A Performative Type Hierarchy and Other Interesting Considerations in the Design of the CASA Agent Architecture

Rob Kremer
Department of Computer Science
University of Calgary
2500 University Dr. N.W.
Calgary, Canada, T2N-1N4
(403) 220-5112
kremer@cpsc.ucalgary.ca

Roberto Flores
Institute of Cognitive Sciences and Technologies
National Research Council
Viale Marx 15, 00137, Rome, Italy
39 (06) 86090518
robertof@ip.rm.cnr.it

Chad La Fournie
Department of Computer Science
University of Calgary
2500 University Dr. N.W.
Calgary, Canada, T2N-1N4
(403) 210-9546
laf@cpsc.ucalgary.ca

ABSTRACT

In this paper, we describe several interesting design decisions we have taken (with respect to inter-agent messaging) in the re-engineered CASA architecture for agent communication and services. CASA is a new architecture designed from the ground up; it is influenced by the major agent architectures such as FIPA, CORBA, and KQML but is intended to be independent (which doesn't imply incompatible). The primary goals are flexibility, extendibility, simplicity, and ease of use. The lessons learned in the earlier implementation have fed the current design of the system. Among the most interesting of the design issues are the use of performatives that form a type lattice, which allows for observers, who do not necessarily understand all the performatives, to nonetheless understand a conversation at an appropriate semantic level. Furthermore, we found it difficult to arrange a portion of the FIPA performatives in a lattice without duplication and complexity. We solved this problem by diverging from FIPA performatives by separating a lot of the performatives into a separate lattice of *acts*, which work with the performatives, but greatly simplify and enhance the semantic interpretation of the messages. Yet another innovation is the addition of several new fields within a KQML-style message header that allow for further semantic interpretation of the message by an observer who does not necessarily understand the content language. The traditional FIPA fields *to, from, sender*, and *receiver* have been extended to include *agent* (the requester of a service), and *actor* (the party designated responsible for performing the action specified in the *performative/act*). These new design considerations add a great deal of flexibility and integrity to an agent communications architecture.

Keywords

Conversation Protocols, Social Commitments, CASA, Agent-based Systems, Agent Communication Languages.

1. INTRODUCTION

We have been working on an infrastructure for agent-based system that would easily support experimentation and development of agent-based systems. We implemented the first-cut system, and began work on the analysis of what we could learn from our experience.

We had gained enough knowledge to formalize our theory of agent conversations based on social commitments [5], but the CASA infrastructure itself, although seemingly quite adequate, seemed rather ad-hoc, and lacking in solid theoretical foundation.

The various types of service agents in the CASA system each sent and received what seemed to be a set of ad-hoc messages, which did the job, but still seemed unsatisfying in that there seemed to be no apparent pattern to them. We had more-or-less followed the FIPA [8] performative model (the CASA message structure closely follows the FIPA structure, which, in turn borrows heavily from KQML [1]).

We decided to take a hard look at the system with an eye to looking for patterns to modify our design to a more satisfying structure. This paper describes some of our analysis that lead to the current design.

In the remainder of this section we give a brief introduction to the CASA infrastructure. In Section 2, we describe various problems and solutions (a.k.a. design decisions) we have encountered in our building and using the CASA system. These include the arranging of performatives in a type lattice, specification of some relational constraints, and the resulting compositional properties of some messages, the addition of several new fields in messages, and the removal of some performatives into a separate type lattice called *acts*. In Section 3 focus on conversations, rather than messages, and describe how CASA can support either conversation protocols, or social commitments as a basis for inter-agent communication, and weigh some of the merits of the two different approaches. In sections 4 and 5 we discuss our future work and conclusions.

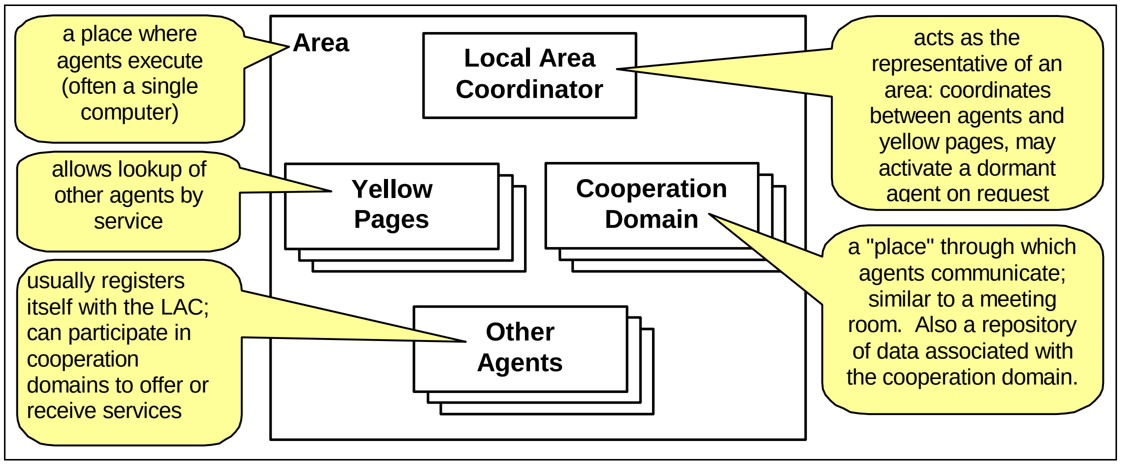

a place where agents execute (often a single computer)
Area
Local Area Coordinator
acts as the representative of an area: coordinates between agents and yellow pages, may activate a dormant agent on request
allows lookup of other agents by service
Yellow Pages
Cooperation Domain
a "place" through which agents communicate; similar to a meeting room. Also a repository of data associated with the cooperation domain.
usually registers itself with the LAC; can participate in cooperation domains to offer or receive services
Other Agents

Figure 1: Basic Communication and Cooperation Services in CASA

1.1 BACKGROUND: CASA

The fundamental objective of CASA is to support communication among agents and related services. Therefore, CASA offers a generic communication and cooperation infrastructure for message passing, archiving (data, knowledge bases, and transaction histories), agent lookup, and remote agent access. It is an open system infrastructure that may be easily extended as the future need for services becomes apparent. CASA makes no demands on *intra*-agent architecture (internal agent architecture), however, agent templates are provided in the form of classes that one can inherit from and specialize (in Java and C++). Currently, generic class-templates are provided and conversation-protocol and social-commitment specialized class-templates are under development.

