SAND REPORT
SAND2002-1760
Unlimited Release
Printed June 2002

Agent-Based Mediation and Cooperative Information Systems

L. R. Phillips, H. E. Link and S. Y. Goldsmith

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/ordering.htm

Agent-Based Mediation and Cooperative Information Systems

Laurence R. Phillips, Hamilton E. Link, and Steven Y. Goldsmith
Advanced Information and Control Systems Department
Sandia National Laboratories
P. O. Box 5800
Albuquerque, NM 87185-0455

ABSTRACT

This report describes the results of research and development in the area of communication among disparate species of software agents. The two primary elements of the work are the formation of ontologies for use by software agents and the means by which software agents are instructed to carry out complex tasks that require interaction with other agents. This work was grounded in the areas of commercial transport and cybersecurity.

This page intentionally left blank.

CONTENTS

FIGURES

1. Introduction

This project was conceived as research into agent mediation issues, focused on developing a technological "Rosetta Stone" that would allow disparate agent systems to collaborate with one another. We accomplished many of the original goals of the project, although not all in the way first proposed. In the course of the project, we discovered additional important aspects of the problem space and explored a number of these aspects as well.

We began by working with CMU to establish an operational version of their agent architecture in our lab, alongside machines running Sandia's 2nd-generation agent architecture (SAA2). As we made these systems interact with one another, we began to better comprehend the problem space of general-purpose communication mechanisms in agent systems. To demonstrate and expand our understanding of the issues we developed generalized mechanisms for describing and executing complex patterns of interaction. This led to the invention of a number of software components that we integrated with SAA2 in the form of perception and schema-processing frameworks.

Around this time, much of our work began to shift towards applicability in network security. This led to the development of a number of security support protocols, and allowed us to test the generality and capabilities of the communications framework that we had developed by applying the technology in a domain it had not been explicitly designed to cope with. The process gave us the opportunity to identify and improve a number of framework components where the implementation had not fully realized the hypotheses. In the end we succeeded in constructing operational systems of agents executing very complex communication patterns with one another, based on the work we originally began with CMU.

The theoretical results of the LDRD effort are:

1. A more complete understanding of the problem of making disparate agent systems communicate with one another,
2. The design of a general-purpose framework for enabling interaction between agents, and
3. An analysis of communication issues associated with security protocols in agent systems.

These theoretical results were realized in agent-based technology for executing speech acts in KQML and a mature implementation of a general-purpose communications framework that has since been leveraged repeatedly for many different tasks on other projects in our lab.

2. Background and Problem Statement

One of the difficulties in building distributed information systems is enabling disparate components to share meaningful messages with one another. The issue is not so much in constructing a network able to transmit data between the components, but rather in developing a system in which all components can grasp the data's semantic meaning.

> "For an initiator to [understand] a respondent ... their messages must be ... grounded in a shared ontology ... the lack of common definitions is known as the ontology problem, and is the most challenging obstacle to widespread interoperability of heterogeneous, distributed co-operating systems." [1]

This problem must of course arise when components have been designed independently, but in fact it can be an issue even when all components initially shared a semantic model, when modifications and additions are not uniformly made. It has been said that "the main barrier to

electronic commerce lies in the need for applications to meaningfully share information" [2] and it is clear that this barrier exists for other application domains as well.

The thesis of this project is that *agents* are a useful adjunct in addressing issues of intercommunication among disparate processes. An agent is a computational thread of execution (or set of interacting threads) that takes action based on input and its state without waiting for explicit commands. As stated in our initial proposal, "agent-mediated information management is currently the most promising solution to the problem of integrating and accessing large legacy data stores and for utilizing networked information sources such as the Internet." [3] The project was engendered to explore the process of realizing this promise in a concrete setting using the SAA2 agents we developed.

We are not alone in our assessment that agents are the relevant approach. "Army, Navy and Air Force researchers—along with defense contractor Lockheed Martin—have recognized software agents as 'absolutely critical' in solving another long-standing frustration: The inability to share data across the military's myriad computer systems." [4]

But an agent-based approach can only facilitate the design and implementation of such systems. What is required to actually enable two disparate agent communities to meaningfully share information? Greaves *et al.* say it well:

> "The dream of agent interoperability is commonly thought to rest on three main characteristics shared by the interoperating agents:
>
> 1. They would be able to access a set of shared infrastructure services for registration, reliable message delivery, agent naming, and so forth (i.e., there must be *structural* interoperability);
> 2. They would share (possibly through translation) a common content ontology, truth theory, and method of binding objects to variables (i.e., there must be *logical* interoperability); and
> 3. They would agree on the syntax and semantics of a common agent communication language (ACL) in which to express themselves (i.e., there must be *language* interoperability)." [4]

In human systems, language interoperability is largely taken for granted; logical interoperability is achieved through training, experience, and convention; and structural interoperability is engineered as necessary—telephones, e-mail, radios—to extend our natural human abilities. In most agent-based systems, language interoperability is achieved by selection of a standard common language, such as Knowledge Query and Manipulation Language (KQML), the Foundation for Intelligent Physical Agents (FIPA) ACL, or DARPA's Agent Markup Language (DAML). Existing media for transmission of information—e.g., the Internet—readily provide structural interoperability. This leaves as the primary issue the realization of a means to achieve logical interoperability: Given that agents can communicate with one another, what do they say, and what must be done to enable the receiving agent to understand the transmitting agent?

For the applications we considered in the context of this research, the Internet provides structural interoperability, and we used KQML to provide language interoperability. Our primary conceptual challenge was therefore to formulate a representation of the knowledge that would enable non-SAA agents to understand SAA agents; that is, to achieve logical interoperability with agents that we did not design.

3. Approach

Our approach was to design and build into our Standard Agent Architecture (SAA) agents a means to share information with another community of agents that had been designed independently from our own. We selected the Reusable Environment for Task Structured Intelligent Network Agents (RETSINA) at Carnegie Mellon University (CMU) as our target community. RETSINA offered several features that supported our goals:

- RETSINA agents communicate using KQML syntax.
- CMU had developed a RETSINA-based Matchmaker system for advertising and finding services that dovetailed nicely with our work on cross-border shipping;
- CMU had developed the Language for Advertisement and Request for Knowledge Sharing (LARKS) with which to construct postings for their Matchmaker; and
- We could readily communicate with operational CMU agent communities via the Internet.

With these features and concepts in mind, we began the following work plan:

1. Develop a mechanism to accept objects in our internal format and emit messages in KQML syntax.
2. Develop a means to express the services our agents would provide as trans-border documentation experts and facilitators in LARKS.
3. Advertise our services with the CMU Matchmaker.
4. Receive and respond to KQML-framed requests for our services.

When we began to carry out domain-specific interactions with CMU's agents (item 3 in the work plan), it became apparent that the high granularity of the procedural language with which we programmed our agents was going to limit the complexity of behavior we could implement, especially when several agents were involved.

We needed a declarative language in which we could write agent behavior descriptions that the agents would execute. This implied not only a language that supported the operations the agents were to perform but also a canonical execution mechanism in each agent so that any SAA agent receiving such a description could execute it.

It was apparent to us that we needed this extension no matter what further operations we decided to pursue with our agent technology. But another change overshadowed all decisions: Because of changes in our business direction and the outcome of our work on other projects, our focus began to shift to security. For us, with our primary focus on agents, this devolved to answering three questions: (1) With what aspects of "security" might agents be concerned? (2) How can an agent protect itself from cyberattack? (3) How can an agent or group of agents protect other cyber resources from cyberattack?

This change in direction of any subsequent real-world work made the move to a declarative-language execution mechanism especially compelling, because many security operations are very complex.

Based on these conditions, instead of completing step 4, we began the following work plan:

4a. Develop and implement a declarative framework for specifying the actions of agents.

4b. Develop and implement an execution engine that can execute action thus specified

5. Develop and implement ontological representations for security elements

6. Develop and implement security operations using the results of 4a, 4b, and 5.

4. Focus on Ontologies and Their Representation

The concept *ontology* appears above as a necessary element in applications that must share meaning among disparate components. *Ontology* has a particular meaning when used in an information technology context that differs from its use elsewhere. In addition, the representation of ontological information takes a special form in our environment.

Definitions of *ontology* from the literature:

I. "An ontology is a description (like a formal specification of a program) of the concepts and relationships that can exist for an agent or a community of agents." [6]

II. "1. <Philosophy> A systematic account of Existence.

2. <Artificial intelligence> (From philosophy) An explicit formal specification of how to represent the objects, concepts and other entities that are assumed to exist in some area of interest and the relationships that hold among them ... A set of agents that share the same ontology will be able to communicate about a domain ...

3. <Information science> The hierarchical structuring of knowledge about things by subcategorizing them according to their essential (or at least relevant and/or cognitive) qualities." [7]

We include the first definition because of its breadth and use of vernacular English. This definition is important because it exposes the important notion that if a concept isn't represented in the agent's ontology, as far as the agent is concerned, it cannot exist; and, conversely, the definition of what can exist for an agent, in whatever form, is the agent's ontology.

The second definition distinguishes the meaning of *ontology* as used by different disciplines. Definition I corresponds to element 2 of definition II. The direct statement "agents that share the same ontology will be able to communicate" indicates that our fundamental conceptual work—sharing an ontology—must be accomplished for heterogeneous agents to share meaning. Element 3 of definition II is noteworthy because SAA2 agents use a hierarchical class-subclass-instance network both intensively (to describe things that may or may not exist, such as electronic messages it is able to create) and extensively (to categorize things that it discovers, such as messages it receives). The agent's ontology-bearing structure, in other words, is a class-subclass-instance network defined at compile time and supplanted during its lifetime.

Ultimately, no matter how our ontology was expressed internally, we needed to delimit a section of it for export to the CMU agents and convert or transform it into a form they could understand. The approach we decided to take can be summarized:

1. Develop a representation of the activity to be advertised on the CMU Matchmaker (we already had ontologies describing every aspect of the cross-border shipping problem. Specifying an activity for the Matchmaker meant expressing it as a state change in some set of information).
2. Determine how to express that activity in LARKS terms
3. Manually build a document in LARKS
4. Embed the LARKS activity description in a KQML message and send it to the Matchmaker

Some of the issues inherent in this process are explored in [8]{reprinted in Appendix II).

5. The Ontology of the Border Trade Facilitation System (BTFS)

The BTFS [9] was developed prior to this research, and the nature of electronically facilitated business transactions was a natural point of departure for this project. As a result some time was spent constructing more formal ontologies for the ecommerce domain to provide CMU's agent system and ours with a domain of concern in which to perform experiments (see example). The purpose of BTFS was to maintain an accurate online representation of the current state of the world, in particular the state of transported goods moving through the US and Mexico. In addition to tracking the physical locations of these items, BTFS maintained the information necessary to automate the paperwork used by customs agencies, manufacturers, and trucking companies. The notion that part of the state of the world is purely informational, such as the fact that a transaction is authorized once a document has been signed, is not uncommon in agent environments.

```
US-MANUFACTURER = (and MANUFACTURER
                       CATEGORY-INSTANCE
                       )

MANUFACTURER = (and COMMERCIAL-ORGANIZATION
                    CATEGORY-INSTANCE
                    (all has-ProductCategories)
                    (all has-ParentCompany)
                    )

COMMERCIAL-ORGANIZATION = (and ORGANIZATION
                               CATEGORY-INSTANCE
                               (all has-DunsNumber)
                               )

ORGANIZATION = (and EXPORT-AS-REFERENCE
                    ADDRESSED-OBJECT
                    PHONE-NUMBERED-OBJECT
                    FAX-NUMBERED-OBJECT
                    (all has-EcaPublicProxy)
                    )

EXPORT-AS-REFERENCE = (and PROXY-REFERENCE
                           )

PROXY-REFERENCE = (and OBJECT-BASE
                       PROXY-MIXIN
                       REMOTABLE-OBJECT-MIXIN
```

Figure 1. BTFS Ontology fragment in LARKS format

In order to experiment with the BTFS domain, the world was divided into services that would typically be provided by an agent. In addition to each service's specific ontology of discourse, all services were described in a service-description ontology. This description was intended for submission by the service provider to CMU's "Matchmaker" agent, which when coupled with their Agent Nameserver (ANS) acted as a Yellow Pages, of sorts. Agents desiring a particular service would describe the service to a Matchmaker agent, and would be referred to an appropriate service provider. The requestor would then make contact with the provider and they would interact using that service's ontology.

In order to interact with CMU's Matchmaker and ANS agents, it was most expedient to enable our agents to speak KQML, an agent communication language. This was straightforward, and was easier than we believed it would be to enable their systems to comprehend and manipulate the more sophisticated (and complex) distributed object representation used by our agents when communicating among themselves.

6. Specifying Patterns of Interaction and Processing Schemata

As we approached the point of carrying out domain-specific interactions with CMU's agents, we became aware that a more highly structured description language was necessary to improve the process by which we designed and implemented agent behaviors. Key elements of the design problem were the identification and naming of distinct information states in a particular context and whether organizing behaviors around discrete named states would be a practical way to

approach agent interactions. Out of this work came the schema processing mechanism now used in SAA2 for most agent-to-agent interaction. The Schema Processing mechanism has been declared in a Technical Advance entitled "Standard Agent Architecture II," dated 4/18/2002, that has not been assigned an identifying number as this is being written.

The underlying premise of the schema mechanism is that most of the information states in the course of an interaction can be characterized and distinguished from one another. These states are then used as the basis for a state diagram, and the transitions between these states and the operations to perform within each state are built around them. The state diagram is realized in a form we call a *schema* (pl. schemata). The schema mechanism consists of an "engine" that executes the schemata and an expectation maintenance system that allows the agents to describe and subsequently quickly select relevant stimuli (and reject irrelevant ones) depending on the current states of the schemata being executed by the agent.

7. Security Policy and Cryptographic Protocols

The schema processing mechanism began to mature and we proceeded to implement more involved operations using that system. Both as part of our work in exploring increasingly complex interactions and their limitations and as part of our growing work in security systems research, we began implementing cryptographic protocols for multiparty authentication in our schema language. The multiparty protocols have been declared in Technical Advance SC-7177/S-98,790 dated 4/26/2002 and entitled "Implementation of Group Threshold Signature System."

An essential component of the security work was the separation of interaction *specification* and interaction *policy*. The specification of an interaction is a description of what information needs to pass among which entities. The policy of an interaction defines conditions that may or must hold or not hold, independent of the specification. Security policy is the basis by which an observer decides whether an observed interaction is "legal" or not. Issues of delineating and representing policy are relevant to this project's goal of exploring communication among agents; in essence policy is an aspect of communication that affects one's own communication as well as one's responses to the communication of others. These issues are discussed further in [10] (reprinted in Appendix II).

These protocols had many attributes, such as firm requirements of asynchronicity and minimum numbers of agents involved, that made them useful demonstrations of the capabilities of agent interactions. We did not have the opportunity to develop matching capabilities in CMU's agents to intermix agents from our two different systems while testing these protocols, but we believe this would not be substantially more difficult than establishing interoperability in the BTFS domain was.

At the end of this experiment, we can with a fair degree of confidence state that, even without the distributed object system used by SAA2, the required object descriptions could be expressed in other agent communication languages. Given language interoperability, SAA2 agents should be able to complete these protocols with any other agent system that was extended to handle the algorithms and ontologies involved. Furthermore we believe from our experience implementing these and other protocols and procedures that SAA2 agents can be readily extended to handle complex new domains and operations.

8. Associated Work

This work was developed in several stages, and resulted in a number of publications (Appendixes I through IV).

During the development of the BTFS, substantial effort was put into discovering and specifying the existing border trade participants and their processes, in order to accurately reproduce the functionality of these components in the virtual version of that system. In both the agent mediation study and the original BTFS system, BTFS service-providing agents performed these operations and roles. In the context of work on agent mediation and interaction, however, these agents also needed to provide service descriptions to an advertising service such as CMU's Matchmaker agents. Appendix I lists some of these operations and describes some of the roles involved in the border trade process that were transformed into an ontology for use with an advertising service. Figures 2, 3, and 4 present this material as the software represents it.

The development of BTFS represented an application of technology developed in our laboratory for allowing agents to elicit information from humans using HTML over the world wide web (Appendix II, section 1). This allowed information to be brought into the agent in a controlled format and using a simple synchronous process, enabling the agents to maintain simple representations of ongoing transactions and map input directly into matching structures in the BTFS ontology. The mechanisms developed to enable this were specialized predecessors of the more general mechanisms that were to follow.

At this time we began to identify architectural obstacles in our initial standard agent architecture (SAA) to rapidly developing new and increasingly complex processes for the agents to execute. This led us to consider common features of such processes that could be exploited if appropriate tools were developed. We submitted some of our initial hypotheses to a workshop at Autonomous Agents '99 on conversation policies in agent systems (Appendix II, section 2). As we refined our ideas these concepts ultimately developed into the proposal of this project.

In the course of the project we worked with CMU to enable our agents to interact with theirs, choosing LARKS as a service description language, while continuing to refine and explore more general means of communication between agents (Appendix II, section 3). Selected portions of this software and descriptions of CMU's agent framework are in Appendixes III and IV.

9. Conclusions

Our initial goals of developing an agent "Rosetta Stone" that would allow a wide variety of agent systems to collaborate led us to cast a wide net into the problem of communication. We began by researching agent communication languages and ontologies, developed many protocols and policies, and examined the impact of security requirements in a multi-agent environment. Based on our experience we drew a number of conclusions, and now have new questions that warrant additional research.

As a result of this project our agents can contain complex ontologies, convert them into alternative forms for consumption by different agents, and communicate with other agents using standard message forms. This represents a proof of principle that *independently designed agent systems can be extended to collaborate with one another*. This process can be very difficult in practice, but the difficulty of the task can be mitigated if the design of one of those systems provides a framework upon which language interoperability and a shared ontology can be built.

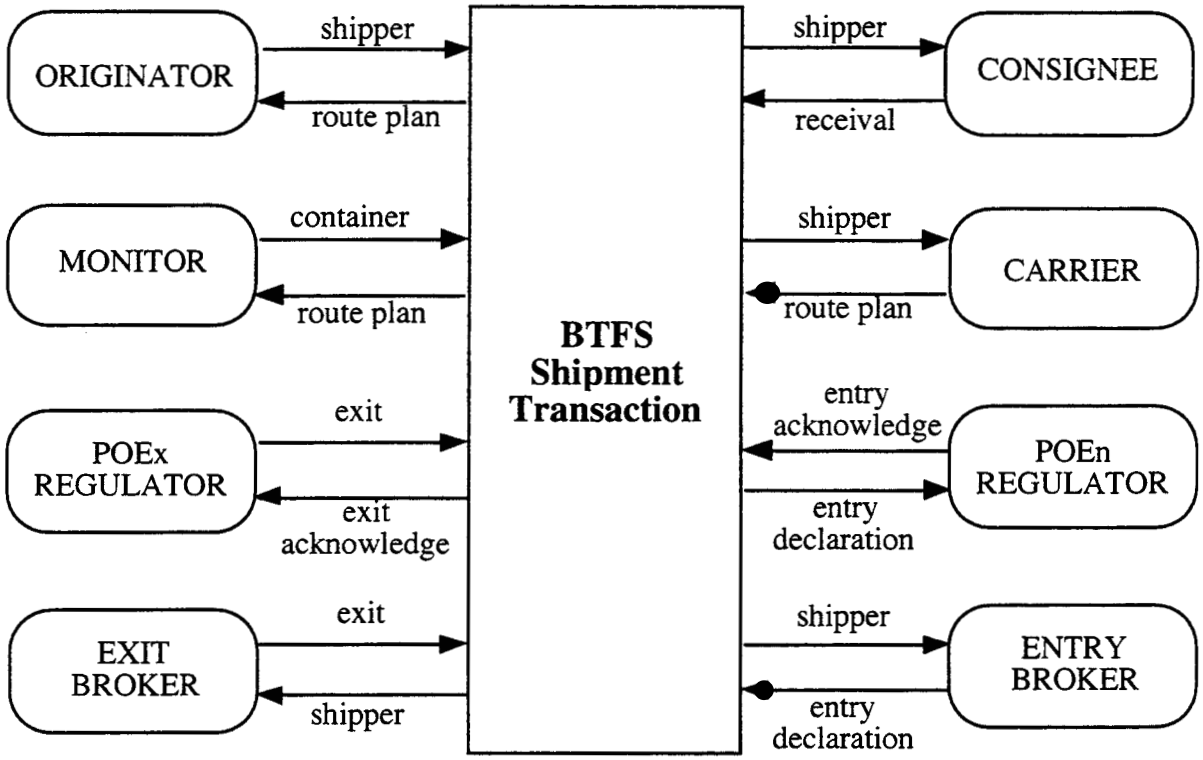
This extension was possible but challenging in the SAA, and our second-generation SAA2 has been designed with this dimension of extensibility. SAA2 can be easily extended to add recognition and processing of new languages dynamically, and is able to be informed online of new ontologies and select for each transaction the appropriate ontology to use when processing information from another agent.

We have a framework mechanism and language for describing complex cooperative tasks that agents can execute. This demonstrated that *common patterns of interaction in agent systems can be exploited using special-purpose process description languages*. In the SAA we were able to build simple interactive processes using conventional object-oriented programming techniques. In order to express more involved conversation procedures that were responsive to dynamically established communication policies, we implemented a more sophisticated state-based mechanism for executing protocols. This allowed us to develop new tools that took advantage of the structure of the communication environment to greatly simplify the programming task.

We were able to exercise these facilities in the context of security operations such as secure key share distribution, and multi-party authorization protocols. Because of the complexity of the security protocols we were now able to implement and come to understand, we realized that *informal approaches to communications security in agent systems are inadequate*. This is a strong statement, made from a point of view gained from working in information surety at a national laboratory: systems developed with security requirements must consider the insider threat model as important and realistic when designing network software.

Data component trading for a shipment transaction

Collaborator actors operate on components of the shared shipment transaction object

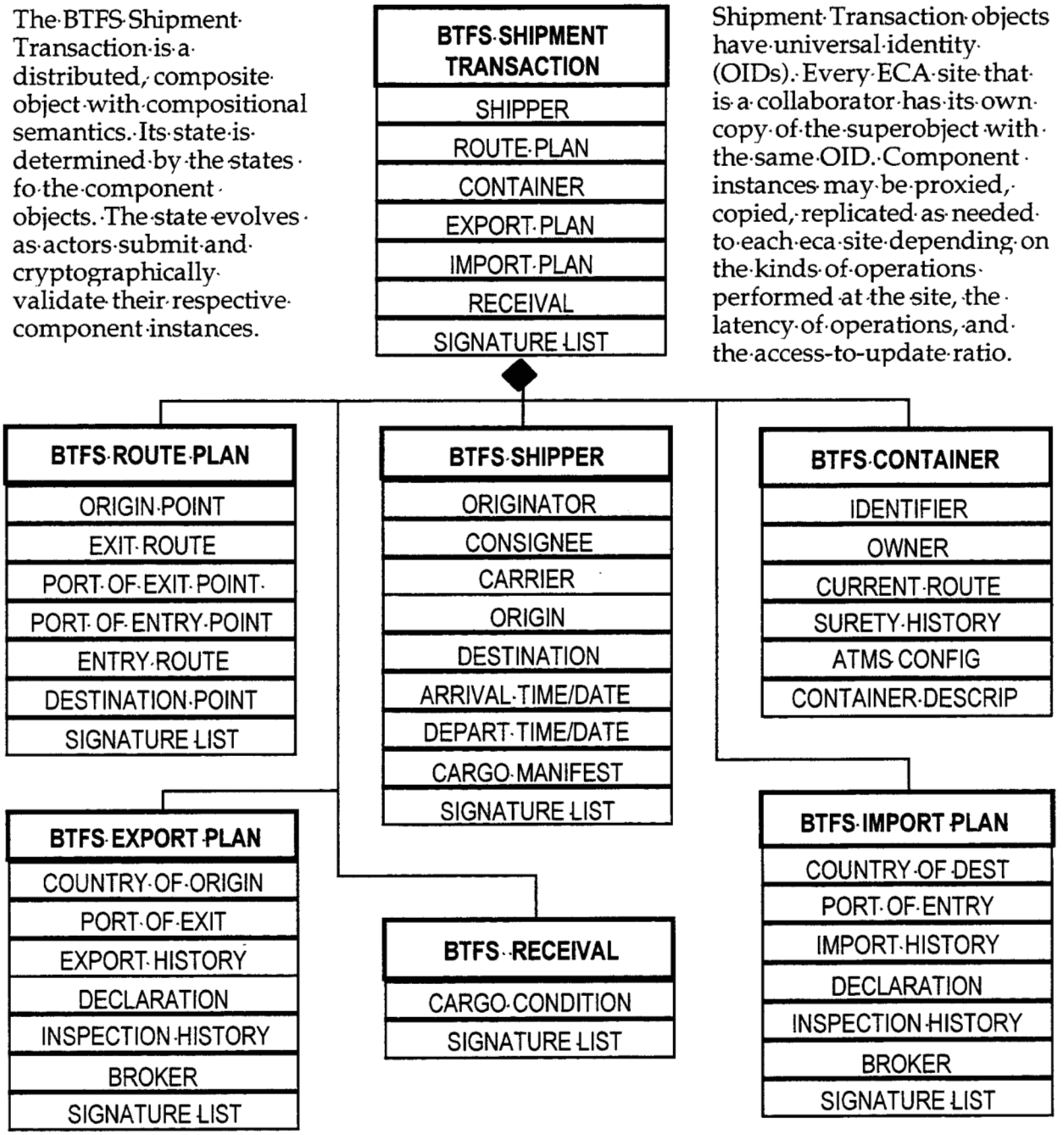


The BTFS Shipment Transaction (ST) is the supremal object (superobject) that is the subject of the collaborative operations. Each collaborator performs its value-added processing on one or more component objects of the ST. Each collaborator modifies the state of the superobject, moving it towards a "goal state."

Figure 2. Data component trading for a shipment transaction

Shipment Transactions & Components Attributes & Associations

The BTFS Shipment Transaction is a distributed, composite object with compositional semantics. Its state is determined by the states fo the component objects. The state evolves as actors submit and cryptographically validate their respective component instances.



Shipment Transaction objects have universal identity (OIDs). Every ECA site that is a collaborator has its own copy of the superobject with the same OID. Component instances may be proxied, copied, replicated as needed to each eca site depending on the kinds of operations performed at the site, the latency of operations, and the access-to-update ratio.

Component classes are defined around collaborator/actor roles. Each collaborator "owns" a piece of the transaction and is responsible for instantiating its piece based on the values of other components. This is appropriate since BTFS transactions exhibit a "strict partition" among operations, knowledge, and data jurisdiction. Actor state-update operations do not overlap, although they share a few access-only "service" operations, such as location and surety reporting.

