Ontologies for Interaction Protocols

Stephen Cranefield, Martin Purvis,
Mariusz Nowostawski and Peter Hwang

Department of Information Science
University of Otago
PO Box 56, Dunedin, New Zealand
scranefield@infoscience.otago.ac.nz

ABSTRACT

In this paper we propose reducing the degree of human interpretation currently necessary to understand an interaction protocol by describing at an abstract level the required agent actions that must be 'plugged into' the protocol for it to be executed. In particular, this can be done by designing and publishing ontologies describing the input and output data that are processed during the protocol's execution together with the actions and decisions that the agents must perform. An agent (or agent developer) that has previously defined mappings between the internal agent code and the actions and decisions in an ontology would then be able to interpret any interaction protocol that is defined with reference to that ontology. The discussion is based on the use of Coloured Petri Nets to represent interaction protocols and the Unified Modeling Language for ontology modelling.

1. INTRODUCTION

Agent communication languages (ACLs) such as the Knowledge Query and Manipulation Language (KQML) [7] and the Foundation for Intelligent Physical Agents (FIPA) ACL [8] are based on the concept of agents interacting with each other by exchanging messages that specify the desired 'performative' (inform, request, etc.) and a declarative representation of the content of the message. Societies of agents cooperate to collectively perform tasks by entering into conversations—sequences of messages that may be as simple as request/response pairs or may represent complex negotiations. In order to allow agents to enter into these conversations without having prior knowledge of the implementation details of other agents, the concept of interaction protocols (also known as conversation policies) has emerged [11]. Interaction protocols are descriptions of standard patterns of interaction between two or more agents. They constrain the possible sequences of messages that can be sent amongst a set of agents to form a conversation of a particular type. An agent initiating a conversation with others can indicate the interaction protocol it wishes to follow, and the recipient (if it knows the protocol) then knows how the conversation is expected to progress. A number of interaction protocols have been defined, in particular as part of the FIPA standardisation process [9].

The specification of the individual messages comprising an interaction protocol is necessarily very loose: usually only the message performative, sender and receiver are described. This is because an interaction protocol is a generic description of a pattern of interaction. The actual contents of messages will vary from one execution of the protocol to the next. Furthermore, the local actions performed and the decisions made by agents, although they may be related to the future execution of the protocol, are traditionally either not represented explicitly (e.g. in an Agent UML sequence diagram representation [17]) or are represented purely as labelled 'black boxes' (e.g. in a Petri net representation [4]).

In this paper we argue that the traditional models of interaction protocols are suitable only as specifications to guide human developers in their implementation of multi-agent systems, and even then often contain a high degree of ambiguity in their intended interpretation. Here we are not referring to the necessity for an interaction protocol to have formal semantics (although that is an important issue). Rather, we see a need for techniques that allow the designers of interaction protocols to indicate their intentions unambiguously so that a) other humans can interpret the protocols without confusion, and b) software agents can interpret protocols for the purposes of generating conversations. Ideally, an agent would be able to download an interaction protocol previously unknown to it, work out where and how to plug in to the protocol its own code for message processing and for domain-specific decision making, and begin using that protocol to interact with other agents.

We propose reducing the degree of human interpretation currently necessary to understand an interaction protocol by describing at an abstract level the required agent actions that must be 'plugged into' the protocol for it to be executed. In particular, this can be done by designing and publishing ontologies describing the input and output data that are processed during the protocol's execution together with the actions and decisions that the agents must perform. An agent (or agent developer) that has previously defined mappings between the internal agent code and the actions and decisions in an ontology would then be able to interpret any interaction protocol that is defined with reference to that ontology.

For example, consider a protocol describing some style of auction. Inherent in this protocol are the concepts of a bid and response and the actions of evaluating a bid (with several possible outcomes). There are also some generic operations related to any interaction protocol such as the parsing of a message to check that it has a particular performative and that its content can be understood by the agent in the current conversational context, and the creation of a message.

2. EXAMPLE: THE FIPA REQUEST PROTOCOL

Figure 1 shows the FIPA Request Protocol as defined in its current experimental-status specification [10] using Agent UML (AUML) [17]. This protocol defines a simple interaction between two agents.

