Supple: Multiagent Communication Protocols with Causal Types

Engineering Multiagent Systems Track

Paper # 187

ABSTRACT

A (communication) protocol captures how agents interoperate by specifying the messages they exchange. In particular, since the information content of messages characterizes the interactions a protocol specifies, message types can improve interoperation by strengthening the specification of what each agent may legitimately expect from another agent. In addition, in implementations, typing information can enable improved verification of agents.

We introduce Supple, a protocol specification language that expresses message schemas with typed parameters. Supple enables definition of *causal types* for parameters that constrain how other parameters are computed in a protocol enactment. We give the formal semantics of Supple; characterize the liveness and safety of Supple specifications; and provide decision procedures for them.

KEYWORDS

Communication protocols; Agent communication

1 INTRODUCTION

The MAS Setting Our setting is a decentralized multiagent system (MAS) of autonomous and heterogeneous agents communicating via asynchronous messaging. Each agent acts on behalf of some real-world principal. An agent would be independently implemented or configured by its principal—without necessarily knowing the details of the other agents. Successful interoperation in such a setting requires precise specification of each agent's expectations—so the agents can comply with them.

In a decentralized MAS, a *protocol* enables interoperation between agents by specifying their mutual expectations separately from their internal decision making. That is, a protocol yields design requirements on how the agents exchange information. Major applications in finance and healthcare specify protocols. For example, the international healthcare standard, HL7 [12], specifies protocols to support interactions among healthcare stakeholders. Financial protocols as well, e.g., for syndicated loans [14], involve complex interactions and information models.

Problem For simplicity, we adopt an insurance auditing scenario as our running example. Auditing protocols are essential in many real world applications. In healthcare, HIPAA (US) and NHS (UK) specify auditing protocols involving several stakeholders. In our scenario, the agents involved are a subscriber S, insurer I, and auditor A. S buys policies from I which it uses to make claims. A requests reports from an insurer for auditing, which list the claims that were paid out but for less than the initial claimed amount.

Proc. of the 18th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2019), N. Agmon, M. E. Taylor, E. Elkind, M. Veloso (eds.), May 2019, Montreal, Canada

https://doi.org/doi

A protocol language must deal with the following interoperation failures. One, liveness failure: the insurer may lack the information needed to compute a report because it may have only a partial view of all the information communicated. Two, integrity failure: if I can compute the report, it should contain all and only the correct results. Three, safety failure: agents must not face a race situation where each attempts to produce authoritative results, e.g., if both I and S are enabled to produce results of the same request.

Our *problem* is how can a protocol language support statically verifying avoidance of the above failures? And, how can we relate the structure of communication in a MAS to the structure of information exchanged between agents.

Approach Our proposed language, Supple, adopts a declarative framework to ensure that protocols can be flexibly enacted in a decentralized and asynchronous manner [22]. An underlying intuition is that communication both produces and is constrained by information. Supple's main idea is to use typed parameters as first-class language elements for dynamically specifying types of the information to be exchanged at run time. In particular, it introduces *causal types* for parameters that constrain how other parameters are computed in a protocol enactment.

A causal type specifies the type of information on which a computation is performed, and the type of information the computation yields. For example, a causal type may specify a computation that can be applied to the claimed and paid amounts of an insurance policy to compute the total payable amount of the policy. The auditor may send a request for a report including a function of the above causal type to express the selection criteria for the claims to be considered in the report and the type of information to be reported for those claims.

Supple yields three main benefits. First, agents may enact the same protocol instantiated with different functions to produce different report types. Second, agents can be verified against those types. Third, protocols can be verified taking parameter types into account. For example, if a protocol enables the auditor to send a function of the above causal type to the insurer, it must enable the insurer to acquire the necessary information to apply the function during the enactment. Otherwise, the protocol is incorrect. Parameter types enable static checking of protocols to prevent such errors even though a function may be formulated at run time.

Literature Existing protocol specification approaches inadequately specify what information may be exchanged through a protocol, leading to ad hoc methods to ensure correctness. Several approaches capture control flow: UML sequence diagrams [4], session types [18], WS-CDL [25], RASA [16]) and 2CL [3], but none captures information flow. FIPA [10] adds ontology annotations to messages but doesn't relate information across messages. Gunay et al. [11], assign meanings to messages but don't specify the message content precisely upon which the meanings rely. Information flow approaches, [22, 24] model content weakly, as uninterpreted values.

Chopra et al.'s [6] Splee incorporates queries into protocols, e.g., to specify that the winner in an English auction protocol is the highest bidder, but considers them only as untyped descriptive annotations, which are not part of Splee's formalization. Supple shares some motivations with business artifacts and data-centric models [9, 19], which combine information abstractions with process. Montali et al. [17] address verification of commitment-based MAS with queries. However these works typically do not address decentralization, treating a multiagent system as a single machine.

Baldoni et al. [2] formalize agent types to check them for compatibility with commitment protocols. Damiani et al. [8] formalize type soundness of MAS in terms of agents and artifacts. Kakas et al. [13] support agent reasoning for communication. Supple focuses on the complementary concern of specifying a protocol independent of agent reasoning as a way to guide the development of interoperable agents. In decentralized MAS, a protocol must handle loosely coupled, asynchronous communication whereas the above approaches require synchronous communication via a shared infrastructure.

Contributions Supple provides (1) a language for enriched protocol specification; (2) a novel definition of safe and live protocols; and (3) associated verification algorithms. Supple's novelty lies in introducing type abstractions for information in interaction whereas existing work does not consider information modeling of interactions. Supple's significance lies in advancing interaction-orientation: by specifying interactions in more detail, we can verify protocols and agents without relying upon internal details. Exposing implementations is anathema to engineering practice in any setting and especially inapplicable in decentralized MAS.

2 BACKGROUND

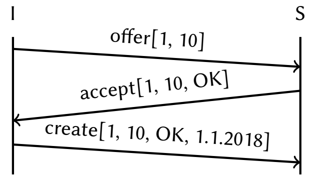
We introduce information-based protocols by example. BSPL [20] specifies protocols declaratively, constraining message via causality and integrity, and omitting control structures (e.g., sequencing, branching, iteration). Listing 1 shows a BSPL protocol, *CreatePolicy*, to create an insurance policy between an insurer (I) and a subscriber (S). It declares three public parameters, *pID* (the protocol's key), *premium*, and *date*. Every complete enactment of the protocol must produce bindings for all of these parameters. The ⌜out⌝ adornment signifies that the bindings are produced by this protocol.

Listing 1: A protocol to create an insurance policy.

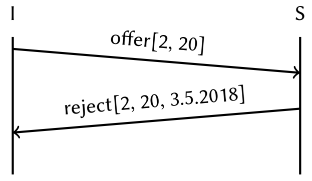
```
CreatePolicy {
 role I, S
 parameter out pID key, out premium, out date
 I ↦ S: offer[out pID, out premium]
 S ↦ I: accept[in pID, in premium, out agreed]
 S ↦ I: reject[in pID, in premium, out date]
 I ↦ S: create[in pID, in premium, in agreed, out date]
}
```

CreatePolicy declares four message schemas. The ordering of message schemas in a protocol listing is irrelevant to their operational ordering in an enactment. The message *offer* is directed from the insurer to the subscriber to offer a new policy. Its two parameters are adorned ⌜out⌝, meaning that their bindings must be produced by the insurer when emitting an *offer* message. In *accept*, the parameters *pID* and *premium* are adorned ⌜in⌝, meaning that the subscriber must know the bindings of these parameters from prior messages (e.g., *offer*) to emit *accept*. Each protocol and message must have a key. The key parameters of a protocol are also key parameters of its messages. Bindings of all parameters are unique for each binding of the key parameters. Note that *date* is adorned ⌜out⌝ in *reject* and *create*. Hence, in any enactment of *CreatePolicy*, either only *reject* or only *create* can be emitted to ensure integrity.

