computer Science, pages 25–36, Kaiserslautern, Germany, September 2008. Springer. doi: 10.1007/978-3-540-87805-6_4.

[13] Matteo Baldoni, Cristina Baroglio, Amit K. Chopra, Nirmit Desai, Viviana Patti, and Munindar P. Singh. Choice, interoperability, and conformance in interaction protocols and service choreographies. In *Proceedings of the 8th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 843–850, Budapest, May 2009. IFAAMAS. doi: 10.5555/1558109.1558129.

[14] Matteo Baldoni, Cristina Baroglio, and Federico Capuzzimati. A commitment-based infrastructure for programming socio-technical systems. *ACM Transactions on Internet Technologies*, 14(4):23:1–23:23, December 2014.

[15] Matteo Baldoni, Cristina Baroglio, Elisa Marengo, Viviana Patti, and Federico Capuzzimati. Engineering commitment-based business protocols with the 2CL methodology. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 28(4):519–557, July 2014.

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

[17] Matteo Baldoni, Cristina Baroglio, Federico Capuzzimati, and Roberto Micalizio. Type checking for protocol role enactments via commitments. *Autonomous Agents and Multi-Agent Systems*, 32(3):349–386, 2018.

[18] Matteo Baldoni, Cristina Baroglio, Federico Capuzzimati, and Roberto Micalizio. Process coordination with business artifacts and multiagent technologies. *Journal on Data Semantics*, 8(2):99–112, June 2019. doi: 10.1007/s13740-019-00100-8.

[19] Matteo Baldoni, Cristina Baroglio, Roberto Micalizio, and Stefano Tedeschi. JADE/-JaCaMo+2COMM: Programming agent interactions. In *Proceedings of the 18th International Conference on Advances in Practical Applications of Agents, Multi-Agent*

Systems, and Trustworthiness (PAAMS), volume 12092 of *Lecture Notes in Computer Science*, pages 388–391. Springer, October 2020. doi: 10.1007/978-3-030-49778-1_33.

[20] Matteo Baldoni, Cristina Baroglio, Roberto Micalizio, and Stefano Tedeschi. Robustness based on accountability in multiagent organizations. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 142–150, Online, May 2021. IFAAMAS. doi: 10.5555/3461017.3461040.

[21] Ettore Battaiola, Fabio Massacci, Chan Nam Ngo, and Pierantonia Sterlini. Blockchain-based invoice factoring: from business requirements to commitments. In *Proceedings of the Second Distributed Ledger Technology Workshop, DLT@ITASEC 2019, Pisa, Italy, February 12, 2019*, volume 2334 of *CEUR Workshop Proceedings*, pages 17–31. CEUR-WS.org, 2019. URL http://ceur-ws.org/Vol-2334/DLTpaper2.pdf.

[22] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley-Blackwell, 2007.

[23] Federico Bergenti, Giovanni Caire, Stefania Monica, and Agostino Poggi. The first twenty years of agent-based software development with JADE. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 34(2):36, 2020. doi: 10.1007/s10458-020-09460-z.

[24] Giovanni Bernardi and Matthew Hennessy. Mutually testing processes. *Logical Methods in Computer Science*, 11(2), 2015. doi: 10.2168/LMCS-11(2:1)2015.

[25] Ken Birman. Evolution of fault tolerance. In *SOSP History Day*, pages 7:1–7:32, New York, 2015. ACM.

[26] Kenneth P. Birman. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. *Operating Systems Review*, 28(1):11–21, 1994. doi: 10.1145/164853.164858.

[27] Kenneth P. Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991. doi: 10.1145/128738.128742.

[28] BitTorrent. uTorrent Transport Protocol, January 2017. URL http://bittorrent.org/beps/bep_0029.html.

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

[30] Olivier Boissier, Rafael H. Bordini, Jomi Fred Hübner, and Alessandro Ricci. Dimensions in programming multi-agent systems. *Knowledge Engineering Review (KER)*, 34:e2, January 2019. doi: 10.1017/S026988918000005X.

[31] Rafael H. Bordini and Jomi Fred Hübner. Semantics for the Jason variant of AgentSpeak (plan failure and some internal actions). In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI)*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 635–640, Lisbon, August 2010. IOS Press. doi: 10.3233/978-1-60750-606-5-635.

[32] Rafael H. Bordini, Jomi F. Hübner, and Michael J. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons, Chichester, United Kingdom, 2007. doi: 10.1002/9780470061848.

