Executable Logic for Reasoning and Annotation of First-Class Agent Interaction Protocols

Tim Miller and Peter McBurney
Department of Computer Science,
University of Liverpool, Liverpool, L69 7ZF, UK
{tim, p.j.mcburney}@csc.liv.ac.uk

Abstract

Interaction protocols are seen a promising approach to coordination in multi-agent systems. However, many practitioners view agent interaction protocols as rigid specifications that are defined *a priori*, and hard-coded their agents with a set of protocols known at design time — a restriction that is out of place with the goals of agents being intelligent and adaptive. To achieve the full potential of multi-agent systems, we believe that it is important that multi-agent interaction protocols are treated first-class computational entities in systems. That is, exist at runtime in systems as entities that can be referenced, inspected, composed, and shared, rather than as abstractions that emerge from the behaviour of the participants. We use the term *first-class protocol* to refer to such protocols. We propose a framework, called $\mathcal{RASA}$, which regards protocols as first-class entities. Rather than having hard-coded decision making mechanisms for choosing their next move, agents can inspect the protocol specification at runtime to do so. In this paper, we present a logic; that is, a syntax, semantics, and proof system, that is part of the $\mathcal{RASA}$ framework, which is used to document the outcomes of first-class protocols, so that agents can maintain a library of protocols, each annotated with their meaning, and can quickly and correctly assess which protocol best achieves their goals.

Keywords: multi-agent systems, agent interaction protocols, dynamic logic

1 Introduction

Research into multi-agent systems aims to promote autonomy and intelligence into software agents. Intelligent agents should be able to interact socially with other agents, and adapt their behaviour to changing conditions. Despite this, research into interaction in multi-agent systems is focused mainly on the documentation of interaction protocols, which specify the set of possible interactions for a protocol in which agents engage. Agent developers use these specifications to hard-code the interactions of agents. We identify three significant disadvantages with this approach: 1) it strongly couples agents with the protocols they use — something which is unanimously discouraged in software engineering — therefore requiring agent code to changed with every change in a protocol; 2) agents can only interact using protocols that are known at design time, a restriction that seems out of place with the goals of agents being intelligent and adaptive; and 3) agents cannot compose protocols at runtime to bring about more complex interactions, therefore restricting them to protocols that have been specified by human designers — again, this seems out of place with the goals of agents being intelligent and adaptive.

We propose a framework, called $\mathcal{RASA}$, which regards protocols as *first-class* entities. These first-class protocols are documents that exists within a multi-agent system, in contrast to hard-coded protocols, which exist merely as abstractions that emerge from the messages sent by the participants. To promote decoupling of agents from the protocols they use, we propose a formal, executable language for protocol specification. This language consists of a process algebra, used to specify the messages that can be sent. The *rules* governing under which conditions messages can be sent, and the *effects* that sending messages has on a system are specified using constraints. Rather than a protocol specification being just a sequence of arbitrary tokens, each message contains a meaning represented as a constraint. Instead of hard-coding the process of message sending, designers can implement goal-directed agents that reason about the effect of the messages they send and receive, and can choose the course of action that best achieves their goals. Agents able to reason about protocols can therefore learn of new protocols at runtime, making them more adaptable, for example, by being able to interact with new agents that insist on using specific protocols. The $\mathcal{RASA}$ language also allows protocols to be composed to bring about more complex interactions. The $\mathcal{RASA}$ protocol specification language and its operational semantics are presented in an earlier paper [16].

A major goal of research into first-class protocols is for agents to maintain a library of interaction protocols, and to be able to select the protocol that best suits the goals that it wants to achieve. For this, agents must be able to quickly and correctly determine the outcomes that can result for an interaction protocol, and compare protocols in their library.

In this paper, we present a logic for $\mathcal{RASA}$, which allows agents and designers to annotate protocols with information about their outcomes; that is, what the protocol achieves. Agent annotate the protocols in their library with formulae in this logic, which can then help determine which protocol best suits the goals that it wants to achieve at particular times. The sub-protocols that make up a protocol can also be annotated, allowing the agent to use the annotations to calculate the path of interaction that best suits its goals. Annotations are derived directly from the protocol specification, although we note that other annotations would be possible, for example, annotations which document whether a protocol is secure. We note that this logic is not used to define the behaviour of protocols, but merely for reasoning about their outcomes.

The outline of this paper is as follows. In what remains of this section, we define our notion of "first-class protocol". In Section 2 and present an overview of the $\mathcal{RASA}$ framework, including the syntax and denotational semantics of the $\mathcal{RASA}$ protocol specification language. Section 3 presents the language syntax and semantics of the logic used to for annotation and reasoning. Section 4 presents a deductive proof system for this logic, which is used to prove properties about $\mathcal{RASA}$ protocols, and to define valid annotations. Section 6 discusses conclusions and future work.

1.1 First-Class Protocol — A Definition

Our notion of first-class protocol is comparable to the notion of first-class object/entity in programming languages [25]. That is, a first-class protocol is a referencable, sharable, manipulatable entity that exists as a runtime value in a multi-agent system. From the definition of a first-class protocol, participating agents should be able to inspect the definition to learn the rules and effects of the protocol by knowing only the syntax and semantics of the language, and the ontology used to describe rules and effects.

To this end, we define four properties that constitute a first-class protocol language:

- *Formal:* The language must be formal to eliminate that possibility of ambiguity in the meaning of protocols, to allow agents to reason about them using their machinery, and to allow agents to pass and store the protocol definitions as values.

- *Meaningful:* The meaning of messages must be specified by the protocol, rather than simply specifying arbitrary communication actions whose semantics are defined outside the scope of the document. Otherwise, one may encounter a communicative action of which they do not know the definition, rendering the protocol useless.
- *Inspectable/executable:* Agents must be able to reason about the protocols at runtime in order to derive the rules and meaning of the protocol, so that they can determine the messages they will send that best achieve their goals, and compare the rules and effects of different protocols.
- *Dynamically composable:* If an agent does not have access to a protocol that helps to achieve its goals, then it should be able to compose new protocols that do at runtime, possibly from existing protocols. This new protocol must also form a first-class protocol in its own right.

We emphasise here that first-class does not equal *global*. By global, we mean languages that specify the protocol from a global view of the interaction, rather than from the view of the individual participants. Therefore, languages such as AgentUML and FSM-based languages are not first-class, as is commonly commented, even though they are global. AgentUML is not meaningful (although one could adapt it quite easily to make it meaningful), and the composability at runtime could also be difficult, if possible at all. FSM approaches could also add meaning, but the authors are not aware of any current FSM approaches that are executable and support dynamic composition.

2 The $\mathcal{RASA}$ Framework

To achieve the full vision of multi-agent systems, we believe it is necessary to treat protocols as first-class entities. The $\mathcal{RASA}$ framework was designed to allow us to study first-class protocols, and the types of comments we can make about them. Our idea, along with other researchers working in this area, is to have goal-directed agents with access to libraries of first-class protocols. If an agent would like a service offered by another, they can negotiate a protocol to use. If an agent would like to interact with another to achieve a particular goal, it can search its protocol library to find the protocol that best achieves its goals. If no such protocol exists, runtime composition may provide an alternative.

The $\mathcal{RASA}$ specification language was designed as an example of the minimal operators that would be required for a successful first-class protocol specification language. First presented in [16], along with its operational semantics, the language uses constraint languages and process algebra to specify interaction protocols. In this section, we briefly present the language, and its denotational semantics, which we use in subsequent sections.

2.1 Modelling Information

Communication in multi-agent systems is performed across a *universe of discourse*. Agents send messages expressing particular properties about the universe. We assume that these messages refer to *variables*, which represent the parts of the universe that have changing values, and use other *tokens* to represent relations, functions, and constants to specify the properties of these variables and how they relate to each other.

Rather than devise a new language for expressing information, or using an existing language, we take the approach that any constraint language can be used to model the universe of discourse, provided that it has a few basic constants, operators and properties. This allows us to express

and study a wider variety of protocols, such as those that use description logics [1], constraint programming languages [22], or even predicate and modal logics [2]. It also permits us to use different mechanisms for defining protocol meaning, such as norms and commitments.

Definition 2.1. Cylindric constraint system

We assume that the underlying communication language fits the definition of a *cylindric constraint system* proposed by De Boer *et al.* [5]. They define a cylindric constraint system as a complete algebraic lattice, $\langle C, \sqsupseteq, \sqcup, \text{true}, \text{false}, Var, \exists \rangle$. In this structure, C is the set of atomic propositions in the language, for example $X \leq Y$, $\sqsupseteq$ is an entailment operator, true and false are the least and greatest elements of C respectively, $\sqcup$ is the least upper bound operator, Var is a countable set of variables, and $\exists$ is an operator for hiding variables. The entailment operator defines a partial order over the elements in the lattice, such that $c \sqsupseteq d$ means that the information in d can be derived from c. The shorthand $c = d$ is equivalent to $c \sqsupseteq d$ and $d \sqsupseteq c$. We will use $\mathcal{L}$ to refer to the language, as well as the set of all constraints in the language; for example, $c \in \mathcal{L}$.