As shown in Figure 1, *Cooperation Domains* (CDs) acts as a central "hub" for multi-agent conversations such that all participants may send messages directly to the Cooperation Domain for point-to-point, multi-cast, type-cast[1], and broadcast communication within the Cooperation Domain (a group of agents working together on some task). Agents within a cooperation domain may also use the cooperation domain to store persistent data that will permanently be associated with the conversation, giving the conversation a lifetime beyond the transient participation of the agents, as is often required. Cooperation Domains may also store transaction histories for future playback of the chronological development of the conversation artifacts. Cooperation Domains may perform all these tasks because all messages use a standard "envelope" format (currently either KQML [1] or XML [15])[2], which flexibly provides a basic semantic "wrapper" on messages that may be otherwise domain specific: both the utility of generic services and the efficiency of domain specific languages are therefore provided.

Yellow page servers (also called Yellow Page Agents, Yellow Pages, or YPs) allow agents to look up other agents by the services they offer. An *area* is defined nominally as a single computer (but could be a cluster of several computers, or a partition on a single computer). There is exactly one *local area coordinator* (LAC) per area, which is responsible for local coordination and tasks such as "waking up" a local agent on behalf of a remote agent. All application agents reside in one or more areas. See Figure 2 for a screen dump of a LAC interface showing several agents running.

Messages are needed to support interactions among agents. Messages are generically defined either as *Request*, *Reply* or *Inform* messages; where *Requests* are used to ask for the provision of a service, *Replies* to answer requests, and *Informs* to notify agents without requiring the receiving agent to perform any action. Since these messages are too ambiguous for the definition of interaction protocols, other, more meaningful, message subtypes are derived from these general definitions of messages. For example, *Refuse* is derived from *Reply*; an agent can *Refuse* a earlier *Request* of another agent as a special type of *Reply* (as *Agree* would be another special type of reply).