[33] Daniela Briola, Daniela Micucci, and Leonardo Mariani. A platform for P2P agent-based collaborative applications. *Software – Practice and Experience*, 49(3):549–558, 2019. doi: 10.1002/spe.2657.

[34] Nicholas Carriero and David Gelernter. Coordination languages and their significance. *Communications of the ACM (CACM)*, 35(2):97–107, February 1992. doi: 10.1145/129630.376083.

[35] Francesco Cesarini and Steve Vinoski. *Designing for scalability with Erlang/OTP: implement robust, fault-tolerant systems*. O'Reilly Media, Inc., 2016.

[36] David R. Cheriton and Dale Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*, pages 44–57, Asheville, North Carolina, December 1993. ACM Press. doi: 10.1145/168619.168623.

[37] Paula Daniela Chocron and Marco Schorlemmer. Vocabulary alignment in openly specified interactions. *Journal of Artificial Intelligence Research (JAIR)*, 68:69–107, May 2020. doi: 10.1613/jair.1.11497.

[38] Young B. Choi, Kathleen E. Capitan, Joshua S. Krause, and Meredith M. Streeper. Challenges associated with privacy in health care industry: Implementation of HIPAA and the security rules. *Journal of Medical Systems*, 30(1):57–64, February 2006.

[39] Amit K. Chopra and Munindar P. Singh. Contextualizing commitment protocols. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and*

Multiagent Systems, pages 1345–1352, Hakodate, Japan, May 2006. ACM Press. doi: 10.1145/1160633.1160884.

[40] Amit K. Chopra and Munindar P. Singh. Cupid: Commitments in relational algebra. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence*, pages 2052–2059, 2015.

[41] Amit K. Chopra and Munindar P. Singh. Custard: Computing norm states over information stores. In *Proceedings of the 15th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 1096–1105, Singapore, May 2016. IFAAMAS. doi: 10.5555/2936924.2937085.

[42] Amit K. Chopra and Munindar P. Singh. From social machines to social protocols: Software engineering foundations for sociotechnical systems. In *Proceedings of the 25th International World Wide Web Conference*, pages 903–914, Montréal, April 2016. ACM. doi: 10.1145/2872427.2883018.

[43] Amit K. Chopra, Samuel H. Christie V, and Munindar P. Singh. Splee: A declarative information-based language for multiagent interaction protocols. In *Proceedings of the 16th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 1054–1063, São Paulo, May 2017. IFAAMAS. doi: 10.5555/3091125.3091274.

[44] Amit K. Chopra, Samuel H. Christie V, and Munindar P. Singh. An evaluation of communication protocol languages for engineering multiagent systems. *Journal of Artificial Intelligence Research (JAIR)*, 69:1351–1393, December 2020. doi: 10.1613/jair.1.12212.

[45] Samuel H. Christie V and Amit K. Chopra. Fault tolerance in multiagent systems. In *International Workshop on Engineering Multi-Agent Systems*, pages 78–86. Springer, 2020.

[46] Samuel H. Christie V, Amit K. Chopra, and Munindar P. Singh. Multiagent protocol refinement. In *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. IFAAMAS, 2020. 258–266.

[47] Samuel H. Christie V, Amit K. Chopra, and Munindar P. Singh. Compositional correctness in multiagent interactions. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 1159–1167, Stockholm, July 2018. IFAAMAS. doi: 10.5555/3237383.3237868.

[48] Samuel H. Christie V, Daria Smirnova, Amit K. Chopra, and Munindar P. Singh. Protocols over Things: A decentralized programming model for the Internet of Things. *IEEE Computer*, 53(12):60–68, December 2020. doi: 10.1109/MC.2020.3023887.

[49] Samuel H. Christie V, Amit K. Chopra, and Munindar P. Singh. Bungie: Improving fault tolerance via extensible application-level protocols. *IEEE Computer*, 54(5):44–53, May 2021. doi: 10.1109/MC.2021.3052147.

[50] Samuel H. Christie V, Amit K. Chopra, and Munindar P. Singh. Hercule: Representing and reasoning about norms as a foundation for declarative contracts over blockchain. *IEEE Internet Computing (IC)*, 25(4):67–75, July 2021. doi: 10.1109/MIC.2021.3080982.