A constraint is one of the following: an atomic proposition, c, for example, $X = 1$, where X is a variable; a conjunction, $\phi \sqcup \psi$, where ϕ and ψ are constraints; or $\exists_x \phi$, where ϕ is a constraint and $x \in Var$. We extend this notation by allowing negation on the right of an entailment operator, for example, $c \sqsupseteq \neg d$, which is equivalent to $c \not\sqsupseteq d$. Other propositional operators are then defined from these: $\phi \vee \psi \mathrel{\widehat{=}} \neg(\neg\phi \wedge \neg\psi)$, $\phi \rightarrow \psi \mathrel{\widehat{=}} \neg\phi \vee \psi$, and $\phi \leftrightarrow \psi \mathrel{\widehat{=}} \phi \rightarrow \psi \wedge \psi \rightarrow \phi$. We will continue to use the symbols ϕ and ψ to refer to constraints throughout this paper. We also use $vars(\phi)$ to refer to the free variables that occur in ϕ; that is, the variables referenced in ϕ that are not hidden using $\exists$.

We introduce a renaming operator, which we will write as $[x/y]$, such that $\phi[x/y]$ means 'replace all references of y in ϕ with x'. The reader may have already noted that $\phi[x/y]$ is shorthand for $\exists_y(y = x \sqcup \phi)$.

2.2 Modelling Protocols

The $\mathcal{RASA}$ protocol specification language is based on process algebras, and resembles languages such as CSP [10]. However, we add the notion of *state* to the language. State is useful, because it allows us to build up the meaning of protocols compositionally, for example, the effect of sending two messages is the effect of sending the second in the state that results after sending the first. The final outcome of the protocol is the end state. A detailed presentation of the specification language, including operational semantics, is available in [16].

A protocol specification is a collection of protocol definitions of the format $N(x, \ldots, y) \mathrel{\widehat{=}} \pi$, in which $N, x, \ldots, y \in Var$, and π represents a protocol.

Let ϕ represent constraints defined in constraint language, c communication channels, N protocol names, and x a sequence of variables. Protocol definitions adhere to the following grammar.

$$\pi \quad ::= \quad \phi \rightarrow \epsilon \;\mid\; \phi \xrightarrow{c.\phi} \phi \;\mid\; \pi;\pi \;\mid\; \pi \cup \pi \;\mid\; N(x) \;\mid\; \mathbf{var}_x^{\phi} \cdot \pi$$

Protocols are defined using the two types of atomic protocol, and algebraic operators for building up compound protocols from these. The first atomic action/protocol is the empty action: $\psi \rightarrow \epsilon$. This specifies that if the precondition ψ is provable from the current state, then no message sending is required.

The second atomic protocol is message sending, $\psi \xrightarrow{c(i,j).\phi_m} \psi'$. This is read as follows: if the precondition, ψ, is provable (using $\sqsupseteq$) from the current state, then the agent i is permitted to send

the message ϕ_m to agent j. The effect of this message on the state is specified by the postcondition, ψ'. Omitting the prefix $c(i,j)$ from a message template implies that ϕ_m is an action that must be performed. In this paper, we omit (i,j) when we do not care who the sender and receiver of the message is. We use the notion of *inertia* in calculating the new state from the postcondition; that is, any variables in ψ' are constrained by ψ' in the new state, and any other variables in the state are left unchanged. We allow agents to send the message ϕ'_m, such that $\phi'_m \sqsupseteq \phi_m$, so that agents can further constrain the values of the messages; thus, ϕ_m is only a template of the message. For example, consider the following atomic protocol:

$$X \geq Bid \xrightarrow{c.bid(X)} Bid = X.$$

in which the sender is bidding on an item, and Bid and X are variables. As part of the interaction, the sender would like to instantiate X to its actual bid, for example, to 10. Therefore, it would send the message $bid(X) \sqcup X = 10$, which constrains the message template by adding further information; that is; that its bid is 10.

Compound protocols can be built up from these atomic protocols. If π_1 and π_2 are two protocols, then the following are also protocols: the protocol $\pi_1;\pi_2$, which represents sequential concatenation, such that π_1 is executed, followed by π_2; the protocol $\pi_1 \cup \pi_2$, which represents a choice between π_1 and π_2; and the protocol $\mathbf{var}_x^{\psi} \cdot \pi_1$, which is a protocol the same as π_1, except that a local variable x is available over the scope of π_1, but with the constraints ψ on x remaining unchanged throughout that scope. Any variable x already in the state is out of scope until π_1 finishes executing. In addition, $\mathcal{RASA}$ supports the referencing of protocols via their names. That is, for a protocol definition $N(x) \mathrel{\widehat{=}} \pi_1$, one can reference this from within another protocol using $N(y)$, where $y \in Var$.

Using such a definition, one can express protocols as sets of possible interactions, in which interactions are sequences of triples containing legal pre-states, messages, and post-states. This does not permit us to express concurrency: two or more messages occuring at the same time. However, we do not see this as a fatal problem with the language, because the concurrency of messages would be difficult to verify in a system using first-class interaction protocols, so would not be useful.

A key feature of this language is that it has the same syntax and semantics at all dialogue levels. Single messages are themselves protocols, and the syntax and semantics for composing two atomic protocols is the same for composing two other composite protocols. Thus, individual utterances, sequences of utterances, protocols, and combinations of protocols can all be reasoned-over, modified, composed and invoked by agents participating in an interaction using the same reasoning mechanism.

Example 2.1. We present a small example of a simple interaction in which an agent, A, proposes that another agent, B, commits to P, and B can accept or refuse this proposal.

The semantics of $\mathcal{RASA}$ is compositional, so it makes sense to present the protocol in a bottom-up manner. First, we define the $Prop$ protocol, an atomic protocol which models A sending the proposal to B:

$$Prop(A,B,P) \mathrel{\widehat{=}} \text{true} \xrightarrow{c(A,B).propose(P)} prop(P)$$

The postcondition $prop(P)$ simply indicates that the current proposal is P. The Acc and Rej protocols model B accepting or rejecting the proposal respectively:

$$Acc(A,B) \mathrel{\widehat{=}} prop(P) \xrightarrow{c(B,A).accept(P)} commit(B,A,P)$$
$$Rej(A,B) \mathrel{\widehat{=}} prop(P) \xrightarrow{c(B,A).reject(P)} \text{true}$$

The notation $commit(B, A, P)$ is a constraint representing B's commitment to perform P for the creditor A.

Finally, we compose these three atomic protocols together into a composite protocol, which defines the order that the messages must occur:

$$Prot(A, B, P) \mathrel{\widehat{=}} Prop; (Acc \cup Ref)$$

This definition enforces the condition that the proposal must be sent by A before B can accept or reject it, and that B can only send either an accept or reject, but not both. In addition, if the path $Prop; Acc$ is taken, then B is committed to P.

One can see that, provided an agent understands the meaning of $commit(B, A, P)$, such a protocol can be reasoned about at runtime. Firstly, agent A decides to use this protocol because it calculate that $commit(B, A, P)$ is an outcome. If B agrees to using the protocol, then, after it receives the proposal, it can reason that accepting the proposal will lead to the state in which it is committed to performing B. If it does not accept, then there is no change, so it can decide its reply by analysing its goals and assessing their compatibility with the outcomes.

2.3 Denotational Semantics

To define a denotational semantics for the $\mathcal{RASA}$ protocol specification language, we adopt an approach similar to the semantics Van Eijk *et al.* give for observable behaviour of their unnamed agent programming language [26]. They present observable behaviour of the set of sequences of actions performed by an agent or system of agents, paired with the resulting states of the system after each sequence. We define the semantics of the $\mathcal{RASA}$ language in a similar manner.

Definition 2.2. Observable Behaviour

We define a trace of observable behaviour as a triple, in which the first element of the triple represents the pre-state of the protocol, the second element represents a sequence of communications across channels, and the third element represents the state resulting after that sequence of communications. The observable behaviour of a $\mathcal{RASA}$ protocol is defined as the set containing all traces of observable behaviour allowed by the protocol rules.

Following the terminology of Van Eijk *et al.*, we call a sequence of communications a *history*. The set of all histories for a language, $\mathcal{L}$, is denoted as $\mathcal{H}_{\mathcal{L}}$. Using this, we define the set of observable behaviour for the language $\mathcal{L}$ as $\wp(\mathcal{L}_+ \times \mathcal{H}_{\mathcal{L}_+} \times \mathcal{L}_+)$, in which $\wp$ represents the power set function, and $\mathcal{L}_+$ represents $\mathcal{L} \setminus \{\text{false}\}$.

Definition 2.3. Compositional, Denotational Semantics

The semantics of $\mathcal{RASA}$ protocols is defined as a function $[\![D, \pi]\!] \in Env \to \wp(\mathcal{L}_+ \times \mathcal{H}_{\mathcal{L}_+} \times \mathcal{L}_+)$, in which D is the set of declarations in which the protocol π is evaluated, and Env is a function from names to sets of histories, $Env \in Name \to \wp(\mathcal{L}_+ \times \mathcal{H}_{\mathcal{L}_+} \times \mathcal{L}_+)$, used for mapping protocol reference names to their semantics. We omit D whenever it is not used. So, the semantics is a function that takes a protocol definition and an environment, returning the observable behaviour of that protocol definition within that environment.