Figure 1 shows two valid enactments of *CreatePolicy* with bindings of the message parameters. In Figure 1a, the subscriber accepts the insurer's offer and in Figure 1b, the subscriber rejects the offer.



(a) S accepts offer.



(b) S rejects offer.

Figure 1: Valid enactments of *CreatePolicy*.

Listing 2 illustrates protocol composition. *ReportPolicy* references *CreatePolicy* via its public parameters, and adds two messages. Here, *rID* identifies report instances, and forms a composite key with *pID* to associate multiple policies with a report. The auditor's request for a report is captured as *request*, where *amount* indicates the minimum premium amount the insurer should report. BSPL enforces no constraints besides integrity on parameter bindings. Hence, although *amount* is meant as a criterion to filter the reported policies, *ReportPolicy* does not represent this intuition. Hence, *ReportPolicy*'s enactments may include a *report* for any known *pID*.

Listing 2: Reporting insurance policies to an auditor.

```
ReportPolicy {
 role I, A
 parameter out rID key, out pID key, out amount, out
    info
 CreatePolicy(I, S, out pID key, out premium, out date)
 A ↦ I: request[out rID, out amount]
 I ↦ A: report[in rID, in pID, in amount, in premium,
    out info]
}
```

3 TYPES IN SUPPLE

What we need is a protocol language that not only tackles decentralized MAS but provides language support for information dependencies. Specifically, BSPL supports ⌜out⌝–⌜in⌝ causal dependencies but ignores ⌜in⌝–⌜out⌝ dependencies needed to correctly link the requested minimum premium amount in *request* with the reported information in *report* as we illustrate in Listing 2.

3.1 Atomic and Composite Causal Types

In Supple, each parameter with its adornment constitutes a distinct *causal type* and may have a binding conforming to its type. For example, in Listing 1, we interpret *out pID* and *in pID*, and *out premium* and *in premium* as distinct causal types.

Whereas a data type in traditional languages defines how a value may be interpreted by an application (e.g., *premium* is money in Euros), a type in Supple defines the flow of information in a protocol.

We disregard data types since they are well-known (assume they are strings throughout) and reserve the term "type" for causal types.

In Supple, the possible bindings of a type, for which the adornment component is ⌜out⌝, include any possible binding (i.e., any string). For instance, the parameter *premium* can be bound to any string by the insurer when emitting *offer*, since the type of the parameter is *out premium*. On the other hand, the possible bindings of a type, for which the adornment component is ⌜in⌝, depend on the existing bindings of the types, which share the same parameter component, in all enactments. For instance, the possible bindings of parameter *premium* in *accept* depend on the existing bindings of the *premium* parameters in all messages.

A type is *atomic* when it consists of a single pair of adornment and parameter components. A type may be a *composite* of multiple atomic types. Protocols (and messages) have composite types over their parameters' types. For example, *offer*'s type in Listing 1 is the composite type (*out pID*, *out premium*) with respect to the types of its parameters (i.e., *pID* and *premium*). Note that an atomic type is a special case of a composite type. That is *out premium* and *(out premium)* correspond to the same type. However, we drop the parentheses from the notation of a composite type when it is defined over a single atomic type.

3.2 Causal Type of a Computation

Our main focus is the notion of the causal type of a computation, which we denote as $\mathcal{T} : \mathcal{I} \rightarrow \mathcal{O}$, where $\mathcal{T}$ is a name, and $\mathcal{I}$ and $\mathcal{O}$ are composite types. $\mathcal{I}$ defines the type of information on which the computation is performed and $\mathcal{O}$ defines the type of information the computation yields. For instance, a computation whose type is *c : (in pID, in premium) → (out c.pID, out c.premium, out c.info)* is performed on the information, whose type is *(in pID, in premium)*, yielding to the information, whose type is *(out c.pID, out c.premium, out c.info)*. Note that *in pID* and *out c.pID* (and similarly *in premium* and *out c.premium*) are different types.

The causal type of a computation enforces certain constraints on the type of information the computation yields with respect to the type of information on which the computation is applied.

Constraint 1: All the atomic types that form the composite type $\mathcal{I}$ must have ⌜in⌝ as their adornment component. The reason is, $\mathcal{I}$ defines the type of information on which the the computation is performed. Hence, this type of information must be known to the agent when the computation is performed, which is signified by the ⌜in⌝ adornment.

Constraint 2: If there is an atomic type t in $\mathcal{I}$, whose parameter component is p (e.g., *pID*), and there is an atomic type t' in $\mathcal{O}$, whose parameter component includes p prefixed with $\mathcal{T}$ (e.g., *c.pID*), then (1) t' must have an ⌜out⌝ adornment, and (2) the bindings of t' to which the computation yields, must be a subset of the existing bindings of t when the computation is performed. That is t' is a dependent type on t. This is an ⌜in⌝-⌜out⌝ dependency, which cannot be captured in BSPL. This kind of dependency enables us to specify the causal type of computations that perform information filtering, which we explore in Section 3.3.

Constraint 3: If there is an atomic type t' in $\mathcal{O}$, whose parameter component is p without the prefix $\mathcal{T}$ (e.g., *info* without the prefix *c.*), but there is no atomic type t in $\mathcal{I}$, whose parameter component is p , then t' must have an ⌜out⌝, which can have any binding. This kind of types enable us to specify production of new information as the result of a computation, which may be used to create mappings or aggregations, which we explore in Section 3.3.

These constraints allow only certain adornments of atomic types in $\mathcal{I}$ and $\mathcal{O}$, which we can infer from the names of the atomic types. Hence, we simplify the notation of causal types as follows. Using Constraint1, we drop ⌜in⌝ adornments from the atomic types in $\mathcal{I}$. Using Constraint 2, we drop ⌜out⌝ adornments and use of $\mathcal{T}$ as a prefix of atomic types in $\mathcal{O}$. Using Constraint 3 we drop ⌜out⌝ adornments of atomic types in $\mathcal{O}$. As a result in the rest of the paper, instead of *c : (in pID, in premium) → (out c.pID, out c.premium, out c.info)* we write *c : (pID, premium) → (pID, premium, info)*.

Listing 3 shows a Supple protocol that uses the causal type *c : (pID, premium) → (pID, premium, info)* for a computation in our scenario. The parameter *c*, whose type is *c : (pID, premium) → (pID, premium, info)* with the adornment component ⌜out⌝ is used in the message *request* to indicate the type of computation that the auditor requests. When emitting *request*, the auditor can bind the parameter *c* to any computation conforming to its type. The adornment of *c* in *report* is ⌜in⌝, meaning that the insurer must know the computation that is bound to *c* in order to send an instance of a *report*.

Supple is indifferent to the meaning of parameter bindings. Hence, the binding of *c* can be anything (i.e., a query, constraint, etc.) that agents can interpret as a computation. Type conformance verification methods are out of our scope but program analysis techniques or run-time verifiers [21] can support verification.

