In Proceedings of the Third International Conference on Multi-Agent Systems, 112--119, IEEE Press, 1998.

Interaction Protocols in Agentis

Mark d'Inverno
Cavendish School of Computer Science
Westminster University
London W1M 8JS, UK
dinverm@westminster.ac.uk

David Kinny
Department of Computer Science
University of Melbourne
Parkville 3052, Australia
dnk@cs.mu.oz.au

Michael Luck
Department of Computer Science
University of Warwick
CV4 7AL, UK
mikeluck@dcs.warwick.ac.uk

Abstract

Agentis *is a framework for building interactive multi-agent applications which is based upon a model of agent interaction whose key elements are services and tasks. Central to the operation of the system is the set of protocols that permit reliable, concurrent request and provision of services and tasks from and to agents, using an underlying asynchronous point-to-point messaging infrastructure. In this paper we focus on this aspect of the* Agentis *system and provide a detailed description of these protocols, together with a formal specification in* Z*. The specification can be seen as part of a more complete formal specification of the entire system, which provides an integrated and coherent way of describing the system at different levels. In so specifying the* Agentis *protocols, however, we also provide some general guidelines which may be applied to the specification of other protocols for agent interaction.*

1 Introduction

The Agentis system is a framework for building interactive application systems which is an elaboration and implementation of an approach to multi-agent system construction described previously [14, 13]. It is based upon an *agent interaction model* which incorporates specific notions of *services* and *tasks*, and employs a set of protocols to ensure reliable, concurrent request and provision of services and tasks from and to agents. These protocols are layered upon an underlying asynchronous point-to-point messaging infrastructure: the standard one provided by dMARS [10, 3].

Services are units of work performed by Agentis agents. The usual flow of control is that a human user of the system requests one or more services, via a user interface process, from an agent which is dedicated to serving that user. Agents may also request services from and provide services to other agents, or from other interface processes such as database servers. Tasks are smaller units of work whose performance is part of the *service contract* that comes into existence when an agent accepts a service request. For example, to enable an agent to successfully complete a service, a user may be obliged to perform one or more alternative selection or information provision tasks.

In every Agentis system, a set of standard agents provide predefined services for process control and monitoring, user login and session management, and event logging. There are also user-configurable agents that provide user-defined services (specified in a high-level, dMARS-like, graphical process description language). Usually, a system also contains non-agent interface processes, such as user, database, and internet interfaces which can request services from or provide services to Agentis agents. The framework is designed to be flexible and scalable, and to decouple the specification of custom, application-specific behaviour from standard functionality. This makes it easier to design, configure, modify and extend application systems whose key characteristic is that their behaviour may be specified as sets of interactive event-driven or goal-driven processes.

The protocol set provides the ability to request, accept or decline services and tasks, to return execution status information and output values progressively as they become available, and to cancel, suspend and resume services and tasks prior to their completion. In addition, it is designed to guarantee that protocol message collisions, e.g. service cancellation with completion, are benign, that deadlocks do not occur, and that other constraints are enforced, such as the requirement that a task request only occurs in the context of a service provision. It also allows the sending of simple notices or queries, and includes a registration facility which provides service descriptions, access control, and the ability for a service provider to inform a requestor about changes in service availability.

The motivation for and properties of the Agentis interaction model, and the complete framework itself, are described in detail elsewhere [12, 11] and are not the main focus of this paper, which is instead concerned with describing the protocols it uses. Our aims in this paper are threefold. Firstly, we aim to provide a definitive description of the inter-agent communication protocols that underlie Agentis and its operation. Secondly, we aim to provide a formal specification of these protocols as part of a more complete and coherent

formal specification of the entire system. Typically, specification of protocols is a distinct and separate part of system specification, clear divided from other aspects of system functionality. In the specification provided in this paper, these aspects can be integrated. Thirdly, in undertaking the previous aims, we draw out guidelines that may be applied more generally to the specification of other protocols for agent communication [1, 4, 6].

We begin in the next section by considering what is required in the general case of such a protocol specification. This overview provides an indication of how other specifications might be developed and serves to give structure to the specification of the Agentis protocols, which is organised as follows. First, the state machines that describe the protocols are specified. Then instances of these protocols are defined, and this leads to the specification of an Agentis agent and its state. Finally we describe the execution of protocols, so providing an operational specification of the system.

2 Protocol Specification

The specification of interaction protocols is a non-trivial process [5]. In this section, we briefly consider each of the distinct elements of such a specification in general, as a prelude to the specification of the Agentis protocols that follow. This general discussion is, nevertheless, informed by the Agentis protocols.

Identification Agents need a means of identifying other agents, services and tasks, and particular instances. Typically, this will involve the use of *identifier* tokens.

Reasons When a protocol interaction terminates by being *declined* or *cancelled*, a *reason* must be supplied. For example, in Agentis, when a service is declined, a reason such as *temporarily unavailable* is supplied.

