Using Constraints and Process Algebra for Specification of First-Class Agent Interaction Protocols

Tim Miller and Peter McBurney

Department of Computer Science,
The University of Liverpool, Liverpool, L69 7ZF, UK
{tim, p.j.mcburney}@csc.liv.ac.uk

Abstract. Current approaches to multi-agent interaction involve specifying protocols as sets of possible interactions, and hard-coding decision mechanisms into agent programs in order to decide which path an interaction will take. This leads to several problems, three of which are particularly notable: hard-coding the decisions about interaction within an agent strongly couples the agent and the protocols it uses, which means a change to a protocol involves a changes in any agent that uses such a protocol; agents can use only the protocols that are coded into them at design time; and protocols cannot be composed at runtime to bring about more complex interactions. To achieve the full potential of multi-agent systems, we believe that it is important that multi-agent interaction protocols exist at runtime in systems as entities that can be inspected, referenced, composed, and shared, rather than as abstractions that emerge from the behaviour of the participants. We propose a framework, called $\mathcal{RASA}$, which regards protocols as first-class entities. In this paper, we present the first step in this framework: a formal language for specification of agent interaction protocols as first-class entities, which, in addition to specifying the order of messages using a process algebra, also allows designers to specify the rules and consequences of protocols using constraints. In addition to allowing agents to reason about protocols at runtime in order to improve their the outcomes to better match their goals, the language allows agents to compose more complex protocols and share these at runtime.

1 Introduction

Research into multi-agent systems aims to promote autonomy and often intelligence into agents. Intelligent agents should be able to interact socially with other agents, and adapt their behaviour to changing conditions. Despite this, research into interaction in multi-agent systems is focused mainly on the documentation of interaction protocols, which specify the set of possible interactions for a protocol in which agents engage. Agent developers use these specifications to hard-code the interactions of agents. We identify three significant disadvantages with this approach: 1) it strongly couples agents with the protocols they use —

something which is unanimously discouraged in software engineering — therefore requiring agent code to changed with every change in a protocol; 2) agents can only interact using protocols that are known at design time, a restriction that seems out of place with the goals of agents being intelligent and adaptive; and 3) agents cannot compose protocols at runtime to bring about more complex interactions, therefore restricting them to protocols that have been specified by human designers.

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

In this paper, we define a syntax and semantics for the $\mathcal{RASA}$ protocol specification language, which forms part of the $\mathcal{RASA}$ framework, a framework for modelling agent interaction as first-class entities. They key ideas that were taken into account in the design of the $\mathcal{RASA}$ language were the following:

- *protocols as first-class entities*: rather than protocol specifications emerging from an abstraction of the behaviour of participating agents, $\mathcal{RASA}$ protocols are *first-class*, meaning that they exist as entities in multi-agent systems;
- *inspectable*: agents are able to inspect and reason about the set of interactions permitted by a protocol, when they can occur, and what their effects are, so that the agents can devise strategies at runtime, therefore de-coupling them from the protocols they use;
- *layered*: other languages used for inspectable interaction protocols, such as OWL-P [7] and Yolum and Singh's Event Calculus extension [27], use the same language for specifying rules and effects as they do for specifying the sequencing of messages. We take a layered approach, in which the language for specifying the sequence of messages is separated from the language for specifying rules and effects. This allows us to develop the $\mathcal{RASA}$ framework independent of the underlying constraint language, and does not enforce the use of a particular language; and

– *reusable, composable and extendable*: existing protocol specifications can be extended and composed with other protocols to bring about new protocols. In addition, composable protocols permits designers to break up their task into smaller subtasks, simplifying the design process.

1.1 $\mathcal{RASA}$ Outline

The idea of first-class protocols is novel, but not entirely new. Desai *et al.* [7], and Yolum and Singh [27] present some initial work in using OWL and the Event Calculus respectively to model protocols as *first-class* entities (although they do not use this term). These approaches adapt existing declarative languages by adding definitions, written in that language, that specify the rules and effects of protocols. However, there are two major downsides to taking this approach. Firstly, the effects and rules must be specified in the declarative language (OWL or the Event Calculus), which is too restrictive. Secondly, the message sequencing is also specified in the language itself. For example, to specify that message a is sent before b, one must write a predicate resembling the following:

$$Happens(a, t_1) \wedge Happens(b, t_2) \wedge t_1 < t_2$$

This means that event a happens at time t_1, event b happens at time t_2, and event a occurs before event b, specified by the predicate $t_1 < t_2$. This can be specified in a process algebra as $a; b$, which we believe is more intuitive to the human reader, and is no less expressive. Robertson [22] takes a similar approach of using a process algebra and an underlying language to specify first-class protocols. We extend his work by, among other things, providing an additional language construct — local variable declaration — and by formalising the relationship between the process algebra and the underlying language.

Using $\mathcal{RASA}$, protocols can be visually represented as annotated trees outlining the interactions that can occur. As well as being annotated with a transitional message, each arc in the tree is annotated with a precondition and postcondition, in which the precondition must be enabled for the message to be sent, and the postcondition represents the effect of sending a message. The nodes represent the states that result from the corresponding postcondition. We incorporate states into the language to allow designers and agents to calculate the effect of a series of transactions; that is, if there are two messages sent in sequence, the effect of the second depends on the state resulting from the first. The root node of the tree is the initial state of the protocol, and the leaf nodes represent terminating states. Branches in the tree represent choices to be made by one or more agents.

Protocol rules, effects, and states are specified using declarative constraint languages. Using such languages allow agents to reason about which messages to send by calculating the which paths best achieve their goals, with each path from the root node to a leaf node representing a possible sequence of interaction. Agents can also reason about sub-protocols, by taking the root node of a sub-protocol as the starting point. We do not insist on a particular constraint language, but instead assume that it contains a few basic operators and constants

common to most constraint languages. Such an approach fulfils our requirement that the language is inspectable, because agents can be equipped with the necessary constraint solvers, while maintaining flexibility by not enforcing a particular language. The constraint language is separate from the language for specifying the possible sequences of interaction in the protocol.

Protocols can be referenced and composed with others to form new protocols. The composition of these interactions provides a precondition and postcondition for an entire protocol. $\mathcal{RASA}$ allows one type of atomic event, comprising a precondition, message, and postcondition. These correspond to an arc in the tree. An atomic event is itself a protocol, meaning that the syntax and semantics of composing existing protocols is the same as composing a single protocol.