Listing 3: Capturing causal types of computations.

```
ReportPoliciesViaConstraint {
 role I, A
 parameter out rID key, out ⟦c⟧
 type c : (pID, premium)→(pID, premium, info)
 CreatePolicy(I, S, out pID key, out premium, out date)
 A ↦ I: request[out rID, out c]
 I ↦ A: report[in rID, in c, out ⟦c⟧]
}
```

Supple introduces ⟦⟧ notation to refer the yielding composite type (i.e., $\mathcal{O}$ in $\mathcal{T} : \mathcal{I} \rightarrow \mathcal{O}$) of a computation. For example, in Listing 3, *report* includes ⟦*c*⟧ which corresponds to the composite type *(out c.pID, out c.premium, out c.info)*.

3.3 Causal Computation Patterns

We introduce major patterns that occur in a variety of applications.

3.3.1 Filter. Filter selects a subset of a parameter's known bindings according to the criteria defined in a computation. For example, the auditor may request the insurer to report only the policies that satisfy some premium criteria (e.g., below of amount). Listing 4 shows *ReportPremium*, in which the parameter *c* is used to define computations to represent such criteria. Listing 5 shows such a computation specified as a (simplified SQL), which can be assigned to *c* by the auditor for requesting only the policies with premium between 50 and 100.

Listing 4: Reporting premiums of selected policies.

```
ReportPremium {
 role I, A
 parameter out rID key, out ⟦c⟧
 type c : (pID, premium)→(pID, premium)
CreatePolicy(I, S, out pID key, out premium, out date)
A ↦ I: request[out rID, out c]
I ↦ A: report[in rID, in c, out ⟦c⟧]
}
```

In Filter, the computation yields to a dependent type of its input. Let K and P be the types of key and non-key parameters, respectively. Filter is represented by the following type definition:

$$(K, P) \to (K, P)$$

Listing 5: Example query in SQL to filter policies.

```
SELECT CreatePolicy.pID AS pID
       CreatePolicy.premium AS premium
WHERE  50 < CreatePolicy.premium < 100
```

3.3.2 Map. Map transforms bindings of a tuple of parameters yielding to new bindings for other tuple of parameters. For instance, the auditor may request the insurer to report its debt for each claim, which is the difference between the claimed and paid amounts. Listing 6 defines *ReportClaimDebt* referring to *MakeClaim*, which makes a claim for an existing policy (identified by *pID*), and produces bindings of *cID* (claim's key), *claimed*, and *paid* amounts. In *ReportClaimDebt*, the insurer's debt for each claim (i.e., *cDebt*) is computed using *claimed* and *paid* according to the computation that is bound to *c* (i.e., the claimed and paid amounts of each claim in each policy are mapped to the debt for the corresponding claim). Note that the outcome of *c* includes *pID*, which is needed to associate the computed debt for a claim with the corresponding policy.

Listing 6: Reporting insurer's debt per claim.

```
ReportClaimDebt {
 role I, A
 parameter out rID key, out ⟦c⟧
 type c : (pID, cID, claimed, paid)→(pID, cID, cDebt)
 CreatePolicy(I, S, out pID key, out premium, out date)
 MakeClaim(S, I, in pID key, out cID key, out claimed,
     out paid)
 A ↦ I: request[out rID, out c]
 I ↦ A: report[in rID, in c, out ⟦c⟧]
}
```

Let K be a the type of key parameters, and P and R be the types of non-key parameters. Map corresponds to the type:

$$(K, P) \to (K, R)$$

3.3.3 Reduce. Reduce yields bindings for a parameter by aggregating another parameter's bindings. For instance, the auditor may want to know the total debt of an insurer for each policy, a solution to which Listing 7 illustrates. In *ReportPolicyDebt*, the insurer's debt per policy is computed using the computation bound to *c*, which assigns the insurer's debt per policy to *pDebt* according to *cDebt* referring to *ReportClaimDebt*. Here, the key *cID* is not involved in the yielding type of *c*, since the reduced debt information is associated only with policies (and reports).

Listing 7: Reporting insurer's debt per policy.

```
ReportPolicyDebt {
 role I, A
 parameter in aID key, in q, in c, out ⟦c⟧
 type c : (rID, pID, cID, cDebt)→(rID, pID, pDebt)
 ReportClaimDebt(I, A, out rID key, out pID key, out
     cID key, out cDebt)
 I ↦ A: aggregate[in aID, in c, out ⟦c⟧]
}
```

Let K_1 and K_2 be the types of key parameters, and P and R be the types of non-key parameters. Reduce corresponds to:

$$(K_1, K_2, P) \to (K_2, R)$$

3.4 A Comprehensive Example

Listing 8 gives a comprehensive example, where the results of one computation are used in another: the auditor requests from the insurer reports on (1) its policies, using filter via *c1*; (2) its debts for those policies, using map via *c2*; and, (3) its total debt over those debts using reduce via *c3*.

Listing 8: Chaining computation results.

```
Report {
 role I, A
 parameter out rID key, out ⟦c3⟧
 type c1 : (pID, premium)→(pID, premium)
 type c2 : (pID, claimed, paid)→(pID, debt)
 type c3 : (pID, debt)→(totalDebt)
 CreatePolicy(I, S, out pID key, out premium, out date)
 MakeClaim(S, I, in pID key, out claimed, out paid)
 A ↦ I: reqPolicies[out rID, out c1]
 I ↦ A: resPolicies[in rID, in c1, out ⟦c1⟧]
 A ↦ I: reqDebts[in rID, in ⟦c1⟧, out c2]
 I ↦ A: resDebts[in rID, in c2, out ⟦c2⟧]
 A ↦ I: reqSum[in rID, in ⟦c2⟧, out c3]
 I ↦ A: resSum[in rID, in c3, out ⟦c3⟧]
}
```

4 FORMAL MODEL AND VERIFICATION

Supple's syntax enhances BSPL with causal types which can be used to define the type of computations that determine bindings of parameters. The requisite computations may be specified in any language that the concerned agents can interpret. Computations may be specified at design time to accommodate restricted situations such as in contracts or regulations.

Below, superscripts $*$ and $+$ denote zero or more, and one or more repetitions, respectively. Delimiters $\lfloor$ and $\rfloor$ identify optional expressions. Cardinality constraints are left informal for readability.

Table 1: Supple's syntax.

Protocol	$\longrightarrow$	*Name*{role *Role*$^+$ parameter$\lfloor$*Parameter*$\lfloor$key$\rfloor\rfloor^+$ $\lfloor$*Type*$^*\rfloor$*Reference*$^+$}
Reference	$\longrightarrow$	*Name*(*Role*$^+$ *Parameter*$^+$) | *Role* $\mapsto$ *Role* : *Name*[*Parameter*$^+$]$\lfloor$:: *Attachment*$\rfloor$
Type	$\longrightarrow$	type *Name* : (*Parameter*$^+$) $\to$ (*Parameter*$^+$)
Attachment	$\longrightarrow$	(*Parameter*$^+$) $\to$ (*Parameter*$^+$){*ComputationSpec*}
Parameter	$\longrightarrow$	*Adornment* (*Name* | ⟦*Name*⟧$\lfloor$.*Name*$\rfloor$)
Adornment	$\longrightarrow$	in | out | nil

A protocol declaration involves a name, two or more roles, one or more parameters, zero or more type definitions, and one or more protocol and message references. A protocol reference consists of a name, one or more roles, and one or more parameters of the referred protocol. Alternatively, a message reference consists of a name, exactly two roles, one or more parameters, and an optional computation attachment. A type definition consists of a parameter name, and two tuples of parameters. A computation attachment consists of two tuples of parameters and a computation specification. A parameter consists of an adornment and a name. Alternatively, the parameter name can be enclosed in double brackets to refer the outcome of a bound computation, from which individual parameters can be accessed using the $\ulcorner\cdot\urcorner$ notation. An adornment is either $\ulcorner$in$\urcorner$, $\ulcorner$out$\urcorner$, or $\ulcorner$nil$\urcorner$.