Formally, the denotational, compositional semantics of protocols, represented as the function $[\![D, \pi]\!]$, is defined as follows:

$$
\begin{array}{lcl}
[\![\psi \rightarrow \epsilon]\!](e) & \widehat{=} & \{(\phi, \langle\rangle, \phi) \mid \phi \sqsupseteq \psi\} \\
[\![\psi \xrightarrow{c.\phi_m} \psi']\!](e) & \widehat{=} & \{(\phi, \langle c.\phi'_m\rangle, \phi') \mid (\phi \sqsupseteq \psi) \wedge (\phi'_m \sqsupseteq \phi_m) \wedge \phi' = \phi'_m \sqcup \phi \oplus \psi'\} \\
[\![\pi_1; \pi_2]\!](e) & \widehat{=} & \{(\phi, h_1 \frown h_2, \phi'') \mid (\phi, h_1, \phi') \in [\![\pi_1]\!](e) \wedge (\phi', h_2, \phi'') \in [\![\pi_2]\!](e)\} \\
[\![\pi_1 \cup \pi_2]\!](e) & \widehat{=} & [\![\pi_1]\!](e) \cup [\![\pi_2]\!](e) \\
[\![\mathbf{var}^{\psi}_{x} \cdot \pi]\!](e) & \widehat{=} & \{(\phi, h, \exists_x \phi' \sqcup \exists_{\widehat{x}} \phi) \mid (\exists_x \phi \sqcup \psi, h, \phi') \in [\![\pi]\!](e) \wedge \exists_{\widehat{x}} \phi' = \exists_{\widehat{x}}(\exists_x \phi \sqcup \psi)\} \\
[\![D, N(x)]\!](e) & \widehat{=} & \begin{array}{ll} [\![N(y)]\!](e)[x/y] & \text{if } N(y) \widehat{=} \pi \in D \\ e(N) & \text{if } N(x) \in dom(e) \\ \mu F & \text{if } N(x) \widehat{=} \pi \in D \text{ and } N(x) \notin dom(e) \end{array} \\
 & & \text{where } F(H) = [\![D, \pi]\!](e \dagger \{N(x) \mapsto H\}) \\
 & & \quad \text{and } \mu F \text{ is the } \textit{least fixpoint} \text{ of a function } F \\
 & & \quad \text{with respect to the partial order } \subseteq \\
 & & \quad \text{i.e. } \mu F = F(\mu F)
\end{array}
$$

We briefly comment on these six definitions. The empty protocol, ϵ, is defined as a set containing a single triple, in which there are no messages, and the pre- and post-state are the same. The behaviour of this is undefined if ψ does not satisfy the state ϕ. An atomic protocol is defined as the set of all triples, with the first element in the triple representing the set of constraints that satisfy the precondition, the second element representing a message, and the third element representing the resulting state. Recall from Section 2.2, that the message can be the constraint, ϕ_m, but can also be a constraint, ϕ'_m, that contains more information that ϕ_m, such that $\phi'_m \sqsupseteq \phi_m$. The resulting state is $\phi \oplus \psi'$, further constrained by ϕ'_m, in which $\oplus \in (\mathcal{L} \times \mathcal{L}) \rightarrow \mathcal{L}$ is an overriding function defined as $\phi \oplus \psi' = \psi' \sqcup \exists_{vars(\psi')}\phi$. Therefore, $\phi \oplus \psi'$ defines a new constraint such that the values of any free variables in ϕ are overridden with the values constrained by ψ', while the free variables in ϕ that are not otherwise in ψ' maintain their pre-state values. $\oplus$ has binds tighter than $\sqcup$.

Any additional information placed in the message, ϕ'_m must also apply to the resulting state. For example, recall the protocol

$$X \geq Bid \xrightarrow{c.bid(X)} Bid = X.$$

in which the sender bids on an item. If the sender constrains X in the message with $X = 10$, this information needs to be shared with the postcondition so that Bid obtains the value 10 as well. Therefore, in the semantics, we enforce the condition that constraint information must be shared between the message and the postcondition: $\phi' = \phi'_m \sqcup \phi \oplus \psi'$. In this example, the only solution for $Price$ in this constraint would be $X = 10$, therefore, the post-state is $Bid = X \sqcup X = 10$, which reduces to $Bid = 10 \sqcup X = 10$.

Sequential composition, $\pi_1; \pi_2$, of two protocols, π_1 and π_2, is defined as each history trace from π_1 concatenated with each history trace from π_2, with the post-state of π_1 substituted as the pre-state for π_2, and the post-state of π_2 being the post-state of the overall composition. Therefore, the end state of a sequential composition is computed as the functional composition of the two end states. A choice, $\pi_1 \cup \pi_2$, between two protocols, π_1 and π_2, is defined as the union of all observable behaviour from π_1 and π_2.

Variable declaration is not a straightforward definition, so we take some time to discuss it. To explain this definition, we first discuss the semantics that we want to give to $\mathbf{var}^{\psi}_{x} \cdot \pi$. Firstly, we want to execute π as normal, but within the context that there is a new variable x, constrained by ϕ. In the case that x is a free variable in the protocol state, this occurrence of x must be hidden.

Both during and after the execution of π, we do not want the constraints over the local variable x to change. Finally, once π is executed, we want the local occurrence of x to be hidden, and the previous free variable x and its constraints to be restored to the state.

To discuss how this definition achieves the above, it is best to start in the middle at $(\exists_x \phi \sqcup \psi, h, \phi') \in [\![\pi]\!](e)$. We execute the sub-protocol π from the state $\exists_x \phi \sqcup \psi$. Here, any free occurrences of variable x in ϕ are hidden in case x is already a variable in this constraint. Once π is executed, we are left with the post-state ϕ'. The condition $\exists_{\widehat{x}} \phi' = \exists_{\widehat{x}} (\exists_x \phi \sqcup \psi)$ specifies that the constraints over x in the post-state are equal to the constraints over x in the pre-state. Finally, we calculate the new post-state of the entire protocol. We want to hide the local variable x and its constraints, so we have $\exists_x \phi'$, but we also want to re-introduce the previous variable x, so we conjoin this with $\exists_{\widehat{x}} \phi$, which specifies that we hide occurrences of all free variables in the pre-state, ϕ, except x. This restores the variable x to the state, but with the constraints from the pre-state.

The semantics for a protocol reference is also not straightforward, due to the fact that we allow recursive definitions; that is, protocol trees that reference themselves, or each other. Without recursive definitions, the semantics would specify that a reference is replaced with its protocol definition in D, and then renamed. However, for recursive definitions, we use *fixpoints* to define the semantics. Defining such a semantics is not a straightforward task; fortunately, De Boer *et al.* [5] have already solved most of this problem for us.

The definition is divided into three cases. Note that we assume some form of correctness in this definition: that a name reference is a valid protocol name in D. If we remove this assumption, one needs only to add a fourth case saying that the protocol is equivalent to some error state, but we omit this. In the first case, the name, N, of the referenced protocol is in the set of declarations, but the variables are mismatched. Therefore, the behaviour of $N(x)$ is equivalent to $N(y)$, but with all references of x renamed to y. In the second case, N is in the environment, so we return the semantics of N, denotes $e(N)$, from that environment. In the third case, $N(x)$ is in the definitions with matching variables, but not in the environment. A function, $F(H)$, is constructed, which gives the semantics of the protocol π with the environment that is the same as e, but with $e(N)$ overridden with H. Then, *least fixpoint* of F, denoted μF, with regards to the partial order $\subseteq$, is the denoted value of N.

2.4 Shorthand Notation

We introduce additional operators that can be defined as shorthand in terms of the primitive operators defined above. The most notable of these shorthand operators is equivalent to the primitive iteration operator found in dynamic logics [9] and Kleene algebras [11], and is written π^*. This defines a protocol that iterates over the sub-protocol π zero or more times. Formally, this is defined as follows:

$$[\![D, \pi^*]\!](e) \quad \widehat{=} \quad [\![D', N]\!](e) \quad \text{where } D' = D \cup \{N \mathrel{\widehat{=}} \epsilon \cup (\pi; N)\}$$

That is, π^* is equivalent to the protocol named N, where N is defined as a choice between the empty protocol or the sequential composition of π followed by a recursive call to N.

We define this as a shorthand rather than a primitive because the class of protocols describable using recursively definable protocols (via names) is a superset of those using iteration with no names, which allows only *regular* protocols. Therefore, iteration is easily simulated using recursively defined protocols.

Additional redundant operators found in dynamic logics and Kleene algebras can similarly be defined, such as π^+, which is defined as $\pi; \pi^*$. We also use an interleaving operator, $\|$, which

interleave that traces from two protocols. Such an operator can be expressed as a choice between all possible interleavings. For the protocol true $\rightarrow \epsilon$, we omit the implication and simply write ϵ.

3 Protocol Entailment

So far in this paper, we have presented the $\mathcal{RASA}$ language for specifying protocols. Rules and effects of protocols are specified using a constraint language, $\mathcal{L}$. A constraint from $\mathcal{L}$ can be used to represent the constraints on the current protocol state.

However, we see the advantages of a logic that can be used to specify the effects of composite protocols; that is, the end states of entire protocols, rather than just atomic protocols. In this section, we define such a logic — that is, a syntax, semantics, and proof system — which can be used to annotate protocol specifications, as well as to reason about the effects that protocols have.

