Propositional Dynamic Logic for Reasoning about First-Class Agent Interaction Protocols

Tim Miller
Department of Computer Science and Software Engineering,
University of Melbourne, Victoria, 3010, Australia
tmiller@unimelb.edu.au

Peter McBurney
Department of Computer Science,
University of Liverpool, Liverpool, L69 7ZF, UK
mcburney@liverpool.ac.uk

For agents to fulfill their potential of being intelligent and adaptive, it is useful to model their interaction protocols as executable entities that can be referenced, inspected, composed, shared and invoked between agents, all at runtime. We use the term first-class protocol to refer to such protocols. Rather than having hard-coded decision making mechanisms for choosing their next move, agents can inspect the protocol specification at runtime to do so, increasing their flexibility. In this paper, we show that propositional dynamic logic (PDL) can be used to represent and reason about the outcomes of first-class protocols. We define a proof system for PDL that permits reasoning about recursively-defined protocols. The proof system is divided into two parts: one for reasoning about terminating protocols, and one for reasoning about non-terminating protocols. We prove that proofs about terminating protocols can be automated, while proofs about non-terminating protocols are unable to be automated in some cases. We prove that, for a restricted class of non-terminating protocols, proofs about them can be transformed to proofs about terminating protocols, making them automatable.

Key words: multi-agent systems, agent interaction protocols, propositional dynamic logic, first-class protocols.

1. INTRODUCTION

Research into interaction protocols for 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 — requiring agent code to change 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, 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 RASA (Miller and McBurney, 2007), which regards protocols as first-class entities. These first-class protocols are documents that exist 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. Rather than a protocol specification being just a sequence of arbitrary tokens, each message explicitly contains a meaning, which is defined as a manipulation of a shared social state; for example, a set of obligations between agents. 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 learn of new
protocols at runtime, making these agents more adaptable. Using social states instead of agents’ mental states allows one to verify whether an agent is behaving according to the protocol rules (Singh, 2000).

The first contribution of this paper is to show that propositional dynamic logic (PDL) provides a sound and complete way to represent and reason about outcomes of \textsc{rasa} protocols. We present a logic, called $\mathcal{L}_\alpha$, that is an instantiation of PDL, because it is applicable only to a certain types of models and programs: the models are shared social states of the agents, and the programs are \textsc{rasa} protocols. We demonstrate that by restricting the models and programs to these constructs, agents equipped with minimal machinery are able to prove properties about \textsc{rasa} protocols.

The proof system for the logic is divided into two parts: one for reasoning about terminating protocols, called finite $\mathcal{L}_\alpha$, and one for reasoning about non-terminating protocols, called infinite $\mathcal{L}_\alpha$. Automation of the former is straightforward, but for the latter, automation is not so straightforward.

The second contribution of this paper is to prove that, for a restricted class of non-terminating protocols, proofs about non-terminating protocols can be transformed to proofs about terminating protocols. This result allows us to automatically prove properties about non-terminating protocols using finite $\mathcal{L}_\alpha$. We believe that the restricted class is large enough that it contains most protocols used in multi-agent systems, so this result allows agents to prove properties about most protocols in the literature.

We believe that the results in this paper can be generalised to other first-class protocol specification languages, and indeed, to action languages in general.

The outline of this paper is as follows. In Section 2, we present an overview of the \textsc{rasa} framework, including the syntax and denotational semantics of the \textsc{rasa} protocol specification language. Section 3 presents the language syntax and semantics of the logic used for reasoning about first-class protocols. Section 4 presents a deductive proof system for this logic, which is used to prove properties about \textsc{rasa} protocols. A discussion of the possibility of automation follows. Section 5 presents a method for transforming proofs about a restricted class of non-terminating protocols into proofs about terminating protocols. Section 6 discusses an implementation of this proof system in Prolog, and presents the results of an experimental evaluation of this implementation. Section 7 presents some work related to this paper, and Section 8 concludes and discusses future work.

1.1. First-Class Protocol — A Definition

Before we define first-class protocol, we define protocol, in the context of agent interaction. An agent interaction protocol is specification that coordinates message-based communication between agents, and is used to give a context for the interaction. A protocol defines the rules of encounter by stating:

- who can send messages;
- what they can say; and
- when they can say it, or more specifically, in which order they can say it.

However, most protocol specification languages, while supporting the specification of the above, are not expressive or formal enough to allow agents to share protocol definitions, and to learn the rules of these protocols.

Our notion of first-class protocol is comparable to the notion of first-class object/entity in programming languages (Strachey, 2000). That is, a first-class protocol is a referencable, sharable, manipulable entity that exists as a runtime value in a multi-agent system. From the specification of a first-class protocol, participating agents should be able to inspect the specification 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. From this, our definition of “first-class protocol” is of a referencable, sharable, manipulable entity that specifies:

- who can send messages;
- what they can say;
- when they can say it;
- which preconditions must hold for messages to be sent; and
what the postcondition or meaning of sending them is; that is, what is the consequence of an agent sending a message.

Typically, the meaning of a message is specified as some manipulation of a shared social state; for example, the rules of a protocol may state that accepting to buy a particular item commits you to paying for it, by specifying that sending this message adds that commitment to a commitment base. Similarly, preconditions are specified with respect to that shared state; for example, an agent can only commit to selling an item if it is not already committed to selling it. Each message is given an explicit precondition and postcondition, so that agents can determine the result of sending a message. The result of multiple messages can be calculated compositionally. For the agents to reason about this, it must be specified in a way such that the agent can understand it.

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/machine-readable**: 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. Such new protocols must also form first-class protocols in their 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, 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 RASA FRAMEWORK

The \textit{RASA} specification language (Miller and McBurney, 2007) was designed as an example of the minimal operators that would be required for a first-class protocol specification language. The language is an action language that uses constraints to represent information. In this section, we present the language, and its denotational semantics.

#### 2.1. Modelling Information

To make \textit{RASA} as widely-applicable as possible, modelling the information in messages and the manipulation of the shared social state is not constrained to a particular language. Instead, we assume that the universe of discourse is modelled using constraints. This permits agent designers to use different underlying languages, such as description logics (Baader et al., 2003) or constraint languages (de Boer et al., 1997).

\textbf{Definition 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. (1997). A cylindric constraint system is a complete algebraic lattice, \((C, \supseteq, \sqcup, \text{true}, \text{false}, \text{Var}, \exists)\). In this structure, \(C\) is the set of atomic propositions in the language, \(\supseteq\) is an entailment operator, \text{true} and \text{false} are the least and greatest elements of \(C\) respectively, \(\sqcup\) is the least upper bound operator, \text{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, for example, \( X = 1 \), where \( X \) is a variable; a conjunction (least upper bound), \( \phi \sqcap \psi \), where \( \phi \) and \( \psi \) are constraints; or \( \exists x \phi \), where \( \phi \) is a constraint and \( x \in \text{Var} \). For simplicity, we assume that \( x \) represents a set of variables. We will continue to use the symbols \( \phi \) and \( \psi \), possibly with subscripts, to refer to constraints throughout this paper.

Negation is permitted the right of an entailment operator: \( c \sqsupseteq \neg d \) is the equivalent of \( \neg(c \sqsupseteq d) \); that is, negation as failure. Other propositional operators are defined from these in the standard way.

We use \( \text{vars}(\phi) \) to refer to the free variables that occur in \( \phi \); that is, the variables referenced in \( \phi \) that are not hidden using \( \exists \). We define the shorthand \( \exists \phi \), which represents the constraint in which all the variables in \( \phi \), except those in \( x \), are hidden: \( \exists x \phi = \exists \text{vars}(\phi) \setminus x \phi \).

We introduce a renaming operator, such that \( \phi[x/y] \) means ‘replace all references of \( y \) in \( \phi \) with \( x' \), which is defined as \( \exists y((y = x') \cup \phi) \).

2.2. Modelling Protocols

The \( \text{RASA} \) protocol specification language is an action language. Messages sent across channels are the actions, and they manipulate a shared social state. When an agent sends a message across a channel, the definition of that message in the protocol informs other participants how to update their copy of the state.

**Definition 2:** Syntax.

The syntax of the \( \text{RASA} \) language is defined using the following grammar:

\[
\begin{align*}
\alpha & ::= \phi \rightarrow \epsilon \mid \phi \Phi_{i,j}^\psi \mid \alpha; \alpha \mid \alpha \cup \alpha \mid \text{var}_x^\psi \alpha \mid N(x) \\
\psi & ::= c \mid \psi \sqcup \psi \mid \exists x \psi \\
\phi & ::= \neg \phi \mid \psi
\end{align*}
\]

in which \( c \in \mathcal{L} \) represents any atomic constraint. The non-terminal \( \psi \) represents constraints in the constraint system, while \( \phi \) represents constraints as well as negated constraints. Therefore, negations are permitted in preconditions only, and cannot be asserted into the social state. This is due to treating negation as failure in the constraint system.

Messages are specified using atomic protocols of the form

\[
\psi \Phi_{i,j}^\psi \psi',
\]

in which \( \psi', \psi_m, \) and \( \psi' \) are all constraints. The constraint \( \psi \) represents the precondition that must hold in the current social state for the message to be sent, \( \Phi_{i,j} \) represents the channel from participant \( i \) to participant \( j \), the constraint \( \psi_m \) represent the message template, and the constraint \( \psi' \) represents the postcondition, which specifies the effect this message has on the state. We omit \((i,j)\) when we do not care who the sender and receiver of the message are. In this paper, the subscript ‘\( m \)’ is used to denote that a constraint represents a message.

We use the notion of inertia in calculating the new state from the postcondition; that is, any variables in the postcondition are defined by the postcondition, while any other variables in the state are left unchanged. Agents are permitted to send the message \( \psi_m \), such that \( \psi_m \sqsupseteq \psi_m \), so that agents can further constrain the values of the messages; thus, \( \psi_m \) is only a template of the message.

For example, consider the following atomic protocol:

\[
\text{true} \xrightarrow{\Phi_{i,j}^\psi} \psi',
\]

in which a seller, \( S \), is presenting a quote to a customer, \( C \), for an item. The seller would like to instantiate \( \text{Cost} \) to the price for which it wants to sell the item; for example, 10. It would send the message \( \text{present-quote}(\text{Item, Cost}) \sqcup \text{Cost} = 10 \), which constrains the message template by adding
further information; that is, the quote is 10. This has implications regarding the state that results from sending this message. This is discussed further in Section 2.3 when we present the semantics of the \( \mathcal{RASA} \) language.

A special type of atomic protocol is the empty protocol \( \psi \rightarrow \epsilon \), in which \( \epsilon \) is a constant specifying that no message is required to be sent if \( \psi \) holds in the current social state. This is similar to the test operator found in PDL, written \( \psi ? \), however, we restrict \( \psi \) to being a constraint, whereas in PDL, \( \psi \) can be a dynamic logic formula. We use the shorthand \( \epsilon \) to represent true \( \rightarrow \epsilon \).

Compound protocols can be built up from atomic protocols and empty protocols using operators. If \( \alpha \) and \( \beta \) are protocols, then \( \alpha ; \beta \) is their sequential composition, \( \alpha \cup \beta \) is a choice between them, and \( \text{var}^\psi_{\alpha} \), \( \alpha \) is \( \alpha \), with the exception that the local variable, \( x \), and the constraints, \( \psi \), on \( x \), are maintained throughout the execution of \( \alpha \). For example, consider a case in which we want to increment the integer value of a variable, \( x \). The only way to specify this is to specify that the value of \( x \) is 1 greater than before the message was sent, which is not possible using an atomic protocol; the postcondition merely represents a constraint between state variables, with no way of referencing pre-state values. However, we use the variable declaration operator to simulate this behaviour:

\[
N \triangleq \text{var}^x_{x_0 = 0} \cdot x \leq 10 \cdot \text{inc}(x). \quad x = x_0 + 1.
\]

Constraints placed on a locally declared variable are maintained throughout its entire scope. Therefore, the constraints on \( x_0 \) in the postcondition are that it equals the constraints on \( x \) in the pre-state. If we were to execute this message sending in the state \( x = 1 \), then the postcondition would resolve to the following: \( 3 \ll x = 1 \ll x_0 = x \ll x = x_0 + 1 \). The only solution for this is \( x_0 = 1 \ll x = 2 \). The scope of the variable \( x_0 \) would end, and the post-state would be \( x = 2 \).