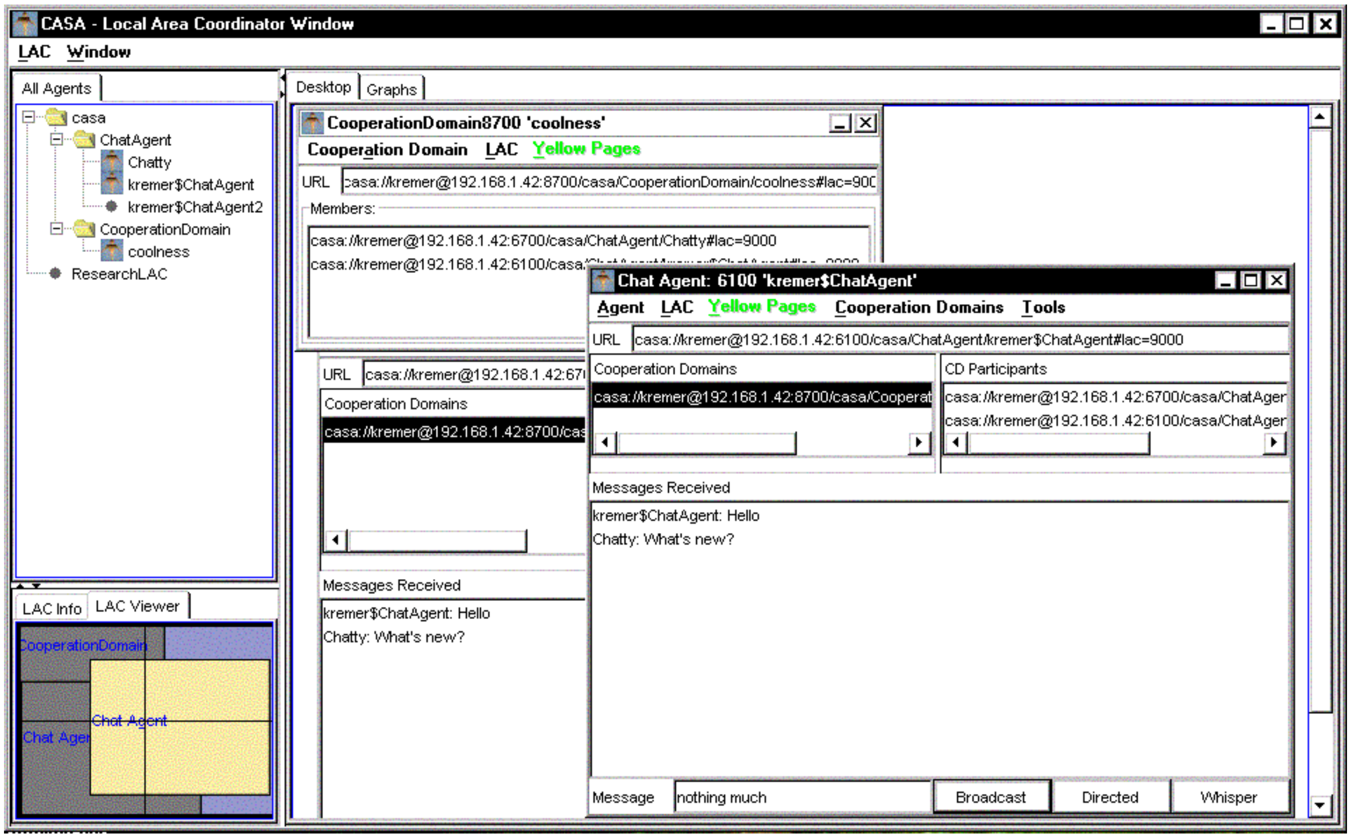
2. MESSAGES

In our analysis of the system we first looked at the performatives in the messages, and existing FIPA performatives. These include:

Accept Proposal	*Agree*	*Cancel*
Call for Proposal	*Confirm*	*Disconfirm*
Failure	*Inform*	*Inform If*
Inform Ref	*Not Understood*	*Propagate*
Propose	*Proxy*	*Query If*
Query Ref	*Refuse*	*Reject Proposal*
Request	*Request When*	*Request Whenever*
Subscribe		

[1] Type-cast is similar to multi-cast, but the destination specification describes agent types (rather than individuals), and the cooperation domain forwards the message to agents known to conform to the type(s).

[2] Actually, the envelope formats can be either KQML or XML (mixed) as we can dynamically switch between the two owing to our employing the Chain of Command pattern [10] to interpret messages.



URL casa://kremer@192.168.1.42:8700/casa/CooperationDomain/coolness#lac=900

casa://kremer@192.168.1.42:6700/casa/ChatAgent/Chatty#lac=9000
casa://kremer@192.168.1.42:6100/casa/

URL casa://kremer@192.168.1.42:67

casa://kremer@192.168.1.42:8700/cas

kremer$ChatAgent: Hello
Chatty: What's new?

URL casa://kremer@192.168.1.42:6100/casa/ChatAgent/kremer$ChatAgent#lac=9000

casa://kremer@192.168.1.42:8700/casa/Cooperat

casa://kremer@192.168.1.42:6700/casa/ChatAger
casa://kremer@192.168.1.42:6100/casa/ChatAger

kremer$ChatAgent: Hello
Chatty: What's new?

Message: nothing much

Figure 2: A screen dump of a CASA LAC interface with a Cooperation Domain and two simple 'chat' agents communicating with one another. All three agents are running in the same process here, but they may all run in independent processes or on different machines.

2.1 The Problem

The FIPA performatives seem rather awkward when one tries to use them in a real application such as this. The meaning of each and its relationship to the others is not always clear. In some cases, it seems they have overlapping meaning. For example, *Refuse* and *Reject Proposal* can occur under the same circumstances, and the former would seem to be a subtype of the latter. Thus, it they should be arranged in a type lattice. Furthermore, our use of them (in developing the CASA infrastructure) seemed to indicate that *Inform*, *Request*, and *Reply* were somehow much more fundamental than the others.

A common technique used in communication protocols (especially in the face of unreliable communication channels) is to always have the receiver return a confirmation (an *acknowledge* message) to the sender so that the sender can verify that the message was received (and resend if necessary). Such an *Ack* message is conspicuously missing from the FIPA performatives, so we chose to extend FIPA's protocol[3], and go with an *Ack* (acknowledgement) to an *Inform* to allow us to check on transmission of a message in the event we are using an unreliable communication channel. (Note that we do not *require* agents to use the *Ack* protocol, but agents may agree to use it if so desired.[4])

When we tried to draw a type hierarchy of all the messages required in CASA, we found it looked something like this[5]:

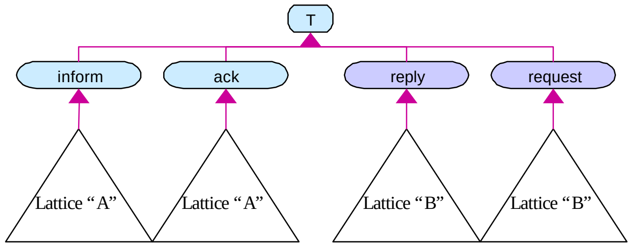


Figure 3

[3] In fact, we're not sure if we really are extending FIPA's protocol, since we could find no protocol for a simple *Inform*. So we aren't sure if a FIPA agent would expect an *Ack* after an *Inform* or not.

[4] Actually, an agent who *isn't* using the *Ack* protocol *will Ack* if it receives a message marked with a request-ack field – see Figure 13 and Section 2.7

[5] This type hierarchy is a simplification of the hierarchy we built containing all the performatives in the FIPA Communicative Act Repository Specification (FIPA00037) [7], which ended up being fairly complex. Although simplified, the above diagram captures the essence of the original.

For example, when an agent wishes to join a cooperation domain, it sends a *joinCD* (lattice B) request to the cooperation domain, and the cooperation domain responds with a *joinCD* (lattice B) reply.

2.2 Rearranging Performatives

It seems interesting that for each specific *Request*, we have a mirrored *Reply* and, similarly, for each specific *Inform* we have a mirrored *Ack*. It seems to us that *Request*/*Reply* and *Inform*/*Ack* are more fundamental than their subtypes. It appears we should keep *Request*/*Reply* and *Inform*/*Ack* as "fundamental message types", and then put all the other things in a separate type lattice of "ACTs". We extend our message headers to include a new field *act*. This more-or-less turns *performatives* into speech acts, and *acts* into physical acts.