We envisage systems in which agents have access to bases of protocol specifications; either locally or centrally. Agents can search through these bases at runtime to find protocols that best suit the goal they are trying to achieve, and can share these protocol specifications with possible future participants. If no single protocol is suitable for the agent, composition of these may offer an alternative.

This paper is structured as follows: Section 2 presents the assumptions we make regarding the constraint language used by the agents engaged in interaction. As will be seen, these assumptions are quite general, allowing wide applicability of the framework. Section 3 then presents the formal syntax of the $\mathcal{RASA}$ modelling language, with an operational semantics for this language presented in Section 4. The paper follows this with a section discussing reuse and composition of protocols. Section 6 then presents a discussion of related work, before Section 7 concludes the paper.

2 Modelling Information

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

In this section, we discuss the minimum requirements for a constraint language that can be used in $\mathcal{RASA}$. We do not believe that these requirements are unreasonable — many languages can be used within the framework. For example, there are many description logics [2], constraint programming languages [23], commitment logics [27], or even predicate and modal logics [3] that contain the necessary constructs, although some of these languages may not be executable, and therefore the protocols would not be inspectable. The content languages proposed by FIPA [9] would also be suitable candidates.

Definition 1. *Constraint*

A *constraint* is a piece of information reducing the set of values that are possible for variables in a universe. For example, if an agent wishes to specify that the price of an item, $Item$, is 10 units, it may express this as follows:

$$PriceOf(Item, 10)$$

In this constraint, $Item$ is a variable, $PriceOf$ a relation, and '10' a constant. Operators used to express constraints over the universe must be defined.

Definition 2. *Constraint System*

We assume agents communicate using a communication language, which has a set of operators used to express constraints over variables. We will refer to such as language as a *constraint language*. Rather than define the syntax and semantics of a new language, or present the details of an existing one, we take the approach that any language can be used as the communication language in our framework, provided it contains a few basic constants and operators with certain properties. As well as using this as a communication language, we assume that this language is used to specify the preconditions and consequences of protocols. We refer to this as the *underlying constraint language* or just *underlying language*. This constraint language is denoted $\mathcal{L}$.

We use the definition of a *constraint system* proposed by De Boer *et al.* [5]. They define a constraint system as a complete algebraic lattice

$$\langle C, \sqsupseteq, \sqcup, \text{true}, \text{false} \rangle$$

In this structure, C is the set of atomic propositions in the language, for example $1 \leq 2$, $\sqsupseteq$ is an entailment operator, true and false are the least and greatest elements of C respectively, and $\sqcup$ is the least upper bound operator. The shorthand $c = d$ is equivalent to $c \sqsupseteq d$ and $d \sqsupseteq c$.

The entailment operator is a partial order over C, in that, for any atomic propositions, c and d, $c \sqsupseteq d$ means that c contains more information than d. This is read that d is provable from c, which means that any values that satisfy the variables in d also satisfy c. For example, $x \leq 5 \sqsupseteq x \leq 6$ specifies that if x is less than or equal to 5, then it is less than or equal to 6, which is trivially true because there exists no value for x that satisfies $x \leq 5$ that does not also satisfy $x \leq 6$. The $\sqcup$ operator specifies the addition of information. For example, to specify the prices of $ItemA$ and $ItemB$ are 5 and 10 units respectively, one could write $PriceOf(ItemA, 5) \sqcup PriceOf(ItemB, 10)$. This is is analogous to conjunction in logic, in that $c \sqcup d$ is the joining of information. Therefore, $c \sqcup d \sqsupseteq d$ is true for any c and d.

A *cylindric constraint system* is a constraint system with an operator for hiding variables. De Boer *et al.* [5] define a cylindric constraint system as a structure, $\langle C, \sqsupseteq, \sqcup, \text{true}, \text{false}, Var, \exists \rangle$, in which Var is a set of variables, and $\exists$ the hiding operator. To hide a variable x in a constraint c, one would write $\exists_x c$. The hiding operator has the following properties:

- $c \sqsupseteq \exists_x c$
- $c \sqsupseteq d$ implies $\exists_x c \sqsupseteq \exists_x d$
- $\exists_x (c \sqcup \exists_x d) = \exists_x c \sqcup \exists_x d$
- $\exists_x \exists_y c = \exists_y \exists_x c$

For example, to specify that the price of $Item$ is between 5 and 10 units inclusive, one could write $\exists_{Price}(PriceOf(Item, Price) \sqcup 5 \leq Price \sqcup Price \leq 10)$.

We use shorthand to represent negation in $\mathcal{L}$. For constraints c and d, $c \sqsupseteq \neg d$ is true if and only if $c \sqsupseteq d$ is not. That is, c entails $\neg d$ if and only if c does not entail d. Note that, from this definition, we assume only that negation can occur on the right hand side of the entailment operator. Depending on the underlying constraint language used, this restriction could be relaxed, but this is not necessary to fit into the framework.

Throughout this paper, constraints will adhere to the following grammar, although a suitable language need not adhere to this grammar to be used in the framework:

$$\phi ::= c \mid \phi \sqcup \phi \mid \neg\phi \mid \exists_x \phi$$

In this grammar, c is any atomic constraint in C, and x any variable in Var. We use ψ and ϕ as meta-variables representing constraints that follow this grammar, adding subscripts and superscripts to denote distinct meta-variables. Brackets are used to remove syntactic ambiguity, although to reduce the need for brackets, we specify that $\neg$ and $\exists$ both bind tighter than $\sqcup$, so $\neg\phi \sqcup \psi$ is $(\neg\phi) \sqcup \psi$, and $\exists_x \phi \sqcup \psi$ is $(\exists_x \phi) \sqcup \psi$.

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

Definition 3. *Free Variables*

The function, $free \in \mathcal{L} \to \wp(Var)$, returns the set of free variables in any constraint; that is, variables referenced in ϕ that are not hidden using $\exists$. For example, $free(x \leq 5) = \{x\}$, and $free(\exists_x (x \leq 5 \sqcup y = x)) = \{y\}$. Calculating the free variables in a constraint can be done at a syntactic level using an inductive definition over the constraints:

$$\begin{aligned}
free(c) &= \ldots \\
free(\text{true}) &= \emptyset \\
free(\text{false}) &= \emptyset \\
free(\phi \sqcup \psi) &= free(\phi) \cup free(\psi) \\
free(\neg\phi) &= free(\phi) \\
free(\exists_x \phi) &= free(\phi) \setminus \{x\}
\end{aligned}$$

We do not define $free(c)$ because that is specific to $\mathcal{L}$. For readability, we use the shorthand $\exists_{\hat{x}} \phi$ to represent $\exists_{free(\phi)\setminus\{x\}} \phi$. That is, $\exists_{\hat{x}} \phi$ means that we quantify over all free variables in ϕ except x.