A protocol specification is a collection of protocol definitions of the form \( N(x, \ldots, y) \triangleq \alpha \), in which \( N \) is a name, \( \alpha \) is a protocol, and \( x, \ldots, y \in \text{Var} \) are the free variables in protocol \( \alpha \). Protocols can be referenced from other protocols via their name, thereby allowing recursive definitions.

Example 1:

We present a running example (modified from Yolum and Singh (2004)) of a customer requesting a quote for a specified item. A supplier may agree or not agree to supply the item, and give a price. If they present a price, the customer may take or leave that price. Finally, the customer can pay and deliver the item.

We use the following propositions, many of which were defined in the motivating example by Yolum and Singh (2004):

- \( \text{request}(	ext{Item}) \): the customer has requested a quote for the item.
- \( \text{deliver}(	ext{Item}) \): the supplier has delivered the item.
- \( \text{pay}(\text{Cost}) \): the customer has paid the specified cost.

In addition, we define the following abbreviations of commitments:

- \( \text{cc_accept}(	ext{Item}, \text{Cost}) \): an abbreviation for \( CC(C, S, \text{deliver(Item)}, \text{pay(Cost)}) \); that is, the customer is committed to paying for the item if they have received it.
- \( \text{cc.promise}(	ext{Item}, \text{Cost}) \): an abbreviation for \( CC(S, C, \text{cc_accept(Item, Cost)}, \text{deliver(Item)}) \); that is, the supplier is committed to supplying the item once the customer has committed to pay for the item.

Throughout this example, we assume that each protocol definition declares the variables \( \text{Item}, \text{Cost}, C, \) and \( S \), representing the item, the quote amount, the customer, and the supplier respectively. We also assume these variables are passed to each of the references to protocol names. These are omitted for readability.
To initiate the interaction, the customer requests a quote for the item. The precondition specifies that the customer must have not requested a quote for this already, and the merchant must have not provided a quote already. The ‘¬’ states that we do not care to whom the other person the merchant may have provided a quote. Neither party is committed to bring about any state of the world at this point.

\[ \text{RequestQuote} \triangleq \neg \text{request}((\text{Item})) \cup \text{cc\_promise}((\text{Item}), \_\_\_) \rightarrow \text{cc\_promise}((\text{Item}), \text{Cost}) \]

In response to this, the supplier sends a quote, and they are committed to selling the item to the customer at the quoted price. Alternatively, the supplier can send a quote to a prospective customer without that customer requesting one.

\[ \text{PresentQuote} \triangleq \text{cc\_promise}((\text{Item}), \_\_\_) \rightarrow \text{cc\_promise}((\text{Item}), \text{Cost}) \]

The customer can reject the quote (omitted), or can agree to purchase the item, at which point they are committed to paying once the item is delivered.

\[ \text{AcceptQuote} \triangleq \text{cc\_promise}((\text{Item}), \text{Cost}) \rightarrow \text{cc\_accept}((\text{Item}), \text{Cost}) \]

The customer can send the Electronic Payment Order (EPO), therefore fulfilling their commitment to paying for the item.

\[ \text{Pay} \triangleq \text{cc\_promise}((\text{Item}), \text{Cost}) \rightarrow \text{cc\_accept}((\text{Item}), \text{Cost}) \]

Finally, the supplier can send the item.

\[ \text{SendItem} \triangleq \text{cc\_accept}((\text{Item}), \text{Cost}) \rightarrow \text{deliver}((\text{Item})) \]

Each of these protocols is specified such that specific preconditions must be met before the agent can send the message. For example, the protocol specifies that the customer can only pay for the item once the supplier has agreed to supply it. Such a constraint may in fact be related to the policy or strategy of the customer itself, however, in the interest of fairness, it is reasonable to say that a specifier of this protocol (the supplier perhaps) would make sure of such a policy.

Despite this, we can also explicitly introduce constraints over the order in which participants send messages using the RASA composition operators. The specification above says nothing about how these messages come together; they are merely a collection of atomic protocols. One can specify that the above may be executed in any order. However, explicitly ordering messages is important in many protocols, and can also reduce the complexity of the atomic protocols from which a composite protocol is built.

In the transaction example, we introduce several constraints using the operators, which can either replace many of the preconditions, or can merely use the preconditions to serve as supplementary constraints. The advantage of using the operators is that the space that an agent must search to decide its preferred behaviour is more straightforward to calculate.

The top-level protocol definition is straightforward. A transaction is broadly divided into two parts: the quotation, and the finalisation (payment and delivery).

\[ \text{Transaction} \triangleq \text{Quotation} \cup \text{Finalise} \]

The quotation stage is defined to be flexible. First, the customer may request a quote, to which the supplier must reply presenting a quote. Alternatively, the supplier may present a quote (for example, an advert) without a request. Either way, once this quote has been presented, the supplier is committed to this quote, as specified by the \( \text{PresentQuote} \) sub-protocol. The customer may choose to accept or reject the quote, committing to pay for the quote if it is accepted.

\[ \text{Quotation} \triangleq \{ \text{RequestQuote}; \text{PresentQuote}; (\text{AcceptQuote} \cup \text{RejectQuote}) \} \]

In the above definition, square brackets are used to specify that \( \text{RequestQuote} \) is optional; that is, it does not have to execute. We define \( \{\alpha\} \) to be equivalent to \( \epsilon \cup \alpha \), so either \( \alpha \) occurs, or nothing occurs.

\( \text{Finalise} \) is defined as the delivery and payment for the item. This can be performed in either order. The final branch in the choice represents the case when no agreement was made; that is, if no commitment to accept the offer has been made (\( \neg \text{cc\_accept} \)), then the protocol terminates without any additional information being exchanged.
This example demonstrates the composition construction of a protocol from the bottom up using atomic protocols and the RASA composition operators. Clearly, the composition of this protocol could be done in many different ways; for example, the reference to Finalise could be moved into the Quotation protocol, directly after AcceptQuote. This would remove the need for the termination condition in Finalise. This particular composition is presented because the authors feel that it provides a better abstraction than the discussed alternative.

Reasoning about this requires the agents to understand the underlying concepts of commitment, as well as the concepts and terms used throughout the transaction process, such as payment and delivery, which could be defined using a shared ontology. An agent that understands these concepts can receive the above protocol definition, and reason about which interactions it prefers. For example, the customer will receive a quote, and look ahead to final outcomes to see that if they accept the quote, they will be committed to paying the quoted amount. If they prefer the state of affairs (assessed using a utility function, for example) in which they keep the quoted amount and not receive the good, they reject the quote. Otherwise, if they prefer the state of affairs in which the receive and pay for the goods, they accept the quote.

2.3. Denotational Semantics

In this section, we present a denotational semantics for the RASA protocol specification language, on which we base our logic. We define the semantics as a set of traces, in which each trace defines a sequence of messages sent by the system of agents, and the resulting outcome state.

Definition 4: Compositional, Denotational Semantics.

We define a trace as a triple, in which the first element of the triple represents the pre-state of the trace, the second element represents a sequence of communications across channels, and the third element represents the state resulting after that sequence of communications. The semantics of a RASA protocol is defined as the set containing all traces permitted by the protocol.

The set of all sequences of communications for a language, \( \mathcal{L} \), is denoted as \( \mathcal{I}_\mathcal{L} \). Using this, we define the set of all possible traces for the language \( \mathcal{L} \) as \( \psi(\mathcal{L}_+ \times \mathcal{I}_\mathcal{L} \times \mathcal{L}_+) \), in which \( \psi \) represents the power set function, and \( \mathcal{L}_+ \) represents \( \mathcal{L} \setminus \{\text{false} \} \).

The semantics of RASA protocols is defined as a function \( [D, \alpha] : \text{Env} \rightarrow \psi(\mathcal{L}_+ \times \mathcal{I}_\mathcal{L} \times \mathcal{L}_+) \), in which \( D \) is the set of declarations in which the protocol \( \alpha \) is evaluated, and \( \text{Env} \) is a function from names to sets of traces, \( \text{Env} : \text{Name} \rightarrow \psi(\mathcal{L}_+ \times \mathcal{I}_\mathcal{L} \times \mathcal{L}_+) \), used for mapping protocol reference names to set of traces they define. So, the semantics is a function that takes a protocol definition and an environment, and returns the set of traces that are defined by the protocol. We omit \( D \) and the environment \( e \) whenever they are not used.

The first definition is that of the empty protocol, \( \psi \rightarrow \epsilon \). This is defined as a set containing a single trace, in which there are no messages, and the pre- and post-state are the same:

\[
\llbracket \psi \rightarrow \epsilon \rrbracket \quad \cong \quad \{(\phi, \emptyset, \emptyset) \mid \phi \supseteq \psi \}.
\]

An atomic protocol is defined as the set of traces in which the first element in the trace satisfies the precondition, the second element is the message, and the third element is the resulting state. Recall from Section 2.2, that the message can be the constraint, \( \psi_m \), but can also be a constraint, \( \phi_m \), that contains more information than \( \psi_m \). The resulting state is \( \phi \oplus (\phi_m \cup \phi') \), in which \( \oplus : (\mathcal{L} \times \mathcal{L}) \rightarrow \mathcal{L} \) is an overriding function defined as \( \phi \oplus \psi' = \psi' \cup \exists_{\text{vars}}(\psi') \phi \). The resulting state is a new constraint such that the values of any free variables in \( \phi \) are overridden with the new values in the postcondition, while any other variables remain unchanged. Formally, the atomic protocol semantics is defined as follows:

\[
\llbracket \psi \leftarrow \psi_m, \psi' \rrbracket \quad \cong \quad \{(\phi, (c, \phi_m), \phi \oplus (\phi_m \cup \phi')) \mid \phi \supseteq \psi \land (\phi_m \cup \phi' \supseteq \psi_m \cup \psi') \}.
\]

Allowing agents to constrain messages clearly has an effect on the resulting state. We enforce that any additional information placed in the message, \( \phi_m \) must also apply to the resulting state. For example, recall the protocol

\[
\text{true} \xrightarrow{\psi[S,C]\cdot \text{present}\_\text{quote}(\text{Item, Cost})} \text{cost(\text{Item, Cost})},
\]
in which a seller presents a quote for an item. If the sender constrains Cost in the message with Cost = 10, this information needs to be shared with the postcondition so that the value of the variable Cost can be determined. In the semantics, we enforce this condition by including the message as part of the post-state: $\phi \oplus (\phi_m \cup \phi')$. In this example, the only solution for Cost would be Cost = 10, so the post-state is \text{cost(Item,Cost)} \cup Cost = 10, which reduces to \text{cost(Item,10)}. Clearly, atomic protocol execution is non-deterministic: the sending agent can choose between different constraints on the message. However, calculating the resulting state is deterministic: the result is always the combination of the message and the postcondition, plus the constraints from the pre-state that are applicable (those containing variables not referenced in the postcondition).

Sequential composition, $\alpha;\beta$, is defined by using the post-states of $\alpha$ as the pre-states of $\beta$:

$$\llbracket \alpha;\beta \rrbracket \equiv \{ (\phi, h_1 \circ h_2, \phi''') | \exists \phi' \bullet (\phi, h_1, \phi') \in \llbracket \alpha \rrbracket \land (\phi', h_2, \phi'') \in \llbracket \beta \rrbracket \}.$$  

The notation $h_1 \circ h_2$ specifies the concatenation of sequences $h_1$ and $h_2$.