Figure 3. Shipment Transactions & Components: Attributes and Associations

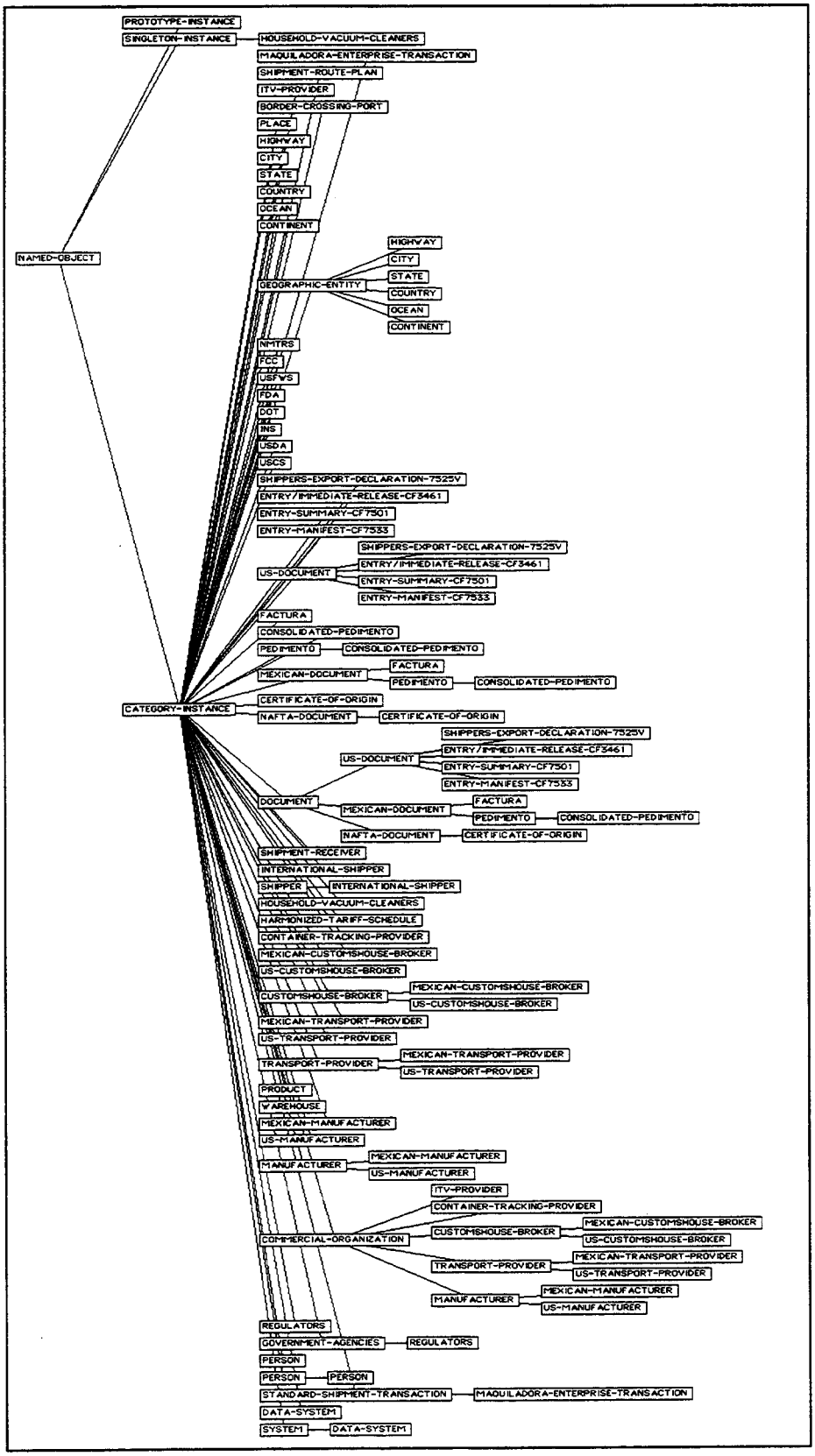


Figure 4. Border Trade Facilitation System Ontology

10. References

1. Collis, J.; Soltysiak, S.; Ndumu, D.: Azarmi, N. *Living with Agents* (http://www.bt.com/bttj/vol18no1/today/papers/j_collis/contents.htm), British Telecommunications plc, 1999.
2. Ontology.Org. *The need for shared ontology* (http://www.ontology.org/main/page1.html)
3. Goldsmith, S. Y. Proposal for LDRD project #10361 *Agent-Based Mediation and Cooperative Information Systems*, 1998
4. Daskiewich, Daniel, CoABS program manager, quoted in USA Today, *A.I.: Latest foot soldier in the war on terror*, http://www.usatoday.com/life/cyber/tech/review/2001/10/1/software-agents.htm, 2001
5. Greaves, M.; Holmback, H., and Bradshaw, J. M., *What is a conversation policy?* In M. Greaves and J. M. Bradshaw, editors, Proceedings of the Autonomous Agents '99 Workshop on Specifying and Implementing Conversation Policies, 1999.
6. Gruber, T. (1993). *Toward Principles for the Design of Ontologies Used for Knowledge Sharing*, Knowledge Systems Laboratory KSL 93-04, Stanford University.
7. Free OnLine Dictionary of Computing (FOLDOC; http://foldoc.doc.ic.ac.uk/foldoc/index.html)
8. Phillips, L.R.; Goldsmith, S.Y.; Spires, S.V.; *Ontological Leveling and Elicitation for Complex Industrial Transactions;* Unpublished manuscript, Advanced Information System Lab, Sandia National Laboratories, 1998
9. Goldsmith, S.; Phillips, L.; and Spires, S. 1998. *A multi-agent system for coordinating international shipping.* In Proceedings of the Workshop on Agent Mediated Electronic Trading (AMET'98), in conjunction with Autonomous Agents '98, Minneapolis/St. Paul, MN, USA
10. Phillps, L.R., and Link, H.E.; *The Role of Conversation Policy in Carrying Out Agent Conversations*; Issues in Agent Communication, Springer, 1998.

Appendix 1. Operations and Roles of U.S./Mexico Cross-border Trade

Operations necessary to move goods across the U.S.-Mexican border; to be advertised in Matchmaker: (illustrated in Figure 2. Data component trading for a shipment transaction)

shipment-initiation
shipment-monitoring
shipment-in-transit-visibility
US-transport
Mexican-transport
Border-crossing-drayage
US-export
Mexican-export
US-import
Mexican-import
US-regulator-filing
Mexican-regulator-filing

Detailed descriptions of the roles of border trade collaborators (Figure 3 illustrates how these roles are connected to an individual transaction):

Originator:

Advertised services: shipment-initiation

Role function(s): Start the process of shipment by giving the fundamental task parameter values

Information: Originator, Origin, Cargo-manifest, elements of signature-list

Matchmaker comments: Normally a manufacturer getting ready to move some goods does this. We expect that an agent will interact with a human to cause the appropriate information objects to be created and the appropriate goals to be opened by the appropriate collaborators. This service doesn't make as much sense in the broad Internet setting, but is perfectly at home in an Intranet environment.

Remarks: Acts to cause the creation of a new transaction object. The originator is sometimes called the "shipper," but we avoid this term because it's also a common name for a document. In the maquiladora environment, the originator and the consignee are different sites of the same corporation.

Consignee:

Advertised services: none

Role function(s): shipment closure

Information: Consignee, destination, arrival time/date,

Matchmaker comments: Not an advertised service, but a role that must be filled by a collaborator in order to finish a shipment.

Remarks: Acts to cause closing of the active transaction object (although archiving, cleanup, etc. goals may ensue). Occasionally referred to as the receiver, but we avoid the term since it's also a common name for a document.

Monitor:

Advertised services: shipment-monitor, shipment-in-transit-visibility

Role function(s): Watch a shipment

Information: Import plan, Export plan, arrival time/date, departure time/date

Matchmaker comments: Not clear whether monitoring is a separable component of a shipment transaction. In-transit visibility is separable and should be advertised as a standalone function.

Remarks: Acts to cause timely completion of necessary information subgoals (technically, the monitor merely notices that some subgoals may fail or have failed, but could naturally ask the appropriate agents to correct some deficiency or put some other contingency plan into action). In-transit visibility requires the presence of onboard sensing and locating hardware as well as a reading and reporting infrastructure that would not necessarily be provided by agents.

Carrier:

Advertised services: US-transport, Mexican-transport, Border-crossing-drayage

Role function: negotiate to plan the route, then physically move goods.

Information: Container, shipper, route plan

Matchmaker comments: Primary service of all those given above is "transport" (i.e., physical translation or "ptrans"). An issue is how to specify the scope within which the service is offered.

Remarks: (none)

POEn/POEx regulator:

Advertised services: none (the service of *filing* with the various regulators is accomplished by the entry/exit broker.

Role function(s): Certify requirements have been met, permit/bar entry/exit

Information: Import/export plans as appropriate, elements of signature list

Matchmaker comments:

Remarks: The agent certifies that certain conditions are met and finally permit entry/exit, as the case may be. This role is filled by the respective customs agencies, although other regulatory agencies may impose additional constraints.

Entry/Exit broker:

Advertised services: US-export, Mexican-export, US-import, Mexican-import, US-regulator-filing, Mexican-regulator-filing

Role function(s): Get all the paperwork right

Information: Import/export plans as appropriate, shipper, route plan, elements of signature list

Matchmaker comments: These services are sometimes offered in combination (e.g., Mexican-export and US-import are closely coupled).

Remarks: The broker needs information about virtually every aspect of the shipment in order to ensure that the appropriate regulatory constraints are met in a timely manner. Note that directionality and nationality can constrain the information needs; the Mexican export broker and the US import broker don't need identical information (although note it is possible for both to be handled by one broker, especially in the maquiladora setting), and Mexican import/US export is almost entirely different from Mexican export/US import in terms of the information needed and which regulatory bodies require it.

This page intentionally left blank.

Appendix II. Publication Reprints

1. *Ontological Leveling and Elicitation for Complex Industrial Transactions*
2. *The Role of Conversation Policy in Carrying Out Agent Conversations*
3. *Agent Communications Using Distributed Metaobjects*

This page intentionally left blank.

Ontological Leveling and Elicitation for Complex Industrial Transactions§

Laurence R. Phillips, Steven Y. Goldsmith, Shannon V. Spires

Advanced Information Systems Laboratory
Sandia National Laboratories
Albuquerque, New Mexico USA
{lrphill, sygolds, svspire}@sandia.gov

Abstract. We present an agent-oriented mechanism that uses a central ontology as a means to conduct complex distributed transactions. This is done by instantiating a template object motivated solely by transaction ontology, then automatically and explicitly linking each element of the basis to an independently constructed interface component. These links are then embedded in acquisition goals and delegated to an agent that knows how to carry out the elicitation process. Having accepted these goals, the agent uses the links to acquire information without reference to interface components and to register this information with the transaction basis. Agents elicit information without disturbing the basis and can integrate the information into the basis without further reference to the link once it is validated. Validation information is attached directly to the links so that the agent need not know *a priori* the semantics of data validity, merely how to execute a general validation process to satisfy the conditions given in the link. An advantage of this arrangement is that the transaction basis, the links with the interface, and the validation requirements are independent of one another and of the elicitation agents. This independence enables an elicitation process to be realized without reference to the interface engine, which is merely an attribute of the links. This means that in practice the interface structure can be instantiated with reference only to link names, remaining sufficiently abstract to enable us to wait until run time to generate the actual interface seen by the informant. It can thus be idiosyncratic; when we generate the interface we can take into account the informant's identity, lexicon, language, time of last contact, etc. Ontological leveling is critical: all terms presented to informants must be semantically coherent with the ontologically motivated basis. To illustrate this approach in an industrial setting, we discuss an existing implementation that conducts international commercial transactions on the World-Wide Web. In this implementation, agents operating within a federated architecture construct, populate by Web-based elicitation, and manipulate a distributed composite transaction object to effect transport of goods over the U.S./Mexico border.

keywords: elicitation, ontological leveling, computer supported cooperative work (CSCW), international commerce

---

§ This work was performed at Sandia National Laboratories, which is supported by the U.S. Department of Energy under contract DE-AC04-94AL85000

This page intentionally left blank.

Ontological Leveling and Elicitation for Complex Industrial Transactions§

Laurence R. Phillips, Steven Y. Goldsmith, Shannon V. Spires
Advanced Information Systems Laboratory
Sandia National Laboratories
Albuquerque, New Mexico USA
{lrphill, sygolds, svspire}@sandia.gov

Abstract. We present an agent-oriented mechanism that uses a central ontology as a means to conduct complex distributed transactions. This is done by instantiating a template object motivated solely by the ontology, then automatically and explicitly linking each template element to an independently constructed interface component. Validation information is attached directly to the links so that the agent need not know *a priori* the semantics of data validity, merely how to execute a general validation process to satisfy the conditions given in the link. Ontological leveling is critical: all terms presented to informants must be semantically coherent within the central ontology. To illustrate this approach in an industrial setting, we discuss an existing implementation that conducts international commercial transactions on the World-Wide Web. Agents operating within a federated architecture construct, populate by Web-based elicitation, and manipulate a distributed composite transaction object to effect transport of goods over the U.S./Mexico border.

1 Introduction

Discussions of elicitation in the literature involve anthropomorphic agents [1], belief revision to accommodate heterogeneous distribution of knowledge [2], shared ontologies [3] and [4], and semantically denotive labels [5]. The notion of teleologically motivated discovery presented in [6] is useful since some elicitation situations need partially-instantiated information structures—cases— to guide the elicitation process.

Ontological leveling is the construction of a central ontology to support several languages. Our approach to ontological leveling builds the central ontology to support one language, then extends it as we add languages. Sharing among languages can occur as the corpus continues to provide translations in both into and out of the central ontology. We use denotive labels, but not in the sense of [5]; labels are used here to maintain the connection between the ontologically motivated basis and the elicitation forms used to populate it.

§ This work was performed at Sandia National Laboratories, which is supported by the U.S. Department of Energy under contract DE-AC04-94AL85000

We have not yet explored teleologically motivated discovery very deeply; although we have extensions in place to permit exploration, we have focused to date on the necessarily very structured communication required by international commerce.

We are interested in the process by which an agent elicits information from another agent when both wish to accomplish a common goal. In general, these agents will be conversant with a common ontology but may use widely divergent syntaxes to entail the semantic content of interest. The efficient mechanism to enable semantically laden communication in this kind of environment is to explicate the common ontology and level it with the relevant fractions of the individually languages. The formal properties of this mechanism are discussed in [7] and include translation, construction, verification, and reversibility.

When an agent requests information of another (referred to as the *informant*), it expects to receive a response. We are interested in the subset of responses in which the informant is acting to assist the agent in satisfying some mutual goal, usually based on an agreement to do so. This occurs in the context of a federated system [3] acting to achieve some goal of the virtual enterprise based on either an existing contract [8] or a trading partner agreement [9]. We assume, therefore, that the informant is acting in good faith—that it is *benevolent* [10]—but does not necessarily provide correct or complete information.

In this circumstance, the informant responds to the request by providing information it thinks is correct in an attempt to satisfy the request. The agent must determine the value of the information proffered by the informant. The agent can then either request more information from the informant or go on to other tasks. In any case, the agent will validate the informant's information, if only by default, and may elicit further information about responses it is unable to validate, perhaps ultimately discarding the information as unreliable and failing to satisfy its goal. This explicit validation at elicitation time helps to to prevent costly dependency-directed backtracking.

2 Ontological Leveling

Using the notation of [7], suppose we have several languages $L_{a\text{-}n}$; an interlingua language L_i; $TRANS_{La,Li}$, a binary relation between top-level forms of L_a and top-level forms of L_i; and BT_{La}, a set of top level-forms in L_i. Suppose further that by some means we have $<TRANS_{La,Li}, BT_{La}>$, an L_i-based semantics for L_a, so that we know how to translate back and forth between L_a and L_i. Normally, this places a burden on the implementors to verify that all statements s_{La} really are equivalent to s_{Li} (their translated $TRANS_{La,Li}$ versions) because L_a will have an independently defined semantics. This is not so difficult for the first language L_a, because its translation can drive the definition of L_i, but becomes increasingly difficult as L_b, L_c, L_d, etc., are added (that is, as $TRANS_{Lb\text{-}}$

$_{n,Li}$ and $BT_{Lb\text{-}n}$ are defined), each with its own semantics. A situation can have features that will collapse the potential combinatorial explosion: First, the $L_{a,...,n}$ languages will have similar semantics when they are "about" the same context. Second, the $L_{a,...,n}$ often are not very expressive, having small vocabularies and simple grammars. Third, the areas where the $L_{a,...,n}$ overlap can be few and denotationally coincident, reducing conflicts. This process of adding additional languages to the set that can be translated into the interlingua and back is called ontological leveling.

Suppose language 1 refers to a property named the "date-of-record" and language 2 refers to a property called "date-of-transaction." In the ontology, we have an object named "filing-date" and another object named "receival-date." We also know, axiomatically, that in order for the transaction to be considered complete, a record of it must first be made. To preserve the semantics of the translation, we can choose to translate date-of-record as filing-date and date-of-transaction as receival-date and mediate during elicitation to ensure that the elicited date-of-transaction is not earlier than the elicited date-of-record, as required by the axiom. Unfortunately, the axioms operate only within each language and its translated terms, not between languages, so formally we can't guarantee that a relationship holds between terms in two different languages just because it holds between their translations. However, when an axiomatic relationship that holds in the interlingua is one that we wish to hold between the reverse-translated terms, we can force the translation to be reversible at elicitation time. In our example, we know that the date-of-record must be no later than the date-of-transaction precisely because we want to force that axiom to hold; we are not going to let an informant make the mistake of saying the transaction is complete before its record has been filed. In practice, we can prevent closure until filing occurs; formally we would also like to prevent the *denotation* of closure until we see the *denotation* of filing, in order to maintain registration of the internal state with the state of the world. We would furthermore maintain the metric information—the dates *qua* dates—as the denotive markers of the events, because dates already have a common semantics. In other words, barring formats, there's a universal calendar already in use, so we need not translate actual dates. Each event object in a transaction, at some point during the transaction's trajectory, will contain a date object that both denotes that the event has occurred and connotes the time of occurrence.

A more difficult case is Total Value (What is the total value of items in this {shipment, invoice, bill of lading, production request, work breakdown, field proposal, ... }). Leveling consists in growing the interlingua to be sufficiently expressive to maintain translation and reversal among several semantic projections, just as it does with the dates, but this is much more difficult to do. For example, the axiom that defines the valuation of one monetary currency with respect to another is time-varying. Does the transaction object contain the value of the shipment that was computed at some time in the past or is

it determined at the time of the request? If the former, must it then also contain the time of that valuation and the axiom that was used?[1] Must *all* such valuation times and conversions be retained? How are we to retain commensurability among the set of valuations in the transaction object? Theory suggests the correct answer is "All such information must be preserved to maintain reversibility," placing the burden on the constructor of a functional interlingua for an industrial setting. As a practical matter, we preserve reversibility where reversal will be applied and denote irreversibility where it will not; relying on use-case analysis [11] to determine which case applies.

3 The Mechanism

Work is assigned to an agent by giving the agent a goal. The goal for elicitation is a *form instantiation goal* (FI-goal); the agent is supposed to return an object that contains the validated results of an elicitation process. FI-goals are members of specialized goal classes that capture the semantics and syntax of the information to be elicited. An FI-goal is a composite object initialized to contain several *unknown objects* (UOs)[12]. A UO is a class instantiation that has no content but is responsive (in a content-free manner) to class protocol. The presence of a UO denotes a lack of information. The UOs from the FI-goal are given to a generating mechanism that creates display code. The display generator knows an appropriate display object for each class of UO. Currently, the display page class for the FI-goal is fixed during design, along with an explicit display object classname for each UO, e.g., "text box" or "radio button." This information is maintained in the automatically-generated initialization code for the display page class and is therefore fixed at compile time.[2]

Each display object contains the name of the datum for which it is the interface. The agent uses this name to re-connect the data retrieved from the informant to the appropriate internal variable. Internally, everything is connected by pointers and composition, but we release "probes" out to some stateless browser, with which we have no contact. At some future time, a probe may return[3] ; if it does, it may contain information we requested, and we must at that time re-connect it to the appropriate data element. However, all probes look alike, so each must contain a denotive signal to allow us to identify the internal object for whom the probe bears information. This identifier is the *name* of the object.

Having retrieved the object, the elicitation agent attempts to verify the information that (supposedly) belongs in it, using verification information contained in the object. This level of verification is relative to this object only (e.g.: "X is supposed to be positive numerical," "Y is supposed to be pure text," a column of figures may be required to add

[1] In practice, the axiom is embodied in a conversion factor.

[2] A planned improvement is to deduce the display object class at display time from the UO class.

[3] We emphasize *may*; the network might go down, the user might decide not to reply, etc.

up to a given total, etc.). When the agent fails to verify, it may continue the elicitation process by pointing out the error and re-requesting the information, perhaps suggesting corrections. Ultimately, an intelligent system could make "do-what-I-mean" corrections and present them to the informant for verification. We continue to examine mechanisms for robust error detection and recovery.

Having completed verification of the data, the elicitation agent passes the object to a mediation agent who attempts to reconcile it with the interlingua-based object. There, it is validated against inter-language constraints based on the axioms formed during leveling. This can cause further rounds of elicitation if conflicts ares found in information from different informants. For example, a receiving entity unable to take delivery at a location specified by the sending entity and agreed to by the transport entity. A robust general mechanism should be able to determine who provided the conflicting information and re-elicit (using the reverse translation out of the interlingua) in a collaborative mode. This general corrective tactic is useful because it can deal with unexpected errors.

4 The Application

The Border Trade Facilitation System (BTFS) [13] is an agent-based collaborative work environment that assists geographically distributed commercial and government users shipping goods across the US-Mexico border. This is currently a complex, paper-based, error-prone process that often incurs expensive inspections and delays. In the BTFS, agents mediate the creation, validation and secure sharing of shipment information and regulatory documentation over the Internet, using the World-Wide Web to interface with human actors. For each transaction, the BTFS coordinates several business entities and their agents, two national customs offices, hundreds of data, and several non-communicating computer systems.

The required regulatory documents for each leg of the trip are numerous and bilingual. North American Free Trade Agreement (NAFTA) requirements have complicated the documentation. A typical package prepared by a Mexican broker includes the original invoice; the Shipper's Export Declaration; a Spanish language invoice called the *factura*; an import *pedimento* (Mexican declaration document; an example form rendered in HTML is shown in [fig. 1]); an English manifest and a Spanish *manifiesto* describing the physical nature of the shipment for transport; a packing list, describing how the shipment is actually arranged on the transport; and any of several possible Mexican regulatory documents. NAFTA documents must be on file certifying the firm as a *maquila*, and each *pedimento* must be registered by the owners to satisfy year-end material-balancing regulations. The driver and vehicle must be licensed and certified. The *maquilas* can consolidate several invoices/*facturas* under a single *pedimento*. Shipment into the US involves several additional US import documents. The documents are syntactically

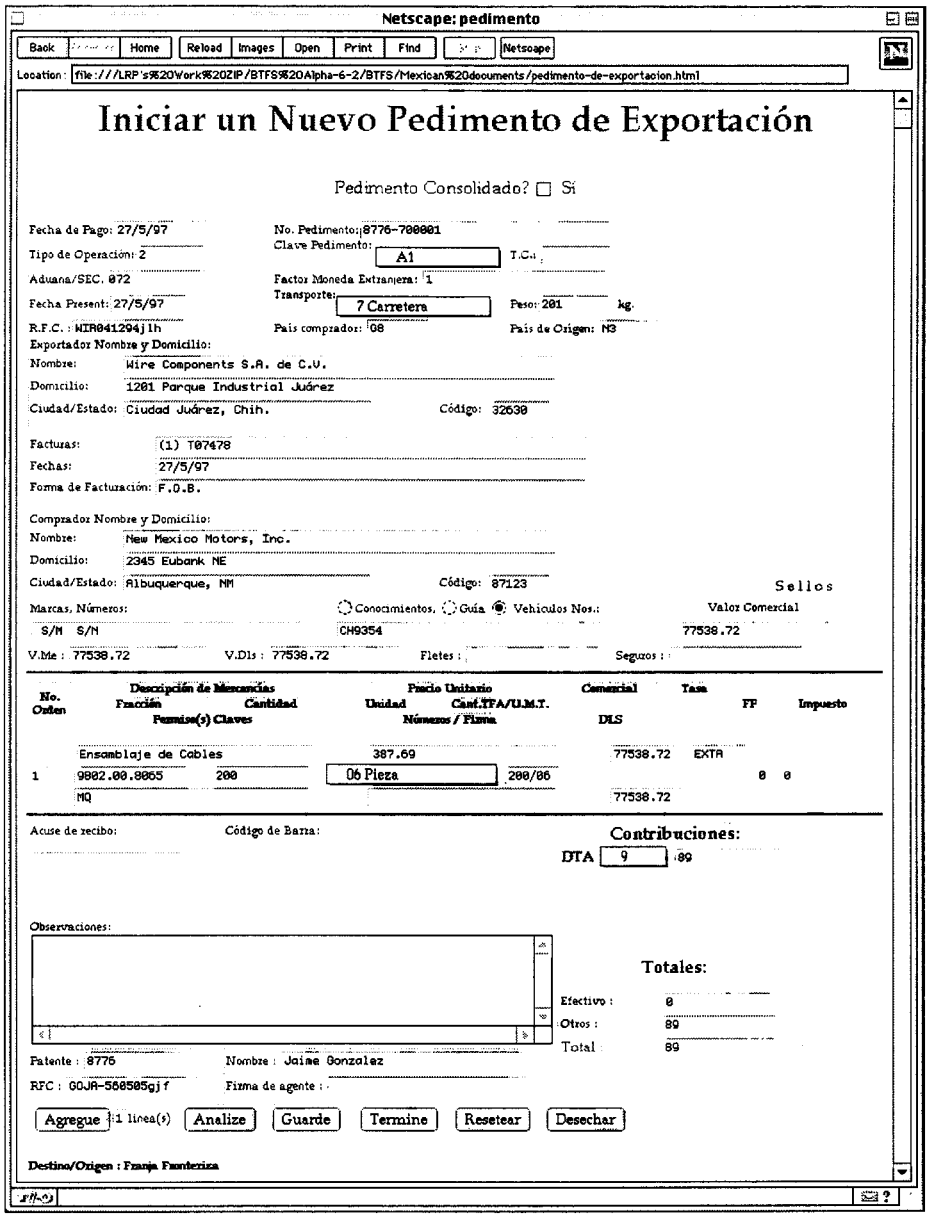


Fig. 1. The Pedimento: In addition to the 57 separate data fields on this form, note that the section between the horizontal bold lines, containing fourteen fields, is one line item and can be repeated any number of times (although practical considerations limit this to a few hundred).

distinct, although there is significant semantic overlap. For example, the total shipment value required on many of these documents is not necessarily given the same name between any given pair nor will the total always be computed on the same basis; and there are at least two currencies involved.

Agents perform four specific functions on behalf of their user organizations: (1) agents elicit information from informants; (2) agents translate information into and out of the central interlingua, thereby eliminating the need for duplicate data entry; (3) cohorts of distributed agents coordinate the work flow among the various information providers and monitor overall progress so that regulatory requirements are met prior to arrival at the border; (4) agents provide status information to human actors and attempt to influence them when problems are predicted. In this paper we discuss functions (1) and (2). See [14] for a more thorough treatment of the Standard Agent class.

We perceived that any electronic system that was to enable *maquila* trade would require a

Maquila-Enterprise-Transaction

Maquila-enterprise-transaction-signature-list ----------

Route-plan ----------

Origin-point	Entry-route	Exit-route
Port-of-entry-point	Port-of-exit-point	Destination-point
Route-plan-signature-list		

Shipper ----------

Originator	Carrier	Origin
Arrival-time-and-date	Departure-time-and-date	Destination
Cargo-manifest	Shipper-signature-list	

Container ----------

Identifier	Owner	Current-route
Surety-history	Sensor Configuration	Container-description

Export-Plan ----------

Country-of-origin	Port-of-exit	Export-history
Export-declaration	Inspection-history	Export-broker
Export-plan-signature-list		

Receival ----------

cargo-condition	Receival-signature-list

Import-Plan ----------

Country-of-destination	Port-of-entry	Import-history
Import-declaration	Import-inspection-history	Import-broker
Import-plan-signature-list		

Fig. 2. The *Maquila* Enterprise Transaction: Each sub-object is a separate entity with tens to hundreds of its own attributes.

central ontology as shown in fig. 2. Several informants would then be required to interact with this ontology to conduct a transaction. Furthermore, the information provided by the informants was to remain distributed for business, cultural, and political reasons. The ontological leveling activity consumed several hundred person-hours.