Messages Several messages may be sent within the context of a particular protocol. Messages are divided into different *types* such as *requesting*, *accepting*, *updating*, *cancelling*, and so on, and each of these may contain additional type-specific content. For example, when requesting a service, an Agentis agent must provide a request identifier, the name of a service, and a set of input data parameters. When suspending a service, only an identifier indicating the service instance is needed.

States A protocol may be in any one of a number of states, each of which must be defined.

Transitions A protocol may be represented as a directed graph in which the nodes denote the states of the protocol and the arcs, which are labelled by message patterns consisting of types and restrictions on content, denote transitions between states. A protocol may thus be defined as a set of pairs of transitions and message patterns, where each transition is represented as an (*Initial*,*Final*) state pair.

Views The protocol must be specified separately from the point of view of both the requestor and the provider, since asynchronous communication may lead to different current protocol states in the different views. As a consequence, the requestor and provider protocols have subtle differences in their designs, such as those considered in the specification of the Agentis registration protocol below.

Notation

The specification presented below uses the Z language [18], which is being increasingly employed in the agent research arena [2, 7, 15]. Z is a model-oriented formal specification language based on set theory and first-order logic. The key components of a Z specification are definitions of the *state space* of a system and the possible *operations* that transform it from one state to another.

3 The Agentis Protocol Hierarchy

The stance taken in the design of the Agentis interaction model is that it is possible to design a single model to cover all of the different types of agent-agent[1] interactions [12]. However, particular agents may not need to implement the entire interface behaviour.

The protocol set used in Agentis involves four levels.

- At the *registration* level, a requestor agent registers its interest in some class of services provided by a provider agent.
- The *service* level is concerned with the provision of services by a provider agent to a requestor.
- At the *task* level, both the requestor and the provider perform particular tasks which they have both implicitly agreed to perform by the requestor making and the provider accepting a service request.
- Finally, the *notification* level allows agents to notify each other of relevant conditions or events and perform simple queries.

These four levels of operation are the basis for the remainder of the paper, providing an exemplar of how such protocols may be specified formally within an overarching structure for system specification, and at the same time providing a detailed description of this aspect of the Agentis system. To save space and avoid redundancy, however, only a limited presentation follows. In particular, we consider only the registration and service protocols, omitting the task level, which is similar to the service level, and the notification level which, as a protocol, is trivial. The treatment of the omitted protocols is analogous to those presented here.

[1] For ease of exposition in what follows, we use the term *agent* to refer to both agents and interface processes.

3.1 Registration Level Protocols

Registration is the process by which potential service requestors are able to acquire information about the services available from providers, including their identity and parameter type signatures. The set of services an agent provides is typically fixed, and is divided into functional groups, known as service classes. A registration request specifies a particular service class. If registration is successful, the requestor is informed about services of that class and their availability, and will be provided with updates when necessary. Registration also has an access-control function, so that the services offered to a particular requestor may vary according to the identity and privileges of the requestor. A requestor may register with a provider more than once (for different service classes), and a provider may grant concurrent registrations to multiple requestors.

Services themselves are described by service descriptors, which are data structures containing the name, class, and availability status of a service (which may change over time), and the type signature of its data input and output parameters.

We begin the specification of the protocols with the primitives needed for registration. First, to define services, we introduce given sets for services, names of services (which are unique to any given agent), service class names, and service data parameters. Service classes are simply sets of services. We also define the availability status of a service (available, and temporarily or permanently unavailable).

$$[Service, SName, SClassName, DataIO]$$

$$SClass == \mathbb{P}\, Service$$

$$AvailabilityStatus ::= Available \mid Temp\_Unavailable \mid Unavailable$$

$$\begin{array}{|l} ServiceDescriptor \\ \hline name : SName \\ class : SClassName \\ avstatus : AvailabilityStatus \\ dataio : \mathbb{P}\, DataIO \\ \hline \end{array}$$

At the start of a protocol interaction, the requestor supplies a request identifier, which allows it to unambiguously match replies to requests when multiple concurrent requests are made. This identifier must be locally unique to the requestor agent, and it may not be re-used. If the registration is successful, a registration identifier is sent by the provider in its reply, and this is used in all subsequent messages relating to this protocol instance. It too must be locally unique to the provider agent, and may not be re-used.

$$[RequestId, RegnId]$$

If a provider declines a request it is obliged to supply a reason for doing so. The possible reasons include that the requested service class is unknown, that registration is unsupported by the provider, that registration is currently unavailable, that a *shutdown* is pending, that the requestor is already registered or has insufficient privileges or, finally, that some resource is unavailable. We represent this as a type.

$$RDReason ::= RDbad\_class \mid RDunsupported \mid RDunavailable \mid RDshutdown \mid RDregistered \mid RDprivilege \mid RDresource$$

Support for registration is not mandatory, since agents may provide services to unregistered requestors, and an agent may validly decline all registration requests of a particular class or classes. In addition, once a registration has been accepted, either the provider or the requestor agent may cancel it. The cancel message must carry a *reason*, which is either a *shutdown* or a cancellation *acknowledgement*.

