A Framework for Developing Conversational Agents

by
Richard Scott Cost

Dissertation submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1999

APPROVAL SHEET

Title of Dissertation: A Framework for Developing Conversational Agents

Name of Candidate: Richard Scott Cost
Doctor of Philosophy, 1999

Dissertation and Abstract Approved: Timothy W. Finin
Timothy Wilking Finin
Professor
Department of Computer Science
and Electrical Engineering

Date Approved: July 15, 1999

ABSTRACT

Title of Dissertation: A Framework for Developing Conversational Agents

Richard Scott Cost, Doctor of Philosophy, 1999

Dissertation directed by: Timothy Wilking Finin
Professor
Department of Computer Science and Electrical Engineering

The term 'agent' has always been a moving target, implying different properties to the designers of different systems. Consequently, agent-based systems have widely varying architectures and are frequently incompatible. The use of development environments or base platforms is an effective solution to this problem, allowing developers to focus on internal aspects of their systems while embedding them in a common framework. This situation has been improved considerably by the development of agent communication languages, which facilitate interoperability by standardizing the ways in which agents communicate. A number of such environments exist for agent development, and a few have become quite successful. This work presents Jackal, a comprehensive communication package that supports the construction and deployment of distributed, Java-based Multi-Agent Systems. Because communication is so central to distributed systems, Jackal integrates two important components: A structured, conversation-based approach to message management, which supports the abstract specification of agent behavior, and the KQML Naming Scheme (KNS), a set

of protocols for advanced address resolution and agent identification. The conversation mechanisms support the specification of agent behavior in high level, abstract protocols, which are interpreted within the context of messages in the Jackal-based agent. These protocols can be easily uploaded by or exchanged among agents at runtime. KNS provides a communication service layer that encompasses agent naming and name resolution, persistent distributed identity, and authentication. Jackal integrates conversation management and KNS in a flexible, extensible framework that can be used by a single agent or shared as a community resource. Its blackboard-based internal architecture and extensive support for inter-agent communication differentiate Jackal from other development tools currently available. This work explores Jackal in some detail, and describes the use of Colored Petri Nets (CPN), a well-research formalism for concurrency, in modeling agent conversation policies. A description language, Protolingua, is proposed, which will allow agents to exchange and manipulate CPN-based protocols.

For my daughters,
Caitlin and Veda.

Acknowledgments

I would like to gratefully acknowledge the assistance and support that I have received from my family, instructors, colleagues and friends. In particular:

My advisor, Professor Tim Finin, and the other members of my dissertation committee: Professors Keith Decker, Kostas Kalpakis, Yannis Labrou, James Mayfield, Charles Nicholas, and Anupam Joshi
Professor Yun Peng
Ian Soboroff and other members of The Laboratory for Advanced Information Technology and The Center for Architectures for Data-driven Information Processing
The Consortium for Intelligent Integration of Manufacturing Planning and Execution
The members of the Design/CPN team at Aarhus University
The Jackal user community

Contents

List of Tables

List of Figures

R. Scott Cost
Department of Computer Science and Electrical Engineering
College of Engineering
University of Maryland Baltimore County
1000 Hilltop Circle
Baltimore, MD 21250
Voice: +1 (410) 419-8538
FAX: +1 (202) 478-0300
E-mail: cost@acm.org

EDUCATION

- COLLEGE OF ENGINEERING
  UNIVERSITY OF MARYLAND BALTIMORE COUNTY
  BALTIMORE, MARYLAND
  Doctor of Philosophy, Computer Science, 1999
  Thesis: A Framework for Developing Conversational Agents.
  Advisor: Tim Finin.

- GOUCHER COLLEGE
  TOWSON, MARYLAND
  Certificate, Post-Baccalaureate Premedical Program, 1993
  Clinical experience in the Department of Neurology and Neurosurgery, Johns Hopkins Hospital, Baltimore, Maryland.

- G.W.C. WHITING SCHOOL OF ENGINEERING
  JOHNS HOPKINS UNIVERSITY
  BALTIMORE, MARYLAND
  Master of Science in Engineering, Computer Science, 1991
  Thesis: An Instance-Based Approach to Learning with Symbolic Features.
  Advisor: Steven Salzberg.

- COLGATE UNIVERSITY
  HAMILTON, NEW YORK
  Bachelor of Arts, Computer Science (Major) and Mathematics (Minor), 1988 High Honors in Computer Science.
  Thesis: Communication on a Hypercube Network.
  Advisor: Christopher Nevison.

TECHNICAL/RESEARCH EXPERIENCE

- DEPARTMENT OF COMPUTER SCIENCE AND ELECTRICAL ENGINEERING
  COLLEGE OF ENGINEERING
  UNIVERSITY OF MARYLAND BALTIMORE COUNTY
  BALTIMORE, MARYLAND
  Research Assistant, June 1995 to Present
  Research and implement tools for software agent communication and coordination. Develop agents for various projects in C, Lisp, Prolog, Tcl and Java, utilizing the Knowledge Query and Manipulation Language (KQML) agent communication language. Created a KQML extension to Tcl/Tk (TKQML) that supports integration of diverse and distributed systems. Currently working on a Java-based conversational implementation of KQML (Jackal), in support of a NIST sponsored enterprise integration and electronic commerce initiative.

  System Administrator, September 1996 to Present
  Administer and support a Microsoft Windows NT domain for the Laboratory for Advanced Information Technology.

- DEPARTMENT OF MENTAL HYGIENE
  SCHOOL OF HYGIENE AND PUBLIC HEALTH
  JOHNS HOPKINS UNIVERSITY
  BALTIMORE, MARYLAND
  Research Associate, July 1994 to January 1995
  Coordinated activities of Eye Tracking Disorders Laboratory. Developed software for analysis of eye tracking data. Trained lab personnel in patient assessment and data analysis.

- DEPARTMENT OF COGNITIVE SCIENCE
  KRIEGER SCHOOL OF ARTS AND SCIENCES
  JOHNS HOPKINS UNIVERSITY
  BALTIMORE, MARYLAND
  Research Associate, June 1993 to June 1994
  Designed a cognitive science research database. Directed and developed software for experiments in human pre-attentive vision. Supported computing needs of the Cognitive Neuropsychology Laboratory.

- DEPARTMENT OF NEUROLOGY/NEUROSURGERY
  JOHNS HOPKINS HOSPITAL
  BALTIMORE, MARYLAND
  Research Programmer, September 1991 to May 1992
  Conducted research into the automated detection of seizures in monitored epilepsy patients. Generally supported computer facilities of Epilepsy Monitoring Unit.

- BENSON COMPUTER RESEARCH CORPORATION
  MCLEAN, VIRGINIA
  Parallel Systems Software Engineer, May 1990 to November 1990
  Designed (Transputer-based) parallel computer architectures for business applications. Directed joint venture with Inovatic Corporation (Grenoble, France) to convert existing optical character recognition software for use on proprietary parallel system.

- VET COMPUTING SERVICES
  NEW YORK STATE COLLEGE OF VETERINARY MEDICINE
  CORNELL UNIVERSITY
  ITHACA, NEW YORK
  Medical Database Programmer, November 1988 to June 1989
  Maintained and developed database systems (MUMPS) for Medical Records and Histopathology departments. Facilitated faculty and student research.

- SUMMER POSITIONS

  - DATA SYSTEMS TECHNOLOGY DIVISION
    NASA GODDARD SPACE FLIGHT CENTER
    GREENBELT, MARYLAND
    Consultant, summer 1991
    Conducted review of a project involving the implementation of an intelligent software engineering aid. Lectured on topics in Machine Learning.

  - DEPARTMENT OF COMPUTER SCIENCE
    COLGATE UNIVERSITY
    HAMILTON, NEW YORK
    Research/Teaching Assistant, summers 1988, 1989
    Investigated message routing techniques for parallel processing (Transputer) systems. Assisted in teaching summer programs, including a computing seminar for university faculty.

- BELLCORE
  SYSTEM SECURITY ADMINISTRATION
  PISCATAWAY, NEW JERSEY
  Programmer, summer 1987
  Developed utilities for detecting improper access to IBM mainframe computer.
- CIBA-GEIGY CORPORATION
  MANAGEMENT SERVICES DIVISION
  ARDSELY, NEW YORK
  Programmer, summer 1986
  Developed financial reporting software.

CLINICAL EXPERIENCE

- DEPARTMENT OF MENTAL HYGIENE
  JOHNS HOPKINS UNIVERSITY
  BALTIMORE, MARYLAND
  Research Associate, July 1994 to January 1995
  Conducted oculomotor assessment of individuals with schizophrenia, manic depression and other psychiatric disorders.

- PEDIATRIC EPILEPSY CLINIC
  THE JOHNS HOPKINS HOSPITAL
  BALTIMORE, MARYLAND
  Volunteer/Research Associate, January 1992 to June 1994
  Interviewed patients with physicians and staff. Reviewed cases and participated in discussion of treatment. Conducted analysis of clinical research data for an FDA study.

- ADULT EPILEPSY CLINIC
  THE JOHNS HOPKINS HOSPITAL
  BALTIMORE, MARYLAND
  Volunteer, October 1991 to December 1991
  Interviewed patients with physicians and staff. Reviewed cases and participated in discussion of treatment.

- EMERGENCY DEPARTMENT
  UNION MEMORIAL HOSPITAL
  BALTIMORE, MARYLAND
  Volunteer, March 1991 to August 1991
  Served as an emergency room volunteer. Transported patients, stocked rooms, assisted with procedures.

PUBLICATIONS (Reverse Chronological Order)

Journal Publications and Submissions

- Yun Peng, Tim Finin, Yannis Labrou, Scott Cost, Bill Chu, Jeremy Long, William J. Tolone, and Akram Boughannam, "An Agent-based Approach for Manufacturing Integration: The CIIMPLEX Experience." *The Journal of Applied Artificial Intelligence*, 13:1-2, 1999.

- John M. Freeman M.D., Eileen P. G. Vining M.D., Scott Cost M.S.E. and Pratibah Singhi M.D., "Does Carnitine Administration Improve the Symptoms Attributed to Anticonvulsant Medications? A Double Blind, Crossover Study." *Journal of Pediatrics*, June 1994.

- Scott Cost and Steven Salzberg, "A Weighted Nearest Neighbor Algorithm for Learning with Symbolic Features." *Journal of Machine Learning*, 10:1, 1993.

- Steven Salzberg and Scott Cost, "A Nearest Neighbor Approach to Predict Protein Structure." *Journal of Molecular Biology*, V. 226, 1992.

Refereed Conference Publications and Submissions

- R. Scott Cost, Tim Finin, Yannis Labrou, Xiaocheng Luan, Yun Peng, Ian Soboroff, James Mayfield and Akram Boughannam, "An Agent-Based Infrastructure for Enterprise Integration." *First International Symposium on Agent Systems and Applications (ASA '99)*, Palm Springs, California, October 3-6, 1999.

- R. Scott Cost, Ye Chen, Tim Finin, Yannis Labrou and Yun Peng, "Using Colored Petri Nets for Conversation Modeling." To appear in *Working Notes of the Workshop on Agent Communication Languages, Sixteenth International Joint Conference on Artificial Intelligence (IJCAI '99)*, Stockholm, Sweden, August 1, 1999.

- Yun Peng, Tim Finin, Harry Chen, Ling Wang, Yannis Labrou and R. Scott Cost, "An Agent System for Application Initialization in an Integrated Manufacturing Environment." To appear in *Proceedings of the 3rd World Multiconference on Systems, Cybernetics and Informatics and the 5th International Conference on Information Systems Analysis and Synthesis (SCI '99/ISAS '99)*, Orlando, Florida, July 31 - August 4, 1999.

- R. Scott Cost, Ye Chen, Tim Finin, Yannis Labrou and Yun Peng, "Modeling Agent Conversations with Colored Petri Nets." *Working Notes of the Workshop on Specifying and Implementing Conversation Policies, Third International Conference on Autonomous Agents (Agents '99)*, Seattle, Washington, May 1, 1999.

- R. Scott Cost, Tim Finin, Yannis Labrou, Xiaocheng Luan, Yun Peng, Ian Soboroff, James Mayfield and Akram Boughannam, "Agent Development with Jackal." (Poster/Abstract) *Third International Conference on Autonomous Agents (Agents '99)*, Seattle, Washington, May 1-5, 1999.

- Ye Chen, Yun Peng, Tim Finin, Yannis Labrou and Scott Cost, "A Negotiation-based Multi-agent System for Supply Chain Management." *Working Notes of the Workshop on Agents for Electronic Commerce and Managing the Internet-Enabled Supply Chain, Third International Conference on Autonomous Agents (Agents '99)*, Seattle, Washington, May 1, 1999.

- R. Scott Cost, Tim Finin, Yannis Labrou, Xiaocheng Luan, Yun Peng, Ian Soboroff, James Mayfield and Akram Boughannam, "Jackal: A Java-Based Tool for Agent Development." *Working Notes of the Workshop on Tools for Developing Agents (AAAI '98)*, 1998. Available as an AAAI Technical Report. Also to appear in *International Journal of Human Computer Interaction.*

- R. Scott Cost, Ian Soboroff, Jeegar Lakhani, Tim Finin, Ethan Miller and Charles Nicholas, "TKQML: A Scripting Tool for Building Agents." *Proceedings of the 1997 Conference on Agent Theories and Agent Languages (ATAL97).*

- R. Scott Cost, Ian Soboroff, Jeegar Lakhani, Tim Finin, Ethan Miller and Charles Nicholas, "Agent Development Support for Tcl." (Abstract) *Proceedings of the 1997 Tcl/Tk Workshop (TCL97).*

- Priscilla W. Sheldon-Cost, Raymond DePaulo and R. Scott Cost, "Characteristics of Patients Treated with Valproate." Proceedings of the Conference of the American Psychiatric Association, 1995.

- Scott Cost and Steven Salzberg, "Exemplar-based Learning to Predict Protein Folding." *Proceedings of the Symposium on Computer Applications to Medical Care (SCAMC '90)*, 1990.

Technical Reports

- Ian Soboroff, R. Scott Cost and Peter Finin, "TKQML User's Manual." Technical Report, Department of Computer Science and Electrical Engineering, University of Maryland Baltimore Country.

- R. Scott Cost, Ian Soboroff, Jeegar Lakhani, Tim Finin, Ethan Miller and Charles Nicholas, "TKQML: A KQML Extension to Tcl." Technical Report CS-97-04, Department of Computer Science and Electrical Engineering, University of Maryland Baltimore Country.

Other Publications

- R. Scott Cost, "A Naming and Addressing Protocol for Multi-Agent Systems: A Response to FIPA's Fifth Call for Proposals." January 24, 1999.
- R. Scott Cost, "A Naming and Addressing Protocol for Multi-Agent Systems." (Summary) July 15, 1998.
- R. Scott Cost, "Rational Agents and the Art of Persuasion." Working Notes of the *Symposium on Rational Agency (AAAI Technical Report)*, AAAI Fall Symposium Series, 1995.
- R. Scott Cost, "An Instance Based Approach to Learning with Symbolic Features." Masters Thesis, Department of Computer Science, Johns Hopkins University, 1991.
- R. Scott Cost, "Communication on a Hypercube Network." Undergraduate Honors Thesis, Department of Computer Science, Colgate University, 1988.

SERVICE

Department of Computer Science and Electrical Engineering

- Served as the Representative for the Department to the UMBC Graduate Student Association (GSA) Senate, 1997-98.
- Developed and directed a mentoring program for incoming graduate students, which is now being formally integrated into the Department's graduate program, 1997-99.
- Served as a student member of the Department's Promotion and Tenure Review Committee, 1998.
- Represented the Department at several UMBC open house functions, 1995-99.

HONORS AND AWARDS

- Merck Summer Research Fellowship, The Johns Hopkins University, 1993.
- Bell Atlantic Fellowship, The Johns Hopkins University, 1989-90.
- Honorable Mention, NSF Graduate Fellowship Competition, 1989.
- Award for Academic Excellence in Computer Science, Colgate University, 1988.
- CASE Research Fellowship, Colgate University, 1986-88.

MEMBERSHIPS

- American Association for Artificial Intelligence (AAAI).
- American Medical Informatics Association (AMIA).
- Association for Computing Machinery (ACM).
  - Special Interest Group on Artificial Intelligence (SIGART).
  - Special Interest Group on Information Retrieval (SIGIR).
- Institute for Electrical and Electronics Engineers (IEEE).
  - IEEE Computer Society.
- Phi Kappa Phi, National Honor Society (ΦΚΦ)
- Upsilon Pi Epsilon; National Computer Science Honor Society (ΥΠΕ).
- USENIX Association.
  - System Administrators' Guild (SAGE).

Chapter 1

Introduction

The term 'agent' has always been a moving target within the computer science community, implying different properties to designers of different systems. To some, an agent must be communicative and intelligent. To others, communication may not be important, but mobility may, and so on. Thus, many different agent-based systems have widely varying architectures and use customized languages for inter-agent communication. Consequently, they are frequently incompatible, and cannot easily borrow features from one another. The use of development environments or base platforms is an effective solution to this problem, allowing developers to focus on higher level aspects of their systems, while embedding them in a common framework and drawing from libraries of standard utilities. A number of such environments exist for agent development, and a few have become quite successful. This situation has been improved enormously by the development of agent communication languages and associated messaging infrastructures, which facilitate interoperability by standardizing the ways in which agents communicate.

This work discusses Jackal,[1] a comprehensive communications infrastructure that fa-

[1] Jackal is an acronym for the Java Application for Communicating with the KQML Agent Language

cilitates the construction and use of distributed, Java-based Multi-Agent Systems (MAS). Jackal integrates two powerful components: A conversation management layer, which supports the abstract specification of agent behaviors, and the KQML Naming Scheme (KNS). [2], a set of protocols for advanced address resolution and agent identification. Jackal combines these elements with an internal blackboard interface and a number of advanced features, providing a flexible, extensible framework that facilitates the rapid deployment of agent systems. The contribution of this work then is threefold: The design and implementation of a framework for creating conversation-based agents, the demonstration of the merits of this approach in real-world scenarios, and the development of a methodology for formulating Conversation Policies (CP) based on the (**CPN model!**). In combination, these contributions serve to advance the state of inter-agent communication and interoperability.

Support for interaction specification has been a central theme in Jackal's development. With the spread of agent-based systems, we face the proliferation of software of increasing complexity, principally due to stronger demands for autonomy and adaptability. As we endow these systems with greater responsibility, we must be certain of their reliability. One way to determine the agent behavior with greater certainty is through the use of principled techniques in their design and construction. Conversation management facilitates this by providing a specification language, and an ontology of agent interaction, for the construction of protocols defining agents' communicative behaviors [3]. These protocols can be modified, shared, and are executed directly on interpreters within the agent framework, thus satisfying the need for flexibility and reuse (above). As high-level specifications, these pro-

[2]KNS originated in early 1997 as J3Comm, and is being developed jointly with Ian Soboroff (UMBC) KNS is named for its origins the the University of Maryland Baltimore County (UMBC) KQML environment; it is not directly tied to KQML, however, but is specified at a more abstract level and can be implemented over many different Agent Communication Languages (ACL).

[3]This work argues for the specification of communicative behavior; unless otherwise stated in the text, references to *behavior* should be assumed to have this meaning. Section 6.4.4, however, does show how this model can be used to specify generalized interactive behavior as well.

tocols are amenable to inference regarding various properties, thus affording such benefits as verifiability of system behavior, clarity of design, and ease of integration and reuse.

Communication is a central problem in distributed systems. Without the ability to communicate, a set of agents would be a merely a collection of isolated components. All classic distributed systems problems, such as coordination and distributed reasoning, depend on an underlying communication framework. [4] Jackal is designed to support multiple concurrent agent communication languages, but implements the KQML agent communication language as the default. However, specifying communicative behavior is only half of the battle. We still need to facilitate communication with support infrastructure and protocols for naming, addressing and location; this is the role of KNS. KNS, like the KQML Agent Name Server (ANS) [FPT+95] scheme, adds a communication layer in which symbolic names are mapped to actual network addresses. In addition, however, it offers more advanced support for dynamic group formation and disbanding, [5] and a more abstract notion of agent identity, independent of any particular name or globally unique identifier.

Jackal provides the framework necessary to integrate conversation specifications with the real resources of the Agent Execution Environment (AEE). A fast, flexible infrastructure drives the message stream through one or several interpreters, and provides general agent access to the message stream through a specialized blackboard interface. Modules can be added dynamically to this infrastructure for use by Jackal (e.g. modules for different network protocols) or the agent (e.g. reasoning systems) in a plug-and-play fashion. Jackal can be used as a tool for individual agents, or as a shared resource for a community of agents.

[4] This is not the case in systems where agents gather data indirectly about other agents. An example of such a system is a agents flying in formation with no direct inter-agent communication; only observation.

[5] Group formation here refers to the simple association by name of a given agent with a given group, or domain, of agents.

Jackal differentiates itself from other existing agent development platforms in its extensive support for high-level agent communication, abstract protocol specification, and its internal message distribution architecture. While many other systems have focused on agent mobility, they have often neglected remote communication or omitted it entirely, intending agents to collocate and communicate unstructured or semi-structured data directly. Jackal is committed to supporting inter-agent communication at a universal, symbolic level. This aspect makes Jackal-based MASs more flexible, and enhances the likelihood of their compatibility with other systems.[6]

For simplicity, we limit our discussion in this work to the specification of interaction among agents. However, this paradigm extends easily to the interaction of components within a single agent as well; an idea that will be elaborated on in Section 6.4.4. At some level, however, it is computationally beneficial, or necessary, to have some actions be 'hard-coded'. We see such actions residing in libraries.

Given this abstract specifications of behavior, it is possible to consider the benefits of applying formal methods to the specification layer. There is a broad continuum between formally and informally specified systems; current agents cover a spectrum, clustering toward the middle. Informally specified agent systems abound. These range from vending machines with network connections to very advanced AI systems. What qualifies these as 'informally specified' is that agent behavior(s) are a manifestation of the underlying mechanisms, not a well-defined set of guidelines for directing or constraining the agent's activity. While such systems are easier to build, their behavior in the environment is more difficult to predict, making their complete development more expensive overall.

Much current research focuses on semi-formally specified systems [DPSW97, NU97a,

[6]This presupposes that it is easier to add a language capability to an agent system than to extend an execution environment to support a new type of agent, or extend the agent similarly.

BF97b, KIO95]. In these systems, the agent's behavior is described by some set of protocols, and the agent implementation uses these protocols to direct or constrain the agent's actions. In some of these systems, the protocols are used as a guideline for the agent; in others, it constrains behavior. All show the value of some degree of formality.

Formal specification provides the greatest reward, but is not always practical. Very formal methods are somewhat less accessible and therefore less frequently adopted by developers, making them a poor choice for integration into a development system or scheme. Also, the use of very rich notational systems and logics does not always yield the execution speed of some cryptically concise C code. One example of more formal agent work is the DESIRE project [BKJT95], a successful endeavor in formally specifying MASs. DESIRE models agents and agent-interaction at a conceptual level, and develops plans for interaction through task decomposition. This level of reasoning requires a very high level, formal approach, which is clearly important, but not required for all aspects of agent functionality.

At run time, a Jackal agent has protocols to chose from, perhaps in a very reactive or mechanistic fashion. If possible, it should be able to reason about protocols and make informed selections. Ideally, this agent would be backed by a planner, which could dynamically create protocols as necessary for novel situations, or reuse protocols from a library. A specification language should allow the encoding of all interesting behaviors in a way which is sufficiently agent/domain independent that they can be exchanged, adopted or discussed. Thus, the specification language should be as simple as possible, but carefully structured, which uses a common or standard ontology.

This work is an outgrowth UMBC's involvement with the Consortium for Intelligent Integration of Manufacturing Planning and Execution (CIIMPLEX) . Now in its third release, Jackal (formerly named Lewak) began as a Java implementation of KQML, supporting a project in manufacturing integration. The significant advance in Jackal over earlier,

C-based implementations is its multithreaded support for conversations. These protocols define units of conversation, such as a query-response interaction, and are based loosely on a Finite State Machines (FSM). Jackal provides the ideal environment for the development of and experimentation with protocols and protocol engines.

Chapter 2 gives an introduction to agents and agent-based system in general. Chapter 3 surveys the current work in agent development systems. Motivation for conversation-based specification of agent interaction and communication is presented in Chapter 4. KNS is presented in Chapter 5, and a detailed review of Jackal as it currently exists is given in Section 6. Chapter 7 describes some of the key domains in which Jackal has been used. The remainder of this work is devoted to alternative mechanisms of conversation specification, starting with an introduction to CPNs in Chapter 8. Chapter 9 demonstrates the application of CPNs to conversation and MAS specification. Chapter 10 presents Protolingua, a language independent notation for CPNs.

Chapter 2

Agent Systems

The enormous interest recently in agent systems demonstrates a continuing desire to develop software that is more capable of assisting us in our everyday lives. In this section, we will review some of the current ideas in agent research. Some good surveys of the field are [WJ94, GHN+97, Nwa96, Bra97].

2.1 Overview

The definition of the term *agent* varies widely, largely as a function of the interests of those employing it. For example, researchers in networks will likely claim that agents must be mobile, while reasoning enthusiasts may insist that they be highly intelligent. Some of the more broadly accepted properties required for agenthood are autonomy, communicability, sociability/cooperativity and intelligence. Other popular requirements include mobility/portability, real-time reactivity, reliability/fault tolerance, personalizability, lightness and discretion. Some of these properties, such as intelligence, are relatively independent. Others overlap significantly; for example, sociability and communicability.

Agents have been applied to numerous problem domains; a survey of the most recent Autonomous Agents Conference [GB99b] shows applications to electronic commerce, user interfaces, engineering and maintenance, manufacturing automation, and information retrieval, to name a few.

2.2 Properties Derived versus Properties Desired

For ease of comparison, we will consider these properties in the context of a Client-Server model. Every agent needs an [agent] execution environment in which to run, and resources to use, local or external. Each capability with which we endow an agent can then be characterized by:

1. Support for that capability within the agent (client)
2. External resource (server)
3. Access to local and remote resources.

Figure 2.1 graphically depicts this characterization of agent capabilities

This model can be used to characterize any feature of an agent system. For example, Figure 2.2 shows a rough Client-Server characterization of KQML-based communication in a Jackal-based agent. It has the additional benefit that it separates features involving interaction from those strictly related to the executing program (e.g. robustness) or AEE (e.g. resource protection for the host computer). This allows for a clearer evaluation of the merits of different systems. For example, some claim that because a particular agent architecture supports agent movement, its agents are robust and social. Mobility may lend a social quality, as agents which are collocated may have increased opportunities to interact,

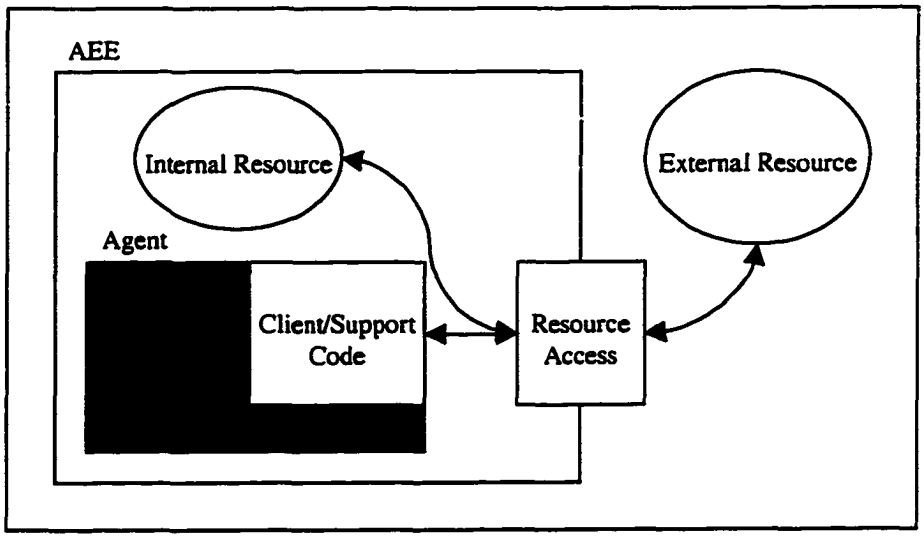


Figure 2.1: Client-Server Characterization of Agent Capabilities

but robustness, while required of a successful mobile agent, is not necessarily a result of access to mobility resources.