However, once the *maquila* ontology had been explicated and embodied in the interlingua, names could be assigned to links in generated software that would permit registration of information passing between the BTFS and the human informants through the web pages. The process of using the interlingua during system realization is:

Step 1. Generate a template of the form that will be used during elicitation. Currently, any mechanism that generates HTML can be used. [Fig.1] is an example.

Step 2. Objectify the HTML using an automated process attached as a utility to the BTFS. A fragment of such code follows; note the "payment-date" value given to the "name:" attribute of the "TEXTBOX" object. This object contains the information that will appear in the "Fecha de Pago" textbox in [Fig. 1].

```
•
•
•
(olp::make-object
    'TABLE-CELL t
    :valign 'TOP
    :components
    (list (olp::make-object
            'PARAGRAPHS t
            :components
            (list (olp::make-object
                    'TEXT t
                    :visible-aspects (list "Fecha de Pago:")
                    )
                (olp::make-object
                    'TEXTBOXES t
                    :name "payment-date"
                    :size 12
                    :old-value "27/5/97"
                    :current-value "27/5/97"
                    :preset-value "27/5/97"
                    :instructions nil
                    )
•
•
•
```

Step 3: At elicitation time, using the code created in Step 2, instantiate the object whose translation into HTML will produce the display of [Fig. 1]. The automatically - generated "payment-date" object is shown in [Fig. 3].

Step 4. The instantiated object is given to an elicitation agent as part of a form instantiation goal. As part of the process of achieving that goal, the agent generates the HTML for the web page that recreates [fig. 1] for the informant. A fragment of such agent-generated HTML follows. Again note the explicit "name" attribute.

```
•
•
•
<P><TABLE BORDER=0 CELLSPACING=0 CELLPADDING=0
          WIDTH="100%" HEIGHT="100%"
          name="date-&-pedimento-number-table">
   <TR>
      <TD VALIGN=top WIDTH="34%">
          <P>Fecha de Pago:
          <INPUT TYPE="text"
          NAME="payment-date"
          VALUE="27/5/97"
          SIZE=12>
      </TD><TD VALIGN=top COLSPAN=2 WIDTH="63%">
•
•
•
```

Human Actors are people that inhabit the agency through an interface device and interact with agents to accomplish tasks. Human actor objects are temporary objects that contain an interface address, an interface object that captures the display, data entry and control functions currently available to the person, and a persistent person object that holds personal data, passwords, email address, and an account object that provides access to past and current workspaces. A workspace object contains objects created and stored by the person during work sessions.

Agents and human actors have access to *resources* such as databases, fax machines, telephones, email handlers, and other useful services. Resource objects provide concurrency control and access protocols for agency resources. Subclasses of the resource class implement objects representing data bases, fax machines, printers, email ports, EDI ports and other commonplace legacy devices in the agency environment.

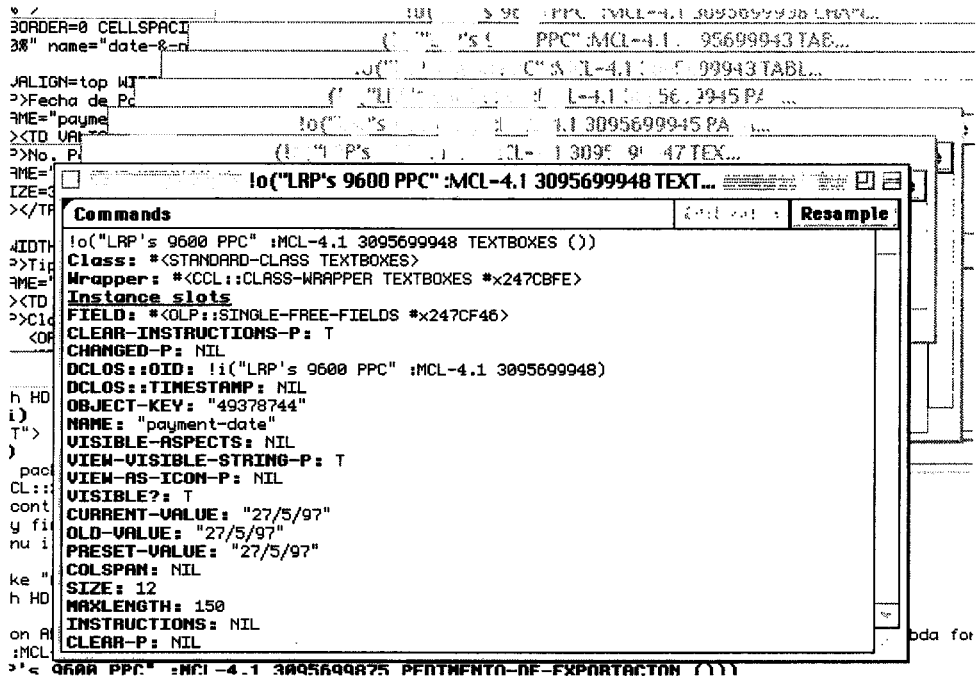


Fig. 3. The actual "payment-date" object (nested several levels down in the page object) generated when the *pedimento* web page is automatically generated for the informant.

Agency objects may be distributed in a network environment to create a collaborative enterprise structure of interconnected agencies. An *electronic commerce agency* (ECA) is a specialized subclass of an agency that implements architectural features specific to electronic commerce applications. An ECA has the additional attributes of *transactions* and *organizations*. The transactions attribute holds a collection of open and closed transaction objects. The organizations attribute holds a collection of public proxy objects pointing to agencies that represent trading partners.

The BTFS agent society comprises several federated ECAs analogous to the interested business entities. Each ECA is populated by a heterogeneous collective of agents, each of which is able to perform a fragment of the information tasks needed to effect trans-border shipment. Business rules are idiosyncratic, so an operational ECA must be tailored and situated for each business. Constructing the ECA and the agents that make it up consists in specializing agents from a set of standard agent classes constructed for commerce. ECA classes are also pre-defined for the various required roles: originator, receiver, transport provider, and import/export broker.

In addition to domain and task specialists, several varieties of housekeeping agents perform maintenance tasks for the ECA. Security agents control access by human actors to each agency within the parent organization. A human actor logged into the ECA "inhabits" the agency for the duration of the work session. An agent handles all interactions with the human actor. Task agents initiate requirements to obtain information based on activated goals, monitor the appropriate information sites to see whether the goals have been achieved, and take corrective or contingency measures when failures occur. Dispatch agents allocate new transactions to the appropriate agents. Supervisory agents allocate work to task agents, deal with rejected goals, collate agency-level data, and respond to outside requests for task status information. Various agents incorporate reporting facilities for humans, including customs offices of both governments.

5 Conclusions and Remarks

The BTFS prototype demonstrates a multi-agent approach to coordinating a complex, knowledge-intensive shipping process. We have demonstrated the following agent behaviors: elicitation, mediation with a central ontology, negotiation, delegation, monitoring, and goal satisfaction.

The most challenging aspects of integrating a diverse enterprise such as border trade are: (1) knowledge-intensive elicitation of form information; (2) mediation and ontological leveling of information across multiple organizations; (3) knowledge engineering in general; and (4) secure distributed object computing.

Ontological leveling proved to be a demanding but effective strategy for centralizing and making coherent a diffuse and permanently decentralized operation. Current research is looking at further automation of the realization process that produces usable applications with demonstrable formal properties.

References

1. Isbister, K., and Hayes-Roth, B. (1997) *Social Implications of Using Synthetic Characters: an Examination of a Role-Specific Intelligent Agent*, Knowledge Systems Laboratory KSL 98-01, Stanford University

2. Dragoni, A. and Giorgini, P. (1995) Distributed knowledge elicitation through the Dempster-Shafer theory of evidence: a simulation study. In *Proceedings of the International Conference on Multi-Agent Systems (ICMAS 96)*, December 1996.

3. Genesereth, M., and Ketchpel, S. (1994). Software agents. In *Communications of the ACM*, 37(7):48-53, 1994

4. Gruber, T. (1993). *Toward Principles for the Design of Ontologies Used for Knowledge Sharing*, Knowledge Systems Laboratory KSL 93-04, Stanford University.

5. Luke, S., Spector, L., and Rager, D. (1996). Ontology-based knowledge discovery on the world-wide web. In *Proceedings of the Workshop on Internet-based Information Systems*, AAAI-96, Portland, Oregon, 1996

6. Balabanović, M., Shoham, Y., Yun, Y. (1995). *An Adaptive Agent for Automated Web Browsing*, Technical Report SIDL-WP-1995-0023, Stanford University

7. Van Baalen, J. and Fikes, R. (1993) *The Role of Reversible Grammars in Translating Between Representation Languages*. Knowledge Systems Laboratory, KSL-93-67, November 1993

8. Sandholm, T., and Lesser, V. (1995) On automated contracting in multi-enterprise manufacturing. In *Proceedings of Improving Manufacturing Performance in a Distributed Enterprise: Advanced Systems and Tools*, Edinburgh, Scotland

9. Röscheisen, M., and Winograd, T. (1996) A communication agreement framework for access/action control. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland

10. Rosenschein, J. S. and Genesereth, M. R. (1985). Deals among rational agents. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, Los Angeles, CA

11. Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G. (1992). *Object-Oriented Software Engineering*, ACM Press.

12. Goldsmith, S., Spires, S., and Phillips, L. (1997). *The Object Lifecycle Protocol*, Advanced Information Systems Laboratory Technical Report, Sandia National Laboratories, Albuquerque, NM

13. Goldsmith, S., Phillips, L., and Spires, S. (1998) A multi-agent system for coordinating international shipping, submitted to *Workshop on Agent Mediated Electronic Trading (AMET'98)*, in conjunction with Autonomous Agents '98, Minneapolis/St. Paul, MN USA

14. Goldsmith, S. (1997). *The Standard Agent Framework*, Advanced Information Systems Laboratory Technical Report, Sandia National Laboratories, Albuquerque, NM

The Role of Conversation Policy in Carrying Out Agent Conversations

Laurence R. Phillips and Hamilton E. Link

Sandia National Laboratories
MS 0445
Albuquerque, NM 87185
lrphill@sandia.gov , helink@sandia.gov

Abstract. Structured conversation diagrams, or conversation specifications, allow agents to have predictable interactions and achieve predefined information-based goals, but they lack the flexibility needed to function robustly in an unpredictable environment. We propose a mechanism that dynamically combines conversation structures with separately established policies to generate conversations. Policies establish limitations, constraints, and requirements external to specific planned interaction and can be applied to broad sets of activity. Combining a separate policy with a conversation specification simplifies the specification of conversations and allows contextual issues to be dealt with more straightforwardly during agent communication. By following the conversation specification when possible and deferring to the policy in exceptional circumstances, an agent can function predictably under normal situations and still act rationally in abnormal situations. Different conversation policies applied to a given conversation specification can change the nature of the interaction without changing the specification.

1 Introduction

A: An argument is a connected series of statements intended to establish a proposition.
B: No, it isn't!
A: Yes, it is! It isn't just contradiction!
Policy discussion, Monty Python,
Argument Clinic Sketch

Software agents communicate while they pursue goals. In some cases, agents communicate specifically in order to accomplish goals. We restrict our interest in this paper to goals that can be described as information states, that is, *information goals.* We discuss agents that intend to accomplish information goals by communicating.

Although individual speech acts have been well-characterized, consensus on higher-order structured interactions has not been reached. There is little or no discussion in the literature of how to constrain the behavior of an agent during communication in response to a dynamic environment.

When a set of communication acts among two or more agents is specified as a unit, the set is called a *conversation.* Agents that intend to have a conversation require internal information structures that contain the results of deliberation about which communication acts to use, when to use them, whom the communications should address, what responses to expect, and what to do upon receiving the expected responses. We call these structures *conversation specifications*, or specifications for short. We claim that specifications are inadequate for fully describing agent behavior during interaction.

Consider two agents who are discussing the location of a surprise party for a third agent, who is not present. When that agent enters the room, all discussion of the party suddenly ceases. The cessation occurs because the first two agents understand that the third agent cannot receive any information that such a party is being considered. Conversely, suppose that the conversation is about a party in honor of the third agent and all three agents know the third agent is aware of it. Now, when the third agent enters the room, the conversation continues.

Are the first two agents having the same conversation in both cases? We claim the answer is "Yes, but they're operating under different policies." In both cases, they are having a conversation whose essence is organizing the party. The conversation might roughly be specified to contain information exchange components (e.g., to establish a set of possible locations), allocation components ("I'll call these two restaurants, and you call this other one"), and a continuation-scheduling component ("I'll call you tomorrow with what I find out and we'll take it from there"). These are all matters that we expect to find in a conversation specification. On the other hand, the decision of whether to stop talking when a specific third party enters the room is based on a mutually understood policy and might reasonably be applied to any number of conversations, for example, negotiations about the price of a commodity on which the third agent is bidding.

Historically the agent communication literature has used the word "policy" to refer to the description of the structure of interaction between a number of agents, generally two but sometimes more (Bradshaw et al. 1997). The dictionary, however, defines "policy" as "a high-level overall plan capturing general goals and acceptable procedures." This coincides with what we expect of a conversation policy: An agent using a conversation *policy* would operate within certain constraints while attempting to satisfy general information-based goals. When discussing procedures and constraints of interaction beyond the basic structure of a conversation, the word "policy" has connotation that we feel is more appropriately bound to the procedures and constraints rather than to the basic structure. For the latter, then, we will instead use the word "specification," and use the word "policy" to refer to the former.

2 Policies for Interaction

The focus of our work is to create a mechanism for combining specifications with policies that constrain the behavior of an agent in order to generate conversations among agents.

We have begun to design a mechanism that uses the specification's description of input states and actions based on them and the policy's description of constraints, limitations, and requirements together to determine an agent's response to a message. Given a suitable mechanism, the specification and the policy can be implemented as data objects. The specification defines the structure for the conversation, and the policy defines the acceptable procedures, rules, and constraints for the conversation.

We can interact with and speak of agents as intentional systems (Dennett 1987). We assume that agents are able to emit illocutions and that illocutions can have perlocutionary effect on other agents that "hear" them (Searle 1969). (We follow Searle in using *illocution* to mean an utterance intended to affect the listener and *perlocution* to mean the production of effect on the listener). This means that an agent can emit information with the intent of altering the information state of some other agent, that the information can be received by some other agent, and that receipt of this information can cause the recipient to be in an information state intended by the emitter. The emitter desires the recipient to be in a certain state because the emitter believes that this either is or assists in achieving one or more of its goal states.

Conversation specifications are distinctly similar to KAoS conversation policies (Bradshaw et al. 1997). The specification dictates the transitions and outputs made by the agent in response to input. A conversation policy is a set of constraints on the conversation specification that limit the behavior of an agent beyond the requirement of following the procedures and structures of the conversation specification. The policy object is used by the mechanism to make decisions about acceptable courses of action when the conversation specification fails to completely determine a course of action. Lynch and Tuttle said it well: "Our correctness conditions are often of the form 'if the environment behaves correctly, then the automaton behaves correctly.'" (Lynch and Tuttle, 1989) This stems from the constraint that IOA's cannot block inputs, the automaton is permitted to exhibit arbitrary behavior when "bad" or unexpected inputs occur. What happens when the environment *doesn't* behave "correctly?" This is where policy applies.

Policy differs from specification in that specifications describe individual patterns of interactions, while policies are sets of high-level rules governing interactions. It is possible for a class of conversation policies to have subclasses. For one policy to be a subclass of another, the subclass must be more strict (more constraining) in at least one attribute and no less constraining in any.

Our new mechanism combines the policies and specifications to determine the set of conversations that can be generated. When policies change in the midst of a conversation, the goal may become infeasible. In our formulation, the conversation policy does not specify the types of messages that can occur. It is made up of constraints on who can participate, and under what circumstances, whether sub-conversations can be initiated within an existing open conversation, whether equivalent conversations can take place in parallel with the same participating entities (e.g., an agent can't carry on two price negotiation conversations with the same entity w.r.t. the same object). We claim that issues of specification are orthogonal to issues of policy; specifications define the structure of interactions, while policies govern the way interactions are carried out.

3 Methods

We developed our current agent conversation mechanism using the Standard Agent Architecture (SAA) developed by the Advanced Information Systems Lab (Goldsmith, Phillips, and Spires 1998) at Sandia National Laboratories. The SAA provides a framework for developing goal-based reasoning agents, and we are currently using a distributed object system that enables agents to send each another simple objects or sets of information. We are using the Knowledge Query and Manipulation Language (KQML) (Labrou and Finin 1997) as our message protocol.

Interacting with an agent first requires that the agent be able to correctly identify and respond to illocutionary messages. A situated agent in pursuit of goals must be able to answer two questions: To which, if any, of its current goals does new information relate, and what actions, if any, should it execute based on new

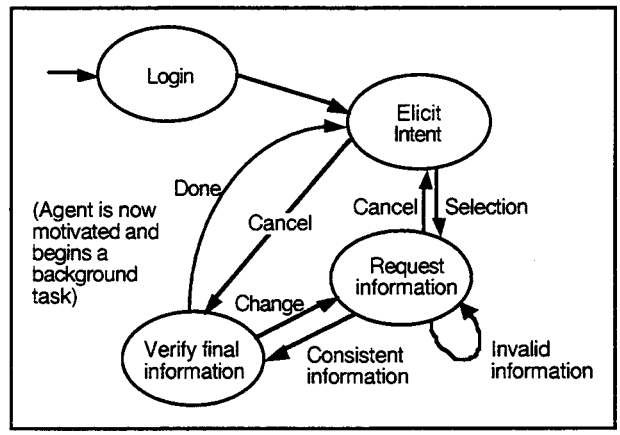


Fig. 1. A conversation specification that does not specify a variety of potential constraints on the agent's activities

information that relates to a given goal? In the SAA, the primary structure that enables this is the agent's stimulus-response table (SRT). An agent anticipating input of a certain type puts an entry into its SRT, which maps stimuli (by class or instance) to the appropriate action. Our system currently requires messages to contain an explicit reference to the context within which the SRT entry was created. The reference is realized as the object identifier (OID) of the current conversation object that gave rise to the message.

When an input arrives, the appropriate SRT entry is retrieved and its goal is undeferred (having previously been deferred, presumably awaiting relevant input), which activates the goal. The agent now determines how the new information in the context affects the goal and either marks it satisfied, failed, or deferred or continues to attempt to satisfy the goal. When satisfaction of the goal requires a speech act, the

agent creates an utterance, delineates the context, embeds the context signature in the utterance, attaches the goal to the context, places the entry in the SRT, defers the goal, and executes the utterance. In short, illocution is a deliberate act that creates an utterance and sets up an expectation of the response that the recipient will make.

To engineer a conversation, the entire set of context descriptors of interest is laid out as a set of subgoals, each of which is satisfied by gathering specific information. We have automated the construction of an utterance from a context, the updating of the context to reflect the new information conveyed by the input, and the connectivity that enables the utterance and the input to refer to the same context. Specialized code is written to construct goals, execute side effects, maintain the SRT, and so on.

Composing speech acts in a theoretically predictable fashion is more difficult; this is the motivation for creating a structured way of merging specification and policy at run time to get a structured interaction that is forced to remain within certain operational boundaries.

In our current mechanism, policy is embedded in the conversation mechanism as part of the design. A policy change, for example, that an agent should institute a timeout and ignore all messages responding to a particular request after the timeout expires, would require reengineering the conversation. The mechanism would be much more maintainable given an explicit policy object that could just be changed to reflect the fact that there's now a timeout. Our essential thesis is that policies and conversation specifications should be independent so that conversations could be switched under the same policy and policies could be changed without changing existing conversations.

4 Conversation policy

Consider the conversation in Figure 1. It describes a session allowing agent A to determine agent B's identity, offer B a choice of services and ascertain B's selection, and perform a task based on the selection. Describing the conversation is generally simple for such things: when a request or assertion comes in, the agent deliberates, returns information to the initiator, and anticipates the continuation. The two participants are responding to one another in turn, barring interruption, retransmission, or communication failure. There is no representation of what happens when the conversation is interrupted or when an agent retransmits a message. These issues are matters of policy that must be dealt with separately.

KAoS conversation "policies" enable definite courses of action to be established and fail-stop conditions to be dealt with (Bradshaw et al. 1997). They also imply mechanisms for initiating and concluding conversations. Specifications play the crucial role in agent communication of providing structure, but they do not, for example, describe whether a discussion can be postponed, or, if so, under what conditions or for how long. Indeed, KAoS conversation "policies" appear to concern matters of conversation *specification*, fundamentally how to respond to input given the current information state, rather than matters of conversation *policy*, such as what to

do when interrupted, whether the conversation can be postponed, or whether there is a time constraint on reaching an end state.

Policy issues are important. One constraint imposed by the policy in Figure 1 is that it requires turn-taking. If agent A receives several messages in a row, it may respond to each in turn without realizing that, say, B's third message was sent before A's second response. If agent A cannot detect the violation of the turn-taking policy, it might consider the second and third messages in an outdated context. A similar situation could occur if several agents were communicating and one were speaking out of turn. Without policy, designing a mechanism to deal with these violations means that a conversation specification that enforced turn-taking and one that merely allowed it would be two different things that would need to be maintained separately and activated separately by the agent. Furthermore, designing them into a system that had no notion of turn-taking would require that every state/action pair of every conversation specification be examined to see what should now happen if turn-taking is violated. At worst, accommodating a single policy issue doubles the number of conversation specifications an agent might be called upon to employ.

Examining constraints immediately leads to ideas for policies that replicate familiar patterns of interaction, such as a forum policy or a central-point-of-contact policy. Different classes of states, changes in context, and the particular protocol of communication used are independent of the conversation policy, although some make more sense in one policy or another. The web page and information-state context, for example, make the most sense in a 1:1 turn-taking policy when dealing one-on-one with a number of individual humans. KQML, in contrast, has many performatives that support broadcasting to a group of agents involved in the same conversation. In practical terms we may end up having to constrain which policies can be upheld based on communication details.

An explicit representation of policy also enables an agent to express the policy under which it is operating. It is easy to transmit, say, a policy message outlining the level of security required for any of several possible upcoming conversations for which the recipient already has the specifications. In contrast, without policy, the "secure" version of each conversation specification needs to be transmitted anew. If two agents agree on a policy at the beginning of a conversation, the amount of communication required to determine a course of action once a violation has occurred can be minimized.

The structure of the conversation depends thus on the nature of the information and how this changes the state of the conversation. By abstracting to the policy level, we enable a set of constraints to support the execution of several conversations, as long as they have the same *kinds of states* and the same *kinds of transitions*, i.e., the nature of information in a state does not matter as long as there is a common means of mapping input and state to another state in the conversation. If the conversation can be described as a collection of states with transitions between them, then the conversation policy should be describable as a *form* of transition function operating on the current perceived state of the world and the communications the agent is receiving.

This abstraction is powerful because the individual conversation policies can be combined with specifications to create several classes of conversations, all similarly constrained. The constraints the framework imposes are then the conversation policy;

and specializations of the conversation policy framework methods are implementations of particular transition functions, which operate on particular classes of conversations. These conversation policies would support transformations by our mechanism, each of which defines a range of possible specializations within the high-level constraints. Radically different behavior between two sets of conversations would imply radically different frameworks, just as the difference between context-free grammars and regular languages implies a greater difference in both the nature of states and the transition function forms of finite automata and stack machines.

5 Example

Consider the specification in Figure 2. Agent A_1, the announcer, broadcasts a message to a group of agents $A_2 \ldots A_n$ and gathers responses from the group before continuing. By itself, however, this specification leaves many questions unanswered—for example, if some agent doesn't respond at all, or responds more than once in a cycle, what should agent A_1 do? These questions may be asked of many specifications, and may have different answers even from one interaction to the next.

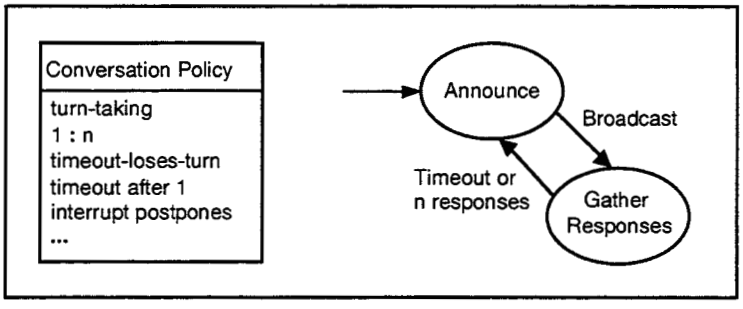


Fig. 2. Policy and specification as seen by the announcer. The policy allows conversations to be postponed, which the conversation specification need not explicitly state.

The policy in Figure 2 provides answers to some questions of this sort. The policy enforces turn-taking, meaning that agents in the group have only one opportunity to respond to each broadcast. If they do not respond within one minute of each broadcast, they lose the chance to do so during that turn. This might be the case if broadcasts were frequent. If more pressing matters come up during a session, the discussion is postponed (perhaps leaving messages in the announcer's queue to be dealt with later), but it can be expected that the session will resume at some future time.

How might we tailor policies to get usefully different behavior? For policies concerned with fault-tolerance, the same policies could be used in many conversations

to handle the same expected problems, but policy can also be used to control conversations during the course of normal interaction as well.

Suppose we combine the specification above with a policy that does not enforce turn taking, but rather says that newer messages from an agent take precedence over older messages. The announcer is forbidden from sharing message data among group members, and the time allowed for responses to each broadcast is 24 hours. Combining the policy and specification with a sales announcer produces a silent auction. If the policy were replaced with one that had a time limit of a few minutes and required the announcer to rebroadcast new information to the group, the same specification could be used to produce an English auction. Using different policies with the same specification as a foundation can produce a variety of desirable behaviors with minimal changes to the agent's code.

6 The Impact of a Policy Mechanism

In this section we discuss the relationship between conversation specifications, policies, and an operational mechanism. We show how policy information can be used to direct the action of an agent without reference to the conversation that agent is having.

Consider a set of state/action pairs with the property that when an agent perceives the world to be in a given state and executes the corresponding action, the world state that results is described by the "state" component of one of the pairs (I/O automata fall conveniently close to this). States with no corresponding actions are end states. Such a set embodies no notion of intent, but an agent can commit to achieving one of the end states by executing the actions. The point of an action specification is to explicate a series of acts that will result in one of a known set of states.

A conversation specification is such a set of state/action pairs; the specified states are information states and the specified actions are speech acts. A conversation specification explicates a series of speech acts and their triggering states that will result in a one of a known set of information states. An end state may be a goal state, i.e., a state whose achievement is the agent's intent, or a state in which the desired state is known or believed to be either no longer desirable or unachievable.

The conversation specification may specify states and actions that are never realized; e.g., failure-denoting states or error-correcting actions. All actions and states are only partially specified, in the sense that none specify the entire state of the world, because the number of features that might be observed at execution time is infinite, and only a few of these are perceived at specification design time as having any material effect on movement towards the goal.

For example, a plan that includes forming a team might specify neither who is to fill every role on the team, though a specific agent must be cast in each role, nor in what order the roles are to be filled, because the specific order has no effect on the goal state.

Neither the conversation specification nor the policy controls the thread of conversation; the specification specifies the invariant part of the conversation's course, and policy specifies constraints on behavior, not the behavior itself. Control falls to

the mechanism that combines the specification object and the policy object to arrive at an executable action at deliberation time. In the remainder of this section, we examine team formation with respect to what is determined by the conversation specification and what is determined by policy.