3 $\mathcal{RASA}$ Protocols

In this section, we present the language for modelling $\mathcal{RASA}$ protocols, and some definitions relevant to this.

The $\mathcal{RASA}$ protocol specification language resembles that of other process algebras, such as CSP [11]. However, we specify the rules and effects of protocols using an underlying constraint language, which would be declarative by definition. This allows agents equipped with the necessary tools to reason about this language to determine when rules are satisfied, to calculate the effect of sending a particular message, and to devise strategies for interaction at runtime.

Definition 4. *Communication Channel*

We assume that a *communication channel* is a one-to-one connection between two agents. The notation $c(i, j)$ denotes the communication channel between the sending agent with identity i, and the receiving agent with identity j, in which identities are represented in the underlying language.

We employ the notation $c(i, j).\phi_m$ to represent the message ϕ_m being sent by agent i to agent j via the channel $c(i, j)$. The event of agent i sending a message to j is the same event as agent j receiving this message. That is, the event is the communication over the channel. Agent identities are omitted when the sending and receiving agents are not relevant; that is, we write $c.\phi_m$.

An alternative way to represent communication between agents is to have many-to-many channels, with the sender and receiver identities as part of the message. However, we choose the first approach so that we can reason about communication in our framework language, rather than mixing this with the underlying constraint language.

Definition 5. $\mathcal{RASA}$ *Protocol*

A $\mathcal{RASA}$ *protocol* is an annotated tree of interactions between entities. An *annotation* is a triplet of constraints, in which the first constraint represents the precondition that must hold for a transition to occur, the second constraint represents the message to be sent, and the third constraint is a postcondition, which must hold after an enabled transition occurs. Branches in the tree represent choices to be made by one or more agents.

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

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

We use π as a meta-variable to refer to protocols; subscripts and superscripts are used to denote distinct meta-variables. ϵ represents the empty protocol, in which no message is sent and there is no change to the protocol state. A protocol of the format $\phi \xrightarrow{c.\phi_m} \phi'$ is an atomic protocol. It represents the value ϕ_m being sent over channel c if ϕ holds in the current state. After the value is sent, the

new state of the protocol is updated using ϕ'. We use this to specify rules and effects of protocols: the precondition represents a rule for a protocol because ϕ_m can only be sent if this precondition is true; and the postcondition represents the effect that sending ϕ_m has. For atomic protocols, meta-variables with a prime ($'$) are used to refer to the postconditions; that is, ϕ is the precondition and ϕ' the postcondition. We use ϕ_m (that is, constraints subscripted with m) to denote message constraints.

The protocol $\pi_1; \pi_2$ denotes the sequential composition of two protocols, such that all of protocol π_1 is executed, then protocol π_2. The protocol $\pi_1 \cup \pi_2$ denotes a choice of two protocols. $N(x)$ denotes a reference to a protocol $N(y)$, with variables y renamed to x, such that the referenced protocol is expanded into this protocol. For brevity, we use x and y to represent sequences of variables as well as single variables. Protocols can reference themselves, and can mutually reference each other, which introduces the possibility of non-terminating protocols. The protocol $\mathbf{var}_x^{\phi} \cdot \pi$ denotes the declaration of a local variable x, with the constraints ϕ on x. The scope of x is limited to the protocol π.

We permit brackets to group together protocols, and to reduce the use of brackets, operators have a strict ordering. The infix operators always bind tighter than variable declaration, with sequential composition binding tighter than choice. Therefore, the protocol $\mathbf{var}_x^{\psi} \cdot \pi_1; \pi_2 \cup \pi_3$ would be equivalent to $\mathbf{var}_x^{\psi} \cdot ((\pi_1; \pi_2) \cup \pi_3)$.

Definition 6. *Protocol Specification*

Let N be a name, y be a sequence of variable names, and π be a protocol. A *protocol specification* is defined as a set of definitions of the form

$$N(y) \mathrel{\widehat{=}} \pi$$

Definition 7. *Protocol Instance*

Let D be a protocol specification, π a protocol, and ϕ a constraint. A *protocol instance* is a tuple, $\langle D, \pi, \phi \rangle$, in which π can reference the protocol names defined in D, and ϕ is a constraint representing the state of the protocol at that instance. Instances evolve via message sending and the changing of the state.

Example 1. As an example of specifying a protocol, we use a simple negotiation protocol. In this example, a buyer, B, is bidding for an item. If the price that the buyer suggests, $Price$, is greater than the current price in the protocol state, the seller, S, accepts the bid, otherwise rejecting it. In the case that the bid is accepted, $Price$ becomes the new current price. Such a protocol could be a sub-protocol of an English auction protocol, and would be iterated over until no more bids are received, or until some timeout is reached. This protocol is specified as follows:

$$Buy(Item, Price, B, S) \mathrel{\widehat{=}} Bid; AcceptOrReject$$

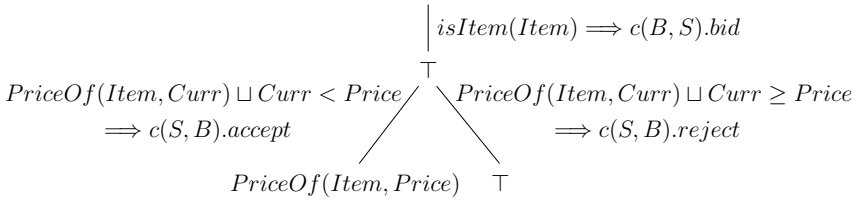
$$Bid(Item, PriceB, S) \mathrel{\widehat{=}} isItem(Item) \xrightarrow{c(B,S).bid(Item,Price)} \text{true}$$

$$AcceptOrReject(Item, Price, B, S) \mathrel{\widehat{=}} \mathbf{var}_{Curr}^{PriceOf(Item,Curr)} \cdot (Accept \cup Reject)$$

$$Accept(Item, Price, Curr, B, S) \mathrel{\widehat{=}}$$
$$Curr < Price \xrightarrow{c(S,B).accept(PriceOf(Item,Price))} PriceOf(Item, Price)$$

$$Reject(Item, Price, Curr, B, S) \mathrel{\widehat{=}}$$
$$Curr \geq Price \xrightarrow{c(S,B).reject(PriceOf(Item,Price))} \text{true}$$

So, the bidder sends a bid to the seller. The declaration of the local variable $Curr$ represents the current bid; that is, $Curr$ is equivalent to the value that satisfies $PriceOf(Item, Curr)$ in the state. If the bid is greater than $Curr$, the seller sends an acceptance, and the constraint $PriceOf(Item, Curr)$ is added to the constraint store, overriding any previous constraints on $Item$. If the bid is less than the current price, the bid is rejected and the state remains unchanged. This specification corresponds to the following tree, in which the nodes refer to the consequence of the previous action, and the arcs are of the format $\psi \Longrightarrow c.\phi_m$, interpreted as "if ψ holds, then the transition $c.\phi_m$ can occur." For presentation, we have left out some details that are in the specification.