2.3 Design Requirements

From the above properties, we can see several key requirements emerging, including:

1. Predictability. If systems will have increased autonomy and/or responsibility for more complex tasks, their behavior in all situations should be more predictable and well circumscribed.

2. Behavioral Flexibility. Since agents will be interacting in a highly dynamic environment, they must be able to modify their behavior as appropriate, either by adopting new behaviors, or adapting to situations on their own.

3. Portability. This is primarily an issue with respect to agent mobility, if we assume that different agent implementations can still apply the same higher level techniques

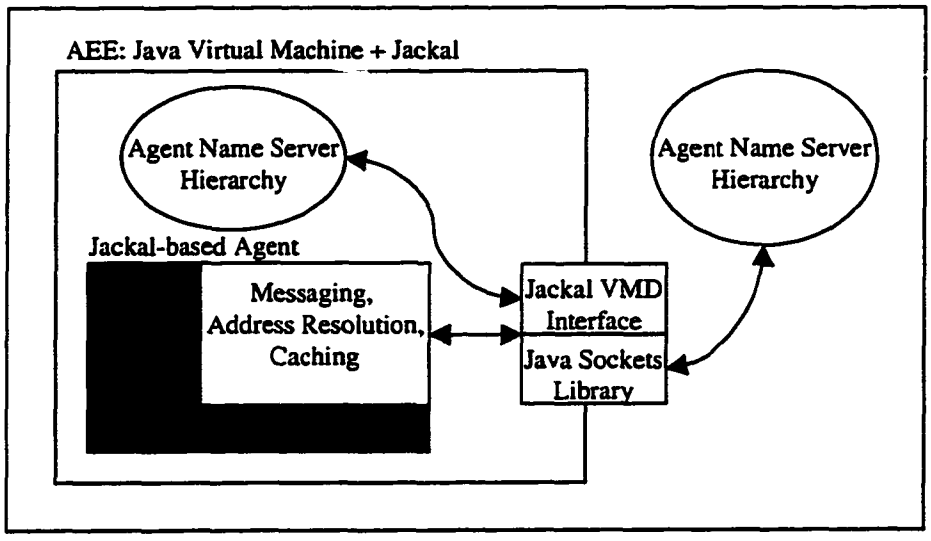


Figure 2.2: Client-Server Characterization of KQML Communication in Jackal. Client modules for formulating and sending address queries, caching addresses and so forth reside with the agent. These modules interact with external sources that can provide address information about agents.

and constraints to their operation. Still, interoperability is better assured if agents are compatible at all levels of their design. Then the line between data and code can be blurred, as agents can swap code modules as knowledge.

The latter two points argue for declarative specification of agent behavior. Mobile agents must be able to run on a variety of host platforms, and not all of these platforms may be known or intended at the time of the agent's creation. Thus, the use of a platform independent specification language, like Java, may be necessary. Beyond this cooperating agents may wish to share or discuss component behaviors, such as protocols for achieving some goal, or for communication via a specified language. In this case, it is to the agents' advantage to have at least a partial specification in a form that can be easily exchanged, modified and reused.

That agents should be predictable suggests that we should be able to reason about agent behaviors, either initially or at runtime, in order to verify certain properties. It also indicates a need for constraints that bound the valid behavior of the agent. This suggests a need for techniques from the formal methods arena.

We do not include here requirements that generally apply to all or most systems, such as reliability.

2.4 Some Definitions

As is evident from the above discussion, the domain of activity covered by agents is very broad and ill-defined, much as the term itself. Many properties that researchers would ascribe to agents are relevant to some domains and not others. In order to clarify discussion, we will enumerate here some definitions to be used in this work.

The term 'agent' is traditionally used to name something or someone in the service of another, or the causative aspect of some effect, and is not restricted in use by the existence or absence of any particular properties in the agent, except that it is an 'actor'. Consider:

> Sweet love! sweet lines! sweet life!
> Here is her hand, the **agent** of her heart;
> Here is her oath for love, her honour's pawn.
> O, that our fathers would applaud our loves,
> To seal our happiness with their consents!
> O heavenly Julia!
>
> *William Shakespeare, "The Two Gentlemen of Verona", Act 1, Scene 3*

> O Sacred, Wise, and Wisdom-giving Plant,
> Mother of Science, Now I feel thy Power
> Within me cleere, not onely to discerne
> Things in thir Causes, but to trace the wayes
> Of highest **Agents**, deemed however wise.

Milton, Paradise Lost, Book 8

I once asked this literary **agent** what writing paid the best, and he said, "ransom notes."

Harry Zimm in "Get Shorty" (1995)

As such, it lends little additional meaning to the term 'software'. It's use in the computer science community has been overloaded with the implication of some sense of rational agency [CL90] on the part of the agent, and some corresponding set of corresponding properties. Regular debates take place over which characteristics should be used to define an 'agent', as if there were an a priori set of properties to be uncovered. In this work, we will observe the following definitions:

1 DEFINITION (AGENT) *For the purposes of this work, the term agent, which will be considered an abbreviation for 'autonomous software agent', and will be used to refer to any software entity that acts autonomously on behalf of/in service of the goals of some other entity. No special properties are implied, except those that can be inferred from the surrounding text, or are explicitly stated (e.g.* **mobile** *agent).*

Agents are applied to many different tasks, and the individual requirements for completion vary widely with the given task. For example, mobility might be required for a data-mining task, or communication for a sensor fusion task. In this sense, it is meaningful to talk about properties of a **particular** agent or **class** of agents. Some properties that are common to many agents include some degree of sophistication, communicability, and some dynamic, persistent model of the task requirements.

2 DEFINITION (*Agent* BEHAVIOR) *With respect to agents (above), that which characterizes the agent's actions and/or state change as a partial function of the agent's model of*

the world and its perceptions. Behavior is the only aspect that we will use to characterize an agent.

Given this view of behavior, for example, agents will not be considered 'mobile agents' simply because they have access to tools which support mobility, but because they exhibit behaviors which make effective use of these resources. In light of these definitions, software components that facilitate mobility, communicability, and so on are merely tools at the agent's disposal.

Chapter 3

Survey of Current Work

Many groups are currently developing agent systems, for a variety of different purposes, and using a variety of different methodologies. We will briefly survey a few of the more significant projects here, with emphasis on systems designed to facilitate agent construction.

Although many different attributes describe agents' abilities, the field can be most easily divided into those projects working towards mobility and those that are not (although these two are certainly not mutually exclusive). The latter tend to have a greater emphasis on communication infrastructure. The former are surveyed here in order to give a more complete depiction of work in the field, and also to demonstrate the significant incompatibility problems arising from mobility and a corresponding reliance on custom AEEs and local communication.

3.1 Mobile Agent Projects

3.1.1 Agent Tcl

Bob Gray's mobile agent project at Dartmouth has generated a tremendous amount of interest and spawned a number of worthwhile research projects. Based on Tcl/Tk, Agent Tcl [Gra96, GKN+96, GKCR97, Gra98] is based on a modified version of the Safe-Tcl [OLW96] Tcl interpreter extension. The use of Tcl affords relative platform independence, due to Tcl's widespread popularity, but the restriction to a nonstandard interpreter does make it somewhat less accessible. Perhaps Agent Tcl's greatest shortcoming is a political one; its foundation in Tcl at a time when Java is prevailing. Although Java is vastly superior for writing many types of programs, Tcl has traditionally been the language of choice for binding existing heterogeneous systems, leaving Agent Tcl a rather broad and solid niche.

Agent Tcl interpreters form the core of the agent 'meeting place'. The system derives considerable security from its basis in the Safe-Tcl Security Model [OLW96].

Like many mobile agent systems, communication facilities exist at a very low level. Agents communicate with one another via the direct transfer of unstructured data while agents are collocated.

See also [KGR96, NCK96, RGK96, RGK97, BKR97] for extensions of Agent Tcl to various problem domains.

3.1.2 Aglets Workbench

The Aglet project [LO98b, AL98, LO98a], originated by Danny Lange while at IBM, is a very promising framework for agent mobility. Based in Java , it provides support for the

construction of small, roving agents called Aglets, which are constructed by extending an agent shell class. Aglets move from place to place by agent-initiated, single entry-point transfer. Places maintain a persistent 'context', which the agent can access.

Some security is achieved by the use of an agent proxy. All access to the agent, including peer-to-peer communication, takes place via the proxy agent. Since the agent and its proxy need not be collocated, this allows for location transparency. Access to the agent's methods can also be selectively restricted with the proxy's intervention.

Communication is supported only through direct or remote method invocation on the receiving agent (or its proxy).

3.1.3 Ara

The principal focus of the Ara project [PS97a, PS97b, Pei97] is portability. Ara agents are mobile, and execute within a custom execution environment on each host. In order to provide for maximal proliferation of their host platform, the designers have constructed a paradigm in which language interpreters (e.g. Tcl, Java) are modified to provide language specific state management functionality. The interpreters then work in concert with an Ara 'core', which provides agent specific resources. Resources are allocated by reservation. Agents are written in a standard interpreted language, and use functionality provided by the interpreter and the 'core'. This platform has been partially implemented for both Tcl and C/C++. For maximal concurrency, agents are implemented as threads. Ara Agents move via self-initiated thread migration.

Little else is implemented within Ara. Work is in progress to incorporate authentication and security, and to adapt a platform for Java. Communication, in the designers' views, should be encouraged at the local level, so remote communication is supported only by raw

transmission over sockets.

3.1.4 Mole

Mole [BHRS97] (See also [SBH96, BHR97]) claims to be the first mobile agent system written in Java, with the first implementation having been completed in 1995. Regardless of the accuracy of this assertion, it is a very well thought out system and a good starting place for anyone researching mobile agents.

Mole supports 'weak migration', which they define as agent initiated transfer of code and data with restoration of data state. This is in contrast to 'strong migration', in which the internal state of the thread is also restored. The decision not to support strong migration was based on a desire for broad platform portability and the difficulty of internal state manipulation in most languages. Migration takes place between 'places'. Mole agents are identified by globally unique IDs (GUID).

Communication in Mole is 'session', or circuit, oriented. Sessions may be established between two collocated or remote agents, but the endpoints are fixed. The mode of communication is RMI for Client-Server interaction, message passing for peer-to-peer. Anonymous communication is based on the event model. These methods are intended to provide low level communication interaction, and possibly support higher level protocols such as KQML. Individual directories are available at each 'place'; there is no hierarchical or global directory service.

A layer of immobile system agents provide agents with access to local resources, giving some measure of security. These agents handle resource accounting.

3.2 Stationary Agent Projects

Each of the projects above requires its agents to situate themselves within custom AEEs, making indirect communication the only means of interactions available between systems. However, these agents do not posses sufficiently powerful/general communication skills required to bridge this gap.

Projects which have emphasized communication have traditionally used KQML as a basis, although they have typically developed new dialects, and use incompatible infrastructure elements (e.g. name servers). More and more recent systems are adopting the FIPA ACL [FIP97].

3.2.1 AgenTalk

AgenTalk [KIO95, KIO96] is a language for specifying protocols for interaction only, entirely independent of the agent system using it. The language itself is straightforward, using an extended FSM as a model (Equation 3.1), and adding a reference for inheritance, variables, and a set of local functions:

$$
\begin{aligned}
Script\ SC_i &= (S_i, s0_i, R_i, V_i, F_i, I_i) \\
S_i &= States \\
s0_i &= Initial\ State, s0_i \in S_i \\
R_i &= Transition\ Rules \\
V_i &= Set\ of\ Local\ Variables \\
F_i &= Script\ Functions \\
I_i &= Parent\ Script(for\ Inheritance)
\end{aligned}
\tag{3.1}
$$

AgenTalk models behavior in terms of message passing, so transition conditions in a script are based on message content. Inheritance works as follows: among the possible actions taken at a transition, a new script or scripts may be spawned. This new script inherits all of the properties of the parent script, such as functions and variables, except where overridden in the child. In the case that multiple transition conditions are enabled, the transition of the most recently spawned child is selected. Multiple child scripts are expected to use shared variables in the parent script in order to resolve local conflicts. Agents participating in an exchange are *not* expected to have the same script, although they may. The inheritance mechanism facilitates the construction of arbitrarily complex scripts through composition.

A clear application of AgenTalk to the contract net protocol and multistage negotiation can be found in [KIO95].

3.2.2 COOL

At the Enterprise Integration Laboratory (EIL) of the University of Toronto, Mihai Barbuceanu and Mark Fox have been developing an agent infrastructure to solve the problem of supply chain coordination [BTF97, BF96c] This is the oldest and in some ways the most well developed project applying the conversational approach. It began with the introduction of the Information Agent (IA) [BF94a, BF94b] in 1994, an entity which consisted primarily of a problem-solving engine and an agent program/interface (Figure 3.1). The IA's role was to serve as a common store and information service to a small community of agents (Figure 3.2). A description logic language, in this case MODEL [Bar93], was used to implement the problem solver component. Agents in the environment communicated using KQML. The agent component translated between KQML messages and expressions in MODEL. The interaction protocols were therefore those prescribed by the then loosely defined KQML semantics.

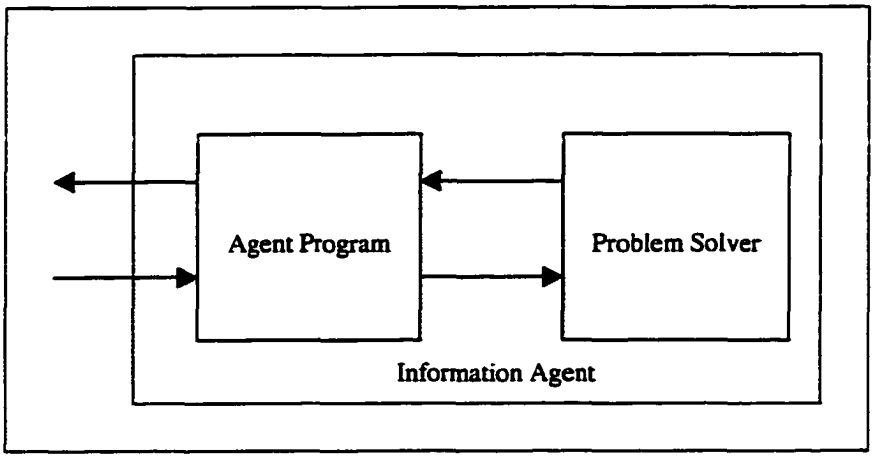


Figure 3.1: EIT's Information Agent

The IA's use of interaction protocols was made more concrete in 1995 with the introduction of the Coordination Language COOL [BF95, FBG95, BF96d, Bar96, BF97b, BF97a].

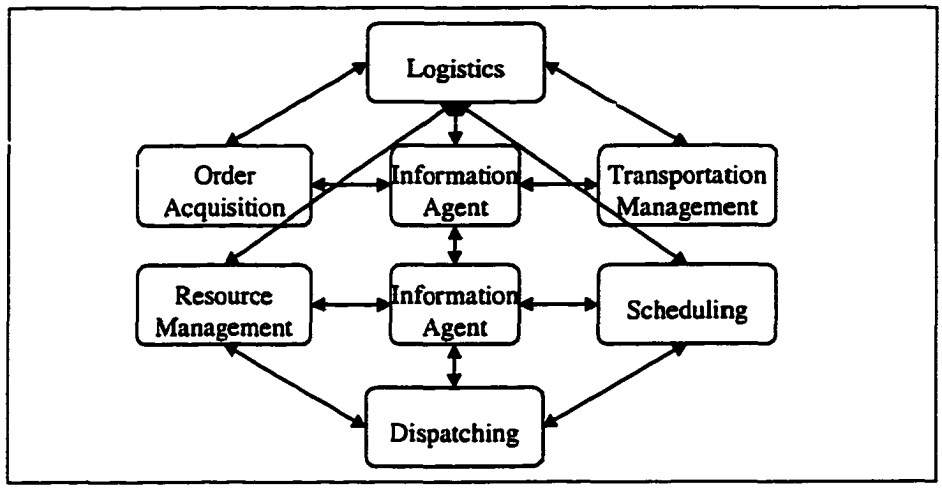


Figure 3.2: Information Agent (EIT) Environment

Four levels of agent interaction were defined: content (of the KQML message), intention (as identified by the KQML message performative), convention (interaction protocols, COOL) and agent modeling. The model chosen for the convention layer was a collection of independent FSMs, each responsible for one of potentially many concurrently executing interactions. Conversations are extended from generic conversation classes, of the form:

$$
\begin{aligned}
Conversation\ C &= (I, R, V, S_0, A, R, F_R, E, F_E) \\
I &= Initiator \\
R &= Name\ of\ Respondent \\
V &= Set\ of\ Local\ Variables \\
S_0 &= Initial\ State \\
A &= Set\ of\ Accepting\ States \\
R &= Set\ of\ Conversation\ Rules \\
F_R &= Conversation\ Rule\ Applier \\
E &= Set\ of\ Error\ Rules \\
F_E &= Error\ Rule\ Applier
\end{aligned}
\tag{3.2}
$$

(3.2) is formally equivalent to:

$$
\begin{aligned}
Conversation\ C &= M, S, R, P \\
M &= Memory \\
S &= Set\ of\ States \\
R &= Set\ of\ Rules \\
P(s,p) &= Value\ of\ Property\ p\ of\ State\ s
\end{aligned}
\tag{3.3}
$$

Conversation rules 3.4 have the form:

$$
\begin{aligned}
ConversationRuleR &= (N, S_i, I, S_f, O, A, E) \\
N &= Name \\
S_i &= Initial\ State \\
I &= Input\ (Message) \\
S_f &= Resulting\ State \\
O &= Output\ (Message) \\
A &= Action(s) \\
E &= Errorhandler
\end{aligned}
\tag{3.4}
$$

Finally, the state machine SM follows Algorithm 1

Algorithm 1 DFA Algorithm

```
Require: None
Ensure: None
  for arc = 1 to current_state.num_out_arcs() do
    if arc(current_state, arc).condition() then
      for act = 1 to act.num_actions() do
        act[act].execute()
      end for
    end if
  end for
```

This model allows for a conversation to specify its continuation, and for nesting of conversations. Since conversations can be examined generally, there is no special contextual relationship between nested conversations and their parents.

This conversation-based approach, along with associated ideas of knowledge management, reasoning and conflict management, were absorbed into a generic agent architecture, dubbed the 'Agent Building Shell' [BF96a, Bar99]. More information on this work can be

found in [BTF97, BF96b, FBG95].

Jackal is most similar to the EIL work in its use of a conversation framework embedded in an overall agent model, and in the similarity of the conversation model used. There are some differences, however: Jackal uses the incoming message to infer the appropriate conversation, whereas COOL expects the conversation type to be explicitly stated. While type inference certainly allows for greater flexibility, it is a very difficult task, and will occasionally err in the face of ambiguity. Also, COOL has a global conversation context. In Jackal, contexts local to conversations can be generally accessed, but only through the hierarchical structure of conversation threads.

Also of note is JAFMAS [Cha97], originally a re-implementation of COOL in Java.

3.2.3 Information Agents

Decker's work in distributed planning [OPLD95, DL94] and task representation [Dec97] embodies the ideal complement to Jackal; a system that can massage a high level problem or goal into smaller subtasks, and schedule their execution appropriately. Taking his expertise to Carnegie Mellon in 1995, he and Sycara created the IA [SDP+96, DPSW97, DS97, DSW97, Dec95, WDS96] (see Figure 3.3). This agent architecture encompasses planning and task decomposition, scheduling, execution and monitoring.

3.2.4 InfoSleuth

The InfoSleuth project [NU97a, JS96, BBC+97] is very much committed to the domain of distributed information retrieval, although the agent architecture is fairly general. We will not discuss their overall system design, which employs a standard resource brokered approach. InfoSleuth implements a Java agent shell, which is carefully separated into cleanly

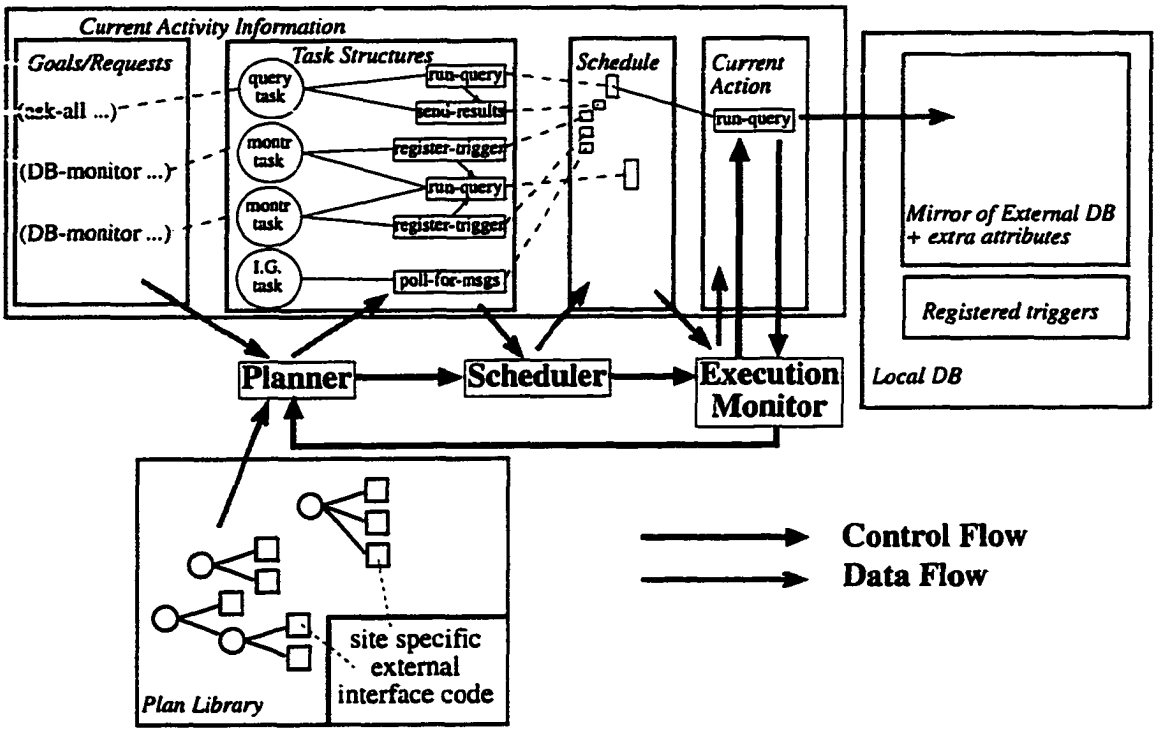


Figure 3.3: Information Agent (Decker) Architecture

interfaced layers: message (astride the Transport Layer), conversation, generic agent, and agent application. The message layer handles message addressing, parameter marshaling, and so forth. The conversation layer imposes language constraints on sequences of messages. The generic agent layer provides the agent application layer with basic services.

Conversation policies enforced roughly correspond to the Finin and Labrou [Lab96b, LF97c] semantics for KQML, but the model used is a basic Deterministic Finite State Automaton (DFA); transitions are determined by performative name only.

3.2.5 KAoS

KAoS [BDBW98] is an agent communication language and open architecture for agents and agent systems. As an architecture, it is remarkably similar to Jackal, providing basic

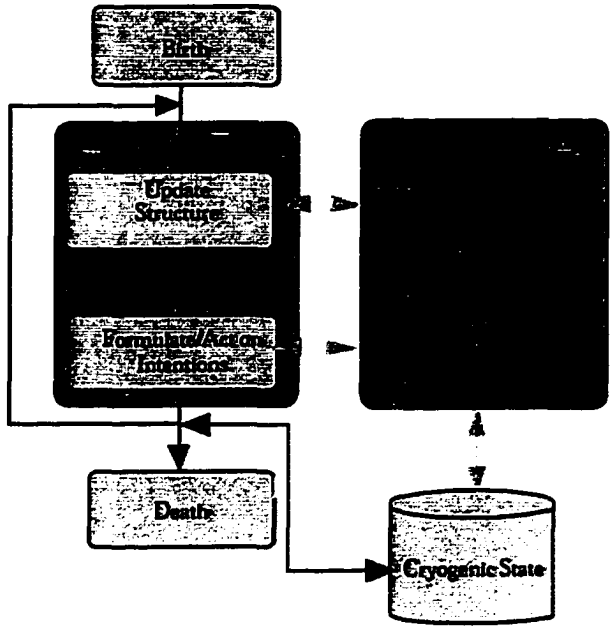


Figure 3.4: KAoS

code support (in Java) for essential agent functions only, and a prescription for the organization of agents into a MAS. As Jackal is independent of the communication languages it supports, the similarity ends here. KAoS, the language, is speech act based.

KAoS agents extend a standard agent shell, unlike Jackal-based agents. The KAoS architecture employs the conversation-based approach to context management. While the actual model is not discussed, it appears on the surface to be a standard DFA, with transitions keyed on the speech act 'verbs', and support for composition.

The initial work on KAoS has recently been expanded into an effort to develop a comprehensive, heterogeneous reasoning system, called the Conversation Design Tool (CDT) [GHB98], within the context of a larger framework for designing agent plans, communication and security [BGH+98]. The CDT project has the goal of identifying several representational models for agent interaction, and providing support for unified reasoning over all of them, through the use of the OpenProof system of Barwise and Etchemendy

[SCB+98, BE98]. The ideas behind this project are similar to those of Protolingua, but are at a significantly higher level. Protolingua does not offer any inference capabilities; only a specification which is amenable to inference within one or several formal systems. In this way, it is much lighter than the proposed CDT. Also, CDT will include representational models which will allow extended reasoning about agents and groups of agents (e.g. Venn Diagrams); the focus in Protolingua is restricted to reasoning about interactions, with the assumption that higher level reasoning and planning will occur at the agent level, if required.

3.2.6 Java Agent Template (JAT)

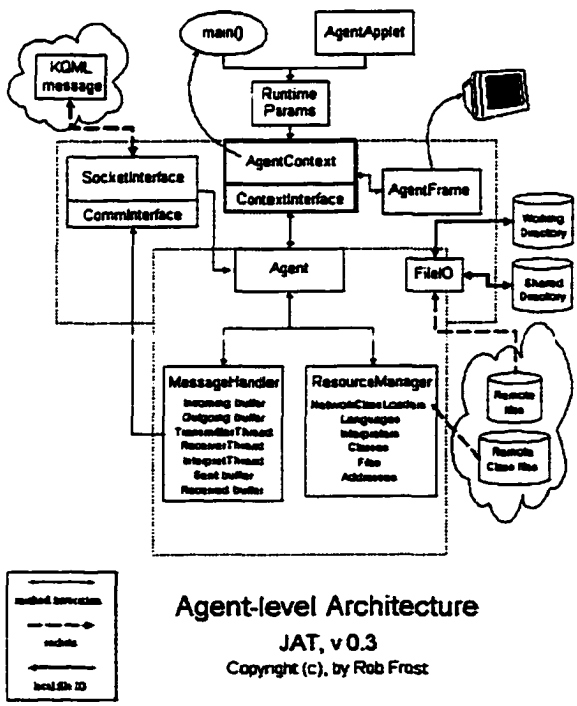


Figure 3.5: JAT Agent Architecture

JAT [Fro99] is essentially a Java implementation of KQML, in the form of an agent

shell. JAT agents can run stand-alone, or as applets with some restrictions. Basic message passing is supported for KQML. An ANS is used to coordinate agents, and the AEE is the basic Java Virtual Machine (JVM). JAT is intended to be flexible yet comprehensive, and so can be a bit cumbersome. Agents developed with JAT are tightly integrated with the agent shell.