A choice, $\alpha \cup \beta$, between two protocols, $\alpha$ and $\beta$, is defined as the union of all traces from $\alpha$ and $\beta$:

$$\llbracket \alpha \cup \beta \rrbracket \equiv \llbracket \alpha \rrbracket \cup \llbracket \beta \rrbracket.$$  

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 $\text{var}_x^\psi \alpha$. First, we want to execute $\alpha$ as normal, but within the context that there is a new variable $x$, constrained by $\psi$. 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 $\alpha$, we do not want the constraints over the local variable $x$ to change. Finally, once $\alpha$ 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. The formal definition of this is:

$$\llbracket \text{var}_x^\psi \alpha \rrbracket \equiv \{ (\phi, h, \exists_x \phi' \cup \exists_x \phi) \mid (\exists_x \phi \cup \psi, h, \phi') \in \llbracket \alpha \rrbracket \land \exists_x \phi' = \exists_x (\exists_x \phi \cup \psi) \}.$$  

To discuss how this definition achieves the intended semantics, it is best to start in the middle at $(\exists_x \phi \cup \psi, h, \phi') \in \llbracket \alpha \rrbracket$. We execute the sub-protocol $\alpha$ from the state $\exists_x \phi \cup \psi$. Any free occurrences of variable $x$ in $\phi$ are hidden in case $x$ is already a variable in this constraint. Once $\alpha$ is executed, we are left with the post-state $\phi'$. The condition $\exists_x \phi' = \exists_x (\exists_x \phi \cup \psi)$ 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 $\exists_x \phi$, which specifies that we hide occurrences of all free variables in the pre-state, $\phi$, except $x$. This restores the variable $x$ to the state with its pre-state values.

The semantics for a protocol reference is also not straightforward, due to the fact that we allow recursive definitions; that is, protocol that reference themselves. Without recursive definitions, the semantics would specify that a reference is replaced with its protocol definition in $D$. However, for recursive definitions, we use Scott-Strachey $\text{fixpoint}$ semantics ($\text{Stoy}$, 1977). Defining such a semantics is no a straightforward task; fortunately, de Boer et al. (1997) have defined a semantics for a similar construct, which we modify as:

$$\llbracket D, N(x) \rrbracket(e) \equiv \begin{cases} \llbracket N(y) \rrbracket(e)(x/y) & \text{if } N(y) \equiv \alpha \in D \text{ and } x \neq y \\ e(N) & \text{if } N \in \text{dom}(e) \\ \mu F & \text{if } N(x) \equiv \alpha \in D \text{ and } N \notin \text{dom}(e) \\ \text{where } F(H) = \llbracket D, \alpha \rrbracket(e\{N \mapsto H\}) \\ \text{and } \mu F \text{ is the least fixpoint of a function } F \\ \text{with respect to the partial order } \subseteq \\ \text{i.e. the least } \mu F \text{ such that } \mu F = F(\mu F). \end{cases}$$  

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. 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 $N$,
denoted \( e(N) \), 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 \( \alpha \) with the environment that is the same as \( e \), but with \( e(N) \) overridden with \( H \). The least fixpoint of \( F \), denoted \( \mu F \), with regards to the partial order \( \subseteq \), is the denoted value of \( N \). For any subsequent references to \( N \), the semantics are defined by the second case, \( e(N) \).

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 iteration operator found in dynamic logics (Harel et al., 2000) and Kleene algebras (Kozen, 1990), and is written \( \alpha^* \). This defines a protocol that iterates over the sub-protocol \( \alpha \) zero or more times. Formally, this is defined as follows:

\[
[D, \alpha^*] \equiv [D', N] \quad \text{where} \quad D' = D \cup \{N \equiv e \cup (\alpha; N)\}.
\]

That is, \( \alpha^* \) is equivalent to the protocol named \( N \), where \( N \) is defined as a choice between the empty protocol or the sequential composition of \( \alpha \) 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.

Additional redundant operators found in dynamic logics and Kleene algebras can similarly be defined, such as \( \alpha^+ \), which is defined as \( \alpha; \alpha^* \).

3. PROTOCOL ENTAILMENT

Propositional dynamic logic (PDL) is a modal logic used to reason about programs. Programs are explicit constructs in PDL. The programs in PDL consist of atomic programs, which are generally programming language statements, and compound operators, such as sequential composition and choice, used to build more complex programs from atomic programs. The compositionality of programs is reflected in the logic, and properties about programs can be used by breaking compound programs into their atomic parts, and proving properties about these.

In this section, we present a PDL for the RASA language. This logic is referred to as \( \mathcal{L}_{\alpha} \). We view this logic as an instantiation of propositional dynamic logic (PDL) (Harel et al., 2000). By “instantiation”, we mean that RASA protocols are used in place of abstract programs, and constraints in place of models in which propositions are interpreted. Operators used to compose complex programs in PDL correspond to protocol composition operators in RASA.

3.1. Syntax of \( \mathcal{L}_{\alpha} \)

In this section, we introduce the logic \( \mathcal{L}_{\alpha} \), which is used to reason about the outcomes of protocols. \( \mathcal{L}_{\alpha} \) is built on \( \mathcal{L} \), in that any constraint \( \phi \in \mathcal{L} \) is also in \( \mathcal{L}_{\alpha} \). To avoid ambiguity, we distinguish the two languages by using the term proposition to refer to properties about protocols, while continuing to use the term constraint to refer to a formula of the language \( \mathcal{L} \). We use \( \phi \) and \( \psi \) to represent both propositions and constraints; however, we subscript constraints with a number, for example, \( \phi_0 \), to indicate that a \( \phi_0 \) is strictly in \( \mathcal{L} \); that is, \( \phi_0 \) is not in \( \mathcal{L}_{\alpha} \setminus \mathcal{L} \).

Definition 5: Syntax.

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

\[
\phi \ ::= \ \phi_0 \mid \phi \land \phi \mid \neg \phi \mid [\alpha]\phi.
\]

Each formula is evaluated within a protocol state; that is, a constraint. In the this grammar, \( \phi_0 \) is a constraint from \( \mathcal{L} \), and is true if it is entailed by the protocol state. \( \land \) and \( \neg \) take on the usual meanings. The final branch of the grammar contains the interesting operator. The meaning of the proposition \( [\alpha]\phi \) is as follows: if the protocol \( \alpha \) was to be executed, then at every end state (terminating state) of the protocol \( \alpha \), \( \phi \) would hold. We call this protocol entailment (executing this
Recall the specification of the electronic transaction in Example 1. Assume that the constant self is the ID of the agent that is reasoning about this example. The following specifies
that, if the agent accepts the quote, it is committed to paying if the item is delivered.

\[ C = self \rightarrow [AcceptQuote]CC(self, S, deliver(Item), pay(Cost)). \]

In the case of the Quotation 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 quote. This can be represented using the following formula.

\[ C = self \rightarrow (Quotation)CC(self, S, deliver(Item), pay(Cost)). \]

If an agent has a goal of having the item delivered, it can match the above characterisation with its goal, and use this protocol to request a quote from a supplier. Upon receiving the quote, it can characterisation on the AcceptQuote sub-protocol to calculate that, if it accepts, it will be committed to paying for the item.

3.4. Expressiveness

We briefly and informally discuss expressiveness of the logic. For this, we consider a protocol as a tree, like a game tree, in which nodes represent states, arcs represent messages, and branches of more than one arc at a node represent choices at that node.

Due to the indexing of the modal operators with protocol definitions, one can specify properties about any node in the tree. For example, consider the protocol \( \alpha; (\beta \cup \gamma) \). The proposition \( \psi \rightarrow [\alpha]\phi \) specifies that under the conditions specified by \( \psi \), the property \( \phi \) holds in every outcome of the protocol \( \alpha \). One can specify that \( \phi \) holds for every outcome of \( \alpha \) and every outcome of \( \beta \) (when executed as part of the larger protocol; that is, after \( \alpha \)), but for no outcomes of \( \gamma \) (when executed after \( \alpha \)), with the proposition \( [\alpha \cup \alpha; \beta] \phi \land \neg[\alpha; \gamma] \phi \).

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

\[ \phi \land [\alpha]\phi \land [\alpha; \beta]\phi \land [\alpha; \gamma]\phi. \]

From this, we can also define temporal-like properties of a protocol; for example, that \( \phi \) holds at every node for at least one path in the protocol \( \alpha; (\beta \cup \gamma) \). This could be defined as follows:

\[ \phi \land (\alpha)(\phi \land ((\beta)\phi \lor (\gamma)\phi)). \]

One can define similar properties such as \( \phi \) holds for at least one node of all paths. It would be straightforward to define shorthand operators that specify such properties.

The properties that can be specified in the logic are restricted by the underlying constraint language. One cannot express a property about the relationship between two items in a protocol outcome if the underlying constraint language cannot express this.

We 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 representing outcomes of protocols, and the effect of messages are specified by their respective postconditions.

As a final note on expressiveness, we stress that our semantics assumes a closed-world view with respect to protocols. That is, during the execution of a protocol, only the messages sent as a part of that protocol can have an effect on the shared state. This assumption implies that an agent can commit to something as part of a protocol, while also being committed to its opposite. We do not believe the closed-world view is unreasonable. In fact, protocol languages generally make this assumption. FIPA protocols, for example, consider preconditions and postconditions in which agents believe or do not believe certain statements. There is no restriction in place preventing an agent from participating in two different conversations at once that require it to believe contradictory statements. If we were to take an open-world view using socially-based semantics, it is not possible
for the agents participating in an interaction to know the shared state of that interaction at any time, because it could be changed by interactions in which the agent is not involved.

4. PROOF SYSTEM FOR $L_\alpha$

To prove properties about RASA protocols, we define a deductive proof system for $L_\alpha$. We present axioms and inference rules for $L_\alpha$, and prove that this system is sound and complete with respect to the semantics defined in Section 3.

First, we define a proof system for all-terminating protocols in finite domains. By “all-terminating”, we mean that the protocol terminates for all executions, and by “finite domains”, we imply that the underlying constraint language is over a finite domain, such as all 32-bit integers. We refer to this proof system as finite $L_\alpha$. We show that, given the assumption of finiteness, an agent can prove any proposition using the axioms in the proof system and the underlying constraint solver.

Then, we extend this proof system to handle non-terminating protocols and infinite domains. By “non-terminating”, we mean that the protocol is not all-terminating; therefore, there is at least one non-terminating execution of the protocol, or the underlying constraint language is infinite. This extension consists of the addition of a single inference rule, however, it has consequences on the ability to automate proofs. We refer to this proof system as infinite $L_\alpha$. Infinite $L_\alpha$ is also applicable for all-terminating protocols.

We use the proof system itself to demonstrate that proofs about a restricted subset of non-terminating protocols can be reduced to proofs in the finite proof system. This is of particular interest because it allows automated proofs that would otherwise be difficult to automate. This is discussed further in Section 5.

4.1. A Deductive Proof System for Finite $L_\alpha$

In this section, we present a deductive proof system for finite $L_\alpha$, and discuss its limitations.

We write $\langle D \mid \psi \vdash \phi \rangle$ to indicate that $\phi$ is provable in all models in which $\psi$ holds, where $D$ is the relevant protocol specification. We omit $D$ whenever the context is clear. We write $\vdash \phi$ to indicate that $\phi$ is provable in all models ($\phi$ is a theorem).

4.1.1. Axiomatization of Finite $L_\alpha$. Because we have not specified a particular underlying constraint language, we cannot provide a full set of axioms. Instead, we state simply that any axioms of $L$ are also axioms in our proof system. Because the underlying constraint language satisfies the properties of a lattice, as described by de Boer et al. (1997), so we assume several axioms over $L$, such as De Morgan’s laws and the law of double negation. We also assume that $L$ is sound and complete.

Definition 7: Axioms for $L_\alpha$.