$$RCReason ::= RCshutdown \mid RCacknowledge$$

With these primitives defined, we can now define the set of messages in the registration protocol, as well as the *type* of a message. The first definition below details the *content* of each message, while the second details the type of each message and is used to define the graph of the protocol. Notice that since both provider and requestor can send cancel messages it is necessary to distinguish them in the formal representation of the graph.

$$\begin{aligned} RMessage ::= {} & \mathsf{RRequest}\langle\langle RequestId \times SClassName\rangle\rangle \\ & \mid \mathsf{RAccept}\langle\langle RequestId \times RegnId \times (\mathrm{seq}\, ServiceDescriptor)\rangle\rangle \\ & \mid \mathsf{RDecline}\langle\langle RequestId \times RDReason\rangle\rangle \\ & \mid \mathsf{RUpdate}\langle\langle RegnId \times (\mathrm{seq}\, ServiceDescriptor)\rangle\rangle \\ & \mid \mathsf{RReqCancel}\langle\langle RegnId \times RCReason\rangle\rangle \\ & \mid \mathsf{RProCancel}\langle\langle RegnId \times RCReason\rangle\rangle \end{aligned}$$

$$RMessageType ::= \mathsf{RRequest} \mid \mathsf{RAccept} \mid \mathsf{RDecline} \mid \mathsf{RUpdate} \mid \mathsf{RReqCancel} \mid \mathsf{RProCancel}$$

The state machines for requestor and provider appear on the right in Figure 1. The messages that label the arcs are prefixed to denote whether they are sent or received and, in some cases, are distinguished according to their content. If no content restriction is specified, then all possible contents are allowed. For example, in the *RAccepted* state, it is illegal to receive a *cancel(acknowledge)* message, whereas in the *RCanReq* state, a received *cancel* message will normally carry the reason *acknowledge*, but may carry the reason *shutdown* if a message collision has occurred.

To specify the registration protocols, we describe the directed graphs as sets of tuples. As stated earlier, each tuple represents a pair of adjacent states and the message type that labels the arc between them, with an optional content parameter for some message types. The states are enumerated first.

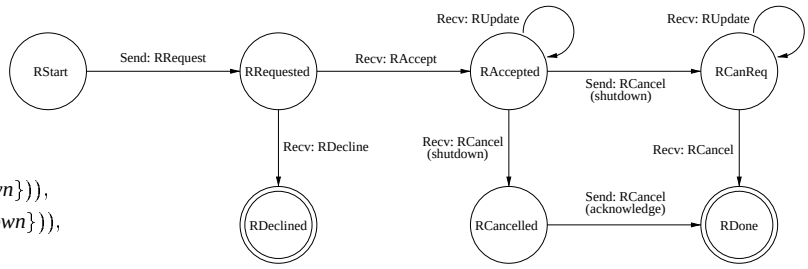
$$RState ::= RStart \mid RRequested \mid RDeclined \mid RAccepted \mid RCancelled \mid RCanReq \mid RDone$$

The protocol specifications appear on the left in Figure 1. The provider protocol is defined as a modification of the requestor protocol, highlighting some subtle but fundamental differences between the two. Notice, for example, that while it is not legal for the provider to send an *update* message after receiving or sending a *cancel* message, it is possible for the requestor to receive an *update* after sending a *cancel* (but not after receiving one), since the asynchronous nature of the

$\mathbf{RRequestorGraph} : \mathbb{P}((RState \times RState) \times (RMessageType \times \text{optional } [RCReason]))$

$\mathbf{RRequestorGraph} =$

$\{((RStart, RRequested), (\text{RRequest}, \{\})),$
$((RRequested, RDeclined), (\text{RDecline}, \{\})),$
$((RRequested, RAccepted), (\text{RAccept}, \{\})),$
$((RAccepted, RAccepted), (\text{RUpdate}, \{\})),$
$((RAccepted, RCanReq), (\text{RReqCancel}, \{RCshutdown\})),$
$((RAccepted, RCancelled), (\text{RProCancel}, \{RCshutdown\})),$
$((RCanReq, RCanReq), (\text{RUpdate}, \{\})),$
$((RCanReq, RDone), (\text{RProCancel}, \{\})),$
$((RCancelled, RDone), (\text{RReqCancel}, \{RCacknowledge\}))\}$



$\mathbf{RProviderGraph} : \mathbb{P}((RState \times RState) \times (RMessageType \times \text{optional } [RCReason]))$

$\mathbf{RProviderGraph} = \mathbf{RRequestorGraph} \setminus$

$\{((RCanReq, RCanReq), (\text{RUpdate}, \{\})),$
$((RCanReq, RDone), (\text{RProCancel}, \{\})),$
$((RCancelled, RDone), (\text{RReqCancel}, \{RCacknowledge\}))\}$
$\cup$
$\{((RCanReq, RDone), (\text{RProCancel}, \{RCacknowledge\})),$
$((RCancelled, RDone), (\text{RReqCancel}, \{\}))\}$