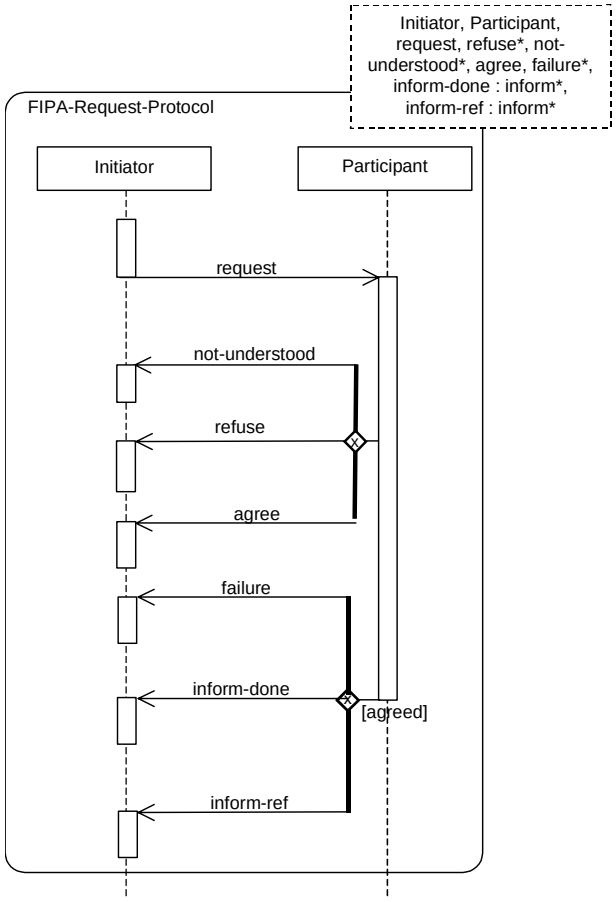

Figure 1: The FIPA Request Protocol defined using AUML

One agent plays the Initiator role and sends a request for an action to be performed to another agent which plays the Participant role. The protocol illustrates that there are three alternative responses that the participant can make after receiving the request: it can refuse or agree to the request or it may signal that it did not understand the request message. If it agreed, it subsequently sends a second response: a message indicating that its attempt to fulfil the request action failed, a message signalling that the action has been performed, or a message containing the result of performing the requested action.

There are some aspects of this protocol that are not specified. The choice of whether the final response should be the message labelled `inform-done` or the one labelled `inform-ref` depends on the body of the original request (the latter choice only seems to be valid if the initiator requested an `inform-ref` to be performed). It is also not specified that each of the `not-understood`, `agree` and `refuse` messages should contain the original request in their content tuple along with an additional proposition (representing respectively an error message, a precondition for the action to be performed, and a reason for refusal). To make the specification more precise there needs to be a way of annotating the protocol with constraints on the contents of and relationships between the messages. These constraints would need to be expressed in terms of a vocabulary relating to the structure of messages—i.e. an ontology for messages.

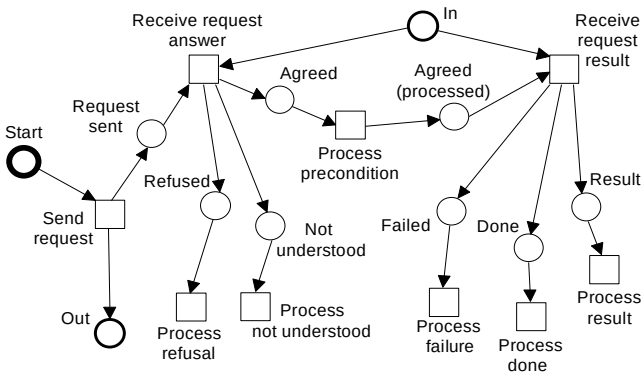

Figure 2: The Initiator role for the Request protocol as a CPN (outline only)