4.1 Semantics

Supple's semantics enhances BSPL's semantics [22]. The main contribution of Supple pertains to the specification of causal types. As a result, message instances in Supple must satisfy the type of their schema, meaning that each binding matches to the type of the bound parameter. Below, $\vec{\sigma}$ denotes a finite list, which can be treated in places as a set, and $\vec{\sigma} \downarrow_{\vec{\gamma}}$ denotes projection of $\vec{\sigma}$ on to the elements of $\vec{\gamma}$.

Definition 4.1. A *protocol* $\mathcal{P}$ is a tuple $\langle n, \vec{x}, \vec{p}, \vec{k}, \vec{q}, F, T\rangle$, where n is the protocol's name and $\vec{x}, \vec{p}, \vec{k}, \vec{q}$ are lists of roles, public parameters, key parameters, and private parameters, respectively, such that $\vec{k} \subseteq \vec{p}$. F is a finite set of references, such that $\forall f \in F\colon f = \langle n_f, \vec{x}_f, \vec{p}_f, \vec{k}_f\rangle$ is a public projection of a protocol $\mathcal{P}_f = \langle n_f, \vec{x}_f, \vec{p}_f, \vec{k}_f, \vec{q}_f, F_f, T\rangle$ satisfying $\vec{x}_f \subseteq \vec{x}$, $\vec{p}_f \subseteq \vec{p} \cup \vec{q}$, and $\vec{k}_f = \vec{p}_f \cap \vec{k}$. T is a finite set of causal types, such that $\forall t \in T\colon t = \langle p_t, \vec{u}_t, \vec{w}_t\rangle$ satisfying $p_t \in \vec{p} \cup \vec{q}$, $\vec{u}_t \subseteq \vec{p} \cup \vec{q}$, and $\vec{w}_t \subseteq \vec{p} \cup \vec{q}$.

T formalizes the causal types of the form $\mathcal{T} : \mathcal{I} \rightarrow \mathcal{O}$ that we introduce in Section 3.2. Specifically, in a type $\langle p_t, \vec{u}_t, \vec{w}_t\rangle \in T$, p_t, $\vec{u}_t$, and $\vec{w}_t$ correspond to $\mathcal{T}$, $\mathcal{I}$, and $\mathcal{O}$, respectively. For convenience, we treat causal types as global constructs shared by all references of a protocol.

Definition 4.2. A *message schema* $\ulcorner s \mapsto r\colon m\ \vec{p}(\vec{k})\urcorner$ is an atomic protocol $\langle m, \{s, r\}, \vec{p}, \vec{k}, \emptyset, \emptyset, \emptyset\rangle$ with roles s and r, and no references.

Definition 4.3. A *message instance* $m[s, r, \vec{p}, \vec{v}]$ associates a message schema $\ulcorner s \mapsto r\colon m\ \vec{p}(\vec{k})\urcorner$ with a list of values, where $|\vec{v}| = |\vec{p}|$.

Definition 4.4. A *universe of discourse* (UoD) is a pair $\langle \mathcal{R}, \mathcal{M}\rangle$ where $\mathcal{R}$ is a set of roles, and $\mathcal{M}$ is a set of message names, each with its parameters, and sender and receiver roles from $\mathcal{R}$.

Definition 4.5. The *history* of a role x, denoted by H^x, is a sequence of message instances $m_1, m_2, \ldots$, such that each m_i is either emitted or received by x.

A role's history captures the local view of the role with respect to the message instances that are sent and received by the role.

Definition 4.6 captures when a message m is viable in the history of role x. Below we use $\vec{p_I}$ and $\vec{p_O}$ for the lists of $\ulcorner$in$\urcorner$ and $\ulcorner$out$\urcorner$ adorned parameters, respectively, and $\vec{p}_{[\![\,]\!]}$ is the list of causal type parameters, which are enclosed in $[\![\,]\!]$ (i.e., the computation that is bound to the causal type parameter is performed to yield bindings of parameters in the outcome of its type definition). Intuitively, (1) ensures that m is either sent or received by x; (2) ensures that m does not violate uniqueness of parameter bindings; (3) ensures that x knows the bindings of all $\ulcorner$in$\urcorner$ adorned parameters and does not know the bindings of any $\ulcorner$out$\urcorner$ or $\ulcorner$nil$\urcorner$ adorned parameter; (4) ensures that a causal type parameter is bound to a computation before the computation yields to bindings; (5) ensures that, if p is a causal type parameter that is bound to a computation, x knows the bindings of every parameter in $\vec{u}$, before emitting the message with the outcome of the computation that is bound to p—which satisfies Constraint 1 in Section 3.2; and (6) ensures that for every parameter p in $\vec{w}$, if there is a parameter u in $\vec{u}$ whose base name is equal to p, then the bindings of p must be a subset of the known bindings of u—which satisfies Constraint 2 in Section 3.2. In (6) *base* returns the base name of a parameter (i.e., base$(p) = p$, and base$(p.q) = q$). Note that Definition 4.6 satisfies Constraint 3 from Section 3.2 implicitly via (2) and (3), which ensure that $\ulcorner$out$\urcorner$ adorned parameters are bound preserving integrity.

Definition 4.6. A message instance $m[s, r, \vec{p}, \vec{v}]$ with key parameters $\vec{k} \subseteq \vec{p}$ is *viable* at role x's history H^x if and only if all the following conditions hold:

1. $r = x$ (reception) or $s = x$ (emission)
2. $\forall m_i[s_i, r_i, \vec{p_i}, \vec{v_i}] \in H^x$: if $\vec{k} \subseteq \vec{p_i}$ and $\vec{v_i} \downarrow_{\vec{k}} = \vec{v} \downarrow_{\vec{k}}$ then $\vec{v_i} \downarrow_{\vec{p} \cap \vec{p_i}} = \vec{v} \downarrow_{\vec{p} \cap \vec{p_i}}$ (messages respect keys)
3. $\forall p \in \vec{p} : p \in \vec{p_I}$ if and only if $(\exists m_i[s_i, r_i, \vec{p_i}, \vec{v_i}] \in H^x$ and $p \in \vec{p_i}$ and $\vec{k} \subseteq \vec{p_i})$
4. $\forall p \in \vec{p}$: if $p \in (\vec{p_O} \cap \vec{p}_{[\![\,]\!]})$ then $\exists m_i[s_i, r_i, \vec{p_i}, \vec{v_i}] \in H^x$ and $p \in \vec{p_i}$ and $\vec{k} \subseteq \vec{p_i}$.
5. $\forall p \in \vec{p}$: if $p \in \vec{p}_{[\![\,]\!]}$ and $\langle p, \vec{u}, \vec{w}\rangle \in T$ then $\forall u_i \in \vec{u}$: $\exists m_i[s_i, r_i, \vec{p_i}, \vec{v_i}] \in H^x$ and $u_i \in \vec{p_i}$ and $\vec{k} \subseteq \vec{p_i}$
6. $\forall p \in \vec{p}$: if $\langle q, \vec{u}, \vec{w}\rangle \in T$ and $p \in \vec{w}$ and $u \in \vec{u}$ and base$(p) =$ base(u) then $\exists m_i[s_i, r_i, \vec{p_i}, \vec{v_i}] \in H^x$ and $\vec{k} \subseteq \vec{p_i}$ and $\vec{v_i} \downarrow_{\vec{k}} = \vec{v} \downarrow_{\vec{k}}$ and $\vec{v_i} \downarrow_u = \vec{v} \downarrow_p$

An enactment of a protocol is a vector of its roles' histories.