An agent with ID represented by the variable a, wishing to sell an item, i, to another agent b, may propose that this protocol is used to determine a price by proposing the following protocol instance:

$$\langle P, Bid(i, Price, b, a), PriceOf(i, 0)\rangle$$

In which P is the protocol specification above, $Bid(i, Price, b, a)$ is the Bid protocol with renamed variables, and $PriceOf(i, 0)$ is the initial state. The task is now that the agents must find an instantiation for the variable $Price$ on which they both agree.

Clearly, agreeing on a protocol instance from which to begin the negotiation is itself a negotiation problem, which would likely be solvable with another protocol. Such *meta-protocols* are, in the context of a system, at a higher level than other protocols, such as the negotiation protocol above. However, this does not rule out the option of using the same protocol at both levels.

Meta-protocols, their use, and their control are dependent on the system in which they are employed, and on the agents within these systems, so proposing a

general solution for this problem is not possible. For example, some agents may adopt a "take it or leave it" approach, in which other participants either use a certain protocol to interact with them, or do not interact with them at all. In other cases, adopting a specific meta-protocol may be a condition of entry into the system. Other systems may leave it up to the agents themselves to decide. In future work, we plan to specify a collection of meta-protocols that can be used to negotiate which $\mathcal{RASA}$ protocol to use, and look at the contexts in which these protocols can be employed.

4 Semantics

In this section, we define and discuss the semantics of the $\mathcal{RASA}$ protocol specification language.

4.1 Structural Operational Semantics

We view the semantics of protocols as commands on a virtual machine, in which states incorporate protocol instances, and the commands are the messages being sent over communication channels. To model these semantics, we make use of *structural operational semantics*, as defined by Plotkin [20].

Using structural operational semantics, a system is defined as a set of transitions, with each transition linking two states. In the case of our protocol semantics, a state is defined as a protocol instance. Recall from Definition 7 that a protocol instance is a tuple $\langle D, \pi, \phi \rangle$, in which D is a protocol specification, π is the protocol that is to be executed, and ϕ a constraint representing the state of the protocol. Thus, a transition takes the form:

$$\langle D, \pi, \phi \rangle \xrightarrow{l} \langle D, \pi', \phi' \rangle$$

This denotes the protocol π being executed in the state ϕ, the transition l occurring at this point. π' denotes the part of the protocol left to execute, and ϕ' denotes the new state of the protocol. l refers to either the communication of a constraint ϕ_m over a channel, written $c.\phi_m$, or the empty transition. We use $\xrightarrow{\varepsilon}$ to represent the empty transition. D is invariant over the course of execution, therefore, we omit it whenever it is not referenced in a transition.

As a shorthand, we use the following to indicate that $\langle D, \pi, \phi \rangle$ evolves to $\langle D, \pi', \phi' \rangle$ over the sequence of transitions $l_1, \ldots, l_n$:

$$\langle D, \pi, \phi \rangle \xrightarrow{l_1, \ldots, l_n} \langle D, \pi', \phi' \rangle$$

We specify the semantics as inference rules of the following form:

$$\frac{antecedents}{conclusion} \quad conditions$$

In which *antecedents* are the assertions about what can occur at the current state, *conditions* are the side conditions under which this rule is enabled, and *conclusion* is the transition that can occur. We use the special protocol, E, to represent the protocol whose execution is complete.

4.2 Renaming

To define the semantics of protocols specified using $\mathcal{RASA}$, we include syntax for renaming over protocols. Specifically, renaming is defined inductively over the structure of the protocols such that the protocol $\pi[x/y]$ represents the protocol π, with every free occurrence of the variable y substituted with the variable x, including variables in the constraints. Formally, renaming is defined as follows, in which I denotes either of the binary infix operators:

$$
\begin{aligned}
(\psi \xrightarrow{c.\phi_m} \psi')[x/y] &= \psi[x/y] \xrightarrow{c.(\phi_m[x/y])} \psi'[x/y] \\
(\pi_1 \ I \ \pi_2)[x/y] &= \pi_1[x/y] \ I \ \pi_2[x/y] \\
N(z_1, \ldots, y, \ldots, z_n)[x/y] &= N(z_1, \ldots, x, \ldots, z_n) \\
(\mathbf{var}_y^{\psi} \cdot \pi)[x/y] &= \mathbf{var}_x^{\psi[x/y]} \cdot \pi[x/y] \\
(\mathbf{var}_z^{\psi} \cdot \pi)[x/y] &= \mathbf{var}_z^{\psi[x/y]} \cdot \pi[x/y]
\end{aligned}
$$

Informally, this says that renaming y to x in an atomic protocol is equivalent to performing the same rename over the constraints in the protocol. Renaming a infix composition is equivalent to renaming the two sub-protocols of that composition. Renaming y to x in a protocol reference consists of renaming any instances of y in the variable list to x — the name of the protocol is not renamed. Variable declaration has two rules: the first if the declared variable is y, in which case the variable is changed to x; the second if the declared variable is not y, in which case the variable remains the same. In both cases, the constraint on x and the protocol in the scope of the variable are both renamed.

As an example of protocol renaming, we use the negotiation example from Section 3. Suppose that we wish to use this within an auction protocol, with an auctioneer represented by the variable *Auctioneer*, then one could use the renamed protocol $Buy[Auctioneer/S]$ to represent the same protocol, but in which every occurrence of the variable S replaced with *Auctioneer*. Therefore, the message representing a bid on the item would be $c(B, Auctioneer).bid(Item, Price)$.

4.3 Operational Semantics of Protocol Operators

Now we have some basic definitions covered, we define the semantics of the protocol operators in the $\mathcal{RASA}$ language. That is, of executing a protocol within the context of a protocol specification and a constraint representing the protocol state.