Furthermore, the underlying *intention* of this protocol is not explicitly specified. In order to customise this protocol to a particular domain, a request initiator agent must 'plug in' domain-specific procedures at six different points: the handling of `not-understood`, `refuse` and `failure` messages, analysing an `agree` message to check if a precondition is specified by the participant, and the handling of the two different types of final response. Similarly, there are (in one possible decomposition) three pieces of domain-specific functionality that an agent wishing to play the role of participant must supply: the recognition of the type of the request (corresponding to the two types of response and possibly resulting in a failure to understand the message) and procedures for performing the two different types of action that may be requested. We believe that an interaction protocol is not completely specified until the interface between the domain-specific agent-supplied code and the generic interaction protocol is defined. Clearly interaction protocols should remain as generic as possible, making no commitment to any particular agent platform or implementation language. Thus the specification of this interface should be in terms of a programming-language independent representation. Furthermore, the agent operations related to a particular protocol will be related to the types of entity involved in the execution of that protocol, e.g. the notion of a bid in a 'call for proposals' protocol. This model of protocol-related concepts and the operations that act on them is an ontology that needs to be supplied along with the interaction protocol to give it a full specification.

3. A COLOURED PETRI NET APPROACH

The above discussion was based on an analysis of an interaction protocol expressed as an AUML sequence diagram. However, this form of diagram currently has some shortcomings for further investigation of these ideas. First, AUML is currently underspecified and the intended interpretation of an AUML sequence diagram is not always clear. Second, the authors know of no tools that support the use of AUML. Finally, AUML sequence diagrams do not have a way of explicitly modelling the internal actions of agents[1]—which are exactly the points of the protocol at which we wish to attach annotations refering to an ontology. We have therefore adopted an alternative modelling language for our research in this area: coloured Petri nets.

[1] AUML activity diagrams have this capability and can be used on their own or in conjunction with sequence diagrams to specify the internal agent processing [17]. However there are few examples of their use for modelling agent interaction protocols.