3.2.7 JATLite

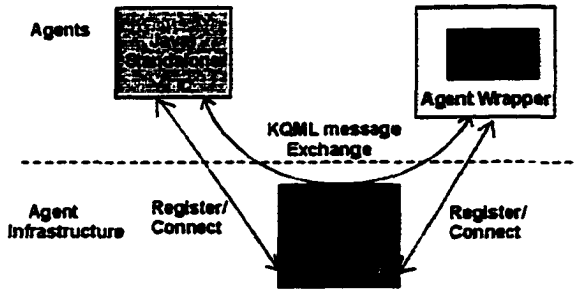


Figure 3.6: JATLite Framework

JATLite [Pet98] is a successor to JAT, so we will describe only the differences with respect to the original. JATLite is intended to be a much lighter-weight package, and is suitable for use with applets. Of note is its Router facility: applet agents can communicate with other applet agents by sending messages back through an associated Router or Routers (communication by proxy). The Router buffers undeliverable messages, and is supported by a standard ANS. JATLite agents in general are not constrained to communicate through the Router.

3.2.8 Zeus

Zeus [NNLC99] is a toolkit for building complete agents, from the ground up. It consists of a component library, a suite of visual design tools, and a set of predefined utility agents. The components are designed such that their behaviors are largely defined declaratively, and can be changed at runtime. Agents constructed with Zeus typically have components which facilitate planning and reasoning, communication (via KQML), and which provide a collection of interaction protocols. Agents and MASs are created through a process of task and relationship specification. The resulting entities are produced as Java source code, for independent compilation and execution.

3.2.9 AgentBuilder

Reticular Systems, Inc., has developed a commercially successful platform, AgentBuilder [RS99] for constructing agents based on their RADL (Reticular Agent Definition Language). RADL is an extension of PLACA [Tho94] and AGENT-0 [Sho91], and views the agent as a core of behavioral rules, constrained by beliefs, capabilities, commitments and commitment rules.

3.3 Other Related Work

We have discussed some of the more relevant projects; there are many more; among them Open Agent Architecture (OAA) [CCWB94, MC95, CJ95, MCL96, MOMC97, MCJ+97, MCM98], Mobile Objects and Agents (MOA) [MAC+98], Odyssey [Gen98], Tacoma [JvRS95a, JvRS95b, JSvR98, JSvR97, JvRS96, Sch97, MvRSS96], Voyager [Obj97a, Obj97d, Obj97c, Obj97b, Obj98, Gla98], and Walk the Talk [Joh98]. Each of the systems

described provides agent developers with tools for endowing agents with various combinations of desirable characteristics. And yet, although communication is the basis for interaction and collaboration, [1] few if any of these systems supports intercommunication. This is due to use of custom languages or extensions to standard languages, specialized communication protocols, incompatibility of environments, and so forth. The gap is larger in some cases than in others, and could be remedied by extension, but more fundamental advances will have to be made in order to assure more complete interoperability.

[1] This assumption is not universally held.

Chapter 4

Conversation-Based Specification of Interaction

4.1 Introduction

The formal specification of agent behavior, communicative or otherwise, is an important step towards agent interoperability. This chapter discusses the facets of conversation specification which make it such a valuable tool to the agents community, and then looks at the requirement for the underlying formal model, reviewing several candidate systems.

4.2 Motivation

The study of ACLs is one of the pillars of current agent research. KQML and the FIPA ACL are the leading candidates as standards for specifying the encoding and transfer of messages among agents. While KQML is good for message-passing among agents, the message-passing level is not actually a very good one to exploit directly in building

a system of cooperating agents. After all, when an agent sends a message, it has expectations about how the recipient will respond to the message. Those expectations are not encoded in the message itself; a higher-level structure must be used to encode them. The need for such conversation policies is increasingly recognized by the KQML community [Lab96b, LF97a, LF97b, LF97c], and has been formally recognized in the latest Foundation for Intelligent Physical Agents (FIPA) draft standard [FIP97, Dic97].

It is common in KQML-based systems to provide a message handler that examines the message performative to determine what action to take in response to the message. Such a method for handling incoming messages is adequate for very simple agents, but begins to break down as the range of interactions in which an agent might participate increases. Missing from the traditional message-level processing is a notion of message context.

A notion growing in popularity is that the unit of communication between agents should be the conversation. A conversation is a pattern of message exchange that two (or more) agents agree they will follow in communicating with one another. In effect, a conversation is a communications protocol, albeit one that may be initiated through negotiation, and may be short-lived relative to the way we are accustomed to thinking about protocols. A conversation lends context to the sending and receipt of messages, facilitating more meaningful interpretation. The adoption of conversation-based communication carries with it numerous advantages to the developer, including:

- A better fit with intuitive models of how agents will interact than is found in message-based communication.
- A closer match to the way that network research approaches protocols, which allows both theoretical and practical results from that field to be applied to agent systems.
- Separation of conversation structure and the actions to be taken by an agent engaged

in the conversation. This allows the same conversation structure to be used by more than one agent, in more than one context. In particular, two agents can use the same conversation structure to ensure that they will engage in the same dialogue.

- The standard advantages of the underlying ACL, including language-independence and ontology-independence.

Until very recently, little work has been devoted to the problem of conversation specification and implementation for mediated architectures. Increased interest is evidenced by the advent of a workshop on conversation policies at the Third International Conference on Autonomous Agents, in 1999. Strides must be taken to make conversation specifications easy to encode and reuse. Additionally, libraries of specifications must be compiled, along with an ontologies of conversations.

To achieve these goals, we must solve three main problems:

1. Conversation specification: How can conversations best be described so that they are accessible both to people and to machines?

2. Conversation sharing: How can an agent use a conversation specification standard to describe the conversations in which it is willing to engage, and to learn what conversations are supported by other agents?

3. Conversation aggregation: How can sets of conversations be used as agent 'APIs' to describe classes of capabilities that define a particular service or capability?

4.2.1 Conversation Specification

A specification of a conversation that could be shared among agents must contain several kinds of information about the conversation and about the agents that will use it. First, the

sequence of messages must be specified. One popular solution that has many advantages is the use of deterministic finite-state automata (DFAs) for this purpose; DFAs can express a wide variety of behaviors while remaining conceptually simple. Next, the set of roles that agents engaging in a conversation may play must be enumerated. For example, a conversation that allows a sensor to report an unusual condition to all interested agents might have two roles: sensor and broker (which would in turn be specializations of sentinel and sentinel-consumer roles). Many conversations will be dialogues, and will specify just two roles; conversations may have more than two roles, however, and represent the coordination of communication among several agents in pursuit of a single common goal.

DFAs and roles dictate the syntax of a conversation, but say nothing about the conversation's semantics. The ability of an agent to read a description of a conversation, then engage in such a conversation, demands that the description specify the conversation's semantics. However, reliance on a full-blown, highly expressive knowledge representation language may limit a specification's usefulness. We believe that a simple ontology of common goals and actions, together with a way to relate entries in the ontology to the roles, states, and transitions of the conversation specification, will be adequate for many basic purposes. This approach sacrifices expressiveness for simplicity and ease of implementation. It is nonetheless perfectly compatible with attempts to relate conversation policy to the semantics of underlying performatives, as proposed for example by [Bra96, BDBW98]. Most complex interactions, however, will require the use of a model that is more expressive, but which retains many of the positive features of DFAs; we will return to this later.

The capabilities we have outlined will allow the easy specification of individual conversations. To develop systems of conversations though, developers must have the ability to extend existing conversations through specialization and composition. Specialization is the ability to create new versions of a conversation that are more detailed than the original

version; it is akin to the idea of inheriting a subclass in an object-oriented language. Composition is the ability to combine two conversations into a new, compound conversation. Development of these two capabilities will entail the creation of syntax for expressing a new conversation in terms of existing conversations, and for linking the appropriate pieces of the component conversations. It will also demand solution of a variety of technical problems, such as naming conflicts, and the merger of semantic descriptions of the conversations.

4.2.2 Conversation Sharing

A standardized conversation language, as proposed above, dictates how conversations should be represented; however, it does not say how such representations are to be shared among agents. While the details of how conversation sharing is accomplished are more mundane than those of conversation representation, they are nevertheless crucial to the viability of dynamic conversation-based systems. Three questions present themselves:

- How can an agent map the name of a conversation to the specification of that conversation?
- How can one agent communicate to another the identity of the conversation it is using?
- How can an agent determine what conversations are handled by a service provider that does not yet know of the agent's interest?

4.2.3 Conversations Sets as Application Programmer Interface (API)s

The set of conversations in which an agent will participate defines an interface to that agent. Thus, standardized sets of conversations can serve as Abstract Agent Interfaces (AAI), in much the same way that standardized sets of function calls or method invocations serve as APIs in the traditional approach to system building. That is, an interface to a particular class of service can be specified by identifying a collection of one or more conversations in which the provider of such a service agrees to participate. Any agent that wishes to provide this class of service needs only to implement the appropriate set of conversations. To be practical, a naming scheme will be required for referring to such sets of conversations, and one or more agents will be needed to track the development and dissolution of particular AAIs. In addition to a mechanism for establishing and maintaining AAIs, standard roles and ontologies, applicable to a wide variety of applications, will also be required.

As mentioned, until recently there has been little work on communication languages from a practitioner's point of view. If we set aside work on network transport protocols or protocols in distributed computing (e.g., CORBA) as being too low-level for the purposes of intelligent agents, the remainder of the relevant research may be divided into two categories. The first deals with theoretical constructs and formalisms that address the issue of agency in general and communication in particular, as a dimension of agent behavior (e.g., Agent Oriented Programming (AOP) [Sho93]). The second addresses agent languages and associated communication languages that have evolved to some degree to applications (e.g., TELESCRIPT [Whi95], now Odyssey [Gen98]). In both cases, the bulk of the work on communication languages has been part of a broader project that commits to specific architectures.

Agent communication languages like KQML provide a much richer set of interaction

primitives (e.g., KQML's performatives), support a richer set of communication protocols (e.g., point-to-point, brokering, recommending, broadcasting, multicasting, etc.), work with richer content languages (e.g., Knowledge Interchange Format (KIF)), and are more readily extensible than any of the systems described above. However, as discussed above, KQML lacks organization at the conversation level that lends context to the messages it expresses and transmits. Limited work has been done on implementing conversations for software agents, and almost none has been done on expressing those conversations. As early as 1986, Winograd and Flores [WF86] used state transition diagrams to describe conversations. The COOL system [BF95] has perhaps the most detailed current state transition-based model to describe agent conversations. Each arc in a COOL state-transition diagram represents a message transmission, a message receipt, or both. One consequence of this policy is that two different agents must use different automata to engage in the same conversation. We believe that a conversation standard should clearly separate message matching from actions to be carried out when a match occurs; doing so will allow a single conversation specification to be used by all participants in a conversation. This, in turn, will allow conversation specifications to describe standard services, both from the viewpoint of the service provider, and from that of the service user.

COOL also uses an :intent slot to allow the recipient to decide which conversation structure to use in understanding the message. This is a simple way to express the semantics of the conversation. We argue below that more general descriptions of conversation semantics will be needed if agents are to acquire and engage in new conversations on the fly. The challenge will be to develop a language that is general enough to express the most important facts about a conversation, without being so general that it becomes an intellectual exercise, or too computationally expensive to implement.

Other conversation models have been developed, using various approaches. Extended

FSM models, which, like COOL, focus more on expressivity than adherence to a model include Kuwabara et al. [KIO95, Kuw95], who add inheritance to conversations; Wagner et al. [WBLX99]; and Elio and Haddadi [EH99], who defines a multilevel state machine, or Abstract Task Model (ATM). A few others have chosen to stay within the bounds of a DFA, such as Chauhan [Cha97], who uses COOL as the basis for her multi-agent development system, [1] Nodine and Unruh [NU97a, NU97b], who use conversation specifications to enforce correct conversational behavior, and Pitt and Mamdani [PM99], who use DFAs to specify protocols for Beliefs, Desires and Intentions (BDI) agents. Also using automata, Martin et al. [MPRA99] employ Push-Down Transducers (PDT). Lin et al. [LNSK99] and Cost et al. [CCF+99] demonstrate the use of Colored Petri Nets, and Moore [Moo99] applies state charts. Parunak [Par96] employs Dooley Graphs. Bradshaw [Bra96] introduces the notion of a conversation suite as a collection of commonly used conversations known by many agents. Labrou [Lab96b] uses definite clause grammars to specify conversations. While each of these works makes contributions to our general understanding of conversations, more work must be done in getting agents to share and use conversations.

4.2.4 Defining Common Agent Services via Conversations

A significant impediment to the development of agent systems is the lack of basic standard agent services that can be easily built on top of the conversation architecture. Examples of such services are: name and address resolution; authentication and security services; brokerage services; registration and group formation; message tracking and logging; communication and interaction; visualization; proxy services; auction services; workflow services; coordination services; and performance monitoring services. Services such as these have

[1] More recent work with this project, JAFMAS, explores conversion of policies to standard Petri Nets for analysis [GB99a].

typically been implemented as needed in individual agent development environments. Two such examples are an agent name server and an intelligent broker.

Agent Name Server

At first blush, the problem of mapping from an agent name to information about that agent (such as its address) seems trivial. However, solving this problem in a way that can easily scale as the number of users and amount of data to be processed grows is difficult. We believe that development of a successful symbolic agent addressing mechanism demands at least two advances:

1. A simple naming convention to place each role an agent might play in an organization at a unique point in a namespace for that organization. Currently there is no widely accepted mechanism for universal unique agent naming (in the way that there now is, e.g., for Internet hosts or web documents).

2. An efficient, scalable name service protocol for mapping from symbolic role names to information about the agents that fill those roles.

   The proposed KNS (Section 5) meets both of these demands.

To a large extent, the desired techniques can be modeled after existing name service techniques such as the Domain Naming Scheme (DNS) (which is widely implemented) and CORBA (whose namespace mechanisms are only narrowly implemented). Such techniques are well studied, highly reliable, and scalable. Agent name service will differ from DNS primarily in that agents will tend to appear, disappear, and move around more frequently than do Internet hosts. This will necessitate the development of naming conventions that are less rigid than those used in DNS, and algorithms for mapping from names to agent information that do not rely on the static local databases found in DNS.

Intelligent Broker

A system that is to respond to the demands of multiple users, with needs that vary over time, under an ever-increasing query load must be able to do on-the-fly matching of queries to documents and services. With respect to agent-based architectures, that implies the ability to dynamically discover other agents based on the content of their knowledge. It should exploit the research on conversations and the symbolic agent-addressing scheme described above, while at the same time fitting neatly into existing brokered systems. Such systems will continue to see a single broker where they had a single broker all along; now, however, that broker will have the option of coordinating many other disparate brokers of varying capabilities.

4.3 Implementing Conversation-Based Formalisms

While formal methods are typically and best applied to relatively small, well understood domains, software agents research typically involves the integration of ideas from a number of areas and problem solving techniques that are often more heuristic than algorithmic. In this section, we discuss the advantages and disadvantages of combining the two, and show that in some areas, the marriage can be quite fruitful.

In this work, we restrict our attention to agent interaction as agent communication, and claim that the two are equivalent, based on the assumption that any action or perception can be expressed in some linguistic form, or at least at a linguistic transmission of some complex data. Consider, for example, an agent whose perceptual models translate all signals into messages that it communicates to itself. Extending this line of reasoning one step further, we can characterize all agent behavior as communicative interaction, by ob-

serving that agents can be decomposed into modules whose interaction can be restricted to communication.

The conversation-based approach to agent modeling supports the bundling of messages into conversational units with a common context. We will adopt this approach, and discuss it at length below.

4.3.1 Why Formalize the Specification of Agent Behavior?

The formal specification of agent behaviors offers a number of tangible benefits, including: The ability to verify the behavior of an agent before it is executed, the potential for reasoning about behaviors and their composition, and the possibility of behavior reuse. These benefits will assist us in providing a convenient, standard representation for agents specification, which facilitates agent coding from specification, and which allow formal and probabilistic analysis of agents and systems.

4.3.2 Requirements

A formal specification scheme should meet the following requirements:

- Completeness/Generality. The system should be sufficient to allow the specification of a wide range of interactions. This implies varying levels of abstraction as well.
- Independence. It should be portable across platforms, not tied to any specific machine implementation.
- Efficiency. The use of this scheme should not require space and compute resources that would make it, for most common cases, impractical.

- Simplicity. In order for a system to be widely accepted, it must be clear enough to merit its adoption in favor of other approaches.
- Adaptivity. While the conversation-based approach should guide interaction, interaction will not always conform to some expected structure. The system must be capable of coping with unexpected or dynamic situations.
- Scalability. Efficiency is gained by composition of existing conversations into larger, more complex ones.

Some of the above requirements can easily be or have already been met. For example, some degree of platform independence can be achieved by implementing interpreters in a relatively platform independent language such as Java. More progress can be made by developing an interpreter algorithm that is fairly straightforward, and can be implemented in most any programming language. Others will not be completely solved within the scope of this work. In this work, we will describe: A framework for modeling agent interaction, an ontology for agent interaction and communication, and a specification language for agent interaction protocols.

4.3.3 Desirable Properties of a Protocol Description Language

A language for describing CPs should be of maximal use to both the designed and developer, supporting abundant expression and good design and verification techniques. Its properties should include:

- Accessibility; Should be relatively easy to learn and use.
- Should allow for the specification of all relevant behaviors.

- Support the use of tools for formal and probabilistic analysis.
- Allow modification and (de)composition of protocols.
- Support online reasoning about protocol selection.
- Have a mapping to executable form, or be directly executable.
- Simplicity.

Formal specification and methods have been applied to other level of agent design as well. Perhaps the best example is the DESIRE project [BKJT95], which has for years been working on the formal verification of distributed knowledge bases. This work is very important for agent research, but is at a more abstract level than what we are trying to accomplish. The protocols that we consider could be hand tooled by a human, or inferred from first principles by a high level reasoning system.

A comprehensive system for conversation-based interaction would greatly facilitate the development and use of autonomous agents in the human community. As the conversation-based approach grows in popularity, it is essential that we find a common framework within which all agents can cooperate. This is the motivation behind the development of Protolingua.

4.3.4 Underlying Formal Computational Model

Conversation Models

The core of this theory will be the formalism chosen for modeling conversations. The choice of formal system has great impact on the value of the specification scheme, in terms

of its accessibility, expressivity, and support for verification. Below, we examine a number of systems that are promising candidates.

The Status Quo: Deterministic Finite State Automatons As discussed above, a popular model is the DFA (Equation 4.1).

$$
\begin{aligned}
&DFA\ M = (Q, \Sigma, \delta, q_0, F), where: \\
&\quad Q = Finite\ set\ of\ states \\
&\quad \Sigma = Input\ alphabet \\
&\quad \delta = Function\ Q \times \Sigma \rightarrow Q \\
&\quad q_0 = Initial\ state;\ q_0 \in Q \\
&\quad F = Set\ of\ final\ states;\ F \subset Q
\end{aligned}
\tag{4.1}
$$

This model is simple and easily interpreted, and effectively captures the essence of simple, sequential protocols. While this has proven to be of great use to date, it is not sufficiently powerful to model conversations which involve an arbitrarily large or changing number of participants or which involve broadcast messages. Also, the term DFA is heavily misused in the agents literature. The formal definition of a DFA is given in Equation 4.1.

Most models of conversation that have been referred to as DFAs differ in two respects. First, the inputs are not discrete elements of an alphabet, but complex expressions, and are evaluated with complex matching expressions or procedures. Expressions need not be mutually exclusive, admitting non-determinism in the form of order dependence. Second, a random access read/write store is often available to the procedures. This modified machine is effectively an augmented transition network (ATN), and as such Turing equivalent. [2]

[2]We use ATNs here to broadly encompass all automata extended in such a way that they are equal in

While an ATN is far more expressive than a DFA, it is much more difficult to analyze formally.

We will next examine briefly a number of formal models that could potentially be used as a basis for conversation specification: CPNs, Evolving Algebra (EA)s, temporal logic and π-calculus. Each of these has merits of varying degree with respect to the properties required, so it is not likely that one will appear to be clearly superior to the others. Rather, the best approach may involve a combination of two or more models and algorithms for interconversion. Conversation specifications could include the model in which the conversation was to be interpreted.

Augmented Transition Networks The ATN expands on the FSM by adding recursion (to support Context Free Grammars), and various extensions such as registers, tests and arbitrary functions or actions. While the ATN is a very expressive model, it is not a very good model for concurrency. As it is used currently, the state of the ATN must represent the joint state of both participants in a dialogue; the implied, necessary assumption is that both parties transition from state to state simultaneously. This is not a valid assumption for asynchronous systems. Consequently, both parties must follow the same path, meaning that the ATN state represents the joint state for both participants. This approach does not scale well. If conversations of more than two participants were to be described, the form of the ATN used to model a conversation would be dependent on the number of participants, and for m participants with n local states each, an ATN with n^m states would be required.

Let us consider the following informal extension to the ATN, in support of concurrency (Equation 4.2):

power to a Turing machine (e.g. a DFA with random access store). The ATN is transition-diagram based, and includes many extensions commonly made to basic automata, making it an easy target.

$$
\begin{aligned}
&DFA\ M = (Q, \Sigma, \delta, q_0, F, P), where: \\
&\quad Q = Finite\ set\ of\ states \\
&\quad P = Finite\ set\ of\ participants \\
&\quad \Sigma = Input\ alphabet \\
&\quad \delta = Function\ Q \times \Sigma \times P \rightarrow Q \\
&\quad q_0 = Initial\ state;\ q_0 \in Q \\
&\quad F = Set\ of\ final\ states;\ F \subset Q
\end{aligned}
\tag{4.2}
$$

Here, we provide for a set of participants P, each with individual state. δ computes the state change for a single participant based on its current state and input. q_0 denotes the initial state for all participants. Note that we have not yet considered the problem of synchronization among participants.

Before going any further, we should note that this model looks remarkably like the CPN model (discussed below). Rather than recreating this powerful model, we will instead plan to consider the CPN itself as an alternative.

Executable Temporal Logic Temporal logic provides a powerful basis for modeling systems. It is also, however, very computationally expensive. Satisfiability for propositional temporal logic is PSPACE-complete, and for first-order temporal logics is Σ_1^1-complete. In order to make executable temporal logics feasible, designers have dealt with this problem by either restricting expression in the logic itself, or by applying advanced heuristics to the theorem proving mechanism. We will not expand on this topic; see [Mer95, FO95] for an excellent survey of executable and modal logic. A good example, which has been used suc-

cessfully in quite a number of systems, including agent systems, is MetateM/Concurrent MetateM [KF97, Fis94, BFG+95].

Process Calculi Process algebras allow for very thorough modeling of complex, concurrent systems. There is an extensive literature on their use, and the wide variety of tools generally available makes them an attractive modeling candidate. Two of the best known are Communicating Sequential Processes (CSP) [Hoa85, Ros97, HJ95] and the Calculus of Communicating Systems (CCS) [Mil80]. Although they provide very powerful means for systems analysis, CCS and CSP are not executable languages.

An extension of CCS that allows for dynamic change in systems structure is Milner's π-calculus [MPW89a, MPW89b]. Since we would like to model conversations with dynamic structure, this is a significant improvement over CCS. While we make significant gains in expressivity, the actual notation of the π-calculus or polyadic π-calculus (see [Mil91]) is somewhat more burdensome. The syntax and semantics are less accessible than those of state based models, such as Petri Nets, and interpreters will be more complex. Far fewer tools for the use of monadic π-calculus exist than for CPNs, for example.[3]

EA Evolving Algebras (EA) [Gur93, Gur95], or Abstract State Machines (ASM) are the result of an effort by Gurevich to create a formal model of computation which is closer to actual system specification than the Turing Machine (TM). Essentially, the ASM differs from the TM in that it simulates each step of the modeled algorithm in a bounded number of steps of the underlying ASM. Further, the ASM has the advantage over similar abstract machines [KU58, Sch80] in that it can be used to model any or varying levels of abstraction. EAs have been used to specify the C and Prolog languages and the architectures for the

[3]To date, the author is only aware of one: the Mobility Workbench [VM94, Vic95], which runs under New Jersey SML.

PVM and the Transputer, and to validate the standard language implementations of Prolog and Occam, just to cite a few examples. Models can be run on an EA interpreter, of which there are several implementations [HM]. See [Bör95] for motivation behind using EAs and [BGR95] for a demonstration of the EA specification of the familiar Bakery Algorithm.

Of particular interest to this project is that the π-calculus can in fact be faithfully modeled as an EA; this is demonstrated by Glavan and Rosenzweig [GR93] in their development of a theory of concurrency for the EA framework. Also of potential interest is the current effort to specify Java and the JVM using the ASM methodology [BS98, Wal97, KP97].

A very useful source of information on EAs in an annotated bibliography by Börger [BH98].

Petri Nets Petri Net (PN)s [Pet77, Age79] offer a very simple language with which to model a wide range of systems, with true concurrency. However, because of the simplicity of the modeling language, it is often time consuming and complex to model higher level processes, such as communication.

CPNs [Jen92, Jen94a, Jen94b, Jen96, Jen97b, Jen97a] are an extension which allow tokens in a PN to be 'colored' with arbitrarily complex data. While CPNs are equivalent in expressive power to PNs, they make it possible to model systems that would otherwise require a prohibitive number of states. Figure 4.1 depicts a CPN describing a simple communication protocol.

4.3.5 Conversational Ontology

The use of an ontology provides a standardized, common body of knowledge about the domain in question; in this case, agent interaction. See [GTW92, Gru93, Gru92, Gua94, GCG94] for discussion of ontologies as portable, common knowledge representation mechanisms. Ontologies provide a common vocabulary for referring to a set of objects and

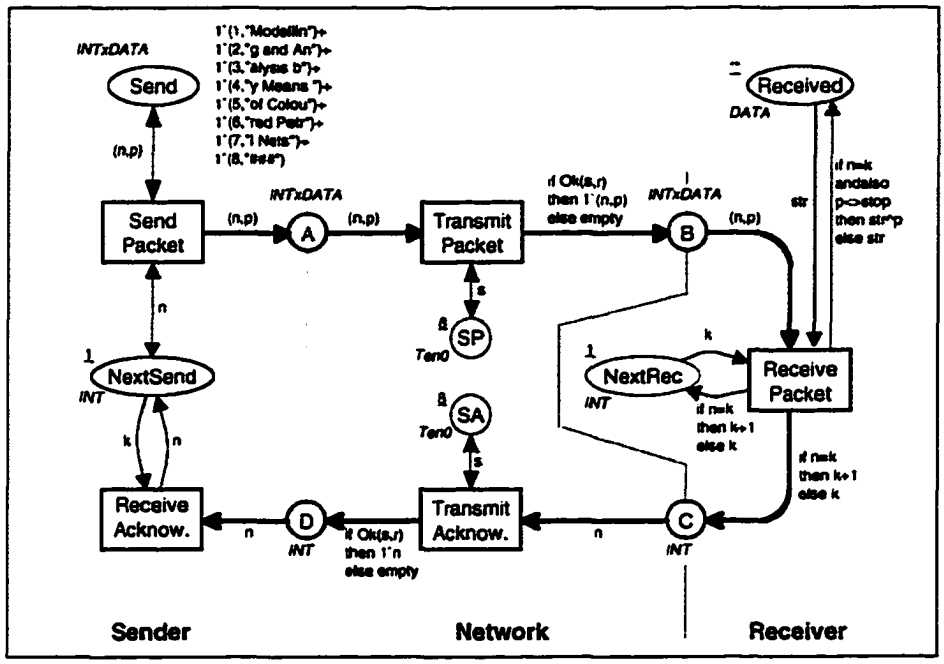


Figure 4.1: CPN describing a simple communication protocol. Rectangles denote actions, and ovals places in the traditional PN notation.

relationship among them, usually within some prescribed domain. They therefore make it possible to work concisely within a domain (e.g. pose unambiguous statements), and provides a means for those in other domains to interpret yours (e.g. facilitates translation of your statements). In this work, we will discuss two 'levels' of ontology; a higher level ontology of conversations, which captures the domain of agent communicative interaction, and a lower level ontology of conversation models. The latter is used to talk about computational models, specification components, interpreters, ans so on. The Nishida Lab at the Nara Institute of Technology has done considerable work in the creation and use of ontologies in agent-based systems [TIT+96, TIN95, TTN97, ITN96]. Tools for constructing ontologies have been developed, such as Stanford's Ontolingua ontology editor, an online research system [FFR96].