[51] Samuel H. Christie V, Amit K. Chopra, and Munindar P. Singh. Deserv: Decentralized serverless computing. In *Proceedings of the 19th IEEE International Conference on Web Services (ICWS)*, pages 51–60, Virtual, September 2021. IEEE Computer Society. doi: 10.1109/ICWS53863.2021.00020.

[52] Samuel H. Christie V, Amit K. Chopra, and Munindar P. Singh. Pippi: Practical protocol instantiation. In *Proceedings of the 21st International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 281–289, Auckland, May 2022. IFAAMAS. doi: 10.5555/3535850.3535883.

[53] Samuel H. Christie V, Amit K. Chopra, and Munindar P. Singh. Mandrake: Multiagent systems as a basis for programming fault-tolerant decentralized applications. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 36(1):16:1–16:30, April 2022. doi: 10.1007/s10458-021-09540-8.

[54] Andrei Ciortea, Olivier Boissier, Antoine Zimmermann, and Adina Magda Florea. Responsive decentralized composition of service mashups for the internet of things. In *Proceedings of the 6th International Conference on the Internet of Things (IoT)*, pages 53–61, Stuttgart, 2016. ACM. doi: 10.1145/2991561.2991573.

[55] David Clark. The network and the OS. In *SOSP History Day*, pages 11:1–11:19, Monterey, California, October 2015. ACM. doi: 10.1145/2830903.2830912.

[56] Benjamin Cox, J. D. Tygar, and Marvin Sirbu. NetBill security and transaction protocol. In *Proceedings of the 1st USENIX Workshop on Electronic Commerce*, pages 77–88, New York, 1995. USENIX.

[57] Edward Curry, Desmond Chambers, and Gerard Lyons. A JMS message transport protocol for the JADE platform. In *IEEE/WIC International Conference on Intelligent Agent Technology, 2003.*, pages 596–600. IEEE, 2003.

[58] Mehdi Dastani, Virginia Dignum, and Frank Dignum. Role-assignment in open agent societies. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 489–496, Melbourne, July 2003. ACM Press. doi: 10.1145/860575.860654.

[59] Mehdi Dastani, Paolo Torroni, and Neil Yorke-Smith. Monitoring norms: A multi-disciplinary perspective. *Knowledge Engineering Review*, 33:e25, December 2018. doi: 10.1017/S0269888918000267.

[60] Marc de Graauw. Nobody needs reliable messaging. https://www.infoq.com/articles/no-reliable-messaging/, June 2010.

[61] Joost T. de Kruijff and Hans Weigand. An introduction to commitment based smart contracts using ReactionRuleML. In Jaap Gordijn and Erik Proper, editors, *Proceedings of the 12th International Workshop on Value Modeling and Business Ontologies, VMBO 2018, Amsterdam, The Netherlands, February 26th - 27th, 2018*, volume 2239 of *CEUR Workshop Proceedings*, pages 149–157. CEUR-WS.org, 2018. URL http://ceur-ws.org/Vol-2239/article_16.pdf.

[62] Rocco De Nicola and Matthew Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984. doi: 10.1016/0304-3975(84)90113-0.

[63] Rocco De Nicola and Hernán C. Melgratti. Multiparty testing preorders. In *Trustworthy Global Computing - 10th International Symposium, TGC 2015, Madrid, Spain, August 31 - September 1, 2015 Revised Selected Papers*, volume 9533 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2015. doi: 10.1007/978-3-319-28766-9_2.

[64] Lavindra de Silva, Felipe Meneguzzi, and Brian Logan. BDI agent architectures: A survey. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 4914–4921. ijcai.org, 2020. doi: 10.24963/ijcai.2020/684.

[65] Nirmit Desai and Munindar P. Singh. On the enactability of business protocols. In *Proceedings of the 23rd Conference on Artificial Intelligence (AAAI)*, pages 1126–1131, Chicago, July 2008. AAAI Press.

[66] Nirmit Desai, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh. Interaction protocols as design abstractions for business processes. *IEEE Transactions on Software Engineering*, 31(12):1015–1027, December 2005. doi: 10.1109/TSE.2005.140.

[67] Ulrich Endriss, Nicolas Maudet, Fariba Sadri, and Francesca Toni. Protocol conformance for logic-based agents. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 679–684, Acapulco, Mexico, 2003. IJCAI.

[68] Ethereum. ERC-20 Token Standard, December 2020. https://ethereum.org/en/developers/docs/standards/tokens/erc-20/.