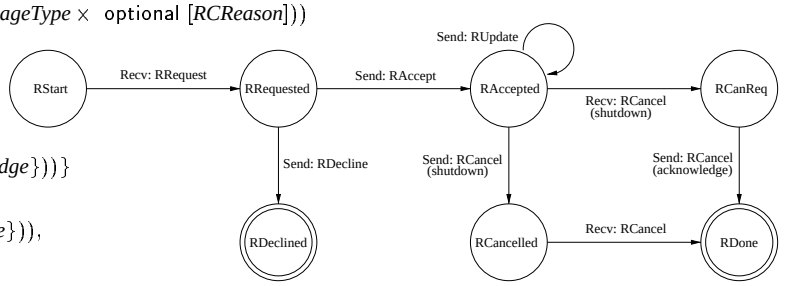


Figure 1. Registration Requestor and Provider Protocols

underlying message system may lead to message collisions. The protocols are designed to ensure that all collisions that may legally occur are benign.

Furthermore, the protocols are also designed to ensure an important safety property: *termination*. One aspect of this is that both requestor and provider can safely assume that once their view of an instance of the protocol has reached a terminal state, no further messages associated with that instance will be received, and thus it may safely be deleted. This explains, for example, why the canceller must wait for an acknowledgement, since otherwise an update or cancel from the other might arrive after the instance had been deleted, causing a spurious protocol error even though the protocol had not actually been violated. Another aspect is to ensure that termination cannot be infinitely delayed, which is addressed by a timeout mechanism discussed in Section 4.2.

3.2 Service Level Protocols

As a further example, we formalise the service level protocols. Once a requestor has registered with a provider[2] it may request one or more services, perhaps concurrently. Requestors are required to identify the service and to supply values for all mandatory input parameters. Requesting a service may affect the availability of that service and of others, both to the requestor agent and to other agents. Any such effects are notified to agents registered with the provider agent via the update facility of the registration level.

[2] As mentioned previously, service providers may accept requests from unregistered agents. In this case, descriptions of the services that will be requested must be built into the requestor agent.

The service level protocols are somewhat more complex than those of the registration level, as a result of the greater functionality provided. A service provider may return execution status information and output values progressively: at acceptance[3], during execution (via *update* messages), and at completion. Prior to completion, the status of the service is said to be *interim*, while at completion it is *final*. The requestor may suspend, resume or cancel the service instance. Note that the provider may not cancel, however the same effect may be achieved by completing with a failure status.

The formal description below is similar to that above and is presented with little elaboration, as the meaning should be readily apparent. The task and notification level protocols may be similarly specified. We first introduce service identifiers, and service parameters which need to be supplied by the requestor when making a request.

$[ServId, SParam]$

If a service is declined a reason must be provided.

$$SDReason ::= Sbad\_service \mid SDunavailable \mid SDnot\_registered \mid SDtry\_later \mid SDalready\_active \mid SDshutdown \mid SDbad\_params \mid SDprivilege \mid SDresource$$

Service instance statuses are defined as follows.

$$SStatus ::= Spending \mid Sactive \mid Ssuspended \mid Ssusreq \mid Sfailed \mid Ssucceeded \mid Sshutdown \mid Scancelled$$

$Sinterim, Sfinal : \mathbb{P}\ SStatus$

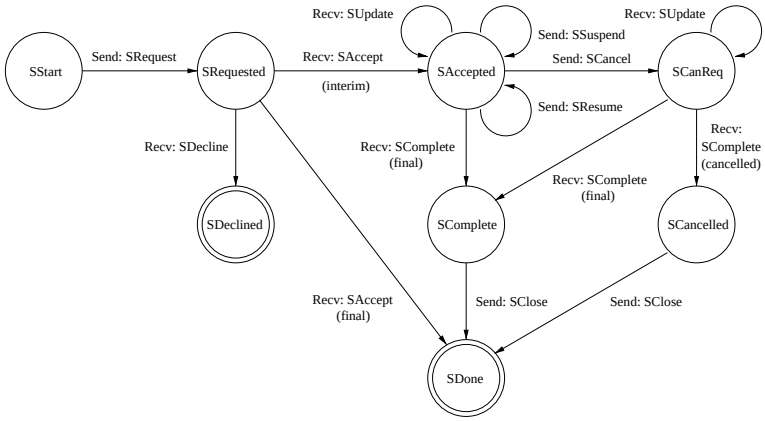
$Sinterim = \{Spending, Sactive, Ssuspended, Ssusreq\}$
$Sfinal = \{Ssucceeded, Sfailed, Sshutdown\}$

[3] This allows requestor and provider to negotiate the terms of the service agreement, and the optimization of message traffic for brief services [12].