4.3.6 A Language for Specifying Behavior

The specification language should allow us to describe agent protocols in a formal way, using all of the machinery we have defined above. The language should be sufficiently general so that it can be used across various platforms. For example, although the language will initially be used within a Java -based agent system, protocol specifications should not access Java classes or invoke Java methods directly, but should rely on functionality provided by the language interpreter.

4.4 Summary

The conversation paradigm is a very useful way of organizing communication among groups of agents, lending context to the sending and receipt of messages, thereby facilitating more meaningful interpretation. Other benefits include a better fit with intuitive models of agent interaction, and separation of conversation structure and the corresponding actions. The choice of underlying model significantly affects the value of various aspects of the representation, such as expressiveness and verifiability. A specification language for the representation, in combination with a an ontology of conversations, would facilitate the exchange, use and reuse of CPs across systems of agents.

Chapter 5

KNS

Communication is a central problem in distributed systems. Without the ability to communicate, a set of agents would be merely a collection of isolated components. All classic distributed systems problems, such as coordination and distributed reasoning, depend on an underlying communication framework. Before communication can take place, there must be a known destination. KNS adds a communication layer in which symbolic names are mapped to actual transport addresses. In addition, however, it offers advanced support for dynamic group formation and disbanding, and maintenance of persistent, distributed agent identity. KNS is currently being used within Jackal, UMBC's Java-based agent development framework. This chapter introduces the basic concepts underlying KNS.

The problem of agent naming is central to agent communication. We would like to be able to talk about agents with reasonable certainty that we are all discussing the same ones, and we would like to be able to send messages to agents that we know by name. The former statement argues that names should be unique, within some context, and the latter that they should be resolvable into addresses which can be used by our underlying transport mechanism. This can be accomplished by having the address either implicitly or explicitly

encoded in the name, or by providing a service to perform the resolution.

We can think of the problem in three layers of abstraction. At the top is the agent's identity, that which differentiates it from all other agents. One step below this is the name, and at the base is the address. Although an agent's identity will never change (by definition), its name(s) may, based on changing roles or associations. Addresses may change even more frequently, because of physical relocation or constraints of the underlying operating system. This dynamism argues for the use of service-based resolution (SBR) between both layers. In addition, SBR allows for the use of symbolic names with useful meaning.

Uniqueness is a more difficult problem. It is trivial to assign agents simple unique identifiers (e.g. serial numbers) from some central authority. However, if an agent holds such an identifier, it must still present the tag to some authority for resolution. An address, for instance a URL, eliminates the need for SBR altogether, but ties the agent to that address. We propose to represent an agent's identity by the collection of names it uses which we call the persistent distributed identity (PDI). This set can change as names are added or removed, but it remains a constant reference point for the agent itself. Protocols added to the basic agent registration scheme maintain the PDI with little overhead. In addition to the benefits of identity, the scheme provides a valuable mechanism for storing and retrieving information relating to the agent such as certificates.

KNS is a set of protocols for agent naming and addressing. They were developed and used as a basis for the design of Jackal 3.0. This section provides an overview of KNS.

The KNS covers several layers of abstraction, and provides basic support for agent operation. It should be noted that the KNS protocols are layered on top of the KQML, or linguistic, layer.

First, some definitions:

3 DEFINITION (GIVEN NAME) *A name chosen for the agent application by itself or some other authority.*

4 DEFINITION (LOCAL NAME) *A Given Name qualified by a numeric index, and assigned by a Domain Registrar upon registration.*

5 DEFINITION (FQAN) *Fully-Qualified Agent Name, the canonical form for names in KNS. Every Fully-Qualified Agent Name (FQAN) names a domain.*

6 DEFINITION (DOMAIN) *A virtual group, defined by registration and unregistration, and managed by the owner of the FQAN which names it.*

7 DEFINITION (OPEN DOMAIN) *A Domain which will accept registrations.*

8 DEFINITION (CLOSED DOMAIN) *A Domain which will* not *accept registrations.*

9 DEFINITION (REGISTRAR) *The agent which manages a given Domain.*

10 DEFINITION (REGISTRANT) *An agent registered with a given domain.*

11 DEFINITION (ALIAS) *A FQAN that is owned by the same agent as another FQAN is an alias for that FQAN.*

12 DEFINITION (ALIAS SET) *Also 'Alias Net'; for an agent A, the set of domains with which A is registered.*

13 DEFINITION (AS) *Agent Server; holds a database of information for given agent.*

14 DEFINITION (PAS) *Primary AS; there is only one for any given agent.*

15 DEFINITION (SAS) *Backup (secondary) AS; serve as backup to Primary Agent Server (PAS). There can be any number.*

5.1 Assumptions

KNS makes some basic assumptions about the environment in which it is used:

- Message delivery by the underlying transport mechanisms is reliable. The protocols do not incorporate any retry mechanism for delivery failure to a specified address. Further, if KNS protocols are properly implemented, including strong message delivery, an agent may be consider unreachable if an expected acknowledgment is not received on a single transmission.

- Authentication (KNS does not specify what kind) in the message transport layer assures that the name in the sender field of the message is in fact the sender of the message. Security in KNS is identity based, so any privileges enjoyed by the named sender are applied to the accompanying transaction.

- Agents purporting to implement KNS correctly and responsibly render services as appropriate.

- It is possible to distinguish agents that implement KNS from those that do not. This relates to the general problem of determining an agents language or message format. Initially, an agent should be given the name of another with which to register; that agent does implement KNS, as do any agents located through KNS name/address resolution. However, new contacts in unrelated systems may not.

5.2 Agent Names

The foundation of KNS is its agent-naming scheme. It encompasses both symbolic and direct (URL-based) names. The symbolic component is modeled after the DNS

scheme [Moc87a, Moc87b], and extends it to allow a Uniform Resource Locator (URL) to fill the root position of a name. A FQAN is defined as follows:

$$GivenName = [a-zA-Z0-9\_]\{1,64\} \quad (5.1)$$

$$NameIndex = 0 \mid ([1-9][0-9]\{,10\}) \quad (5.2)$$

$$LocalName = < GivenName >< NameIndex > \quad (5.3)$$

$$FQAN = (< LocalName > .) * (< GivenName > | < LocalName > | < URL >) \quad (5.4)$$

As in DNS, names registered within a Domain must be unique. Rather than accepting only applications for unique names, KNS adopts the policy of accepting any name and adding a distinguishing suffix. Some examples of FQANs are: bob[4].ans, freida, barbecue[34].cs and fred[2].http://www.umbc.edu/. Since names correspond to entity/Domain relationships, an agent may have any number of names, and may use them interchangeably.

In light of our earlier discussion of name uniqueness, it should be clear that this definition allows for unqualified names. This is included as a convenience, since many contained systems use well-known names for common resources. We assume then that unqualified names are used only in closed contexts in which the address of name root is publicly known. In general, the use of fully qualified names is preferred. The technically correct definition of a FQAN is:

$$GivenName = [a-zA-Z0-9\_]\{1,64\} \quad (5.5)$$

$$NameIndex = 0 \mid ([1-9][0-9]\{,10\}) \quad (5.6)$$

$$LocalName = < GivenName >< NameIndex > \quad (5.7)$$

$$FQAN = (< LocalName > .)* < URL > \quad (5.8)$$

Every FQAN represents a Domain. Thus, an agent can 'have', or manage, multiple Domains, although none is required to actively accept Domain registrations. An agent registers with a Domain either with its Given Name, or under another FQAN that it holds. In the latter case, protocols are engaged to update the Alias Set for that agent. In either case, the agent is given a new FQAN, which is derived from the Given Name of the name submitted. For example, if an agent registers orianus.local with freckles.cs[1].umbc.ans (alternatively, freckles.cs[1].umbc.http://jackal.cs.umbc.edu/ans), it may receive the FQAN orianus[14].cs[1].umbc[23].ans.

An alternative is to represent the name of an agent as an actual URL. While this would be enormously convenient, it creates unacceptable naming ambiguity. Any URL should be usable as a legal agent name, for reasons of flexibility, and for compatibility with systems that use only URLs as agent names. Given that constraint, it becomes impossible to determine which portion of a URL constitutes the root, and which the domain. For example, http://jackal.cs.umbc.edu/ans.umbc.cs[1].freckles could indicate four different names, depending on where one decided the root name ended. It is difficult to remedy this problem directly without abusing or outright violating the URL syntax.

5.3 KNS Architecture

KNS is served by a dynamic, distributed database system, depicted in Figure 1. The two databases maintained are the Domain Registry (one for each Domain), and the Agent Registry (one for each agent, and one or more backups). We impose one additional virtual structure on the name hierarchy, called the Alias Set (or AliasSet). The Alias Set consists of all Domains with which an agent has ever registered. One Domain is designated as the Primary Agent Server (PAS), and it hosts the primary agent registry. Likewise, a Backup Agent Server (BAS) hosts the backup agent registry. All Agent Servers (AS) maintain a reference to the target agent's PAS and BASs. ASs are arranged in a star configuration in order to minimize messaging overhead. The AliasSet itself is treated as a single entity; queries are directed to any member, and if necessary, are forwarded directly to the PAS. Member agents notify the PAS of any changes, and the PAS broadcasts updates to the remaining members of the set.

One dependency is that agent information is not discarded. While this is not entirely realistic, is means that agents can be located most of the time, and that more resources can be dedicated to specific localities to increase the level of fault tolerance. For example, under the KNS scheme, if agent bob.erols.ans unregisters from erols.ans, it will still be possible to locate bob through the erols.ans domain. If erols.ans terminates, and ans has lifted its domain, location is still possible. However, if erols.ans goes down catastrophically or otherwise dissolves the domain, it will not be possible to reach bob via its previous name. Agents who are concerned with reachability would therefore prefer to register with strong domains, and would show preference for names that they felt would more reliably persist. This situation could be improved by allowing agents to register, have included in their address information, or send with messages an alternate name; this is reminiscent of the

use of sender and reply-to fields.

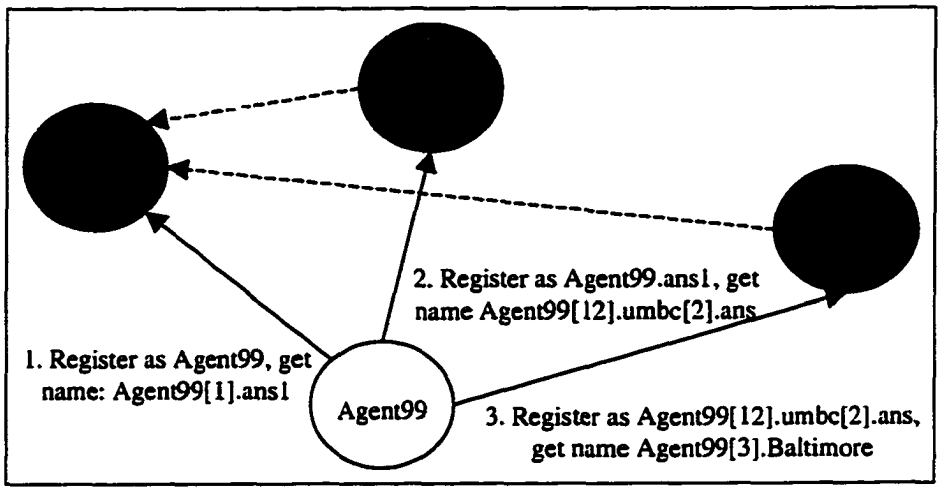


Figure 5.1: KNS Alias Network. The registrars of Agent99 coordinate to maintain the agent's distributed identity.

5.4 Protocols

KNS specifies protocols for agent addressing and naming, authentication, aliasing and Domain registration. These are sketched below:

1. Group Membership

   (a) Register: Register with a new Domain (multiple registrations are permitted). Registration implies a commitment to membership in a Domain. A registration must contain one address that is reachable by the registrar. The name given must be a FQAN. Registration causes the intended registrar to invoke the protocol for joining an AliasSet, if the name given by the registrant indicates a prior domain association.

(b) Join: Identify the Alias Set for a registrant, and join. The agent accepts the responsibility of forwarding PAS queries, and becomes eligible to become a BAS for the registering agent, though the latter is not required.

(c) Unregister: Terminate association with a Domain. The registration entry is not deleted; it is moved to a dormant status, and the addresses are cleared.

The potential for unregistration creates instability in the naming hierarchy. For this reason, one of two protocols should be followed in the event that an agent must leave a Domain.

- Domain Lifting: For each domain owned by the departing agent, the owner of the parent domain takes on the subdomain and its registration responsibilities. (e.g. umbc[2] takes over naming responsibilities for the phil[7].umbc[2] domain as phil[7].umbc[2] unregisters. This involves a transfer of the registry, and a conversion of the departing agents registration entry from real to virtual. The unregistration is automatic, and no disruption is perceived externally.
- Recursive Domain Dissolution: The agent wishing to unregister first excuses or discharges all agents registered in the Domain that is to be eliminated, using existing protocols. Each in turn does the same, so the Domain and all of its subdomains are eliminated.

Clearly, lifting is preferable to dissolution, since no naming information is lost. However, dissolution does at least prevent the use of names after they have become invalid.

(d) Excuse: Request that a registrant unregister from a named group. A positive acknowledgment constitutes an implicit unregistration. A negative or no ac-

knowledgment is followed by a discharge.

(e) Discharge: Revoke an agent's membership in a Domain. This action does not require consent or acknowledgment; it should be used only in order to elicit a response once a request to unregister has failed.

(f) Leave an Alias Set: Terminate relationship with principal for that set. If an agent is the principal or a secondary, it must first arrange successful transfer of the database and database responsibilities. Technically, this protocol is inconsistent, since a registrar accepts the responsibility for a registering agent even after the registration is no longer valid. However, this is more of an ideal in KNS than a rigorous expectation; agents will occasionally cease to provide KNS services, intentionally or unintentionally. This protocol provides a clean mechanism for withdrawal.

2. Registry Query/Update

(a) Query for the address(es) of an agent: Note that address queries are posed to Domain registries; therefore, querying an agent for its own address(es) is not permitted. KNS does not prohibit responding to queries about one's own addresses. However, some systems which integrate KNS, e.g. Jackal, do not provide agents with access to information at the message transport level directly. In any event, a well formed KNS address query could not be made to a KNS compliant agent. The response for an agent with no valid addresses is a null address packet. For a non-present agent, it is a `sorry`, or the equivalent.

(b) Update a registry entry: by adding or deleting an address or other data. It is permitted for an agent to remove all addresses from its registration entry; this

does not imply unregistration.

(c) Invalidate: notify an agent that an address it has provided is invalid. The agent receiving the invalidate should take steps to right the registry for the domain in question, either by posing queries, marking or canceling the offending entry. The precise form of the action is not currently defined.

3. Agent Information Server Query/Update

(a) Identify the alias server for an agent of a Given Name.

(b) Verify a FQAN: This is implemented as an address query, which will return an address packet if the agent's name is found in the registry.

(c) Get the aliases for an agent of a Given Name.

(d) Request that another agent replicate a (local) alias database: An agent's PAS may at its discretion request that any or all members of the agent's AliasSet replicate the AID. Agents are not required to comply. If an agent accepts the request, it becomes a BAS, and receives updates from the PAS. Its new status is broadcast to the members of the AliasSet.

(e) Abdicate: A PAS relinquishes control of the AliasSet to a member BAS. Upon acceptance, the abdicating PAS begins forwarding all incoming traffic to the new PAS, while the new PAS broadcasts the change of status to all members of the set. Any agent that serves as a BAS accepts the responsibility of potentially serving as PAS. This is a critical protocol; if it fails, the agent's identity is lost, and the former PAS becomes a BAS.

(f) Resign: A BAS notifies the AliasSet's PAS that it will no longer serve as BAS.

Only a cursory acknowledgment is required. The resigning agent is still a member of the AliasSet.

4. Additional Features

   (a) Broadcast. Messages sent to a virtual Domain are automatically copied by the registrar of the virtual Domain to all members. This is done as a 'direct' forward; that is, no modification or wrapping of the message. This process repeats itself recursively.

5.5 Additional Notes

5.5.1 Security

As mentioned briefly above, security protocols are not part of KNS, but it is clear that the scheme could not function properly without some measure of authentication in real world situations. For example, no agent should be able to unregister another agent from some domain, or register using another agent's name. We assume the use of an underlying authentication scheme that assures that the identity of the sender of a message is truthfully given. Access to KNS protocols and data is identity based, so this satisfies our requirements for operation in an open system.

5.5.2 Performance

The AliasSet scheme was carefully designed to provide comprehensive information access with minimal overhead. The cost in messages of a single name/address resolution query is constant (2), and for a composite name is linear in the number of components in the

name. The latter can be reduced significantly with address caching. Best of all, the cost of a query to the PAS of any sort, such as joining or alias resolution, is constant: 2 or 4 messages, depending on which member of the AliasSet the agent first contacts. The only significant costs to the system are replication and status change of a BAS or PAS. PASs replicate entries to all BASs in the AliasSet. If one BAS per set is used, this adds two messages to each PAS update. However, for higher fault-tolerance, it is possible to make every member of an AliasSet a BAS. Then, PAS update operations become linear in the number of domains to which the agent belongs. Of less significance, if any member of an alias set changes status (to normal, BAS or PAS), a broadcast to all members of the set is necessary, also linear in the size of the set. However, the latter operation is relatively infrequent.

Chapter 6

Jackal

Jackal has served the agent communication needs of UMBC's CIIMPLEX project (Section 7.1) for three years. This chapter describes Jackal's design and operation in some detail.

6.1 Overview

In its current form, Jackal is a Java package that allows applications written in Java to communicate via the KQML [FLM97] agent communication language. [1] It is designed to be used as a 'tool' by other applications, in that it does not require that applications be modified or extend some standard shell. Additionally, Jackal is designed so that multiple instances of it, and therefore multiple agents, may be run within the same Java Virtual Machine. Jackal:

- Facilitates the transmission and receipt of KQML messages.
- Implements a conversation-based approach to dialogue management.

[1] Jackal has been traditionally used with KQML, but was designed to be relatively language independent.

- Presents a flexible blackboard interface to the agent.
- Provides portability through Java.
- Supports the use of multiple, user defined, plug-and-play transport protocols.
- Requires little modification to existing code.
- Implements a complete scheme for agent naming and addressing (KNS).
- Supports agent extensibility through attachment of modules with message-based interfaces.

Adding communication abilities to any Java program requires no modification of existing code. This is because Jackal's functionality is accessed through a class instance, which can be shared among agent components like a portable two-way radio. This is in contrast to systems that require that a program subclass an agent shell, or otherwise restructure itself. With this Jackal instance, the agent gains more than just the ability to send and receive messages, however. Jackal's design is based in large part on, and implements, KNS (Chapter 5), an evolving standard for resolving agent names in a hierarchically structured, dynamic environment. This means that the agent application need only deal with symbolic agent names, and may leave issues such as physical address resolution and alias identification to the Jackal infrastructure.

Two components which work together to provide the greatest benefit to the agent are the conversation management routines and the Distributor, a blackboard for message distribution. The conversation system supports the use of easily interchangeable protocols for interaction, which guide the behavior of the system. The Distributor presents a flexible, active interface for internal message retrieval by agent components. While the Distributor optimizes access to the message flow, it is the conversation system that gives it its

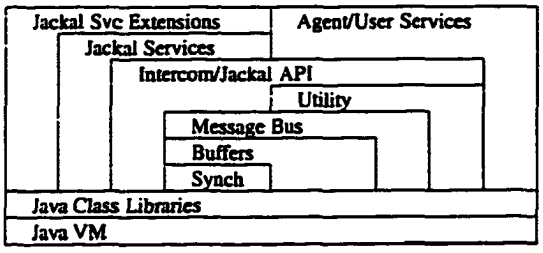


Figure 6.1: Jackal Architecture

real value; the next section will discuss in depth the rational behind the conversation-based approach.

6.2 Jackal's Design

Jackal was designed to provide comprehensive functionality, while presenting a simple interface to the user. Thus, although Jackal consists of roughly seventy distinct classes, all user interactions are channeled through one class, hiding most details of the implementation. Although there are significant benefits in some cases to sharing a Jackal instance among several agents, the typical usage is as an accessory to an individual agent. Thus, the Jackal architecture does not describe a MAS based around a shared tuple space, as it is often perceived, but a private system of which each agent in a system owns an instance.

6.2.1 Architecture

As illustrated in Figure 6.2, Jackal has a layered architecture that facilitates dynamic reconfiguration. Its native execution environment is standard, off-the-shelf Java. Central to Jackal's operation is a set of enhanced synchronization primitives and buffers, which are used to tie together its loosely coupled components. The *Message Bus* is the essence of

Jackal. Consisting principally of the conversation interpreters and a message redistribution system, it is the common path for all message traffic in a Jackal-based agent. This Bus, wrapped with some additional utilities by the Jackal API, is referred to as the Jackal *Core*. Both Jackal and agent services interact with the Core and each other through the API. Some examples of Jackal services are the Agent Naming Services, and Message Transport Services. The Jackal *Package* as it is typically distributed consists of the Core and a set of standard services.

One major strength of Jackal is its extensibility. Modules, such as name registration services, have two interfaces: a standard API, and a message-based interface. Methods can therefore be invoked directly, or via commands sent through Jackal's communication infrastructure. In the latter case, commands are received by the module through the common blackboard interface. This flexible interface makes it possible to dynamically reconfigure agents.

Figure 6.2 presents the principal Jackal components, and the basic message path through the system. We will first discuss each of the components, and then, to illustrate their interaction, trace the path of a message through the system (that is, as it is received by Jackal, passed on to and replied to by an agent thread, and the reply sent back to the original sender).

6.2.2 Intercom

The Intercom class is the bridge between the agent application and Jackal. It is the only visible element of the Core. Intercom controls startup and shutdown of Jackal, provides the application with access to internal methods, houses some common data structures, and plays a supervisory role to the communications infrastructure.

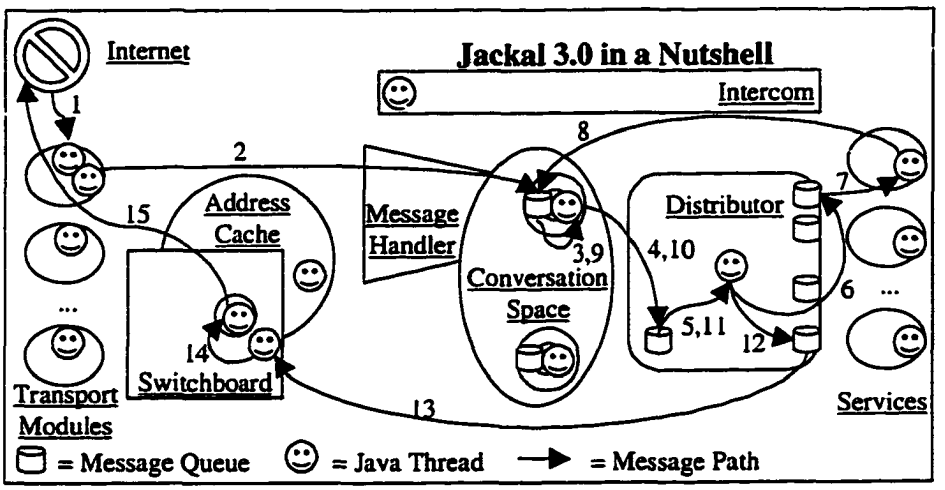


Figure 6.2: Jackal Architecture and Message Flow

6.2.3 Transport Interface

Jackal runs a Transport Module for each protocol it uses for communication. Jackal 3.0 comes with a module for TCP/IP, and users can create additional modules for other protocols. A Transport Module is responsible for receiving messages at some known address, and transmitting messages out via a given protocol.

6.2.4 Message Handler

Messages received by the Switchboard must be directed to the appropriate place in the Conversation Space; this is the role of the Message Handler. Messages are associated with current (logical) threads based on their ID (the value of the 'reply-with' field). This directs their assignment to ongoing conversations when possible. If no such assignment can be made, a new conversation appropriate to the message is started.

6.2.5 Conversations

Based largely on the work of Labrou and Finin [Lab96b, LF97c] regarding a semantics for KQML, we have created protocols, which describe the correct interactions for various performatives and subsequent messages. The protocol for `ask-one`, for example, specifies among other things that the reply must be a `tell`, `untell`, `deny`, `sorry` or `error`. These protocols are 'run' as independent threads for all current conversations. This allows for easy context management, while providing constraints on language use and a framework for low level conversation management. This is in contrast with earlier approaches (e.g., Tcl/Tk Adapter for KQML (TKQML) [CSL+97]) which require the agent to maintain context on their own.

The Conversation Space is a virtual entity, consisting of the collection of conversations started by the Message Handler. These conversations run individual protocol interpreters.

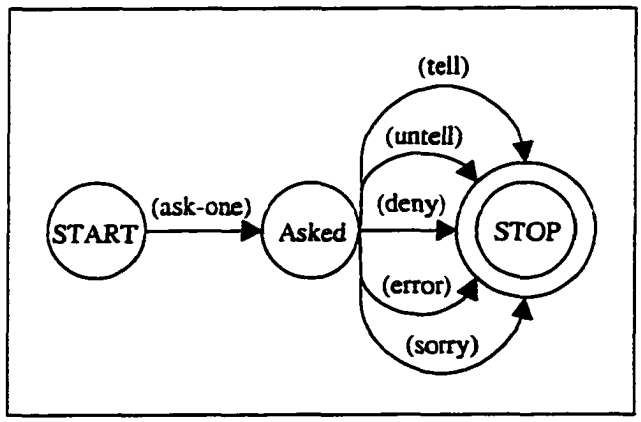


Figure 6.3: DFA for KQML **Ask-one** conversation

As of Jackal 3.0.4, conversation templates (or specifications) are completely independent from the Java library. Rather, they are specified by URLs, and loaded at runtime from some remote source. Figure 6.4 shows the conversation template for a standard KQML conversation available from the Jackal host. This template corresponds to the finite state

```
// Conversation Template
// Convention: Initial and accepting states all caps,
//             other states initial caps,
//             arc-labels lower case.
(conversation
  (name kqml-ask-one)
  (author "R. Scott Cost")
  (date "3/4/98")
  (start-state START)
  (accepting-states TOLD)
  (transitions
    (arc (label ask-one) (from START) (to Asked)
         (match "(ask-one)"))
    (arc (label tell)    (from Asked) (to TOLD)
         (match "(tell)"))
    (arc (label deny)    (from Asked) (to TOLD)
         (match "(deny)"))
    (arc (label untell)  (from Asked) (to TOLD)
         (match "(untell)"))
    (arc (label sorry)   (from Asked) (to TOLD)
         (match "(sorry)"))
    (arc (label error)   (from Asked) (to TOLD)
         (match "(error)"))))
```

Figure 6.4: Conversation Template for KQML **Ask-one**

machine depicted in Figure 6.3.

The conversation management component offers a number of significant benefits to the agent:

- Running conversations in individual threads provides maximum flexibility.
- Conversations, in conjunction with the Distributor, route messages automatically to the threads which need them.
- Each conversation maintains a local store, which can be accessed by the agent via a

message ID, and which serves as the conversation's context.

- Since conversations are declaratively specified, they can be loaded (remotely or locally) on demand. Our current agents download at initialization only the conversations they will need.

- The conversation mechanisms and the specification are almost completely independent of the content or message language used, and so could be easily be tuned work in a 'multi-lingual' environment.