Assume that an agent is in a state where will listen until it receives a message from another agent. When a message arrives, the agent's policy is to select and commit to achieve one of the end states of a particular conversation specification; in other words, the agent's policy is to have a conversation when contacted. Leaving aside for the moment the question of how the agent makes the selection, assume the agent receives a message asking it to commit to achieving a goal and that it selects a conversation specification wherein it will inform the requester that it has committed if it commits and that it will not commit if it doesn't. This could be a matter of policy; suppose there were many agents available and this was known to the agent. The agent might reason that the best policy would be to report only when it could commit and to keep silent otherwise, in order not to use bandwidth.

Now what happens when an agent achieves a goal to which it has committed? Should the agent report satisfaction to the requester, when there is one? If this were a matter of policy, it could be turned on or off as overarching issues (security, priority, traffic levels, etc.) dictated and overridden as needed by specific policy overrides from the requester.

What should the agent do when it is asked to achieve a goal it believes it cannot achieve by itself? It might be the agent's policy to refuse to commit and to so report. An alternative policy would be to attempt to acquire commitments from other agents to assist. This would begin the team formation phase.

When the agent has acquired commitments from agents whose combined effort it believes can achieve the goal, it builds the team roster of agents $\{A_1, \ldots, A_n\}$, marks the team formation goal satisfied, and ends the team formation phase (this ignores the issue of whether everyone on the team must believe the team can achieve the goal in order to commit). It might be the case that the agent must form the team within a given time period; what the agent should do when it does not receive sufficient commitments within the allotted time is a matter of policy. A reasonable policy would be to report to the original requester that the goal is unsatisfiable. This can be enforced at a high level, that is, *whenever* the agent has committed to achieving a goal, and the source of that goal is another agent, the agent must notify the source agent as to the achievement or non-achievement of that goal. The agent holding a team roster for a given goal constructs a joint persistent goal (JPG) (Cohen and Levesque 1991), allocates the subgoals (assume the goal is linearizeable so that allocation is deterministic) and sends each subgoal and the JPG to the appropriate team member. The JPG contains a statement of the original goal and the team roster. When an agent A_i has achieved its subgoal, it multicasts this fact to the rest of the team using the roster in the JPG. Here, policy to notify only the requester must be overridden by JPG-specific policy. Every team member now believes A_i has achieved its subgoal. Once A_i believes that every team member has achieved its subgoal, it believes that the JPG has been satisfied and it multicasts this fact to the rest of the team. At this point, A_i believes that every team member believes that the JPG has been satisfied and is free to leave the team.

7 Conclusions

A conversation policy must be established so that the communicating agents (who may have differing languages) have a common logical and contextual structure for communicating. This allows each agent to establish predictive models of one another's behavior in response to information and to plan and reason about the outcome of conversations with the other agent. Each agent can establish this model based on information that another agent can perform a certain conversation specification while conforming to certain requirements.

We advocate a separate conversation policy structure that embodies the constraints that will be enforced while a conversation is going on—using a conversation specification as a template or model. A participant in a conversation must have some means of determining whether events that transpire during the conversation bear on the realization of its goals. It is relatively straightforward to specify the normative events in a conversation; the speaker intends to have engendered a specific state in the listener, and the normative response types are limited. On the other hand, it is not generally possible to specify all the exceptions. Even if we could, the necessary responses depend on states of the environment, not states of the conversation. To take the state of the environment into consideration, a policy must be able to constrain the behavior of virtually any conversation specification to which it is applied.

8 Future Developments

It would be useful to define and prove certain formal properties of policies when combined with specifications, for example,

1. Is the question of whether a conversation conforms to a given conversation policy decidable, and if so, how can this be tested?
2. Does conversation X conform to some conversation policy, and if so, which one?
3. What is the maximally confining policy to which a set of conversations conforms?
4. Will the conversation generated from a specification terminate when following a particular policy?
5. Under certain circumstances, a policy may render given specifications impossible. What is the minimal set of constraints that can be established that will still allow a set of conversations to take place?
6. Given a policy that has the potential to render a conversation impossible, what should an agent do?

Consider for a moment the agent as an I/O automaton (IOA) (Lynch and Tuttle, 1989). The IOA's I/O table specifies the agent's behavior. The IOA's input column describes agent's information states. These states can be entered as an agent internalizes information in messages it has received (i.e., as those messages have perlocutionary effect). The agent then executes the specified internal and external actions specified by the right-hand side of the automaton's I/O table. This formalism has some appeal because it makes a very clear distinction between actions under

control of the automaton and those under control of the environment and allows a readable and precise description of concurrent systems.

Analyzing collections of speech acts in terms of I/O automata would be possible if it were not for the dependency of the proofs about the IOAs on their being input-enabled. Agents that filter their stimuli before taking action or replying do not meet this requirement, so the applicability of the IOA theory is questionable.

A formal theory that establishes conversation semantics, describes how the semantics of individual speech acts contribute to conversation, and allows us to demonstrate certain characteristics of combinations of specifications and policies may or may not be useful. When discussing a system whose purpose is to deal with the unexpected, it may be more reasonable to engineer a policy that provides some reasonable capstone when an unanticipated problem arises. Engineering conversations that meet certain requirements, dynamically generating policies and specifications based on beliefs and intentions, and modifying conversations based on changing constraints may allow productive agent behavior even in the absence of a complete theoretical description.

9 In Context

Throughout these papers we see two common issues being addressed: by what means can an agent intend to have, and then have, a conversation, and by what means can an agent manage the process of having conversations in a dynamic environment? Two recurring subproblems are declaring behavioral models for an agent's own use and transmitting these models to other agents; agents need to be able to express the following in both internal and external settings: "This `<conversation_spec>` is the conversation I want to have" and "This `<conversation_policy>` is the policy I want to follow." In this paper we have labeled these structures *conversation specifications* and *conversation policies*, respectively.

A primary question roughly separates the papers in this volume into two categories: Are issues of specification and policy to be addressed by a single structural form that unifies specification and policy (Category 1), or by two separate structural forms, one for specification and one for policy, that are somehow composed during the conduct of a conversation (Category 2)? We are in category 2, having explicitly proposed a *policy object* to be communicated among conversing agents.

An essential question, approached by some authors, but not genuinely disposed of, is: what, exactly, is gained by having two structures? Although efficiency, complexity, and realizeablility have been used as motivators, we'd like to see a formal approach that enables decisions of where a particular aspect of discourse should be represented and, in particular, how such decisions are realized when policies and specifications are composed during a conversation.

10 Acknowledgements

This work was performed at Sandia National Laboratories, which is supported by the U.S. Department of Energy under contract DE-AC04-94AL85000.
Regina L. Hunter made numerous valuable comments on the manuscript.

References

1. Bradshaw, J.; Dutfield, S.; Benoit, P.; and Wooley, J. 1997. "KAoS: Toward an Industrial-Strength Open Agent Architecture," in Software Agents, AAAI Press/MIT Press.
2. Cohen, P. R., and Levesque, H. J. 1991. Confirmation and Joint Action. In Proceedings of the 12th Annual International Joint Conference on Artificial Intelligence.pp 951-959, Menlo Park, CA, Morgan Kaufmann
3. Dennett, D.C. 1987. The Intentional Stance. Cambridge, MA: MIT Press.
4. Goldsmith, S.; Phillips, L.; and Spires, S. 1998. A multi-agent system for coordinating international shipping. In Proceedings of the Workshop on Agent Mediated Electronic Trading (AMET'98), in conjunction with Autonomous Agents '98, Minneapolis/St. Paul, MN, USA
5. Labrou, Y. and Finin, T 1997. A Proposal for a new KQML Specification, Technical Report, CS-97-03, Dept. of Computer Science and Electrical Engineering, University of Maryland, Baltimore County
6. Lynch, N. A. and Tuttle, M. R. 1989. An Introduction to Input/Output Automata, Technical Memo, MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology
7. Searle, J. 1969. Speech Acts, Cambridge, UK: Cambridge University Press.

Agent Communications Using Distributed Metaobjects

Steven Y. Goldsmith
Shannon V. Spires
Advanced Information Systems Laboratory
MS 0455
Sandia National Laboratories
Albuquerque, NM 87185
505-845-8926
sygolds@sandia.gov, svspire@sandia.gov

Abstract

There are currently two proposed standards for agent communication languages, namely, KQML (Finin, Lobrou, and Mayfield 1994) and the FIPA ACL. Neither standard has yet achieved primacy, and neither has been evaluated extensively in an open environment such as the Internet. It seems prudent therefore to design a general-purpose agent communications facility for new agent architectures that accommodates many agent communications languages. In this paper we exhibit the salient features of an agent communications architecture based on distributed metaobjects. We are primarily concerned with the pragmatics of agent communications using objects rather than agent communications languages per se. We are particularly concerned with agent communications in the open Internet environment. Our architecture captures design commitments at a metaobject level, leaving the base-level design and implementation up to the agent developer. The scope of the metamodel is broad enough to accommodate many different communication protocols, interaction protocols, and knowledge sharing regimes through extensions to the metaobject framework. We conclude that with a powerful distributed object substrate that supports metaobject communications, a general framework can be developed that will effectively enable different approaches to agent communications in the same agent system. Moreover, we explicate some seeming peripheral issues to ACL (e.g. authentication, integrity, reasoning and memory) that are actually critical to the concerns of agent communications and that certainly impact effective communications in an open environment.

Keywords: agent communication language, multiagent system, metaclass, metaobject protocol, distributed objects

1 Introduction

Communication among autonomous asynchronous agents is an essential function in network-based multiagent systems. There are currently two proposed standards for agent communication languages, namely, KQML (Finin, Labrou, and Mayfield 1994) and the FIPA ACL. Until a standard emerges, an agent designer must accommodate this uncertainty in agent designs. One approach is to exploit the considerable syntactic commonalities between the two, but this can produce implementations with serious semantical problems, at least from the perspective of the speech acts underlying both languages. Moreover, both languages have inherent problems with semantics based on modalities that are not supported by the components that interface closely with the language, primarily the agent's deliberative mechanisms and its implementation of ontologies in an agent's long-term memory. Unless the agents implements a belief, desire, intention (BDI) architecture (georgeff, et al), the semantics of objects communicated through KQML or the FIPA ACL is limited to modal propositions and cannot be readily interpreted by another deliberative architecture.

Our design philosophy is to develop a general object-centered framework that enables programming of multiple protocols for communication and interaction alongside multiple approaches to deliberation and action (of which BDI is an instance). Figure 1 shows the general architecture for agent communication, discussed in detail in subsequent sections. The components are: (1) the send-object protocol that provides a standard interface for remote communication of objects; (2) a message object protocol that interprets the structure of the message object, enabling multiple communication protocols (e.g. KQML, ACL); (3) a metamodel that manages the update of remote agent models and the local agent's model; and (4) the model of local agent and models of remote agents. The framework includes an infrastructure for agent modeling because communication among two agents requires both a common message format and a shared ontology. Since agents may be in different states, communications is mediated through the receiver's model to ensure common semantics. The agent's self-model contains the deliberative mechanisms and knowledge bases that are exclusive to itself. The self-model has control over the operations of the remote agent models

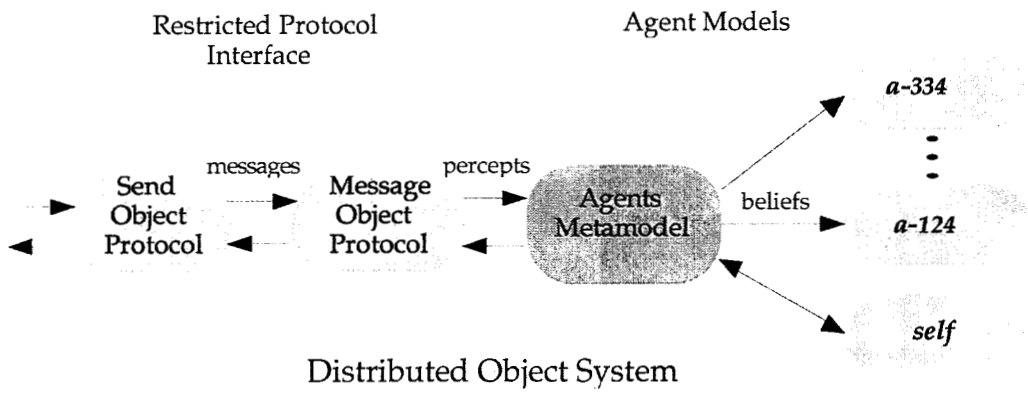


Figure 1. Distributed Object Agent Communications Architecture

through the metamodel. We assume that the agents communicate both the structure and the state of their models to one another for the purpose of collaboration by trading objects. The entire architecture is based on the object framework concept. The classes and methods comprising the architecture are designed to be specialized with subclasses and methods that implement the agent designer's favorite communication, interaction, reasoning and representation mechanisms. Our objective is to provide both a research tool for evaluating new regimes and a practical system capable of operating in heterogeneous environments such as the Internet.

2 Distributed Objects

Our approach to the design and implementation of network agents relies heavily on a comprehensive distributed object subsystem implemented in the Common LISP Object System (CLOS). Agent designs involve compositions of objects and metaobjects, many of which are intrinsically capable of distribution in a network environment. Communicating among agents that are described as compositional objects has a natural interpretation; it is an instance of message passing among objects and as such has a well understood syntax and semantics. A distributed object is an object that has a commonly-known identity and is represented by some form of surrogate object in multiple address spaces around the network. Distributed object surrogates are of three primary types: proxies, copies, and replicants. (There can also be a fourth, hybrid type which combines features of the main three.)

Proxies are pure surrogates. A proxy object "stands in" for a real object that is located elsewhere. The proxy accepts messages destined for its "real" object, delegates them to the real object for processing, receives the result of the message, and passes the result along to the original message sender. Proxies are very handy for projecting an object's capabilities from its current location to other places on a network. They are immune to update issues, since any change to a real object will be immediately reflected in the responses of all of its proxies. Proxies are the primary object distribution mechanism of CORBA [ref http://www.omg.org]. The downside of proxies, of course, is that every message sent to a proxy invokes a network transaction.

Copies are just that; an object is copied and sent from one network location to another. Pure copies keep no information about their "source" object (and vice-versa) so they cannot be updated if the source object changes. But of course, if the data and functionality contained in the copy is needed frequently at another location, this may be an acceptable price to pay to avoid the network overhead of a proxy.

Replicants are copies that keep track of their source (and/or vice-versa) such that they can be updated if their source object changes. Replicants thus provide the best features of both proxies and copies: information currency with low network overhead, as long as accesses are more frequent than updates and we are willing to pay the price of more bookkeeping.

Hybrid objects can exhibit proxy, copy, or replicant behavior on a slot-by-slot basis. Hybrids are probably the most useful form of object distribution in general because the distribution mechanism choice can be made at a fine level of granularity.

In our discussion of copies and replicants above, we omitted one nasty detail: objects in a modern inheritance-based dynamic OO system [in which class and method meta data exist at runtime] never exist alone. Objects themselves are but the tips of two massive icebergs: an inheritance graph and a containment graph. In order to truly copy an object from Point A to Point B on a network, we must also copy its inheritance graph—its class, and its class's superclasses, and methods thereof—and we must also somehow distribute any objects it references or contains. In an OO system like C++ where classes are not first-class objects, this can only be done if the requisite classes and methods already exist on the destination machine. But in an OO system like CLOS where classes and methods are first-class objects, we can treat the classes and methods themselves as merely more objects to be copied and copy them on-demand, using the same mechanism we use to copy pure instances. It is the classes and methods that we refer to with the term *metaobjects.*

Distribution by proxy is popular in the distributed object community because it is immune to update problems and it does not require that classes or methods be present at the target node; it is fundamentally based on delegating messages to a remote "real" object. But as we've already noted, the performance penalty for such delegation can be large and sometimes must be avoided. Therefore distribution by transporting whole copies of objects is essential, especially when moving an agent on a network or sharing ontologies among fixed agents. But copying objects also requires copying class lattices (distributing class lattices by proxying them usually won't work) and methods. And even if the objects we move are pure copies (no updating expected), we must usually transport their class lattices and methods as *replicants,* not pure copies, because if a class definition or method changes, the changes must be promulgated. This is why

most distributed object systems either make no attempt to copy or replicate objects or do so in only a limited fashion. Solving the replicant problem in general is quite difficult, especially in static OO languages. It gets even worse: in CLOS, classes themselves are instances of metaclass objects. If any transported class is an instance of a special metaclass, the metaclass must be transported also. Fortunately, the replication problem is soluble in CLOS because of its extensive introspective capabilities and its metaobject protocol.

The actual movement of a CLOS object takes place in two stages: serialization and materialization. To serialize an object means to flatten it into a sequence of bytes that can be used to reconstruct the object at another place. In CLOS, the essential information that must be serialized is the object's class name and its slot contents. Serializing an object is relatively straightforward, provided we are careful to maintain referential integrity among slot contents, and to recursively serialize any other objects that may be referenced in its slots. Once a sequence of bytes is produced, it is transmitted over the network to the receiver.

At the receiver, materialization begins. The receiver looks at the class name of the incoming object and checks to see if that class is present locally. If not, it asks the sender to serialize and transmit the class metaobject. (When the class metaobject is materialized at the receiver, the receiver will check to see that all its superclasses and metaclasses are also present and may recursively request their transmission as well.) If the class is already present at the receiver, the receiver may check its timestamp, hashcode, or some other version-maintenance identifier to ensure that it has the latest version. If not, it may request that the sender transmit the latest version of the class. Methods and generic functions are also transmitted or updated along with the class metaobjects that specialize them. Finally, once the receiver is satisfied that the object's requisite infrastructure is present, it simply allocates space for an object of the appropriate class and fills in its slots with the original serialized data.

The above is the standard "pull" mechanism for demanding an object's infrastructure when the object is pushed. Objects that are replicated, not merely copied, can also be updated on a "push" basis by the sender when necessary.

Proxies are still very useful in many cases and can be implemented in CLOS much more dynamically than in CORBA: no a priori knowledge of allowed messages is needed. Any message sent to a proxy that the proxy does not immediately understand can be automatically delegated to the proxy's "real" counterpart by overriding the CLOS no-applicable-method mechanism. New messages can thus be created on-the-fly for real objects and any proxies to those real objects can immediately take advantage of them.

We have demonstrated that there is no inherent barrier to providing copies, replicants, and proxies as distribution mechanisms for objects and metaobjects. Nevertheless, the reader will have noted we have said nothing yet about the security implications of such wide-open distribution. Even though our basic mechanism is quite general, it is usually necessary to impose some limitations on its power because of security considerations. The architecture discussed in subsequent sections addresses some security issues.

Our distributed object substrate provides a general purpose communications mechanism capable of implementing many different agent communication systems, including KQML. However, most standard distributed object systems are not powerful enough to implement the features needed to provide security, shared knowledge/ontologies and agent modeling. [JAVA and CORBA discussion here].

3 Autonomy, Identity, and Integrity

Autonomy is a cornerstone in the modern specification of intelligent agents. Roughly speaking, autonomy implies an agent acts without the direct intervention of humans or others, and have some king of control over their actions and internal state (Castelfranchi 1995). Our operational definition of autonomy is:

1. An agent is a locus of unique, persistent identity
2. An agent is a unique locus of self-control
3. An agent is a unique locus of reasoning

An autonomous agent will be self-determined with respect to its beliefs, goals and actions. It will be known to other agents by a unique name that identifies it as an independent entity within the agent community. In a multiagent collaborative system, agents rely on the autonomy of one another to make certain inferences about the other agents. Casterfranchi (1995) identifies two distinct classes of autonomy: stimulus autonomy and executive autonomy. A message from another agent qualifies as a stimulus to the receiver. An agent may choose to respond to a stimulus or not to respond, depending on the current state of the agent's deliberations. Executive autonomy requires that an agent cannot be directly motivated with the goals of another agent unless the agent decides that the goals are congruent with its own. Under no circumstances should an agent attempt to satisfy a goal object obtained directly from another agent without first evaluating and criticizing the goal within the context of its own knowledge state and goals.

Implementing stimulus autonomy and executive autonomy requires the design of a safe communications protocol that maintains the integrity of the agent while allowing effective communication. We propose that the functional property of *agent integrity* is a necessary element for agent autonomy. Integrity is an operational concept that seeks to protect the agent's internal structures from direct manipulation by another agent, including human actors. An agent cannot be self-determined or self-controlled unless it is impossible for others to directly influence its beliefs and actions unbeknownst to the agent. Distributed object protocols can introduce vulnerabilities that undermine agent integrity. The Nefarious Neurosurgeon of Dennett (1984) introduces electrodes into the brain of the victim Jones and controls his every thought in an undetectable manner. An agent that can dispatch an arbitrary method invocation on an object argument in the address space of another agent is capable of direct intervention in the agent's deliberations and actions. Agents operating within a multiagent system that does not restrict the remote method invocation (RMI) process cannot believe in a distinct locus of identity and control for one another, since control of an agent by another nefarious agent is possible. Integrity mechanisms force RMI to implement a restricted protocol that cannot address arbitrary objects and methods within an agent program. In its full exposition, this problem is identical to security concerns identified for mobile agents (Chessman 1994, Vigna 1998). Since our agents are composite objects with full support for object distribution, they can potentially send one another any of their structural and procedural components, including the entire agent corpus of the sending agent, to function as an endosymbiont within the receiving agent, for example[1] . Unrestrcited trading of metaobjects, i.e., classes and methods can pose a serious threat to the receiver agent. An agent must have a restricted object trading protocol that implements a criticism policy to protect it against dangerous foreign objects.

Agents operating in an open network environment are also vulnerable to impersonation through active attacks on the communications links. The maintenance of agent integrity requires a cryptographic authentication protocol among collaborating agents. Agents must have a high degree of trust in the authenticity of the source of a message in order to ascribe attributes such as beliefs and state to the sending agent. Models of other agents must be managed as distinct loci of reasoning and knowledge to detect inconsistent states among agents and to effectively maintain reputation structures (Zacharia 1999) for other agents .

4 Object Communication

A careful look at the life cycle of a single agent-to-agent message, i.e. the simplest an instance of agent communication, reveals that messaging involves the most fundamental actions of an agent. Messaging is a deliberate, motivated action, designed to achieve a specific goal. In the speech act interpretation of a message, the agent desires to entrain a specific mental state in the receiver. Our model of agent communications is more general, enabling agents to share both communicated objects and other elements of their implementation such as ontologies and goals. Figure 2 represents the sequence of events leading to transmission.

[1] An endosymbiont is an agent with persistent identity operating within the address space occupied by another agent.

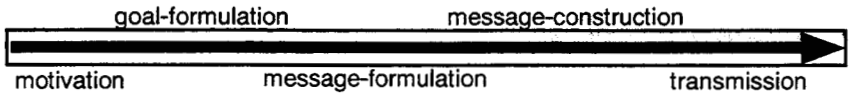


Figure 2. Events Leading to Transmission

motivation
The motivation for transmission is generally derived from some higher goal of the agent. Fundamentally, the agent must inform another agent or obtain information from another agent, obviously in a social setting with other known agents.

goal-formulation
The agent creates a goal object that encapsulates the details of the communication act. Satisfaction of the goal is complete when the object closure is obtained with respect to the goals of the communicative act. This involves spawining subgoals to receive a response, if any, and to evaluate the response in the context of the goal.

message-formulation
The actual message is formulated with a sender, receiver, and content object. Depending on the communication protocol employed (e.g. KQML), additional information may be added. The exact formulation is compatible with the communication protocol employed by the receiver object.

message construction
A specific class of message object is constructed for transmission as copied distributed object. The copied object will be transmitted directly to the receiver.

transmission
The message object is transmitted to the receiver.

The Send-Object method (Figure 1) implements transmission of a message object. Each agent is registered in the network with a well-known proxy object. An agent holds the proxy to another agent in the agent model (discussed below).

Send-object(agent-proxy, message-object)

The send-object method is invoked in the target agent's environment through remote delegation via the proxy. The invocation is restricted to a specific namespace in the target agent that contains the agent proxy and proxy class, the send-object metaobject, the classes of possible message objects, and filtering functions to evaluate the message and its content. The distributed object system checks the serialized message for references to other namespaces and rejects the message it contains other references. Thus the send-object protocol is a virtual chokepoint for messages, preventing direct invocation of methods on objects outside the restricted namespace. We call this element the Restricted Protocol Interface.

Receiving an object from another agent is also a deliberate act on the part of the receiver. It requires the necessary motivation and goal creation to create the context for evaluating the communicated object. In general, an agent must associate the communicated beliefs with persistent goals to determine their salience and to formulate the proper actions in response.

motivation
The motivation for reception is derived from a normative persistent goal provided by the framework that creates within the agent the desire to receive information from other agents.

goal-formulation
The agent creates a goal object that determines which agents will be considered for interaction. The goal is mutable, and agents may be removed from consideration for a variety of reasons, including security, chronic poor performance on collaborative tasks, and prioritization under severe resource constraints.

message-expectation
Certain messages or classes of messages may be expected, perhaps in response to a previous transmission in the context of a conversation. The framework enables a message object to directly invoke a specific achievement goal in the self-model that has been deferred pending more information. An expectation mechanism within the Message-Object element (Figure 1) can directly determine the context for message processing through a reference to the context goal. This provides a mechanism for implementation continuous conversations between agents.

message deconstruction
Each message must be deconstructed according to its class. For example, a KQML message will be reduced to its component fields and the salient objects extracted by the Object-Message protocol. The components representing the percepts are then passed to the metamodel for processing.

reception
The new beliefs are presented to the self-model and updates the remote agent model.

elaboration
A deliberation mechanism within the receiving agent is activated to determine the ramifications of the new beliefs with respect to the agent's goals.

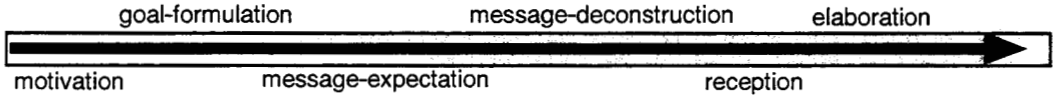


Figure 3. Events Leading to Reception

The architecture provides the source of motivation for social interaction among agents. The framework provides classes and and method metaobjects that enable the construction of sending and listening goals.

5 Agent Models

Agents have local beliefs about other agents and the world. In order to distinguish its local beliefs from those of other agents, each agent has a distinct model of itself and distinct models of other agents. The object constant self denotes the local agent and constants of the form **a-1**, **a-2** , and **a-100** denote the other agents in the environment. Models of other agents allow the local agent to reason about the beliefs, goals, and actions of others. The Agents Metamodel (Figure 1) manages the update of an agent's models from communicated information. The communications protocol passes message objects to the Agent Metamodel (Fig 1) for elaboration and interpretation. The metamodel makes certain inferences about the beliefs of the local agent and other agents based on communicated messages. First, the receiving agent must be able to recognize the sender agent as the true source of a message. Each agent in the system has a unique, persistent, and verifiable identity. Cryptographic authentication of each message by digital signature enables the receiver to attribute the message to the identified sender with certainty. Although the exact operation of the metamodel depends on the particular representation of belief, the following logical model based on deductive belief (Konolige 1984) illustrates the point. The predicate **message**(y,x,z) denotes a message with content object x sent from agent y signed with digital signature z. The metamodel computes the signature using the digital signature function, reified as a trinary relation **dsa**(x,y,v), where x is the message, y is the agent id (used to obtain the public key) and z is the computed digital signature. Note that

this digital signature scheme is distinct from the authentication protocols used at the transport level. Agents require a different signature scheme to authenticate their identity to one another at the knowledge level. Certain collaborative activities may require more specialized signature schemes still. Protocols for encryption and authentication at the link level may be constrained by the network and transport layer underlying the communications system.