$\mathbf{SRequestorGraph} : \mathbb{P}((SState \times SState) \times (SMessageType \times \text{optional } [SStatus]))$

$\mathbf{SRequestorGraph} =$

$\{((SStart, SRequested), (\text{SRequest}, \{\})),$
$((SRequested, SDeclined), (\text{SDecline}, \{\})),$
$((SRequested, SAccepted), (\text{SAccept}, \{s : Sinterim\})),$
$((SAccepted, SAccepted), (\text{SUpdate}, \{\})),$
$((SAccepted, SAccepted), (\text{SSuspend}, \{\})),$
$((SAccepted, SAccepted), (\text{SResume}, \{\})),$
$((SAccepted, SComplete), (\text{SComplete}, \{s : Sfinal\})),$
$((SComplete, SDone), (\text{SClose}, \{\})),$
$((SRequested, SDone), (\text{SAccept}, \{s : Sfinal\})),$
$((SAccepted, SCanReq), (\text{SCancel}, \{\})),$
$((SCanReq, SCanReq), (\text{SUpdate}, \{\})),$
$((SCanReq, SComplete), (\text{SComplete}, \{s : Sfinal\})),$
$((SCanReq, SCancelled), (\text{SComplete}, \{Scancelled\})),$
$((SCancelled, SDone), (\text{SClose}, \{\}))\}$



$\mathbf{SProviderGraph} : \mathbb{P}((SState \times SState) \times$
$(SMessageType \times \text{optional } [SStatus]))$

$\mathbf{SProviderGraph} = \mathbf{SRequestorGraph} \setminus$
$\{((SCanReq, SCanReq), (\text{SUpdate}, \{\})),$
$((SCanReq, SComplete), (\text{SComplete}, \{s : Sfinal\}))\}$
$\cup$
$\{((SComplete, SComplete), (\text{SSuspend}, \{\})),$
$((SComplete, SComplete), (\text{SResume}, \{\})),$
$((SComplete, SCancelled), (\text{SCancel}, \{\}))\}$

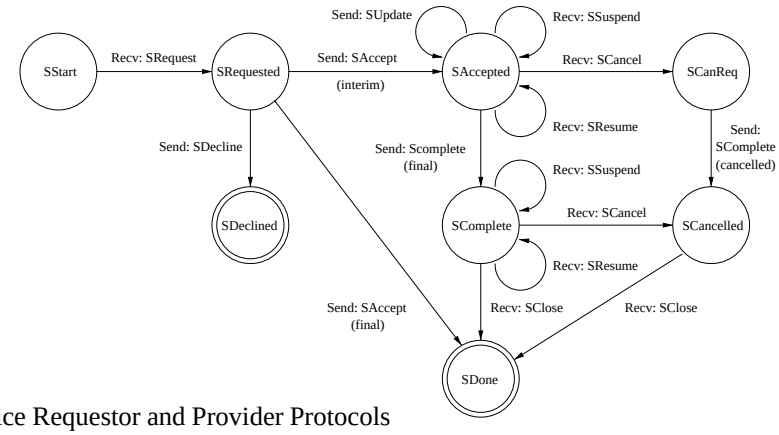


Figure 2. Service Requestor and Provider Protocols

The set of messages associated with the service level protocols and the corresponding message types are defined in the same mannner as for the registration level.

$$\begin{aligned} SMessage ::= {} & \text{SRequest}\langle\langle RequestId \times SName \times \mathbb{P}\, SParam\rangle\rangle \\ & | \text{SAccept}\langle\langle RequestId \times ServId \times SStatus \times \mathbb{P}\, SParam\rangle\rangle \\ & | \text{SDecline}\langle\langle ServId \times SDReason\rangle\rangle \\ & | \text{SUpdate}\langle\langle ServId \times SStatus \times \mathbb{P}\, SParam\rangle\rangle \\ & | \text{SSuspend}\langle\langle ServId\rangle\rangle \\ & | \text{SResume}\langle\langle ServId\rangle\rangle \\ & | \text{SComplete}\langle\langle ServId \times SStatus \times \mathbb{P}\, SParam\rangle\rangle \\ & | \text{SCancel}\langle\langle ServId\rangle\rangle \\ & | \text{SClose}\langle\langle ServId\rangle\rangle \end{aligned}$$

$$\begin{aligned} SMessageType ::= {} & \text{SRequest} \mid \text{SAccept} \mid \text{SDecline} \mid \\ & \text{SUpdate} \mid \text{SSuspend} \mid \text{SResume} \mid \\ & \text{SComplete} \mid \text{SCancel} \mid \text{SClose} \end{aligned}$$

The protocol states are enumerated as follows.

$$\begin{aligned} SState ::= {} & SStart \mid SRequested \mid SDeclined \mid SAccepted \mid \\ & SComplete \mid SDone \mid SCanReq \mid SCancelled \end{aligned}$$

The service requestor and service provider protocol specifications appear on the left in Figure 2, and the corresponding state machines on the right.

4 Protocol Instances

At run-time, an Agentis agent may be involved in several protocol instances as either a requestor or a provider. In order to record the *state* of every instance of a protocol in which the agent is engaged, the agent must record all the information given by the content of the protocol. In this section, we provide details of what is needed for the general case of specifying protocol instances, and then consider the different possible modes of such a protocol instance, which may be either requested (initial), current or finished. Finally, we show how the general case may be instantiated for a particular protocol by using the registration level as an example.