We view this logic as an instantiation of propositional dynamic logic (PDL) [9]. By *instantiation*, we mean that we use $\mathcal{RASA}$ protocols in place of abstract programs, and constraints in place of models in which predicates are interpreted. Operators used to compose complex programs in PDL correspond to protocol composition operators in $\mathcal{RASA}$, except for the test operator in PDL, which we considered unnecessary in $\mathcal{RASA}$ because it can be represented using logical implication. This logic is referred to as $\mathcal{L}_\pi$.

3.1 Protocol Outcomes and Annotation

The primary motivations for this logic are: (1) to annotate protocols with their outcomes; and (2) to reason about the which paths of interaction within a protocol best achieve the agent's goals.

Considering that a protocol is a tree-like structure, it seems that a modal logic containing an modality for expressing properties of outcomes would be useful. For example, annotating the head of a protocol with the formula $\Box\phi$ specifies that ϕ holds for every possible outcome of that protocol. However, we feel that this is not expressive enough, especially to achieve our second goal. For example, this says nothing about the intermediate states in the protocol, nor does it allow us to say anything about the individual paths in the protocol, meaning that annotations offer little support for agents deciding which paths best achieve their goals.

To overcome this weakness, our logic permits us to index the modal operator with the definition of the protocol for which the property holds, in the same way that dynamic logic operators are indexed with programs. So, for a protocol π, the formula $[\pi]\phi$ specifies that ϕ holds for every possible outcome of π. This indexing allows us to express properties about paths and sub-protocols; for example, for the protocol $\pi_1;(\pi_2 \cup \pi_3)$, we can express properties about the entire definition, as well as the paths $\pi_1;\pi_2$ and $\pi_1;\pi_3$. In addition, it allows us to express properties about intermediate states of the protocol, rather then only outcomes, by taking the intermediate state as the outcome of one of the sub-protocols. That is, the $[\pi_1]\phi$ specifies a property about the state after π_1, but before the choice $\pi_2 \cup \pi_3$. Indexing with protocols is key to this, otherwise there is no way to reference that the formula $\Box\phi$ is only referring to outcomes of the intermediate state.

A key reason for deriving the logic $\mathcal{L}_\pi$ is to annotate protocols with their outcomes, including both outcomes that they do achieve, and outcomes that they can possibly achieve. Although this information can be derived from information in the protocol definitions, it is safe to assume that, at runtime, the overhead of calculating outcomes is impractical for more than a handful of protocols, therefore, we want to reduce the amount of calculation for agents as much as possible.

In other work [14], we have investigated methods for deriving annotations from protocol specifications, and for searching protocol libraries to find a protocol, using annotations on protocols, that

achieves a stated goal. The proof system for $\mathcal{L}_\pi$, defined in Section 4, is used to prove that our method for annotation is sound and complete.

3.2 Syntax of $\mathcal{L}_\pi$

In this section, we introduce the logic $\mathcal{L}_\pi$, which is used to reason about the outcomes of protocols. $\mathcal{L}_\pi$ is built on $\mathcal{L}$, in that any constraint $\phi \in \mathcal{L}$ is also in $\mathcal{L}_\pi$. To avoid ambiguity, we distinguish the two languages by using the term *predicate* to refer to properties about protocols, while continuing to use the term *constraints* to refer to a member of the language $\mathcal{L}$. We use ϕ and ψ to represent both predicates and constraints, however, we subscripts constraints with a number, that is ϕ_0, to indicate that the ϕ_0 is in $\mathcal{L}$, but not in $\mathcal{L}_\pi$.

The set of well-formed formulae of $\mathcal{L}_\pi$ is defined by the following grammar:

$$\phi \quad ::= \quad \phi_0 \;\mid\; \phi \wedge \phi \;\mid\; \neg\phi \;\mid\; [\pi]\phi$$

Each predicate of these forms are evaluated within a protocol state; that is, a constraint. In the this grammar, ϕ_0 is a constraint from $\mathcal{L}$, and is true if it is entailed by the protocol state. $\wedge$ and $\neg$ take on the usual meanings. The final branch of the grammar contains the interesting operator. The meaning of the predicate $[\pi]\phi$ is as follows: at the current state of the protocol, if we were to enter into the protocol π, then at every *end state* of the protocol π, ϕ would hold. We call this *protocol entailment* (executing this protocol entails that a certain predicate will hold). This operator is analogous to the same operator found in dynamic logic. Brackets are used to group predicates, and similar to $\mathcal{L}$, $\neg$ and [ ] bind tighter than $\wedge$, therefore $[\pi]\phi \wedge \psi$ is equivalent to $([\pi]\phi) \wedge \psi$.

Protocol entailment therefore allows us to specify properties and to reason about possible future outcomes of protocols, which gives us more flexibility and power than reasoning only about the current state of protocols. However, we view these properties as side effects of protocols. That is, they are not used to give definitions of protocols by constraining the outcomes, but instead specify a property that holds for a set of possible future interactions given the current state. Therefore, the truth of any formula $[\pi]\phi$ can be established directly from the definition of π itself.

3.3 Semantics of $\mathcal{L}_\pi$

Our logic is based on PDL, so we use standard terminology from dynamic logic to name our structures. Predicates and constraints within our framework only have meaning when evaluated within a *model*. A model is a pair $(\mathfrak{K}, \psi_0)$, in which $\mathfrak{K}$ is the *frame*, and ψ_0 is the *state*. A frame is a pair $(\mathcal{L}, D)$, in which $\mathcal{L}$ is the set of possible states (in our case, all constraints), and D is the set of definitions of named protocols. In PDL, one would say that D represents the meaning function, which assigns meanings to atomic programs. However, because the meaning of an atomic protocol is directly derivable from its definitions, our meaning functions are only functions from named protocols to their definition, which can be atomic or compound. Using these definitions, $\mathfrak{K}, \psi_0 \models \phi$ means that ψ_0 satisfies ϕ under D. Throughout this paper, we omit $\mathfrak{K}$ because the context of $\mathcal{L}$ and D is clear. If ϕ is true in every $\mathfrak{K}$ and for every state $\phi_0 \in \mathcal{L}$, then we write $\models \phi$ and say that ϕ is *valid*. $\psi_0 \not\models \phi$ means that ψ_0 does not satisfy ϕ. Using these definitions, the semantics of $\mathcal{L}_\pi$ is defined as follows:

$$\begin{array}{lll}
\psi_0 \models \phi_0 \text{ where } \phi_0 \in \mathcal{L} & \text{iff} & \psi_0 \sqsupseteq \phi_0 \\
\psi_0 \models \phi \wedge \psi & \text{iff} & \psi_0 \models \phi \text{ and } \psi_0 \models \psi \\
\psi_0 \models \neg\phi & \text{iff} & \psi_0 \not\models \phi \\
\psi_0 \models [\pi]\phi & \text{iff} & \forall(\psi_0, h, \psi_0') \in [\![\pi]\!](\emptyset) \bullet \psi_0' \models \phi
\end{array}$$

We comment briefly on each of these definitions. For ϕ_0 to be true within a model ψ_0, it must be entailed from ψ_0 using $\sqsupseteq$. $\phi \wedge \psi$ is true within a model if and only if both ϕ and ψ are both true within the model, while $\neg\phi$ is true if and only if ϕ is not. Finally, $[\pi]\phi$ is true if and only if, starting from the model ψ_0, for every end state, ψ_0', of the protocol, π, ϕ holds. That is, for any history of the protocol π, ϕ will hold in every end state of that history.

One can see that, given a model ψ_0 and constraints ϕ_0 and ϕ_0', the truth of $\psi_0 \models \phi_0 \wedge \phi_0'$ is equivalent to the truth of $\psi_0 \sqsupseteq \phi_0 \sqcup \phi_0'$, and the truth of $\psi_0 \models \neg\phi_0$ is equivalent to the truth of $\psi_0 \sqsupseteq \neg\phi_0$.

3.4 Shorthand Notation

Standard logical operators such as disjunction ($\vee$), implication ($\rightarrow$), and equivalence ($\leftrightarrow$) are defined for protocol entailment predicates using $\neg$ and $\wedge$. In the true spirit of modal and dynamic logic, we introduce a dual to the [ ] operator: $\langle\ \rangle$. Used in the same context as its dual, $\langle\pi\rangle\phi$ means, at the current state of the protocol, if we were to enter into the protocol π, then for *at least one* end state of the protocol π, ϕ would hold. The semantics for this is defined as follows:

$$\psi_0 \models \langle\pi\rangle\phi \quad \text{iff} \quad \exists(\psi_0, h, \psi_0') \in [\![\pi]\!](\emptyset) \bullet \psi_0' \models \phi$$

However, like many other modal operators, this primitive definition is not required to give a semantics. Instead, the $\langle\ \rangle$ operator can be defined in terms of the [ ] operator, as follows:

$$\psi_0 \models \langle\pi\rangle\phi \quad \text{iff} \quad \psi_0 \models \neg[\pi]\neg\phi$$

That is, ϕ is true for at least one end state of the protocol π if and only if it is not the case that $\neg\phi$ is true at all end states of the protocol π. This shorthand definition can be easily shown to be equivalent to the primitive definition. Expanding $\neg[\pi]\neg\phi$ gives us the following:

$$\psi_0 \models \neg\forall(\psi_0, h, \psi_0') \in [\![\pi]\!](\emptyset) \bullet \psi_0' \models \neg\phi$$

We know from first-order logic axioms that $\forall x \bullet P(x) \iff \neg\exists x \bullet \neg P(x)$, so substituting the above into this equivalence gives us the following:

$$\psi_0 \models \neg\neg\exists(\psi_0, h, \psi_0') \in [\![\pi]\!](\emptyset) \bullet \psi_0' \not\models \neg\phi$$

Eliminating the double negation, this is trivially equivalent to the primitive definition of $\langle\rangle$.