In addition to the axioms for $L$, we propose the following additional set of axiom schemas for the proof system:
In this section, we present a result that shows that proofs in $L_\alpha$ can be performed automatically by using the axioms presented in Section 4.1.1, and the entailment operator of the underlying constraint system.

**Theorem 1:** Given any proposition $\phi$ in finite $L_\alpha$, $\phi$ can be reduced to a constraint $\phi_0$ in $L$, using the axioms, such that $\models_\alpha \phi_0 = \models_\alpha \phi$.

**Proof.** This can be proved using induction over the structure of $\phi$. We assume that the axioms are valid theorems of $L_\alpha$, which is discussed further in Section 4.1.3.

There are two base cases for the induction: $[\psi_0 \to \epsilon] \phi$ and $[\psi_0 \xrightarrow{\varsigma_m \psi_\varsigma} \psi_0] \phi$. Axioms 4.1.1(ii) and 4.1.1(iii) respectively can be applied from left to right, removing the $[\ ]$ operators. Applying the induction hypothesis to $\phi$ in both cases yields a result that is in $L$.

For the case of sequential composition, choice, variable declaration, and name references, one can apply Axioms 4.1.1(iv)-(vi) respectively, from left to right. Protocol definitions form a 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 exception is for recursively-defined protocols, which will unfold an infinite number of times.

Regarding the infinite unfolding of names, we note that for a finite $L_\alpha$ proposition, this unfolding can be terminated by only applying Axiom (vii) if the precondition of $N$ holds. For example, consider the proof $\psi_0 \to [N] \phi$. If $\psi_0$ does not satisfy the precondition of $N$, then this proposition holds

---

2 For simplicity, we assume that $x$ is not free in $\phi$. If this is not the case, $x$ must be renamed to a fresh variable on the right-hand side of the equivalence.
regardless of $\phi$. For $N$ to be all-terminating, then it must be that before each recursive call to $N$, the state changes, and that eventually, the state will not satisfy the precondition of $N$. A proof of the form $|N|\phi$ can be structured as $(\psi_0 \lor \ldots \lor \psi_n) \rightarrow |N|\phi$, in which $(\psi_0 \lor \ldots \lor \psi_n)$ is the precondition of $N$, and $\psi_i$ is the precondition of a single path through the protocol (Miller and McBurney (2008b) discuss how to calculate preconditions). Due to the assumption that the underlying constraint system is finite, there must be a finite number of preconditions. By checking whether the precondition of $N$ is satisfied before unfolding, the proof will terminate.

This is a useful result, because it shows that, given any finite $L_\alpha$ proposition, a proof of a PDL proposition can be reduced to a proof of a constraint, which can then be proved using a constraint solver. This is similar to regression in the situation calculus (Levesque et al., 1998), which transforms a query about future states into a query about the current state. Theorem 1 has implications for automation of proofs, and means that an agent can prove properties about protocol entailments using only the axioms and a constraint solver, and can be designed to derive properties about any new protocols that it learns.

However, there are two problems with this axiomatisation. First, it is not possible for an agent to prove that a recursively-defined protocol is all-terminating. Second, although the unfolding of name references terminates under finite domains, this unfolding could take up considerable memory and CPU time — more than is available.

If the protocols are non-recursive, then this proof system is adequate. Proving termination is straightforward (we just have to prove there are no recursive calls), and the unfolding of name references will terminate, leaving us with a constraint in $L$, which can then be proved using the constraint solver.

4.1.3. Soundness and Completeness of Finite $L_\alpha$. A logic is sound if any proposition that is provable in the logic, is true. A logic is complete if any true proposition is provable.

Theorem 2: Finite $L_\alpha$ is sound and complete.

Proof. Taking the above definition of sound, it follows that our logic is sound if we cannot prove any proposition that is not true, therefore, soundness is demonstrated by proving that each of the axioms are valid. A proof of this is shown in Appendix A. The modus ponens inference rule is trivially justified.

A logic is complete if we can prove any true proposition. A completeness proof for $L_\alpha$ is 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 proposition. To do this, we observe from Theorem 1 that every proposition in finite $L_\alpha$ can be reduced to a constraint in $L$. Taking this result into account, and our assumption that $L$ is complete, we can construct a proof for $\phi$ by reducing it to a constraint, and proving the constraint using the entailment operator of $L$. The soundness of the logic ensures that the reduction is correct, so the proof is also correct, and finite $L_\alpha$ is complete.

4.2. A Deductive Proof System for Infinite $L_\alpha$

Axiom 4.1.1(vii), which is used to prove properties about name references, is sound; however, applying this axiom to infinite protocols results in an infinitely long proof, making finite $L_\alpha$ system incomplete for infinite protocols.

4.2.1. Axiomatisation of Infinite $L_\alpha$.

Definition 8: Induction inference rule.

To overcome the problem of infinite proofs, we propose an additional inference rule. For this rule, we introduce new syntax. First, the expression $\psi\{N \mapsto \alpha\}$ represents unfolding (or overriding); that is, the proposition $\psi$ with all occurrences of $N$ occurring in a protocol replaced with $\alpha$, which represents a single unfolding of the reference to $N$ when $N \models \alpha$. The semantics of a modified
protocol, \( \beta(N \rightarrow \alpha) \), is defined as \([\beta](e\{N \rightarrow [\alpha](e))\). Second, the shorthand \( \Omega \) represents the non-executable (bottom) protocol, defined as false \( c \text{false} \) false. This protocol cannot be executed because the precondition is false.

Using this syntax, we define a new rule, called the induction rule, based on Scott induction (Scott and de Bakker, 1969):

\[
(D \setminus \{N(y) \equiv \alpha\}, \psi\{N \rightarrow \Omega\}[x/y]),
\]

\[
\frac{(induction)}{(D \setminus \{N(y) \equiv \alpha\} | \psi \vdash \psi\{N \rightarrow \alpha[x/y]\}) \quad \text{where } N(y) \equiv \alpha \in D.}
\]

This rule is somewhat complicated at first glance. Instead, consider the following simplification, which ignores \( D \), and has no parameters:

\[
\frac{(induction2)}{[\Omega] \phi,}
\]

\[
\frac{(\{N\phi \vdash [\alpha] \phi\} | N \equiv \alpha \in D.}
\]

The rule states that, to prove \( [N]\phi \), first prove the base case, \( [\Omega] \phi \), which proves that \( \phi \) holds after the “zero-th” unfolding of \( N \) (in fact, the proposition \( [\Omega] \phi \) always holds, so can be omitted from the rule induction2). Then, prove the inductive step: if \( N \) holds for an arbitrary unfolding (the \( i \)-th unfolding), prove that it holds for one more (the \( i+1 \)-th unfolding). If both of these can be proved, then it holds for all unfoldings, so \( [N] \phi \).

The idea behind this rule is straightforward to demonstrate. Consider the protocol:

\[
N \equiv \psi c \rightarrow \psi' \cup (\phi c \rightarrow \phi'; N).
\]

This protocol can be summarised using the following infinite tree structure

\[
\text{...}
\]

\[
\text{...}
\]

\[
\text{...}
\]

\[
\text{...}
\]

\[
\text{...}
\]

\[
\text{...}
\]

in which the dashed arrow indicates that the unfolding of \( N \) continues infinitely. In this tree, we can see that the only terminating states are those at the nodes labelled \( \psi' \). To prove \( [N]\psi' \), we need to prove that \( \psi'' \) is satisfied in all terminating states. To prevent the infinite unfolding, we assume at the top level (the \( \bullet \) node) that \( \psi'' \) holds in every end state below \( \phi' \), and prove this for the remaining end states, of which there is only one: \( \psi' \). One can see that this is sound, because if we unfold \( N \), the only terminating states are the \( \psi' \) states. These states are able to be deduced directly from the finite definition of \( N \), and therefore, the proof is finite.

Now, we build on the rule induction2 by generalising for cases in which the proof is about a proposition that contains a reference to \( N \), rather than about \( N \) itself. The argument for this case is the same as the simplified case: to prove \( \psi \), first prove the base case of substituting \( \Omega \) for \( N \) (the “zero-th” unfolding of \( N \)), and then try to prove that \( \psi \) holds for one unfolding of \( N \), written \( \psi\{N \rightarrow \alpha\} \), under the assumption that \( \psi \):

\[
\frac{(induction3)}{\psi\{N \rightarrow \Omega\},}
\]

\[
\frac{(\psi \vdash \psi\{N \rightarrow \alpha\} \quad \text{where } N \equiv \alpha \in D.}
\]

Returning to the original induction rule, we want to prove the proposition \( \psi \) with respect to the protocol specification \( D \). As with induction3, we prove the base case and the inductive case. The main difference is the consideration of the protocol specification \( D \). In this case, we include \( D \), and furthermore, the definition of \( N \) is removed from \( D \) for the proofs of the premise of the rule, because its definition is not required. Another minor difference is the expression \( \alpha[x/y] \), which is the definition of \( N \) with the variable \( y \) substituted by \( x \).
Despite the existence of parameters in the RASA language, the induction inference rule is sufficient for proving properties about recursive protocols. Parameters in RASA exist to make variable substitution more readable. Call-by-value and nested recursive calls (e.g. \(N(N(x))\)) are not permitted by the language syntax. In fact, nested recursive calls do not make sense semantically because \(N(x)\) is not an expression in the underlying constraint language. By defining parameters as a straightforward variable substitution, the induction rule could be written without parameters, and we could add a substitution rule of the form (omitting \(D\)):

\[
\text{(substitution)} \quad \frac{\phi[y/x] \rightarrow \psi[y/x]}{\phi \rightarrow \psi}
\]

The induction inference rule can now operate as if the named protocols are parameterless. This approach is taken by Hoare (1971) for dealing with procedures and parameters.

Performing this inductive proof is only necessary if \(N\) is infinite. Otherwise, the finite \(L_\alpha\) proof system is sufficient.

4.2.2. Automation of Proofs in Infinite \(L_\alpha\). The addition of the induction inference rule has an impact on the automation of proofs in infinite \(L_\alpha\). Future work will determine how much automation is possible, but what is clear is that it cannot be performed using a straightforward syntactical transformation of propositions to constraints, as can be done in finite \(L_\alpha\). Harel et al. (2000) show that even simple extensions to regular PDL are undecidable, so automatically proving properties about an arbitrary non-regular protocol is not possible.

Section 5 discusses how, for protocols that conform to a specific syntactic form, proofs in infinite \(L_\alpha\) can be reduced to proofs in finite \(L_\alpha\). This allows the finite proof system to be used.

4.2.3. Soundness and Completeness of Infinite \(L_\alpha\). The soundness and completeness of \(L_\alpha\) relies on the soundness and completeness of the induction inference rule, which is, in turn, related to the following definition.

**Definition 9:** (Anti-)Continuity

A function, \(f \in X \rightarrow Y\), between two partially ordered sets, \(X\) and \(Y\), is (i) continuous, (ii) anti-continuous, if, for a sequence of the form \(x_1 \leq x_2 \leq \ldots\), where \(x_i \in X\), its holds that

(i) \(f(\sqcup_i x_i) = \sqcup_i f(x_i)\)

(ii) \(f(\sqcap_i x_i) = \sqcap_i f(x_i)\),

in which \(\sqcup_i\) and \(\sqcap_i\) are the least upper bound and greatest lower bound for a sequence of elements respectively. From this, we can define (anti-)continuity for \(L_\alpha\) propositions.

A proposition, \(\psi\), is (i) continuous, (ii) anti-continuous with respect to \(N\) if, for a sequence of protocols, \([\alpha_1] \subseteq [\alpha_2] \subseteq \ldots\).

(i) \(\psi[N \rightarrow \bigcup \alpha_i] \iff \bigvee \psi[N \rightarrow \alpha_i]\)

(ii) \(\psi[N \rightarrow \bigwedge \alpha_i] \iff \bigwedge \psi[N \rightarrow \alpha_i]\),

in which \(\bigcup \alpha_i\) is the choice between all \(\alpha_i\); that is \(\alpha_1 \cup \alpha_2 \cup \ldots\). In this example, the “function” \(f\) is one that determines the truth of a proposition.