Thus, our message header looks like this:

```
performative: request
act:          inviteToJoinCD
to:           Bob
from:         Alice
receiver:     CDagent1
sender:       Alice
...
```

Getting back to acknowledging protocols, we can write down a very simple conversation protocol: that of an *Inform*/*Ack* pair (here we use diamond-shaped arrowheads to indicate sequence):

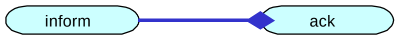


Figure 4

This adds a certain sense of reliability to the message transmission: Alice will always know (by the *Ack*) that Bob received her *Inform*[6]. In the case of a *Request/Reply*, if Bob received a *Request* from Alice, and *Replies*, then Alice will know that Bob received the *Request*, so there is no need for an *Ack* in this case. But Bob (the receiver of the request) has no way of knowing if Alice received the *Reply*. It would be appropriate for Alice to send an *Ack* to Bob:

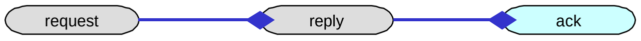


Figure 5

If this is the case, then *Reply* looks a lot like inform. It's a specialization (subtype) of *Inform*:

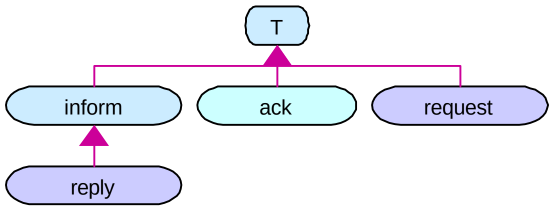


Figure 6

This makes sense: after all, if I'm *replying* to you, I'm *informing* you of what I'm doing with your request.

[6] But Alice won't know if Bob *didn't* receive the *Inform*, since the *Ack* could have been lost. This is a well known problem in the literature.

Furthermore, it would be consistent to think of a *Reply* as being a kind of *Ack* to a *Request*. After all, if we reply to a request, the reply certainly carries all the functionality of an *Ack*. Therefore, *Reply* would be a subtype of both *Inform* and *Ack*:

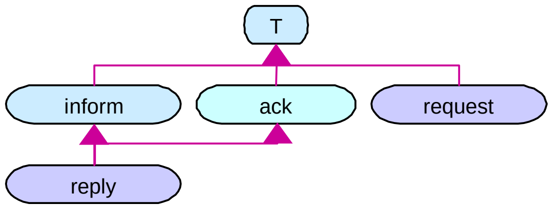


Figure 7

By the same reasoning as a reply is an inform, we can derive that a request must also be in an form. After all, if I'm *requesting* something of you, I'm *informing* you that I want you to do something for me:

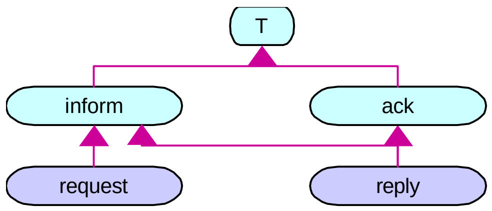


Figure 8

This seems to be a rather satisfying configuration, although slightly asymmetrical.

2.3 Adding Constraints

We will overload the last diagram by adding a sequence relation to indicate that an *Inform* is followed by an *Ack*:

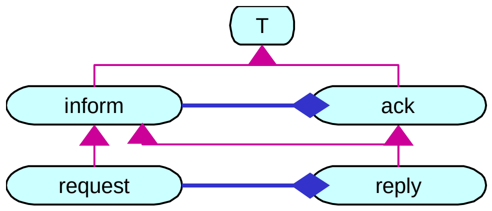


Figure 9

One can observe that an *Inform* must be followed by an *Ack*, and a *Request* must be followed by a *Reply* (and not an *Ack*). This does not violate the *Inform*/*Ack* constraint since *Reply* is a subtype of *Ack* anyway. In addition, the above diagram says that one can't *Ack* a *Request*, because it's not a *Reply*[7]. Just what we want.

2.4 Composing Messages

All this has interesting consequences in implementation. Nothing in the CASA message exchange changes, but it does tell us that a *request* protocol is a composition of two *inform*

[7] One might argue that a request might be followed by *both* an *Ack* and a *Request*, and conform to this diagram, but this cannot be because in a proper conversation thread, speech acts are well ordered: one message cannot be followed by more than one other message.

protocols (where the [overloaded] middle message is a specialized *Ack*):

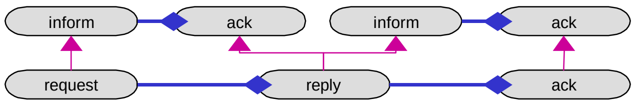


Figure 10

This composition is legal only because we had previously observed that a *Reply* is a subtype of both *Ack* and *Inform*. The fact that we can do this composition lends support to our earlier analysis that *Request* is a subtype of *Inform* and *Reply* is a subtype of both *Inform* and *Ack*.

2.5 When things go wrong

All this is fine when the world is unfolding as it should. But sometimes things don't go as planned and errors happen. An agent may want to negatively acknowledge (*Nack*) an inform ("I don't believe you", "I don't understand the message"). Or an agent may be unable (or unwilling) to perform a request ("Sorry, can't do it"). Or a message may simply not be returned for whatever reason.