Example 3.1. Recall the specification from Example 2.1, in which an agent requests that another agent commits to some action. If we assume that the constant *self* refers to an agents representation of itself, then the following specifes that, if the the agent accepts the request, it is committed to it:

$$S = self \rightarrow [Acc]commit(self, A, P)$$

In the case of the entire protocol, this is not the case for every outcome. However, it is the case for at least one – the one in which the agent accepts a bid. This can be represented using the following formula:

$$S = self \rightarrow \langle Prot \rangle commit(self, A, P)$$

Therefore, if an agent has a goal of having P performed, but cannot perform P itself, it can match the above annotation with its goal, and use this protocol to request that another agent performs P. The other participant, when it has the choice of rejecting or accepting the request, can use the annotation on the *Acc* sub-protocol to calculate that, it it accepts, it will be committed to performing P.

While these examples are somewhat trivial, one can see that, for any protocols with more than a handful of interactions, or a protocol library containing more than a handful of definitions, documenting outcomes is useful.

3.5 Expressiveness

We have already briefly discussed the expressiveness of this logic in Section 3.1, but here we discuss expressiveness again now that we have introduced the syntax and semantics of the language. For this, we consider a protocol as a tree, like a game tree, in which nodes represent states, arcs represent messages, and brances of more than one arc represent choices.

For any collection of nodes in a tree, the logic allows us to specify a property that holds for all of those nodes. This property is also specified in the logic. This is straightforward to show. If ϕ is the property one wishes to express, and $\{n_1, \ldots, n_m\}$ the collection of nodes about which one wants to specify that ϕ holds, then one can specify that ϕ holds at each of these nodes by taking taking the path to each node as a sequential composition, and expressing that ϕ holds at the end of each of these compositions. For example, to specify that ϕ holds in every node of $\pi_1; (\pi_2 \cup \pi_3)$ (that is, ϕ is an invariant of the protocol), one would write

$$\phi \wedge [\pi_1]\phi \wedge [\pi_1; \pi_2]\phi \wedge [\pi_1; \pi_3]\phi.$$

One shortfall of the logic is that it does not allow us to express the number of percentage of end states for which a property holds, but only that it holds for all, at least one, or none. Using the definition of a constraint system in Section 2 means that this is not possible, because some constraints systems may permit reasoning over infinite domains, for example, the integers. Therefore, the number of possible instantiations of a message (recall that agents are permitted to constrain the message template in a protocol), and as a result, the number of possible outcomes, may be uncountable. This is an unfortunate restriction, but one which allows us to consider a greater number of constraint systems.

We also note that one cannot express properties about the messages that are sent. This is a deliberate omission from the logic, because we are interested only in documenting outcomes of protocols. Additional operators that consider which messages are possible in a protocol could be added, however, for goal-directed agents using first-class protocols, we do not believe they are of great interest, because the meaning of the message is specified by its relative post-state.

4 Proof System for $\mathcal{L}_\pi$

In this section, we define a deductive proof system for $\mathcal{L}_\pi$. By this, we mean that we present axioms and inference rules for $\mathcal{L}_\pi$, and prove that this system is sound and complete with respect to the semantics defined in Section 3. This proof system allows us to prove statements made in $\mathcal{L}_\pi$, which is useful for proving properties about protocols, and for proving that annotations derived for protocols are sound, as in [14].

We write $\psi_0 \vdash \phi$ to indicate that ϕ is provable in the state ψ_0 in this proof system, in which the state ψ_0 is in $\mathcal{L}$. This is constrast to the usual reading, which would be that ϕ is true in all models that ψ_0 holds. Our models are made up of a frame and a state, and we believe that proofs and annotations would be made with respect to eiter a particular frame or all frames, therefore we prefer the interpretation that ψ_0 represents the current state. $\psi \vdash \phi$, where ψ is not in $\mathcal{L}$, would unambiguously represent the standard reading because ψ cannot be a state. We use $\vdash \phi$ to indicate that ϕ is provable in all models in this proof system (ϕ is a theorem).

The proof system is defined such that, every proof of a predicate in $\mathcal{L}_\pi$ reduces to a proof in $\mathcal{L}$. That is, every proof of the form $\psi_0 \vdash \phi$ reduces to a proof of the form $\psi_0 \sqsupseteq \phi_0$. This is a useful result because it implies that, given the entailment operator for $\mathcal{L}$, $\sqsupseteq$, and the above axioms, agents can prove properties about protocols without other machinery. It also means that automated support for proving predicates of $\mathcal{L}_\pi$ would be straightforward to implement, depending on the support available for proving properties of $\mathcal{L}$.

4.1 Axiomatisation of $\mathcal{L}_\pi$

Because we have not specified a particular underlying constraint language, we cannot provide a complete set of axioms. Instead, we state simply that any axioms of $\mathcal{L}$ are also axioms in our proof system. However, we do assume that the underlying constraint language satisfies the properties of a lattice, as described by De Boer *et al.* [5], so we assume several axioms over $\mathcal{L}$, such as De Morgan's laws and the law of double negation, *et cetera*, and also that $\mathcal{L}$ is sound and complete.

In addition to the axioms for $\mathcal{L}$, we propose the following additional set of axiom schemas for the proof system:

(i)	$[\pi](\phi \wedge \psi)$	$\leftrightarrow$	$[\pi]\phi \wedge [\pi]\psi$	(conjunction axiom)
(ii)	$[\psi_0 \rightarrow \epsilon]\phi$	$\leftrightarrow$	$\psi_0 \rightarrow \phi$	(empty protocol axiom)
(iii)	$[\psi_0 \xrightarrow{c.\phi_m} \psi_0']\phi$	$\leftrightarrow$	$\psi_0 \rightarrow (\phi_m \sqcup \psi_0' \rightarrow \phi)[y/x]$	(atomic protocol axiom) where $x = vars(\phi_m \sqcup \psi_0')$ and y fresh
(iv)	$[\pi_1;\pi_2]\phi$	$\leftrightarrow$	$[\pi_1][\pi_2]\phi$	(sequential comp. axiom)
(v)	$[\pi_1 \cup \pi_2]\phi$	$\leftrightarrow$	$[\pi_1]\phi \wedge [\pi_2]\phi$	(choice axiom)
(vi)	$[\mathbf{var}_x^{\psi_0} \cdot \pi]\phi$	$\leftrightarrow$	$x_0 = x \wedge \psi_0 \rightarrow [\pi](x_0 = x \rightarrow \phi)$	(local variable axiom)[1]
(vii)	$[N(x)]\phi$	$\leftrightarrow$	$[\pi[x/y]]\phi$	(protocol name axiom) where $N(y) \mathrel{\widehat{=}} \pi$

Theorem 4.1. *Given any ϕ in $\mathcal{L}_\pi$, ϕ can be reduced to ϕ_0 in $\mathcal{L}$, using the axioms, such that $\psi_0 \models \phi \iff \psi_0 \models \phi_0$ for any ψ_0*

Proof. This can be proved using induction over the structure of ϕ. We assume that the axioms are valid theorems of $\mathcal{L}_\pi$, which is discussed further in Theorem 4.2.

There are two base cases for the induction: $[\psi_0 \rightarrow \epsilon]\phi$ and $[\psi_0 \xrightarrow{c.\phi_m} \psi_0']\phi$. Axioms 4.1(ii) and 4.1(iii) respectively can be applied from left to right, removing the [ ] operators. Applying the induction hypothesis to ϕ in both cases yields a result that is in $\mathcal{L}$.

[1] For simplicity, we assume that x and x_0 are fresh variables — that is, they are not free variables in the model. If this is not the case, x must be renamed to a fresh variable on the right-hand side of the equivlance, and a fresh name must be used for x_0.

For the cases of sequential composition, choice, and variable declaration, one can apply Axioms 4.1(iv)-(vi) respectively, from left to right. Protocols form a finite tree structure, and these axioms break each proof into their child trees until we are left only with the empty and atomic protocols (the base cases). The only tricky step is for name references, which can introduce cycles when Axiom 4.1(vii) is applied. In this case, the axiom is only applied if this has not been already, otherwise it is removed.

Finally, we are left with the cases of conjunction and negation. Using the induction hypothesis, reduce the arguments in these cases to constraints, and then replace $\wedge$ with $\sqcup$, which is equivalent. Negation is given the same semantics in both languages, so leave this as is. $\square$

This is a useful result, because it shows that, given any predicate, ϕ, a proof of ϕ can be reduced to a proof of a constraint. This means that an agent can prove properties about protocol entailments using only the axioms and the constraint solver. It also has implications regarding the completeness of $\mathcal{L}_\pi$.