[69] Ethereum. Ethereum whitepaper, February 2021. https://ethereum.org/en/whitepaper/.

[70] Angelo Ferrando, Michael Winikoff, Stephen Cranefield, Frank Dignum, and Viviana Mascardi. On enactability of agent interaction protocols: Towards a unified approach. In *Proceedings of the 7th International Workshop on Engineering Multi-Agent Systems (EMAS)*, volume 12058 of *Lecture Notes in Computer Science*, pages 43–64, Montréal, May 2019. Springer. doi: 10.1007/978-3-030-51417-4_3.

[71] FIPA. FIPA interaction protocol specifications, 2003. FIPA: The Foundation for Intelligent Physical Agents, http://www.fipa.org/repository/ips.html.

[72] Michael Fisher, Viviana Mascardi, Kristin Yvonne Rozier, Bernd-Holger Schlingloff, Michael Winikoff, and Neil Yorke-Smith. Towards a framework for certification of reliable autonomous systems. *AAMAS*, 35(1):8, 2021. doi: 10.1007/s10458-020-09487-2.

[73] Nicoletta Fornara and Marco Colombetti. 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)*, pages 520–527, Melbourne, July 2003. ACM Press. doi: 10.1145/860575.860659.

[74] Stéphane Galland, Sebastian Rodriguez, and Nicolas Gaud. Run-time environment for the SARL agent-programming language: The example of the Janus platform. *Future Generation Computer Systems*, 107:1105–1115, June 2020. doi: 10.1016/j.future.2017.10.020.

[75] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Pearson, 2nd edition, 2008.

[76] Simon J. Gay. Subtyping supports safe session substitution. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 95–108. Springer, 2016. doi: 10.1007/978-3-319-30936-1_5.

[77] Scott N. Gerard and Munindar P. Singh. Formalizing and verifying protocol refinements. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 42(2):21:1–21:27, March 2013. doi: 10.1145/2438653.2438656. Appendix pages 1–7.

[78] Laura Giordano, Alberto Martelli, and Camilla Schwind. Specifying and verifying interaction protocols in a temporal action logic. *Journal of Applied Logic*, 5(2):214–234, 2007.

[79] Google. Protocol buffers. https://developers.google.com/protocol-buffers/, January 2022.

[80] Guido Governatori, Florian Idelberger, Zoran Milosevic, Régis Riveret, Giovanni Sartor, and Xiwei Xu. On legal contracts, imperative and declarative smart contracts, and blockchain systems. *Artificial Intelligence and Law*, 26(4):377–409, 2018. doi: 10.1007/s10506-018-9223-3.

[81] Miguel Escrivá Gregori, Javier Palanca Cámara, and Gustavo Aranda Bada. A jabber-based multi-agent system platform. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1282–1284, 2006.

[82] Roberto Grenna, Matteo Baldoni, Guido Boella, Leendert van der Torre, Mauro Dorni, Andrea Mugnaini, and Valerio Genovese. Adding organizations and roles as primitives to the jade framework. In Rafael Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors, *Programming Multi-Agent Systems*, number 08361 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. URL http://drops.dagstuhl.de/opus/volltexte/2008/1639.

[83] Akın Günay and Amit K. Chopra. Stellar: A programming model for developing protocol-compliant agents. In *Preproceedings of the 6th International Workshop on Engineering Multi-Agent Systems*, volume 11375 of *LNCS*, pages 117–136, Stockholm, 2018. Springer.

[84] Akın Günay, Michael Winikoff, and Pınar Yolum. Generating and ranking commitment

protocols. In *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems*, pages 1323–1324. IFAAMAS, 2013.

[85] Akın Günay, Michael Winikoff, and Pınar Yolum. Dynamically generated commitment protocols in open systems. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 29(2):192–229, March 2015. doi: 10.1007/s10458-014-9251-7.

[86] Anastasiia Gurzhii, A. K. M. Najmul Islam, A. K. M. Bahalul Haque, and Venkata Marella. Blockchain enabled digital transformation: A systematic literature review. *IEEE Access*, 10:79584–79605, 2022. doi: 10.1109/ACCESS.2022.3194004. URL https://doi.org/10.1109/ACCESS.2022.3194004.

[87] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[88] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 235–245, Stanford, 1973. William Kaufmann.

[89] HL7. Laboratory order conceptual specification, May 2013. URL https://wiki.hl7.org/index.php?title=Laboratory_Order_Conceptual_Specification.