Definition 4.7. Let $\langle \mathcal{R}, \mathcal{M}\rangle$ be a UoD. A *history vector* over $\langle \mathcal{R}, \mathcal{M}\rangle$ is $[H^1, \ldots, H^{|\mathcal{R}|}$, such that $\forall s, r\colon 1 \leq s, r \leq |\mathcal{R}| \Rightarrow H^s$ is a history and $(\forall m[s, r, \vec{p}, \vec{v}] \in H^r\colon m \in \mathcal{M}$ and $m[s, r, \vec{p}, \vec{v}] \in H^s)$.

Definition 4.8. A history vector is *viable* if and only if each of its histories is empty or it arises from the addition of the emission or reception of a viable message by any role to a viable history vector.

Definition 4.9. Let $\langle \mathcal{R}, \mathcal{M}\rangle$ be a UoD. The *universe of enactments* $\mathcal{U}_{\mathcal{R},\mathcal{M}}$ for $\langle \mathcal{R}, \mathcal{M}\rangle$ is the set of viable history vectors with exactly $|\mathcal{R}|$ dimensions over the instances of messages in $\mathcal{M}$.

Definition 4.10. The *intension* of a message schema $\ulcorner s \mapsto r\colon m\ \vec{p}(\vec{k})\urcorner$ for the UoD $\langle \mathcal{R}, \mathcal{M}\rangle$ is $(\![s \mapsto r\colon m\ \vec{p}(\vec{k})]\!)_{\mathcal{R},\mathcal{M}} = \{H | H \in \mathcal{U}_{\mathcal{R},\mathcal{M}}$ and $\exists \vec{v}, i, j\colon H^s_i = m[s, r, \vec{p}, \vec{v}]$ and $H^r_j = m[s, r, \vec{p}, \vec{v}]$.

Definition 4.11. Let $\mathcal{P} = \langle n, \vec{x}, \vec{p}, \vec{k}, \vec{q}, F, T\rangle$ be a protocol. The *intension* of $\mathcal{P}$ for the UoD $\langle \mathcal{R}, \mathcal{M}\rangle$ is $(\![\mathcal{P}]\!)_{\mathcal{R},\mathcal{M}} = (\cup_{\text{cover}(\mathcal{P},G)}$ $(\cap_{G_i \in G}(\![G_i]\!)_{\mathcal{R},\mathcal{M}})) \downarrow_{\vec{x}}$, where cover$(\mathcal{P}, G) \equiv G \subseteq F$ such that $\forall p \in \vec{p}\colon (\exists G_i \in G\colon G_i = \langle n_i, x_i, p_i\rangle$ and $p \in \vec{p_i})$.

Definition 4.12. Let $\mathcal{P} = \langle n, \vec{x}, \vec{p}, \vec{k}, \vec{q}, F, T\rangle$ be a protocol. The *universe of discourse* of $\mathcal{P}$ is UoD$(\mathcal{P}) = \langle$roles$(\mathcal{P})$, msgs$(\mathcal{P})\rangle$, where roles$(\mathcal{P}) = \vec{x} \cup (\cup_i$roles$(F_i))$ and msgs$(\mathcal{P}) = \cup_i F_i$.

4.2 Liveness and Safety

Liveness and safety are two key properties of Supple protocols. A protocol is live if every enactment of the protocol can be completed by producing bindings of all public parameters.

Definition 4.13. A protocol $\mathcal{P}$ is *live* if and only if each history vector in the protocol's universe of enactments UoD$(\mathcal{P})$ can be extended through finitely many message emissions and receptions to a history vector in UoD($\mathcal{P}$) that is complete.

A protocol is safe if no key constraint is violated despite local decisions on messages emitted by each role.

Definition 4.14. A protocol $\mathcal{P}$ is *safe* if and only if each history vector in its $(\!|\mathcal{P}|\!)_{\mathrm{UoD}(\mathcal{P})}$ is safe. A history vector is safe if and only if all key uniqueness constraints apply across all histories in it.

Liveness and safety in Supple go beyond those of BSPL. In Supple, liveness requires, if there is causal type parameter that is bound to a computation, the role, who performs the computation, must know the bindings of the parameters on which the computation is performed. For instance, consider *LivenessFailure* in Listing 9, which is a variant of *ReportClaimDebt* in Listing 6.

Listing 9: Liveness fails for variant of ReportClaimDebt.

```
LivenessFailure {
 role I, A, S
 parameter out rID key, out ⟦c2⟧
 type c1 : (pID, cID, claimed, paid)→(pID, cID, cDebt)
 type c2 : (pID, cID, complaint)→(pID, cID, cDebt)
 // CreatePolicy and MakeClaim as before
 A ↦ I: auditReportRequest[out rID, out c1]
 I ↦ A: auditReport[in rID, in c1, out ⟦c1⟧]
 S ↦ A: complaintSubmission[out rID, out complaint]
 A ↦ I: complaintReportRequest[in rID, out c2]
 I ↦ A: complaintReport[in rID, in c2, out ⟦c2⟧]
}
```

LivenessFailure includes an additional causal type *c2* and new messages *complaintSubmission*, *complaintReportRequest*, and *complaintReport* to capture the scenario where a subscriber makes a complaint about a policy to the auditor, and the auditor requests a report from the insurer about the policy that is subject to the complaint. The type definition of *c2* requires *complaint* to be available to the insurer. However, *complaint* appears only in *complaintSubmission*, which is sent from the subscriber to the auditor. Thus, the insurer can never send a *complaintReport* in any enactment where it receives a *complaintReportRequest* message. Note that some enactments of *LivenessFailure* can be completed, e.g., enactments where the auditor sends a *auditReportRequest*, which does not refer to *c2*. This protocol would be live if *complaint* was included as an ⌜in⌝ parameter in *complaintReportRequest*.

Safety of a protocol means that each of its enactments ensures integrity of information exchanged. Safety may be violated if two or more roles (1) as in BSPL, may bind the same parameter; or (2) as added by Supple, concurrently perform the computation that is bound to a causal type parameter. For instance, *SafetyFailure* in Listing 10, where the auditor requests the total claimed amount for a policy from both insurer and subscriber, is unsafe.

Listing 10: Safety failure.

```
SafetyFailure {
 role I, A, S
 parameter out rID key, out ⟦c⟧
 type c : (pID, cID, cClaim)→(pID, pClaim)
 // CreatePolicy and MakeClaim as before
 A ↦ I: reqTotalCalimI[out rID, out c]
 A ↦ S: reqTotalClaimS[in rID, in c]
 I ↦ A: repTotalClaimI[in rID, in c, out ⟦c⟧]
 S ↦ A: repTotalClaimS[in rID, in c, out ⟦c⟧]
}
```

Both *repTotalClaimI* and *repTotalClaimS* use the same computation bound to *c* to determine the total claims for a policy. However, because of asynchrony, the same information is not available to all parties and applying the same computation may produce different results, thus violating integrity. For example, in Figure 2, which shows a possible enactment omitting the binding of *c* for readability, the auditor sends *reqTotalClaimI* and *reqTotalClaimS* to the insurer and subscriber, respectively. The insurer computes the total claim amount for the policy, where *pID* is equal to *p*1, before receiving the *submitClaim* message of the subscriber. Hence, the insurer sends *repTotalClaimI* with *pClaim* of 0 to the auditor. In the meantime, the subscriber submits a claim for 15 to the insurer and, therefore, responds to the auditor's request by sending *repTotalClaimS* with *pClaim* of 15. As a result, the auditor receives inconsistent bindings of *pClaim*, which violates integrity for the binding *p*1 of key *pID*.