Since these forms of failure can be regarded as being speech acts (as opposed to physical acts), we can simply add them as subtypes of *replies* (which also makes them subtypes of *acks* as well):

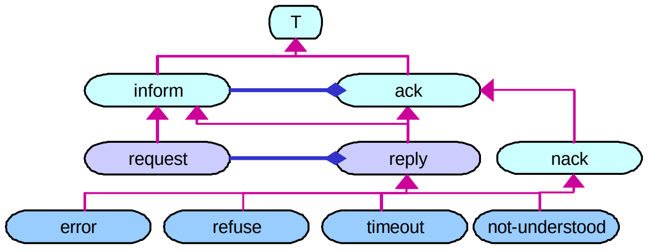


Figure 11

Thus, conversation specifications (protocols), don't need to worry about error conditions: since an error is just a specialization of a *reply* or an *ack*, it is a "normal" part of a regular conversation, at least at the speech-act level. Agents may and will alter their behaviour when errors occur, but there is no reason to unduly complicate a protocol description if the circumstances do not warrant it.

The errors defined in Figure 11 are:

? *error*, which is general catch-all for some exceptional error that may occur.

? *refuse*, which is used when an agent explicitly refuses a request (or possibly to deny an inform, similar to the FIPA *disconfirm*).

? *not-understood*, which is used to inform the sending agent that the receiving agent cannot interpret the message.

? *timeout*, which is a special case explained below.

There are several ways to handle the case where an acknowledge to an inform or a reply to a request is not returned. We find it very convenient to make use of a timeout message, which, in fact, are not ordinary messages in that they are not normally sent between agents. But rather, agents "send" timeout messages to themselves as a simple "bookkeeping" technique: Whenever a message is sent it includes a timeout field which specifies by when an acknowledge/reply is expected. If no acknowledge/reply is received by the time the timeout expires, the agent merely "sends" a matching timeout message to itself. This technique greatly simplifies agent logic because agents are always guaranteed to receive a reply (even if it may only be a timeout). In addition, since overworked receiving agents also are aware of the timeout encoded in the message header, they may safely ignore expired messages without further processing.

2.6 Acts

Now we turn our attention the type lattice of *act*s we had mentioned in Section 2.2. We have created a separate type lattice of *acts* that is somewhat subservient to the smaller type lattice of performatives. We attempted this type lattice for the fundamental messages necessary to manage the CASA infrastructure (see Figure 12)[8].

The figure shows remarkable regularity. There are three fundamental (but not necessarily exclusive) types of acts: *create*, *destroy*, and *get*, corresponding to the three fundamental things one can do with data: write, delete, and read. For each of the agent types, there is a corresponding triplet of messages that represent it's functionality. For example, an agent may request of a YellowPages agent to be advertised (*advertise*), to be removed from the list of advertised agents (*unadvertised*), or to search for some other agent among the YellowPages agent's registered advertisers (*search*). This otherwise perfect symmetry is broken only in one case: An agent may delete a history list (*deleteHistoryCD*) and read the history list (*getHistoryCD*), but it may not write into the history list (the history list is written to as the Cooperation Domain records all the messages it processes).

2.7 A More Powerful Message

As already mentioned, CASA exchanges messages with headers in either KQML formal or XML format. Both of these formats contain exactly the same information, so it matters little (except that XML is more verbose) which is used. The XML DTD in Figure 13 describes our message format.

Most of the fields in the message are either described above or are standard to FIPA [8].

Why did we extend the FIPA message header format? Largely, this is because we are concerned with the observable properties (as opposed to the internal, private properties) of agent interaction. We want an external observer to be able to judge the soundness (conformance to social conventions or protocols) of a conversation without necessarily understanding all of the subtleties and details of the conversation. That is, an external observer should be able "understand" a conversation even though the observer does not understand the language and ontology used in the *content* section of the message. FIPA's message format is a very good foundation for such understanding. For example, the

[8] The initial formulation of the ACT type lattice, or course, was not so clean. The process of building this diagram caused us to tweak our design by renaming a few messages, combining similar functions into single more powerful functions, and adding a few messages that had remained missing.