[90] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 273–284, San Francisco, January 2008. ACM. doi: 10.1145/1328438.1328472.

[91] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM*, 63(1):9:1–9:67, March 2016. doi: 10.1145/2827695.

[92] Raymond Hu. Scribble loan application example, 2019. URL https://github.com/scribble/scribble-java/tree/master/scribble-demos/scrib/loan/src/loan. Accessed 2022-10-8.

[93] Michael N. Huhns, Nigel Jacobs, Tomasz Ksiezyk, Wei-Min Shen, Munindar P. Singh, and Philip E. Cannata. Enterprise information modeling and model integration in Carnot. In Charles J. Petrie, Jr., editor, *Enterprise Integration Modeling: Proceedings of the First International Conference*, pages 290–299. MIT Press, Hilton Head, South Carolina, October 1992. doi: 10.7551/mitpress/2768.003.0036.

[94] Health Level Seven International. Laboratory order conceptual specification, 2013. URL http://wiki.hl7.org/index.php?title=Laboratory_Order_Conceptual_Specification.

[95] Istio. What is Istio? https://istio.io/latest/docs/concepts/what-is-istio/, July 2020.

[96] ITU. Message sequence chart (MSC), April 2004. http://www.itu.int/ITU-T/2005-2008/com17/languages/Z120.pdf.

[97] Jana Iyengar and Martin Thomson. QUIC: A UDP-based multiplexed and secure transport. Technical report, Internet Engineering Task Force (IETF), Fremont, California, October 2020. Proposed standard; https://datatracker.ietf.org/doc/draft-ietf-quic-transport/.

[98] Henning Kagermann, Wolfgang Wahlster, , and Johannes Helbig. Recommendations for implementing the strategic initiative INDUSTRIE 4.0: Securing the future of German manufacturing industry, 2013.

[99] Warda El Kholy, Jamal Bentahar, Mohamed El-Menshawy, Hongyang Qu, and Rachida Dssouli. SMC4AC: A new symbolic model checker for intelligent agent communication. *Fundamenta Informaticae*, 152(3):223–271, 2017. doi: 10.3233/FI-2017-1519.

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

[101] Mark Klein and Chrysanthos Dellarocas. Exception handling in agent systems. In *Proceedings of the 3rd International Conference on Autonomous Agents*, pages 62–68, Seattle, May 1999. ACM. doi: 10.1145/301136.301164.

[102] Yoann Kubera, Philippe Mathieu, and Sébastien Picault. IODA: An interaction-oriented approach for multi-agent based simulations. *Autonomous Agents and Multi-Agent Systems*, 23(3):303–343, 2011.

[103] Joey Lam, Frank Guerin, Wamberto Vasconcelos, and Timothy J Norman. Building multi-agent systems for workflow enactment and exception handling. In *International Workshop on Coordination, Organizations, Institutions, and Norms in Agent Systems*, pages 53–69. Springer, 2009.

[104] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978. doi: 10.1145/359545.359563.

[105] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994. doi: 10.1145/197320.197383.

[106] Mark Little. Virtual panel: Microservices in practice. https://www.infoq.com/articles/microservices-in-practice/, February 2017.

[107] Nancy Lynch. Multilevel atomicity—a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

[108] Ashok U. Mallya and Munindar P. Singh. An algebra for commitment protocols. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 14(2):143–163, April 2007. doi: 10.1007/s10458-006-7232-1.

[109] Hamza Mazouzi, Amal El Fallah Seghrouchni, and Serge Haddad. Open protocol design for complex interactions in multi-agent systems. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 517–526, Bologna, July 2002. ACM Press. doi: 10.1145/544862.544866.

[110] Jarred McGinnis and David Robertson. Dynamic and distributed interaction protocols. In *Proceedings of the Workshop on Adaptive Agents and Multi-Agent Systems*, number 3394 in Lecture Notes in Computer Science, pages 167–184, Melbourne, July 2005. Springer. doi: 10.1007/978-3-540-32274-0_11.

[111] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M. Kim, and Marek Laskowski. Understanding a revolutionary and flawed grand experiment in blockchain: The DAO attack. *J. Cases Inf. Technol.*, 21(1):19–32, 2019. doi: 10.4018/JCIT.2019010102.

[112] Tim Miller and Peter McBurney. Propositional dynamic logic for reasoning about first-class agent interaction protocols. *Computational Intelligence*, 27(3):422–457, 2011.