4.2 Rules of Inference

In addition to the axioms, the system includes two inference rules; that of *modus ponens*:

$$\text{if} \quad \psi_0 \vdash \phi \text{ and } \psi_0 \vdash \phi \rightarrow \psi \quad \text{then} \quad \psi_0 \vdash \psi$$

which states that if ϕ is provable in ψ_0, and if $\phi \rightarrow \psi$ is provable in ψ_0, then one can infer that ψ is provable in ψ_0; and that of *necessitation*:

$$\text{if} \quad \vdash \phi \quad \text{then} \quad \vdash [\pi]\phi$$

which states that if ϕ holds in every model, then it must also hold at every end state representing a model in any protocol π. Dynamic logics, such as that defined by Harel *et al.* [9], generally contain a generalisation inference rule, and this rule is trivially valid in $\mathcal{L}_\pi$, however, taking into account Theorem 4.1, it is unnecessary because $[\pi]\phi$ can be reduced to a constraint in $\mathcal{L}$.

These rules are sound with respect to the logic. Modus ponens is trivially justified, and necessitation is justified by noting that if $\vdash \phi$, then ϕ is true in every model, including any end state of any protocol.

4.3 Soundness and Completeness of $\mathcal{L}_\pi$

A logic is *sound* is any predicate that is provable in it, is true. A logic is *complete* if any true predicate, is provable.

Theorem 4.2. *$\mathcal{L}_\pi$ is sound and complete*

Proof. We claim that our logic is sound and complete with respect to the semantics of $\mathcal{L}_\pi$. Taking the above definition of *sound*, it follows that our logic is sound if we cannot prove any predicate that it is not true, therefore, in our axiomatic system, soundness is demonstrated by proving that each of the axioms are theorems. Appendix A contains such a proof.

Our logic is complete if we can prove any true predicate in our logic. A completeness proof for $\mathcal{L}_\pi$ is quite straightforward, because we have built the logic on a constraint system, which we assume is sound and complete.

To prove completeness, we have to demonstrate that there exists a proof for any true predicate. To do this, we use Theorem 4.1, which states that every predicate in $\mathcal{L}_\pi$ can be reduced to a constraint in $\mathcal{L}$, such that $\psi_0 \models \phi \iff \psi_0 \models \phi_0$ for any ψ_0. Taking this result into account, and our assumption that $\mathcal{L}$ is complete, we can construct a proof for ϕ by reducing it to a constraint, and proving the constraint using the entailment operator of $\mathcal{L}$. The soundness of the logic ensures that the reduction is correct, so the proof is also correct. Theorem 4.1 states that the reduction holds for any predicate, so there exists a proof for any true predicate, and therefore, our logic is complete. □

4.4 Additional Theorems and Relation to PDL

Dynamic logics, such as the propositional dynamic logic defined by Harel *et al.* [9], contain additional axioms that are not reflected in the $\mathcal{L}_\pi$ axioms. These axioms are omitted from the $\mathcal{RASA}$ proof system because they can be derived from the minimal set of axioms and inference rules of $\mathcal{L}_\pi$. The following theorem states that the following PDL axioms missing in the axiomatisation of $\mathcal{L}_\pi$ are theorems of $\mathcal{L}_\pi$.

Theorem 4.3. The following forumlae are valid theorems of $\mathcal{L}_\pi$:

(i)	$[\pi](\phi \rightarrow \psi)$	$\rightarrow$	$[\pi]\phi \rightarrow [\pi]\psi$
(ii)	$\phi \wedge [\pi][\pi^*]\phi$	$\leftrightarrow$	$[\pi^*]\phi$
(iii)	$\phi \wedge [\pi^*](\phi \rightarrow [\pi]\phi)$	$\rightarrow$	$[\pi^*]\phi$

Proof. See the extended technical report version of this paper [15]. □

This is a useful result because, having shown that all of the axioms of PDL are valid theorems in $\mathcal{L}_\pi$, any provable theorem of PDL is also a provable theorem of $\mathcal{L}_\pi$.

5 Related Work

As far as the author are aware, there has been no work to date regarding annotation or selection of first-class protocols. There has, however, been much work done on interaction languages, and first-class protocol languages.

Process algebras, such as CSP [10], CCS [17], and the π-calculus [18] have been used to model processes and their interactions. While the combination of processes can form the basis of a protocol specification, these languages have no notion of state, so cannot specify protocol meaning, and meaning is important to goal-directed agents. Languages such as Object-Z/CSP [24], which mixes process algebras with state-based languages, are often too heavy for designers and agents alike.

There are a handful of languages that have been used for first-class protocol specification. Various authors have had success with approaches based on Petri Nets [6] and on declarative specification languages [7, 8, 28], as well as an algebraic language similar to $\mathcal{RASA}$ called the Lightweight Coordination Calculus [21]. [13] presents a detailed comparison of these languages, including $\mathcal{RASA}$, so we do not cover this here.

Viroli and Ricci [27] propose a method for formalising *operating instructions* for use on mediating coordination artifacts. Sequences of operation instructions resemble our first-class protocols, however their language does not provide the necessary constructs to document the meaning of

protocols. In addition, as Viroli and Ricci explicitly comment that their goals are to provide a methodology for environment-based coordination, rather than to provide a general approach to agent interaction semantics.

De Boer *et al.* [3] present a language with similar syntax, semantics, and assumptions about underlying languages. However, like many interaction modelling languages, the interaction is emergent from the model of the participants, rather than being first class.

Serrano *et al.* [23] describe a multi-agent programming framework in which interactions are represented by first-class objects. These objects assert some control over message passing at runtime to guide the interaction. However, this requires the identification of roles at design time, and appears to force participating agents to implement certain interfaces, which we explicitly aim to prevent.

There is also work related to protocol composition in the agent communications literature. McBurney and Parsons [12] propose a formalism for composing dialogue game protocols, which enables similar types of composition, but over a more restricted class of protocols. Reed *et al.* [20] present a framework which allows agents to assign meanings to messages at run-time, and thus, indirectly, to create new interaction protocols.

Propositional dynamic logic (PDL) [9] is clearly related to our work. We view the $\mathcal{L}_\pi$ logic as an instantiation of PDL, as discussed in Section 3. However, there has been some work using PDL to specify interaction protocols. PDL has been extended [19] with belief and intention modal operators to define a language, PDL-BI, for modelling agent interaction. PDL has also been used directly as a protocol specification language [4]. The main difference between this work and our approach is that we use an instantiation of PDL for annotation and reasoning about protocols constructed in the $\mathcal{RASA}$ language, in which the protocol definitions themselves provide the allowed behviour, while the work in [19] and [4] define a protocol as a collection of PDL predicates, which would make them difficult to use as first-class protocol languages, especially PDL-BI, which is based on ungrounded BDI logic. Other modal and temporal logics [2] that discuss future outcomes are related, but none of these support protocol (or programs) referencing in the way that PDL and $\mathcal{L}_\pi$ do.

6 Conclusions and Further Work

In this paper, we present a logic, $\mathcal{L}_\pi$, for reasoning about and annotating protocols in $\mathcal{RASA}$, a framework for executable protocol specification. This logic is built upon the protocol specification language, and an underlying constraint language. $\mathcal{L}_\pi$ contains a language, semantics, and deductive proof system. Proofs in $\mathcal{L}_\pi$ can be reduced to proofs in the underlying constraint language, meaning that agents can discharge proofs about protocols themselves using the axioms of the proof language and the entailment operator of constraint language. This result is useful for agents when composing new protocols from existing ones. The logic $\mathcal{L}_\pi$ is an instantiation of propositional dynamic logic, which is useful because it allows us to take advantage of a collection of sound work on dynamic logic.

By treating interaction protocols as first-class entities, $\mathcal{RASA}$ permits protocols to be dynamically inspected, referenced, composed, and shared by ever-changing collections of agents engaged in interaction. The task of protocol selection and invocation may thus be undertaken by agents rather than agent-designers, acting at run-time rather than at design-time. Frameworks such as this will be necessary to achieve the full vision of multi-agent systems.

Before such visions are realised, significant further work is required. We aim to identify the conditions that must hold for two protocols to be composed; for example, $\pi_1; \pi_2$ is only a valid composition if the precondition of π_2 is enabled by the postcondition of π_1. The logic presented in

this paper can be used to verify that such properties hold for protocols. In addition, meta-protocols are needed that allow agents to propose and negotiate which protocols are to be used, and suitable protocols for doing so will be investigated. To develop and test these ideas, we plan a prototype implementation in which agents negotiate the exchange of information using protocols specified using the $\mathcal{RASA}$ framework.

References

[1] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation, Applications*. Cambridge University Press, 2003.

[2] P. Blackburn, J. van Benthem, and F. Wolter, editors. *Handbook of Modal Logic. North Holland*. Elsevier, 2006.

[3] F. S. Boer, W. de Vries, J-J. Ch. Meyer, R. M. van Eijk, and W. van der Hoek. Process algebra and constraint programming for modelling interactions in MAS. *Applicable Algebra in Engineering, Communication and Computing*, (16):113–150, 2005.

[4] R. L. Brak, J. D. Fleuriot, and J. McGinnis. Theorem proving for protocol languages. In *Proceedings of the European Union Multiagent Systems Workshop*, 2004.

[5] F. S. De Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. *ACM Transactions on Programming Languages and Systems*, 19(5):685–725, September 1997.

[6] L. P. de Silva, M. Winikoff, and W. Liu. Extending agents by transmitting protocols in open systems. In *Proceedings of the Challenges in Open Agent Systems Workshop*, Melbourne, Australia, 2003.

[7] N. Desai, A. U. Mallya, A. K. Chopra, and M. P. Singh. OWL-P: A methodology for business process modeling and enactment. In *Workshop on Agent Oriented Information Systems*, pages 50–57, July 2005.