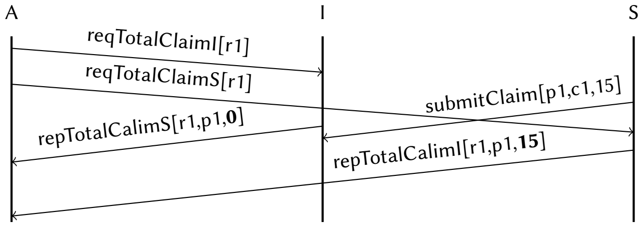


Figure 2: Integrity of *pClaim* is violated in unsafe protocol.

4.3 Verification of Supple Protocols

To verify correctness, we derive the following propositional logic expressions from a Supple protocol (1) C: its causal structure, indicating how information flows via its messages; (2) $\mathcal{S}$: its unsafety, i.e., that two messages that produce bindings for the same parameter both occur; and (3) $\mathcal{L}$: its liveness failure, i.e., some parameters remain unbound even tough every agent sends every viable message, and the infrastructure delivers all messages. Safety and liveness hold when $C \wedge \mathcal{S}$ and $C \wedge \mathcal{L}$, respectively, are unsatisfiable.

Algorithm 1 defines the algorithm to create the causal structure of a protocol as a propositional logic expression. Algorithm 1 creates the expression step by step by as a conjunction of subexpression. For readability, we define the subexpressions separately in Definitions 4.15–4.19.

Algorithm 1: Generation of the causal structure of a protocol $\mathcal{P}$ as a proposition logic expression.

```
input  : $\mathcal{P} = \langle n, \vec{x}, \vec{p}, \vec{k}, \vec{q}, F, T \rangle$
output : $C^{\mathcal{P}}$ // expression of $\mathcal{P}$'s causal structure
$C^{\mathcal{P}} \leftarrow$ message-reception-exp // Definition 4.15
$C^{\mathcal{P}} \leftarrow C^{\mathcal{P}} \wedge$ information-transmission-exp // Definition 4.16
$C^{\mathcal{P}} \leftarrow C^{\mathcal{P}} \wedge$ information-reception-exp // Definition 4.17
$C^{\mathcal{P}} \leftarrow C^{\mathcal{P}} \wedge$ information-minimality-exp // Definition 4.18
$C^{\mathcal{P}} \leftarrow C^{\mathcal{P}} \wedge$ message-ordering-exp // Definition 4.19
return $C^{\mathcal{P}}$
```

Given a protocol $\mathcal{P} = \langle n, \vec{x}, \vec{p}, \vec{k}, \vec{q}, F, T \rangle$, Algorithm 1 uses the following sets of propositional symbols. (1) A finite set $\mathbb{P}$ of p_x symbols for each parameter $p \in \vec{p} \cup \vec{q}$ and for each role $x \in \vec{x}$, which model observation of p by x. (2) A finite set $\mathbb{M}$ of m_x symbols for each message $m \in F$ and for each role $x \in \vec{x}$, which model observation of m by x. (3) A finite set of b_{e_i,e_j} symbols for every (e_i, e_j) pair in $\mathbb{P} \cup \mathbb{M}$, which model observation of e_i before e_j. For readability, we write b_{e_i,e_j} in predicate form as $\mathit{before}(e_i, e_j)$. (4) A finite set of t_{e_i,e_j} symbols for every (e_i, e_j) pair in $\mathbb{P} \cup \mathbb{M}$, which model observation of e_i together with e_j. For readability, we write t_{e_i,e_j} in predicate form as $\mathit{together}(e_i, e_j)$.

Definitions 4.15–4.19 define the subexpressions in Algorithm 1. We provide examples for Report protocol in Listing 8.

Definition 4.15. (Message Reception) A message is received only if it is emitted earlier:

$$\bigwedge_{m \in F} (\neg m_r \vee \mathit{before}(m_s, m_r))$$

For message *reqDebts*, Definition 4.15 yields to the expression $\neg \mathit{reqDebts}_I \vee \mathit{before}(\mathit{reqDebts}_A, \mathit{reqDebts}_I)$.

Definition 4.16. (Information Transmission) For every message schema $\ulcorner s \mapsto r\colon m\ \vec{p}(\vec{k}) \urcorner \in F$, s either does not emit m, or:

- s observes all $\ulcorner$in$\urcorner$ parameters in $\vec{p}$ that are not bound to a computation, before the emission of m:

$$exp_1 = \bigwedge_{p \in \vec{p}_I \setminus \vec{p}_{[\![]\!]}} \mathit{before}(p_s, m_s)$$

- and, s observes all $\ulcorner$in$\urcorner$ parameters in the outcome of all causal type parameters in $\vec{p}$ before the emission of m:

$$exp_2 = \bigwedge_{\langle q, \vec{u}, \vec{w} \rangle \in \vec{p}_I \cap \vec{p}_{[\![]\!]}} \Big(\bigwedge_{p \in \vec{w}} \mathit{before}(p_s, m_s) \Big)$$

- and, s observes all the requisite parameters of all $\ulcorner$out$\urcorner$ causal type parameters in $\vec{p}$ before the emission of m:

$$exp_3 = \bigwedge_{\langle q, \vec{u}, \vec{w} \rangle \in \vec{p}_O \cap \vec{p}_{[\![]\!]}} \Big(\bigwedge_{p \in \vec{u}} \mathit{before}(p_s, m_s) \Big)$$

- and, s observes all $\ulcorner$out$\urcorner$ parameters in $\vec{p}$ that are not bound to a computation together with the emission of m:

$$exp_4 = \bigwedge_{p \in \vec{p}_O \setminus \vec{p}_{[\![]\!]}} \mathit{together}(p_s, m_s)$$

- and, s observes all the outcome parameters of all $\ulcorner$out$\urcorner$ causal type parameters in $\vec{p}$ together with the emission of m:

$$exp_5 = \bigwedge_{\langle q, \vec{u}, \vec{w} \rangle \in \vec{p}_O \cap \vec{p}_{[\![]\!]}} \Big(\bigwedge_{p \in \vec{w}} \mathit{together}(p_s, m_s) \Big)$$

Hence, we generate:

$$\bigwedge_{m \in F} (\neg m_s \vee (exp_1 \wedge exp_2 \wedge exp_3 \wedge exp_4 \wedge exp_5))$$

For example, for $\ulcorner$*A* $\mapsto$ *I: reqDebts[in rID, in* $[\![$*c1*$]\!]$*, out c2]*$\urcorner$ and *c1* of type *(pID, premium)⟶(pID, premium)*, Definition 4.16 yields to $\neg \mathit{reqDebts}_A \vee (\mathit{before}(\mathit{rID}_A, \mathit{reqDebts}_A) \wedge \mathit{before}(\mathit{pID}_A, \mathit{reqDebts}_A) \wedge \mathit{before}(\mathit{premium}_A, \mathit{reqDebts}_A) \wedge \mathit{together}(c2_A, \mathit{reqDebts}_A))$.