Definition 8. *Semantics of the Empty Protocol*

The empty protocol terminates under no transition, and has no effect on the protocol state.

$$\frac{}{\langle \epsilon, \phi \rangle \xrightarrow{\varepsilon} \langle E, \phi \rangle}$$

Definition 9. *Semantics of Atomic Protocols*

Firstly, we define the semantics for the atomic protocol. That is, the protocol consisting only of a message being sent over a channel if the precondition is satisfied, resulting in a new protocol state.

$$\frac{}{\langle \psi \xrightarrow{c.\phi_m} \psi', \phi \rangle \xrightarrow{c.\phi'_m} \langle E, \phi' \rangle} \qquad \text{if } \phi \sqsupseteq \psi \text{ and } \phi'_m \sqcup \phi' \sqsupseteq \phi_m \sqcup \mathcal{O}(\phi, \psi')$$

This states that, if the precondition ψ is true under the model ϕ, then the transition can occur. This transition can be the constraint, ϕ_m, but can also be a constraint, ϕ'_m, that contains more information that ϕ_m, such that $\phi'_m \sqsupseteq \phi_m$. This allows the message sender to place additional constraints on the message, and, as a consequence, the resulting state. The resulting state is $\mathcal{O}(\phi, \psi')$, in which $\mathcal{O} \in (\mathcal{L} \times \mathcal{L}) \to \mathcal{L}$ is an overriding function defined as $\mathcal{O}(\phi, \psi') = \psi' \sqcup \exists_{free(\psi')}\phi$. Therefore, $\mathcal{O}$ defines a new constraint such that the values of any free variables in ψ' are overridden with the values constrained by ψ', while the free variables in ϕ that are not otherwise in ψ' maintain their pre-state values. However, any additional information in the message, ϕ'_m must also apply to the resulting state. For example, considering the following atomic protocol from the auction example in Section 3:

$$Curr < Price \xrightarrow{c.accept(PriceOf(Item,Price))} PriceOf(Item, Price)$$

The sender confirms that the bid for $Item$ at the price $Price$ has been accepted, in which $Item$ and $Price$ are variables. As part of the interaction, the sender would like to instantiate both variables — $Item$ with an item, and $Price$ with a number. If the sender wants to confirm that the price of the $Item$ is 10, then the message will be $c.accept(PriceOf(Item, Price) \sqcup Price = 10)$. The constraint $Price \sqcup 10$ needs to be shared with the postcondition. The semantics enforces this: $\phi'_m \sqcup \phi' \sqsupseteq \phi_m \sqcup \mathcal{O}(\phi, \psi')$. In this example, the only solution for $Price$ in this constraint would be $Price = 10$, therefore, the post-state is $PriceOf(Item, Price) \sqcup Price = 10$, which simplifies to $PriceOf(Item, 10)$. Such an approach allows protocol specifications to be general, and then instantiated at runtime.

Definition 10. *Semantics of Sequential composition*

Sequential composition is defined as executing the left-hand protocol until it terminates, and then executing the right-hand protocol until it terminates. The semantics of this is given by two rules.

$$\frac{\langle \pi_1, \phi \rangle \xrightarrow{l_1,\ldots,l_n} \langle E, \phi' \rangle}{\langle \pi_1; \pi_2, \phi \rangle \xrightarrow{l_1,\ldots,l_n} \langle \pi_2, \phi' \rangle} \qquad \frac{}{\langle E; \pi_2, \phi \rangle \xrightarrow{\varepsilon} \langle \pi_2, \phi \rangle}$$

The first of these rules specifies that if sequence of transitions, $l_1, \ldots, l_n$, can be made from $\langle \pi_1, \phi \rangle$ taking the system to the state $\langle E, \phi' \rangle$, then we can perform this transition under the state $\langle \pi_1; \pi_2, \phi \rangle$, leaving us to execute π_2 under the protocol state ϕ'. The second rule specifies that at all times, $E; \pi$ is equivalent to π.

Definition 11. *Semantics of Non-Deterministic Choice*

Non-deterministic choice is defined using two rules.

$$\frac{\langle \pi_1, \phi \rangle \xrightarrow{l_1,\ldots,l_n} \langle E, \phi' \rangle}{\langle \pi_1 \cup \pi_2, \phi \rangle \xrightarrow{l_1,\ldots,l_n} \langle E, \phi' \rangle} \qquad \frac{\langle \pi_2, \phi \rangle \xrightarrow{l_1,\ldots,l_n} \langle E, \phi' \rangle}{\langle \pi_1 \cup \pi_2, \phi \rangle \xrightarrow{l_1,\ldots,l_n} \langle E, \phi' \rangle}$$

These two rules state that, for a protocol $\pi_1 \cup \pi_2$, if one of the arguments can progress, then that argument progresses, and the other is discarded. If both can progress, then a non-deterministic choice is made between the two of them, and the other is discarded. The new protocol state of the argument that is chosen is the new protocol state of the entire transition. The entire protocol terminates when the chosen protocol terminates. A choice between a protocol and the terminated protocol, E, cannot occur because the choice is made before progressing, and because E is not a part of the language syntax.

Definition 12. *Semantics of Protocol References*

$$\frac{}{\langle D, N(x), \phi \rangle \xrightarrow{\varepsilon} \langle D, \pi[x/y], \phi \rangle} \qquad \text{if } N(y) \mathrel{\widehat{=}} \pi \in D$$

This rule specifies that, if there is a protocol named N with variables y and protocol π in the set D, then the reference $N(x)$ is equivalent to the protocol π, with the variables y renamed to x.

For example, the $Bid(Item, Price, B, S)$ protocol, from Section 3, can be referenced as with $Bid(i, p, a, b)$, in which i, p, a, and b are variables. This is equivalent to the following:

$$isItem(i) \xrightarrow{c(a,b).bid(i,p)} \text{true}$$

In which $Item, Price, B$, and S are renamed to i, p, a, and b respectively.

Definition 13. *Semantics for Variable Declaration*

A naïve attempt to specify the semantics for variable declaration would give the following.

$$\frac{\langle \pi, \psi \wedge \phi \rangle \xrightarrow{l_1,\ldots,l_n} \langle E, \phi' \rangle}{\langle \mathbf{var}_x^{\psi} \cdot \pi, \phi \rangle \xrightarrow{l_1,\ldots,l_n} \langle E, \phi' \rangle}$$

The protocol $\mathbf{var}_x^{\psi} \cdot \pi$ specifies that a new variable x is declared with constraints ψ, and then the protocol π, which may refer to x, is executed. Thus, if the protocol π can progress under the protocol state $\psi \wedge \phi$ to the protocol π' and state ϕ', then make this transition.