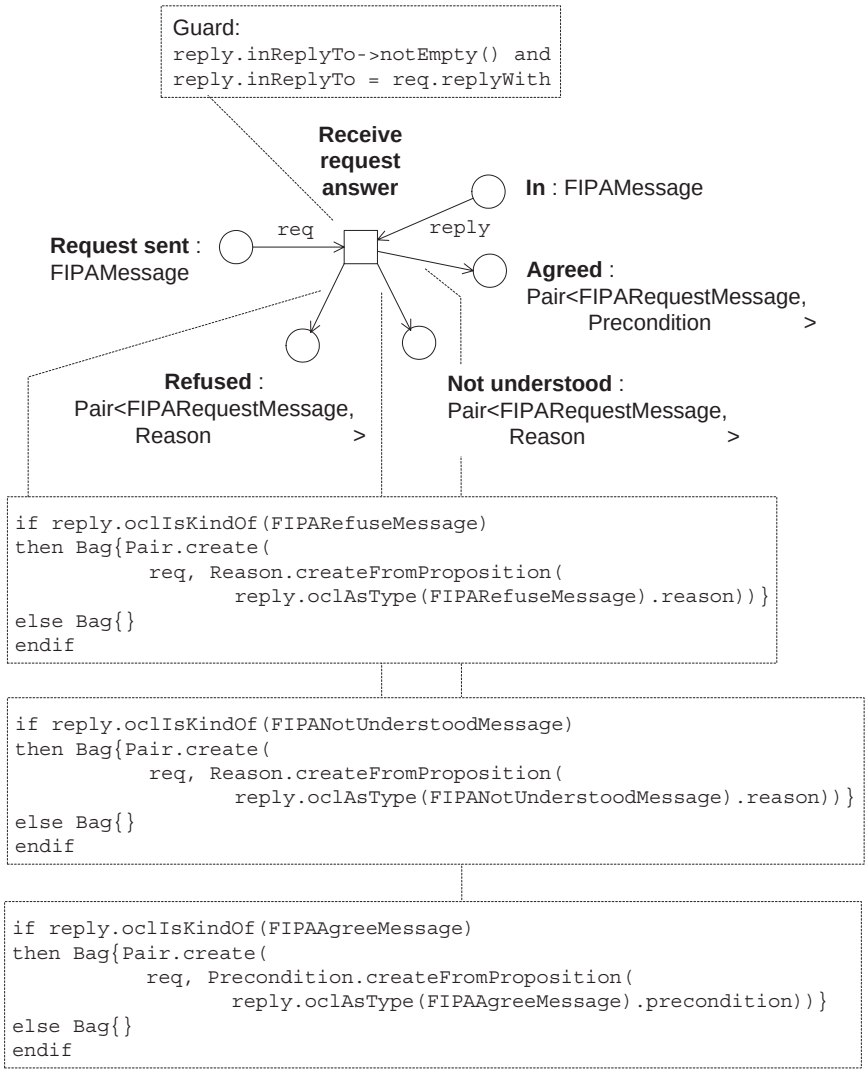

Figure 3: Details of the 'Receive request answer' transition

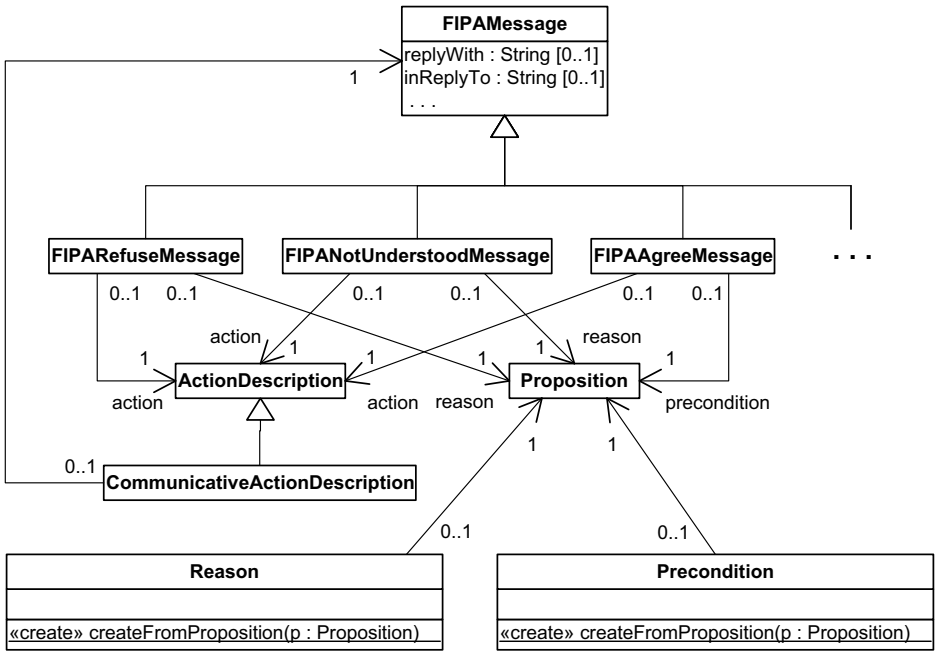

Figure 4: A partial ontology for the Request interaction protocol

Petri Nets [14] are a formalism and associated graphical notation for modelling dynamic systems. The state of the system is represented by *places* (denoted by hollow circles) that can contain *tokens* (denoted by symbols inside the places). The possible ways that the system can evolve are modelled by defining *transitions* that have input and output arcs (denoted by arrows) connected to places. The system dynamics can be enacted (non-deterministically) by determining which transitions are *enabled* by the presence of tokens in the input places, selecting one and *firing* it, which results in tokens being removed from its input places and new tokens being generated and placed in the output places.

Coloured Petri nets (CPNs) [12] are an elaboration of ordinary Petri nets. In a coloured Petri net, each place is associated with a 'colour', which is a type (although the theory of CPNs is independent of the actual choice of type system). Places can contain a multiset of tokens of their declared type. Each input arc to a transition is annotated with an expression (possibly containing variables) that represents a multiset of tokens. For a transition to be enabled, it must be possible to match the expression on each input arc to a sub-multiset of the tokens in the associated input place. This may involve binding variables. In addition, a Boolean expression associated with the transition (its *guard*) must evaluate to true, taking into account the variable bindings. When a transition is fired, the matching tokens are removed from the input places and multiset expressions annotating the output arcs are evaluated to generate the new tokens to be placed in the output places. If the expression on an output arc evaluates to the empty multiset then no tokens are placed in the connected place.