4.1 Generic Protocol Instances

To define an instance of a protocol, we need to introduce several new components:

- the protocol type,
- the *view* of the agent as a provider or requestor (which can be determined by inspecting the protocol type),
- the provider identifier (which is optional since it is not defined until acceptance),

- the current state of the protocol,
- the identifier of the agent with whom the protocol interaction is taking place,
- the status value associated with the request (which may be a list of service descriptors or the status of an individual service or task), and
- a reference to the resource the protocol is concerned with (which is either a service class, a service or a task).

Each of these is specified formally below. Firstly, if protocol information is maintained locally, then the protocol of which it is an instance must be recorded.

$$\begin{aligned} \mathit{Protocol} ::= {} & \mathit{rr}\langle\langle \mathbf{RRequestorGraph} \rangle\rangle \\ | \; & \mathit{rp}\langle\langle \mathbf{RProviderGraph} \rangle\rangle \\ | \; & \mathit{sr}\langle\langle \mathbf{SRequestorGraph} \rangle\rangle \\ | \; & \mathit{sp}\langle\langle \mathbf{SProviderGraph} \rangle\rangle \end{aligned}$$

However, in this specification protocols have been defined as global rather than local data structures, so an agent need only record the type of the protocol from which the instance is generated.

$$\mathit{ProtocolType} ::= \mathit{RR} \mid \mathit{RP} \mid \mathit{SR} \mid \mathit{SP}$$

The remaining components are straightforward.

$$\mathit{View} ::= \mathit{Req} \mid \mathit{Pro}$$
$$\mathit{ProvideId} ::= \mathit{regnid}\langle\langle \mathit{RegnId} \rangle\rangle \mid \mathit{servid}\langle\langle \mathit{ServId} \rangle\rangle$$
$$\mathit{State} ::= \mathit{regstate}\langle\langle \mathit{RState} \rangle\rangle \mid \mathit{servstate}\langle\langle \mathit{SState} \rangle\rangle$$
$$[\mathit{AgentId}]$$
$$\mathit{Status} ::= \mathit{regstatus}\langle\langle \mathrm{seq}\, \mathit{ServiceDescriptor} \rangle\rangle \mid \mathit{servstatus}\langle\langle \mathit{SStatus} \rangle\rangle$$
$$\mathit{Resource} ::= \mathit{class}\langle\langle \mathit{SClassName} \rangle\rangle \mid \mathit{service}\langle\langle \mathit{SName} \rangle\rangle$$

It is now possible to define a protocol instance.

ProtocolInstance

$$\begin{aligned} &\mathit{type} : \mathit{ProtocolType} \\ &\mathit{view} : \mathit{View} \\ &\mathit{requestid} : \mathit{RequestId} \\ &\mathit{provideid} : \mathsf{optional}\ [\mathit{ProvideId}] \\ &\mathit{state} : \mathit{State} \\ &\mathit{who} : \mathit{AgentId} \\ &\mathit{status} : \mathit{Status} \\ &\mathit{resourcename} : \mathit{Resource} \end{aligned}$$

4.2 Modes of Protocols Instances

From the perspective of an agent, a protocol instance may be in one of three modes (sets of states): *requested*, *current* or *finished*. When an agent sends a request, it instantiates the protocol in the requested state. When this is received by the provider, it does likewise. In this mode, the provider identifier is not defined. The finished mode consists of the terminal states of the protocol; *declined*, *done*, *timedout* and *violated*. The latter two states exist to guarantee termination and guard against implementation errors; the protocols are augmented by adding an edge with a timeout condition from every state in which an agent can only receive a message to the *timedout* state, and an edge for every possible illegal message to the *violated* state. Once a protocol instance is in the finished mode, it may be deleted. If a protocol instance is neither in the requested or finished mode, it is current.

The requested mode is defined below as *InitInstance*. It is straightforward to similarly specify the current and finished modes.

InitInstance

$$\mathit{ProtocolInstance}$$

$$\begin{aligned} &\mathsf{undefined}\ \mathit{ProvideId} \\ &\mathit{state} \in \{\mathit{regstate\ RRequested}, \mathit{servstate\ SRequested}\} \end{aligned}$$

5 Agentis Agents

Now that a detailed picture of the Agentis protocols has been painted, we can proceed to consider the structure of individual agents.

A general Agentis agent is defined by a list of components as follows: the set of services it can provide (which may be empty); the set of service classes, with each service belonging to at least one service class; a unique name for each of its services; and a unique name for each of its classes. Similar data structures for tasks and notices are also needed.

Although agents can, in principle, be involved in any protocol, a particular agent may only be able to access a limited number or type of protocols. For example, it may be restricted to being a service provider. In addition, an agent may only be able to communicate with a set of *contactable* agents. Further requirements for individual systems may also hold, but are not discussed here.