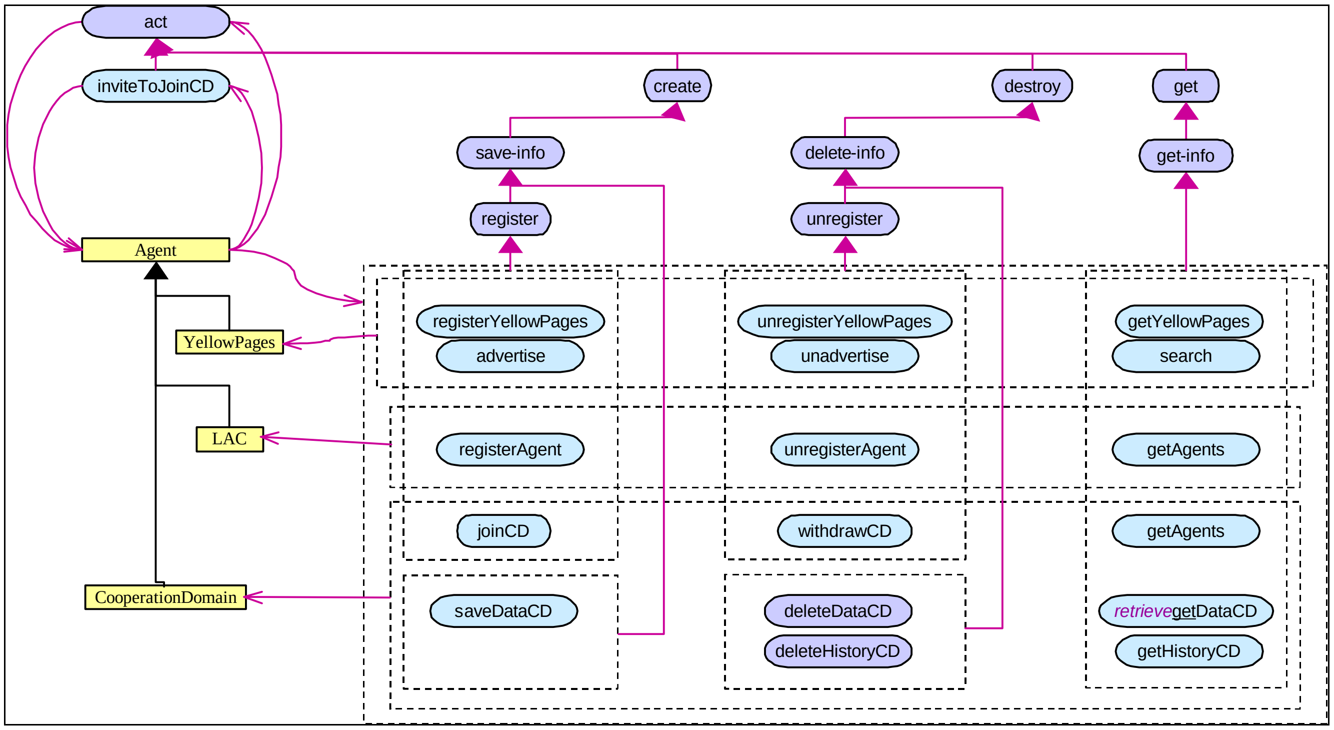


Figure 12: Partial ontology of messages. This diagram describes this ontology in terms of typing (the solid arrows represent the *is-a* relationship), sequence (the V-arrows represent the direction of the message, as in "from-agent ✍ message ✍ to-agent"), and grouping (the messages in the dotted boxes all conform the to relationships impinging on the dotted box). Thus, we may read the *advertise* message as being a subtype of *register*, and being sent by an *AbstractAgent* and received by either a *AbstractYellowPages* agent or an *AbstractLAC* agent.

ideas of the *performative* label to enable an observer to superficially understand the message type; the idea of the separate *sender/receiver* and *to/from* labels to enable an observer

```
<!DOCTYPE CASAmessage [
  <!ELEMENT CASAmessage (version,
   performative, act?, sender, receiver,
   from?, to?, timeout?, reply-with?,
   in-reply-to?, request-ack?, language?,
   language-version?, ontology?,
   ontology-version?, content?)>
  <!ELEMENT version           (#PCDATA)>
  <!ELEMENT performative      (#PCDATA)>
  <!ELEMENT act               (#PCDATA)>
  <!ELEMENT sender            (#PCDATA)>
  <!ELEMENT receiver          (#PCDATA)>
  <!ELEMENT from              (#PCDATA)>
  <!ELEMENT to                (#PCDATA)>
  <!ELEMENT timeout           (#PCDATA)>
  <!ELEMENT reply-with        (#PCDATA)>
  <!ELEMENT in-reply-to       (#PCDATA)>
  <!ELEMENT request-ack       (#PCDATA)>
  <!ELEMENT language          (#PCDATA)>
  <!ELEMENT language-version  (#PCDATA)>
  <!ELEMENT ontology          (#PCDATA)>
  <!ELEMENT ontology-version  (#PCDATA)>
  <!ELEMENT content            ANY     >
]>
```

Figure 13: A CASA message DTD (there are additional elements for thematic roles and semantic modifiers, but these have be omitted in the interests of brevity.

to understand when a message is merely be forwarded as opposed to being actually addressed to the recipient agent; the idea of the *language* label to enable the observer to judge whether the observer can understand the *content*; and the *reply-with* and *in-reply-to* labels to enable the observer to disambiguate multiple concurrent conversational threads. However, our experience with strict reliance on observable properties motivated us to put more "semantic" information in the message. These include:

? We have arranged the performatives in a type lattice to enable an external observer, not conversant in the details of some specialized conversational domain, to understand the conversation based on being able to follow a particular foreign performative (or act) up the type lattice until the observer finds a performative (or act) that *is* understood (see section 2.2).

? We have added a separate *act* label, motivated by our observation that the more fundamental performative, such as inform, request, and reply, are qualitatively different from the lower-level "performatives". (see section 2.6). We have therefore taken out some many of FIPA's (and or own) "performatives", and placed them in a separate type lattice under the message label *act*.

? We have added an ontology label, which is intended to extend the FIPA language label, since an observer my understand a certain language (e.g. prolog) but not understand the ontology used (e.g. the prolog predicate library used).

? We have also qualified the message and language and ontology labels with version numbers to support the evolution of these definitions. The version numbers allow observers (and conversation participants) to recognize when they have an older version of the message/language/ontology definition, and may not understand the entire contents, or when they are dealing with a conversation participant who has an older definition, and may be able to adjust their interpretations and utterances to match.

All of these extensions are optional, and allow us to still be compliant with FIPA message, so long as the FIPA agents are willing to ignore foreign labels in incoming messages. Figure 14 shows an example of a CASA request/reply message pair.

These extensions allow an external observer to monitor messages and make sense of them without having the omniscient and gaze into the internal "thought process" of the participating agents. While the original FIPA definition of the agent allows this to some extent, our extensions support further semantic interpretation of messages, and offer pragmatic solutions to some of practical problems that occur in real-life, evolving situations.

All of this is useful, but must be extended to the *composition* of messages, which is described in the next section.

3. CONVERSATIONS

In our work with CASA, we are aiming a high degree of flexibility with respect to the construction of agents. We therefore choose to design in enough flexibility to allow agents that are based on either conversation protocols or on social commitments [4].