- Actions can be associated with conversation structures, enhancing their utility. While our system supports this, the preliminary specification language does not. Extending it to include actions will require first the development of an ontology enumerating actions implemented by conversation interpreters.

6.2.6 Distributor

The Distributor is a Linda-like [CG89] blackboard, which serves to match messages with requests for messages. This is the sole interface between the agent and the message traffic. Its concise API allows for comprehensive specification of message requests. Requesters are returned message queues, and receive all return traffic through these queues. Requests for messages are based on some combination of message, conversation or thread ID, and syntactic form. They also permit actions, such as removing an acquired message from the blackboard or marking it as read only. A priority setting determines the order or specificity of matching. Finally, requests can be set to persist indefinitely, or terminate after a certain number of matches.

It is through this interface that Jackal supports agent extensibility. Modules can be

Parameter	Description
message: Message	If message is given, it will be sent first, the message_id collected and used to match a specified reply or thread.
template: String	(in)complete message in string form. This may be used in conjunction with or in lieu of an actual message.
msg_ID: String	Id of some message - Use this in lieu of the message field.
priority: int	Requests are examined in order of priority; this controls the specificity of the match.
delete: Boolean	Delete a message which matches this (after processing); otherwise, it will remain for others to match.
write: Boolean	Capture write privileges on any matched message; no other requests matching it subsequently may respond to it, but they may read it.
lifetime: int	Number of matches before request is removed. Zero will cause the request to remain indefinitely.
in: Boolean	Match incoming messages.
out: Boolean	Match outgoing messages.

Table 6.1: Distributor Parameters: These parameters describe the type of message that should be returned to the requester, if available, and also define any actions that should be taken.

dynamically loaded as services, and accessed through the communication infrastructure as well as their standard API.

6.2.7 Services

A service here is any thread; this could be a Jackal service, or threads within the agent itself. The only thing that distinguishes among threads is the request priority they use. System, or Jackal, threads choose from a set of higher priorities than agent threads, but each chooses a level within its own pool. Jackal reserves the highest and lowest priorities for services directing messages out of the agent and for those cleaning the blackboard, respectively.

6.2.8 Message Routing

The Switchboard acts as an interface between the Transport Modules and the rest of Jackal. It must facilitate the intake of new messages, which it gathers from the Transport Modules, and carry out send requests from the application. The latter is a fairly complicated procedure, since it has multiple protocols at its disposal. The Switchboard must formulate a plan for the delivery of a message, with the aid of the Address Cache, and pursue it for an unspecified period of time, without creating a bottleneck to message traffic. In addition, it is equipped to handle the delivery of messages with multiple recipients or 'cc' fields.

Note that the Switchboard is essentially a service in the context of the Jackal architecture. It receives message traffic from the Distributor, and differentiates itself only in that it has access to the internal API for handling incoming messages (from the Transport Interface).

6.2.9 Naming and Addressing/Address Cache

In any MAS, the problem of *agent naming* arises: how do agents refer to each other in a simple, flexible, and extensible way? If the system in question employs a standard communication language such as KQML, another requirement is that agents must be able to refer to KQML-speaking agents in the outside world. Concurrently with Jackal, we developed KNS (See Section 5), in order to support collaborating, mobile KQML-speaking agents using a variety of transport protocols. Jackal implements a portion of the KNS specification Jackal supports KNS transparently through an intelligent address cache.

The Address Cache holds agent addresses in order to defray lookup costs. It is a multi-layered cache supporting various levels of locking, allowing it to provide high availability. Unsuccessful address queries trigger underlying KNS lookup mechanisms, while blocking

access to only one individual listing.

6.2.10 Message Path

Having described the various components of Jackal, we will trace the path of a received message and the corresponding reply, using the numbered arcs in Figure 6.2 for reference.

The message is first received by a connection thread within a Transport Module [1], perhaps TCP/IP, and is processed and transferred directly to the input queue of either a waiting or new conversation [2]. A unique thread manages each conversation. Methods for the initial processing of the message reside in the Message Handler, but are called by the responsible transport thread. The target conversation, awakened, takes the message from its input queue [3] and tries to advance its state machine accordingly. If accepted, the message is entered into the Distributor [4], an internal blackboard for message distribution. The Distributor examines the message [5] in turn, and tries to match it with any pending requests (placed by Jackal or agent code), in order of a specified priority. Ideally, a match is found, and the message is placed in the queue belonging to the requester [6]. The message may in fact be passed to several requesting threads. This is the point at which the agent gains access to the message flow; through services attending to the blackboard.

Once the requesting service thread picks the message out of its queue [7], it presumably performs some action, and may send a reply or new message; we assume it does. The service has two options at this point. If it does not expect a reply to the message it is sending, the message may be sent via Intercom's send_message method [8]. Otherwise, it should be sent indirectly through the Distributor. This is equivalent to sending alone and then requesting the reply, but allows the request to be posted *first*, eliminating possible nondeterminism. Either way, the message is eventually processed through send_message,

Interfaces	
Language	A J3 Language has certain conversion methods and is serializable
MHInterface	For use by Transport Modules; access to input stream
TransportProtocol	Defines the interface for Transport Modules
Classes	
FIFO	Message buffer.
FQAN	KNS Fully Qualified Agent Name
Intercom	Jackal interface
Jif	KIF-like content language
Message	Jackal message class
Version	Package build information
Exceptions	
FQANX	FQAN exception
JifX	Jif exception
MessageX	Message exception

Table 6.2: Jackal API

which directs it into the conversation space. The message then traces the same path as the previous incoming message [9,10] to the distributor. Note that every message, incoming or outgoing, passes through the conversation space and the Distributor. The message is captured by the Switchboard's outbound message request [11], which has a special, reserved priority. The Switchboard removes new messages from its queue and assigns them each individual send threads [12]; this results in some overhead, but allows sends to proceed concurrently, avoiding bottlenecks due to wide variation in delivery times. The send thread uses the send method of the appropriate transport module to transmit the message.

6.3 API and Operation

Jackal's functionality is accessed primarily through the Intercom class, but there are a few additional classes present in the API. These are summarized in Table 6.3.

The interfaces provided allow limited interaction with the internal workings of Jackal. All object placed in the content field of a Message should be a Language; this assures that they are both automatically and manually serializable for transport. TransportProtocol provides developers with an interface for developing custom transport modules. These modules interact with Jackal via the MHInterface (Message Handler Interface).

FIFO buffers are the glue which bind the various internal components of Jackal. Built from custom synchronization methods, they provide true FIFO buffering, along with various mechanisms of bi-directional control for two connected processes. FQAN, Jif and Message provide the building blocks for communication. Although any content language may be employed by the Jackal user, Jackal itself uses Jif for low level communication, and so it is bundled with the package. Jif is an s-expression-based language with variables and mechanisms for resolution.

The central element in the API is Intercom, the primary access or control class of Jackal. Intercom presents a number of methods for message transmission, message receipt, and control.

6.4 Putting it all Together

Figure 6.6 showcases the Jackal framework, a fast, flexible communications infrastructure. Jackal's conversation space will eventually accommodate a collection of potentially heterogeneous Conversation Policy (CP) interpreters. Basic services (such as name serving) will be loadable modules. The agents behavior will be loaded in the form of conversations onto the behavior stack; first language and basic services, then higher level behavior.

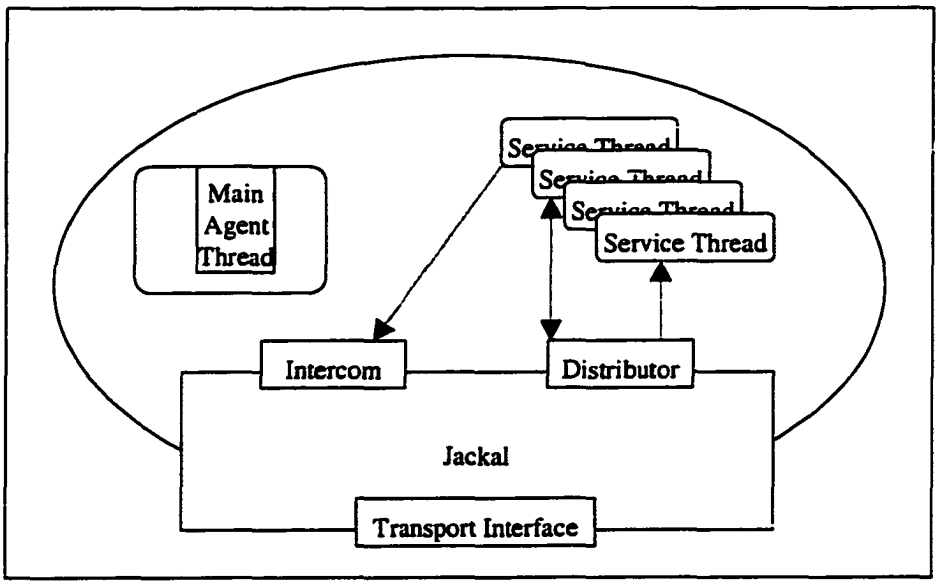


Figure 6.5: Jackal Agent Architecture

6.4.1 Jackal Abroad

Jackal was designed to work with multiple languages simultaneously. Messages are handled internally as abstract objects. Insofar as a message can be either transformed into or wrapped by a common message object (at the transport interface), the message path can process a heterogeneous stream of ACL messages. Only the CPs applicable to a particular ACL would be sensitive to differences in type. The conversation space runs individual interpreters for each conversation, so protocols for different languages can coexist. Depending on how messages of different language type are identified, they can be routed to the appropriate (or language independent) CPs. Extending the interpreter mechanism to use an interface would allow the concurrent use of multiple types of interpreters as well.

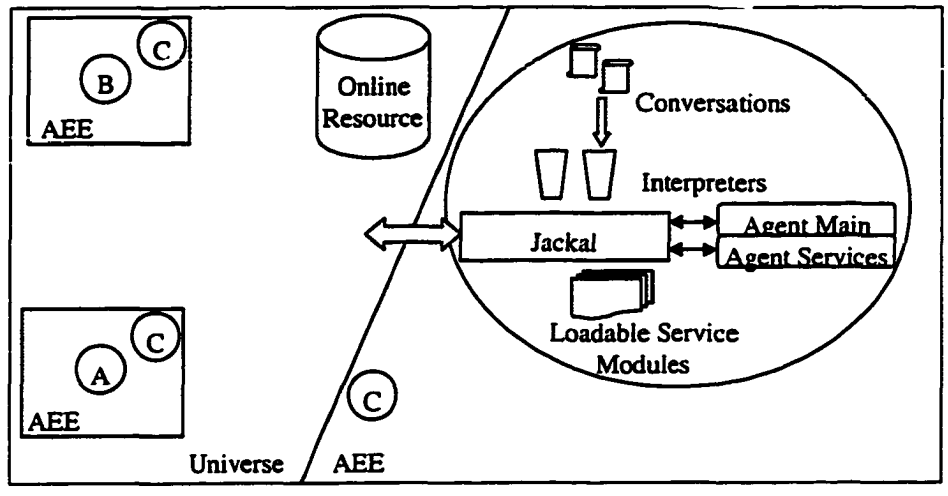


Figure 6.6: Java Application for Communicating with the KQML Agent Language: Agents are supported by the (optional) presence of name servers (A), brokers(B) and control agents (C).

6.4.2 Jackal Agent Architecture

Jackal as it was originally designed was intended to be a communication tool, like a hand-set, for a single agent. A blackboard-style interface was included to allow maximum flexibility, with a clean boundary between Jackal and the agent's code.

Some thread within the agent must create an instance of Jackal (Intercom), and should control its operation. By passing around a reference to Intercom, other components may use the messaging facilities. Other interacting agent components, then, are best modeled as sub-threads, using the Distributor as an interface. This arrangement is depicted in Figure 6.5. These threads typically have responsibilities such as waiting to process new query requests, waiting to handle errors, and so on.

6.4.3 Multi-Agent Jackal-based Systems

Jackal agents can function autonomously; they do not require a special environments, or services provided by some external agent. However, certain conventions exist which facilitate their collaboration as groups of agents. These dictate the provision of agent location by name and by service required, and the extension of the environment.

Most agent systems provide some means of resolving an agent name to a physical or logical network address. Often, this means that a designated agent will play the role of an ANS. In Jackal, name service capabilities are provided in a standard Jackal module, in order to facilitate arbitrary group formation among agents. While it is always necessary to designate one or more name servers in a multi-agent environment, this means that any Jackal agent can pick up that responsibility, in addition to its other tasks.

It is important that agents in a MAS are able to locate the resources that they need. In a more static environment, this can be handled by services like CORBA, which locate required services by name. More general brokers can identify potential service providers based on a description of the service required. The latter are often employed in MASs, and provide a much more flexible solution to the problem.

Figure 6.6 shows a typical arrangement of agents in a Jackal-based MAS.

6.4.4 Multi-Agent Sub-Jackal Systems

Thread to thread communication is not limited to threads within different agents. A planned extension to Jackal is internal registration, which will allow individual threads within an agent to register with that agent, and use their corresponding names to communicate as first class entities. [2] Figure 6.7 depicts a log thread communicating informing a user-interface

[2]In fact, this is possible in the current implementation. However, there are currently no mechanisms to arbitrate control over one Jackal instance. The proposed mechanism will allow an agent using Jackal to

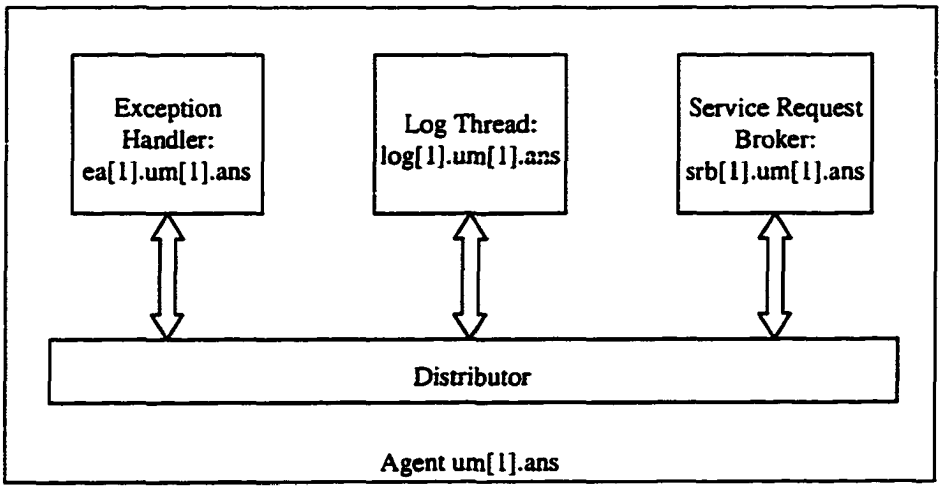


Figure 6.7: Intra-agent communication in Jackal. Internal agent threads can act as simple components or individual agents, using the distributor as a plug-and-play bus.

component that the log file is full. This feature has two powerful potential uses:

1. For lightweight, populous, centrally located systems (e.g. a set of text filtering agents swarming over a pile of local documents), this allows an arbitrary number of agents to share the same Jackal instance.[3]

2. Agent may be created in a completely plug-and-play fashion. Mirroring the MAS architecture, one internal thread could be assigned the role of broker, and could direct new modules, as added, to the resources that they need.

These two paradigms can be used together. Since Jackal can support multiple concurrent languages, internal communication among some threads might use KQML, while others could use an abbreviated systems language.

internally distribute a restricted (Intercom) reference, which allows access to the messaging methods, but not the control methods.

[3]The conditions are not exactly the same as for agents with private Jackal instances. Control over Jackal in the sub agent case is limited, because many agents are sharing it. Also, Jackal will at some point become a communication bottleneck.

The notion of a core message bus in Jackal is reminiscent of message-based distributed systems, in which the local computation units coordinate/communicate through some standard message protocol. Many of the most current ideas in message passing systems have been incorporated into the Message Passing Interface (MPI) Standard [DOSW, For95] Jackal can use this approach to form scalable systems of heterogeneous components. An example of a more sophisticated approach is the Software Bus, utilized by the Problem Solving Environment (PSE) of Weerawarana et al. [WJHC95] (also [DWJ95]). This Software Bus is more active, incorporating resource brokering (in the form of a bus master), distributed object management and object interface/language translation. Such a scheme could be implemented with Jackal by adding translation filters, and by placing a resource broker right at the Distributer, as in the second paradigm (above). Local units link together via various transport protocols to form a distributed object bus. Also related are systems which base a distributed computing framework atop a shared tuple space in order to achieve distributed communication and coordination [HV98, Tol98].

6.5 Summary

Jackal is a Java package that allows applications written in Java to communicate via the KQML [FLM97] agent communication language It is a flexible, extensible 'tool', not a shell, which supports the use of conversation policies, and which offers a flexible, blackboard-based interface to the agent. The plug-and-play protocol handlers and integrated support for KNS enhance its ability to interface with communities of agents, but its real value is in the support it offers for using conversation policies to direct the communicative actions of agents.

Chapter 7

Application Domains

Jackal has been developed as part of a larger effort to develop an agent infrastructure for manufacturing information flow. It has been used to facilitate communication among diverse agents responsible for collecting, processing and distributing information on a manufacturing shop floor. The next section describes this project in some detail.

7.1 CIIMPLEX: Enterprise Integration

The production management system used by most of today's manufacturers consists of a set of separate application software, each for a different part of the Planning, Scheduling and Execution (P/E) processes [VBW92]. For example, Capacity Analysis (CA) software determines a Master Production Schedule that sets long-term production targets. Enterprise Resource Planning (ERP) software generates material and resource plans. Scheduling software determines the sequence in which shop floor resources (people, machines, material, etc.) are used in producing different products. The Manufacturing Execution System (MES) tracks real-time status of work in progress, enforces routing integrity, and

reports labor/material claims. Most of these P/E applications are legacy systems developed over years. Although each of these software systems performs well for its designated tasks, they are not equipped to handle complex business scenarios [Ber96, JFN+96, TWG93]. Typically, such scenarios involve coordination of several P/E applications to respond to external environment changes (price fluctuations, changes of requests from customers and suppliers, etc.) and internal execution dynamics within an enterprise (resource changes, mismatches between plan and execution, etc.). Timely solutions to these scenarios are crucial to agile manufacturing, especially in the era of globalization, automation, and telecommunication [DB92]. Unfortunately, these scenarios are primarily handled by human managers, and the responses are often slow and less than optimal.

The CIIMPLEX, consisting of several private companies and universities, was formed in 1995 with matching funds from the National Institute of Standards and Technology of the U. S. federal government. The primary goal of the consortium is to develop technologies for intelligent enterprise-wide integration of planning and execution for manufacturing [CTW+96]. Our vision of a successful integrated P/E system for manufacturing would has the following features.

1. Interoperability. Heterogeneous P/E applications from different vendors are able to operate together as integrated parts of the system.

2. Integration. Software tools and infrastructures to support integration tasks not covered by existing P/E applications are provided. In particular, the integrated solution should support runtime dynamic coordination in dealing with unexpected events.

3. Distributed. Resources such as software and data are allowed to be physically or logically distributed.

4. Openness. The user shall be able to select and change different applications and tools easily and with little additional integration cost.

One approach to an integrated P/E system might be to rewrite all application software into a monolithic integrated planning-execution system capable of handling all foreseeable scenarios. This approach is judged to be infeasible because of high development and maintenance cost, and the closed-ness and inflexibility of such monolithic systems [Ham96]. Conventional object-oriented distributed systems also seem inadequate because they work at the level of objects, and thus lack the support for abstraction at higher levels [JW98].

Instead, CIIMPLEX has adopted as one of its key technologies the approach of intelligent software agents, and is developing a MAS for enterprise integration. In sharp contrast to traditional software programs, software agents are programs that help people solve problems by collaborating with other software agents and other resources in the network [BDBW98, Com96, JW98, Nwa96, PBC97]. For instance, individual agents can be designed to perform data collection and analysis of plans and schedules and to keep constant vigil against mismatches among these plans and schedules at different levels of abstraction and time horizons. Other agents can be designed to resolve the conflicts either by themselves or in coordination with human managers and analysts. Personal assistant agents can be designed to assist human managers/analysts. Still other agents can be created to provide legacy systems with better communication and coordination capabilities so that they can more effectively cooperate with each other and with other agents. Moreover, MAS, as a society of autonomous agents, is inherently open and distributed, and inter-agent communication capability provides the essential means for agent collaboration.

The environment of the CIIMPLEX Consortium is different from academic-oriented research laboratories. Most of the companies in the consortium are P/E application sys-

tem vendors and users. This situation gives us the opportunity to work with real-world P/E systems (rather than imagined toy problems) and appreciate the complexity of realistic business scenarios. On the other hand, the agent-based approach, as a relatively immature technology which has not yet reached industrial strength, understandably receives only guarded enthusiasm by some members in the consortium. They are more concerned with integration of the P/E systems, using more mature technologies, to better handle normal or expected business scenarios. Our immediate priority is thus not to design and develop a complete agent system that integrates all aspects of manufacturing planing and execution, but to develop one that is limited in scope but reliable and scalable, and clearly adds commercial value to the end user. In addition, the initial prototype agent systems must have minimum interference with the normal work of existing P/E applications.

Based on these considerations, we have decided to concentrate on those P/E scenarios which represent exceptions to the normal or expected business processes and whose resolution involves several P/E applications. For example, consider the scenario involving a delay of the shipment date on a purchased part from a supplier. This event may cause one of the following possible actions: (a) the manufacturing plan is still feasible, no action is required; (b) order substitute parts; (c) reschedule; or, (d) reallocate available material. To detect this exception and determine which of these actions to take, different applications (e.g., MES, ERP, CA, and Scheduler) and possibly human decision-makers must be involved. Examples of other similar scenarios include a favored customer's request to move ahead the delivery date for one of its orders, a machine breakdown being reported by MES, or a crucial operation having its processing rate decreased from the normal rate, to mention just a few.

Figure 7.1 illustrates at a conceptual level how an exception (e.g., a shipment of a purchased part is delayed) should be handled by an integrated system. The decision module

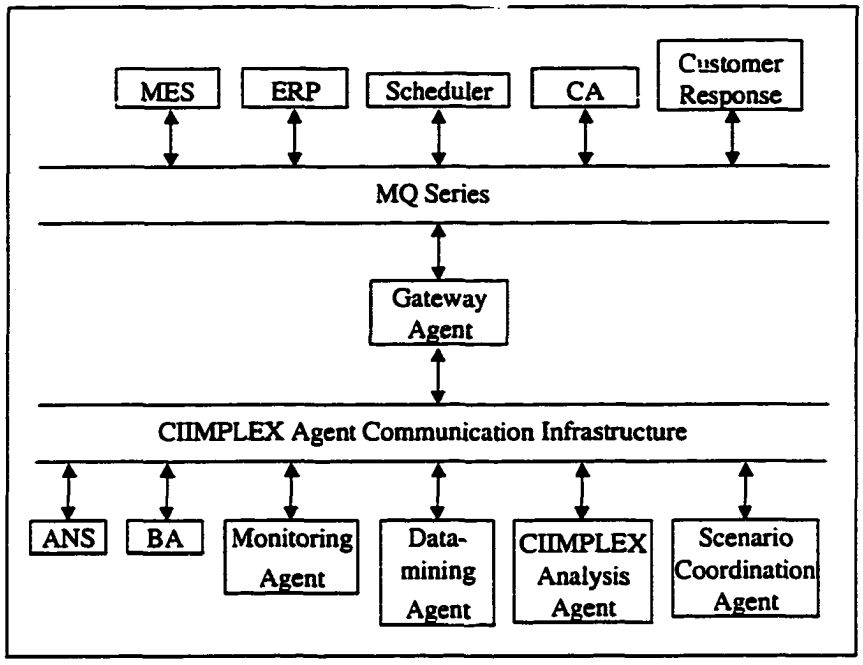


Figure 7.1: A manufacturing integration example: handling exceptions

decides, with the instruction from a human or an analysis module, what constitutes an exception. The monitoring module determines what data is to be monitored to detect such exceptions and conducts actual monitoring of the data stream. When notified by the monitoring module of the occurrence of an exception, the decision module makes appropriate decisions in consultation with other P/E applications and the analysis module. The decision (e.g., a request to reschedule) will then be carried out by the designated P/E application(s). Note that what constitutes an exception and how to monitor it is a dynamic decision which cannot be specified prior to the plan execution. For example, a factory may not normally consider a delay of shipment of an ordered part exceptional unless the delay is greater than five days. However, if a part is crucial for an order of a preferred customer or the inventory of a part is below a threshold, then a delay of greater than three days may become an exception. To make the situation more complicated, an action taken to address one exception

may trigger another exception to occur (e.g., a reschedule to handle the delay of shipment of one part may delay the delivery date of an important order for which the respective sales representative needs to be notified).

To provide an integrated solution to the above outlined scenarios, simple as they are, is by no means a trivial undertaking. First of all, a set of agents of specialized expertise need to be developed to provide functions, such as those performed by the analysis module, decision modules, and monitoring modules in Figure 7.1, which are not covered by any of the existing P/E applications. As integration tasks, these functions fall in the "white space" between these P/E applications. Secondly, a reliable and flexible inter-agent communication infrastructure needs to be developed to allow agents to effectively share information, knowledge, and services. Thirdly, some means to support interaction with P/E applications, which will not be agentified at this stage, need to be provided. And finally, a mechanism for the runtime collaboration of all these pieces also needs to be developed. In this chapter, we describe our experience of developing an agent-based system for the CIIMPLEX project.

7.1.1 CIIMPLEX Agent Architecture

In this section, we describe the MAS architecture that supports inter-agent cooperation in the CIIMPLEX project, emphasizing on the agent communication infrastructure. Figure 7.2 below gives the architecture of CIIMPLEX enterprise integration with MAS as an integrated part.

At the current stage of the project, the entire P/E integration architecture is composed of two parts: the P/E application world and the agent world, and supported by two separate communication infrastructures. Although these legacy P/E applications have not being agentified, they have been wrapped with APIs, which provide them with limited commu-

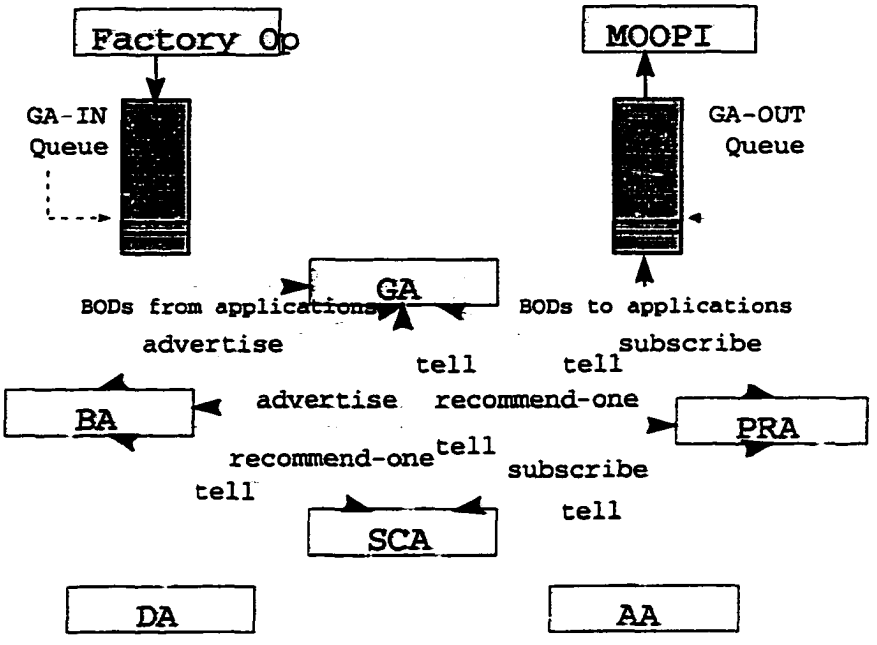


Figure 7.2: CIIMPLEX Integration Architecture

nication capability [CLM+98]. Different transport mechanisms (e.g., MQ Series of IBM and VisualFlow of Envisionit) are under experimentation as communication infrastructures for the wrapped P/E applications. These mechanisms are persistent, but only support static, predetermined communication patterns. In the agent world, besides the service agents ANS and Broker Agent (BA), several other types of agents are useful for enterprise integration. For example, data-mining/parameter-estimation agents are needed to collect, aggregate, interpolate and extrapolate the raw transaction data of the low level (shop floor) activities, and then to make this aggregated information available for higher level analyses by other agents. Event monitoring agents monitor, detect, and notify about abnormal events that need to be attended. The CIIMPLEX Analysis Agents (CAA) evaluate disturbances to the current planned schedule and recommend appropriate actions to address each disturbance. The Scenario Coordination Agents (SCA) assist human decision making for specific business scenarios by providing the relevant context, including filtered information, actions,

as well as workflow charts. All these agents uses the KQML as the agent communication language, and use a subset of KIF that supports Horn clause deductive inference as the content language. TCP/IP is chosen as the low-level transport mechanism for agent-to-agent communication. The shared ontology is an agreement document established by the P/E application vendors and users and other partners in the consortium. The agreement adopts the format of the Business Object Document (BOD) defined by the Open Application Group (OAG). BOD is also used as the message format for communication among P/E applications such as MES and ERP, and between agents and applications. A special service agent, called the Gateway Agent (GA), is created to provide interface between the agent world and the application world. GA's functions, among other things, include making connections between the transport mechanisms (e.g., between TCP/IP and MQ Series) and converting messages between the two different formats (KQML/KIF and BOD). The agent system architecture outlined above is supported by Jackal [CFL+98]. As indicated by the name, Jackal is written in JAVA to support Agent Communication using the KQML Agent communication Language. The decision to select JAVA as the implementation language was based mainly on its inter-platform portability, its networking facilities, and its support for multi-thread programming.