For example, proposition \([N]\phi\) is anti-continuous in \(N\) because

\([\alpha_1 \cup \alpha_2 \cup \ldots] \phi \iff [\alpha_1]\phi \land [\alpha_2]\phi \land \ldots\)

holds from Axiom 4.1.1(v).

As done by de Bakker (1980), we define continuity and anti-continuity syntactically:

1. If \(N\) does not occur in \(\psi\), then \(\psi\) is both continuous and anti-continuous in \(N\).
2. \([N]\phi\) is anti-continuous in \(N\).
3. If \(\phi\) is anti-continuous in \(N\), then \([\alpha]\phi\) is anti-continuous in \(N\).
4. If \(\psi\) is continuous in \(N\) then \(\neg \psi\) is anti-continuous in \(N\).
5. If \(\psi\) is anti-continuous in \(N\) then \(\neg \psi\) is continuous in \(N\).
6. If \(\phi\) and \(\psi\) are both (anti-)continuous in \(N\), then \(\phi \land \psi\) is (anti-)continuous in \(N\).
Put simply, a proposition, \( \psi \), containing a recursive reference \( N \), is anti-continuous in \( N \) if the reference to \( N \) in \( \psi \) is not within a negation.

The following definition and lemma are used to prove the soundness and completeness of the induction rule.

**Definition 10:** We define shorthand to represent infinite unfoldings of a name \( N \). Assume that \( N \equiv \alpha \), then \( \alpha_i \) is defined as follows:

\[
\begin{align*}
\alpha_0 & \equiv \Omega \\
\alpha_{i+1} & \equiv \alpha(N \mapsto \alpha_i).
\end{align*}
\]

**Lemma 1:** \( \psi(N \mapsto \bigcup \alpha_i) \rightarrow \psi \).

Proof. This lemma states that, for any proposition that contains a reference to a name \( N \), unfolding \( N \) an infinite number of times does not change the truth value of that proposition.

It holds trivially that \( \psi(N \mapsto N) \rightarrow \psi \), so we need to prove that \( N \) is equivalent to \( \bigcup \alpha_i \). The definition of \( \llbracket N \rrbracket(e) \) in Section 2.3 is of the least fixpoint, \( \mu F \). Given this, \( \mu F \) is equivalent to \( \bigcup f_i \), in which \( f_0 = \llbracket \Omega \rrbracket(e) \) and \( f_{i+1} = [D, \alpha][e\{N \mapsto f_i\}] \). Using this definition, we can prove \( N = \bigcup \alpha_i \) by proving the following:

\[
\bigcup \alpha_i[e\{N \mapsto f_{i-1}\}] = \bigcup \alpha_i[N \mapsto \alpha_{i-1}][e],
\]

which holds from the continuity of the function \( \llbracket \rrbracket \). The function \( \llbracket \rrbracket \) can be shown to be continuous by taking a sequence \( [\alpha_1] \subseteq [\alpha_2] \subseteq \ldots \), and noting that

\[
\bigcup \alpha_i[e] = \bigcup \alpha_i[\llbracket e \rrbracket)
\]

holds from the definition of the choice operator, \( \cup \).

**Theorem 3:** The inference rule induction is sound and complete for anti-continuous \( \psi \).

Proof. To prove this, we need to prove that from \( \psi(N \mapsto \Omega) \) and \( \psi \vdash \psi(N \mapsto \alpha_i) \), we can infer \( \psi \).

If we apply the base case and induction of the rule, the result is \( \forall i \in \mathbb{N} \bullet \psi(N \mapsto \alpha_i) \).

This is equivalent to \( \bigwedge_i \psi(N \mapsto \alpha_i) \).

From the anti-continuity of \( \psi \), this is further equivalent to \( \psi(N \mapsto \bigcup \alpha_i) \), which is equivalent to \( \psi \) from Lemma 1.

We have proved that the induction rule is sound and complete only for propositions that are anti-continuous in \( N \). However, this does not imply that only anti-continuous propositions can be proved using the proof system. For example, consider the proposition \( \neg[N] \phi \), which is continuous in \( N \). We can construct a proof for this by constructing one for \( [N] \phi \), which is anti-continuous, and negating the result. In fact, for any proposition \( \psi \) that is continuous in \( N \), a proof can be constructed by proving \( \neg \psi \) and negating the result.

Propositions that are neither continuous nor anti-continuous are more difficult to deal with. The trivial situation is for a proposition \( \psi \wedge \phi \), in which \( \psi \) is anti-continuous and \( \phi \) is continuous. We can split this into two proofs: one about \( \psi \) and one about \( \neg \phi \), negating the result of \( \neg \phi \), and then conjoining these two results. Constructing proofs for cases such as \( [\beta] \neg[N] \phi \) would prove more difficult.

4.3. Relationship to PDL

We claim that our logic is an instantiation of PDL. Readers familiar with PDL may have noted that several axioms of PDL are not listed in our proof system. These axioms are omitted because they can be derived from the minimal set of axioms and inference rules of \( \mathcal{L}_\alpha \), due to the existence of name references in the \( \mathcal{RASA} \) language. The following theorem states that the following PDL axioms not listed in the axiomatisation of \( \mathcal{L}_\alpha \) are theorems of \( \mathcal{L}_\alpha \):

**Theorem 4:** The following formulae are valid theorems of \( \mathcal{L}_\alpha \):
(i) $[\alpha](\phi \rightarrow \psi) \rightarrow [\alpha]\phi \rightarrow [\alpha]\psi$

(ii) $\phi \land [\alpha][\alpha^*]\phi \rightarrow [\alpha^*]\phi$

(iii) $\phi \land [\alpha^*]((\phi \rightarrow [\alpha]\phi) \rightarrow [\alpha^*]\phi)$

Proof. See Appendix B.

This is a useful result because, having shown that all of the axioms of PDL are valid theorems in $\mathcal{L}_\alpha$, any provable theorem of PDL is also a provable theorem of $\mathcal{L}_\alpha$. Such a result means that we can use a large body of existing work on PDL to reason about RASA protocols. For example, any theorem of PDL proved using its axiomatic system can be demonstrated in $\mathcal{L}_\alpha$ using the same proof.

5. FROM INFINITE $\mathcal{L}_\alpha$ TO FINITE $\mathcal{L}_\alpha$

In Section 4.2, we question how much of infinite $\mathcal{L}_\alpha$ is automatable. The problem regarding this proof system is that the induction inference rule is not a syntactic manipulation of a proposition. In this section, we show that, for certain types of infinite protocols, their proof can be recast into finite $\mathcal{L}_\alpha$.

We note that some theorems discussed in this section are not theorems in the proof system, but theorems about the proof system. In other words, they are inference rules. We use $\phi \vdash \psi$ to mean that a proof of $\phi$ is equivalent to a proof of $\psi$, and that they are equivalent theorems in the proof system, but not (necessarily) equivalent propositions.

In this section, we show that, for certain types of infinite protocols, proofs can be transformed to proofs about finite protocols. Using this approach, an agent equipped with the finite $\mathcal{L}_\alpha$ axioms and a constraint solver can prove properties about these types of infinite $\mathcal{L}_\alpha$ propositions.

The reason we are interested in investigating equivalent theorems is for the purpose of characterising the preconditions and postconditions of protocols. That is, given a protocol $\alpha$, obtain the weakest precondition and the strongest postcondition of that protocol. This produces a summary of protocol outcomes and the preconditions under which those outcomes can be achieved. These summaries, called outcome characterisations (Miller and McBurney, 2008a), because they can be characterised to the protocol definition, are $\mathcal{L}_\alpha$ theorems (they are theorems about a specific protocol). We express these characterisations in the form $\psi_0 \rightarrow [\alpha]\phi_0$, in which $\psi_0$ is the weakest precondition under which $\alpha$ terminates, and $\phi_0$ is the strongest postcondition achieved under the precondition $\psi_0$. An agent can use these characterisations to quickly and correctly assess whether a protocol achieves a given goal.

5.1. Properties of $^*$ and $^+$

Recall from Section 2.4 that $\alpha^*$ is defined as $N$, where $N = \epsilon \cup (\alpha; N)$, and $\alpha^+$ is defined as $\alpha; \alpha^*$. It is difficult to derive a fixpoint for the postcondition of protocols of this format because they are infinite: $\alpha$ can be iterated over any number of times. Theorem 4 demonstrates how to prove properties about protocols of this format, however, those theorems cannot be used to reduce propositions to constraints, because they do not remove the $^*$ operator.

For the purpose of characterisation, we are interested in determining the strongest postcondition when executed under the weakest precondition, as we do for characterisation, so we can define several inference rules to help us.

**Theorem 5**: The following is a valid inference rule of $\mathcal{L}_\alpha$:

$$[\alpha]\phi \vdash [\alpha^+]\phi.$$  

Proof. The proof for this is straightforward. If $\alpha$ achieves $\phi$ for every outcome in every model, then no matter how many times $\alpha$ is iterated, it will achieve $\phi$. Similarly, if any number of iterations of $\alpha$ achieves $\phi$, then one iteration will achieve $\phi$ also.

The implications of this are clear: for any conjectured theorem $\vdash [\alpha^+]\phi$, one needs only to show...
\[\vdash [\alpha] \phi; \text{ and to characterise postconditions } \alpha^+, \text{ one needs only to characterise } \alpha. \text{ From here, it is a small step to show } \phi \land [\alpha] \phi \vdash [\alpha^+] \phi.\]

This can not be used directly to characterise a protocol of the format \(\alpha^+\) under its weakest precondition \(\psi_0\). For this, we propose the following.

**Theorem 6:** If \(\psi_0\) is the weakest precondition of \(\alpha\), then the following is a valid inference rule of \(L_\alpha\):

\[\psi_0 \rightarrow [\alpha] \phi \vdash [\alpha^+] \phi.\]

Proof. The proof for this is sketched as follows. The right-to-left case holds trivially, so we do not discuss this further. For the left-to-right case, if \(\psi_0\) holds, then 1 iteration of \(\alpha\) will result in \(\phi\). At this point, one of two properties hold: \(\neg (\phi \rightarrow \psi_0); \text{ or } \phi \rightarrow \psi_0\). If the former, then \(\alpha\) cannot iterate again, because \(\psi_0\) is its weakest precondition, and the precondition is not satisfied. This implies that the final postcondition is \(\phi\). If the latter, then \(\alpha\) can either terminate, leaving the postcondition as \(\phi\), or it can iterate again. If it iterates again, we know that the result will be \(\phi\), because \(\psi_0\) holds and \(\alpha\) preserves \(\phi\) whenever \(\psi_0\) holds. Therefore, the resulting postcondition is \(\phi\). Applying this argument inductively, we see that the strongest postcondition of \(\alpha^+\) is \(\phi\).

5.2. Properties of Recursive Protocols

As discussed in Sections 4.1 and 4.2, the induction inference rule may prove difficult to apply automatically. In this section, we demonstrate that a restricted set of propositions about infinite protocols can be recast into the domain of finite \(L_\alpha\), offering a way to automate their proof.

**Definition 11:** Choice normal form.

We say that a protocol is in choice normal form if and only if it is a choice between one or more protocols not involving other choice protocols. That is, for a protocol \(\alpha_1 \cup \ldots \cup \alpha_n\), each of \(\alpha_1\) to \(\alpha_n\) does not contain a choice. This is analogous to disjunctive normal form in Boolean logic.

**Theorem 7:** Any \(\mathcal{RASA}\) protocol can be reduced to choice normal form.

Proof. Any protocol that fits our definition can be reduced to choice normal form by using the property of sequential composition distributing over choice:

\[\alpha; (\beta \cup \gamma) \equiv (\alpha;\beta) \cup (\alpha;\gamma),\]

and commutative and associate rules, which have been proved using our logic. Names are not unfolded when reducing to choice normal form. Iteration operators are pushed inwards over choice protocols using

\[(\alpha \cup \beta)^* \equiv \alpha^*; (\beta;\alpha^*)^*,\]

which is an axiom direct from Kleene algebras (Kozen, 1990).