Definition 4.17. (Information Reception) For every message schema $\ulcorner s \mapsto r\colon m\ \vec{p}(\vec{k}) \urcorner \in F$, r either does not observe m, or r observes all parameters of m either before or together with the reception of m:

$$\bigwedge_{m \in F} \Big(\neg m_r \vee \Big(\bigwedge_{p \in \vec{p}} (\mathit{before}(p_r, m_r) \vee \mathit{together}(p_r, m_r))\Big)\Big)$$

For example, considering only *premium* for brevity, for message $\ulcorner$*I* $\mapsto$ *A: resPolicies[in rID, in c1, out* $[\![$*c1*$]\!]$*]*$\urcorner$ and *c1* of type *(pID, premium)⟶(pID, premium)* Definition 4.17 yields $\neg \mathit{resPolicies}_A \vee (\mathit{before}(\mathit{premium}_A, \mathit{resPolicies}_A) \vee (\mathit{together}(\mathit{premium}_A, \mathit{resPolicies}_A)))$:

Definition 4.18. (Information Minimality) For every role x in $\vec{x}$ and parameter p in $\vec{p}$, p is either not observed or p is observed together with a message m in F (for readability, assume that $F' \subseteq F$ such that $\ulcorner s \mapsto r : m\ \vec{p}_m(\vec{k}_m) \urcorner \in F$ and ($x = s$ or $x = r$, and $p \in \vec{p}_m$)):

$$\bigwedge_{x \in \vec{x} \text{ and } p \in \vec{p}} \Big(\neg p_x \vee \bigvee_{m \in F'} \mathit{together}(p_x, m_x)\Big)$$

Definition 4.19. (Ordering) For every pair of messages m^i and m^j in F that are emitted by x in $\vec{x}$, x may observe them in some order, but not together (for readability, assume that $F' \subseteq F$ such that $\ulcorner s \mapsto r\colon m\ \vec{p}_m(\vec{k}_m) \urcorner \in F$, and $x = s$):

$$\bigwedge_{m^i, m^j \in F'} (\neg m^i_s \vee \neg m^j_s \vee \mathit{before}(m^i_s, m^j_s) \vee \mathit{before}(m^j_s, m^i_s))$$

4.3.1 *Correctness of Causal Structure Generation.*

Let $\mathcal{P}$ be a protocol for which Algorithm 1 generates the causal structure $C^{\mathcal{P}}$.

Theorem 4.20. (Correspondence) *For every viable history vector of $\mathcal{P}$, there is a model of $C^{\mathcal{P}}$, and vice versa.*

Proof Sketch. We use induction in the forward direction. An empty H corresponds to an empty $C^{\mathcal{P}}$ without any propositions. Inductively, for every viable message m that extends H, we can extend $C^{\mathcal{P}}$ for m's emission using the above information transmission clauses, and for m's reception using the reception clause. Conversely, given $C^{\mathcal{P}}$, we can construct H, simply appending messages instances that correspond to the message emission and reception propositions to the histories of the corresponding roles. ◻

Theorem 4.21. (Termination) *Algorithm 1 always terminates.*

Proof Sketch. The lists of roles, messages, and parameters of a protocol are finite, and the rules in Definitions 4.15–4.19, when applied to finite lists, generate finite expressions. Algorithm 1 uses each rule once and terminates. ◻

4.3.2 *Safety.*

A protocol's safety requires that, if any parameter is adorned $\ulcorner$out$\urcorner$ in two or more messages, only one of these messages is emitted. Hence, integrity of parameter bindings cannot be violated. This means that for any pair of messages m^i and m^j in a protocol (and with corresponding $\ulcorner$out$\urcorner$ adorned parameters p^i_O and p^j_O, respectively) where $p^i_O \cap p^j_O \neq \emptyset$, we must not infer the clause $m^i_{s^i} \wedge m^j_{s^j}$ from the causal structure of a protocol.

Definition 4.22. Given a protocol's list of messages F, let $\vec{f}$ be the list of every message pair (m^i, m^j) for which p^i_O and p^j_O are the respective $\ulcorner$out$\urcorner$ adorned parameters and $p^i_O \cap p^j_O \neq \emptyset$. *Unsafety expression* of the protocol is:

$$\bigvee_{(m^i, m^j) \in \vec{f}} (m^i_s \wedge m^j_s)$$

Let $\mathcal{P}$ be a protocol, $C^{\mathcal{P}}$ be the causal structure of $\mathcal{P}$, and $\mathcal{S}^{\mathcal{P}}$ be the unsafety expression of $\mathcal{P}$ as in Definition 4.22. We decide on $\mathcal{P}$'s safety by checking unsatisfiability of $C^{\mathcal{P}} \wedge \mathcal{S}^{\mathcal{P}}$.

Theorem 4.23. *A protocol $\mathcal{P}$ is safe if and only if $C^{\mathcal{P}} \wedge \mathcal{S}^{\mathcal{P}}$ is not satisfiable.*

Proof Sketch. If $C^{P} \wedge \mathcal{S}^{P}$ is satisfiable, by Theorem 4.20, we can construct a history vector that contains at least two messages that bind the same ⌜out⌝ adorned parameter. Conversely, if $C^{\mathcal{P}} \wedge \mathcal{S}^{\mathcal{P}}$ is not satisfiable, by Theorem 4.20, we cannot construct a history vector that contains more than one message to bind the same ⌜out⌝ adorned parameter. □

4.3.3 Liveness. A protocol's is live if every enactment of the protocol can be completed (i.e., every public parameter are bound). To verify liveness of a protocol, we represent its incompleteness as follows. First, we consider lack of a public parameter's observation. That is, for each public parameter p, we create a conjunction that is composed of negated p_x symbols for each role x who observes p in a message, and compose these clauses into a disjunction.

Definition 4.24. (Lack of Public Parameter Observation) Let $\vec{p}$ be the list of a protocol's public parameters and X_p be the set of roles in the protocol who are either sender or receiver of at least one message in the protocol which includes p as a parameter.

$$exp_a = \bigvee_{p \in \vec{p}} (\bigwedge_{x \in X_p} \neg p_x)$$

Second, to avoid situations in which an agent intentionally decides not to emit any of the currently viable messages, we should have an expression to ensure that if a sender of a message observes its ⌜in⌝ adorned parameters but does not observe its ⌜out⌝ adorned parameters, then the sender observes the message.

Definition 4.25. (Emission Enforcement of Viable Messages) Let m be a message, $\vec{p}_I^m$ and $\vec{p}_O^m$ be the list of ⌜in⌝ and ⌜out⌝ adorned parameters of m, and s be the sender of m.

$$exp_b = \bigwedge_{m \in F} (m_s \vee \bigvee_{p_i \in \vec{p}_I^m} \neg p_i \vee \bigvee_{p_o \in \vec{p}_O^m})$$

Third, to avoid situations where a lossy communication channel drops some of the emitted messages, we should have an expression to ensure that every emitted message is received

Definition 4.26. (Nonlossy Channel) Let m be a message with sender role s and receiver role r.

$$exp_c = \bigwedge_{m \in F} (\neg m_s \vee m_r)$$

Let, $\mathcal{L}^{\mathcal{P}}$ be $exp_a \wedge exp_b \wedge exp_c$, where exp_a, exp_b, and exp_c are the corresponding expressions in Definitions 4.24, 4.25, and 4.26, respectively, for protocol $\mathcal{P}$. We decide on the liveness of $\mathcal{P}$ by checking unsatisfiability of $C^{\mathcal{P}} \wedge \mathcal{L}^{\mathcal{P}}$.