The coloured Petri net formalism provides a powerful technique for defining system dynamics and has previously been proposed for use in modelling interaction protocols [4]. In this paper we take a different approach from previous work (including our own [16, 15, 18]) in the application of CPNs to interaction protocol modelling. We choose to model each side of the conversation (a *role*) using a separate CPN. Figure 2 shows an overview of the net for the Initiator role of the FIPA Request protocol (the 'colours' of places, the arc inscriptions and the initial distribution of tokens are not shown). In the figure, places are represented by circles and transitions are represented by squares. No tokens are shown. The places labelled In and Out are *fusion* places: they are shared between all nets for the roles the agent can play (in any interaction protocol). The agent's messaging system places tokens representing received messages in the In place and removes tokens from the Out place (these represent outgoing messages) and sends the corresponding messages.

The fully detailed version of this Petri net encodes the following process. The Initiator begins the conversation by sending a request with its `reply-with` parameter set to a particular value. When an answer with a matching `in-reply-to` parameter value is received, the Receive request answer transition is enabled and can subsequently fire at the agent's discretion. This transition generates a single token that is placed in one of the Agreed, Refused or Not understood places, depending on the communicative act of the reply (the remaining two of these output places each receive an empty multiset of tokens). In the case that the other agent agreed to the request, another message is subsequently expected from that agent containing the result of the requested action. This is handled by the right hand side of the net in a similar fashion.

In this Petri net we have included transitions that correspond to internal actions of the agent, such as those labelled Process refusal and Process not understood. These are not part of the protocol when it is viewed in the pure sense of simply being a definition of the possible sequences of messages that can be exchanged. However, we believe these communicate the underlying intent of the protocol: there are a number of points at which the agent must invoke particular types of computation to internalise and/or react to the different states that can occur. In the example shown, most of these 'extra' transitions occur after the final states of the pro-

tocol. However, this is not necessarily the case, e.g. the Process precondition transition gives the agent a chance to reason about the precondition that may be specified by the other agent when it agrees to the request. This precondition must become true before the other agent will fulfil the request (in the simple case it can just be the expression `true`). Although the Request protocol does not allow for any extra communication between the two agents regarding this precondition, an agent might wish to do something outside the scope of the conversation to help satisfy the precondition (e.g. perform an action). Therefore, the initiator needs an opportunity to notice the precondition.

Figure 3 shows the details of the Receive request answer transition. This is where we make the connection with ontologies: the types used as place colours and within arc expressions are concepts in an associated ontology (a portion of which is shown in Figure 4). We use the object-oriented Unified Modeling Language (UML) [3] to represent the ontology and UML's associated Object Constraint Language (OCL) [19] as our expression language. For brevity, we adopt the convention that a variable `x` appearing on an input arc represents the singleton multiset `Bag{x}` (`Bag` is the OCL type corresponding to a multiset).

In the case of the Request protocol, the concepts that need to be defined in the associated ontology are message types. Figure 4 therefore defines an inheritance hierarchy of FIPA ACL message types[2] (generalisation relationships are represented by arrows with triangular heads, while associations are represented by arrows with open heads). In addition, we have chosen to explicitly model the concepts of a reason and a precondition that are associated with the Request protocol. Within a FIPA ACL message these are both represented as propositions, but here the `Reason` and `Precondition` classes can be used (via their constructors) to achieve an additional level of interpretation of a proposition. Note that although the ontology is shown here as being a monolithic model, in practice some of the classes shown would be imported from a separate UML package.

In addition to the classes shown, the ontology is assumed to include a UML *class template* called `Pair`. A class template is a class that is defined in terms of one or more other classes, which are specified only as parameter names. When it is used (as in Figure 3) specific types must be supplied to instantiate the parameters. `Pair` represents a pair of elements with the type of each argument being the corresponding supplied parameter.

The arc expressions in Figure 3 use the operations `oclIsKindOf` and `oclAsType`. These are predefined OCL operations used for run-time type checking and type casting respectively.

4. MODELLING INTERNAL AGENT OPERATIONS

In Figure 3, all processing represented by the transition is performed by the guard and the output arc expressions. This is not always the case. Consider the Process request refusal transition from Figure 2 (shown in detail in Figure 5). This represents the computation that an agent must do to react to the participant's refusal of the request. Although any future actions of the initiator

[2] A more complex UML model for FIPA messages has been presented elsewhere [5], but that serves a different purpose. The model in this paper is not intended as an update of that previous work, but instead provides a different view of FIPA message types.