**Definition 12:** Linear choice normal form.

We say that a protocol definition, \(N(x) \equiv \alpha_1 \cup \ldots \cup \alpha_n\), is in linear choice normal form if and only if \(\alpha_1 \cup \ldots \cup \alpha_n\) is in normal form, and both of the following hold:

1. For all choices in \(\alpha_1 \cup \ldots \cup \alpha_n\), there is at most one recursive call to \(N\); and
2. For at least one choice in \(\alpha_1 \cup \ldots \cup \alpha_n\), there is no recursive call to \(N\).

This means that, for a possibly infinite protocol, 1) there exists at most one recursive call in each branch of that protocol definition; and 2) at least one branch is non-recursive, and must terminate.

It is our assertion that we can prove properties about a restricted subset of infinite protocols by transforming them to an equivalent finite protocol. For example, consider the protocol \(N \equiv \epsilon \cup (\alpha; N)\), which is equivalent to \(\alpha^+\) (from the definition of \(^+\)). The proposition \([N] \phi\) is then equivalent to \([\alpha^+] \phi\), and can be proved using the Theorem 5. We generalise this idea to any protocol in linear choice normal form.
Theorem 8: For a protocol \( N(x) \cong \alpha \cup (\beta; N(x); \gamma) \) in linear choice normal form, the following proposition holds:

\[
[N(x)]\phi \vdash [\alpha \cup (\beta; \alpha; \gamma)]\phi
\]

Proof. To prove this, we propose the following lemma:

Lemma 2: \([N]\phi \iff [\alpha \cup (\beta^+; \alpha; \gamma^+) ]\phi\).

For this, we use the following two axioms from our proof system:

\[
\begin{align*}
(iv) & \quad [\alpha; \beta]\phi \iff [\alpha][\beta]\phi \\
(v) & \quad [\alpha \cup \beta]\phi \iff [\alpha]\phi \land [\beta]\phi
\end{align*}
\]

and the Scott induction inference rule. Therefore, to prove our lemma, we assume

\[
[N]\phi \iff [\alpha \cup (\beta^+; \alpha; \gamma^+) ]\phi,
\]

(the assumption of \([N]\psi\) in the inference rule), and then prove this holds for the unfolded definition:

\[
[\alpha \cup (\beta; N; \gamma)]\phi \iff [\alpha \cup (\beta^+; \alpha; \gamma^+) ]\phi.
\]

To prove the above, we rewrite the left-hand side as follows using Axioms (iv) and (v):

\[
[\alpha \cup (\beta; N; \gamma)]\phi \iff [\alpha]\phi \land [\beta]\phi
\]

Using our assumption that \([N]\phi \iff [\alpha \cup (\beta^+; \alpha; \gamma^+) ]\phi\), we perform the inductive step of the proof:

\[
[\alpha]\phi \land [\beta][\alpha \cup (\beta^+; \alpha; \gamma^+) ][\gamma]\phi.
\]

Using our axioms, this can be reduced as follows:

\[
[\alpha]\phi \land (\beta; \alpha; \gamma) \cup (\beta; \beta^+; \alpha; \gamma^+; \gamma)]\phi
\]

\[
\implies [\alpha \cup (\beta; \alpha; \gamma) \cup (\beta; \beta^+; \alpha; \gamma^+; \gamma)]\phi.
\]

We can merge the second and third branches of the choice to \(\beta^+; \alpha; \gamma^+\), which leaves us with a proposition that is equivalent to the right-hand side of the lemma, and therefore, the lemma holds.

Returning to the proof of Theorem 8, we observe from Theorem 5 that proving properties about a protocol \(\alpha^*\) is equivalent to proving the same about \(\alpha\). Using Axiom (iv), any proposition \([\beta^+; \alpha; \gamma^+]\phi\) will be reduced to a proposition about the \(\beta, \alpha,\) and \(\gamma\), which can be proved using finite \(L_\alpha\).

5.2.1. Discussion Point. Theorem 8 provides us with the ability to prove properties about a restricted subset of infinitely recursive protocols: those in linear choice normal form. This discussion point addresses the usefulness of Theorem 8 in the context of proof and characterisation.

First, let us identify which protocols do not fit into this class. Theorem 7 shows that all RASA protocols can be reduced into choice normal form. This leaves us with two classes that cannot be reduced (those that do not fit into Definition 12): 1) protocols in which one of the choice branches contain a non-linear recursive call, for example, \(N(x); \beta; N(x); \alpha\); and 2) protocols in which all of the branches contain a recursive call, meaning that none of the branches terminate.

Protocols in class 2 have no end states, so we can say nothing sensible about their outcomes. In fact, for any protocol \(\alpha\) that never terminates, the proposition \([\alpha]\phi\) holds for any \(\phi\), including false. This does not imply that such protocols are worthless — non-terminating agents and protocols are a regular occurrence. This simply means that other types of characterisation may be required. That leaves us with protocols in class 1: non-linear recursive branches. Our theorem does not apply to these protocols, and these will be considered in future work. However, we have not been able to identify a protocol with this property in the multi-agent systems literature, which leads us to believe our theorem has wide applicability.

We note that, while Theorem 8 assumes a protocol definition of the form \(N \cong \alpha \cup (\beta; N; \gamma)\), any protocol in linear normal form can be expressed in this way. That is, \(\alpha\) may be a choice protocol itself that contains a reference to \(N\), however, as long as one of those choice protocols terminate,
PDL for Reasoning about First-Class Agent Interaction Protocols

this is still in linear normal form. The protocols $\alpha$, $\beta$ and $\gamma$ are not restricted to be atomic protocols; they can be compound protocols.

The recursive branch, $\beta; N; \gamma$, is expressive enough to handle head or tail recursion. For example, to handle tail recursion, simply substitute $\epsilon$ in for $\beta$ and note that $\epsilon; \alpha \equiv \alpha$. From this, it is straightforward to show the following theorems for head and tail recursive protocols respectively:

\[
\vdash [N] \phi \iff [\beta^*; \alpha] \phi \quad \text{where } N \equiv \alpha \cup (\beta; N)
\]

\[
\vdash [N] \phi \iff [\alpha; \beta^*] \phi \quad \text{where } N \equiv \alpha \cup (N; \beta).
\]

We note that these are theorems of the logic, and not inference rules.

6. EXPERIMENTAL EVALUATION

In this section, we briefly discuss some experimental evaluations related to the proof system presented in Section 4.

6.1. Proof System Implementation

The finite deductive proof system has been implemented in Prolog, and is based on the CLP bounds solver, an integer constraint solver with variables. This implementation uses the result from Theorem 1 to reduce a proposition of the form $[\alpha] \phi$ into a constraint. For example, the following Prolog clause is used to prove that a sequential composition achieves a proposition $\Phi$ for all end states:

\[
\text{Premise} \Rightarrow \text{allp}(\text{Ptree1 then Ptree2, Phi}) :- \\
\text{Premise} \Rightarrow \text{allp}(\text{Ptree1, allp(Ptree2, Phi)}).
\]

In this example, $\text{allp}$ ("all paths") represents the $[\ ]$ operator, and $\text{then}$ represents sequential composition. In this implementation, proofs are of the form $\text{Premise} \Rightarrow \Phi$ (proofs of the form $\text{allp}(\text{Ptree, Phi})$ are proved using $\text{true} \Rightarrow \text{allp}(\text{Ptree, Phi})$). This makes it straightforward to map these proofs into entailments in the underlying constraint system.

The source code for the implementation and the test suite can be downloaded from the RASA web page at http://www.csse.unimelb.edu.au/~tmill/rasa/.

6.2. Characterisation and Matching Protocols

In other work (Miller and McBurney, 2008a), we have used this logic to characterise protocols in a protocol library. We proposed a method for automatically characterising the outcomes of a protocol, and matching which protocols in a library achieve a given goal, using the characterisations. The $\mathcal{L}_\alpha$ logic is used to represent the characterisations, and to prove that the summary and matching methods are sound and complete.

The characterisation method has been implemented using Prolog, and as with the proof system, is based on the CLP bounds solver. The source code for the implementation and the test suite can be downloaded from RASA web page.

6.3. Testing the Characterisation Method and Proof System

In software testing, in the absence of an automated test oracle — a program that determines whether the output of a test case is correct with respect to its specification — it is common to use identity relations to define oracles. For example, if we are testing a program that computes the cosine trigonometric function, and have a program that computes the sine trigonometric function, we can make use of the identity relation $\cos(\theta) = \sin(\frac{\pi}{2} - \theta)$.

Using this theory, we used the implementations of the proof system to test the characterisation method implementation. Given that an characterisation is valid if it is a theorem of the logic, we tested whether the characterisations that our method derives are sound using the proof system. Despite the fact that characterisation method was proved using the proof system, the diversity of
the methods in deriving similar information is large enough such that we believe that the proof system can be used as an oracle for the characterisation algorithm.

Testing these was done by writing a set of Prolog clauses that randomly derived a library of protocols, and applied the characterisation method to each of the protocols in the library. Characterisations are in the form of theorems, so each characterisation is checked using the proof system. A failure to prove a characterisation identifies an inconsistency between the proof system and the characterisation method, which implies a fault in one of these, or both.

The results from this allowed us to identify three problems: one in the characterisation of variable declarations, one in the characterisation of atomic protocols, and one in the proof of variable declarations.

The source for randomly deriving protocols, characterising the protocols, and testing the characterisations can be downloaded from RASA web page.

7. RELATED WORK

In this section, we discuss the work most closely related to ours.

7.1. Protocol Specification

Process algebras, such as CSP (Hoare, 1985), CCS (Milner, 1980), and the π-calculus (Milner, 1999) 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 (Smith and Derrick, 2002), 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 (de Silva et al., 2003) and on declarative specification languages (Desai et al., 2005; Desai and Singh, 2007; Yolum and Singh, 2004), as well as an algebraic language similar to RASA called the Lightweight Coordination Calculus (Robertson, 2004). Miller and McGinnis (2007) present a detailed comparison of these languages, including RASA. None of the cited work provides any formal method for reasoning about these languages. We believe that the work in this paper would be directly applicable to these languages with some minor changes to the proof system.

van Eijk et al. (2003) have used constraint systems and process algebra for modelling multi-agent systems. The verification framework they propose focuses on interaction between the agents. Their logic resembles PDL, in spirit, if not in syntax, in that it supports reasoning about outcomes; however, unlike our logic, it also allows reasoning about individual messages. Recursive definitions are proved using an inference rule that appears to be Scott induction; however, van Eijk et al. do not discuss automation of proofs. We do not consider their models to be first-class protocols, as they explicitly use the beliefs of individual participants to model the semantics.

7.2. Propositional Dynamic Logic

Propositional dynamic logic (PDL) (Harel et al., 2000) is clearly related to our work. We view the $L_{\alpha}$ logic as an instantiation of PDL, as discussed in Section 3.

Several authors have considered the decidability of PDL for non-regular programs (that is, programs with recursive definitions) (Harel et al., 1983; Löding and Serre, 2006). Such work tends to focus on the types of programs that are decidable in PDL, rather than general proof systems, and does not use inductive rules such as Scott induction. Levant (2008) investigates $\mu$PDL, a PDL for recursive programs. Levant uses a rule that is similar to Scott induction to prove properties of recursive programs.

With respect to PDL for reasoning in agents, van Riemsdijk et al. (2006) have derived a PDL for 3APL, a programming language for goal-direct agents. Like RASA, 3APL permits recursive definitions. Van Riemsdijk et al. use an inference rule based on Scott induction to reason about recursive 3APL programs. They discuss the issue of automation, but leave this as future work. It
does not appear that this logic is used by the agents themselves, but is instead used to provide a
more declarative semantics for the 3APL language and to verify 3APL programs.