3.1 Conversation Protocols

When in conversation protocols mode, CASA conversations are modeled by using conversation protocols (CPs) to define the possible utterances and responses that can be exchanged between agents. Following FIPA standards for interaction protocols [8], we define a variety of patterns in which initiators and participants interact. This model is flexible in that any new interaction or conversation can be added by simply defining the possible messages (such as INFORM/REQUEST), and their counterparts (ACK/REPLY). For example, suppose a very simple interaction was needed to model two agents: one with the role of requesting a service, and another that may or may not agree to provide the service. This can be done by adding the following to each of the agents' list of possible messages:

Agent 1	Agent 2
performative: *Request* act: *service X*	performative: *Reply* act: *service X*
	performative: *Reject* act: *service X*

In addition, the agents' library of reactive protocols in updated (which is consulted when a CP agent receive a message "out of the blue" starting a new conversation), and a pattern is written to handle the particular protocol. In this fashion, more complex interactions can be added.

The internal mechanisms, which determine how an agent will decide what (if any) message to send, are implementation dependant. In fact, we leave such details out since we wish to concentrate on the way in which agents communicate and on observable agent behaviour. We leave the matter of how agents should behave and their internal workings to the individual agent developer. The key is that agents involved in the interaction require the protocol to be pre-defined.

3.2 Social Commitments

The social commitment (SC) model is an alternative to the CP model described above, and provides a mechanism by which agents can negotiate obligations among one another. A social commitment is an obligation that is negotiated between potential debtor and creditor agents (see [5], [4], [11], [6], [2] & [14]). Such negotiation of obligations mimics the behaviour of agents

```
( :request
  :act         register.instance
  :sender      casa://kremer@192.168.1.42:8700/casa/CooperationDomain/coolness
  :receiver    casa://kremer@192.168.1.42:9000
  :timeout     1066456360438
  :reply-with  casa://kremer@192.168.1.42:8700/casa/CooperationDomain/coolness-0
  :language    casa.URLDescriptor
  :content     "casa://kremer@192.168.1.42:8700/casa/CooperationDomain/coolness true"
)

( :reply
  :act         register.instance
  :sender      casa://kremer@192.168.1.42:9000/casa/LAC/ResearchLAC
  :receiver    casa://kremer@192.168.1.42:8700/casa/CooperationDomain/coolness
  :timeout     1066456360438
  :reply-with  casa://kremer@192.168.1.42:9000/casa/LAC/ResearchLAC-1
  :in-reply-to casa://kremer@192.168.1.42:8700/casa/CooperationDomain/coolness-0
  :language    casa.StatusURLandFile
  :content     "( 0 \"Success\" casa://kremer@192.168.1.42:8700/casa/CooperationDomain/coolness#lac=9000
               \"/casaNew/root/casa/CooperationDomain/coolness.casa\" )"
)
```

Figure 14: An example of a CASA request/reply message pair using the KQML message format. In this case, a Cooperation Domain is registering with a LAC (to let the LAC know it is running). The LAC responds with a success status together with the Cooperation Domain's fully-qualified URL and the file it should use to use to store any persistent data. (Backslashes are used as escape characters in the content part.)

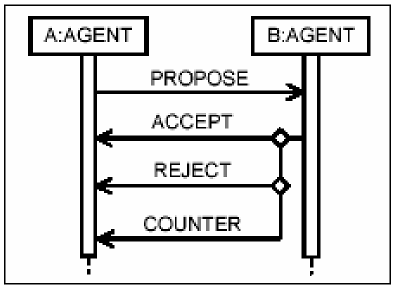


Figure 15: A simple proposal

using CPs, but in more flexible manner. For example, suppose an agent (agent A) were to issue a proposal to another agent (agent B). Figure 15 shows the sequence of events and the possibilities agent B can reply with. If the *debtor* agent (agent B) accepts the proposal by responding with an *accept* reply to agent A, it adopts the obligation to provide the *creditor* agent (agent A) with some service. It should also be noted that the commitment is not imposed on the debtor agent; it is negotiated. A more complex example might allow for agent B to refuse the proposal and make a counterproposal which would, in turn, be evaluated by agent A.

The possible replies that might be issued by the receiving agent include acceptance, rejection or counterproposal, and the sending agent may at any time issue a withdrawal its own proposal.

The value of using the social commitments model is twofold. First, it allows for linking the external world (including interactions with other agents) with the internal agent mechanisms. An agent maintains a list of the current and past obligations both owed to other agents and to itself. This information could be used to "rationally" influence the agent's future actions. Thus, a conversation is never rigidly "scripted" as in conversation protocols, but flows naturally from an agent disposing its current obligations. Second, it provides a degree of autonomy (through the negotiation process) for adopting commitments. An agent is free to accept or reject proposals as described above.

3.3 Observer Agent

One aspect that enhances the flexibility of the CASA model is that agents can be used in various capacities other than being directly involved in conversations. A sufficiently privileged agent may join a cooperation domain and observe the communication between other agents. This agent, which may simply be a "plug-in agent" for the cooperation domain, could perform various functions. It could perform the role of a conversation manager, tracking and organizing the messages sent between agents; it could perform the role of janitor, cleaning up obsolete conversations and connections; or it could perform the role of a translator, resending messages broadcast in one language in another language. An agent may also observe conversations between other agents in order to gather information about how it can best proceed toward its goals. Note that, it is owing to the arrangement of the performatives and acts into a type lattice that there is sufficient information in the message "envelope" to allow an "outside" agent to "understand" a conversation even if it does not understand the domain-specific performatives, acts, content language, and ontology of the participating agents.

In the context of the social commitments model, it might be useful to have a judging or policing agent observing interactions between agents, checking on their conformance to social norms, such as whether or not agents discharge their obligations within an acceptable period of time (or at all). This is only possible because the SC model relies only on *observable* behaviour.