Validation of the digital signature sanctions the belief by the local agent in the belief of the sender via the schema:

message(a-123, x, z) « **dsa**(x,a-123,v) « **eq**(z,v) Æ **Bel**(***self***, **Bel**(a-123, x))

Bel(***self***, **Bel**(a-123, x)) is asserted in the local (self) model of the agent, while the argument **Bel**(a-123, x) is asserted in the model of agent a-123. Alternatively, an invalid message is not believed by the local agent[2] :

message(a-123, x, z) « **dsa**(x,a-123,v) « ¬**eq**(z,v) Æ ¬**Bel**(***self***, **Bel**(a-123, x))

The conclusion **Bel**(c, x), where c is an arbitrary constant, is asserted in the model corresponding to the "unknown agent". This captures the notion "somebody believes x".

Control of an agent's models of other agents is mediated through the metamodel. The local agent may wish to check an agent's model for consistent beliefs. The metamodel provides a uniform protocol to the local agent for performing queries, proving assertions, and importing hypothetical beliefs from a model into its self-model.

Each model of a remote agent comprises a distinct namespace, a set of metaobjects (classes and methods) that implement the interface to the metamodel, and a separate thread to control execution of methods. At the framework level, instances and metaobjects transmitted by the actual remote agent are represented as simple beliefs of the form **Bel**(a, x), where a is the agent name and x is any object or metaobject. This captures the primitive notion that an agent believes in the existence of the referenced object or metaobject. Included are complex compositions of objects implementing part-whole relationships. Compositions are handled naturally by the underlying distributed object system by coercing the message content object and all its components into copied objects during materialization.
The framework is easily specialized for a particular representation. Candidates include categorical taxonomies such as description logics (e.g. CLASSIC, LOOM, KL-ONE), KIF (Finin, Labrou, and Mayfield 1994), first-order logic and theorem provers, deductive data bases, BDI architectures, and so on. Custom representations rendered in the object language are also possible. These different representations may be active simultaneously in different agent models provided the necessary interface protocol to the metamodel exists.

Direct communication of metaobjects between agents enables agents to share their models of one another and the environment. An agent decides which elements of its representation and in what representational scheme will be used by other agents to model its reasoning and behavior. Through an interaction protocol, agents can negotiate detailed descriptions of their shared models, enabling cooperation on joint tasks. The framework supports this in two ways. First, every model is ultimately rendered in CLOS through metaobjects and instances, providing a common programming language with which the agents remotely but safely program their corresponding models residing in other agents. This in effect creates an endosymbiont within the local agent representing a special projection of the remote agent without degrading the integrity of the local agent. Secondly, a model of another agent is a dynamic process under the control of the local agent. The local agent can use the model to predict the behavior of a remote agent, to the extent that model allows. This enables a powerful simulation mechanism within an agent that facilitates cooperative actions.

---

[2] The metamodel will attempt to validate the message for all agents in its knowledge base. If this fails, the message is invalid. If it succeeds, the valid agent id is substituted in the message.

6 Conclusions

We have discussed several pragmatic issues associated with agent communications in an open network environment. We have described a general architecture that ensures agent integrity, supports agent modeling, and enables multiple representations and communications protocols to coexist in the same agent.

References

Castelfranchi,C. 1994. Guarantees for autonomy in cognitive agent architectures. In *Intelligent Agents ECAI-94 Workshop on Agent Theories, Architectures and Languages*. Springer-Verlag.
Dennett, D. 1984. *Elbow Room*. MIT Press. Cambridge MA.

Finin, T., Labrou, Y., and Mayfield, J. 1994. KQML as an agent communication language. Computer Science Department, University of Maryland Baltimore County.

Goldsmith, S., Phillips, L. and Spires S. 1998.

Konolige, K. G. 1984. A deduction model of belief and its logics. Technical Note 326. Menlo Park, CA: SRI International, Artificial Intelligence Center.

Labrou, Y., Finin, T. and Peng, Y. 1999. Agent communication languages: The current landscape. In I*EEE Intelligent Systems and their applications.* March/April 1999. IEEE Computer Society.

Phillips, L.R., Goldsmith, S. Y. , and Spires, S.V. 1999. CHI: A general agent communication framework," Proc. of the Hawai'i International Conference on System Sciences.

Vigna, G., ed.1998. *Mobile Agents and Security.* In Lecture Notes in Computer Science, vol. 1419. Springer-Verlag.

Zacharia, G. 1999. Trust management through reputation mechanisms. In *Proceedings of the Workshop on Deception, Fraud, and Trust in Agent Societies*. Autonomous Agents 99, Seattle Wa.

This page intentionally left blank.

Appendix III. Code and Programs

1. Code file of the ontology of "documented-transportings" in Common Lisp Object System (CLOS) form
2. Code file showing a "DocumentedTransportings" act in LARKS format
3. Code file of frame specification for exporting goods to Mexico with Ontology information in LARKS format
4. LARKS language support functions for generating ontologies, etc.
5. Code enabling an SAA agent to interact with the CMU Agent Name Server (ANS)
6. Code enabling an SAA agent to interact with the CMU Matchmaker
7. Code enabling an SAA agent to process and transmit KQML

This page intentionally left blank.

```
;; Code file of the ontology of "documented-transportings" in Common Lisp Object
;; System (CLOS) form used by the Standard Agent Architecture (SAA) agents

;The service to be advertised is:
;
;  (our agent) will (generate) and (execute) all (necessary documents) for the
;  (transport) of (goods) between the (US) and (Mexico)
(defpackage :ONTOLOGY
  (:use :ut-lrp :CL :CL-USER :DCLOS :AISL :chi)
  (:nicknames :ONT)
  )

(in-package :ONT)

(#+Allegro excl:without-package-locks #+MCL progn
(defmethod slot-unbound  :around ((me #+MCL ccl::class #+Allegro class)
                                  instance slot-name)
  "Override default slot-unbound behavior. Return nil if slot unbound."
  (declare (ignore instance slot-name))
  (if *su-override*
    nil
    (call-next-method)
    )
))

(defun class-instance-slot-names    (class)
(mapcar #'ccl::slot-definition-name (#+MCL ccl::class-instance-slots
                                    #+Allegro aisl::class-instance-slots class)))

(defmethod instance-slot-names   ((me standard-object))
  "Returns list of names of instance slots of object."
  (class-instance-slot-names (class-of me)))

;;;;

(defclass larks-frame  ()
  ((context :initarg :context  :accessor context  :initform nil)
   (types :initarg :types  :accessor types  :initform nil)
   (input :initarg :input  :accessor input  :initform nil)
   (output :initarg :output  :accessor output  :initform nil)
   (inconstraints :initarg :inconstraints  :accessor inconstraints  :initform nil)
   (outconstraints :initarg :outconstraints  :accessor outconstraints  :initform nil)
   (concdescriptions :initarg :concdescriptions  :accessor concdescriptions :initform nil)
   )
  )

(defmethod completed-p  ((the-frame larks-frame))
  (funcall (outconstraints the-frame) the-frame)
  )

(defmethod actionable-p  ((the-frame larks-frame))
  (funcall (inconstraints the-frame) the-frame)
  )
```

```
(defclass transportable-things    ()
  ((object-id :initarg :object-id
              :accessor object-id
              :type 'OBJECT-ID
              :initform nil)
   (customer-id :initarg :customer-id
                :accessor customer-id
                :initform nil)
   (transport-id :initarg :transport-id
                 :accessor transport-id
                 :initform nil)
   (nl-description :initarg :nl-description
                   :accessor nl-description
                   :initform nil)
   (weight :initarg :weight  :accessor weight  :initform 5000)
   (height :initarg :height  :accessor height  :initform 3.0)
   (width :initarg :width  :accessor width  :initform 3.0)
   (depth :initarg :depth :accessor depth  :initform 3.0)
   )
  )

(defclass documented-transportings    (larks-frame)
  ((goods :initarg :goods
          :accessor goods
          :initform nil)
   (the-documents :initarg :the-documents
              :accessor the-documents
              :initform nil)
   (start-location :initarg :start-location
                   :accessor start-location
                   :initform nil)
   (desired-end-location :initarg :desired-end-location
                         :accessor desired-end-location
                         :initform nil)
   )
  )

(defmethod in ((the-location t) (the-country t)) t)

(defparameter *US*   "Stand-in for an object that represents the United States")

(defparameter *Mexico*    "Stand-in for an object that represents the United States")

(olp::defmaker ((prototype  documented-transportings)     &key)
  (setf (context prototype) :commercial-transport)
  (setf (input prototype) (list (goods prototype)
                                (start-location prototype)
                                (desired-end-location prototype)))
  (setf (output prototype)
        (list (olp::make-object 'bill-of-lading t :shipment prototype)
              (olp::make-object 'border-crossing-permit t))
        )
```

```
  (setf (inconstraints prototype)
        #'(lambda (item)
            (and (or (and (in (start-location item) *Mexico*)
                          (in (desired-end-location item) *US*)
                          )
                     (and (in (desired-end-location item) *Mexico*)
                          (in (start-location item) *US*)
                          )
                     )
                 (< (length (goods item)) 500)
                 (<= (apply #'max (mapcar #'weight (goods item)))
                     10000)
                 (< (apply #'max (mapcar #'depth (goods item))) 4)
                 (< (apply #'max (mapcar #'width (goods item))) 4)
                 (< (apply #'max (mapcar #'height (goods item))) 4)
                 (eval (cons 'and
                             (mapcar #'(lambda (doc)
                                         (not (signed-p doc))
                                         )
                                     (the-documents item))
                             )
                       )
                 )
            )
        )
  (setf (outconstraints prototype)
        #'(lambda (item)
            (and (mapcar #'signed-p (the-documents item))))
        )
  )

(defclass documents  ()
  ((signature :initarg :signature
              :accessor signature
              :initform nil)
   )
  )

(defmethod signed-p  ((the-document documents))
  (signature the-document)
  )

;;
;; "shipment" will contain an instance of "documented-transportings"
;; This is the pointer from the documentation to what it documents
;;
(defclass shipment-documents   (documents)
  ((shipment :initarg :shipment
             :accessor shipment
             :initform nil)
   )
  )
```

```
(defclass bill-of-lading   (shipment-documents)
  ((goods-listing :initarg :goods-listing
                  :accessor goods-listing
                  :initform nil)
   )
  )

(olp::defmaker2 ((prototype   bill-of-lading)   &key)
  (loop for unit in (goods (shipment prototype))
        do
        (push
         (format nil
                 "[Description] ~a [weight in kg] ~a" (nl-description unit) (weight unit))
         (goods-listing prototype)
         )
        )
  )

(defclass border-crossing-permit    (documents)
  ()
  )

(defclass LARKS-INTERFACES   (interfaces)
  ()
  )

(defparameter *larks-interface*   (make-instance 'LARKS-INTERFACES))

(defmethod chi::view-as-interface    ((myself  larks-frame)
                                      (interface interfaces)
                                      stream &key &allow-other-keys)
  (format stream "~a~%  = (and ~{ ~%       (all has-~a) ~} ~%    )"
          (class-name (class-of myself))
          (mapcar #'car (ccl::class-instance-slots (class-of myself)))

          )
  )

#| Execution test and expected results

(defparameter *some-goods*
  (list (olp::make-object
         'transportable-things t
         :object-id (format nil "OID~a" (random 10000))
         :customer-id "Hewlett-Packard"
         :transport-id (format nil "CUSTOMS-MARK--a--a"
                               (random 1000)
                               (random 100000))
         :nl-description "16 4-gross cases single-use surgical gloves palletized"
         :weight 186
         :height 1.86
         :width 1.45
         :depth 2.16
```

```
        )
       (olp::make-object
        'transportable-things t
        :object-id (format nil "OID-a" (random 10000))
        :customer-id "Hewlett-Packard"
        :transport-id (format nil "CUSTOMS-MARK--a--a"
                              (random 1000)
                              (random 100000))
        :nl-description "21 cartons outer shell electronic amplifier palletized"
        :weight 131
        :height 2.45
        :width 2.45
        :depth 2.45
        )
       )
  )

(olp::make-object
 'documented-transportings t
 :goods *some-goods*
 :start-location '(:latitude 95.34.123 :longitude 123.78.342)
 :desired-end-location '(:latitude 51.32.239 :longitude 111.03.893)
 )

(defparameter *the-doc*
  (olp::make-object 'documented-transportings t
                    :goods *some-goods*
                    :start-location '(:latitude 95.34.123 :longitude 123.78.342)
                    :desired-end-location '(:latitude 51.32.239 :longitude 111.03.893)
                    )
       )

*THE-DOC*

? (outconstraints *the-doc*)
#≤Anonymous Function #x6BAC306≥

? (funcall (outconstraints *the-doc*) *the-doc*)
NIL

? (inconstraints *the-doc*)
#≤Anonymous Function #x69C5DAE≥

? (completed-p *the-doc*)
NIL

? (actionable-p *the-doc*)
T

|#
```

```
;; Code file showing a "DocumentedTransporting" act in LARKS format

DocumentTransportableGoods

----------Context

DocumentedTransporting * DocumentedTransportings

----------Types

;; All types are defined as components of DOCUMENTED-TRANSPORTINGS

----------Input

DocumentedTransporting

----------Output

Documents

----------InConstraints

;; YOU'LL NEED TO TRANSLATE THE CONSTRAINTS INTO LARKS
;; NB: contents of the "goods" slot are TRANSPORTABLE-THINGS
; Let's pretend the "in" predicate exists
; note "item" is the arg to the Lambda function

(and (or (and (in (start-location item) *Mexico*)
              (in (desired-end-location item) *US*)
              )
         (and (in (desired-end-location item) *Mexico*)
              (in (start-location item) *US*)
              )
         )
     (< (length (goods item)) 500)
     (<= (apply #'max (mapcar #'weight (goods item)))
         10000)
     (< (apply #'max (mapcar #'depth (goods item))) 4)
     (< (apply #'max (mapcar #'width (goods item))) 4)
     (< (apply #'max (mapcar #'height (goods item))) 4)
     (eval (cons 'and
                 (mapcar #'(lambda (doc)
                                   (not (signed-p doc))
                                   )
                         (the-documents item))
                 )
           )
     )


----------OutConstraints

(and (mapcar #'signed-p (the-documents item)))
```

```
----------ConcDescriptions

DOCUMENTED-TRANSPORTINGS
  = (and (all has-goods (list-of TRANSPORTABLE-THINGS))
         (all has-the-documents (list-of DOCUMENTS))
         (all has-start-location LOCATION)
         (all has-desired-end-location LOCATION)
         )

TRANSPORTABLE-THINGS
  = (and (all has-object-id IDENTFIER)
         (all has-customer-id IDENTIFIER)
         (all has-transport-id IDENTIFIER)
         (all nl-desctiption TEXT)
         (weight REAL)
         (height REAL)
         (width REAL)
         (depth REAL)
         )

DOCUMENTS
  = (and (all has-shipment (list-of IDENTIFIER))
         (all has-signatures (list-of SIGNATURE))
         )

LOCATION
  = (and (all has-latitude REAL)
         (all has-longitude REAL)
         )

SIGNATURE [this is a primitive]

IDENTIFIER [this is a primitive]

TEXT [this is a primitive]

REAL [this is a primitive]
```

```
;; Code file of frame specification for exporting goods to Mexico with ontology
;; information in LARKS format

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Frame specification for LARKS
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

ExportUSGoodsToMexico

Context Shipper*shipper

Input

originator:	USFirm*Firm;
country-of-origin:	country*country;
invoice-number:	integer;
lading-information:	ListOf(piece-description*piece-description)
shipment-initiator:	USFirm*Firm;
shipment-init-date:	date*date;
consignee:	MexFirm*Firm;
arrival-time-window:	duration*duration;
arrival-time:	time*time;
arrival-date:	date*date;
departure-time-window:	duration*duration;
departure-time:	time*time;
departure-date:	date*date;
Ship-from:	USLocation*location;
Ship-to:	MexLocation*location;

Output

US-carrier:	USTransportProvider*TransportProvider
Mex-carrier:	MexTransportProvider*TransportProvider
drayage-carrier:	DrayageProvider*DrayageProvider
exit-broker:	USExitBroker*broker
entry-broker:	MexEntryBroker*broker
scheduled-departure-time:	time*time
scheduled-departure-date:	date*date
predicted-arrival-time:	time*time
predicted-arrival-date:	date*date
fee:	Price*prices

InConstraints

Outconstraints

```
le(departure-time,+(scheduled-departure-time,departure-time-window),)
le(scheduled-departure-time,+(departure-time,departure-time-window))
le(arrival-time,+(predicted-arrival-time,arrival-time-window))
le(predicted-arrival-time,+(arrival-time,arrival-time-window))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Ontology for LARKS
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

Firm

USFirm

MexFirm

broker

USExitBroker

USEntryBroker

MexExitBroker

MexEntryBroker

location                (and (all has-address physical-addresses))

USLocation              (and location
                             (all has-address (all country-of aset(US)))
                             )

MexLocation             (and location
                             (all has-address (all country-of aset(Mexico)))
                             )

TransportProvider

USTransportProvider

MexTransportProvider

DrayageProvider
```

```
;;
;; LARKS language support functions for generating ontologies, etc.
;;

(defclass LARKS-INTERFACES   (chi::interfaces)
  ()
  (:documentation "The dispatching class for all VIEW-AS-INTERFACE methods."))

(defparameter *LARKS-interface*   (make-instance 'LARKS-INTERFACES))

(defun convert-slotname-to-LARKS-name     (slotname)
  (convert-thing-to-LARKS-name (first (third slotname)))
  )

(defun convert-thing-to-LARKS-name     (thing)
  (apply #'concatenate
         'string
         (mapcar #'string-capitalize
                 (cl-user::listify-string (chi::prettify (string thing)))
                 )
         )
  )

(defun CREATE-LARKS-ONTOLOGY-FILE     (classname  &optional
                                                  (the-pathname nil)
                                                  &key (overwrite-existing? t))
  (when (not the-pathname)
    (setf the-pathname
          (make-pathname :directory (pathname-directory (choose-directory-dialog))
                         :name "Larks-ontology"
                         :type "txt")
          )
        )
  (with-open-file (the-filestream the-pathname :direction :output
                                  :if-does-not-exist :create
                                  :if-exists (if overwrite-existing?
                                                 :supersede
                                                 :append)
                                  )
    (view-as-interface classname *LARKS-interface* the-filestream)
    )
  )

(defmethod view-as-interface   ((myself t) (interface LARKS-INTERFACES)
                                stream &key &allow-other-keys)
  (declare (ignore myself interface stream)
           )
  )

(defmethod view-as-interface   ((myself string) (interface LARKS-INTERFACES)
                                stream &key &allow-other-keys)
  (view-as-interface (find-class myself) interface stream)
  )
```

```
(defmethod view-as-interface   ((myself symbol) (interface LARKS-INTERFACES)
                                stream &key &allow-other-keys)
  (view-as-interface (find-class myself) interface stream)
  )

(defmethod view-as-interface   ((myself standard-class)
                                (interface LARKS-INTERFACES)
                                stream
                                &key (visited-hash-table (make-hash-table))
                                &allow-other-keys)
  (let ((method-namelist nil)
        (method-arglist nil)
        (method-name nil)
        (the-method nil)
        (its-a-class-p nil)
        (classes-yet-to-be-visited nil)
        )
    (loop for method in (remove-if #'(lambda (item)
                                       (or (typep item 'standard-reader-method)
                                           (typep item 'standard-writer-method)))
                                   (inspector::specializer-direct-methods myself)
                                   )
          when (not (member (ccl::method-name method) method-namelist :key #'first))
          do
          (push (cons (ccl::method-name method) method) method-namelist)
          )
    (cond ((gethash myself visited-hash-table)
           nil)
          (t (setf (gethash myself visited-hash-table) t)
             (let* ((superclasses (class-direct-superclasses myself))
                    (classname (class-name myself))
                    (indent (+ (length (string classname)) (length " = (and ")))
                    )
               (when (not (listp superclasses)) (setf superclasses
                                                      (list superclasses)))
               (format stream "~a = (and " classname)
               (loop for superclass in superclasses
                     do
                     (format stream "~a~%" (class-name superclass))
                     (dotimes (i indent) (format stream " "))
                     )
               (loop for slot in (class-direct-instance-slots myself)
                     do
                     (when (equal 'quote (fourth slot))
                       (setf its-a-class-p
                             (find-class (intern (string-upcase (symbol-name
                                                                 (fifth slot)))
                                                 :chi))
                             )
                       )
                     (when its-a-class-p (pushnew its-a-class-p
                                                  classes-yet-to-be-visited))
```

```
                    (format stream
                            "(all has-~a~@[ TYPE: ~a~])~%"
                            (convert-slotname-to-LARKS-name slot)
                            (class-name its-a-class-p))
                    (dotimes (i indent) (format stream " "))
                    )
                  (loop for method-name-pair in (reverse method-namelist)
                        do
                        (setf method-name (first method-name-pair))
                        (setf the-method (rest method-name-pair))
                        (setf method-arglist (ccl::arglist the-method))
                        (format stream "(Provides ~a~@[ inputs:~{ ~a~}~])~%"  ; ** ~a~%"
                                method-name
                                (reverse
                                 (set-difference
                                  method-arglist
                                  (list '&METHOD 'NEXT-METHOD-CONTEXT
                                        '&REST 'the-rest '&key '&allow-other-keys)
                                  :key #'symbol-name
                                  :test #'string-equal
                                  )
                                 )
                                ;method-arglist
                                )
                        (dotimes (i indent) (format stream " "))
                        )
                  (format stream ")~%~%")
                  (mapcar #'(lambda (item) (view-as-interface
                                            item interface stream
                                            :visited-hash-table visited-hash-table))
                          superclasses
                          )
                  )
                )
              )
  (loop for one-class in classes-yet-to-be-visited
        do
        (view-as-interface one-class interface stream
         :visited-hash-table visited-hash-table)
        )
    )
  )

(defmethod ccl::class-name   ((myself  symbol))
  myself
  )

#|
(mapcar #'(lambda (item) (view-as-interface item *larks-interface* t))
        (class-direct-instance-slots (find-class 'btfs::us-manufacturer))
        )
|#
```

```
;; Code in this file implements the agent's ability to interact with CMU's Agent Name
;; Server (ANS). This includes formatting ANS commands, interpreting ANS responses,
;; and the process of interacting with that server.

(in-package :genome)

#|
doesn't have training examples... probably should, anyway this file should contain
(ultimately) everything you need for interacting with or running an ANS service
|#

(defgene-fun *ip-exoculos-site*
  delimit ((stream-type (eql :ans)) mstream)
  (let* ((state 0)
         (eol ;; read to the first <cr><lf>
          (loop for next-char = (aisl::stream-read-char-no-hang mstream)
              until (or (null next-char) (eq next-char :eof))
              do (case state
                   (0 (when (eq next-char #\return) (incf state)))
                   (1 (if (eq next-char #\linefeed) (return t) (setf state 0)))))))
    (if eol
        (values (subseq (aisl::buffer mstream) 0) :success)
      (values (subseq (aisl::buffer mstream) 0) :undelimited))))

(defgene-class *oculos-site*
  ans-sorry (material-percept) ())
(defgene-class *oculos-site*
  ans-success (material-percept) ())
(defgene-class *oculos-site*
  ans-fail (material-percept) ())
(defgene-class *oculos-site*
  ans-error (material-percept) ())

(defgene-fun *oculos-site*
  elaborate ((stim-type (eql :ans)) stimulus)
  (cond
   #| ;; recognizing ANS commands, so we could provide such a service; low priority
   ((string-equal "list all" (subseq (content stimulus) 0 8))
   )
   ((string-equal "register" (subseq (content stimulus) 0 8))
   )
   ((string-equal "unregister" (subseq (content stimulus) 0 10))
   )
   ((string-equal "lookup" (subseq (content stimulus) 0 8))
   )
   ((string-equal "bye" (subseq (content stimulus) 0 4))
   )|#
   ;; just the responses that can be generated by register and unregister, for the moment
```

```
   ;; currently sorry<cr><lf> and fail<cr><lf> are too short to be disambiguated. grrr.
   ((string-equal "sorry" (subseq (content stimulus) 0 5))
    (make-instance 'ans-sorry))
   ((string-equal "success" (subseq (content stimulus) 0 7))
    (make-instance 'ans-success))
   ((string-equal "fail" (subseq (content stimulus) 0 4))
    (make-instance 'ans-fail))
   ((string-equal "ERROR" (subseq (content stimulus) 0 5))
    (make-instance 'ans-error))))

(adding-to-gene *oculos-site*
  (define-schema ans-register-schema ()
    (ans-ipaddr ans-port name contact-info)
    :do-if (string-equal (name schema) (name *N*))
    :do (send-message "IP-ARTICULOS-NUCLEUS"
                      (list "TCP-SEND-MSG"
                            :destination (list (ans-ipaddr schema) (ans-port schema))
                            :raw-data (format nil "register ~a ~a~c~c"
                                              (name schema) (contact-info schema)
                                              #\return #\linefeed)
                            ;; all send-action-msgs have ff now -- note that tcp-send-msg isn't a send-action
                            :final-fn #'(lambda () (complete schema)))))
  (define-schema ans-success-schema ()
    ()
    :match-if t
    :match (ans-success t))
  (define-schema ans-error-schema ()
    ()
    :match-if t
    :match (ans-error t))

  (define-schema ans-registration ()
    (ans-ipaddr ans-port)
    :subschemata (:register (ans-register-schema :ans-ipaddr ans-ipaddr :ans-port ans-port
                                                 :name (name *N*)
                                                 :contact-info "kqml://134.253.158.211:5557")
                  :success (ans-success-schema)
                  :sorry (ans-error-schema))
    :dfa ((:start :transitions ((:register :wait)))
          (:wait :transitions ((:success :done) (:sorry :fail)))
          (:done :signal :complete)
          (:fail :signal :failed)))
  )

;; this basically works, except that the response is not being detected
#+ignore
(do-schema (make-instance 'ans-registration
             :ans-ipaddr (pip:dotted-to-ipaddr "134.253.158.26")
             :ans-port 6677))
```

```
;; Code in this file implements the processes for interacting with the CMU
;; Matchmaker server. The Matchmaker uses KQML to exchange information.