There has been some work using PDL to specify interaction protocols. PDL has been extended
(Paurobally et al., 2005) 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
(Brak et al., 2004). The main difference between this work and our approach is that we use PDL for
characterising, representing, and reasoning about protocols constructed in the RASA language, in
which the protocol definitions themselves provide the allowed behaviour, while Paurobally et al.
(2005) and Brak et al. (2004) define a protocol as a collection of PDL formulae, which would
make them difficult to use as first-class protocol languages, especially PDL-BI, which is based on
ungrounded BDI logic.

7.3. Action Languages

The RASA language is an action language that restricts actions to be messages sent along a
channel. We believe that the work in this paper has wide applicability, and could be used to reason
about actions other than messages.

Support for formally reasoning about actions is already well-matured in artificial intelligence
research. Several languages exist for reasoning about actions, such as the situation calculus (Reiter,
2001) and the event calculus (Kowalski and Sergot, 1986). In our work, we use PDL instead of other
action languages for two major reasons. First, actions languages such as the situation calculus and
the event calculus use propositional logic to specify the situations/states of the world, rather than
constraint languages. A particular criterion of the RASA language is that constraint languages are
used to specify messages and states, rather than propositional logic. Second, PDL offers a compact
way of representing protocol outcomes that is useful for our work in characterisation of protocols
(Miller and McBurney, 2008a).

As far as the authors are aware, there is no semantics defined for recursive action definitions in
the event calculus or the situation calculus. The GOLOG programming language (Levesque et al.,
1997), which is based on the situation calculus, supports compound actions (called complex actions)
by treating them as abbreviations of underlying situation calculus expressions. These abbreviations
resemble the axioms in Section 4. Recursive actions can be defined using parameterised procedures.
The semantics of recursive procedures is defined using a Scott-Strachey least fixed point definition
(Stoy, 1977), on which Scott induction is based. Therefore, an induction inference rule similar to
that in Section 4.2 would be straightforward to derive for GOLOG recursive procedures.

It is possible that other action languages support recursive action definitions, however, the
authors are unaware of any work that discusses transforming recursively-defined actions/plans into
iteratively-defined actions/plans. The authors believe that non-linear plans would be more common
than non-linear protocols, so the approach taken in this paper is likely to be more applicable for
protocols instead of plans.

7.4. Goal-Directed Plans

There is a clear relation between our work, and that of goal-directed (BDI) plans, such as in
the Procedural Reasoning System (PRS) (Georgeff and Lansky, 1987). At first glance, one can see
several similarities, such as libraries of plans, annotating plans with their outcomes, and plans being
made of a sub-plans, which are all aspects that we have incorporated into our framework. In fact,
the motivation behind much of this work is based around goal-directed agents, and design decisions
are influenced by current approaches to goal-directed agency. There are fundamental differences
between PRS-like frameworks, and RASA. The first and biggest difference is in their aims. With
respect to interaction, protocols are public documents that specify the rules of interaction by which all
participating agents should abide, whereas plans, such as those found in PRS, are typically private,
and specify how the individual agent should behave with respect to the protocol rules. One can view
the protocol as the rules, and an agent’s plan as the strategy to best achieve its goals within those
rules.

A further difference is in the way BDI plans are specified. Preconditions and postconditions of
plans are supplied by the developer, and represent the preconditions and postconditions of the entire
plan, not the individual atomic actions within the plan. As a result, using a logic such as PDL to reason about these would only make sense if one wanted to compose new plans from existing plans. Clement and Durfee (Clement and Durfee, 1999) present a method for characterising precondition and postconditions of hierarchical task network (HTN) plans for plan composition. Their method is similar to our method for characterisation (Miller and McBurney, 2008a), however, they do not consider iterative or recursive plans.

7.5. Recursion Elimination

Recursion elimination has been investigated since the advent of procedural programming, and many modern compilers implement automatic elimination of tail recursion to remove use of stack frames during program execution. The authors are unaware of any work using PDL to prove recursion elimination methods.

Most work on recursion elimination uses approaches similar to ours, in that recursive programs were transformed into iterative programs. Much of this is built on early work such as (Bird, 1977; Auslander and Strong, 1978). Bird (1977) and Auslander and Strong (1978) present step-by-step methods for recursion elimination that use stacks to record intermediate calculations. Their approaches are not applicable to our work due to the absence of stacks in \( RASA \). Bird (1977) and Auslander and Strong (1978)'s contributions both consider programming languages that support call-by-value and nested function calls. As such, our approach is not applicable to these languages. The simpler nature of the \( RASA \) language allows us to provide a more elegant solution (a simple inference rule), rather than a step-by-step method for transformation.

Arsac and Kodratoff (1982) propose an interesting solution that generalises recursive programs into tail recursion, which can then be transformed into an iterative program in a straightforward manner. However, their work focuses on rewriting expressions, for example, an arithmetic expression \( x \ast f(y) + z \), in which \( f(y) \) is a recursive call to a function. \( RASA \) does not support functions with return values, so such a method is unnecessary. Similarly, our method is not applicable to such expressions because it does not support functions with return values.

Liu and Stoller (2000) provide a method from transforming recursion into iteration based on incrementalisation. An increment is identified; that is, the change of value in arguments from one recursive call to the next, deriving an incremental version of the program, and transform this into an iterative version. This method is only applicable to programs that contain increments, and is therefore restricted to all-terminating programs — a restriction that our method does not have. It is also difficult to see whether it is possible to automatically identify the increments.

Some of the work in this paper may be applicable to programming languages with recursive, parameterless procedures. However, some theorems, such as Theorems 5 and 6, may not be applicable, due to the absence of explicit preconditions in programming languages. Further investigations could be done to conclude whether this is the case. The authors feel that theorems such as these are of more interest to the artificial intelligence community than to the programming language community.

8. CONCLUSIONS

In this paper, we have presented a propositional dynamic logic for reasoning about and representing outcomes of protocols in \( RASA \), a language for executable protocol specification. A proof system for the logic is presented, and shown to be sound and complete. This proof system consists of two sub-parts: one for proving properties about terminating protocols, which is automatable, as demonstrated by an implementation of this; and one for proving properties about non-terminating protocols, for which automation may prove difficult. We prove that proofs about certain types of recursively-defined protocols can be reduced to proofs about protocols that have no recursion, making them automatable. We believe that this work can be generalised to other first-class protocol languages, such as those discussed in the related work. In fact, treating non-terminating definitions as terminating definitions for the purpose of proofs is likely to be applicable to action languages in general; however, our interest is in protocols, so we have not investigated this.

In other work (Miller and McBurney, 2008a), we have presented several methods that use our logic to characterise and match protocols for goal-directed agents. Protocols in a library are annotated
with their outcome characterisations, and using the proof system, suitable protocols can be matched from the library via their characterisations. In Miller and McBurney (2008b), we have identified the conditions that must hold for two protocols to be composed; for example, $\alpha; \beta$ is only a valid composition if the precondition of $\beta$ is enabled by the postcondition of $\alpha$.

By treating interaction protocols as first-class entities, RASA permits protocols to be dynamically inspected, referenced, composed, shared, and invoked 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 (Luck et al., 2005).

Acknowledgements

The authors would like to thank Jarred McGinnis for his valuable feedback on this work, as well as Wiebe van der Hoek and Peter Schachte for their discussions on the proof system. The authors would also like to thank the anonymous reviewers for their detailed reviews. We are particularly grateful to one reviewer for helping to significantly improve the induction rule.

The authors are grateful for financial support from the EC-funded PIPS project (EC-FP6-IST-507019), the EC-funded ASPIC project (IST-FPC-002307), and the EPSRC Market-Based Control project (GR/T10657/01).

REFERENCES


APPENDIX A. Soundness Proof for Finite $L_\alpha$

In this appendix, we prove that the proof system for $L_\alpha$ is sound (see Theorem 2). That is, we prove that, for any proposition $\phi$ in $L_\alpha$, if $\phi$ is provable then $\phi$ is true:

$$\vdash \phi \implies \models \phi.$$  

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

Definition 13: Upward Closure of Constraints.

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

$$\uparrow \phi_0 = \{ \psi \in L \mid \psi \supseteq \phi_0 \}.$$  

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

Theorem 9: $L_\alpha$ is sound.

Proof. To prove the soundness of $L_\alpha$, we prove that each of the axioms in the proof system is valid.

Axiom (i) $\models [\alpha](\phi \land \psi) \leftrightarrow [\alpha]\phi \land [\alpha]\psi.$
First, we prove the equivalence from right to left. That is, we prove $[\alpha]\phi \land [\alpha]\psi \rightarrow [\alpha](\phi \land \psi).$ Consider arbitrary start and end states $\psi_0$ and $\psi'_0$ of protocol $\alpha$. From the premise, we know that $\psi'_0 \models \phi$ and $\psi_0 \models \psi$. From the definition of $\land$, we know that $\psi'_0 \models \phi \land \psi$. Because $\psi'_0$ is an arbitrary end state, then this must be true for all end states, therefore, $\psi_0 \models [\alpha](\phi \land \psi).$ $\psi_0$ is arbitrary, so this holds for every state. The left-to-right case is simply the reverse.

Axiom (ii) $\models [\psi_0 \rightarrow \epsilon]\phi \leftrightarrow \psi_0 \rightarrow \phi.$
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 $\psi_0$ holds. Therefore, if a proposition, $\phi$, holds at the current state, $\phi$ will also hold after $\epsilon$, and vice-versa. If $\psi_0$ 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) $\models [\psi_0 \xrightarrow{\psi_0 \models \phi} \psi'_0]\phi \leftrightarrow \psi_0 \rightarrow (\psi_m \cup \psi'_0 \rightarrow \phi)[x' / x].$
To prove this, we break up the equivalence into two cases: all models in which $\psi_0$ holds; and all models in which $\psi_0$ does not hold.

Case $\psi_0$ does not hold: If $\psi_0$ does not hold in the current model, then $[\psi_0 \xrightarrow{\psi_0 \models \phi} \psi'_0]\phi$ is trivially
true, because the precondition of the protocol is not enabled, and therefore the set of traces is empty. Similarly, if \( \psi_0 \) does not hold, \( \psi_0 \rightarrow (\ldots) \) is trivially true.

Case \( \psi_0 \) holds: Firstly, we expand the definition of the left-hand side using the denotational semantics from Section 2.3:

\[
\forall \phi_1, h, \phi_1' \in \{(\phi_0, c, \phi_m, \phi_0 \oplus (\phi_m \cup \phi_0')) | (\phi_0 \supseteq \psi_0) \land (\phi_m \cup \phi_0 \supseteq \psi_m \cup \psi_0')\} \bullet \phi_1' \models \phi.
\]

So, to prove that \( [\psi_0 \xrightarrow{c,x \mapsto m} \psi_0']\phi \), we must prove that for every end-state, \( \phi_1' \) and refined message \( \phi_m, \phi_0 \) satisfies \( \phi \). Noting that \( \phi_1, h, \) and \( \phi_1' \) corresponding with \( \phi_0, c, \phi_m, \) and \( \phi_0 \oplus (\phi_m \cup \phi_0') \) respectively in the set comprehension, we rewrite this to get the following:

\[
\forall \phi_0, \phi_m, \phi_0' \in L \bullet (\phi_0 \supseteq \psi_0 \land (\phi_m \cup \phi_0' \supseteq \psi_m \cup \psi_0')) \rightarrow (\phi_0 \oplus (\phi_m \cup \phi_0') \rightarrow \phi).
\]

Before we continue, we first prove the following lemma, which states that if every refinement, \( \phi_0' \), of a constraint, \( \phi_0 \), satisfies a certain property, then \( \phi_0 \) also satisfies that property, and vice-versa.