We have labelled this definition "naïve" because it does not consider three cases. Firstly, it does not consider that case that x is already a free variable in the state. This will cause problems because the behaviour would be to evaluate π in the protocol state $\psi \wedge \phi$, which could be inconsistent. A designer writing

the protocol $\mathbf{var}_x^{\psi} \cdot \pi$ would surely want any references of x in π to refer to the most recently declared x, therefore, in the antecedent of the rule, x is hidden in ϕ using the cylindric operator, so the constraints of the x in the state are hidden. Secondly, it does not remove the local variable x from the state after execution, meaning that the scope of x is not restricted to π. Finally, it does not maintain the constraints on x over the protocol. A second attempt to specify this rule leads to the following:

$$\frac{\langle \pi, \psi \wedge \exists_x \phi \rangle \xrightarrow{l_1,\ldots,l_n} \langle E, \phi' \rangle}{\langle \mathbf{var}_x^{\psi} \cdot \pi, \phi \rangle \xrightarrow{l_1,\ldots,l_n} \langle E, \exists_x \phi' \sqcup \exists_{\widehat{x}} \phi \rangle} \quad \text{where } \exists_{\widehat{x}} \phi = \exists_{\widehat{x}} \phi'$$

In this definition, the state $\psi \wedge \exists_x \phi$ has all references to the previously declared x hidden. The side condition, $\exists_{\widehat{x}} \phi = \exists_{\widehat{x}} \phi'$, says that hiding all variables except x in the pre-state and the post-state will result in the same constraint, therefore, the constraints on x are the same in the post-state and pre-state. Finally, the post-state, $\exists_x \phi' \sqcup \exists_{\widehat{x}} \phi$ hides all references to the local variable x in ϕ', and reinstates the global reference by hiding all variables except the global x in the pre-state, ϕ.

Example 2. We turn to an example to help with the understanding of variable declaration, as its definition is not straightforward. Take the following protocol, which is an expanded version of accepting a bid from the example in Section 3:

$$\mathbf{var}_{Curr}^{PriceOf(Item,Curr)} \cdot \\ Curr < Price \xrightarrow{c(S,B).accept(PriceOf(Item,Curr))} PriceOf(Item, Price)$$

If this is executed with the protocol state $PriceOf(Item, 2) \sqcup Curr = 10$, in which $Curr$ is a variable unrelated to the current bid, then the state value of $Curr$ is inconsistent with the constraints on the local variable $Curr$. So, it is executed under the following state (corresponding to $\psi \wedge \exists_x \phi$ in the definition):

$$PriceOf(Item, Curr) \sqcup \exists_{Curr}(PriceOf(Item, 2) \sqcup Curr = 10)$$

This ensures that any reference to $Curr$ will be referring to the local variable name instead of the global name, and will be constrained to the price of $Item$, which is 2. Consider a message in which the buying agent constrains $Price$ to be 3; that is, sends the message $PriceOf(Item, Price) \sqcup Price = 3$. After this message, the state would be (corresponding to ϕ' in the definition):

$$\begin{array}{ll} (1) & PriceOf(Item, Price) \sqcup Price = 3 \sqcup \\ (2) & \exists_{Item,Price}(PriceOf(Item, Curr) \sqcup PriceOf(Item, 2)) \end{array}$$

From the atomic protocol semantics, we keep the constraints for variables not referenced in the post-state by hiding these variables in the pre-state constraint (line 2), and conjoining this with the postcondition (line 1). This constraint can be simplified to the following:

$$PriceOf(Item, 3) \sqcup Price = 3 \sqcup Curr = 2$$

However, the scope of the local declaration $Curr$ ends there, so we want to remove all references to this $Curr$, and reinstate the global $Curr$ with the same constraints it had prior to the local declaration. Therefore, the local $Curr$ in this constraint is hidden, and the resulting constraint conjoined with the pre-state with all variables except $Curr$ hidden (corresponding to $\exists_x \phi' \sqcup \exists_{\hat{x}} \phi$ in the definition):

$$(1) \qquad \exists_{Item,Price}(PriceOf(Item, 2) \sqcup Curr = 10) \sqcup$$
$$(2) \qquad \exists_{Curr}\big(PriceOf(Item, 3) \sqcup Price = 3 \sqcup Curr = 2\big)$$

Line 1 is the constraint that reinstates the global $Curr$ with the constraints it had prior to the local declaration. The constraint in Line 2 hides the local variable of $Curr$. This can be simplified to the following:

$$Curr = 10 \sqcup PriceOf(Item, 3) \sqcup Price = 3$$

Which is the expected end state of the protocol. With this simplification, one may wonder why we hide the local reference of $Curr$ rather than just removing all references to it. This is because the postcondition may refer to the local variable, in which case $\exists_{Curr}$ could not be removed, because the constraints on $Curr$ may also constrain other variables, such as $Item$ or $Price$.

5 Reusing, Composing, and Reasoning about Protocols

So far, we have outlined how protocol designers can specify a protocol, and have defined the semantics for a protocol given a protocol specification and an initial state. However, crucial to the goals of agent-oriented software engineering is the fact that first-class interaction protocols should be *reusable* and *composable*, and *inspectable* such that agents can reason about protocols to decide their course of action. How designers and agents reuse, compose, and reason about $\mathcal{RASA}$ protocols is not the topic of this paper, however, in this section, we briefly outline how the $\mathcal{RASA}$ protocol specification language supports these requirements. In future work, we plan to investigate these areas in more detail.

Protocols can be referenced via their name, which allows protocols and protocol specifications to be reused to create larger, compound protocols. For example, the simple negotiation protocol specified in Section 3 could be embedded within an auction protocol. Assuming the existence of protocols called $Start$, $DeclareWinner$, and $NoBids$, which represent the auction starting, a winner being declared, and no bids received before a certain condition is met, such as a timeout, one could specify an English auction as follows, in which Bs is a set of bidders:

$$Auction(Item, Price, Bs, S) \mathrel{\widehat{=}} Start; Bids; (DeclareWinner \cup NoWinner)$$

$$Bids(Item, Price, Bs, S) \mathrel{\widehat{=}} \epsilon \cup ((\mathbf{var}_B^{B \in Bs} \cdot Buy); Bids)$$

The protocol $Bids$ is zero of more iterations of the Buy protocol, in which one bidder from the set of bidders, Bs, submits a bid, and it is either accepted or rejected.