AgentisAgent

$$\begin{aligned} &\mathit{services} : \mathbb{P}\, \mathit{Service} \\ &\mathit{serviceclasses} : \mathbb{P}\, \mathit{SClass} \\ &\mathit{servicename} : \mathit{Service} \rightarrowtail \mathit{SName} \\ &\mathit{serviceclassname} : \mathit{SClass} \rightarrowtail \mathit{SClassName} \\ &\mathit{protocols} : \mathbb{P}\, \mathit{ProtocolType} \\ &\mathit{agents} : \mathbb{P}\, \mathit{AgentId} \end{aligned}$$

$$\begin{aligned} &\mathit{services} \subseteq \bigcup \mathit{serviceclasses} \\ &\mathrm{dom}\, \mathit{servicename} = \mathit{services} \\ &\mathrm{dom}\, \mathit{serviceclassname} = \mathit{serviceclasses} \end{aligned}$$

An agent's runtime state is now completely defined by the *AgentisAgent* schema and its set of protocol instances.

AgentisState

$$\begin{aligned} &\mathit{AgentisAgent} \\ &\mathit{Rinstances}, \mathit{Sinstances} : \mathbb{P}\, \mathit{ProtocolInstance} \end{aligned}$$

6 Protocol Execution

To complete the specification, we turn to a consideration of the execution of the protocols. We first define an auxiliary relation, *rel*, which holds between a received message and a protocol instance if the message can be validly associated with the protocol instance, on the basis of the the message type, its sender, and the contained requestor or provider identifier. Only the signature is provided here.

$$rel\_ : \mathbb{P}(Message \times ProtocolInstance)$$

Now we can specify the actual communication between agents. In this paper we limit our consideration to the receipt of messages as opposed to sending them, which is a more complex matter. We also do not consider here details of the handling of protocol violations and timeouts.

6.1 Receiving Messages

Whenever a message is received, the message is an input, and the agent does not change (though its state might).

ReceiveMessage

$in? : Message$

$\Xi AgentisAgent$

$AgentisState$

There are then two possible instantiations of this general case: either the message is legitimate and the agent can respond by updating its state accordingly or the message is illegitimate and must not change the agent's state. In the latter case an error should be reported.

ReceiveMessageSuccess

$ReceiveMessage$

$\Delta AgentisState$

ReceiveMessageFail

$ReceiveMessage$

$\Xi AgentisState$

$report! : Report$

On receipt of a message, two further scenarios may arise depending on whether or not the message is associated with an existing protocol instance of the same protocol type.

Messages Associated with an Existing Protocol

The general schema for the scenario with an associated message is specified below, followed by considerations of various sub-cases. Note that, due to the uniqueness constraints imposed upon identifiers, a message can be be associated with at most one protocol instance. We assume that the agent has access to both service and registration protocols as provider and requestor, and that it can communicate with the sender of the input message.

AssociatedInstance

$ReceiveMessage$

$in? \in (\operatorname{ran} regnmsg) \Rightarrow (\exists\, i : Rinstances \bullet rel(in?, i))$

$in? \in (\operatorname{ran} servmsg) \Rightarrow (\exists\, i : Sinstances \bullet rel(in?, i))$

Let us assume that the associated protocol instance is P_i, that the message type is M, and that the graph of the associated protocol instance is G. There are then three cases (which we do not specify in full).

1. $\#(\{P_i.current\} \lhd (\{G\} \rhd \{M\})) = 0$

   The message does not move the protocol to a legal state. For example, consider Figure 1, protocol instance which is in the *RAccepted* state. If it receives a *RDecline* message associated with this instance then there is no next state defined. In this case there is a *protocol violation*.

2. $\#(\{P_i.current\} \lhd (\{G\} \rhd \{M\})) = 1$

   The message type moves the protocol instance to a unique new state. For example, if the same protocol instance is in the *RRequested* state and receives a *RDecline* message, there is only one possible arc to traverse: the one which moves the protocol instance to the *RDeclined* state.

3. $\#(\{P_i.current\} \lhd (\{G\} \rhd \{M\})) > 1$

   The message type moves the protocol instance to more than one state. In this case, the agent needs to consider the content of the message in order to determine the appropriate transition to take. For example, given a *SAccept* message in the *SRequested* state, the next state could be either *SDone* or *SAccepted*, depending on whether the status is final or interim.

Messages Unrelated to an Existing Protocol

Alternatively, a message may not be associated with an existing protocol instance – it is either illegal or a new request. The general schema is given below, with further relevant predicates following, rather than a complete specification of the operation. The *MessageSender* function returns the identifier of the agent who sent the message.

NoAssociatedInstance

$ReceiveMessage$

$in? \in (\operatorname{ran} regnmsg) \Rightarrow \neg\, (\exists\, i : Rinstances \bullet rel(in?, i))$

$in? \in (\operatorname{ran} servmsg) \Rightarrow \neg\, (\exists\, i : Sinstances \bullet rel(in?, i))$

$\{RP, RR, SP, SR\} \subseteq protocols$

$MessageSender\ in? \in agents$

If the message is a registration or service request, it starts a new instance of the corresponding protocol in the initial state. Otherwise, it is necessarily illegal.