[113] Tim Miller and Jarred McGinnis. Amongst first-class protocols. In *Proceedings of the 8th International Workshop on Engineering Societies in the Agents World (ESAW 2007)*, number 4995 in Lecture Notes in Computer Science, pages 208–223, Athens, 2008. Springer.

[114] Naftaly H. Minsky. The imposition of protocols over open distributed systems. *IEEE Transactions on Software Engineering*, 17(2):183–195, 1991. doi: 10.1109/32.67599.

[115] Naftaly H. Minsky and Takahiro Murata. On manageability and robustness of open multi-agent systems. In *Software Engineering for Multi-Agent Systems II, Research Issues and Practical Applications (SELMAS)*, volume 2940 of *Lecture Notes in Computer Science*, pages 189–206, Portland, Oregon, May 2003. Springer. doi: 10.1007/978-3-540-24625-1_11.

[116] Sanjay Modgil, Nir Oren, Noura Faci, Felipe Meneguzzi, Simon Miles, and Michael Luck. Monitoring compliance with e-contracts and norms. *Artificial Intelligence and Law*, 23(2):161–196, June 2015. doi: 10.1007/s10506-015-9167-9.

[117] Marco Montali, Diego Calvanese, and Giuseppe De Giacomo. Verification of data-aware commitment-based multiagent system. In *Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems*, pages 157–164, Paris, May 2014. IFAAMAS.

[118] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice architecture: aligning principles, practices, and culture*. O'Reilly Media, Inc., 2016.

[119] Satoshi Nakamoto. Bitcoin whitepaper, 2009. https://bitcoin.org/bitcoin.pdf.

[120] Qassim Nasir, Ilham A. Qasse, Manar Wasif Abu Talib, and Ali Bou Nassif. Performance analysis of Hyperledger Fabric platforms. *Security and Communication Networks*, 2018: 3976093:1–3976093:14, 2018. doi: 10.1155/2018/3976093.

[121] OASIS. MQTT 3.1.1 specification document, October 2014. OASIS Standard previously known as the Message Queuing and Telemetry Transport; http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf.

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

[123] OMG. Business Process Model and Notation (BPMN), version 2.0.2, January 2014. URL https://www.omg.org/spec/BPMN/. Object Management Group.

[124] Andrea Omicini and Franco Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, September 1999.

[125] Lin Padgham and Michael Winikoff. Prometheus: A practical agent-oriented methodology. In Brian Henderson-Sellers and Paolo Giorgini, editors, *Agent-Oriented Methodologies*, chapter 5, pages 107–135. Idea Group, Hershey, PA, 2005.

[126] Frank Pallas, Philip Raschke, and David Bermbach. Fog computing as privacy enabler. *IEEE Internet Computing*, 24(04):15–21, jul 2020.

[127] David Michael Ritchie Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science, 5th GI-Conference*, pages 167–183, Karlsruhe, Germany, 1981. Springer. doi: 10.1007/BFb0017309.

[128] Chris Peltz. Web service orchestration and choreography. *IEEE Computer*, 36(10): 46–52, October 2003.

[129] Jeremy Pitt, Julia Schaumeier, and Alexander Artikis. Axiomatization of socio-economic principles for self-organizing institutions: Concepts, experiments and challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 7(4):39:1–39:39, December 2012. doi: 10.1145/2382570.2382575.

[130] Eric Platon et al. *Modeling exception management in multi-agent systems.* PhD thesis, UNIVERSITÉ PARIS 6, 2007.

[131] Jon Postel. User datagram protocol. *RFC*, 768:1–3, 1980. doi: 10.17487/RFC0768.

[132] Kevin Purnell and Rolf Schwitter. Towards declarative smart contracts. In *Symposium on Distributed Ledger Technology (4th: 2019)*. Griffith University, 2019.

[133] Alessandro Ricci, Michele Piunti, Mirko Viroli, and Andrea Omicini. Environment programming in CArtAgO. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *Multi-Agent Programming, Languages, Tools and Applications*, chapter 8, pages 259–288. Springer, Dordrecht, Netherlands, 2009. doi: 10.1007/978-0-387-89299-3_8.

[134] Alessandro Ricci, Andrei Ciortea, Simon Mayer, Olivier Boissier, Rafael H. Bordini, and Jomi Fred Hübner. Engineering scalable distributed environments and organizations for MAS. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 790–798, Montréal, May 2019. IFAAMAS. doi: 10.5555/3306127.3331770.