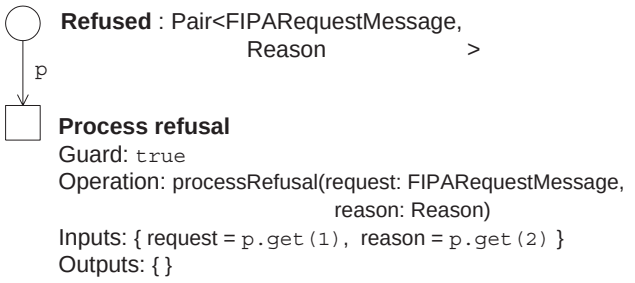

Figure 5: Details of the 'process request refusal' transition

«role» **Initiator**
createPendingRequest() : FIPARequestMessage processRefusal(req : FIPARequestMessage, rsn : Reason) processRequestNotUnderstood(req : FIPARequestMessage, rsn : Reason) processAgreementPrecondition(req : FIPARequestMessage, pre : Precondition) processRequestedActionFailure(req : FIPARequestMessage, rsn : Reason) processRequestDone(req : FIPARequestMessage) processRequestResult(result : GroundTerm)

Figure 6: Specification of the Initiator role for the Request protocol

agent are outside the scope of the Request protocol, in order for the protocol model to act as a stand-alone specification (without relying on implicit assumptions about the meaning of certain places) it should define the way in which the agent transfers information from the Petri net to its own internal processes. To support this, we optionally associate an *operation* with each transition, specifying the inputs to the operation as OCL expressions and providing a list of variables to which the outputs should be assigned (note that UML allows multiple output parameters in an operation). Figure 5 illustrates this for the Process request refusal transition.

The operations required to interface an agent with the CPN for a given role constitute part of the ontology for the protocol. In this section we describe two approaches for using UML to model the operations required for particular roles: a simple "static" approach and a more flexible but complex "dynamic" approach.

4.1 The Static Approach

Figure 6 illustrates the static approach to including a role's operations in a UML ontology. This figure shows a class (annotated with the «role» stereotype) representing the role and containing all required operations. Although this looks like the specification of an application programmer interface rather than an ontology, it is not intended that an agent must implement operations with the same signatures as shown here. Instead an agent may be able to map these operations into those it does possess. To do this, the role's required operations would need to have some information about their semantics specified, possibly using OCL pre- and postcondition expressions. This is a subject for future research.

The representation in Figure 6 does not model the operations required for a given role as first class objects in UML, but as features of a class representing the role. Although this is a simple representation, it has a number of shortcomings. Essentially it treats a role as an interface that an agent must implement if it wants to act in that role. We call this the static approach because it doesn't accommodate in a straightforward way the possibility of agents dynamically changing the roles they support. The UML object model does not

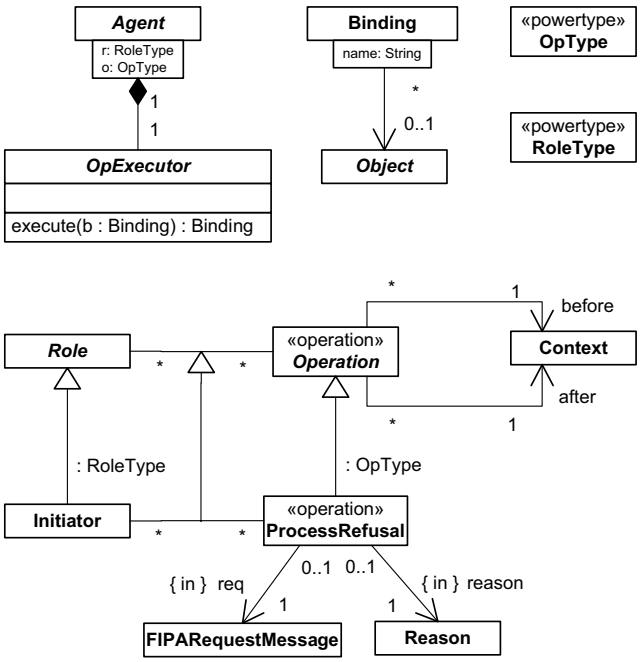

Figure 7: Specifying operations as first-class objects

allow classes (or agent types in this scenario) to change their set of implemented interfaces at run time. Also, the notation does not show graphically the relationships between the operations and the ontological concepts on which they are dependent.