The above examples are meant to illustrate the flexibility of CASA, in that it can be extended in many ways other than adding additional conversation protocols. It supports future developments in the areas of rationality and social knowledge as they relate to conversations and interactions among agents in a society.

4. FUTURE WORK

Although the work in CASA and social commitment theory is progressing well, there is still much to do. We have several projects underway in various related research fields. We are working on further extending the CASA message "envelope" to include more formal reference objects and relationships based on theories from linguistics (e.g. [12], [13]). For example, we can add fields such as *agent* (initiator of an act), *patient* (the entity undergoing the effects of an act), *theme* (the entity that is changed by the act), *experiencer* (the entity that is a aware, but not in control of, the act), and *beneficiary* (the entity for who's benefit the act is preformed.

The detailed semantics of messages, as described here, has not yet been worked out. Some of our future work involves the formal specification of message semantics, at least as it relates to the services offered within CASA. Some formal specification of the social commitment model has been done (see [4] & [5]), but the formal specification of CASA services has hardly been touched so far.

Furthermore there a several pragmatic areas that need to be worked out if CASA, or a CASA-like system is to be used in real settings (such as what we are working for manufacturing systems). For example, security will be an issue, and we are currently undertaking research to determine an appropriate approach to protecting stored data, messages themselves, and cooperation membership in such a dynamic and distributed environment.

5. CONCLUSIONS

In this paper, we have described some our design decisions in the CASA infrastructure primarily with respect to the structure of inter-agent messages. Most of the discussion contrasts our structure with that of the FIPA model. We differ from FIPA's viewpoint primarily in that we concern ourselves more with the *observable* behaviour of agents, and the ability of an outside observer to understand the meaning of a conversation. Minimally, we would like the outside observer to be able to deduce if the agents are participating in logical conversation and conforming to a set of social norms.

To this end, we have conformed to the FIPA message structure with a few changes (primarily ordering the performatives in a type lattice), and extending it in several ways. We have extended the message structure by adding the fields such as *act*, and *ontology*. In addition, we have added version numbers as attributes to the message language itself, and to the language and ontology fields.

We also briefly discussed conversations, which are composed of messages, and discussed CASA's support for agents based on either conversational protocols or social commitments.

6. ACKNOWLEDGMENTS

We are grateful to the Natural Sciences and Engineering Council of Canada (NSERC), Smart Technologies, Inc., the Alberta Software Engineering Research Consortium (ASERC) and the Alberta Informatics Circle of Research Excellence (iCORE) for their financial support throughout the realization of this work.

7. REFERENCES

[1] Becerra, G. "A Security Pattern for Multi-Agent Systems". In Proceedings of Agent Based Technologies and Systems, Far, B.H., Rochefort, S. and Moussavi, M. (eds.), Calgary, Canada, August, 2003, pp 142-153.

[2] Chopra, A. and Singh, M.P. "Nonmonotonic Commitment Machines". In Dignum, F. (ed.), Advances in Agent Communication, LNAI, Springer Verlag. In this volume.

[3] Finin, T. and Labrou, Y. "KQML as an Agent Communication Language". In Bradshaw, J.M. (ed.), Software Agents, MIT Press, Cambridge, 1997. pp. 291-316.

[4] Flores, R.A. "Modelling Agent Conversations for Action". Ph.D. Thesis, Department of Computer Science, University of Calgary, Canada. 2002.

[5] Flores, R.A. and Kremer, R.C. "To Commit or Not To Commit: Modelling Agent Conversations for Action". In B. Chaib-draa and F. Dignum (eds.), Computational Intelligence, Special Issue on Agent Communication Languages, Blackwell Publishing, 18:2, May 2002, pp. 120-173.

[6] Fornara, N. and Colombetti, M. "Protocol Specification using a Commitment-based ACL". In Dignum, F. (ed.), Advances in Agent Communication, LNAI, Springer Verlag. In this volume.

[7] Foundation for Intelligent Physical Agents (FIPA). FIPA Communicative Act Repository Specification. Foundation for Intelligent Physical Agents. Oct. 18, 2002. http://www.fipa.org/specs/fipa00037/

[8] Foundation for Intelligent Physical Agents (FIPA). FIPA Interaction Protocol Library Specification. Foundation for Intelligent Physical Agents. Aug. 10, 2001. http://www.fipa.org/specs/fipa00025/XC00025E.html#_Toc505480198

[9] Foundation for Intelligent Physical Agents (FIPA). FIPA Specifications, Version 1. 1997.

[10] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley Professional Computing Series, Addison-Wesley, Reading Mass., 1994.

[11] Mallya, A.U., Yolum, P. and Singh, M. "Resolving Commitments Among Autonomous Agents". In Dignum, F. (ed.), Advances in Agent Communication, LNAI, Springer Verlag. In this volume.

[12] Saeed, J.I. "Semantics". Blackwell Publishers Ltd., Oxford, UK, 1997.

[13] Thurgood, G. "English 222: Pedagogical Grammar". Chapter 9. English Department, California State University, Chico. Aug, 2002. http://www.csuchico.edu/~gt18/222/Ch%2009.pdf

[14] Verdicchio, M. and Colombetti, M. "A logical model of social commitment for agent communication". In Dignum, F. (ed.), Advances in Agent Communication, LNAI, Springer Verlag. In this volume.

[15] World Wide Web Consortium (W3C) "Extensible Markup Language (XML)". Architecture Domain, World Wide Web Consortium, Oct. 14, 2002. http://www.w3.org/XML/.