(in-package :genome)



(defconstant *CMUMatchMakerServer* 'SandiaLarksMatchMaker)
(defconstant *CMUANSServer* '("artemis.cimds.ri.cmu.edu" 6677))
(defconstant *LocalANSServer* '("beldin" 6677))

(defgene-fun *oculos-site*
  advertise-content ()
  (let ((this-advert
         (make-instance 'mm-advertise :operation "advertise"
                        :sender (name *N*)
                        :receiver *CMUMatchMakerServer*
                        :advertisement "(advertise :name  advertisement
                          :name \"getInformation\"
                          :ontology \"weather\"
                          :requiredFieldCategories
                           (listof
                            (category
                             :name \"primary-keys\"
                             :fields (listof (cfield \"city\" \"string\"))
                             :attributes (listof )))
                          :optionalFieldCategories (listof )
                          :constraintFieldCategories  (listof )
                          :outputOnlyFieldCategories
                           (listof
                            (category
                             :name \"output\"
                             :fields (listof (cfield \"weather\" \"string\"))
                             :attributes (listof )))
                           :host ethel.sandia.gov
                           :port 5556)"))
         (this-dest *CMUANSServer*))
    (send-kqml this-dest this-advert)))
(adding-to-gene *oculos-site*
                (define-schema advertise-schema ()
    ()
    :do-if t
    :match-if nil
    :do (advertise-content)
    ))

;;(do-schema (make-instance 'advertise-schema))
```

```
(adding-to-gene *oculos-site*
  (define-schema kqml-schema ()
    (sender receiver content)
    :do-if nil
    :match-if t
    :match (kqml-message-percept (progn (aisl:diag schema percept)
                                        (maybe-join-equal sender (sender percept)
                                                          receiver (symbol-name (receiver percept))
                                                          content (content percept)
                                                          )))
    )
  (define-schema kqml-response-schema ()
    (sender receiver content)
    :do-if t ;(string-equal sender (name *N*))
    :match-if nil
    :do (let ((percept (make-instance 'kqml-tell
                         :sender (sender schema)
                         :receiver (receiver schema)
                         :content (content schema))))
          (send-message "IP-ARTICULOS-NUCLEUS"
                        (list "TCP-SEND-MSG"
                              ;; this has to be fixed, should really interact with ANS...
                              ;; which would require ascii things to be matched, not impossible though
                              :destination (list (pip:dotted-to-ipaddr "134.253.158.37") 6540)
                              :raw-data (serialize percept)
                              :final-fn #'(lambda () (complete schema))
                              :close? t))))

  (define-schema kqml-sequence-schema ()
    (sender receiver content)
    :subschemata (:kqml (kqml-schema :sender sender :receiver receiver :content content)
                  :kqml-respond (kqml-response-schema :sender receiver :receiver sender :content content))
    :dfa ((:start :transitions ((:kqml :one)))
          (:one :transitions  ((:kqml-respond :two)))
          (:two :signal :complete)))

  (defun *oculos-site* kqml-tell-message (sender receiver content)
    (let ((this-message
           (make-instance 'kqml-tell
                          :sender sender
                          :receiver receiver
                          :content content))
          (this-dest *LocalANSServer*))
      (send-kqml this-dest this-message)))
  (do-schema (make-instance 'kqml-sequence-schema))
;; )

  )
```

```
;; Code in this file implements structures for processing the Knowledge Query and
;; Manipulation Language (KQML) as perceptual stimuli and for transmitting messages
;; in that format.

(in-package :genome)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; KQML percept classes -- creating a message object that can be dispatched on
;; note that these are not all instantiable classes, only the non-mixin
;; classes should ever be created (that inherit from these mixins and
;; kqml-messag-percept)

(defgene-class *oculos-site* kqml-message-percept (material-percept)
  (;; (conversation-id :initform nil :initarg :conversation-id :accessor conversation-id)
   ;; all kqml messages have these three slots -- should the performative be a slot?
   (performative :initform nil :initarg :performative :accessor performative :allocation :class)
   (sender       :initform nil :initarg :sender       :accessor sender)
   (receiver     :initform nil :initarg :receiver     :accessor receiver)))

; any content-bearing message (all but 2) has these three slots -- the language and ontology
; describe the context and protocol which should be used to decode the contents
(defgene-class *oculos-site* kqml-content-mixin (kqml-message-percept)
  ((language     :initform nil :initarg :language     :accessor language)
   (ontology     :initform nil :initarg :ontology     :accessor ontology)
   (content      :initform nil :initarg :content      :accessor content)))

; a response object -- this should be used to undefer goals :pending-agent-response
(defgene-class *oculos-site* kqml-response-mixin (kqml-message-percept)
  ((in-reply-to  :initform nil :initarg :in-reply-to  :accessor in-reply-to)))

; a request object -- a useful counterpart to response objects (note that some objects
; can be both responses and requests, in the middle of a dialog or whatever)
(defgene-class *oculos-site* kqml-request-mixin (kqml-message-percept)
  ((reply-with   :initform nil :initarg :reply-with   :accessor reply-with)))

; a message whose contents can be emphasized -- agent X _KNOWS_ this to be true
(defgene-class *oculos-site* kqml-assertion-mixin (kqml-message-percept)
  ((force        :initform nil :initarg :force        :accessor force)))

; used for deletes and asks across a domain, specifies a filter to use
(defgene-class *oculos-site* kqml-aspect-mixin (kqml-message-percept)
               ((aspect       :initform nil :initarg :aspect       :accessor aspect)))
(defgene-class *oculos-site* kqml-operation-mixin (kqml-message-percept)
               ((operation    :initform nil :initarg :operation    :accessor operation)))
(defgene-class *oculos-site* mm-advert-mixin (kqml-message-percept)
               ((advertise    :initform nil :initarg :advertisement :accessor advertisement)))

;;;;
```

```
;; individual performative classes that can be created by the parser
;; NOTE: kqml content mixin is later precedence than the others so that
;;       methods may specialize (on responses, say) that override the more general
;;       but not necessarily required content mixin class
;;;;

(defgene-class *oculos-site* kqml-tell (kqml-response-mixin kqml-request-mixin kqml-assertion-mixin kqml-content-mixin)
  ((performative :initform 'tell :accessor performative :allocation :class)))

(defgene-class *oculos-site* kqml-achieve (kqml-assertion-mixin kqml-content-mixin)
  ((performative :initform 'achieve :accessor performative :allocation :class)))

(defgene-class *oculos-site* kqml-deny (kqml-response-mixin kqml-content-mixin)
  ((performative :initform 'deny :accessor performative :allocation :class)))

(defgene-class *oculos-site* kqml-untell (kqml-response-mixin kqml-assertion-mixin kqml-content-mixin)
  ((performative :initform 'untell :accessor performative :allocation :class)))

(defgene-class *oculos-site* kqml-insert (kqml-response-mixin kqml-request-mixin kqml-assertion-mixin kqml-content-mixin)
  ((performative :initform 'insert :accessor performative :allocation :class)))

(defgene-class *oculos-site* kqml-delete (kqml-response-mixin kqml-request-mixin kqml-content-mixin)
  ((performative :initform 'delete :accessor performative :allocation :class)))

(defgene-class *oculos-site* kqml-delete-one (kqml-response-mixin kqml-request-mixin kqml-aspect-mixin kqml-content-mixin)
  ((performative :initform 'delete-one :accessor performative :allocation :class)
   (order        :initform nil :initarg :order        :accessor order)
   )
  )

(defgene-class *oculos-site* kqml-delete-all (kqml-response-mixin kqml-request-mixin kqml-aspect-mixin kqml-content-mixin)
  ((performative :initform 'delete-all :accessor performative :allocation :class)))

(defgene-class *oculos-site* kqml-error (kqml-response-mixin)
  ((performative :initform 'error :accessor performative :allocation :class)
   (comment      :initform nil :initarg :comment      :accessor comment)
   (code         :initform nil :initarg :code         :accessor code)
   )
  )

(defgene-class *oculos-site* kqml-sorry (kqml-response-mixin)
  ((performative :initform 'sorry :accessor performative :allocation :class)
   (comment      :initform nil :initarg :comment      :accessor comment)
   )
  )

(defgene-class *oculos-site* kqml-evaluate (kqml-request-mixin kqml-content-mixin)
  ((performative :initform 'evaluate :accessor performative :allocation :class)))
```

```
(defgene-class *oculos-site* kqml-reply (kqml-response-mixin kqml-assertion-mixin kqml-content-mixin)
  ((performative :initform 'reply :accessor performative :allocation :class)))

(defgene-class *oculos-site* kqml-ask-if (kqml-request-mixin kqml-content-mixin)
  ((performative :initform 'ask-if :accessor performative :allocation :class)))

(defgene-class *oculos-site* kqml-ask-about (kqml-request-mixin kqml-content-mixin)
  ((performative :initform 'ask-about :accessor performative :allocation :class)))

(defgene-class *oculos-site* kqml-ask-one (kqml-request-mixin kqml-aspect-mixin kqml-content-mixin)
  ((performative :initform 'ask-one :accessor performative :allocation :class)))

(defgene-class *oculos-site* kqml-ask-all (kqml-request-mixin kqml-aspect-mixin kqml-content-mixin)
               ((performative :initform 'ask-all :accessor performative :allocation :class)))
(defgene-class *oculos-site* mm-advertise (kqml-operation-mixin kqml-mmadvert-mixin)
               ((performative :initform 'tell :accessor performative :allocation :class)
                (operation    :initform 'advertise :accessor operation :allocation :class)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; parsing functions for building percepts from disambigutated kqml
;; messages based on performative (first element of list in message)
(defgene-method *oculos-site* find-kqml-percept ((kqml list))
                ;;The first chunk in the list is the type of message.  The rest of the message
                ;;is a keyword and then either a value or nil
                (let ((the-class (find-class (find-symbol
                                              (concatenate 'string "KQML-" (symbol-name (car kqml)))
                                              (locus *N*))))
                      (the-parameters (get-kqml-params (cdr kqml))))
                  (apply 'make-instance the-class the-parameters)))

(defgene-method *oculos-site* get-kqml-params ((kqml list))
  (maplist #'(lambda (x) (if (and  (keywordp (car x)) (or (keywordp (cadr x)) (null (cdr x))))
                             (setf kqml (delete (car x) kqml :count 1))))  kqml)
 kqml)

(defgene-method *oculos-site*
  elaborate ((stim-type (eql :kqml)) (stimulus unit-stimulus))
  (let ((*read-eval* nil)
        (*package* (locus *N*)))
    (find-kqml-percept (read-from-string (content stimulus)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; associate method for messages that have direct links to their schema

;; i'm notioning that find-schema on a direct id would be fast, otherwise
;; find-schema would have to look at a bunch of data or look through
;; expectations or active schema to see what the input is appropriate to
;; based on sender or whatever
(defgene-method *oculos-site* match-directly ((the-percept kqml-message-percept))
```

```
  (schemap (in-reply-to the-percept)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defgene-method *ip-exoculos-site*
  delimit ((stream-type (eql :kqml)) mstream)
  (if (loop for next-char = (aisl::stream-read-char-no-hang mstream)
        until (or (null next-char) (eq next-char :eof))
        do (if (char= next-char #\^d) (return t)))
      (values (subseq (aisl::buffer mstream) 0) :success)
    (values (subseq (aisl::buffer mstream) 0) :undelimited)))

;; capstones which return nil when a message doesn't have such a slot (for now)
(defgene-method *oculos-site* language ((kqml kqml-message-percept)))
(defgene-method *oculos-site* ontology ((kqml kqml-message-percept)))
(defgene-method *oculos-site* content ((kqml kqml-message-percept)))
(defgene-method *oculos-site* in-reply-to ((kqml kqml-message-percept)))
(defgene-method *oculos-site* reply-with ((kqml kqml-message-percept)))
(defgene-method *oculos-site* force ((kqml kqml-message-percept)))
(defgene-method *oculos-site* aspect ((kqml kqml-message-percept)))
(defgene-method *oculos-site* order ((kqml kqml-message-percept)))
(defgene-method *oculos-site* comment ((kqml kqml-message-percept)))
(defgene-method *oculos-site* code ((kqml kqml-message-percept)))

; get rid of package delimiter or ""s around performative
; make sure oids etc can be read by their system (may have to eliminate some spaces)
(defgene-method *oculos-site* serialize ((msg kqml-message-percept))
  (let ((sm nil))
    (macrolet ((maybe-include-as (keyword slotname)
                 `(when (,slotname msg)
                    (setf sm (list* (,slotname msg) ,keyword sm)))))
      (maybe-include-as :sender sender)
      (maybe-include-as :receiver receiver)
      (maybe-include-as :language language)
      (maybe-include-as :ontology ontology)
      (maybe-include-as :content content)
      (maybe-include-as :in-reply-to in-reply-to)
      (maybe-include-as :reply-with reply-with)
      (maybe-include-as :force force)
      (maybe-include-as :aspect aspect)
      (maybe-include-as :order order)
      (maybe-include-as :comment comment)
      (maybe-include-as :code code)
      (setf sm (reverse sm)))
    (format nil "(~a~{ ~s~})" (performative msg) sm)))

(defgene-fun *oculos-site* send-kqml (destination percept)
  (send-message "IP-ARTICULOS-NUCLEUS"
                (list "TCP-SEND-MSG"
                      :destination destination
```