ReceiveRegistrationRequest

NoAssociatedInstance

ReceiveMessageSuccess

$in? \in \operatorname{ran} regnmsg$

$regnmsg^{-1} in? \in \operatorname{ran} \mathsf{RRequest}$

$Rinstances = Rinstances' \cup \{(\mu\, i : InitInstance \mid$
$\quad i.view = Pro \land i.requestid = Requestid\, in? \land$
$\quad i.who = MessageSender\, in?)\}$

ReceiveServiceRequest

NoAssociatedInstance

ReceiveMessageSuccess

$in? \in \operatorname{ran} servmsg$

$servmsg^{-1} in? \in \operatorname{ran} \mathsf{SRequest}$

$Sinstances = Sinstances' \cup \{(\mu\, i : InitInstance \mid$
$\quad i.view = Pro \land i.requestid = Requestid\, in? \land$
$\quad i.who = MessageSender\, in?)\}$

ReceiveIllegalNonRequest

NoAssociatedInstance

ReceiveMessageFail

$in? \in \operatorname{ran} regnmsg \land regnmsg^{-1} in? \notin \operatorname{ran} \mathsf{RRequest}$

$in? \in \operatorname{ran} servmsg \land servmsg^{-1} in? \notin \operatorname{ran} \mathsf{SRequest}$

$report! = Illegal\_Non\_Request\_Message$

7 Summary and Conclusions

In this paper we have presented key elements of the Agentis agent interaction model by describing the Agentis protocols. These have been specified formally using the Z specification language, which is widely used for system description and specification, both in academia and in industry, and which provides an effective way of reconciling descriptions of both the system and the protocols in a single coherent whole. The paper also illuminates the specification of other protocols by providing a clear structure for their organisation. Indeed, the structure of the specification is described in general terms before the specification itself.

Our aim is to contribute to bringing together the different strands of research and development in multi-agent systems by taking a commercially developed system and providing it with a firm, formal foundation that lends itself to further analysis and investigation. Ideally, we aim to arrive at a situation in which these feed off each other, with application being informed by formal analysis and *vice versa.*

The immediate next stage in the development of this work is to take advantage of the wealth of tool support for Z for both animation [8] and proof [16]. The specification presented here has been checked for type correctness using the fuzz package [17]. We are also specifying the protocols using CSP [9] in order to prove critical safety and liveness properties of the Agentis system.

References

[1] J. A. Campbell and M. P. D'Inverno. Knowledge interchange protocols. In *Decentralized AI: Proceedings of the First European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, pages 63–80. Elsevier, 1990.

[2] I. D. Craig. *The Formal Specification of Advanced AI Architectures*. Ellis Horwood, 1991.

[3] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages (ATAL-97). LNAI 1365*. Springer, 1997.

[4] T. Finin and R. Fritzson. KQML — a language and protocol for knowledge and information exchange. In *Proceedings of the Thirteenth International Workshop on Distributed Artificial Intelligence*, Lake Quinalt, WA, 1994.

[5] M. Fisher and M. Wooldridge. Specifying and executing protocols for cooperative action. In *Proceedings of the Second International Working Conference on Cooperating Knowledge-Based Systems (CKBS-94)*. Springer, 1994.

[6] Foundation for Intelligent Physical Agents. *FIPA 97 Specification Part 2: Agent Communication Language*, November 1997. Version 1.0.

[7] R. Goodwin. A formal specification of agent properties. *Journal of Logic and Computation*, 5(6), 1995.

[8] M. A. Hewitt, C. M. O'Halloran, and C. T. Sennet. Experiences with PiZA, an animator for Z. In *10th International Conference of Z Users (ZUM'97)*. Springer, 1997.

[9] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1986.

[10] D. Kinny. *The Distributed Multi-Agent Reasoning System Architecture and Language Specification*. Australian Artificial Intelligence Institute, Melbourne, Australia, 1993.

[11] D. Kinny. Agentis – a framework for commercial multi-agent system development. Technical Report 83, Australian Artificial Intelligence Institute, Melbourne, Australia, 1998.

[12] D. Kinny. The Agentis agent interaction model. In *Intelligent Agents V: Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98)*, Paris, 1998.

[13] D. Kinny and M. Georgeff. Modelling and design of multi-agent systems. In *Intelligent Agents III: Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL-96). LNAI 1193*. Springer, 1996.

[14] D. Kinny, M. Georgeff, and A. Rao. A methodology and modelling technique for systems of BDI agents. In *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW '96). LNAI 1038*. Springer, 1996.

[15] M. Luck and M. d'Inverno. A formal framework for agency and autonomy. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 254–260. AAAI Press / MIT Press, 1995.

[16] M. Saaltink. The Z/EVES system. In *10th International Conference of Z Users (ZUM'97)*. Springer, 1997.

[17] J. M. Spivey. *The fuzz Manual*. Computing Science Consultancy, 2 Willow Close, Garsington, Oxford OX9 9AN, UK, 2nd edition, 1992.

[18] J. M. Spivey. *The Z Notation*. Prentice Hall, Hemel Hempstead, 2nd edition, 1992.