Theorem 4.27. *A protocol $\mathcal{P}$ is live if and only if $C^{\mathcal{P}} \wedge \mathcal{L}^{\mathcal{P}}$ is not satisfiable.*

Proof Sketch. If $C^{\mathcal{P}} \wedge \mathcal{L}^{\mathcal{P}}$ is satisfiable, by Theorem 4.20, we can construct a viable history vector that cannot be extended via message emission (maximality) or reception (lossless transmission), and yet is incomplete. Conversely, if $\mathcal{P}$ is live, we know that each history vector of $\mathcal{P}$ is either complete or can be finitely extended to a complete history vector. Hence, $C^{\mathcal{P}} \wedge \mathcal{L}^{\mathcal{P}}$ is not satisfiable. □

5 DISCUSSION

Supple introduces a new abstraction of types in protocols, on which there is little work, and supporting checking of important properties. These contributions coincide with the growing interest in richer data-aware specifications. As noted in Section 1, existing work when it applies types does so only for static purposes (ontology annotations on fields). It doesn't consider information modeling of *interactions*, let alone the advanced typing techniques we introduce.

Our overarching contribution lies in elucidating an important aspect of information-based protocols via causal types. Supple extends information-based protocol specification approaches with causal types for constraining the information that is communicated in a protocol. Supple treats causal type parameters as first-class information parameters, which enables the agents (1) to define computations during enactment of a protocol, and (2) to communicate them and their results as they communicate any other information. Supple formalizes causal types and incorporates them into verification of a protocol's safety and liveness. At the technical level, Supple provides a flexible and formal method to define constraints on the exchanged information in a protocol.

Aspects of data have begun to receive more attention in protocol specification. HAPN [24] complements state machine-based representation of protocols with guards and effects. A major conceptual difference is that HAPN supports *system* parameters, whose bindings are produced exogenously to the interaction, indicating shared state between agents and incorporation of internal decision-making in the specification of public interactions. In Supple, by contrast, interaction state as captured by parameter bindings is neither global nor includes any agent's internal state. SPY [18] adds assertions to session types to constrain communicated values. However, neither HAPN nor SPY support causal types as in Supple.

Winikoff and Cranefield [23] study the testability of BDI agent programs and identify challenges in scalability. Supple could facilitate helping manage those challenges in a MAS setting in two ways. First, the decoupling of agents through interactions would reduce the verification problem to each agent separately. Second, the existence of a strong type discipline can reduce the burden on testing by eliminating certain kinds of interaction errors early.

At the conceptual level, Supple's contribution is in line with the challenge of connecting operational and meaning-based protocol specifications [15]. In fact, most previous formalizations of commitments and norms, in general, are in terms of computations [1, 5, 7], and could thus benefit from Supple's enhancements.

An interesting future direction is whether norm-related computations can be mapped at least partially to Supple computations with the aim of regimenting some interactions. Splee gives the example of auctions: the auctioneer's commitment to seller to declare the highest bidder as winner yields a query attachment (i.e., a computation in Supple) that aggregates over all bids receives to produce a binding for winner. Security is an interesting direction: norms could be used to generate information protocols with computations that serve as policies, e.g., for privacy.

REFERENCES

[1] Alexander Artikis, Marek J. Sergot, and Jeremy V. Pitt. 2009. Specifying Norm-Governed Computational Societies. *ACM Transactions on Computational Logic* 10, 1 (Jan. 2009), 1:1–1:42.

[2] Matteo Baldoni, Cristina Baroglio, Federico Capuzzimati, and Roberto Micalizio. 2018. Type checking for protocol role enactments via commitments. *Autonomous Agents and Multi-Agent Systems* 32, 3 (2018), 349–386.

[3] Matteo Baldoni, Cristina Baroglio, Elisa Marengo, Viviana Patti, and Federico Capuzzimati. 2014. Engineering Commitment-Based Business Protocols with the 2CL Methodology. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 28, 4 (July 2014), 519–557.

[4] Bernhard Bauer and James Odell. 2005. UML 2.0 and Agents: How to Build Agent-Based Systems with the New UML Standard. *Engineering Applications of Artificial Intelligence* 18, 2 (March 2005), 141–157.

[5] Federico Chesani, Paola Mello, Marco Montali, and Paolo Torroni. 2013. Representing and Monitoring Social Commitments using the Event Calculus. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 27, 1 (2013), 85–130.

[6] Amit K. Chopra, Samuel H. Christie V, and Munindar P. Singh. 2017. Splee: A Declarative Information-Based Language for Multiagent Interaction Protocols. In *Proceedings of the 16th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, São Paulo, 1054–1063.

[7] Amit K. Chopra and Munindar P. Singh. 2016. Custard: Computing Norm States over Information Stores. In *Proceedings of the 15th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. IFAAMAS, Singapore, 1096–1105.

[8] Ferruccio Damiani, Paola Giannini, Alessandro Ricci, and Mirko Viroli. 2012. Standard Type Soundness for Agents and Artifacts. *Scientific Annals of Computer Science* 22, 2 (2012), 267–326.

[9] Riccardo De Masellis, Chiara Di Francescomarino, Chiara Ghidini, Marco Montali, and Sergio Tessaris. 2017. Add Data into Business Process Verification: Bridging the Gap between Theory and Practice. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. 1091–1099.

[10] FIPA. 2003. FIPA Interaction Protocol Specifications. (2003). FIPA: The Foundation for Intelligent Physical Agents, http://www.fipa.org/repository/ips.html.

[11] Akın Günay, Michael Winikoff, and Pınar Yolum. 2015. Dynamically Generated Commitment Protocols in Open Systems. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 29, 2 (2015), 192–229.

[12] HL7. 2007. Health Level Seven. (2007). http://www.hl7.org.

[13] Antonis C. Kakas, Nicolas Maudet, and Pavlos Moraitis. 2005. Modular Representation of Agent Interaction Rules through Argumentation. *Autonomous Agents and Multi-Agent Systems* 11, 2 (2005), 189–206.

[14] Bhavik Katira. 2015. *Syndicated Loan FpML Requirements: Business Requirements Document Version 2.0*. TR. The LSTA Agent Bank Communications Working Group, International Swaps and Derivatives Association.

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

[16] Tim Miller and Jarred McGinnis. 2008. Amongst First-Class Protocols. In *Proceedings of the 8th International Workshop on Engineering Societies in the Agents World (ESAW 2007) (Lecture Notes in Computer Science)*, Vol. 4995. Springer, Athens, 208–223.

[17] Marco Montali, Diego Calvanese, and Giuseppe De Giacomo. 2014. Verification of data-aware commitment-based multiagent system. In *Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems*. IFAAMAS, Paris, 157–164.

[18] Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. 2013. SPY: Local Verification of Global Protocols. In *Proceedings of the International Conference on Runtime Verification (LNCS)*, Vol. 8174. Springer, 358âĂŞ363.

[19] Michael Rovatsos, Dimitrios I. Diochnos, and Matei Craciun. 2015. Agent Protocols for Social Computation. In *Advances in Social Computing and Multiagent Systems - 6th International Workshop on Collaborative Agents Research and Development, CARE 2015 and 2nd International Workshop on Multiagent Foundations of Social Computing, MFSC (Communications in Computer and Information Science)*, Vol. 541. Springer, Istanbul, 94–111.

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

[21] Munindar P. Singh. 2011. 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)*. IEEE Computer Society, Washington, DC, 57–64.

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

[23] Michael Winikoff and Stephen Cranefield. 2014. On the Testability of BDI Agent Systems. *Journal of Artificial Intelligence Research (JAIR)* 51 (Sept. 2014), 71–131.

[24] Michael Winikoff, Nitin Yadav, and Lin Padgham. 2018. A New Hierarchical Agent Protocol Notation. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)* 32, 1 (Jan. 2018), 59–133.

[25] WS-CDL. 2005. Web Services Choreography Description Language Version 1.0. (Nov. 2005). www.w3.org/TR/ws-cdl-10/.