```
        :raw-data (serialize percept)
        :close? t)))
```

This page intentionally left blank.

Appendix IV. Reprint of CMU reference that defines LARKS

This page intentionally left blank.

Interoperability among Heterogeneous Software Agents on the Internet

Katia Sycara, Jianguo Lu, and Matthias Klusch

CMU-RI-TR-98-22

The Robotics Institute
Carnegie Mellon University
Pittsburgh, USA.

October 1998

This research has been sponsored in part by Office of Naval Research grant N-00014-96-16-1-1222.

Contents

List of Figures

1 Introduction

Due to the exponential increase of offered services in the most famous offspring of the Internet, the World Wide Web, searching and selecting relevant services is essential for users. Various search engines and software agents providing various different services are already deployed on the Web. However, novice users of the Web may have no idea where to start their search, where to find what they really want, and what agents are available for doing their job. Even experienced users may not be aware of every change in the Web, e.g., relevant web pages might not exist or their content be valid anymore, and agents may appear and disappear over time. The user is simply overtaxed by manually searching in the Web for information or appropriate agents.

On the other hand, as the number and sophistication of agents on the Web that may have been developed by different designers increases, there is an obvious need for a standardized, meaningful communication among agents to enable them to perform collaborative task execution. We distinguish two general agent categories, service providers and service requester agents. Service providers provide some type of service, such as finding information, or performing some particular domain specific problem solving (e.g. number sorting). Requester agents need provider agents to perform some service for them. Since the Internet is an open environment, where information sources, communication links and agents themselves may appear and disappear unpredictably, there is a need for some means to help requester agents find providers. Agents that help locate others are called *middle agents*.

We have identified different types of middle agents in the Internet, such as matchmakers (yellow page services), brokers, billboards, etc. [3], and experimentally evaluated different protocols for interoperation between providers, requesters and various types of middle agents. Figure 1 shows the protocol for two different types of middle agents: brokers and matchmakers. We have also developed protocols for distributed matchmaking [12]. The process of finding an appropriate provider through a middle agent is called *matchmaking*. It has the following general form:

- Provider agents advertise their capabilities such as know-how, expertise, and so on, to middle agents.
- Middle agents store these advertisements.
- A requester asks some middle agent whether it knows of providers with desired capabilities.
- The middle agent matches the request against the stored advertisements and returns the result.

While this process at first glance seems very simple, it is complicated by the fact that providers and requesters are usually heterogeneous and incapable in general of understanding each other. This difficulty gives rise to the need for a

common language for describing the capabilities and requests of software agents in a convenient way. In addition, one has to devise a mechanism for matching descriptions in that language. This mechanism can then be used by middle agents to efficiently select relevant agents for some given tasks.

In the following, we first elaborate the desiderata of an agent capability description language (ACDL), and propose such an ACDL, called LARKS, in detail. Then we will discuss the matchmaking process using LARKS and give a complete working scenario with some examples. We have implemented LARKS and the associated powerful matchmaking process, and are currently incorporating it within our RETSINA multi-agent infrastructure framework [22]. The paper concludes with comparing our language and the matchmaking process with related works.

2 Matchmaking Among Heterogeneous Agents

In the process of matchmaking (see Fig. 1) are three different kinds of collaborating agents involved:

1. **Provider** agents provide their capabilities, e.g., information search services, retail electronic commerce for special products, etc., to their users and other agents.

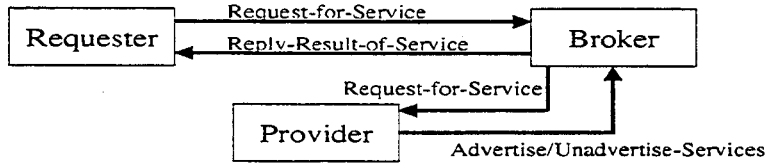
2. **Requester** agents consume informations and services offered by provider agents in the system. Requests for any provider agent capabilities have to be sent to a matchmaker agent.

3. **Matchmaker** agents mediate among both, requesters and providers, for some mutually beneficial cooperation. Each provider must first register himself with a matchmaker. Provider agents advertise their capabilities (advertisements) by sending some appropriate messages describing the kind of service they offer.

   Every request a matchmaker receives will be matched with his actual set of advertisements. If the match is successful the matchmaker returns a ranked set of appropriate provider agents and the relevant advertisements to the requester.

In contrast to a broker agent, a matchmaker does not deal with the task of contacting the relevant providers, transmitting the service request to the service provider and communicate the results to the requester. This avoids data transmission bottlenecks, but it might increase the amount of interactions among agents.

☛ *Brokering*



☛ *Matchmaking*

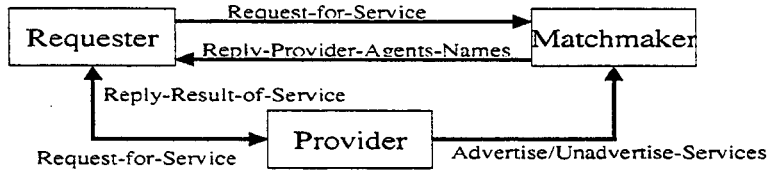


Figure 1: Service Brokering vs. Matchmaking

2.1 Desiderata for an Agent Capability Description Language

There is an obvious need to describe agent capabilities in a common language before any advertisement, request or even matchmaking among the agents can take place. In fact, the formal description of capabilities is one of the major problems in the area of software engineering and AI. Some of the main desired features of such a agent capability description language are the following.

- **Expressiveness.**
  The language is expressive enough to represent not only data and knowledge, but also to describe the meaning of program code. Agent capabilities are described at an abstract rather than implementation level. Most of existing agents can be distinguished by describing their capabilities in this language.

- **Inferences.**
  Inferences on descriptions written in this language are supported. A user can read any statement in the language, and software agents are able to process, especially to compare any pair of statements automatically.

- **Ease of Use.**
  Every description should not only be easy to read and understand, but also easy to write by the user. The language supports the use of domain or common ontologies for specifying agents capabilities. It avoids redundant work for the user and improves the readability of specifications.

- **Application in the Web.**
  One of the main application domains for the language is the specification of advertisements and requests of agents in the Web. The language allows for automated exchange and processing of information among these agents.

In addition, the matchmatching process on a given set of capability descriptions and a request, both written in the chosen ACDL, should be efficient, most accurate, not only rely on keyword extraction and comparison, and fully automated.

3 The Agent Capability Description Language LARKS

Representing capabilities is a difficult problem that has been one of the major concerns in the areas of software engineering, AI, and more recently, in the area of internet computing. There are many program description languages, like VDM or Z[28], to describe the functionalities of programs. These languages concern too much detail to be useful for the searching purpose. Also, reading and writing specifications in these languages require sophisticated training. On the other hand, the interface definition languages, like IDL, WIDL, go to the other extreme by omitting the functional descriptions of the services at all. Only the input and output information are provided.

In AI, knowledge description languages, like KL-ONE, or KIF are meant to describe the knowledge instead of the actions of a service. The action representation formalisms like STRIPS are too restrictive to represent complicated service. Some agent communication languages like KQML and FIPA concentrate on the communication protocals (message types) between agents but leave the content part of the language unspecified.

In internet computing, various description format are being proposed, notably the WIDL and the Resource Description Framework(RDF)[27]. Although the RDF also aims at the interoperablity between web applications, it is rather intended to be a basis for describing metadata. RDF allowes different vendors to describe the properties and relations between resources on the Web. That enables other programs, like Web robots, to easily extract relevant information, and to build a graph structure of the resources available on the Web, without the need to give any specific information. However, the description does not describe the functionalities of the Web *services*.

Since none of those languages satisfies our requirements, we propose an ACDL, called LARKS (**L**anguage for **A**dvertisement and **R**equest for **K**nowledge **S**haring) that enables for advertising, requesting and matching agent capabilities. It satisfies the desiderata given in the former section.

3.1 Specification in LARKS

A specification in LARKS is a frame with the following slot structure.

`Context`	Context of specification
`Types`	Declaration of used variable types
`Input`	Declaration of input variables
`Output`	Declaration of output variables
`InConstraints`	Constraints on input variables
`OutConstraints`	Constraints on output variables
`ConcDescriptions`	Ontological descriptions of used words

The frame slot types have the following meaning.

- `Context`.
  The context of the specification in the local domain of the agent.

- `Types`.
  Optional definition of the used data types. If not used, all data types are assumed to be defined in the following slots for input and output variables.

- `Input` and `Output`.
  Input/output variables for required input/output knowledge to describe a capability of an agent: if the input given to an agent fits with the specified input declaration part, then the agent is able to process an output as specified in the output declaration part. Processing takes all specified constraints on the input and output variables into consideration.

- `InConstraints` and `OutConstraints`.
  Logical constraints on input/output variables in the input/output declaration part. The constraints are specified as Horn clauses.

- `ConcDesriptions`.
  Optional description of the meaning of words used in the specification. The description relies on concepts in a given local domain ontology. Attachement of a concept C to a word w in any of the slots above is done in the form: w*C. That means that the concept C is the ontological description of the word w. The concept C is included in the slot `ConcDescription`.

In our current implementation we assume each local domain ontology to be written in the concept language ITL (Information Terminological Language). the syntax and semantics of the ITL are given in the appendix. Section 3.3 gives an example for how to attach concepts in a LARKS specification, and also shows an example domain ontology in ITL. A generic interface for using ontologies in LARKS expressed in languages other than ITL will be implemented in near future.

Every specification in LARKS can be interpreted as an advertisement as well as a request; this depends on the purpose for which an agent sends a specification to some matchmaker agent(s). Every LARKS specification must be wrapped up in an appropriate KQML message by the sending agent indicating if the message content is to be treated as a request or an advertisement.

3.2 Examples of Specifications in LARKS

The following two examples show how to describe in LARKS the capability to sort a given list of items, and return the sorted list. Example 3.1 is the the specification of the capability to sort a list of at most 100 integer numbers, whereas in example 3.2 a more generic kind of sorting real numbers or strings is specified in LARKS. Note that the `ConcDescriptions` slot is empty, i.e. the semantics of the words in the specification are assumed to be known to the matchmaker

Example 3.1: *Sorting integer numbers*

IntegerSort	
`Context`	Sort
`Types`	
`Input`	xs: `ListOf Integer`;
`Output`	ys: `ListOf Integer`;
`InConstraints`	le(length(xs),100);
`OutConstraints`	before(x,y,ys) <- ge(x,y); in(x,ys) <- in(x,xs);
`ConcDescriptions`	

Example 3.2: *Generic sort of real numbers or strings*

GenericSort	
`Context`	Sorting
`Types`	
`Input`	xs: `ListOf` Real | String;
`Output`	ys: `ListOf` Real | String;
`InConstraints`	

`OutConstraints`	before(x,y,ys) <- ge(x,y);
	before(x,y,ys) <- preceeds(x,y);
	in(x,ys) <- in(x,xs);

`ConcDescriptions`	

The next example is a specification of an agent's capability to buy stocks at a stock market. Given the name of the stock, the amount of money available for buying stocks and the shares for one stock, the agent is able to order stocks at the stock market. The constraints on the order are that the amount for buying stocks given by the user covers the shares times the current price for one stock. After performing the order the agent will inform the user about the stock, the shares, and the gained benefit.

Example 3.3: *Selling stocks by a portfolio agent*

sellStock	
`Context`	Stock, StockMarket;
`Types`	
`Input`	symbol: StockSymbols; yourMoney: Money; shares: Money;
`Output`	yourStock: StockSymbols; yourShares: Money; yourChange: Money;
`InConstraints`	yourMoney >= shares*currentPrice(symb);
`OutConstraints`	yourChange = yourMoney - shares*currentPrice(symb); yourShares = shares; yourStock = symbol;
`ConcDescriptions`	

o

3.3 Using Domain Knowledge in LARKS

As mentioned before, LARKS offers the option to use application domain knowledge in any advertisement or request. This is done by using a local ontology for describing the meaning of a word in a LARKS specification. Local ontologies can be formally defined using, e.g., concept languages such as ITL (see Appendix), BACK, LOOM, CLASSIC or KRIS, a full-fledged first order predicate logic, such as the knowledge interchange format (KIF), or even the unified modeling language (UML).

The main benefit of that option is twofold: (1) the user can specify in more detail what he is requesting or advertising, and (2) the matchmaker agent is able to make automated inferences on such kind of additional semantic descriptions while matching LARKS specifications, thereby improving the overall quality of matching.

Example 3.4: *Finding informations on computers*

Suppose that a provider agent such as, e.g., HotBot, Excite, or even a meta-searchbot, like SavvySearch or MetaCrawler, advertises the capability to find informations about any type of computers. The administrator of the agent may specify that capability in LARKS as follows.

FindComputerInfo	
Context	Computer*Computer;
Types	InfoList = ListOf(model: Model*ComputerModel, brand: Brand*Brand, price: Price*Money, color: Color*Colors);
Input	brands: SetOf Brand*Brand; areas: SetOf State; processor: SetOf CPU*CPU; priceLow*LowPrice: Integer; priceHigh*HighPrice: Integer;
Output	Info: InfoList;
InConstraints	
OutConstraints	sorted(Info).
ConcDescriptions	Computer = (and Product (exists has-processor CPU) (all has-memory Memory) (all is-model ComputerModel)); LowPrice = (and Price (ge 1800)(exists in-currency aset(USD))); HighPrice = (and Price (le 50000)(exists in-currency aset(USD))); ComputerModel = aset(HP-Vectra,PowerPC-G3,Thinkpad770,Satellite315); CPU = aset(Pentium,K6,PentiumII,G3,Merced) [Product, Colors, Brand, Money]

Most words in this specification have been attached with a name of some concept out of a given ontology. The definitions of these concepts are included in the slot ConcDescriptions. Concept definitions which were already sent to the matchmaker are enclosed in brackets. In this example we assume the underlying ontology to be written in the concept language ITL. An example for such an ontology is given in the next section.

Suppose that an agent registers himself at some matchmaker agent and sends the above specifications as advertisements. The matchmaker will then treat that agent as a provider agent, i.e., an agent who is capable to provide all these kinds of services.

3.3.1 Example for a Domain Ontology in the Concept Language ITL

As mentioned before, our current implementation of LARKS assumes the domain ontology to be written in the concept language ITL.

The research area on concept languages (or description logics) in AI has its origins in the theoretical deficiencies of semantic networks in the late 70's. KL-ONE was the first concept language providing a well-founded semantic for a more native language-based description of knowledge. Since then different concept languages are intensively investigated; they are almost decidable fragments of first-order predicate logic. Several knowledge representation and inference systems, such as CLASSIC, BACK, KRIS, or CRACK, based on such languages are available.

Conceptual knowledge about a given application domain, or even common-

sense, is defined by a set of concepts and roles as terms in the given concept language; each term as a definition of some concept C is a conjunction of logical constraints which are necessary for any object to be an instance of C. The set of terminological definitions forms a *terminology*. Any canonical definition of concepts relies in particular on a given basic vocabulary of words (primitive components) which are not defined in the terminology, i.e., their semantic is assumed to be known and consistently used across boundaries.

The following terminology, is written in the concept language ITL and defines concepts in the computer application domain. It is in particular used in the example 3.4 in the former section.

Product	=	(and (all is-manufactured-by Brand) (atleast 1 is-manufactured-by) (all has-price Price))
Computer	=	(and Product (exists has-processor CPU) (all has-memory Memory) (all is-model ComputerModel))
Notebook	=	(and Computer (all has-price (and (and (ge 1000) (le 2999)) (all in-currency aset(USD))) (all has-weight (and kg (le 5)) (all is-manufactured-by Company)) (all is-model aset(Thinkpad380,Thinkpad770,Satellite315))))
Brand	=	(and Company (all is-located-in State))
State	=	(and (all part-of Country) aset(VA,PA,TX,OH,NY))
Company	=	aset(IBM,Toshiba,HP,Apple,DEC,Dell,Gateway)
Colors	=	aset(Blue,Green,Yellow,Red)
Money	=	(and Real (all in-currency aset(USD,DM,FF,Y,P)))
Price	=	Money
LowPrice	=	(and Price (ge 1800)(exists in-currency aset(USD))),
HighPrice	=	(and Price (le 50000)(exists in-currency aset(USD)))
ComputerModel	=	aset(HP-Vectra,PowerPC-G3,Thinkpad380,Thinkpad770,Satellite315)
CPU	=	aset(Pentium,K6,PentiumII,G3,Merced)

○

3.3.2 Subsumption Relationships Among Concepts

One of the main inferences on ontologies written in concept languages is the computation of the *subsumption* relation among two concepts: A concept C subsumes another concept C' if the extension of C' is a subset of that of C. This means, that the logical constraints defined in the term of the concept C' logically imply those of the more general concept C.

Any concept language is decidable if it is for concept subsumption among two concepts defined in that language. The concept language ITL we use is NP-complete decidable. The well-known trade-off between expressiveness and tractability of concept languages in practice is surrounded almost by subsumption algorithms which are correct but incomplete. We use an incomplete inference algorithm for computing subsumption relations among concepts in ITL.

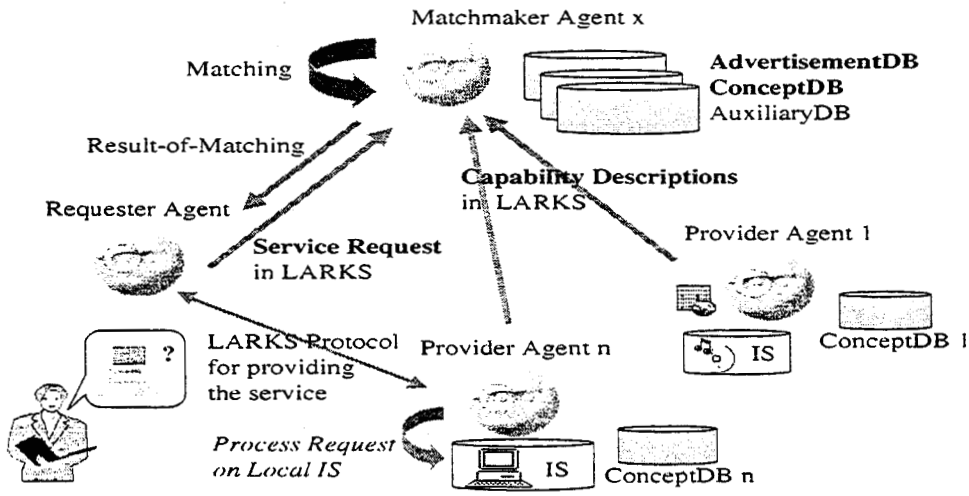


Figure 2: Matchmaking using LARKS: An Overview

For the mechanism of subsumption computation we refer the reader to, e.g., [19, 14, 20, 21].

The computation of subsumption relationships among all concepts in a ontology yields a so-called concept hierarchy. Both, the subsumption computation and the concept hierarchy are used in the matchmaking process (see section 4.1.2).

4 The Matchmaking Process Using LARKS

As mentioned before, we differentiate between three different kinds of collaborating information agents: provider, requester and matchmaker agents. The following figure shows an overview of the matchmaking process using LARKS.

The matchmaker agent process a received request in the following main steps:

- Compare the request with all advertisements in the advertisement database.
- Determine the provider agents whose capabilities match best with the request. Every pair of request and advertisement has to go through several different filtering during the matchmaking process.
- Inform the requesting agent by sending them the contact addresses and related capability descriptions of the relevant provider agents.

For being able to perform a steady, just-in-time matchmaking process the information model of the matchmaker agent comprises the following components.

1. *Advertisement database* (ADB).
   This database contains all advertisements written in LARKS the matchmaker receives from provider agents.

2. *Partial global ontology.*
   The ontology of the matchmaker consists of all ontological descriptions of words in advertisements stored in the ADB. Such a description is included in the slot `ConcDescriptions` and sent to the matchmaker with any advertisement.

3. *Auxiliary database.*
   The auxiliary data for the matchmaker comprise a database for word pairs and word distances, basic type hierarchy, and internal data.

Please note that the ontology of a matchmaker agent is not necessarily equal to the union of local domain ontologies of all provider agents who are actually registered at the matchmaker. This also holds for the advertisement database. Thus, a matchmaker agent has only partial global knowledge on available information in the overall multi-agent system; this partial knowledge might also be not up-to-date concerning the actual time of processing incoming requests. This is due to the fact that for efficiency reasons changes in the local ontology of an provider agent will not be propagated immediately to all matchmaker agents he is registered at. In the following we will describe the matchmaking process using LARKS in a more detail.

4.1 The Filtering Stages of the Matchmaking Process

The matching process of the matchmaker is designed with respect to the following criteria:

- The matching should *not be based on keyword retrieval only.* Instead, unlike the usual free text search engines, the semantics of requests and advertisements should be taken into consideration.

- The matching process should be *automated.* A vast amount of agents appear and disappear in the Internet. It is nearly impossible for a user to manually search or browse all agents capabilities.

- The matching process should be *accurate.* For example, if the matches returned by the match engine are claimed to be exact match or the plug-in match, those matches should satisfy the definitions of exact matching and plug-in matching.

- The matching process should be *efficient*, i.e., it should be fast.

- The matching process should be *effective*, i.e., the set of matches should not be too large. For the user, typing in a request and receiving hundreds of matches is not necessarily very useful. Instead, we prefer a small set of highly rated matches to a given request.

To fulfill the matching criteria listed in the above section, the matching process is organized as a series of increasingly stringent filters on candidate agents. That means that matching a given request into a set of advertisements consists of the following five filters that we organize in three consecutive filtering stages:

1. *Context Matching*

   Select those advertisements in the ADB which can be compared with the request in the same or similar context.

2. *Syntactical Matching*

   This filter compares the request with any advertisement selected by the context matching in three steps:

   (a) Comparison of profiles.

   (b) Similarity matching.

   (c) Signature matching.

   The request and advertisement profile comparison uses a weighted keyword representation for the specifications and a given term frequency based similarity measure (Salton, 1989). The last two steps focus on the (input/output) constraints and declaration parts of the specifications.

3. *Semantical Matching*

   This final filter checks if the input/output constraints of any pair of request and advertisement logically match (see section 4.1.5).

For reasons of efficiency the context filter roughly prunes off advertisements which are not relevant for a given request. In the following two filtering stages, syntactical and semantical matching, the remaining advertisements in the ADB of the matchmaker are checked in a more detail. All filters are independent from each other; each of them narrows the set of matching candidates with respect to a given filter criteria.

In our current implementation the matchmaker offers different types and modes of matching a request to a given set of advertisements.

4.1.1 Different Types of Matching in LARKS

Agent capability matching is the process of determining whether an advertisement registered in the matchmaker matches a request. But when can we say two descriptions *match* against each other? Does it mean that they have the same text? Or the occurrence of words in one discription sufficiently overlap with those of another discription? When both descriptions are totally different in text, is it still possible for them to match? Even if they match in a given sense, what can we then say about the matched advertisements? Before we go into the details of the matchmaking process, we should clarify the various notions of matches of two specifications.

4.1.1.1 Exact Match Of course, the most accurate match is when both descriptions are equivalent, either equal literally, or equal by renaming the variables, or equal logically obtained by logical inference. This type of matching is the most restrictive one.

4.1.1.2 Plug-In Match A less accurate but more useful match is the so-called $plug - in$ match. Roughly speaking, plug-in matching means that the agent which capability description matches a given request can be "plugged into the place" where that request was raised. Any pair of request and advertisement can differ in the signatures of their input/output declarations, the number of constraints, and the constraints themselves. As we can see, exact match is a special case of plug-in match, i.e., wherever two descriptions are exact match, they are also plug-in match.

A simple example of a plug-in match is that of the match between a request to sort a list of integers and an advertisement of an agent that can sort both list of integers and list of strings. This example is elaborated in section 5. Another example of plug-in match is between the request to find some computer information without any constraint on the output and the advertisement of an agent that can provide these informations and sorts the respective output.

4.1.1.3 Relaxed Match The least accurate but most useful match is the so-called *relaxed* match. A relaxed match has a much more weaker semantic interpretation than a exact match and plug-in match. In fact, relaxed match will not tell whether two descriptions semantically match or not. Instead it determines how close the two descriptions are by returning just a numerical distance value. Two descriptions match if the distance value is smaller than a preset threshold value. Normally the plug-in match and the exact match will be a special case of the relaxed match if the threshold value is not too small.

An example of a relaxed match is that of the request to find the place (or address) where to buy a Compaq Pentium233 computer and the capability description of an agent that may provide the price and contact phone number for that computer dealer.

Different users in different situation may want to have different types of matches. Although people usually may prefer to have plug-in matches, such a kind of match does not exist in many cases. Thus, people may try to see the result of a relaxed match first. If there is a sufficient number of relaxed matches returned a refined search may be performed to locate plug-in matching advertisements. Even when people are interested in a plug-in match for their requests only, the computational costs for this type of matching might outweigh its benefits.

As mentioned above we have five different matching filters:

1. context matching
2. profile comparison

3. similarity matching

4. signature matching

5. semantical matching

The first three filters are meant for relaxed matching, and the signature and semantical matching filter are meant for plug-in matching. Please note, that the computational costs of these filters are in increasing order. Users may select any combinations of these filters according their demand. Since the similarity filter also performs intensive computation one may just select the context filter and the profile filter if efficiency is of major concern.

Based on the given notions of matching we did implement four different modes of matching for the matchmaker:

1. **Complete Matching Mode**. All filtering stages are considered.

2. **Relaxed Matching Mode**. The first two filtering stages are considered except signature matching, i.e., the context, profile and similarity filter only.

3. **Profile Matching Mode**. Only the context matching and comparison of profiles is done.

4. **Plug-In Matching Mode**. In this mode, the matchmaker performs the signature and semantical matching.

As said above, the matching process proceeds in different filtering stages. If the considered advertisement and request contain conceptual attachments (ontological description of used words), then in most of the filtering stages (except for the comparison of profiles) we need a way to determine the semantic distance between the defined concepts. For that we use the computation of subsumption relationships and a weighted associative network.

4.1.2 Computation of Semantic Distances Among Concepts

We have presented the notion of concept subsumption in section 3.3.2. But the concept subsumption gives only a generalization/specialization relation based on the definition of the concepts via roles and attribute sets. In particular for matchmaking the identification of additional relations among concepts is very useful because it leads to a deeper semantic understanding. Moreover, since the expressivity of the concept language ITL is restrictive so that performance can be enhanced, we need some way to express additional associations among concepts.

For this purpose we use a so-called weighted associative network, that is a semantic network with directed edges between concepts as nodes. Any edge denotes the kind of a binary relation among two concepts, and is labeled in addition with a numerical weight (interpreted as a fuzzy number). The weight

indicates the strength of belief in that relation, since its real world semantics may vary[1]. We assume that the semantic network consists of three kinds of binary, weighted relationships: (1) generalization, (2) specialization (as inverse of generalization), and (3) positive association among concepts (Fankhauser et al., 1991). The *positive association* is the most general relationship among concepts in the network indicating them as synonyms in some context. Such a semantic network is called an *associative network* (AN).

In our implementation we create an associative network by using the concept hierarchy of a given terminology defined in the concept language ITL. All subsumption relations in this concept hierarchy are used for setting the generalization and specialization relations among concepts in the corresponding associative network. Positive associations may be set by the administrator or user. Positive association, generalization and specialization are transitive.

As mentioned above, every edge in the associative network is labeled with a fuzzy weight. These weights are set by the user or automatically by default. The distance between two concepts in an associative network is then computed as the strength of the shortest path among them. Combining the strength of each relation in this path is done by using the following triangular norms for fuzzy set intersections (Kruse et al., 1991):

$$\begin{aligned} \tau_1(\alpha,\beta) &= max\{0, \alpha+\beta-1\} \quad n=-1 \\ \tau_2(\alpha,\beta) &= \alpha\cdot\beta \quad n=0 \\ \tau_3(\alpha,\beta) &= min\{\alpha,\beta\} \quad n=\infty \end{aligned}$$

Since we have three different kinds of relationships among two concepts in an AN the kind and strength of a path among two arbitrary concepts in the network is determined as shown in the following tables. For a formal discussion of that issue we refer to the work of Fankhauser et al. (1991), Kracker (1992), and Fankhauser and Neuhold (1992).

	g	s	p
g	g	p	p
s	p	s	p
p	p	p	p

Table 1: Kind of paths in an AN.

	g	s	p
g	τ_3	τ_1	τ_2
s	τ_1	τ_3	τ_2
p	τ_2	τ_2	τ_2

Table 2: Strength of paths in an AN.

For all $0 \leq \alpha, \beta \leq 1$ holds that $\tau_1(\alpha,\beta) \leq \tau_2(\alpha,\beta) \leq \tau_3(\alpha,\beta)$. Each triangular norm is monotonic, commutative and associative, and can be used as axiomatic sceletons for fuzzy set intersection. We restrict ourselves to a pessimistic, neutral, and optimistic t-norm τ_1, τ_2 and τ_3, respectively.

Since these triangular norms are not mutually associative the strength of a path in an associative network depends on the direction of strength composition. This asymmetry in turn might lead to unintuitive derived results: Consider, e.g., a path consisting of just three relations among four concepts C_1, C_2, C_3, C_4 with

[1]The relationships are fuzzy, and one cannot possibly associate all concepts with each other.

$C_1 \Rightarrow_{g,0.6} C_2 \Rightarrow_{g,0.8} C_3 \Rightarrow_{p,0.9} C_4$. It holds that $\tau_2(\tau_3(0.6, 0.8), 0.9) = 0.54$, but the strength of the same path in opposite direction is $\tau_2(\tau_2(0.9, 0.8), 0.6) = 0.43$. According to Fankhauser and Neuhold (1992) we can avoid this asymmetry by imposing a precedence relation ($3 > 2 > 1$) for strength combination (see Table 3).

	g	s	p
g	2	3	1
s	1	2	1
p	1	1	3

Table 3: Computational precedence for the strength of a path.

The computation of semantic distances among concepts is used in most of the filtering stages of the matching process. We will now describe each of the filters in detail.

4.1.3 Context Matching

It is obvious that any matching of two specifications has to be in an appropriate context. Suppose a provider agent advertises to sell several different types of products, like cars, computers, shoes, etc. Further assume that all his advertisements include the only input variable declaration: brand: `SetOf` Brand; But what is meant by the type 'Brand' in the context of any specification of a capability of finding a *particular* item? Without any additional knowledge about the particular context, a request to find information about a particular item, like computers, would match with *all* product advertisements.

In LARKS there are two possibilities to deal with this problem which is connected to the well-known ontological mismatch problem. First, the `Context` slot in a specification S contains a (list of) words denoting the domain of discourse for matching S with any other specification. When comparing two specifications it is assumed that their domains, means their context, are the same (or atleast sufficiently similar) as long as the real-valued distances between these words do not exceed a given threshold[2]. The matching process only proceeds if that is true.

Second, every word in a LARKS specification may be associated with a concept in a given domain ontology. Again, if the context of both specifications turned out to be sufficiently similar in the step before then the concept definitions describe the meaning of the words they are attached to in a more detail in the same domain. In this case, two concepts with same name but different definitions will be stored separately by extending each concept name by the identifier of the agent who did send this concept.

To summarize, the context matching consists of two consecutive steps:

[2] Any distance between two words is computed by an appropriate word distance function using the auxiliary database of the matchmaker.