[8] N. Desai and M. P. Singh. A modular action description language for protocol composition. In *Proceedings of the 22nd Conference on Artificial Intelligence (AAAI)*. AAAI Press, 2007. (To appear).

[9] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[11] D. Kozen. On Kleene algebras and closed semirings. In B. Rovan, editor, *Proceedings of Mathematical Foundations of Computer Science*, volume 452 of *LNCS*, pages 26–47. Springer, 1990.

[12] P. McBurney and S. Parsons. Games that agents play: A formal framework for dialogues between autonomous agents. *Journal of Logic, Language and Information*, 11(3):315–334, 2002.

[13] J. McGinnis and T. Miller. Amongst first-class protocols. In *Engineering Societies in the Agents World VIII*, LNAI, 2007. (To Appear).

[14] T. Miller and P. McBurney. Annotating and matching first-class agent interaction protocols. Technical Report ULCS-07-???, University of Liverpool, Department of Computer Science, 2007.

[15] T. Miller and P. McBurney. Executable logic for reasoning and annotation of first-class agent interaction protocols. Technical Report ULCS-07-015, University of Liverpool, Department of Computer Science, 2007.

[16] T. Miller and P. McBurney. Using constraints and process algebra for specification of first-class agent interaction protocols. In G. O'Hare, A. Ricci, M. O'Grady, and O. Dikenelli, editors, *Engineering Societies in the Agents World VII*, volume 4457 of *LNAI*, pages 245–264, 2007.

[17] R. Milner. *A Calculus of Communicating Systems*. Springer Verlag, 1980.

[18] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[19] S. Paurobally, J. Cunningham, and N. R. Jennings. A formal framework for agent interaction semantics. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 91–98. ACM Press, 2005.

[20] C. Reed, T. J. Norman, and N. R. Jennings. Negotiating the semantics of agent communications languages. *Computational Intelligence*, 18(2):229–252, 2002.

[21] D. Robertson. Multi-agent coordination as distributed logic programming. In *Proceedings of the International Conference on Logic Programming*, volume 3132 of *LNCS*, pages 416–430. Springer, 2004.

[22] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pages 333–352. ACM Press, 1991.

[23] J. M. Serrano, S. Ossowski, and S. Saugar. Reusable components for implementing agent interactions. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Programming Multi-Agent Systems, Third International Workshop*, pages 101–119, 2005.

[24] G. Smith and J. Derrick. Abstract specification in Object-Z and CSP. In C. George and H. Miao, editors, *Formal Methods and Software Engineering*, volume 2495 of *LNCS*, pages 108–119. Springer, 2002.

[25] C. Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1):11–49, April 2000.

[26] R. M. van Eijk, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. A verification framework for agent communication. *Autonomous Agents and Multi-Agent Systems*, 6(2):185–219, 2003.

[27] M. Viroli and A. Ricci. Instructions-based semantics of agent mediated interaction. In N. R. Jennings, C. Sierra, L. Sonenberg, and M. Tambe, editors, *Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 102–109. IEEE Computer Society, 2004.

[28] P. Yolum and M. P. Singh. Reasoning about commitments in the event calculus: An approach for specifying and executing protocols. *Annals of Mathematics and AI*, 42(1–3):227–253, 2004.

A Soundness Proof for $\mathcal{L}_\pi$

In this appendix, we prove that the proof system for $\mathcal{L}_\pi$ is sound. That is, we prove that, for any predicate ϕ in $\mathcal{L}_\pi$, if ϕ is provable in state ψ_0, then ϕ is true in state ψ_0:

$$\psi_0 \vdash \phi \implies \psi_0 \models \phi$$

Soundness is proved by showing that each of the axioms proposed in Section 4 is valid, except for the assumed axioms for $\mathcal{L}$, for which the axioms are already assumed to be valid.

Definition A.1. Upward Closure of Constraints

Throughout this proof, we make use of the following shorthand. For a constraint ϕ_0, we define the *upward closure* of that constraint, written $\uparrow\phi_0$, as being every constraint from which ϕ_0 can be proven using the entailment relation of $\mathcal{L}$. This is defined as follows:

$$\uparrow\phi_0 \quad = \quad \{\psi_0 \in \mathcal{L} \mid \psi_0 \sqsupseteq \phi_0\}$$

Therefore, $\psi_0 \in \uparrow\phi_0$ iff $\psi_0 \sqsupseteq \phi_0$. For example, if we assume that the possible values of the variable *Price* range over the natural numbers, the upward closure of the constraint $Price \leq 10$ would be any constraint in which the possible values of *Price* are between 0 and 10 inclusive. Therefore, $Price \leq 9$, $Price \in [0..5]$, and $Price = 0$ are all members of the upward closure of $Price \leq 10$.

Theorem A.1. *$\mathcal{L}_\pi$ is sound*

Proof. To prove the soundness of $\mathcal{L}_\pi$, we prove that each of the axioms in the proof system is valid.

Axiom (i) $$\models \quad [\pi](\phi \wedge \psi) \quad \leftrightarrow \quad [\pi]\phi \wedge [\pi]\psi$$

First, we prove the equivalence from right to left. That is, we prove $[\pi]\phi \wedge [\pi]\psi \rightarrow [\pi](\phi \wedge \psi)$. Consider arbitrary start and end states ψ_0 and ψ_0' of protocol π. From the premise, we know that $\psi_0' \models \phi$ and $\psi_0' \models \psi$. From the definition of $\wedge$, we know that $\psi_0' \models \phi \wedge \psi$. Because ψ_0' is an arbitrary end state, then this must be true for all end states, therefore, $\psi_0 \models [\pi](\phi \wedge \psi)$. ψ_0 is arbitrary, so this holds for every state.

Now, we prove the equivalence from left to right. That is, we prove $[\pi](\phi \wedge \psi) \rightarrow [\pi]\phi \wedge [\pi]\psi$. Consider arbitrary start and end states ψ_0 and ψ_0' of protocol π. From the premise, we know that $\psi_0' \models \phi \wedge \psi$. From the definition of $\wedge$, we know that $\psi_0' \models \phi$ and $\psi_0' \models \psi$. Because ψ_0' is an arbitrary end state, then this must be true for all end states, therefore, $\psi_0 \models [\pi]\phi$ and $\psi_0 \models [\pi]\psi$, which is equivalent to $\psi_0 \models [\pi]\phi \wedge [\pi]\psi$. ψ_0 is arbitrary, so this holds for every state.

Axiom (ii) $$\models \ [\psi_0 \rightarrow \epsilon]\phi \quad \leftrightarrow \quad \psi_0 \rightarrow \phi$$

From the definition of $[\![\psi_0 \rightarrow \epsilon]\!]$, we can see that the post-state of the protocol is the same as the pre-state, provided that ψ_0 holds. Therefore, if a predicate, ϕ, holds at the current state, ϕ will also hold after ϵ, and vice-versa. If ψ_0 does not hold, then $[\psi_0 \rightarrow \epsilon]\phi$ trivially holds because the set of end states is empty, and $\psi_0 \rightarrow \phi$ trivially holds because its premise is false.

Axiom (iii) $$\models \quad [\psi_0 \xrightarrow{c.\phi_m} \psi_0']\phi \quad \leftrightarrow \quad \psi_0 \rightarrow (\phi_m \sqcup \psi_0' \rightarrow \phi)[y/x]$$

To prove this, we break up the equivalence into two cases: all models in which ψ_0 holds; and all models in which ψ_0 does not hold.

Case ψ_0 does not hold: If ψ_0 does not hold in the current model, then $[\psi_0 \xrightarrow{c.\phi_m} \psi_0']\phi$ is trivially true, because the precondition of the protocol is not enabled, and therefore the set of histories is empty. Similarly, if ψ_0 does not hold, $\psi_0 \to (\ldots)$ is trivially true.

Case ψ_0 holds: Firstly, we expand the definition of the left-hand side:

$$\forall(\phi_1, h, \phi_1') \in \{(\phi_0, c.\phi_m', \phi_0') \mid (\phi_0 \sqsupseteq \psi_0) \wedge (\phi_m' \sqsupseteq \phi_m) \wedge \phi_0' = \phi_m' \sqcup \phi_0 \oplus \psi_0'\} \bullet \phi_1' \models \phi$$

So, to prove that $[\psi_0 \xrightarrow{c.\phi_m} \psi_0']\phi$, we must prove that for every end-state, ϕ_1', such that ϕ_1' and the refined message ϕ_m' entail ϕ_m and $\phi_0 \oplus \psi_0'$ respectively, ϕ_1' satisfies ϕ, which, using the induction hypothesis, is equivalent to proving ϕ from ϕ_1'. We are proving this only for models in which ψ_0 holds, so, this, combined with the above, allows us to rewrite this as follows:

$$\forall\phi_0, \phi_m', \phi_0' \in \mathcal{L} \bullet \left(\phi_0 \sqsupseteq \psi_0 \wedge \phi_m' \sqsupseteq \phi_m \wedge \phi_0' = \phi_0 \oplus \psi_0'\right) \to \phi_0' \vdash \phi$$

Using the one-point rule on the equality $\phi_0' = \phi_0 \oplus \psi_0'$, this is trivially equivalent to the following:

$$\forall\phi_0, \phi_m' \in \mathcal{L} \bullet \left(\phi_0 \sqsupseteq \psi_0 \wedge \phi_m' \sqsupseteq \phi_m\right) \to \phi_0 \oplus \psi_0' \vdash \phi$$

Before we continue, we first prove the following lemma, which states that if every refinement, ϕ_0', of a constraint, ϕ_0, satisfies a certain property, then ϕ_0 also satisfies that property, and vice-versa.