4.2 The Dynamic Approach

Figure 7 shows an alternative approach that addresses the concerns raised above. The majority of the figure represents a base ontology containing classes to which a specific role ontology would make reference. Only the four classes at the bottom of the figure represent a specific ontology: a portion of the ontology for the Request protocol.

Modelling both entities and the operations that act on them as first class objects is difficult to do in a straightforward way without departing from a "strict metamodelling architecture" where there is a firm distinction between instances and classes [1]. In this case, in order to define the types of operation arguments using associations, each operation must be defined as a class. The abstract class `Operation` represents the concept of an operation as being associated with a role and relating two contexts: the relevant local states of the world before and after the operation is performed. Particular operations are modelled as subclasses of `Operation` with their input and output arguments represented by associations labelled with the 'tagged values' `in` or `out` (the stereotype «operation» specializes the notion of a class by allowing the use of these tagged values).

If operations are classes, we need to consider what their instances are. The answer is that the instances represent snapshots of the operation's execution in different contexts and with different arguments, in the same way that a mathematical function can be regarded as the set of all the points on its graph. However, the operation class only serves as a description of the operation: it will not be instantiated by an agent. Instead we model an agent as containing a collection of `OpExecution` objects, each being an instance of some class that implements an operation. These objects are indexed by role and operation (this is shown using UML's qualified association notation). Roles and operations are both modelled as classes, so the types for these association qualifiers must be 'powertypes' of `Operation` and `Role`. A powertype is a class whose instances are all the subclasses of another class [13, Chapter 23].

To invoke an operation, an agent calls `execute` on an `OpExecutor` object. The arguments to this method must be completely generic, so a binding structure is provided as an argument. This maps the operation's argument names to objects. The operation returns another binding list specifying values for the "result" parameter and any output parameters.

5. CONCLUSION

In this paper we have identified two weaknesses in traditional mechanisms for specifying agent interaction protocols: a lack of precision in defining the form of messages that are exchanged during the protocol and the relationships between them, and the lack of any explicit indication of where and how the protocol interfaces with an agent's internal computation. We have proposed the use of an ontology associated with a protocol to define the relevant concepts and the internal operations that an agent needs in order to partake in a conversation using that protocol.

We note that some uses of interaction protocols are not concerned with the internal actions of agents, e.g. external monitoring of conversations for the purpose of compliance testing. For this type of application it may be beneficial to provide a simpler view of protocols that abstracts away the transitions representing internal actions and we plan to investigate techniques for this.

The discussion in this paper was based on a particular way of using coloured Petri nets to model conversations. However, the principle applies to other approaches to specifying interaction protocols. In particular, we propose that AUML sequence and/or activity diagrams should be extended to include the types of ontology-related annotation we have discussed.

Two techniques were proposed for modelling the agent internal actions necessary to use an interaction protocol: a static model and a dynamic model. We believe the dynamic model, although more complex, is more flexible and has more scope for adding semantic annotations to define the operations—an extension necessary to enable agents to deduce how to use their existing operations to implement those required by an interaction protocol.

The type of ontology discussed in this paper combines descriptions of concepts and operations that act on them in a single model. To date, there has been little work on the inclusion of actions in ontologies, although methodologies for agent-oriented software engineering typically include diagrams describing agent capabilities [2]. In the knowledge acquisition research community there has been considerable study of techniques for building libraries of reusable problem-solving methods, and work has been done on combining such libraries and ontologies in a single system [6]. This research may provide some insights into the problems of integrating action descriptions into ontologies.

The aim of the work described in this paper is to reduce the degree of human interpretation required to understand an interaction protocol. The solution proposed here achieves this by including more de-

tailed information about the actions that participating agents must perform. The use of an associated ontology provides terminology for describing how the messages received and sent by agents are related to each other, and also allows signatures to be defined for the operations that agents must be able to perform to use the protocol for its intended purpose. These signatures provide a syntactic specification for the points in the protocol at which the agents must provide their own decision-making and information-processing code, and agent developers could use this to bind internal agent code to these points in the protocol. There is further work to be done to find ways of defining the meaning of these operations so that this binding can be performed on a semantic rather than syntactic basis. This will provide the ability for agents to engage in previously unknown interaction protocols by interpreting the specifications of the protocol and its associated ontologies.