Chapter 8

Colored Petri Nets

Having examined a number of approaches, we find the CPN to be an excellent formalism for modeling conversations, for reasons discussed below. This chapter presents a detailed introduction to CPNs, both the formal model and the notation, and shows how they can be applied to the familiar task of specifying and implementing conversations.

8.1 Rational

Of the models considered, CPNs are the most attractive candidate for the following reasons:

1. CPNs are among the simplest; though less accessible than DFAs.
2. CPNs, like DFAs, have an intuitive graphical representation, which significantly aids model development.
3. CPNs support the modeling of concurrency systems, something we feel will be essential in supporting more sophisticated agent interaction.

4. Petri Nets in general are a very well understood and researched formalism with proven value in many domains.

5. There exist a wealth of tools and techniques for designing, simulating and formally analyzing CPNs. Many of them are freely available to the community. Of note is the Design/CPN modeling and simulation tool, made available by Aarhus University, Denmark, which we have used to develop and verify the models presented in the work.

While many of these qualities apply to a number of the potential solutions examined, CPNs show the most promise overall, offering both relative simplicity and significant representational power.

It is worth noting that these advantages are shared by a family of CPN-related formalisms. Each of these formalisms, enumerated below, has a unique appeal in modeling various systems [Tro99].

1. Traditional High Level Petri Nets - These include Predicate/Transition Nets [GL81] and CPNs [Jen81].

   (a) CPNs - Defined using types, variables and expressions, from functional programming languages and lambda calculus.

   (b) Predicate Transition Nets - Defined using the notation and concepts of many-sorted algebras.

2. High-Level Petri Nets with Abstract Data Types - High-level algebraic Petri Nets, where tokens and firing rules are given over an algebraic specification.

3. Environment Relationship Nets - High-level Petri Nets where the tokens represent environments.

4. Product Nets - A high-level Petri Net formalism defined for the Product Net Machine tool.

5. Well-Formed Nets - Similar to CPNs Nets where the color functions are defined in a different way.

6. Regular Nets - Adopt the same notation as Well-Formed Nets, allowing each basic object class to appear only once in each Cartesian product of color domains.

This chapter discusses uses the term CPN to include Hierarchical Colored Petri Net (HCPN)s. The HCPN is an extension of the CPN formalism which provides for the hierarchical composition of multiple CPNs into larger, unified models. HCPNs have equivalent representations as nonhierarchical CPNs (see [Jen92]), but can facilitate the development of large, complex models.

8.2 Formal Definitions

We present here the formal definitions for CPNs and HCPNs.

Formally, CPNs are defined (see [Jen92]) as:

16 DEFINITION (A NONHIERARCHICAL CP-NET) *A tuple* $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ *satisfying the requirements below:*

1. Σ *is a finite set of nonempty types, called* **color sets.**
2. P *is a finite set of* **places.**
3. T *is a finite set of* **transitions.**

4. *A is a finite set of* **arcs** *such that:*
   - $P \cap T = P \cap A = T \cap A = \emptyset$
5. *N is a* **node** *function. It is defined from A into $P \times T \cup T \times P$.*
6. *C is a* **color** *function. It is defined from P into Σ.*
7. *G is a* **guard** *function. It is defined from T into expressions such that:*
   - $\forall a \in A : [Type(E(a)) = C(p(a))_{MS} \wedge Type(Var(E(a))) \subseteq \Sigma]$ *where $p(a)$ is the place of $N(a)$.*
8. *I is an* **initialization** *function. It is defined from P into closed expressions such that:*
   - $\forall p \in P : [Type(I(p)) = C(p)_{MS}]$.

A Hierarchical CPN is defined (see [Jen92]) as:

17 DEFINITION (HIERARCHICAL CP-NET) *A tuple $HCPN = (S, SN, SA, PN, PT, PA, FS, FT,$ satisfying the requirements below:*

1. *S is a finite set of* **pages** *such that:*
   - *Each page $s \in S$ is a nonhierarchical CP-net:* $(\Sigma_s, P_s, T_s, A_s, N_s, C_s, G_s, E_s, I_s)$.
   - *The sets of net elements are pairwise disjoint:* $(\forall s_1, s_2 \in S : [s_1 \neq s_2 \rightarrow (P_{s_1} \cup T_{s_1} \cup A_{s_1}) \cap (P_{s_2} \cup T_{s_2} \cup A_{s_2}) = \emptyset]$.
2. *$SN \subseteq T$ is a set of* **substitution nodes.**
3. *SA is a* **page assignment** *function. It is defined from SN into S such that:*
   - *No page is a subpage of itself:* $s_0 s_1 \ldots s_n \in S * | n \in N_+ \wedge s_0 = s_n \wedge \forall k \in 1 \ldots n : sk_k \in SA(SN_{s_{k-1}}) = \emptyset$.
4. *$PN \subseteq P$ is a set of* **port nodes.**
5. *PT is a* **port type** *function. It is define from PN into $in, out, i/o, general$.*
6. *PA is a* **port assignment** *function. It is defined from SN into binary relations such that:*
   - *Socket nodes are related to port nodes:* $\forall t \in SN : PA(t) \subseteq X(t) \times PN_{SA(t)}$.

- *Socket nodes are of the correct type:*
  $\forall t \in SN \forall (p_1, p_2) \in PA(t) : [PT(p_2) \neq general \rightarrow ST(p_1, t) = PT(p_2)]$.
- *Related nodes have identical color sets and equivalent initialization expressions:*
  $\forall t \in SN \forall (p_1, p_2) \in PA(t) : [C(p_1 = C(p_2) \wedge I(p_1) <>= I(p_2) <>]$.

7. $FS \subseteq P_s$ *is a finite set of* **fusion sets** *such that:*

   - *Members of a fusion set have identical color sets and equivalent initialization expressions:*
     $\forall fs \in FS : \forall p_1, p_2 \in fs : [C(p_1) = C(p_2) \wedge I(p_1) <>= I(p_2) <>]$.

8. FT *is a* **fusion type** *function. It is defined from fusion sets into global, page, instance such that:*

   - *Page and instance fusion sets belong to a single page:*
     $\forall fs \in FS : [FT(fs) \neq global \rightarrow \exists s \in S : fs \subseteq P_s]$.

9. $PP \in S_{MS}$ *is a multi-set of* **prime pages.**

A much more complete description of CPNs and HCPNs can be found in [Jen92, Jen94a, Jen97b]. It is important to note that the HCPN defines collections of nonhierarchical CPNs only. HCPNs cannot themselves contain HCPNs as subnets. Later, we propose a solution to this; a notational scheme, called Protolingua, for representing CPN-based conversations in a hierarchical, language neutral way.

8.2.1 Related Work

CPNs are not new, and they have been used extensively for a broad range of applications (see [Jen97b] for a survey of current uses). Since their target domain is distributed systems, and the line between that domain and MASs is vague at best, there is much work on which to build. We will review here a few of the more directly related research endeavors.

Holvoet and Verbaeten, at the Katholieke Universiteit Leuven, Belgium, have published perhaps more than any others on the subject of agents and PNs. In their 1995 paper, "Agents

and Petri Nets" [Hol95], they introduced the idea of enhancing AOP by using high-level nets to model agents. They developed this thought further [HV96] to modeling agents with a variant they called 'Generic Nets', which could be extended to other sorts, including high-level nets, but with a clear bias towards Predicate Transition Nets. In 1997, Holvoet and Kielmann brought this work into focus, with the introduction of Petri Net Semantics for Objective Linda (PNSOL) [HK97]. PNSOL was used to model agents which lived in and communicated through the Objective Linda [Kie96] tuple space. This work was developed further in [HV98, HK98]. Also worthy of note is their work in applying agents to the modeling and execution of distributed systems via Petri Nets [HV97]

Merz and Lamersdorf, at the University of Hamburg, picked up the AOP banner and extended their Common Open Services Market (COSM) infrastructure to a COSM-based 'AOP engine architecture' [ML96]. This was accomplished by replacing the FSM model used in their Service Representation (SR) with CPNs, which they extended with *split* and *join* transitions. COSM has played a central role in much of their work regarding electronic markets.

Also at Hamburg, Moldt and Wienberg developed an approach called Agent Oriented Colored Petri Net (AOCPN)s [Wie96, MW97]. This system employed an object-oriented language, syntactically similar to C++, which maps onto CPN, extended by 'test arcs' [CH92, LC93]. They show how this approach can be used to model societies of agents as described by Shoham [Sho93]. Their model extends down to the level of individual agent theorem provers, facilitating the logical specification of agent behavior. Actual verification (of the base CPN description, without extensions) is performed in Design/CPN. Along with Kumer [KMW98], the authors have also investigated the connection of Java processes with the Design/CPN tool.

Yoo, Merlat and Briot, at the Laboratoire d'Informatique de Paris 6, describe a

contract-net based system for electronic commerce that uses a modular design [YMB98]. Among the modular components are Block-like Representation for Interacting Components (BRIC) [Fer96]), which are derived from CPNs. Other components are primitive (Java) components, connecting components, and compound components, which are collections of the other three. There is little discussion of the BRICs in this paper, however, except that they play a role in validation, using CPN-AMI [Lab96a].

Fallah-Seghrouchni (LIPN - Université Paris Nord) and Mazouzi (LAMSADE - Université Paris Dauphine) have demonstrated the use of CPNs in specifying conversation policies in some detail, using FIPA ACL as a framework [FSM98, FSHM99, FSM99]. This work treats both conversation specification with respect to the message components, and (briefly) the use of HCPN for constructing hierarchical structures. This work evolved from earlier LIPN/LAMSADE collaborations involving MAS and Recursive Petri Net (RPN)s (e.g. [BFSH+98]

Also of note, Billington et al. [BFD98], Purvis and Cranefield [PC96], and Lin et al. (above) [LNSK99].

8.3 A Case Study

In this section, we elaborate on the formal definitions (above) by illustrating the application of CPNs to a KQML conversation, examining the various issues that arise. For the purposes of demonstration, we will assume a KQML message of a slightly simplified form, a six element tuple consisting of a performative name, sender, receiver, reply-with and in-reply-to tags (message identifiers), and content. Performative, sender and receiver are of enumerated types, identifiers are integers. Message content is of an enumerated type, for simplicity. Parameters are denoted p, s, r, i, j, and c respectively. It should be clear that

this can be extended to incorporate other message parameters, such as language and ontology, and other parameter types. We have included only those parameters that play a direct role in determining conversation structure, which should be maximally independent of the *form* of the message content.

KQML **Ask-one** is a fairly straightforward, query-response conversation in which on agent requests that another agent supply an 'answer' to a query posed in the content portion of a message. There are only two messages, and no issues of synchronization. Formally, the Definite Clause Grammar (DCG) representation of this conversation [Lab96b] is shown in Figure 8.1.

```
                                S →  s(CC,P,S,R,IR,Rw,IO,C),
                                     {member(P,[advertise,broker-one,broker-all,forward,
                                       broadcast,recommend-one,recommend-all,
                                       recruit-one,recruit-all,register,
                                       unregister,transport-address])}
s(CC,ask-one,S,R,IR,Rw,IO,C) →  [[ask-one,S,R,OR,Rw,IO,C]] |
                                [[ask-one,S,R,OR,Rw,IO,C]], {OI is abs(1-IO)},
                                  r(CC,ask-one,S,R,_,Rw,OI,_)
r(CC,ask-one,S,R,IR,Rw,IO,_) →  r(CC,ask,S,R,IR,Rw,IO,_)
      r(CC,ask,R,S,_,IR,IO,C) →  [[tell,S,R,IR,Rw,IO,_]] |
                                [[untell,S,R,IR,Rw,IO,_]] |
                                [[deny,S,R,IR,Rw,IO,_]] |
                                problem(CC,R,S,IR,_,IO)
   problem(CC,R,S,IR,Rw,IO) →  [[error,S,R,IR,Rw,IO,[]]] |
                                [[sorry,S,R,IR,Rw,IO,[]]]
```

Table 8.1: DCG representation of a KQML **Ask-one** conversation [Lab96b]. Functions and constants are lower-case initial, variables are upper-case initial. The variables S, R, IR, Rw, and C correspond to sender, receiver, in-reply-to, reply-with and content, respectively.

Informally, definition states that a KQML `ask-one` query may be answered with the performative `tell`, `untell`, `deny`, `error`, or `sorry`, and constrains the use of the name and identifier fields intuitively. This conversation,depicted as a DFA, is shown in

Figure 8.1. Note that an alternative DFA representation might explicitly give a distinct terminal state for each reply type; the form is in part dictated by the actual use of the model.

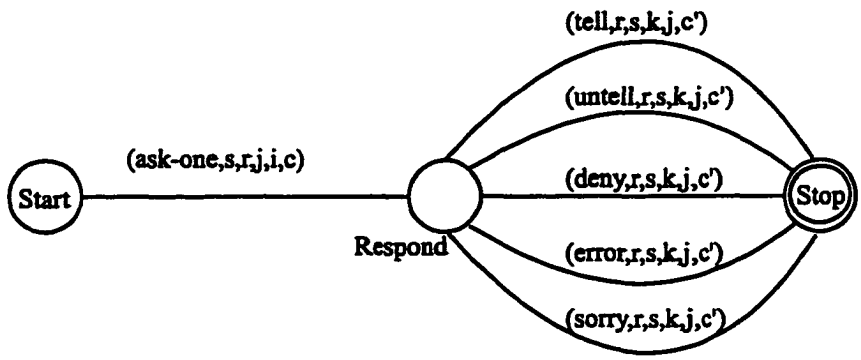


Figure 8.1: DFA representation of a KQML **Ask-one** conversation

In creating a CPN representation of any conversation, we must bear in mind that, unlike with DFAs, states of individual entities need not map directly onto nodes in the net; the state of the conversation is represented in the combined states of all elements of the net.

Consider the CPN representation of **Ask-one** in Figure 8.2. Before discussing the model itself, we must briefly explain the CPN notation. CPNs are generally given in graphic form, because their many-tuple representation (see Figure 8.3 for the many-tuple representation of this net) is considerably less intuitive. Jensen [Jen92] refers to the diagrammatic representation as a CPN diagram, and used the term CP-net to refer to the many-tuple representation. Places are denoted by ellipses and transitions by rectangles. The graphical components (places, transitions and arcs) correspond to the *structural description* mentioned above. The *declarations* are contained in a declaration node, and provide definitions for the colors, variables and functions used in the net inscriptions. The remaining text in the figure constitutes the net *inscription*. Places are inscribed with a name, color (in italics) and an initial marking (adjacent). Transitions are inscribed with a name and guard expres-

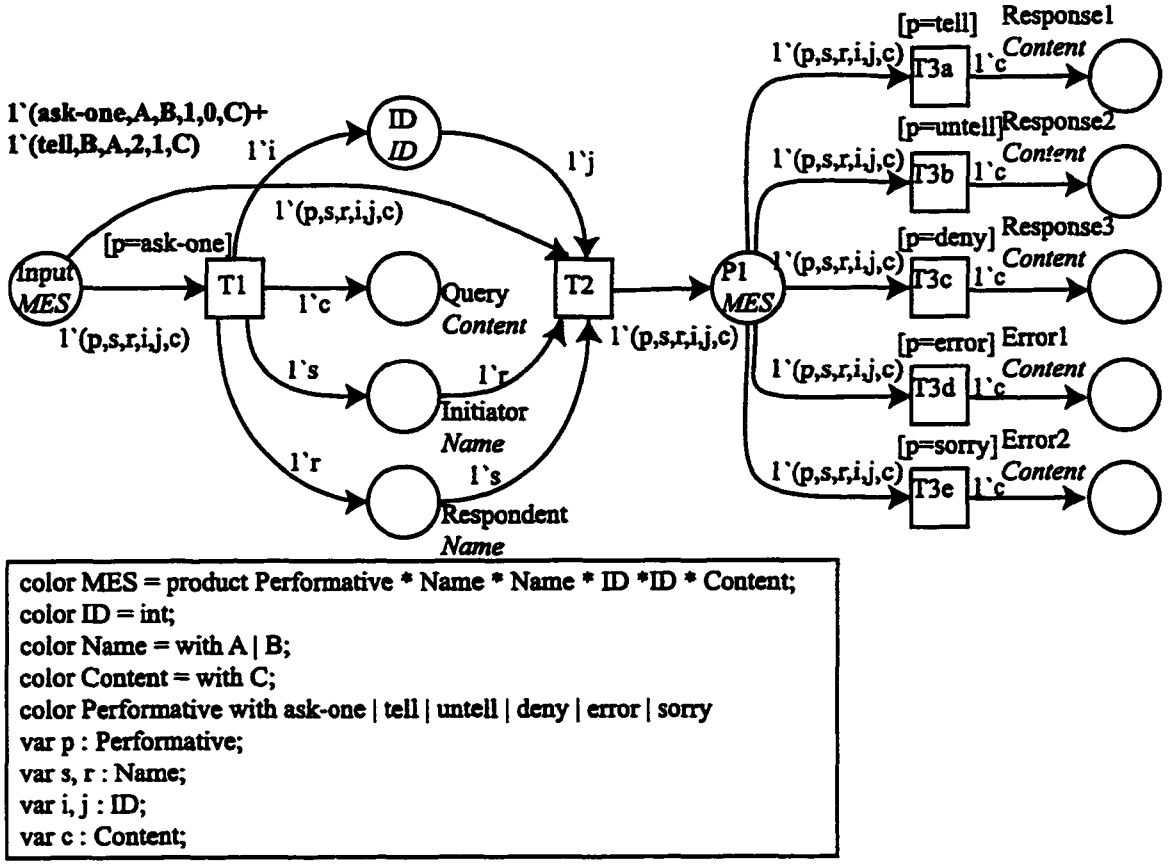


Figure 8.2: CPN representation of a KQML **Ask-one** conversation

sions (in square brackets), which serve to constrain local variables. Arcs are inscribed with expressions which denote the addition or subtraction of color-sets from their corresponding place. A transition may fire when a binding can be found for all of the variables in its guard expressions and adjacent arc expressions, at which time elements are added and/or subtracted to/from adjacent place multisets as appropriate.

Consider Figure 8.2. This CPN has eleven places and seven transitions. The place **Input** serves as a source for all messages in the conversation. Since this is a model of the message stream only, we assume that all messages are present in **Input** initially. Transition **T1** represents the acceptance of the initial message; note that the arc expression for its input

from **Input** specifies that the performative of the initial message must be `ask-one`. **T1** then distributes components of the message to **ID**, **Query**, **Initiator** and **Respondent**. **T2**, the acceptor for the second message, verifies by its connections with the aforementioned places that the second message has the correct names and identifiers associated with it. It then passes the message to a second holding place, **P1**. From **P1**, the final transitions in the CPN will extract the content of the message to a place corresponding to the resulting performative. We can say that the conversation has terminated successfully when there is some element in **Response1**, **Response2**, **Response3**, **Error1** or **Error2**. **T1** and **T2** impose the added constraints that a message may not be sent from an agent to itself; this is not part of the **Ask-one** specification, but has been added to illustrate the use of guard expressions in CPN transitions.

Note that, as with most formalisms, there are many different ways to represent the same interaction. The DFA representation associates an arc with each event, which here is the occurrence of a message. A single place is given for each corresponding agent state. In the CPN representation, transitions represent more than message events, but also arbitrary relationships among messages and other variables in the model. Also, the state of an agent or agents is not linked directly to any one 'place', but is represented in the overall state of the net. The form in Figure 8.2 was designed to minimize the number of arcs, and thus maximize visual clarity. For this reason, the processing of the second message was split into two separate steps; T2 and T3x. We could have unified the last array of transitions T3x into a single transition, and given it arcs to the final places, each conditioned on message performative. Rather, we chose to create a separate transition for each performative, facilitating the addition of response-specific actions on the part of the conversation-initiating agent. Because we chose to split on performative before the transition, we divided the processing of the second message into two stages to avoid the complexity of connecting

all of the generic constraints (e.g. id, sender, and receiver) to each of the final transitions. Jensen [Jen92] presents useful instructions on the construction of effective and visually accessible CPNs.

The use of a bi-directional arc is a notational shortcut, and should be interpreted as two opposing but identical arcs, the net effect of which is to enable a transition with a certain prescribed binding, but which removes no tokens from the corresponding place. This is how constants are effected in CPNs. Also, arc expressions indicate how many of a given element are to be added to/subtracted from a given multiset (e.g. 1'(p,s,r,i,j,c)). This is technically equivalent to having multiple identical arcs each adding/removing a single element to/from the same multiset.

The framework we have described in Figure 8.2 specifies the correct message form and sequence, but does not model the behavior of the agents generating the messages. Messages are assumed to be predefined and present in a message database. Under the DFA model, we assumed that messages were presented to the appropriate set of arcs as they were encountered; only arcs emanating from a node denoting the agent's current state are valid options. While this satisfies some of our goals in modeling, the verification and constraint of the message stream, we would also like to use the conversation representations to model and implement agent behavior as well. Consider the following modifications (Figure 8.4) to the diagram in Figure 8.2.

Note that the 'second stage' of the model is no longer fed by the initial place, but rather by a message generated in the first phase. The action generating the message is associated with the second agent in the conversation. Likewise, functions for processing the response, associated with the initiating agent, can be added to the final stage, most likely on arcs emanating from the T3x array of transitions.

The introduction of generative components (above) raises certain issues with respect

to usage, which we will only briefly mention here. Complications arise because we are modeling a conversation among several distributed entities, as if we had a clear, 'omniscient' view of the transactions. This is fine for conversation analysis. We would like to allow individual agents to use such models in the field, however, to guide them in their conversational practices. These agents have only a local view of message transactions, and the model described above must 'behave' differently depending on the local agent identity. In other words, the initiating agent of a two party conversation can only 'see' and act on messages which it sends or receives, and must infer that other agents in the conversation are executing actions appropriate to their behavior. Thus, synchronization to some degree must be achieved among the various individual instances of the common conversation model, preferably without the addition of messaging overhead.

We do not propose to solve these problems here, but it is important to bear them in mind if the models we create will be amenable to possible solutions. Otherwise, we risk admitting a significant gap between conversation models for analysis and those used in practice, complicating the job of the designer.

8.4 Hierarchical Models

Using the HCPN formalism, we can construct models from component subnets, facilitating the development of more complex models. To illustrate this, we present a two-query KQML conversation specified as an HCPN in Figure 8.5. This conversation makes use of the **Ask-one** conversation developed in Figure 8.2, and describes a simple conversation in which two consecutive `ask-one` queries take place. We will introduce the notation, and then describe the action in more detail.

Individual CPNs in an HCPN model reside on individual **pages**. The page depicted in

Figure 8.5 is labeled page Ask#1. Ask#1 is a **prime page**, meaning that is not a **subpage** for any page in this model. A model may have multiple **prime pages**. For the purposes of this example, we will assume that the net in Figure 8.2 is on a page labeled page AskOne#2. This CPN has two transitions marked **HS** (boxed); this indicates that the transition actually maps to a subnet (it is a **supernode**), and the inscription beside it (in a dashed box) indicates that the subnet is defined on page AskOne#2. The inscription also specifies how all places adjacent to each transition (**socket nodes**) should map onto places in the corresponding subnet (**port nodes**). The result is as if we had overlaid an instance of the net on the subpage, superimposing the transition and the adjacent places. On the corresponding subpage (refer to Figure 8.2), an 'In' inscription by the **Input** place would indicate its role as a **port node** with input only from the superpage. Correspondingly, the place **Response1** would be marked 'Out'.

Ask#1, like the model in Figure 8.2, is a model of the message stream. The initial marking in Figure 8.2 is redundant. In this model, the results of the first **Ask-one** conversation instance will be deposited in the place **Response1** if the response was a `tell`; this toy model ignores all other responses. In that case, transition **T2** will be enabled, and the second stage messages in place **Prep** will be transferred into the initial place of the second `ask-one` query, which will then proceed as did the first, leaving a final result in place **Response2.**

Important but not illustrated in Figure 8.5 is the notion of a **fusion set. Fusion sets** are collections of nodes which behave as if they were one unit. They can be thought of as replicated nodes. **Fusion sets** can vary in scope and type, encompassing only nodes within a specific instance, or selected nodes from throughout a large model.

8.5 Summary

In this chapter, we have introduced the methods and notation needed for constructing CPN models of conversations. As mentioned, HCPNs are an extension which allow us to unify several CPNs into a larger model; we well discuss these in more detail as we demonstrate CPN integration into the supply chain integration scenario below.