Lemma A.2. $\forall\phi_0' \in \uparrow\phi_0 \bullet \phi_0' \vdash \psi \iff \phi_0 \vdash \psi$

Proof. The universal quantification is expanded to the following:

$$\phi_0' \vdash \psi \text{ and } \phi_0'' \vdash \psi \text{ and } \phi_0''' \vdash \psi \text{ and } \ldots$$

for each constraint in the upward closure of ϕ. From the definitions in De Boer *et al.* [5], we know that this is equivalent to the following:

$$\phi_0' \vee \phi_0'' \vee \phi_0''' \vee \ldots \vdash \psi$$

From the definition of upward closure, $\phi_0' \vee \phi_0'' \vee \phi_0''' \vee \ldots$ is equivalent to ϕ_0, and therefore $\phi_0 \vdash \psi$. □

Using Lemma A.2, we can remove the quantification of the predicate above:

$$\phi_m \sqcup \psi_0 \oplus \psi_0' \vdash \phi$$

Using the deduction theorem, we know that $\phi \vdash \psi$ is equivalent to $\vdash \phi \to \psi$. However, we cannot replace the above $\vdash$ with $\to$, because $\phi_m \sqcup \psi_0 \oplus \psi_0' \vdash \phi$ contains free variables that may be part of other predicates, especially ψ_0, which we assume holds for this part of the proof. Therefore, we rename all variables inside the brackets, denoted x, with fresh variables, y, before substituting $\vdash$ with $\to$:

$$(\phi_m \sqcup \psi_0 \oplus \psi_0' \to \phi)[y/x]$$

Combining the two cases of ψ_0 holding and not holding respectively, we have the following equivalence:

$$[\psi_0 \xrightarrow{c.\phi_m} \psi_0]\phi \quad \leftrightarrow \quad \psi_0 \vee (\psi_0 \wedge (\phi_m \sqcup \psi_0 \oplus \psi_0' \to \phi)[y/x])$$

Which, using the definition of $\to$, is equivalent to the following:

$$[\psi_0 \xrightarrow{c.\phi_m} \psi_0]\phi \quad \leftrightarrow \quad \psi_0 \to (\phi_m \sqcup \psi_0 \oplus \psi_0' \to \phi)[y/x]$$

Finally, we are left with something that resembles our axiom, with the exception that we have $\psi_0 \oplus \psi_0'$ on the right-hand side, rather than just ψ_0'. Recall that $\psi_0 \oplus \psi_0'$ is a constraint representing all of the information from ψ_0', plus the information about the variables in ψ_0 that are not in ψ_0'. Therefore, for the implication $\psi_0 \rightarrow (\phi_m \psi_0 \oplus \psi_0' \rightarrow \phi)$, any constraint about a variable x in ϕ must either come from ψ_0', or from ψ_0. If it comes from ψ_0, the variable is not referenced in ψ_0', then it will hold from the premise ψ_0. For example, consider $\psi_0 \vdash \psi_0 \wedge \psi_0' \rightarrow \phi$. Therefore, omiting ψ_0 would hold anyway. However, recall that we renamed all variables inside the brackets on the right-hand side so that all were fresh, therefore, this would not be the case. So, instead of renaming each of them, we rename only those in ϕ_m and ψ_0', as is specified by the side condition of Axiom 4.1(iii), leaving us with the following:

$$[\psi_0 \xrightarrow{c.\phi_m} \psi_0]\phi \quad \leftrightarrow \quad \psi_0 \rightarrow (\phi_m \sqcup \psi_0' \rightarrow \phi)[y/x]$$

In this, any information about a variable x that holds for ϕ will either come from $\phi_m \sqcup \psi_0'$ and therefore be renamed, or it will come from ψ_0 and not be renamed, but will hold from the conditions in ψ_0. Therefore, we conclude that this axiom is valid.

Axiom (iv) $$\models \quad [\pi_1;\pi_2]\phi \quad \leftrightarrow \quad [\pi_1][\pi_2]\phi$$

First, we prove the equivalence from right to left. That is, we prove $[\pi_1][\pi_2]\phi \rightarrow [\pi_1;\pi_2]\phi$. Consider arbitrary start and end states ψ_0 and ψ_0' of protocol π_1. From the semantics of $\pi_1;\pi_2$, we know that π_2 is evaluated under ψ_0'. From the premise, we know that $\psi_0' \models [\pi_2]\phi$ for all such ψ_0', therefore, $\psi_0 \models [\pi_1;\pi_2]\phi$. Since ψ_0 is arbitrary, this holds for all states.

Now, we prove the equivalence from left to right. That is, we prove $[\pi_1;\pi_2]\phi \rightarrow [\pi_1][\pi_2]\phi$. Consider arbitrary start and end states ψ_0 and ψ_0'' of protocol $\pi_1;\pi_2$. Take any intermediate state, ψ_0', on any branch of the protocol $\pi_1;\pi_2$. Let π' be the sub-protocol that remains to be executed at this state. We know that $\psi_0' \models [\pi']\phi$ holds at this point because ϕ holds at every end point of $\pi_1;\pi_2$, and π' is a sub-protocol of this. If this holds for all arbitrary intermediate states, then it must hold for the end states of π_1, which are intermediate states of $\pi_1;\pi_2$. At every end state of π_1, the sub-protocol that remains to be executed is π_2, and therefore $\psi_0' \models [\pi_2]\phi$ holds at all end states of π_1. Therefore, we conclude that $\psi_0 \models [\pi_1][\pi_2]\phi$. Since ψ_0 is arbitrary, this holds for all states.

Axiom (v) $$\models \quad [\pi_1 \cup \pi_2]\phi \quad \leftrightarrow \quad [\pi_1]\phi \wedge [\pi_2]\phi$$

From the compositional semantics defined in Section 2.3, the set of histories resulting from a protocol choice is the union of the histories of both protocols. Therefore, the left hand side of this axiom is represented as $\forall \psi_0 \in [\![\pi_1]\!] \cup [\![\pi_2]\!] \ldots$, and the right hand side as $\forall \psi_0 \in [\![\pi_1]\!] \ldots$ and $\forall \psi_0 \in [\![\pi_2]\!] \ldots$. From the definition of the set union operator, we know that $a \in A \cup B$ if and only if $a \in A$ and $a \in B$, therefore, this demonstrates that the histories of the two predicate about are equivalent, and therefore any predicate that holds for all end states of one will hold for the other.

Axiom (vi) $$\models \quad [\mathbf{var}_x^{\psi_0} \cdot \pi]\phi \quad \leftrightarrow \quad x_0 = x \wedge \psi_0 \rightarrow [\pi](x_0 = x \rightarrow \phi)$$

Recall from the semantics, that variable declarations introduce a new variable with constraint, but that also that the variable can already be in the state. For simplicity, in this proof, we assume that the variable x is fresh. If this is not the case, x must be renamed to a fresh variable.

This axiom is explained as follows. In the protocol $\mathbf{var}_x^{\psi_0} \cdot \pi$, the constraints on x remain unchanged during the execution of π, and otherwise $\mathbf{var}_x^{\psi_0} \cdot \pi$ executes in the same manner as π. Therefore, we say that ϕ holds for every end state in $\mathbf{var}_x^{\psi_0} \cdot \pi$ if and only if it holds in every end state of π in which the contraints on x do not change. To model x not changing, we introduce a fresh variable x_0, and specify that $x_0 = x$ before the execution of π. We know that π does not change

the value of x_0 (it is a fresh variable), so after executing π, the constraints on x_0 remain as before. Therefore, the cases in which $x_0 = x$ in the end are equivalent to x remaining unchanged.

First, we prove the equivalence from left to right. Consider any end state, ψ_0', of $\mathbf{var}_x^{\psi_0} \cdot \pi$. We know that $\psi_0' \models \phi$. We also know that the same path and end state must exist for π, because π is the same as $\mathbf{var}_x^{\psi_0} \cdot \pi$, except it does not maintain the constraints on x throughout, therefore, the paths and end states in $\mathbf{var}_x^{\psi_0} \cdot \pi$ must be a subset of those in π. Clearly, $\psi_0' \models x_0 = x$, because x_0 is fresh and therefore not changed by π, therefore $\psi_0' \models x_0 = x \rightarrow \phi$.

Now, we prove right to left. If we consider that ϕ is true in all end states of π in which x remains unchanged, then it must be that ϕ holds in any end state of $\mathbf{var}_x^{\psi_0} \cdot \pi$, because x remains unchanged by the semantics of variable declarations.

Axiom (vii) $\quad \models \quad [N(x)]\phi \quad \leftrightarrow \quad [\pi[x/y]]\phi \quad$ where $N(y) \mathrel{\widehat{=}} \pi$

This axiom is trivially true from the definition of $[\![N(x)]\!]$ and from $N(y) \mathrel{\widehat{=}} \pi$.

The proof that each of the axioms is valid demonstrates that $\mathcal{L}_\pi$ is sound. $\square$