**Lemma 3:** \( \forall \phi_0' \in \uparrow \phi_0 \bullet \phi_0' \rightarrow \psi \iff \phi_0 \rightarrow \psi \)

Proof. The universal quantification is expanded to the following:

\[
\phi_0' \rightarrow \psi \text{ and } \phi_0'' \rightarrow \psi \text{ and } \phi_0''' \rightarrow \psi \text{ and } \ldots
\]

for each constraint in the upward closure of \( \phi \). From the definitions in De Boer et al. (de Boer et al., 1997), we know the following:

\[
\phi_0 \lor \phi_0'' \lor \phi_0''' \lor \ldots \rightarrow \psi
\]

From the definition of upward closure, \( \phi_0' \lor \phi_0'' \lor \phi_0''' \lor \ldots \) is equivalent to \( \phi_0 \), and therefore \( \phi_0 \rightarrow \psi \).

Note that \( \phi_0, \phi_m, \) and \( \phi_0' \) entail \( \psi_0, \psi_m, \) and \( \psi_0' \) respectively, and therefore are in their upward closures. Using this and Lemma 3, we can remove the quantification of the predicate above:

\[
\psi_0 \oplus (\psi_m \cup \psi_0') \rightarrow \phi.
\]

However, we cannot rewrite a statement made about an atomic predicate to the above, because \( \psi_0 \oplus (\psi_m \cup \psi_0') \rightarrow \phi \) contains free variables that may be part of other propositions. For example, if we are proving the proposition \( \phi \land [\psi_0 \xrightarrow{c,x \mapsto m} \psi_0'] \), the variables in \( \psi_0' \cup \psi_m \) may occur in \( \phi \). Therefore, we rename all variables inside the brackets, denoted \( x \), with fresh variables, \( x' \):

\[
(\psi_0 \oplus (\psi_m \cup \psi_0') \rightarrow \phi)[x'/x] \text{ where } x' = vars(\psi_m \cup \psi_0').
\]

Combining the two cases of \( \psi_0 \) holding and not holding respectively, we have the following equivalence:

\[
[\psi_0 \xrightarrow{c,x \mapsto m} \psi_0] \phi \iff \psi_0 \lor (\psi_0 \land (\psi_0 \oplus (\psi_m \cup \psi_0') \rightarrow \phi)[x'/x])
\]

Which, using the definition of \( \lor \), is equivalent to the following:

\[
[\psi_0 \xrightarrow{c,x \mapsto m} \psi_0] \phi \iff \psi_0 \rightarrow (\psi_0 \oplus (\psi_m \cup \psi_0') \rightarrow \phi)[x'/x]
\]

Finally, we are left with something that resembles our axiom, with the exception that we have \( \psi_0 \oplus (\psi_m \cup \psi_0') \) on the right-hand side, rather than just \( \psi_m \cup \psi_0' \). Recall that \( \psi_0 \oplus (\psi_m \cup \psi_0') \) is a constraint representing all of the information from \( \psi_m \cup \psi_0' \), plus the information about the variables in \( \psi_0 \) that are not in \( \psi_m \cup \psi_0' \). In the implication \( \psi_0 \rightarrow (\psi_0 \oplus (\psi_m \cup \psi_0') \rightarrow \phi) \), any constraint about a variable \( x \) in \( \phi \) must either come from \( \psi_m \cup \psi_0' \), or from the precondition \( \psi_0 \). In the case that it comes from the precondition, then we know the variable is not referenced in \( \psi_m \cup \psi_0' \) (otherwise it would be overridden), and we know that the information specified in \( \psi_0 \) for \( x \) holds already from the premise \( \psi_0 \rightarrow (\ldots) \). Therefore, the \( \psi_0 \) as the left argument to the \( \oplus \) operator is unnecessary.

However, recall that we renamed all variables inside the brackets on the right-hand side of the implication so that all were fresh, therefore, this would not be the case for the renamed variables. Thus, instead of renaming each of them, we rename only those in \( \psi_m \cup \psi_0' \), as is specified by the side condition of Axiom 4.1.1(iii), leaving us with the following:

\[
[\psi_0 \xrightarrow{c,x \mapsto m} \psi_0] \phi \iff \psi_0 \rightarrow (\psi_m \cup \psi_0' \rightarrow \phi)[x'/x].
\]
APPENDIX B. Proof of Theorem 4

In this appendix, we provide a proof for Theorem 4. This theorem states the following formulae are valid theorems of $\mathcal{L}_\alpha$:

In this, any information about a variable $x$ that holds for $\phi$ will either come from $\psi_m \cup \psi_0$ and therefore be renamed, or it will come from $\psi_0$ and not be renamed, but will hold from the precondition $\psi_0$.

Combining the cases of when $\psi_0$ both holds and does not hold means that this axiom is holds for any model, and is therefore valid.

\[\text{Axiom (vi)}\]

\[\models [\alpha; \beta] \phi \leftrightarrow [\alpha] [\beta] \phi\]

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

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

\[\text{Axiom (vii)}\]

\[\models [\alpha \cup \beta] \phi \leftrightarrow [\alpha] \phi \land [\beta] \phi\]

From the compositional semantics defined in Section 2.3, the set of traces resulting from a protocol choice is the union of the traces of both protocols. Therefore, the left hand side of this axiom is represented as $\forall \psi_0 \in [\alpha] \cup [\beta] \ldots$, and the right hand side as $\forall \psi_0 \in [\alpha] \ldots$ and $\forall \psi_0 \in [\beta] \ldots$.

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 traces of the two proposition about are equivalent, and therefore any proposition that holds for all end states of one will hold for the other.

\[\text{Axiom (viii)}\]

\[\models [\text{var}_x^{\psi_0}] \phi \rightarrow x = x \land \psi_0 \rightarrow [\alpha](x_0 = x \rightarrow \phi)\]

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 (that is, not in the state). If this is not the case, $x$ must be renamed to a fresh variable.

This axiom is explained as follows. In the protocol $\text{var}_x^{\psi_0} \cdot \alpha$, the constraints on $x$ remain unchanged during the execution of $\alpha$, and otherwise $\text{var}_x^{\psi_0} \cdot \alpha$ executes in the same manner as $\alpha$.

Therefore, we say that $\phi$ holds for every end state in $\text{var}_x^{\psi_0} \cdot \alpha$ if and only if it holds in every end state of $\alpha$ in which the constraints 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 $\alpha$. We know that $\phi$ does not change the value of $x_0$ (it is a fresh variable), so after executing $\alpha$, 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, $\psi_0'$, of $\text{var}_x^{\psi_0} \cdot \alpha$. We know that $\psi_0' \models \phi$. We also know that the same trace and end state must exist for $\alpha$, because $\alpha$ is the same as $\text{var}_x^{\psi_0} \cdot \alpha$, except it does not maintain the constraints on $x$ throughout, therefore, the traces and end states in $\text{var}_x^{\psi_0} \cdot \alpha$ must be a subset of those in $\alpha$. Clearly, $\psi_0 \models x_0 = x$, because $x_0$ is fresh and therefore not changed by $\alpha$, therefore $\psi_0' \models x_0 = x \rightarrow \phi$.

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

\[\text{Axiom (viii)}\]

\[\models (D, [N(x)] \phi) \leftrightarrow [\alpha][x/y] \phi \quad \text{where } N(x) \equiv \alpha \in D\]

This axiom is trivially true from the definition of $[N(x)]$ and from $N(y) \equiv \alpha \in D$.

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

\[\square\]
(i) \([\alpha](\phi \rightarrow \psi) \rightarrow [\alpha]\phi \rightarrow [\alpha]\psi\)

(ii) \(\phi \land [\alpha][\alpha^*]\phi \rightarrow [\alpha^*]\phi\)

(iii) \(\phi \land [\alpha^*](\phi \rightarrow [\alpha]\phi) \rightarrow [\alpha^*]\phi\)

Proof. This proof uses the proof system defined in Section 4.

Theorem (i) \(\models [\alpha](\phi \rightarrow \psi) \rightarrow [\alpha]\phi \rightarrow [\alpha]\psi\)

This theorem is equivalent the the \(K\) axiom defined in normal modal logics, and is an important theorem in our logic. The proof of \(K\) is straightforward. We start by applying the propositional logic axiom \(p \rightarrow (q \rightarrow r) \leftrightarrow (p \land q) \rightarrow r\), and then prove:

\([\alpha](\phi \rightarrow \psi) \land [\alpha]\phi \rightarrow [\alpha]\psi\)

\(\iff \phi \land [\alpha](\phi \rightarrow \psi) \land [\alpha]\phi \rightarrow [\alpha]\psi\) using Axiom 4.1.1(i)

From modus ponens, \([\alpha]\psi\) trivially holds.

We note that the inverse of this theorem does not hold. This is easy to show using a counterexample. Consider a protocol \(\alpha\) with only two end states: \(\phi \land \neg \psi\) and \(\neg \phi \land \neg \psi\). The proposition \([\alpha]\phi \rightarrow [\alpha]\psi\) is true because the premise is false (\(\phi\) does not hold in all end states), but the proposition \([\alpha](\phi \rightarrow \psi)\) is clearly false because there is one end state in which \(\phi \land \neg \psi\) is true, implying that \(\phi \rightarrow \psi\) is false.

Theorem (ii) \(\models \phi \land [\alpha][\alpha^*]\phi \leftrightarrow [\alpha^*]\phi\)

The prove this, we rewrite the left side of the equivalence to show that it is equivalent to the proposition on the right side.

\(\phi \land [\alpha][\alpha^*]\phi\)

\(\iff [\epsilon]\phi \land [\alpha; \alpha^*]\phi\) using Axioms 4.1.1(ii) and 4.1.1(iv)

\(\iff [\epsilon \cup \alpha; \alpha^*]\phi\) using Axiom 4.1.1(v)

\(\iff [\alpha^*]\phi\) using definition of \(\alpha^*\)

Theorem (iii) \(\models \phi \land [\alpha^*](\phi \rightarrow [\alpha]\phi) \rightarrow [\alpha^*]\phi\)

To prove this, we prove a stronger property — that the left and right sides of the above implication are equivalent. That is, we prove the following:

\(\models \phi \land [\alpha^*](\phi \rightarrow [\alpha]\phi) \leftrightarrow [\alpha^*]\phi\)

The expression \(\alpha^*\) is defined as \(N \equiv \epsilon \cup \alpha; N\). To prove our theorem, we must prove the following:

\(\vdash \phi \land [N](\phi \rightarrow [\alpha]\phi) \rightarrow [N]\phi\)

Using the induction inference rule, we perform one unfolding of \(N\) and prove this assuming the above. This is proved by re-writing the propositional labelled \(Q\), and showing that it is equivalent to \(P\) under the assumption \(A\):

\[
\frac{\phi \land [N](\phi \rightarrow [\alpha]\phi) \rightarrow [N]\phi \quad \phi \land [\epsilon \cup \alpha; N](\phi \rightarrow [\alpha]\phi) \rightarrow [\epsilon \cup \alpha; N]\phi}{Q}
\]

To rewrite \(Q\), we use the axioms and the induction rule:

\(\phi \land [\alpha][N]\phi\)

\(\iff \phi \land [\alpha][N]\phi\) using Axioms 4.1.1(v), 4.1.1(iv), and 4.1.1(ii)

\(\iff \phi \land [\alpha][\phi \land [N](\phi \rightarrow [\alpha]\phi)]\) using the assumption (the inductive step)

\(\iff [\alpha]\phi \land [\alpha][N](\phi \rightarrow [\alpha]\phi)\) using Axioms 4.1.1(v) and 4.1.1(iv)

\(\iff [\alpha][\phi \land [\alpha]\phi \land [\alpha][N](\phi \rightarrow [\alpha]\phi)\) using modus ponens and Axiom 4.1.1(ii)

\(\iff [\epsilon \cup \alpha; N](\phi \rightarrow [\alpha]\phi)\) using Axiom 4.1.1(v)

This is equivalent to \(P\), the left hand side of the equivalence. From this result, we conclude that Theorem 4 holds.