$\mathcal{RASA}$ is well-suited for composition. The syntax and semantics of protocol composition operators treat all protocols the same; that is, atomic protocols are complete protocols themselves. Therefore, the syntax and semantics for constructing protocols from atomic protocols can be used to create compound protocols from other compound protocols. The auction example above is an example of composing new protocols from existing protocols.

Protocol composition need not be restricted to protocol designers. Agents able to reason about protocols specified using $\mathcal{RASA}$ could be equipped, in a straightforward manner, with the ability to compose new protocols from existing protocols using planning techniques.

As an example, consider an intelligent agent, I, that believes it can buy a particular item of a rather unintelligent agent, U, and then sell it back to the same agent at a profit. U may propose that itself and I agent engage in two rounds of the Buy protocol from Section 3, but with the buyer and seller swapped in each case:

$$Buy[U/S, I/B]; Buy[I/S, U/B]$$

This composed protocol represents the unintelligent agent acting as the seller, S, and the intelligent agent representing the buyer, B, followed by the same protocol, but with the buyer and seller swapped. Depending on the initial state of the protocol, if I can convince U to engage in this protocol, then I may be able to make a profit. Note that this is different to I proposing $Buy[U/S, I/B]$, and then after this interaction has taken place, proposing $Buy[I/S, U/B]$, because U may not agree to the second protocol, leaving I stuck with the item it does not want. If, however, U agrees to the composed protocol, it is forced to put in a bid for the item if I buys it. This example is fabricated, and would require a highly intelligent agent to devise such a strategy, but should an agent be intelligent enough to exploit this, this example demonstrates that deriving the composite protocol would be straightforward.

There are cases in which protocol composition can lead to problems. For example, take the sequential composition of two protocols: $\pi_1; \pi_2$. For all possible states resulting from the protocol π_1, the precondition from at least one of the paths in π_2 must be enabled, otherwise the execution of the protocol can become *stuck*, in which there are no possible messages that can be sent. To compose this protocol, one must prove that the protocol can never become stuck. Determining such proof obligations, and defining a proof system to help discharge such proof obligations, are part of our ongoing work on the $\mathcal{RASA}$ framework.

Agents can reason about which messages to send by calculating the best course of action at each point in which they can send a message. Each path from the root node to a leaf node represents a possible sequence of interaction. To choose a course of action, classical planning or reactive planning techniques and algorithms can be adapted and used; or more likely, a combination of both. For example, an agent can calculate the end state of all possible interaction sequences of a protocol to decide the next message they send. However, an agent would have to react to changes when it receives a message from another agent, which would likely reduce the choices for its next move.

6 Related Work

There are many languages that have been designed to model agent interactions, such as Social Integrity Constraints [1], FIPA [9], and Agentis [8]. AgentUML [17] has been given a formal semantics for modelling agent interactions [4]. In this section, we discuss and contrast some of the approaches most relevant to the $\mathcal{RASA}$ framework.

Process algebras, such as CSP [11], CCS [15], and the π-calculus [16] are used to model processes and their interactions. While the combination of processes can form the basis of a protocol specification, these languages cannot be used to specify rules and effects. Languages such as Object-Z/CSP [25], which mixes process algebras with state-based languages, are often not inspectable.

Viroli and Ricci [26] propose a method for formalising *operating instructions* for use on mediating coordination artifacts. Sequences of operation instructions resemble our first-class protocols; however their language does not provide the necessary constructs to document the rules or outcomes of protocols. In addition, Viroli and Ricci explicitly comment that their goals are to provide a methodology for environment-based coordination, rather than a general approach to agent interaction semantics.

De Boer *et al.* [6] present a language that uses constraints and process algebra to model agent interactions. However, like many interaction modelling languages, the interaction is emergent from the model of the participants, rather than being first class.

Propositional dynamic logic (PDL) [10] resembles our notion of protocols. For example, PDL allows one to define a collection of sequences of actions, each with an outcome specified as a predicate. In fact, PDL has been extended [19] with belief and intention modal operators to define a language, PDL-BI, for modelling agent interaction. The main differences between these approaches and our approach is that PDL and PDL-BI are declarative languages, while the $\mathcal{RASA}$ language is algebraic; and our language does not require the use of a specific language to model protocol rules and effects. In addition, the class of protocols describable using PDL(-BI) is *regular*, while $\mathcal{RASA}$ allows a larger class; for example, two named protocols with mutually recursive references.

Related work on inspectable protocol specifications also exists in the literature. OWL-P [7] is a language and ontology for modelling protocols, which is coded in the OWL web ontology language [18]. While the approach and goals are different to ours, OWL-P protocols can be used as first-class protocols, and agents would be able to successfully reason about these using OWL tools, and is therefore of interest to us. The syntax and semantics of OWL-P are significantly different to ours, but, like $\mathcal{RASA}$ protocols, OWL-P protocols can be composed. However, unlike $\mathcal{RASA}$, in which composing two protocols has the same syntax and semantics at all levels, OWL-P protocol composition uses a new process with new syntax for composition. We believe this to be a significant advantage of our approach. In addition, OWL-P is not layered; that is, the message sequencing is specified in the same language as the protocol rules and effects — OWL. This restricts designers to using OWL for specification. We also believe that using a

declarative language to specify message sequencing is less intuitive for human readers.

Yolum and Singh [27] present an extension to the Event Calculus [12] that is tailored to first-class protocol specification. The language is declarative, and the authors discuss the use of an abductive planner for agents to plan their execution paths. Although not explicitly discussed by Yolum and Singh, it appears that protocol composition would be possible using this language. This approach is different to ours in the same way that OWL-P is: the language is not layered, and message sequencing is declaratively specified.

Robertson [22] presents the Lightweight Coordination Calculus (LCC). The goals of Robertson are similar to ours, and indeed, one could view the $\mathcal{RASA}$ language as an extended version of LCC, in which we have taken more consideration of the relationship between the protocol specification language and the underlying constraint language by formalising the behaviour of atomic protocols. However, our language differs from LCC in several ways. Firstly, we take a global view of protocols, whereas LCC takes a local view; that is, two interacting agents will each have a specification of their view of the protocol. We believe it would be straightforward to switch between such views in either language. Secondly, our formalisation of atomic protocols treats protocol state differently. Thirdly, we provide a local variable construct, which is useful for defining constraints over sub-protocols, as demonstrated by the example in Section 3. McGinnis [14] has successfully composed LCC protocols at runtime (although McGinnis refers to this as *synthesis*) — a goal clearly in line with our idea of protocol composition.