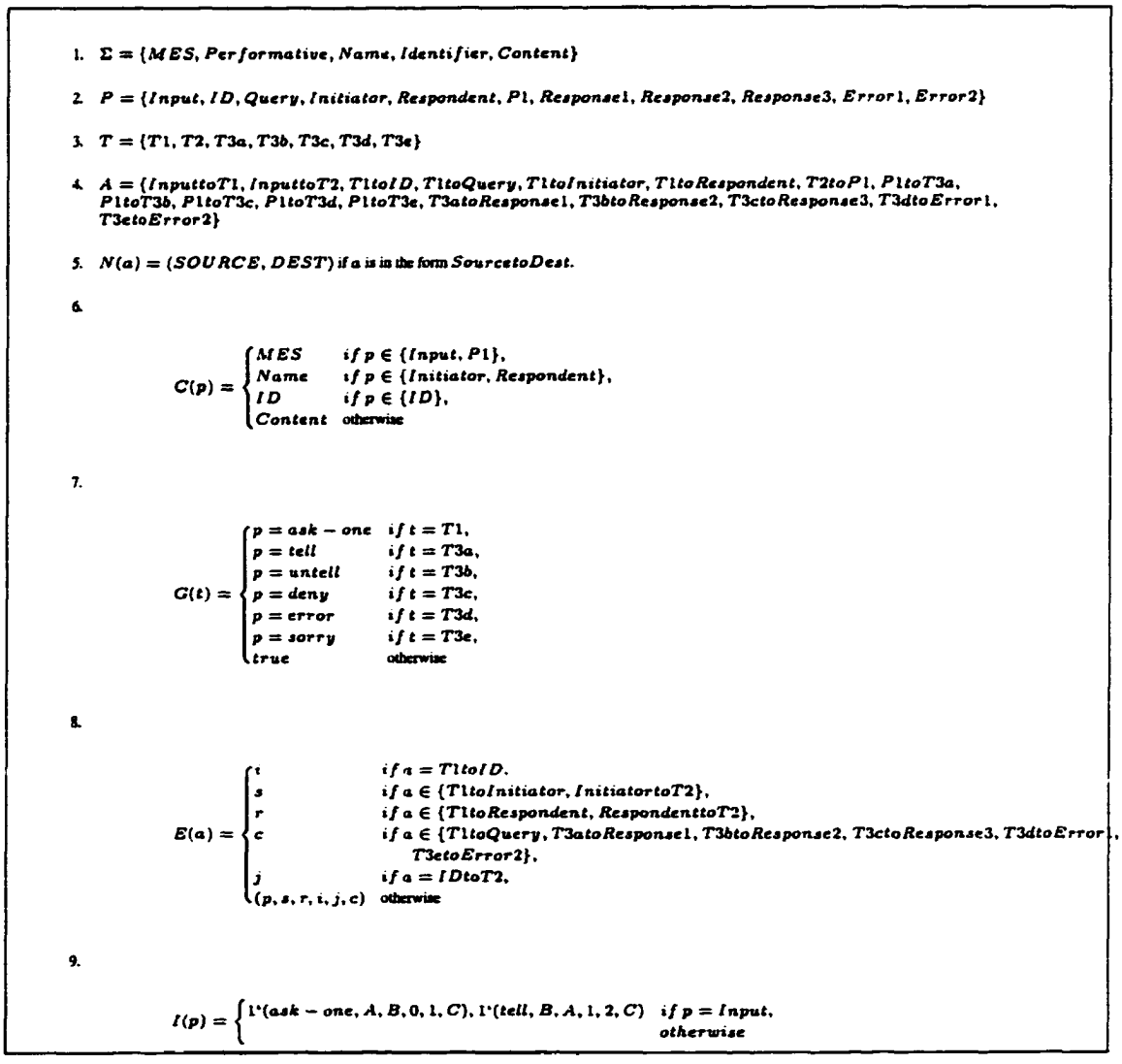


Figure 8.3: CP-net many-tuple representation of a KQML **Ask-one** conversation

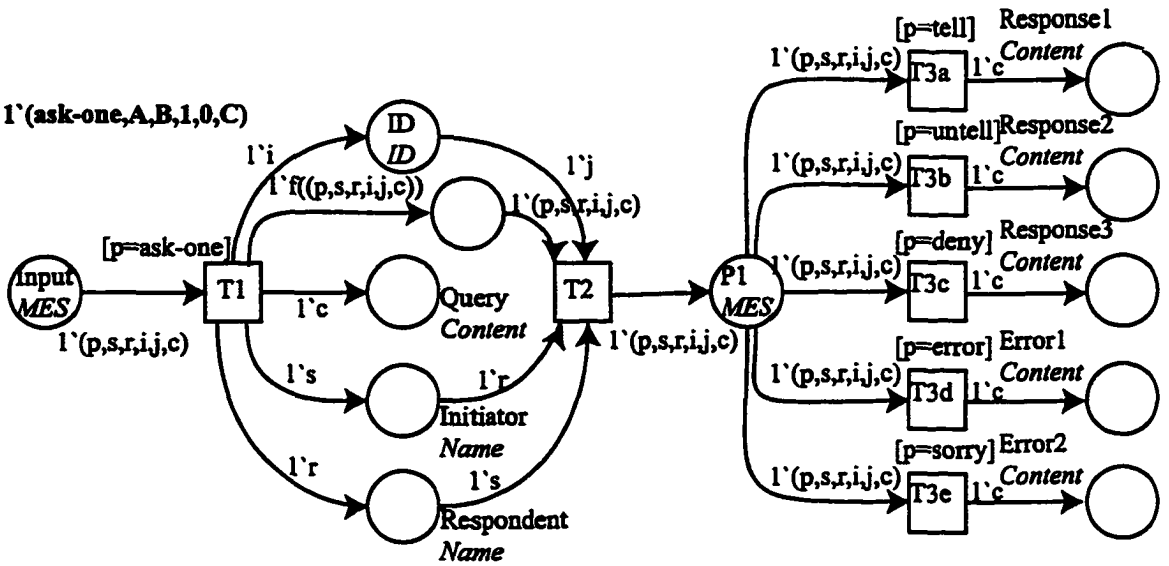


Figure 8.4: CPN representation of a KQML **Ask-One** conversation with agent-specific generative components

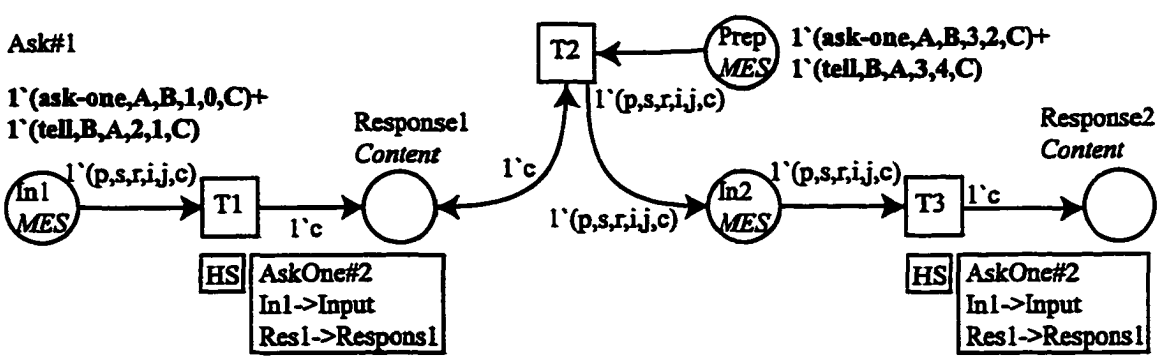


Figure 8.5: HCPN prime page for a multi-query KQML conversation

Chapter 9

Practica

CPNs can be used to specify interaction through many layers of abstraction. In this chapter, we present CPN representations of a number of interaction protocols which exhibit hierarchical relationships. In many scenarios, ACLs (e.g. KQML, FIPA ACL) form the base of all interaction. In this chapter, we show CPN models of some basic protocols from each ACL. Then, we implement an application level scenario, using the lower level (KQML) interactions that we have defined. The main purpose of this model is to demonstrate the utility of the CPN conversation specifications; that they can be executed, composed, and embedded in larger working models.

An issue that arises when modeling conversations is that the model itself is not always closed. The conversation specifications describe the message transactions within a small, closed context, but the actual communication is taking place between components of two or more entities at a higher level, or perhaps even outside the model. For instance, the messages of an **Ask-one** conversation are guided by the conversation specification, but the generation and consumption of the messages is performed components external to the specification. So, our model must either act as a verifier for a message stream, or contain

partially defined generative elements. In practice, we would like the models, at a minimum, to be able to constrain the message stream. In our model scenario, we have introduced partial closure in some conversations (KQML **Register** and **Advertise**). However, this becomes increasingly difficult with the level of complexity of the agent.

All models depicted in this section were developed with the Design/CPN tool from Aarhus University. The diagrams were generated by Design/CPN (in embedded postscript form) from the actual working specifications, and are presented without alteration.

9.1 CPN Specification of KQML

We use [Lab96b] as a reference document for KQML conversation specification, deviating in order to accommodate the agent system model in which the conversation [1] will be employed (below). The aforementioned work defines CPs concisely for nineteen performative-based interactions; in other words, the conversation used is dictated by the performative name of the initiating message. We define an instructive subset, as outline in Table 9.1. Conversation definitions themselves will be given in DCG; the corresponding CPN specifications will be used later in constructing a model of a real agent-based manufacturing integration scenario:

Register	Register a name and associated address(es)
Ask-one	Simple query/response
Advertise	Advertise a service that can be performed
Subscribe	Query and request update to the same query
Recommend-one	Query for the name of a service provider

Table 9.1: KQML Conversations Defined in CPN

[1] We will alternatively use the term 'conversation' to refer both to a prescribed sequence of messages, and the specification or model which implements it; the intended interpretation will be clear in context.

9.1.1 Ask-one

KQML **Ask-one** is a standard query-response conversation. The CPN specification in Figure 9.1 follows the DCG specification (Table 8.1). All messages are presented in the **In** place, which feeds all of the primary transitions (in this call, *all* of the transitions) in the net. Note that two transitions, **T1** and **T2** govern the flow of messages in the conversation. **T1** binds new messages with the `ask-one` performative (in our model, 'askOne'). It then copies the message to **Out** and **Query**, and places a **Signature** in **Sig**. **Out** is a place common to all conversations; it holds all messages accepted by the net. Those not destined for the local agent are typically sent immediately to the network. The conversation also has a place for each type of message; in this case, **Query**. **Sig** holds a **Signature** packs, consisting of the message ID, and the names of the sender and receiver. This will be used to constrain the message thread.

T2 binds incoming messages with performatives: `tell`, `untell`, `deny`, `error`, or `sorry`. Like **T1**, it copies the message to **Out**, and also to **Response**. It also binds a **Signature** from **Sig**. At least one must be present which properly constrains the sender, receiver and in-reply-to fields. Note that **Signature** packets are used (rather than IDs alone) in order to allow for the processing of multiple conversations concurrently.

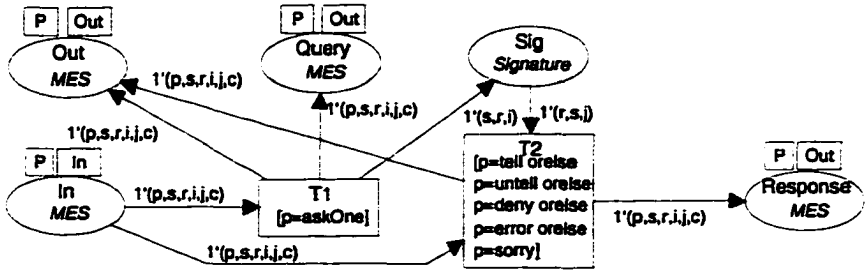


Figure 9.1: KQML **Ask-one** Conversation in CPN

9.1.2 Advertise

KQML **Advertise** is used by agents to advertise services that they provide to a service-based broker, or 'yellow-pages' service. It consists of a single `advertise` message with no positive acknowledgment (see Table 9.2).

The CPN specification is in Figure 9.2. Like the **Ask-one** conversation, places **T1** and **T2** bind new messages based on performative, and copy them to the **Out** place. **T1** caches a **Signature**, which is used to constrain the action of **T2**. Note that **T1** also constrains the performative of the embedded message. Unlike **Ask-one**, however, it also copies the messages to **AdDB**, a database of advertisements. Thus, this conversation, as mentioned above, is 'partially closed', in that it implements some of the functionality prescribed by the conversation's semantics. Note that because this conversation does not include positive acknowledgment, the 'advertiser' must assume success until presented with an `error` or `sorry`. So, the `advertise` is copied to the database immediately, and is removed by **T2** in the event of an exception condition.

```
s(CC,advertise,S,R,_,Rw,IO,_) →  {OI is abs(1-IO)},
                                 [[advertise,S,R,_,Rw,IO,[P1,R,S,Rw,_,OI,C1]]],
                                   {member(P1,[ask-if,ask-all,ask-one,stream-all,subscribe,
                                     insert,delete-one,delete-all,achieve])},
                                   c_adv(CC,P1,S,R,Rw,_,OI,C1)
c_adv(CC,P,R,S,Rw,_adv,_,IO,C) →  s(CC,P,S,R,Rw,_adv,_,IO,C) |
                                 problem(CC,S,R,Rw,_adv,_,IO) |
                                 []
```

Table 9.2: DCG representation of a KQML **Advertise** conversation [Lab96b].

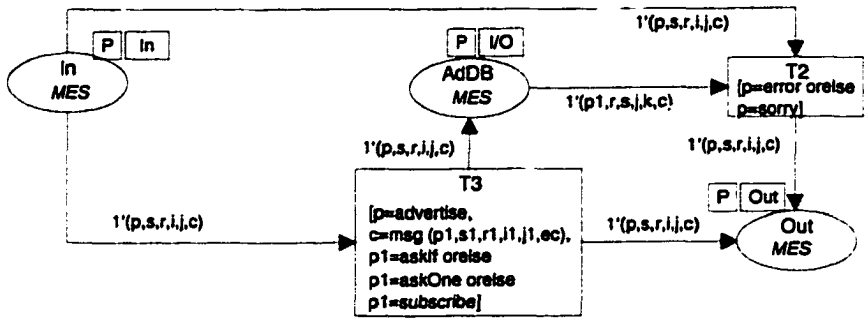


Figure 9.2: KQML **Advertise** Conversation in CPN

9.1.3 Recommend-one

KQML **Recommend-one** provides a protocol for obtaining a service-provider recommendation from a broker. The DCG specification of **Recommend-one** is shown in Table 9.3, and the CPN specification in Figure 9.3. It is essentially equivalent to **Ask-one**, except that: (1) **T1** and **T2** are constrained by different performatives, and (2) **T1** is further constrained by the performative of an embedded message. This message is taken as a description of the service required by the querying agent.

```
s(CC,recommend-one,S,R,IR,Rw,IO,C) →  {OI is abs(1-IO)},
                                      [[recommend-one,S,R,IR,Rw,IO,[P1,_,S,_,_,OI,C1]]],
                                        {member(P1,[ask-if,ask-all,ask-one,stream-all,insert,
                                          delete-one,delete-all,achieve,subscribe])},
                                        c_rec_one(CC,S,R,_,Rw,OI,P1,C1)
c_rec_one(CC,R,S,_,Rw_rec,IO,P,C1) →  problem(CC,S,R,Rw_rec,_,IO) |
                                      {OI is abs(1-IO)},
                                      [[forward,S,R,Rw_rec,Rw,IO,
                                        [advertise,Rec,_,R,Rw_adv,IO,
                                          [P,R,Rec,Rw_adv,_,OI,C]]]]
```

Table 9.3: DCG representation of a KQML **Recommend-one** conversation [Lab96b].

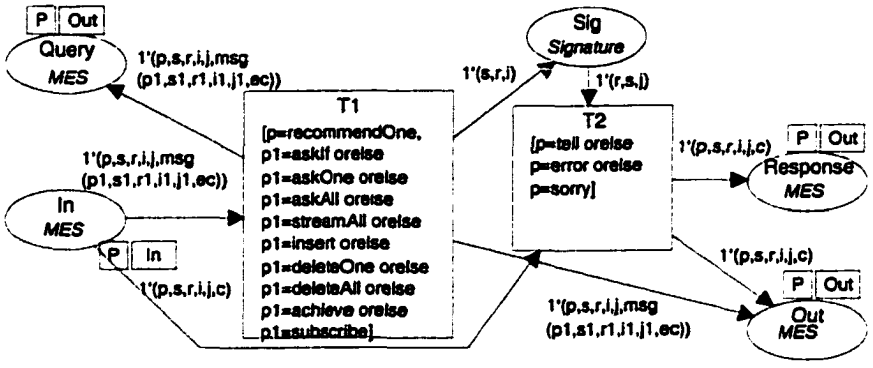


Figure 9.3: KQML **RecommendOne** Conversation in CPN

9.1.4 Register

KQML **Register** is one of the most fundamental KQML conversations, and perhaps one of the most heavily overloaded. KQML-based agent communities revolve around a central 'white-pages' server, the ANS. In early implementations, agents would not even run unless they had a constant connection to the ANS. In more recent evolutions, the role of the ANS has become less centralized and less pervasive. But the basic notion, that agents must register with (an) ANS(s) in order to participate in a community, remains.

A `register` message, like the subsequent `unregister`, is only negatively acknowledged. The DCG specification of **Recommend-one** is shown in Table 9.4. Our CPN specification (Figure 9.4) deviates in two ways. First, for simplicity, we have omitted the rarely-used transport-address performative. Second, in order to partially close this conversation, we have *over-specified* the conversation, by defining the form of the message content for conveying address information.

T1 functions as usual, but constrains the content field to the form of a type of address predicate. It also places registration information into **Reg**, the local registry/cache; this is legitimate, since no positive acknowledgment is forthcoming. As part of the conversation closure, **T2** has been split into two transitions, **T2a** and **T2b**, both bound by **Sig1**. The

former binds exception messages in response to the initial registration, and removes the corresponding entry from **Reg**. The latter binds an `unregister` message from the initial sender. It creates a new **Signature** for the possible responses, and removes registration information from **Reg**. Note that since an `unregister` may also raise an exception, the information is cached in **Arc**. If an exception to the `unregister` is bound by **T3**, the registration information is returned to the registry.

This conversation exhibits a significant synchronization problem, in that the initial registrant can send an `unregister` before receiving a previously sent `error` or `sorry` from the registrar. However, in this case, both agents will reach accepting states in the conversation, the registrant thinking it has successfully registered and then unregistered, and the registrar believing that no registration has taken place. The net effect is the same.

```
s(CC,register,S,R,IR,Rw,IO,C) →  [[register,S,R,IR,Rw,IO,C]], c_reg1(CC,S,R,IR,Rw,IO,C)
c_reg1(CC,S,R,IR_reg,Rw,IO,C) →  {OI is abs(1-IO)}, problem(CC,S,R,Rw,_,OI) |
                                 c_reg2(CC,S,R,IR_reg,_,IO,C) |
                                 []
c_reg2(CC,S,R,IR_reg,_,IO,C) →   [[unregister,S,R,IR_reg,Rw1,IO,C]],
                                   c_reg3(CC,S,R,_,Rw1,IO,C) |
                                 [[transport-address,S,R,IR_reg,Rw1,IO,_]],
                                   c_reg3(CC,S,R,_,Rw1,IO,C) |
                                 []
c_reg3(CC,S,R,_,Rw1,IO,_) →      {OI is abs(1-IO)}, problem(CC,S,R,Rw1,_,OI) |
                                 []
```

Table 9.4: DCG representation of a KQML **Register** conversation [Lab96b].

9.1.5 Subscribe

KQML **Subscribe** can be used to augment many standard query performatives. When embedded in a `subscribe` message, the intended meaning of a query changes from 'please

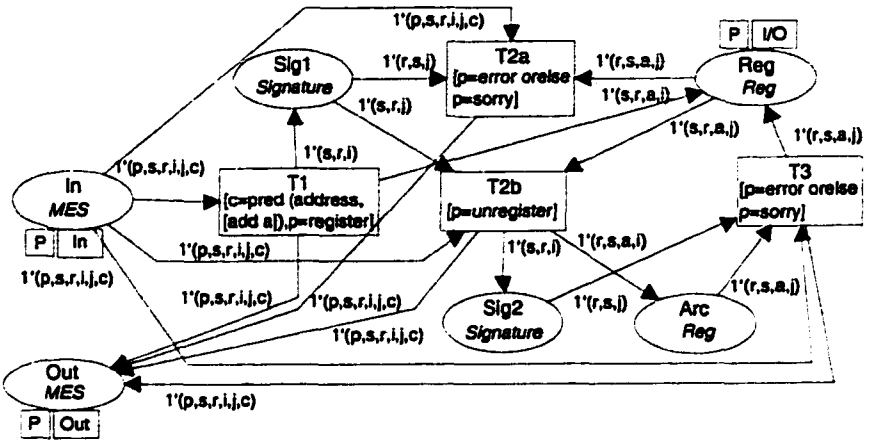


Figure 9.4: KQML **Register** Conversation in CPN

respond as appropriate' to 'please respond as appropriate, and, whenever conditions are such that your response would be different, send an update.'

The DCG specification of **Subscribe** (Table 9.5) shows that the `subscribe` message takes an embedded query. For the purposes of this model, we have significantly changed the form of this message, restricting it to an embedded `ask-one`, and flattening it to one level. Thus, a message of the form:

$$[subscribe, s, r, i, j, [ask - one, s1, r1, i1, j1, c]] \tag{9.1}$$

will be represented as:

$$[subscribe, s, r, i, j, c] \tag{9.2}$$

This step is being taken in order to simplify the model for clarity, by keeping the mes-

sage depth to a maximum of two, while avoiding the use of recursion in the message definition.

The CPN specification is shown in Figure 9.5. It is virtually identical to the **Ask-one** conversation (since this is really an implementation of Ask-one with multiple responses), except for one key difference in **T2**; if **T2** processes a message which is not an exception, it places a new **Signature** in **Sig** to replace the one it just consumed. This allows for an indefinite thread of non-exception responses to the initial `subscribe`.

```
s(CC,subscribe,S,R,IR,Rw,IO,C) →  {OI is abs(1-IO)},
                                  [[subscribe,S,R,IR,Rw,IO,[P1,S,R,IR,Rw1,IO,C1]]],
                                    {member(P1,[ask-if,ask-all,ask-one,stream-all,
                                      recommend-one,recommend-all])},
                                    c_sub(CC,P1,S,R,Rw,Rw1,OI,C1)
c_sub(CC,R,S,R,Rw_sub,Rw,IO,C) →  [] |
                                  problem(CC,S,R,Rw_sub,_,IO) |
                                  r(CC,P,S,R,_,Rw,IO,_),
                                    c_sub(CC,P,S,R,Rw_sub,Rw,IO,_)
```

Table 9.5: DCG representation of a KQML **Subscribe** conversation [Lab96b].

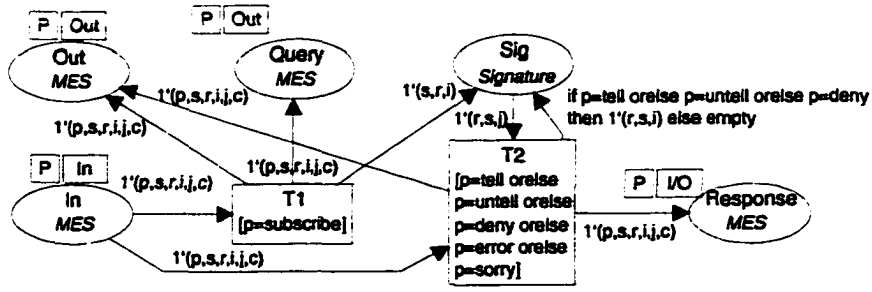


Figure 9.5: KQML **SubscribeOne** Conversation in CPN

9.2 CPN Specification of FIPA

FIPA, in the FIPA97 spec [FIP97], describes 8 protocols for agent communication. These protocols are specified at a slightly higher level than the performative-based CPs of the previous section. We examine an informative subset of these protocols: Request, Request-When, and Iterated Contract Net. The declarations in Figure 9.6, similar to those used for the KQML conversations, apply.

```
color Performative = with request | not-understood | refuse | agree |
failure | inform;
color Name = with A | B;
color ID = int;
color Language = with sl;
color Ontology = with example;
color Content = with C;

color MES = product Performative * Name * Name * ID * ID * Language *
Ontology * Content;
color Signature = product Name * Name * ID;

var p : Performative;
var s,r : Name;
var i,j : ID;
var l : Language;
var o : Ontology;
var c : Content;
```

Figure 9.6: CPN Declarations for FIPA Protocols

Note that the FIPA protocols are specified very loosely in [FIP97]. We have assumed message sequences based on reply-with and in-reply-to fields, and a sender/receiver alternation based on shading in the protocol diagrams.

9.2.1 FIPA Request Protocol

In the FIPA Request Protocol 9.7,agent (A) requests that another agent (B) perform some action. B may refuse or accept, and, if it accepts, must report the results or the reason for failure.

The CPN model for Request 9.8 is quite straightforward. **In** feeds the transitions in the three levels of this CP. **Request** binds an initial request and caches a **Signature** in **Sig1**.

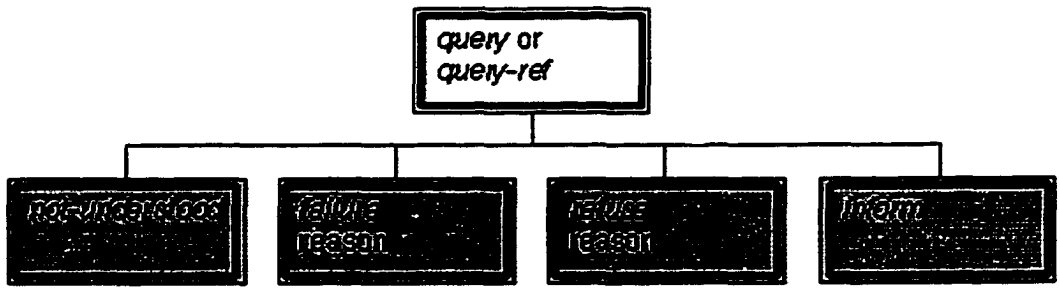


Figure 9.7: FIPA **Request** Protocol

This **Signature** constrains the transitions in the next level, **NU**, **refuse** and **agree**, one of which will bind the second message. If **agree** binds, it caches a **Signature** in **Sig2**, which governs the third and final layer. The last message will be bound by either **failure**, **inform1** or **inform2.**

In the previous section, we included extensions to the CPs which allow their integration into a specific model. Without such a context, it is difficult to meaningfully include the SL components of the protocol, as given in Figure 9.7. Presumably, the SL constraints would be satisfied by the agent after the message was processed in the CP.

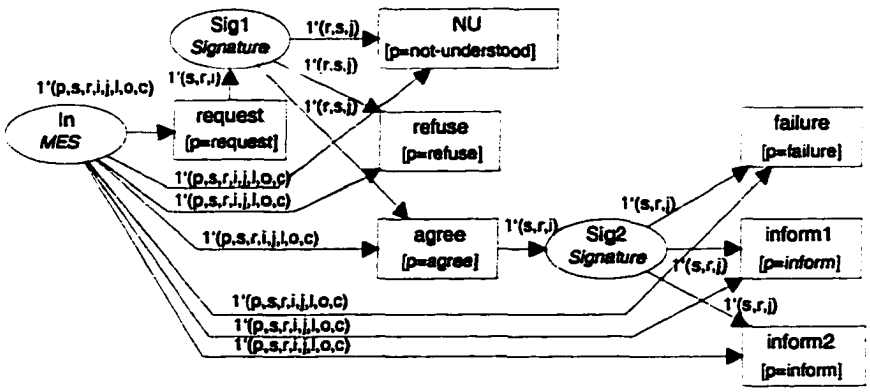


Figure 9.8: FIPA **Request** Protocol in CPN

9.2.2 FIPA Request-When Protocol

Except for some SL constraints, FIPA Request-When 9.9 is virtually identical to FIPA Request. The CPN specification is given in Figure 9.10.