6. REFERENCES

[1] C. Atkinson and T. Kühne. Processes and products in a multi-level metamodelling architecture. *International Journal of Software Engineering and Knowledge Engineering*, 11(6):761–783, 2001.

[2] F. Bergenti and A. Poggi. Exploiting UML in the design of multi-agent systems. In A. Omicini, R. Tolksdorf, and F. Zambonelli, editors, *Engineering Societies in the Agents World*, Lecture Notes in Computer Science 1972, pages 106–113. Springer, 2000. (an earlier version is available at http://lia.deis.unibo.it/confs/ESAW00/pdf/ESAW13.pdf).

[3] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[4] R. Cost, Y. Chen, T. Finin, Y. Labrou, and Y. Peng. Using colored Petri nets for conversation modeling. In F. Dignum and M. Greaves, editors, *Issues in Agent Communication*, volume 1916 of *Lecture Notes in Artificial Intelligence*, pages 178–192. Springer, 2000.

[5] S. Cranefield and M. Purvis. A UML profile and mapping for the generation of ontology-specific content languages. *Knowledge Engineering Review, Special Issue on Ontologies in Agent Systems*, 2002. To appear.

[6] D. Fensel, M. Crubezy, F. van Harmelen, and M. I. Horrocks. OIL & UPML: A unifying framework for the knowledge web. In *Proceedings of the Workshop on Applications of Ontologies and Problem-Solving Methods, 14th European Conference on Artificial Intelligence (ECAI 2000)*, 2000. http://delicias.dia.fi.upm.es/WORKSHOP/ECAI00/14.pdf.

[7] T. Finin, Y. Labrou, and J. Mayfield. KQML as an agent communication language. In J. M. Bradshaw, editor, *Software Agents*. MIT Press, 1997. Also available at http://www.cs.umbc.edu/kqml/papers/kqmlacl.pdf.

[8] Foundation for Intelligent Physical Agents. FIPA ACL message representation in string specification. http://www.fipa.org/specs/fipa00070/, 2000.

[9] Foundation for Intelligent Physical Agents. FIPA interaction protocol library. http://www.fipa.org/repository/ips.html, 2001.

[10] Foundation for Intelligent Physical Agents. FIPA request interaction protocol specification, version F. http://www.fipa.org/specs/fipa00026/, 2001.

[11] M. Greaves, H. Holmback, and J. Bradshaw. What is a conversation policy? In F. Dignum and M. Greaves, editors, *Issues in Agent Communication*, volume 1916 of *Lecture Notes in Artificial Intelligence*, pages 118–131. Springer, 2000.

[12] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Volume 1: Basic Concepts*. Monographs in Theoretical Computer Science. Springer, 1992.

[13] J. Martin and J. J. Odell. *Object-Oriented Methods: A Foundation*. Prentice Hall, Englewood Cliffs, NJ, UML edition, 1998.

[14] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4), 1989.

[15] M. Nowostawski, M. Purvis, and S. Cranefield. A layered approach for modelling agent conversations. In *Proceedings of the 2nd International Workshop on Infrastructure for Agents, MAS, and Scalable MAS, 5th International Conference on Autonomous Agents*, 2001. http://www.cs.cf.ac.uk/User/O.F.Rana/agents2001/papers/06_nowostawski_et_al.pdf.

[16] M. Nowostawski, M. Purvis, and S. Cranefield. Modelling and visualizing agent conversations. In J. P. Müller, E. Andre, S. Sen, and C. Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 234–235. ACM Press, 2001.

[17] J. J. Odell, H. Van Dyke Parunak, and B. Bauer. Representing agent interaction protocols in UML. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering*, volume 1957 of *Lecture Notes in Computer Science*, pages 121–140. Springer, 2001. (Draft version at http://www.auml.org/auml/working/Odell-AOSE2000.pdf).

[18] M. Purvis, S. Cranefield, M. Nowostawski, and D. Carter. Opal: A multi-level infrastructure for agent-oriented software development. Discussion Paper 2002/01, Department of Information Science, University of Otago, PO Box 56, Dunedin, New Zealand, 2002. http://www.otago.ac.nz/informationscience/publctns/complete/papers/dp2002-01.pdf.gz.

[19] J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley, 1998.