1. For every pair of words u, v given in the **context** slots compute the real-valued word distances $d_w(u, v) \in [0.1]$. Determine the most similar matches for any word u by selecting words v with the minimum distance value $d_w(u, v)$. These distances must not exceed a given threshold.

2. For every pair of most similar matching words, check that the semantic distance among the attached concepts does not exceed a given threshold.

4.1.4 Syntactical Matching

4.1.4.1 Comparison of Profiles The comparison of two profiles relies on a standard technique from the Information Retrieval area, called term frequency-inverse document frequency weighting (TF-IDF) (see Salton, 1989). According to that, any specification in LARKS is treated as a document.

Each word w in a document Req is weighted for that document in the following way. The number of times w occurs throughout all documents is called the document frequency $df(w)$ of w. The used collection of documents is not unlimited, such as the advertisement database of the matchmaker.

Thus, for a given document d, the relevance of d based on a word w is proportional to the number $wf(w, d)$ of times the word w occurs in d and inverse proportional to $df(w)$. A weight $h(w, d)$ for a word in a document d out of a set D of documents denotes the significance of the classification of w for d, and is defined as follows:

$$h(w, d) = wf(w, d) \cdot log(\frac{|D|}{df(w)}).$$

The weighted keyword representation $wkv(d, V)$ of a document d contains for every word w in a given dictionary V the weight $h(w, d)$ as an element. Since most dictionaries provide a huge vocabulary we cut down the dimension of the vector by using a fixed set of appropriate keywords determined by heuristics and the set of keywords in LARKS itself.

The similarity $dps(Req, Ad)$ of a request Req and an advertisement Ad under consideration is then calculated by :

$$dps(Req, Ad) = \frac{Req \bullet Ad}{|Req| \cdot |Ad|}$$

where $Req \bullet Ad$ denotes the inner product of the weighted keyword vectors. If the value $dps(Req, Ad)$ does exceed a given threshold $\beta \in \mathbf{R}$ the matching process continues with the following steps.

The matchmaker then checks if the declarations and constraints of both specifications for a request and advertisement are sufficiently similar. This is done by a pairwise comparison of declarations and constraints in two steps:

1. *Similarity matching* and

2. *Signature matching*

4.1.4.2 Similarity Matching Let E_i, E_j be variable declarations or constraints, and $S(E)$ the set of words in E. The similarity among two expressions E_i and E_j is determined by pairwise computation of word distances as follows:

$$Sim(E_i, E_j) = 1 - ((\sum_{(u,v)\in S(E_i)\times S(E_j)} d_w(u,v))/|S(E_i) \times S(E_j)|))$$

The similarity value $Sim(S_a.S_b)$ among two specifications S_a and S_b in LARKS is computed as the average of the sum of similarity computations among all pairs of declarations and constraints:

$$Sim(S_a, S_b) = \sum_{(E_i,E_j)\in(D(S_a)\times D(S_b))\cup(C(S_a)\times C(S_b))} Sim(E_i, E_j)/|(D(S_a)\times D(S_b))\cup(C(S_a)\times C(S_b))|$$

with $D(S)$ and $C(S)$ denoting the input/output declaration and input/output constraint part of a specification S in LARKS, respectively.

4.1.4.3 Signature Matching Consider the declaration parts of the request and the advertisement, and determine pairwise if their signatures of the (input or output) variable types match following the type inference rules given below.

Definition 4.1: *Subtype Inference Rules*

Consider two types t_1 and t_2 as part of an input or output variable declaration part (in the form `Input` $v : t_1$; or `Output` $v : t_2$;) in a LARKS specification.

1. Type t_1 is a subtype of type t_2 (denoted as $t_1 \preceq_{st} t_2$) if this can be deduced by the following subtype inference rules.
2. Two types t_1, t_2 are equal ($t_1 =_{st} t_2$) if $t_1 \preceq_{st} t_2$ and $t_2 \preceq_{st} t_1$ with
   (a) $t_1 =_{st} t_2$ if they are identical $t_1 = t_2$
   (b) $t_1 \mid t_2 =_{st} t_2 \mid t_1$ (commutative)
   (c) $(t_1 \mid t_2) \mid t_3 = t_1 \mid (t_2 \mid t_3)$ (associative)

Subtype Inference Rules:

1) $t_1 \preceq_{st} t_2$ if t_2 is a type variable

2) $\dfrac{t_1 =_{st} t_2}{t_1 \preceq_{st} t_2}$

3) t_1, t_2 are sets,
$\dfrac{t_1 \subset t_2}{t_1 \preceq_{st} t_2}$

4) $t_1 \preceq_{st} t_1 \mid t_2$

5) $t_2 \preceq_{st} t_1 \mid t_2$

6) $$\frac{t_1 \preceq_{st} t_2,\ s_1 \preceq_{st} s_2}{(t_1.s_1) \preceq_{st} (t_2.s_2)}$$

7) $$\frac{t_1 \preceq_{st} t_2,\ s_1 \preceq_{st} s_2}{t_1|s_1 \preceq_{st} t_2|s_2}$$

8) $$\frac{t_1 \preceq_{st} t_2}{\texttt{SetOf}(t_1) \preceq_{st} \texttt{SetOf}(t_2)}$$

9) $$\frac{t_1 \preceq_{st} t_2}{\texttt{ListOf}(t_1) \preceq_{st} \texttt{ListOf}(t_2)}$$

•

Matching of two signatures sig and sig' is done by a binary string-valued function fsm on signatures with

$$fsm(sig, sig') = \begin{cases} sub & sig' \preceq_{st} sig \\ Sub & sig \preceq_{st} sig' \\ \epsilon q & sig =_{st} sig' \\ disj & \text{else} \end{cases}$$

Having described both filters of the syntactical matching we now define the meaning of syntactical matching of two specifications written in LARKS.

Definition 4.2: *Syntactical matching of specifications in* LARKS

Consider two specifications S_a and S_b in LARKS with n_k input declarations, m_k output declarations, and v_k constraints $n_k, m_k \in \mathbf{N}, k \in \{a, b\}$, two declarations D_i, D_j, and constraints C_i, C_j in these specifications, and V a given dictionary for the computation of weighted keyword vectors. Let β, γ, θ be real threshold values for profile comparison and similarity matching.

- The **declarations D_i and D_i syntactically match** if they are sufficiently similar:

  $$Sim(D_i, D_j) \geq \gamma \wedge fsm(D_i, D_j) \neq disj.$$

  The **constraints C_i and C_j syntactically match** if they are sufficiently similar:

  $$Sim(C_i, C_j) \geq \gamma.$$

  If both words in every pair $(u, v) \in S(E_i) \times S(E_j)$ of most similar words are associated with a concept C' and C'', respectively, then the distance among C' and C'' in the so-called associative network of the matchmaker must not exceed a given threshold value θ.

  The syntactical match of two declarations or constraints is denoted by a boolean predicate $Synt$.

- The **specifications S_a and S_b syntactically match** if

  1. their profiles match, i.e., $dps(S_a, S_b) \geq \beta$, and
  2. for each declaration or constraint E_i, $i \in \{1, ...n_a\}$ in the declaration or constraint part of S_a there exists a most similar matching declaration or constraint E_j, $j \in \{1, ..., n_b\}$ in the declaration or constraint part of S_b such that

     $$Synt(E_i, E_j) \ \wedge \ Sim(E_i, E_j) = max\{Sim(E_i, E_y), y \in \{1, .., n_b\}\}$$

     (Analogous for each declaration or constraint in S_b.)
  3. for each pair of declarations determined in (1.) the matching of their signatures is of the same type, i.e., for each (D_i, D_j) in (1.) it holds that the value $fsm(D_i, D_j)$ is the same, and
  4. the similarity value $Sim(S_a, S_b)$ exceeds a given threshold.

•

4.1.5 Semantical Matching

By using the syntactical filter many matches might be found in a large agent society. Hence, it is important to use some kind of semantic information to narrow the search, and to pin down more precise matches.

The most common and natural interpretation for a specification (even for a software program) is using sets of pre- and post-conditions, denoted as Pre_S and $Post_S$, respectively. In a simplified notation, any specification S can be represented by the pair $(Pre_S, Post_S)$.

Definition 4.3: *Semantical matching of two specifications*

Consider two specifications $S(Pre_S, Post_S)$ and $T(Pre_T, Post_T)$.

The specification S **semantically matches** the specification T if

$$(Pre_S \Rightarrow Pre_T) \ \wedge \ (Post_T \Rightarrow Post_S)$$

That means, the set of pre-conditions of S logically implies that of T, and the set of post-conditions of S is logically implied by that of T.

•

The problem in performing the semantical matching is that the logical implication is not decidable for first order predicate logic, and even not for a set of Horn clauses. To make the matching process tractable and feasible, we have to decide on the expressiveness of the language used to represent the pre- and post- conditions, and to choose a relation that is weaker than logical implication. The θ-subsumption relation among two constraints C, C' (denoted as $C' \preceq_\theta C''$) appears to be a suitable choice for semantical matching, because it is computationally tractable and semantically sound.

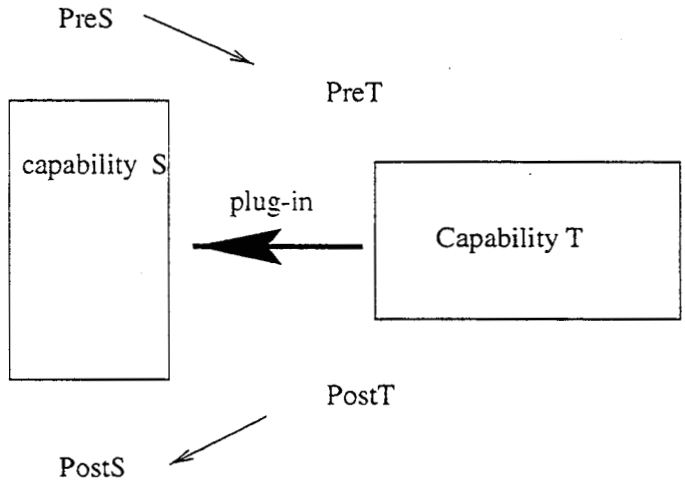


Figure 3: Plug-In Match of Specifications: T plugs into S.

4.1.5.1 Plug-in Semantical Matching in LARKS It is proven in the software engineering area that if the condition of semantical matching in definition 4.3 holds, and the signatures of both specifications match, then T can be directly used in the place of S, i.e., T plugs in S (see figure 4.1.5).

Definition 4.4: *Plug-In semantical matching of two specifications*

Given two specifications $Spec1$ and $Spec2$ in LARKS then $Spec1$ **plug-in matches** $Spec2$ if

- Their signatures matches (see section 4.1.4.2).
- For every clause $C1$ in the set of input constraints of $Spec1$ there is a clause $C2$ in the set of input constraint of $Spec2$ such that $C1 \preceq_\theta C2$.
- For every clause $C2$ in the set of output constraints of $Spec2$ there is a clause $C1$ in the set of output constraints of $Spec1$ such that $C2 \preceq_\theta C1$.

where $\preceq_\theta$ denotes the θ-subsumption relation between constraints.

•

4.1.5.2 θ-Subsumption between Constraints One suitable selection of the language and the relation is the (definite program) clause and the the so-called θ-subsumption relation between clauses, respectively.[3] In the following we will only consider Horn clauses. A general form of Horn clause is

[3] A *clause* is a finite set of *literals*, which is treated as the universally quantified disjunction of those *literals*. A *literal* may be positive or negative. A positive *literal* is an *atom*, a negtive literal is the negation of an *atom*. A *definite program clause* is a clause with one positive

$a_0 \vee (\neg a_1) \vee \ldots \vee (\neg a_n)$, where each $a_i, i \in \{1, \ldots, n\}$ is an atom. This is equivalent to $a_0 \vee \neg(a_1 \wedge \ldots \wedge a_n)$, which in turn is equivalent to $(a_1 \wedge \ldots \wedge a_n) \Rightarrow a_0$).[4] We adopt the standard notation for that clause as $a_0 \leftarrow a_1, \ldots, a_n$; in PROLOG the same clause is written as a_0 :- $a_1, \ldots, a_n$.

Examples of definite program clauses are

- $Date.year > 1995$, $sorted(computerInfo)$,
- $before(x, y, ys) \leftarrow ge(x, y)$, and
- $scheduleMeeting(group1, group2, interval, meetingDuration, meetTime) \leftarrow belongs(p1, group1), belongs(p2, group2), subset(meetTime, interval), length(meetTime) = meetingDuration, available(p1, meetTime), available(p2, meetTime)$.

We say that a clause C **θ-subsumes** another clause D (denoted as $C \succeq_\theta D$) if there is a substitution θ such that $C\theta \subseteq D$. C and D are θ-equivalent if $C \preceq_\theta D$ and D
$preceq_\theta C$.

Examples of θ-subsumption between clauses are

- $P(a) \leftarrow Q(a) \preceq_\theta P(X) \leftarrow Q(X)$
- $P(X) \leftarrow Q(X), R(X) \preceq_\theta P(X) \leftarrow Q(X)$.

Since a single clause is not expressive enough, we need to use a set of clauses to express the pre and post conditions (i.e., the input and output constraints) of a specification in LARKS. A set of clauses is treated as a conjunction of those clauses.

Subsumption between two set of clauses is defined in terms of the subsumption between single clauses. More specifically, let S and T be such sets of clauses. Then, we define that S θ-subsumes T if every clause in T is θ-subsumed by a clause in S.

There is a complete algorithm to test the θ-subsumption relation, which is in general NP-complete but polynomial in certain cases. On the other hand, θ-subsumption is a weaker relation than logical implication, i.e., from $C \preceq_\theta D$ we can only infer that C logically implies D but not vice versa.[5]

5 Examples of Matchmaking using LARKS

Consider the specifications 'IntegerSort' and 'GenericSort' (see example 3.1, 3.2) as a request of sorting integer numbers and an advertisement for some agent's

literal and zero or more negative literals. A *definite goal* is a clause without positive literals. A *Horn clause* is either a definite program clause or a definite goal.

[4] The literal a_0 is called the head of the clause, and $(a_1 \wedge \ldots \wedge a_n)$ is called the body of the clause.

[5] Please also note that the θ-subsumption relation is similar to the query containment in database. When advertisements are database queries, specification matching is reduced to the problem of query containment testing.

capability of sorting real numbers and strings, respectively.

IntegerSort	
Context	Sort
Types	
Input	xs: ListOf Integer;
Output	ys: ListOf Integer;
InConstraints	le(length(xs),100);
OutConstraints	before(x,y,ys) < – ge(x,y); in(x,ys) < – in(x,xs);
ConcDescriptions	

GenericSort	
Context	Sorting
Types	
Input	xs: ListOf Real | String;
Output	ys: ListOf Real | String;
InConstraints	
OutConstraints	before(x,y,ys) < – ge(x,y); before(x,y,ys) < – preceeds(x,y); in(x,ys) < – in(x,xs);
ConcDescriptions	

Assume that the requester and provider agent sends the request IntegerSort and advertisment GenericSort to the matchmaker, respectively. Figure 5 describes the overall matchmaking process for that request.

1. *Context Matching*
   Both words in the `Context` declaration parts are sufficiently similar. We have no referenced concepts to check for terminologically equity. Thus, the matching process proceeds with the following two filtering stages.

2. *Syntactical Matching*

   (a) *Comparison of Profiles*
   According to the result of TF-IDF method both specifications are sufficiently similar:

   (b) *Signature Matching*
   Consider the signatures t_1= (`ListOf Integer`) and t_2= (`ListOf Real|String`). Following the subtype inference rules 9., 4. and 1. it holds that $t_1 \preceq_{st} t_2$, but not vice versa, thus $fsm(D_{11}, D_{21})$ = sub. Analogous for $fsm(D_{12}, D_{22})$ = sub.

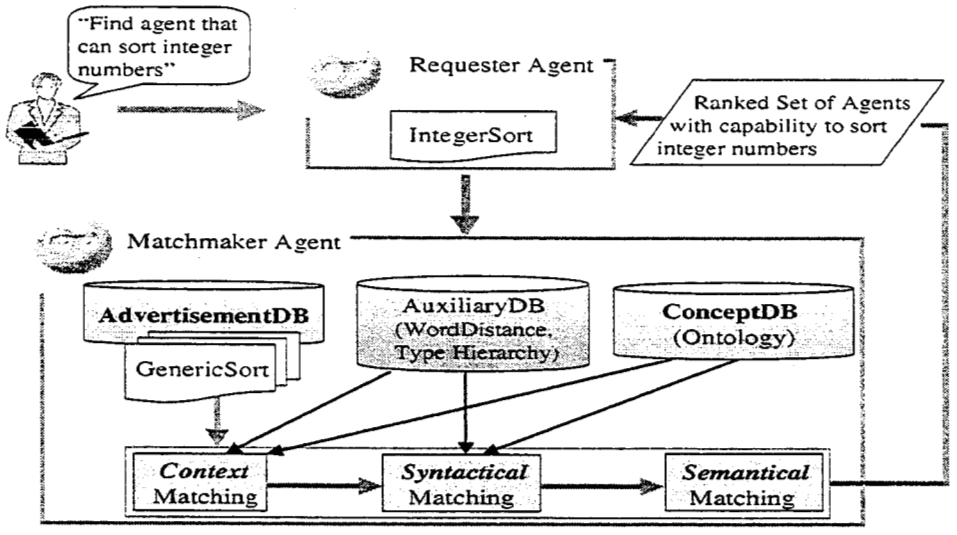


Figure 4: An Example of Matchmaking using LARKS

(c) *Similarity Matching*
Using the current auxiliary database for word distance values similarity matching of constraints yields:

le(length(xs),100))	null	= 1.0
before(x,y,ys) < − ge(x,y)	in(x,ys) < − in(x,xs)	= 0.5729
in(x,ys) < − in(x,xs)	before(x,y,ys) < − preceeds(x,y))	= 0.4375
before(x,y,ys)< − ge(x,y))	before(x,y,ys) < − preceeds(x,y))	= 0.28125

The similarity of both specifications is computed as:
$Sim(\texttt{IntegerSort}, \texttt{GenericSort}) = 0.64$.

3. *Semantical Matching*
The advertisement `GenericSort` also matches semantically with the request `IntegerSort`, because the set of input constraints of `IntegerSort` θ-subsumes that of `GenericSort`, and the output constraints of `GenericSort` θ-subsumes that of `IntegerSort`. Thus `GenericSort` plugs into `IntegerSort`. Please note that this does not hold vice versa.

6 Related works

Agent matchmaking has been actively studied since the inception of software agent research. The earlist matchmaker we are aware of is the ABSI facilitator, which is based on the KQML specification and uses the KIF as the content language. The KIF expression is basically treated like the Horn clauses. The matching between the advertisement and request expressed in KIF is the simple

unification with the equality predicate. Matchmaking using LARKS performs better than ABSI in both, the language and the matching process. The plug-in matching in LARKS uses the θ-subsumption test, which select more matches that are also semantically matches.

The SHADE and COINS[17] are matchmakers based on KQML. The content language of COINS allowes for the free text and its matching algorithm utilizes the tf-idf. The contect language of SHADE matchmaker consists of two parts, one is a subset of KIF, another is a structured logic representation called MAX. MAX use logic frames to declaratively store the knowledge. SHADE uses a frame like representation and the matcher use the prolog like unifier.

A more recent service broker-based information system is InfoSleuth[10, 11]. The content language supported by InfoSleuth is KIF and the deductive database language LDL++, which has a semantics similar to Prolog. The constraints for both the user request and the resource data are specified in terms of some given central ontology. It is the use of this common vocabulary that enables the dynamic matching of requests to the available resources. The advertisements specify agents' capabilities in terms of one or more ontologies. The constraint matching is an intersection function between the user query and the data resource constraints. If the conjunction of all the user constraints with all the resource constraints is satisfiable, then the resource contains data which are relevant to the user request.

A somewhat related research area is the research on information mediators among heterogenous information systems[23][1]. Each local information system is wrapped by a so-called wrapper agent and their capabilities are described in two levels. One is what they can provide, usually described in the local data model and local database schema. Another is what kind of queries they can answer; usually it is a subset of the SQL language. The set of queries a service can accept is described using a grammar-like notation. The matching between the query and the service is simple: it just decides whether the query can be generated by this grammar. This area emphasizes the planning of database queries according to heterogeneous information systems not providing complete SQL sevices. Those systems are not supposed to be searched for among a vast number of resources on the Internet.

The desfription of capabilities and matching are not only studied in the agent community, but also in other related areas.

6.1 Works related with capability description

The problem of capability and service descriptions can be tackled at least from the following different approaches:

1. Software specification techniques.
   Agents are computer programs that have some specific characteristics. There are numerous work for software specifications in formal methods, like model-oriented VDM and Z[28], or algebraic-oriented Larch. Although these languages are good at describing computer programs in a precise

way, the specification usually contains too much details to be of interests to other agents. Besides, those existing languages are so complex that the semantic comparison between the specifications is impossible. The reading and writing of these specifications also require substantial training.

2. Action representation formalisms.
   Agent capability can be seen as the actions that the agents perform. There are a number of action representation formalisms in AI planning like the classical one the STRIPS. The action representation formalism are inadequate in our task in that they are propositional and not involving data types.

3. Concept languages for knowledge representation.
   There are various terminological knowledge representation languages. However, ontology itself does not describe capabilities. On the other hand, it provides auxiliary concepts to assist the specification of the capabilities of agents.

4. Database query capability description.
   The database query capability description technique is developed as an attempt to describe the information sources on the Internet, such that an automated integration of information is possible. In this approach the information source is modeled as a database with restricted quering capabilities.

6.2 Works related with service retrieval

There are three broad approaches to service retrieval. One is the information retrieval techniques to search for relevant information based on text, another is the software component retrieval techniques[26][8][13] to search for software components based on software specifications. The third one is to search for web resources that are typically described as database models[18][23].

In the software component search techniques, [26] defined several notions of matches, including the exact match and the plug-in match, and formally proved the relationship between those matches. [8] propsed to use a sequence of filters to search for software components, for the purpose to increase the efficiency of the search process. [13] computed the distance between similar specifications. All these work are based on the algebraic specification of computer programs. No concept description and concept hierarchy are considered in their work.

In Web resource search techniques, [18] proposed a method to look for better search engines that may provide more relevant data for the user concerns, and rank those search engines according to their relevance to user's query. They propose the directory of services to record descriptions of each information server, called a server description. A user sends his query to the directory of services, which determins and ranks the servers relevant to the user's request. Both the query and the server are described using boolean expression. The search method is based on the similarity measure between the two boolean expressions.

7 Conclusion

The Internet is an open system where heterogeneous agents can appear and disappear dynamically. As the number of agents on the Internet increases, there is a need to define middle agents to help agents locate others that provide requested services. In prior research, we have identified a variety of middle agent types, their protocols and their performance characteristics. Matchmaking is the process that brings requester and service provider agents together. A provider agent advertises its know-how, or capability to a middle agent that stores the advertisements. An agent that desires a particular service sends a middle agent a service request that is subsequently matched with the middle agent's stored advertisements. The middle agent communicates the results to the requester (the way this happens depends on the type of middle agent involved). We have also defined protocols that allow more than one middle agent to maintain consistency of their adevertisement databases. Since matchmaking is usually done dynamically and over large networks, it must be efficient. There is an obvious trade-off between the quality and efficiency of service matching in the Internet.

We have defined and implemented a language, called LARKS, for agent advertisement and request and a matchmaking process using LARKS. LARKS judiciously balances language expressivity and efficiency in matching. LARKS performs both syntactic and semantic matching, and in addition allows the specification of concepts (local ontologies) via ITL, a concept language.

The matching process uses five filters, namely context matching, comparison of profiles, similarity matching, signature matching and semantic matching. Different degrees of partial matching can result from utilizing different combinations of these filters. Selection of filters to apply is under the control of the user (or the requester agent).

Acknowledgements:
We would like to thank Davide Brugali for helpful discussions and Seth Widoff for help with the implementation. This research has been sponsored by ONR grant N-00014-96-16-1-1222.

A Syntax of LARKS

Definition A.1: *Syntax of Larks*

The syntax of LARKS is given by the following production system in EBNF-grammar:

$$
\begin{array}{lcl}
< specification > & ::= & < Ident > \ [< CDeclaration >] \ [< TDeclarations >] \\
 & & [< Declarations >] \ [< Constraints >] \\
< CDeclaration > & ::= & \texttt{'Context'} \ < CDec > \\
< CDec > & ::= & < Ident > \texttt{'*'} \ < Termdefinition > \texttt{';'} \\
< TDeclarations > & ::= & < TDec > \ | \ < TDec > \texttt{';'} < TDeclarations > \\
< Delarations > & ::= & \texttt{'Input'} \ < OptDecList > \ \texttt{'Output'} \ < DecList > \\
 & & \\
< TDec > & ::= & \texttt{'type'} \ < Ident > [\texttt{'::'} < TExp >] \texttt{';'} \mid \texttt{'basicType'} \ < IdentList > \texttt{';'} \\
< Dec > & ::= & < Ident > \texttt{':'} < TExp > [\texttt{'='} < Exp >] \texttt{';'} \\
< DecList > & ::= & < Dec > \ | \ < Dec > \texttt{';'} < DecList > \\
< OptDecList > & ::= & < OptDec > \ | \ < OptDec > \texttt{';'} < OptDecList > \\
< OptDec > & ::= & [\texttt{'Optional'}] \ < Dec > \\
< TExp > & ::= & < TVar > \ | \ < BType > \ | \ < PType > \ | \ < CType > \\
< PType > & ::= & \texttt{'Bool'} \mid \texttt{'Int'} \mid \texttt{'Real'} \mid \texttt{'String'} \\
< CType > & ::= & \texttt{'('} [< Ident > \texttt{':'}] \ < TExp > \texttt{','} \ [< Ident > \texttt{':'}] \ < TExp > \texttt{')'} \mid \\
 & & < TExp > \ \texttt{'|'} \ < TExp > \mid \\
 & & < TExp > \ \texttt{'->'} \ < TExp > \mid \\
 & & \texttt{'SetOf'} \ \texttt{'('} < TExp > \texttt{')'} \mid \\
 & & \texttt{'ListOf'} \ \texttt{'('} TExp \texttt{')'} \mid \\
 & & \texttt{'\{'} < ExpList > \texttt{'\}'} \\
< Exp > & ::= & < aExp > \ \mid \texttt{'('} < ExpList > \texttt{')'} \mid \texttt{'\{'} < ExpList > \texttt{'\}'} \mid \\
 & & < Exp > \texttt{'('} < ExpList > \texttt{')'} \mid < Exp > \texttt{'.'} < Ident > \\
< ExpList > & ::= & < Exp > \ | \ < Exp > \texttt{','} \ < ExpList > \\
< aExp > & ::= & < sConst > \ | \ < var > \ | \ < const > \\
< IdentList > & ::= & < Ident > \ | \ < Ident > \texttt{','} < IdentList > \\
< Constraints > & ::= & [\texttt{'InConstraints'} < formulaList >] \\
 & & [\texttt{'OutConstraints'} < formulaList >] \\
< formulaList > & ::= & < formula > \ | \ < formula > \texttt{';'} \ < formulaList > \\
< formula > & ::= & < atomList > \\
< atomList > & ::= & < atom > \ | \ < atom > \texttt{','} \ < atomList > \\
< atom > & ::= & < predicate > \mid \texttt{'not'} \ < predicate > \\
< predicate > & ::= & < Ident > \\
< var > & ::= & < Ident > \\
< const > & ::= & < Ident >
\end{array}
$$

with non-terminals $< Ident >, < var >$, and $< const >$ denoting an identifier, variable and constant, respectively. The non-terminal $< Termdefinition >$ refers to that in the concept language ITL (see below), thus denoting a kind of a so-called 'escape hatch' from LARKS to ITL.

Convention:
In a capability description or request any term definition will be replaced by the name of the corresponding concept or role which is assumed to be available in the local knowledge base.

B The concept language ITL

Definition B.1: *Syntax of* ITL

The syntax of the concept language ITL is given by the following production system in EBNF-grammar:

$$
\begin{array}{lcl}
< Terminology > & ::= & < Termdefinition >^{+} \\
< Termdefinition > & ::= & < Conceptdefinition > \mid < Roledefinition > \\
< Conceptdefinition > & ::= & < atomicConcept > \text{ '=' } < Concept > \mid \\
 & & < atomicConcept > \text{ ':=' } < Concept > \\
< Roledefinition > & ::= & < atomicRole > \text{ '=' } < Role > \mid \\
 & & < atomicRole > \text{ ':=' } < Role > \\
< Concept > & ::= & < Conc > \mid < AttrConc > \\
< Conc > & ::= & < atomicConcept > \mid \\
 & & < primComponent > \mid \text{'(not' } < primConcComponent > \text{)} \mid \\
 & & \text{'(and' } < Concept >^{+} \text{ ')'} \mid \\
 & & \text{'(atleast' } n < Role > \text{')'} \mid \\
 & & \text{'(atmost' } m < Role > \text{')'} \mid \\
 & & \text{'(exists' } < Role > < Concept > \text{')'} \mid \\
 & & \text{'(all' } < Role >< Concept > \text{')'} \mid \\
 & & \text{'(le' } < num > \text{')'} \mid \text{'(ge' } < num > \text{')'} \mid \\
 & & \text{'(lt' } < num > \text{')'} \mid \text{'(gt' } < num > \text{')'} \\
< AttrConc > & ::= & \text{'aset('} < aval >^{+} \text{ ')'} \\
< Role > & ::= & \text{'(androle' } < Role >^{+} \text{ ')'} \mid \\
 & & < atomicRole > \mid < primRoleComponent > \\
< atomicConcept > & ::= & < identifier > \mid \text{'nothing'} \\
< atomicRole > & ::= & < identifier > \\
< primComponent > & ::= & < primConcComponent > \mid < primRoleComponent > \\
< primConcComponent > & ::= & < identifier > \text{'.'} \\
< primRoleComponent > & ::= & < identifier > \text{':'} \\
< aval > & ::= & < identifier > \\
< Term > & ::= & < Concept > \mid < Role > \\
< ObjectSet > & ::= & < Instance >^{*} \\
< Instance > & ::= & < ConceptInstance > \mid < RoleInstance > \\
< ConceptInstance > & ::= & \text{'('} < Object > < atomicConcept > \text{')'} \mid \\
 & & \text{'(}< Object > \text{ not' } < primConcComponent > \text{')'} \\
< RoleInstance > & ::= & \text{'('} < Object > < atomicRole > < Object > \text{')'} \mid \\
 & & \text{'('} < Object > < NumRestr > < atomicRole > \text{')'} \\
< NumRestr > & ::= & \text{'atleast' } < num > \mid \text{'atmost' } < num > \\
< Object > & ::= & < identifier >
\end{array}
$$

The meaning of (atomic) concept or role, attribute concept, concept and role definition, term definition, term, terminology and object set is defined as the set of strings which can be reduced to the respective non-terminal symbols in the production system.

It is assumed that in every terminology T (written in ITL) all used atomic concepts

and roles are unique identifiers and defined in T; the enumerable sets of identifiers for concepts and roles, attribute values and objects, as well as primitive concept and role components are assumed to be pairwise disjoint. In addition, every primitive component (undefined identifier) in a terminology is assigned a given, fixed meaning[6].

•

Definition B.2: *Semantic of* ITL

Let G be a grammar, $\mathcal{D}$ interpretation domain and D, D_a disjoint subsets with $\mathcal{D} = D \uplus D_a$. $\mathcal{P}(S)$ denotes the power set of any set S. The semantic of ITL terms is defined by the following interpretation function.

$$\epsilon : \begin{cases} Conc & \rightarrow & \mathcal{P}(\mathcal{D}) \\ Role & \rightarrow & \mathcal{P}(D \times \mathcal{D}) \\ Attr & \rightarrow & D_a \end{cases} \tag{1}$$

ϵ is a ITL-**interpretation** if it satisfies the following equations:

$$\epsilon(\texttt{and } C_1 ... C_n)) = \bigcap_{i=1}^{n} \epsilon(C_i) \tag{2}$$

$$\epsilon((\texttt{all } R\ C)) = \{d \in \mathcal{D} : rg(d, \epsilon(R)) \subseteq \epsilon(C)\} \tag{3}$$

$$\epsilon((\texttt{exists } R\ C)) = \{d \in \mathcal{D} : rg(d, \epsilon(R)) \cap \epsilon(C) \neq \emptyset\} \tag{4}$$

$$\epsilon((\texttt{atleast } n\ R)) = \{d \in \mathcal{D} : \mid rg(d, \epsilon(R)) \mid \geq n\} \tag{5}$$

$$\epsilon((\texttt{atmost } n\ R)) = \{d \in \mathcal{D} : \mid rg(d, \epsilon(R)) \mid \leq n\} \tag{6}$$

$$\epsilon(\texttt{aset}(a_1, ..., a_n)) = \{\epsilon(a_1), ..., \epsilon(a_n)\} \tag{7}$$

$$\epsilon((\texttt{not } C^p)) = D \setminus \epsilon(C^p) \tag{8}$$

$$\epsilon((\texttt{androle } R_1 ... R_n)) = \bigcap_{i=1}^{n} \epsilon(R_i) \tag{9}$$

$$\epsilon(nothing) = \emptyset \tag{10}$$

with

$$rg(d.\epsilon(R)) := \{y \in \mathcal{D} : (d, y) \in \epsilon(R)\} \tag{11}$$

$rg(d.\epsilon(R))$ denotes the set of *role fillers* of instance d for the role R.

All attributes $a_1, ..., a_n$ of the concept $\underline{\texttt{aset}}(a_1, ..., a_n)$ are interpreted as constants, i.e., for some $D_a \subseteq Attr$ we assign $\epsilon(a_i) = a_i, i \in \{1, ..n\}$. The interpretation of the operators (`le` n), (`ge` n), (`lt` n), and (`gt` n) for numerical comparison denotes the set of real numbers $x \in D$ with $x \leq n$, $x \geq n, x < n$, and $x > n$, respectively.

•

[6] Primitive components are elements of a minimal common vocabulary used by each agent provider/user for a construction of their local domain-dependent terminologies (and object sets).

References

[1] Jose' Luis Ambite and Craig A. Knoblock. Planning by Rewriting: Efficiently Generating High-Quality Plans. Proceedings of the Fourteenth National Conference on Artificial Intelligence, Providence, RI, 1997.

[2] J. E. Caplan, M. T. Harandi. A logical framework for software proof reuse. Proceedings of the ACM SIGSOFT Symposium on Software Reusability, April 1995. ACM Software Engineering Note, Aug. 1995.

[3] K. Decker, K. Sycara, M. Williamson. Middle-Agents for the Internet. Proc. 15th IJCAI, pages 578-583, Nagoya, Japan, August 1997.

[4] S. Cranefield, A. Diaz, M. Purvis. Planning and Matchmaking for the Interoperation of Information Processing Agents. The Information Science Discussion Paper Series No. 97/01, University of Otago.

[5] P. Fankhauser, M. Kracker, E.J. Neuhold. Semantic vs. Structural Resemblance of Classes. Special Issue: Semantic Issues in Multidatabase Systems, ACM SIGMOD RECORD, Vol. 20, No. 4, pp.59-63, 1991.

[6] P. Fankhauser, E.J. Neuhold. Knowledge based integration of heterogeneous databases. Proceedings of IFIP Conference DS-5 Semantics of Interoperable Database Systems, Lorne, Victoria, Australia, 1992.

[7] T. Finin, R. Fritzson, D. McKay, R. McEntire. KQML as an Agent Communication Language. Proc. 3rd International Conference on Information and Knowledge Management CIKM-94, ACM Press, 1994.

[8] J. Goguen, D. Nguyen, J. Meseguer, Luqi, D. Zhang, V. Berzins. Software component search. Journal of Systems Integration, 6, pp. 93-134, 1996.

[9] G. Huck, P. Fankhauser, K. Aberer, E.J. Neuhold. Jedi: Extracting and Synthesizing Information from the Web. Proceedings of International Conference on Cooperative Information Systems CoopIS'98, IEEE Computer Society Press, 1998.

[10] Jacobs,N., Shea,R., 1995, "Carnot and InfoSleuth - Database Technology and the WWW", ACM SIGMOD Intern. Conf. on Management of Data, May 1995

[11] Jacobs,N., Shea,R., 1996, "The role of Java in InfoSleuth: Agent-based exploitation of heterogeneous information ressources", Proc. of Intranet-96 Java Developers Conference, April 1996

[12] S. Jha, P. Chalasani, O. Shehory and K. Sycara. A Formal Treatment of Distributed Matchmaking. In Proceedings of the Second International conference on Autonomous Agents (Agents 98), Minneapolis, MN, May 1998.

[13] J-J. Jeng, B.H.C. Cheng. Specification matching for software reuse: a foundation. Proceedings of the ACM SIGSOFT Symposium on Software Reusability, ACM Software Engineering Note, Aug. 1995.

[14] M. Klusch. *Cooperative Information Agents on the Internet.* PhD Thesis, University of Kiel, December 1996 (in German) Kovac Verlag, Hamburg, 1998, ISBN 3-86064-746-6.

[15] M. Kracker. A fuzzy concept network. Proc. IEEE International Conf. on Fuzzy Systems, 1992.

[16] R.Kruse, E.Schwecke, J.Heinsohn. *Uncertainty and Vagueness in Knowledge Based Systems*, Springer, 1991.

[17] D. Kuokka, L. Harrada. On using KQML for Matchmaking. Proc. 3rd Intl. Conf. on Information and Knowledge Management CIKM-95, pp. 239-45, AAAI/MIT Press, 1995.

[18] S.-H. Li, P. B. Danzig. Boolean Similarity Measures for Resource Discovery. IEEE Transactions on Knowledge and Data Engineering, Vol.9, No. 6, November/December, 1997.

[19] B. Nebel. *Reasoning and revision in hybrid representation systems*, Lecture Notes in Artificial Intelligence LNAI Series, Vol. 422, Springer, 1990.

[20] G. Smolka, and B. Nebel. *Representation and Reasoning with attributive descriptions.* IWBS Report 81, IBM Deutschland Wissenschaftl. Zentrum, 1989.

[21] G. Smolka, and M. Schmidt-Schauss. Attributive concept description with complements, AI 48, 1991.

[22] K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed Intelligent Agents. IEEE Expert, pp36-46, December 1996.

[23] V. Vassalos, Y. Yapakonstantinou. Expressive Capabilities Description Languages and Query Rewriting Algorithms. available at http://www-cse.ucsd.edu/ yannis/papers/vpcap2.ps

[24] G. Wickler. Using Expressive and Flexible Action Representations to Reason about Capabilities for Intelligent Agent Cooperation. http://www.dai.ed.ac.uk/students/gw/phd/story.html

[25] WordNet - a Lexical Database for English. http://www.cogsci.princeton.edu/ wn/

[26] A. M. Zaremski, J. M. Wing Specification matching of software components. Technical Report CMU-CS-95-127, 1995.

[27] Resource Description Framework (RDF) Schema Specification, http://www.w3.org/TR/WD-rdf-schema/.

[28] Ben Potter, Jane Sinclair, David Till, Introduction to Formal Specification and Z, Prentice-Hall International Series in Computer Science.

This page intentionally left blank.

Distribution:
MS0188 LDRD Program Office, 1030 (Attn: Donna Chavez)
MS0451 S. G. Varnado, 6500
MS0455 R. S. Tamashiro, 6517
MS0455 H. E. Link, 6517
MS0455 S. Y. Goldsmith, 6517
MS0455 L. R. Phillips, 6517
MS0455 S. V. Spires, 6517
MS9018 Central Technical Files, 8945-1
MS0612 Review & Approval Desk for DOE.OSTI, 9612
MS0899 Technical Library, 9616

This page intentionally left blank.