[135] Alessandro Ricci, Angelo Croatti, Rafael H. Bordini, Jomi Fred Hübner, and Olivier Boissier. Exploiting simulation for MAS development and execution - the JaCaMo-Sim

approach. In *Proceedings of the 8th International Workshop on Engineering Multi-Agent Systems (EMAS)*, volume 12589 of *Lecture Notes in Computer Science*, pages 42–60, Auckland, May 2020. Springer. doi: 10.1007/978-3-030-66534-0_3.

[136] Vladimir Rocha and Anarosa Alves Franco Brandão. A scalable multiagent architecture for monitoring IoT devices. *Journal of Network and Computer Applications*, 139:1–14, August 2019. doi: 10.1016/j.jnca.2019.04.017.

[137] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984. doi: 10.1145/357401.357402.

[138] Devavrat Shah. Gossip algorithms. *Found. Trends Netw.*, 3(1):1–125, 2009. doi: 10.1561/1300000014.

[139] Sea Shalunov, Greg Hazel, Janardhan Iyengar, and Mirja Kuehlewind. RFC 6817: Low extra delay background transport (LEDBAT), 2012. URL https://datatracker.ietf.org/doc/html/rfc6817.

[140] Zach Shelby, Klaus Hartke, and Carsten Bormann. The Constrained Application Protocol (CoAP). Technical Report RFC 7252, Internet Engineering Task Force (IETF), Fremont, California, June 2014. Proposed standard; https://tools.ietf.org/html/rfc7252.

[141] Sabrina Sicari, Alessandra Rizzardi, and Alberto Coen-Porisini. Smart transport and logistics: A Node-RED implementation. *Internet Technology Letters*, 2(2):e88, 2019. doi: 10.1002/itl2.88.

[142] Munindar P. Singh. Toward interaction-oriented programming. TR 96-15, Department of Computer Science, North Carolina State University, Raleigh, May 1996. URL https://www.csc.ncsu.edu/faculty/mpsingh/papers/mas/iop.pdf.

[143] Munindar P. Singh. 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)*, number 1365 in Lecture Notes in Computer Science, pages 93–106, Providence, Rhode Island, 1998. Springer. doi: 10.1007/BFb0026752.

[144] Munindar P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):40–47, December 1998. doi: 10.1109/2.735849.

[145] Munindar P. Singh. An ontology for commitments in multiagent systems: Toward a unification of normative concepts. *Artificial Intelligence and Law*, 7(1):97–113, March 1999. doi: 10.1023/A:1008319631231.

[146] Munindar P. Singh. Distributed enactment of multiagent workflows: Temporal logic for web service composition. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 907–914, Melbourne, July 2003. ACM Press. doi: 10.1145/860575.860721.

[147] Munindar P. Singh. Information-driven interaction-oriented programming: BSPL, the Blindingly Simple Protocol Language. In *Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 491–498, Taipei, May 2011. IFAAMAS. doi: 10.5555/2031678.2031687.

[148] Munindar P. Singh. LoST: Local State Transfer—An architectural style for the distributed enactment of business protocols. In *Proceedings of the 9th IEEE International Conference on Web Services (ICWS)*, pages 57–64, Washington, DC, July 2011. IEEE Computer Society. doi: 10.1109/ICWS.2011.48.

[149] Munindar P. Singh. Semantics and verification of information-based protocols. In *Proceedings of the 11th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 1149–1156, Valencia, Spain, June 2012. IFAAMAS. doi: 10.5555/2343776.2343861.

[150] Munindar P. Singh. Norms as a basis for governing sociotechnical systems. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 5(1):21:1–21:23, December 2013. doi: 10.1145/2542182.2542203.

[151] Munindar P. Singh. Bliss: Specifying declarative service protocols. In *Proceedings of the 11th IEEE International Conference on Services Computing (SCC)*, pages 235–242, Anchorage, Alaska, June 2014. IEEE Computer Society. doi: 10.1109/SCC.2014.39.

[152] Munindar P. Singh and Amit K. Chopra. Correctness properties for multiagent systems. In *Proceedings of the 6th AAMAS Workshop on Declarative Agent Languages and Technologies (DALT 2009)*, number 5948 in Lecture Notes in Artificial Intelligence, pages 192–207, Budapest, 2010. Springer. doi: 10.1007/978-3-642-11355-0_12.