Serrano *et al.* [24] describe a multi-agent programming framework in which interactions are represented by first-class objects. These objects assert some control over message passing at runtime to guide the interaction. However, this requires the identification of roles at design time, and appears to force participating agents to implement certain interfaces, which we explicitly aim to prevent.

There is also work related to protocol composition in the agent communications literature. McBurney and Parsons [13] propose a formalism for composing dialogue game protocols, which enables similar types of composition, but over a more restricted class of protocols. Reed *et al.* [21] present a framework which allows agents to assign meanings to messages at run-time, and thus, indirectly, to create new interaction protocols.

7 Conclusions and Future Work

In this paper we have presented a novel language for the specification of agent interaction protocols, defining both the syntax and the semantics formally. This language forms part of the $\mathcal{RASA}$ framework, a framework for creating multi-agent interaction protocols as first-class entities. $\mathcal{RASA}$ is general across both the type of interaction protocol and in the language or ontology used by the agents engaged in interaction. By treating interaction protocols as first-class entities, $\mathcal{RASA}$ permits protocols to be inspected, referenced, composed, and shared, by ever-changing collections of agents engaged in interaction. The task

of protocol selection and invocation may thus be undertaken by agents rather than agent-designers, acting at run-time rather than at design-time. Frameworks such as this will be necessary to achieve the full vision of multi-agent systems.

Before such visions are realised, significant further work is required. We aim to develop a proof system for the $\mathcal{RASA}$ framework, which, as well as providing a system for designers to verify properties about their protocols, will provide agents with a way to make decisions about their actions, and to verify protocols composed at runtime. Also, further work is needed on the verification of protocols using this proof system, and the development of verifiable semantics for them within this framework. In addition, we plan to specify a collection of meta-protocols for negotiating which protocols to use, and identify in which contexts each meta-protocol would be useful. To develop and test these ideas, we plan a prototype implementation in which agents negotiate the exchange of information using protocols specified using the $\mathcal{RASA}$ framework.

Acknowledgements We are grateful for financial support from the EC PIPS project (EC-FP6-IST-507019) and the EPSRC Market-Based Control project (GR/T10664/01). We also thank Jarred McGinnis, the anonymous referees and participants at the ESAW 2006 meeting for their comments.

References

1. M. Alberti, D. Daolio, P. Torroni, M. Gavanelli, E. Lamma, and P. Mello. Specification and verification of agent interaction protocols in a logic-based system. In H. M. Haddad, A. Omicini, and R. L. Wainwright, editors, *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 72–78. ACM Press, 2004.
2. F. Baader, D. Calvanese, D. L. McGuinness, and D. Nardiand P. F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation, Applications*. Cambridge University Press, 2003.
3. P. Blackburn, J. van Benthem, and F. Wolter, editors. *Handbook of Modal Logic. North Holland*. Elsevier, 2006.
4. L. Cabac and D. Moldt. Formal semantics for AUML agent interaction protocol diagrams. In J. Odell, P. Giorgini, and J. P. Mller, editors, *Agent-oriented Software Engineering*, volume 3382 of *LNCS*, pages 47–61. Springer, 2004.
5. F. S. De Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. *ACM Transactions on Programming Languages and Systems*, 19(5):685–725, September 1997.
6. F.S. de Boer, W. de Vries, J-J.Ch. Meyer, R.M. van Eijk, and W. van der Hoek. Process algebra and constraint programming for modelling interactions in MAS. *Applicable Algebra in Engineering, Communication and Computing*, (16):113–150, 2005.
7. N. Desai, A. U. Mallya, A. K. Chopra, and M. P. Singh. OWL-P: A methodology for business process modeling and enactment. In *Workshop on Agent Oriented Information Systems*, pages 50–57, July 2005.
8. M. d'Inverno, D. Kinny, and M. Luck. Interaction protocols in Agentis. In *Proceedings of the 3rd International Conference on Multi Agent Systems*, pages 112–119. IEEE Press, 1998.

9. FIPA. FIPA communicative act library specification, 2001.
10. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
11. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
12. R. Kowalski and M.Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.
13. P. McBurney and S. Parsons. Games that agents play: A formal framework for dialogues between autonomous agents. *Journal of Logic, Language and Information*, 11(3):315–334, 2002.
14. J. McGinnis, D. Robertson, and C. Walton. Protocol synthesis with dialogue structure theory. In N. Maudet, P. Moraitis, I. Rahwan, and S. Parsons, editors, *Argumentation in Multi-Agent Systems: Second International Workshop*, LNAI. Springer-Verlag, 2006.
15. R. Milner. *A Calculus of Communicating Systems*. Springer Verlag, 1980.
16. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.
17. J. Odell, H. Van Dyke Parunak, and B. Bauer. Extending UML for agents. In G. Wagner, Y. Lesperance, and E. Yu, editors, *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, pages 3–17, 2000.
18. OWL. Web Ontology Language. http://www.w3.org/TR/2004/REC-owl-features-20040210, Feb, 2004.
19. S. Paurobally, J. Cunningham, and N. R. Jennings. A formal framework for agent interaction semantics. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 91–98. ACM Press, 2005.
20. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, September 1981.
21. C. Reed, T. J. Norman, and N. R. Jennings. Negotiating the semantics of agent communications languages. *Computational Intelligence*, 18(2):229–252, 2002.
22. D. Robertson. Multi-agent coordination as distributed logic programming. In *Proceedings of the International Conference on Logic Programming*, volume 3132 of *LNCS*, pages 416–430. Springer, 2004.
23. V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pages 333–352. ACM Press, 1991.
24. J. M. Serrano, S. Ossowski, and S. Saugar. Reusable components for implementing agent interactions. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Programming Multi-Agent Systems, Third International Workshop*, pages 101–119, 2005.
25. G. Smith and J. Derrick. Abstract specification in Object-Z and CSP. In C. George and H. Miao, editors, *Formal Methods and Software Engineering*, volume 2495 of *LNCS*, pages 108–119. Springer, 2002.
26. M. Viroli and A. Ricci. Instructions-based semantics of agent mediated interaction. In N. R. Jennings, C. Sierra, L. Sonenberg, and M. Tambe, editors, *Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 102–109. IEEE Computer Society, 2004.
27. P. Yolum and M. P. Singh. Reasoning about commitments in the event calculus: An approach for specifying and executing protocols. *Annals of Mathematics and AI*, 42(1–3):227–253, 2004.