[153] Munindar P. Singh and Amit K. Chopra. The Internet of Things and multiagent systems: Decentralized intelligence in distributed computing. In *Proceedings of the 37th IEEE*

International Conference on Distributed Computing Systems (ICDCS), pages 1738–1747, Atlanta, June 2017. IEEE. doi: 10.1109/ICDCS.2017.304. Blue Sky Thinking Track.

[154] Munindar P. Singh and Amit K. Chopra. Clouseau: Generating communication protocols from commitments. In *Proceedings of the 34th Conference on Artificial Intelligence (AAAI)*, pages 7244–7252, New York, February 2020. AAAI Press. doi: 10.1609/aaai.v34i05.6215.

[155] Munindar P. Singh and Amit K. Chopra. Computational governance and violable contracts for blockchain applications. *IEEE Computer*, 53(1):53–62, January 2020. doi: 10.1109/MC.2019.2947372.

[156] Munindar P. Singh and Samuel H. Christie V. Tango: Declarative semantics for multiagent communication protocols. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 391–397, Online, August 2021. IJCAI. doi: 10.24963/ijcai.2021/55.

[157] Munindar P. Singh, Amit K. Chopra, Nirmit V. Desai, and Ashok U. Mallya. Protocols for processes: Programming in the large for open systems. In *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 73–83, Vancouver, October 2004. ACM. doi: 10.1145/1052883.1052893. *SIGPLAN Notices*, volume 39, number 12, December 2004, pages 73–83.

[158] Benjamin Smith. 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/, April 2021.

[159] Reid G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, 1980.

[160] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), September 1997. doi: 10.5210/fm.v2i9.548.

[161] The Apache Software Foundation. CouchDB, December 2018. http://couchdb.apache.org/.

[162] Javier Vázquez-Salceda, Virginia Dignum, and Frank Dignum. Organizing multiagent systems. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 11(3): 307–360, 2005. doi: 10.1007/s10458-005-1673-9.

[163] Mahadevan Venkatraman and Munindar P. Singh. Verifying compliance with commitment protocols: Enabling open Web-based multiagent systems. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 2(3):217–236, September 1999. doi: 10.1023/A:1010056221226.

[164] Benjamin Vitteau and Marc-Philippe Huget. Modularity in interaction protocols. In Frank Dignum, editor, *Advances in Agent Communication*, volume 2922 of *Lecture Notes in Computer Science*, pages 291–309, Berlin, 2004. Springer.

[165] Georg Henrik Von Wright. *Norm and Action: A Logical Enquiry*. International Library of Philosophy and Scientific Method. Humanities Press, New York, 1963.

[166] Danny Weyns, Andrea Omicini, and James Odell. Environment as a first class abstraction in multiagent systems. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 14(1):5–30, February 2007. doi: 10.1007/s10458-006-0012-0.

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

[168] Michael Winikoff. Implementing commitment-based interactions. In *Proceedings of the 6th International Conference on Autonomous Agents and Multiagent Systems*, pages 1–8, 2007.

[169] Michael Winikoff, Wei Liu, and James Harland. Enhancing commitment machines. In *Proceedings of the 2nd International Workshop on Declarative Agent Languages and Technologies (DALT)*, volume 3476 of *LNAI*, pages 198–220, Berlin, 2005. Springer-Verlag.

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

[171] WS-CDL. Web Services Choreography Description Language, version 1.0, November 2005. http://www.w3.org/TR/ws-cdl-10/.

[172] Pınar Yolum and Munindar P. Singh. 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)*, pages 527–534, Bologna, July 2002. ACM Press. doi: 10.1145/544862.544867.

[173] Pınar Yolum and Munindar P. Singh. Commitment machines. In *Proceedings of the 8th International Workshop on Agent Theories, Architectures, and Languages (ATAL 2001)*, volume 2333 of *LNAI*, pages 235–247, Seattle, 2002. Springer.

[174] Pınar Yolum and Munindar P. Singh. Engineering self-organizing referral networks for trustworthy service selection. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 35(3):396–407, May 2005. doi: 10.1109/TSMCA.2005.846401.

[175] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble protocol language. In *Proceedings of the 8th International Symposium on Trustworthy Global Computing (TGC), Revised Selected Papers*, number 8358 in Lecture Notes in Computer Science, pages 22–41, Buenos Aires, August 2013. Springer. doi: 10.1007/978-3-319-05119-2_3.