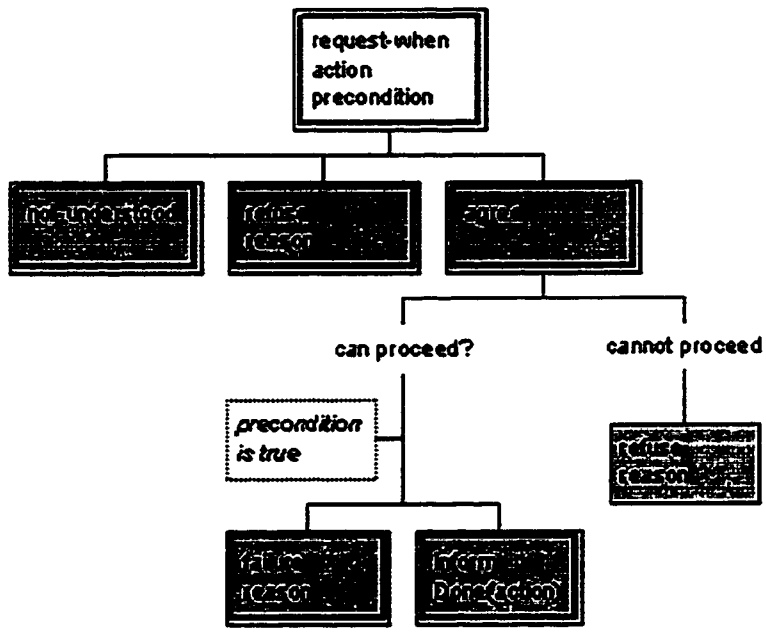


Figure 9.9: FIPA **Request-When** Protocol

9.2.3 FIPA Iterated Contract Net Protocol

Because of its depth, the CPN specification (Figure 9.12) of the FIPA Iterated Contract Net Protocol (Figure 9.11) has been arranged in a top-down fashion; the structure is essentially the same as the previous two, except for the additional level. The one key difference, present in the FIPA specification, is the loop from **reject-proposal** back up to **cfp**, for iterated negotiation. This has been omitted from the CPN specification, because the protocol specification lacks sufficient information to implement it in a meaningful way. We have constraint message sequence by ID, but since there is no constraint on the ID of an initial

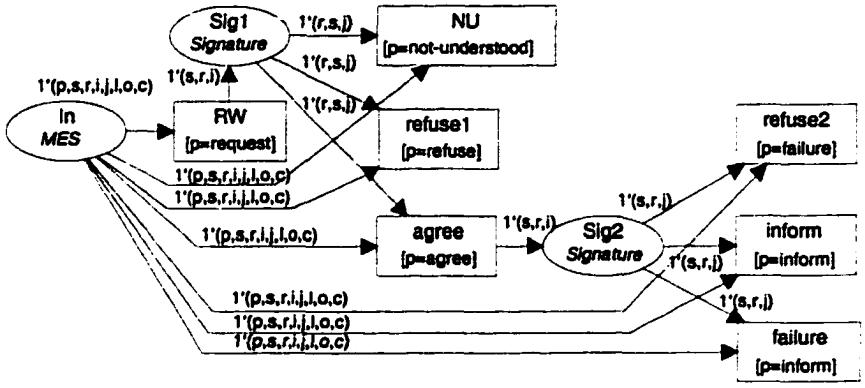


Figure 9.10: FIPA **Request-When** Protocol in CPN

message, all cfps should be accepted. Further, there is no use in the protocol itself for contextual information regarding previous iterations. We assume such information is made use of at a higher level.

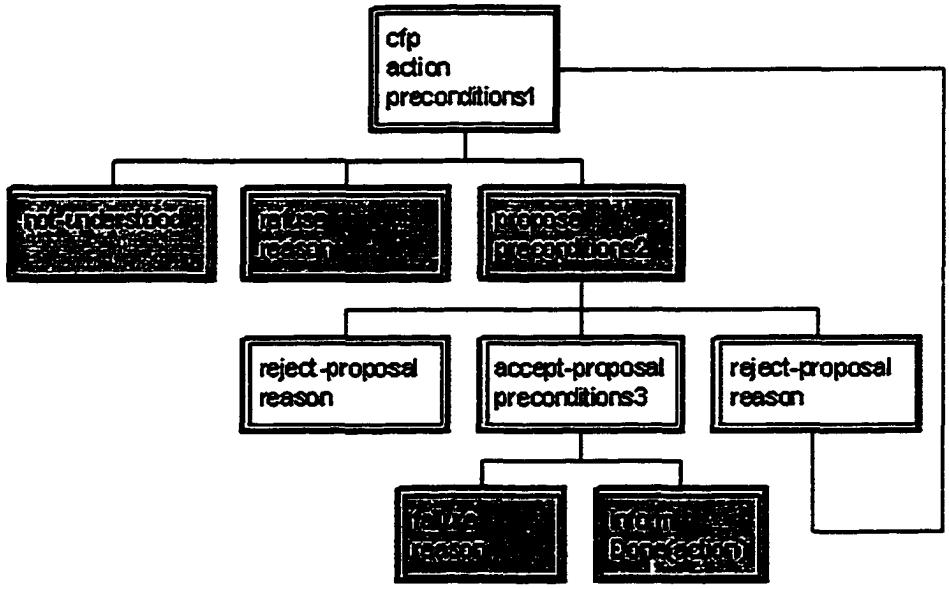


Figure 9.11: FIPA **Iterated Contract Net** Protocol

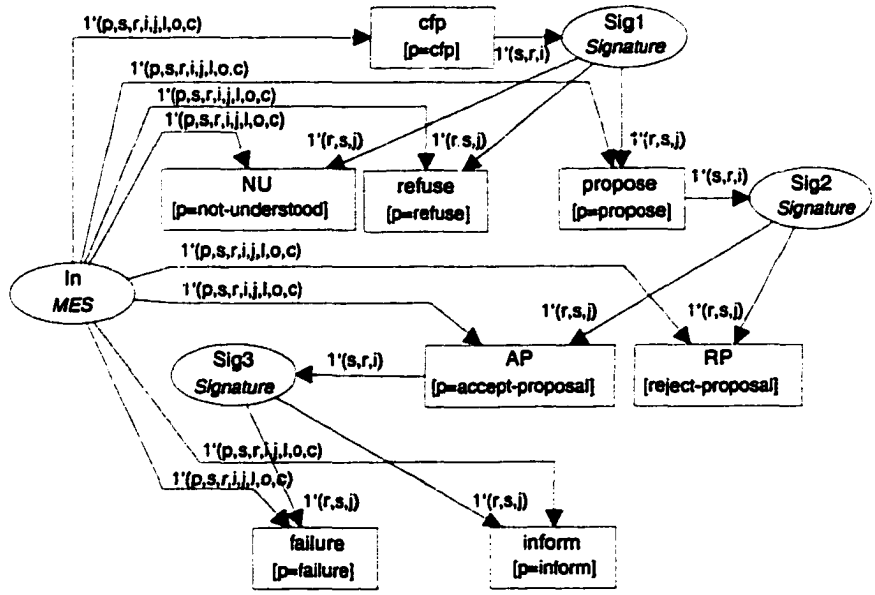


Figure 9.12: FIPA **Iterated Contract Net Protocol** in CPN

9.3 The Process Rate Scenario Revisited

We employ the KQML conversation specifications developed earlier to construct a model of the CIIMPLEX project's Process Rate Scenario. While the model is not complete, it illustrates the approach to conversation modeling we have described.

9.3.1 The Scenario

The scenario consists of six KQML speaking agents connected to a network. An actual address-based network model is used, and serves as the only link between individual agent models. Agents are addressed by symbolic names, but addresses, which are obtained from an ANS, are required in order to communicate through the network. A Gateway Agent (Gateway) provides a data stream to a Process Rate Agent (PRA), which converts the stream into summary data and provides it to the SCA. The SCA then provides event notification

to the Monitor Agent (Monitor), the end-user in this scenario. All services are identified through a Broker Agent (Broker), through advertisements sent by service providers.

9.3.2 Assumptions

In order to construct a complete model of the scenario, we must make a number of simplifying assumptions.

1. Message Format. A message is a six-tuple consisting of performative, sender, receiver, reply-with and in-reply-to tags and content. Sender and receiver names are enumerated, and message tags are integers. Content is either one of a small set of defined predicates, or an embedded message. The embedded message type is the same, except that it cannot contain further embedded messages. Thus, for this model, messages have a fixed maximum depth of two.

2. Time. There is no notion of time in this model.

3. Query Failure. Since we are interested in presenting the initial structure of the interaction, and since this model does not incorporate time, we do not implement error conditions. In particular, message order events are not fully represented. For example, in a more complete model, queries arriving at the ANS for an agent which had not yet registered should be met with `sorry` responses, and the model would then have to be enhanced with action ordering constraints or retry mechanisms. In our simplified model, an early query will wait, and be satisfied when the information becomes available.

4. Network. We implement a simple network model with perfect, instantaneous delivery by static agent address. Each agent's address is known initially only to that agent,

except for the address of the ANS, which is known a priori by all agents.

5. Indefinite Subscribe. **Subscribe**, as it is defined in [Lab96b] results in an indefinitely long stream of responses, if successful. For the CIIMPLEX scenario, the specification was extended to include a subscriber-initiated termination handshake. We use here **Subscribe** in its original form, but have restricted it to use with `ask-one` queries. This allows us to use it without extending our message to three or more levels of embedding.

6. Loose Ends. Given that many events in the model are simply not generated, support for them is often omitted. For example, the **Ask-one** conversation in is used exclusively in this limited scenario to request addresses from the ANS. Therefore, no provision is made for queries received by other agents, even though they incorporate the **Ask-one** conversation, and in theory support query handling.

9.3.3 The Model

A separate model is constructed to represent each agent in the scenario. The models are not connected in any way except through a model network, which accepts addressed packets and routes them to the agent bearing the given address. Agents are initially aware of their own address and that of the ANS only.

All agents have a similar construction; they are composed of references to some common subnets[2] (the conversation templates described above), and some components unique to the function of that agent. For clarity, the later are moved onto separate pages and joined to the respective agent bases with fusion nodes.

[2]Subnets are 'common' in that their specification is shared by a number of agents. However, each reference generated a unique instance, so no nets are actually shared, only their specifications.

Declarations and Hierarchy The Hierarchy graph in Figure 9.13 show the relationships among the 'primary' pages of the model; nets which are not referenced as subnets on any page in the model. The six agents are arranged around the conversation templates, and have connections to those that they reference. **Register** is the only exception, as it is embedded in a **NIC** used by each agent. Agent specific code which has been moved off-page can be seen as nodes connected to **SCA**, **PRA** and **Gateway**.

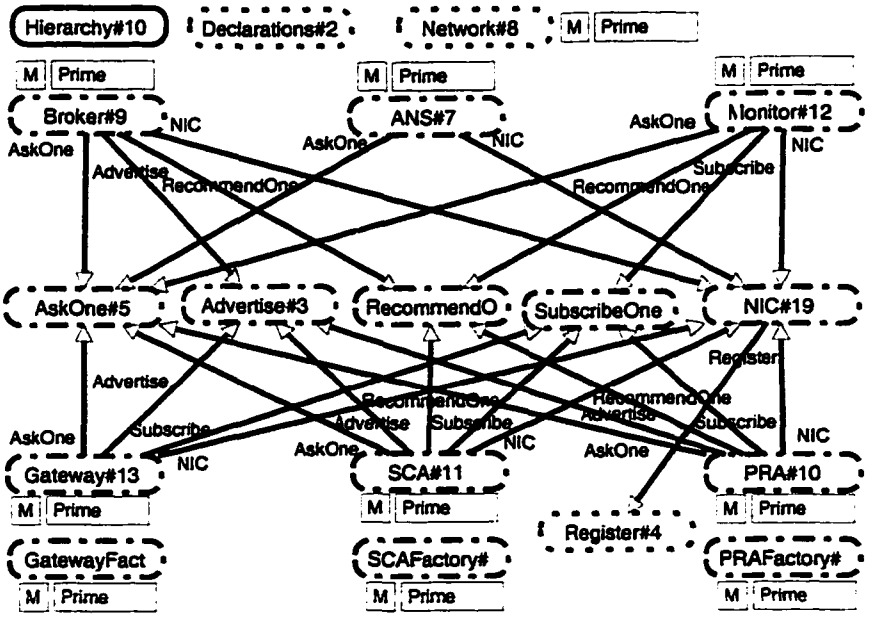


Figure 9.13: Hierarchy of Nets in the CPN Model of the PRA Scenario.

The Declaration Node, Figure 9.14, contains the ML declarations for all colors (types), variables and, if needed, functions. Names (**Name**), addresses (**Address**) and performatives (**Performative**) are enumerated, and message IDs (**ID**) are integers. The only remaining component of the message, the content, is allowed to be a predicate or another message. For this, in order to avoid implementing a recursive message type, more than is needed in this simple scenario, we construct two message types; a regular message (**Mes**) and an embedded message (**EMes**). **Mes** can take a **Predicate** or an **EMes** as content, as dictated

by its **Content** type. **EMes** is essentially the same, but takes **EContent** as content, which permits only **Predicates**.

Other colors are defined for various special purpose data types. **Reg** is used to store registration information about an agent. **Signature** data is used to identify and verify message responses. A **Packet** is used to convey a message and its address through the network. **Data** holds the sample rate data which drives the scenario.

```
color Performative = with register | unregister | advertise | error |
sorry | askIf | askOne | askAll | tell | untell | deny | streamAll |
insert | deleteOne | deleteAll | achieve | subscribe | recommendOne;
color Name = with ANS | Broker | PRA | SCA | Gateway | Monitor |
AnyName;
color ID = int;
color Int = int;
color Address = with ans | broker | pra | sca | gateway | monitor |
anyAddress;

color PVal = union num:Int + W + add:Address + nam:Name;
color PVals = list PVal;
color PName = with rate | address | agentName | rmean | event;
color Predicate = product PName * PVals;

color EContent = union epred:Predicate + EC;
color EMes = product Performative * Name * Name * ID * ID * EContent;
color Content = union pred:Predicate + C + msg:EMes;
color MES = product Performative * Name * Name * ID * ID * Content;

color Reg = product Name * Name * Address * ID;
color Signature = product Name * Name * ID;
color Packet = product Address * MES;
color Data = product Int * Int;

var c, c1 : Content;
var ec : EContent;
var m, n, message : MES;
var s, r, s1, r1, s2, r2, anyName, name : Name;
var i, j, k, i1, j1, i2, j2: ID;
var p, p1 : Performative;
var a : Address;
var x, N, N1, M, M1, S, S1 : Int;
```

Figure 9.14: Declarations of Colors, Variables and Functions

Network The scenario could have been modeled by connecting the agents to a common 'bus' or some such mechanism. However, **Register** and registration queries are some of the conversations we would like to model, and their correctness affects the ability of agents to communicate. So, the address-based network model (Figure 9.15) enforces the requirement that agents correctly register, and that they query for and successfully obtain an address

before they can communicate with any other agent.

The model itself is extremely simple. All agents connect to the **In** place via a global fusion set. For each agent, there is a transition **T***n* which is keyed for that agent's address, and which deposits any packets with that address in a place assigned to that agent. Each agent is bound to its particular output place.

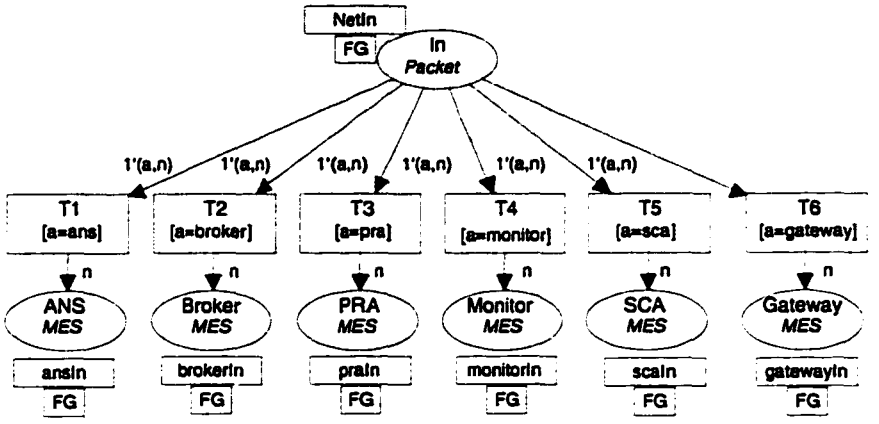


Figure 9.15: Network Model

In order to simplify the agent some, we have removed the comment network interfaces to a subnet, the (**Network Interface Component (NIC)**), depicted in Figure 9.16. Because it is so closely tied to network communication, the **NIC** embeds the **Register** conversation. As with most conversation templates, messages are accepted at the **In** place, and if correct, are copied to the **Out** place. Since **Register** also implements the registration functionality, **RegDB** serves as a registry database or cache.

The **NIC** implements a smart address cache; agent components send messages, addressed by name, by depositing them in the **Out** place. If **Xmit** is able to match a waiting message with an address from **RegDB**, it will assemble them into a packet and forward them to **Network**.

In most cases, the first time an agent tries some other agent, there will be no address

in the cache. In this case, **NIC** fetches the address automatically. This is implemented by assuming that the agent knows the names of all agents in the world; these names reside in **CWorld**, for 'closed-world'. **Cache** will bind any outgoing message with a name from **CWorld**, and send an address query for that agent to the ANS. In the same action, it removes the name from **CWorld**, so that no further inquiries will be made about that agent. If the remainder of the agent works correctly, an address will eventually appear in the cache, and the message will be sent. There is an alternative approach which does not require a closed-world assumption, but it involves implementing a search of the registry, and is slightly more complicated.

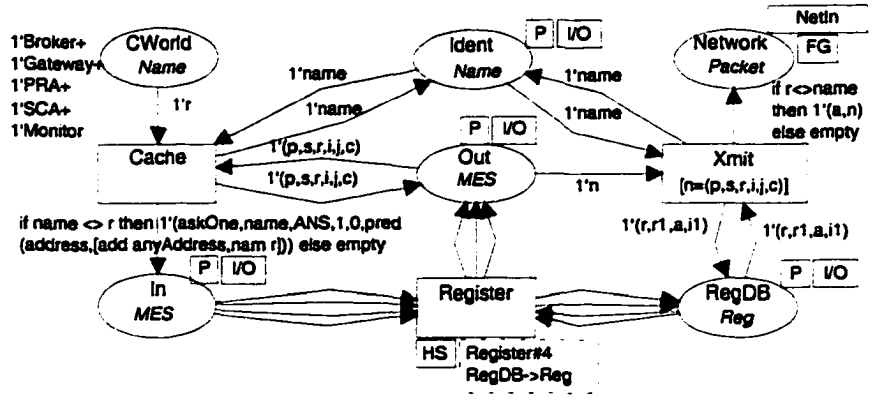


Figure 9.16: Network Interface Component (NIC)

The Agents Now we will examine each of the six agents. As they all follow the same basic pattern, we will restrict description when possible to features not introduced in previous models. Although it might have been simpler to include a template with all conversations in each agent, we have included in each only those conversations required by the agent, for clarity.

All agents are coded to initially register with the ANS and, if appropriate, to advertise their services with the broker. All other actions are triggered at appropriate times in the

course of interaction. The one exception is the **Monitor** agent, which is coded with the one initial message which starts the scenario; a query to the **Broker** requesting that it recommend an agent which can provide notification of rate mean exception events.

ANS As do all agents, the **ANS** (Figure 9.17) embeds the **NIC**. In addition, it embeds the only other KQML conversation that it needs; **AskOne** allows it to handle queries about the addresses of registered agents. This functionality is implemented with only one transition, **ansQuery**. **ansQuery** binds properly formed queries in **Query** with a registration of the same name in **RegDB**, and generates a response containing the corresponding address. If no address is present for that name, the query will wait. A negative response could be generated by implementing a search of **RegDB**, as discussed above.

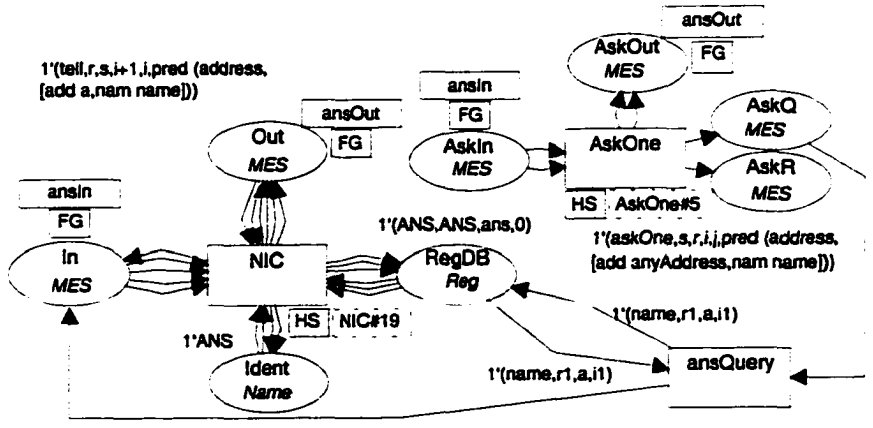


Figure 9.17: Agent Name Server (ANS)

Broker The **Broker** (Figure **??**) has a very similar structure to the ANS, but embeds two new conversations; **Advertise** and **RecommendOne**. Both are required for implementing the broker functionality. Much like the **ANS**, the **Broker** is implemented with only two transitions beyond the embedded conversations. In place of **ansQuery**, we have **ansLook**,

which implements the client end of the ans address query. When a response to a query is returned, it will end up in the **Response** place of **AskOne**. Since the **AskOne** conversation is only used for ans queries in this scenario, we assume that only address-bearing messages will be found in this place. **ansLook** will bind responses from the **ANS**, assumed to be non-error responses, and place the extracted address information into the registry. This is considered to be a cache entry, since the registrar named in the registry entry is the **ANS**, not the current agent.

The implementation of the broker functionality is similar that of the ANS functionality. `Advertise` messages sent to **Broker** end up in the **AdDB** place, by way of **Advertise**. One transition, **AdLook**, binds incoming RecommendOne queries to messages in **AdDB** with matching content, and generates a response containing the name of the advertising agent. As with the **ANS**, unmatchable queries do not result in errors, but wait for a match to become available.

Gateway The **Gateway** agent (Figure ??) does not need **RecommendOne**, since it does not require special services of any agent. It does need the **Subscribe** conversation, since it provides a service other agents need; access to the sample data-stream. It advertises this services as a subscription to the **rate** predicate.

Services provided by the **Gateway** are depicted in Figure 9.20. **Query** is bound to the corresponding place on the agent's **Subscribe** conversation, and **In** is bound to the agent's **In** set. The transition **Serve** binds incoming subscription requests, and extracts and stores the sender's name and message ID. Once these two are present, **T8** is enabled, and may begin sending data. The sample data is stored in **Data**. This small data-stream is enough for two rate-mean samples. The first four are legal values, but the next four constitute an exception condition.

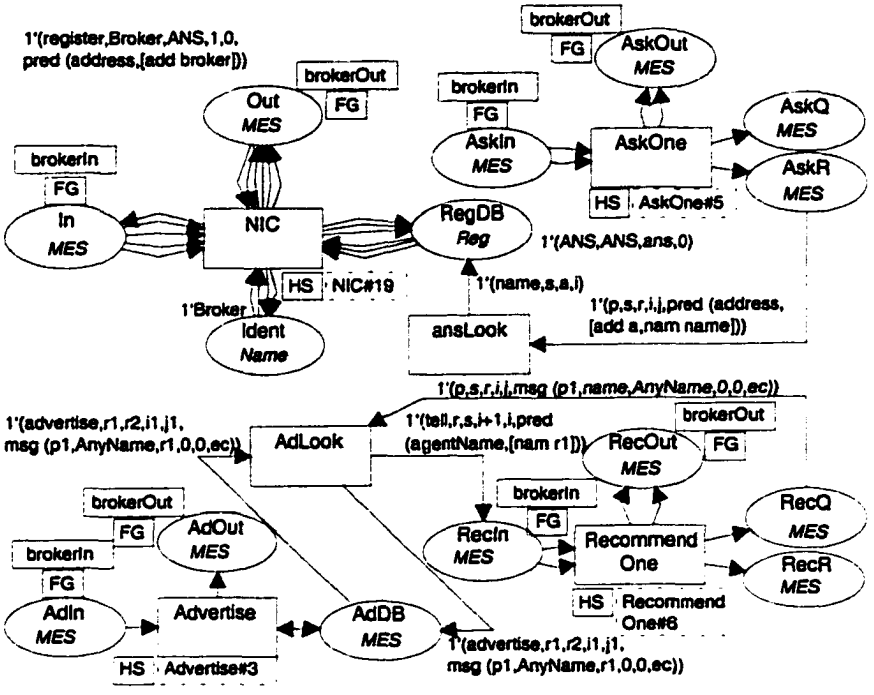


Figure 9.18: **Broker** Agent

PRA The role of the **PRA** is to interface with raw data sources, which provide job completion rate data, and refine that data into periodic rate means.[3] In this scenario, the **PRA** obtains data from the **Gateway** agent, which it identifies by requesting a recommendation from the **Broker**. Means are advertised as subscriptions to the **rmean** predicate. **PRA** requires the use of the **SubscribeOne**, **RecommendOne** and **Advertise** conversations.

Services provided by the **PRA** are depicted in Figure 9.22. The diagram is slightly complex; this is because the model encompasses all aspects of the scenario, including operations on the data stream. Often, it is sufficient to model the *operation* of a MAS, and code numeric operations in the implementation itself, perhaps in some procedural language. Since this model is a proof of concept, we have extended it to a greater level of detail. Three

[3]In this model, an accumulated rate sum is kept, and compared with a threshold.

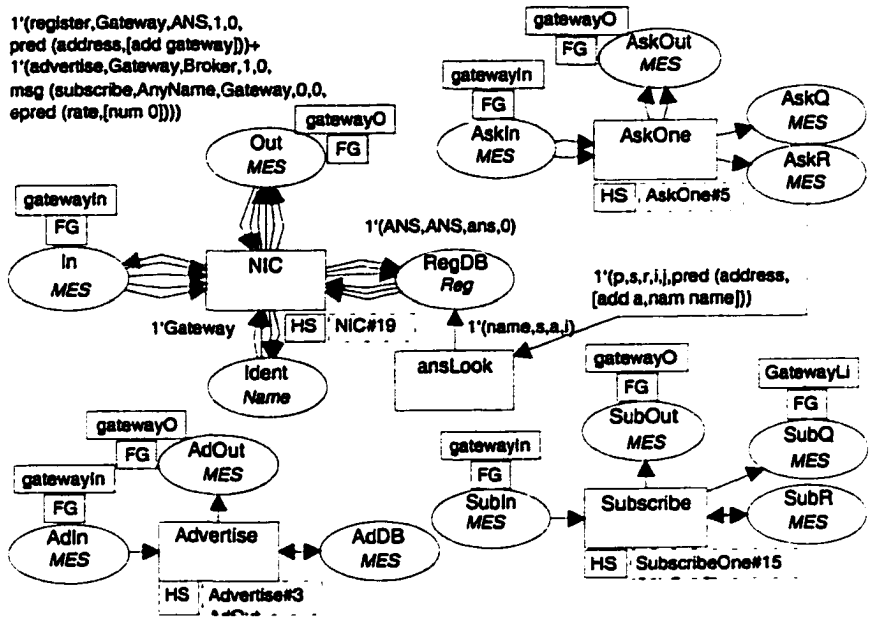


Figure 9.19: **Gateway** Agent

basic operations are represented:

- **T8** picks up an incoming `subscribe` request from **Query**, and caches the name of the agent and the message ID. In addition, it sends a recommendOne request to the **Broker** for a **rate** source.

- The response to the previous `recommendOne` query, when it ends up in **Response**

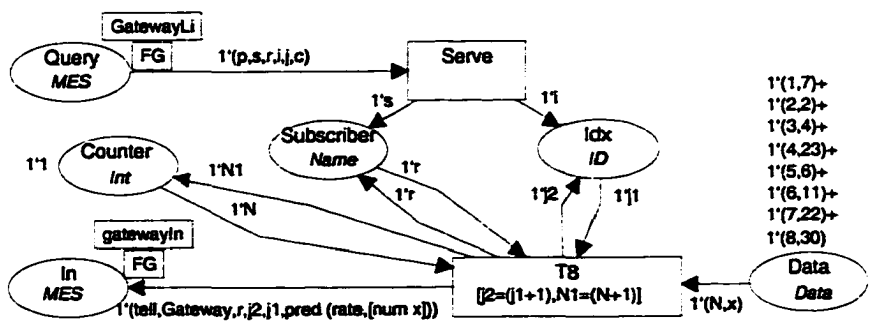


Figure 9.20: **Gateway Factory**

(the uppermost), is consumed by **T7**, which extracts the name of the service provider to **Server**. This enables **T10**, which caches the server name in **Engaged**, and sends a subscription request to the named server, **PRA**.

- The remaining spider-like component, clustered around **T9**, performs the computation. Arriving data will be found in the **Response** (lower) place of the **Subscribe** conversation. **T9** picks up each rate as it arrives, and adds it to the value in **Sum**, also incrementing the value in **N**. Every four turns, **T9** also computes the periodic rate mean. If the mean computed is different from the last value reported (cached in **Record**, it sends an update to the agent which subscribed, using the cached name and message IDs. It also update the value in **Record** and message ID (**Idx**). In all cases, the accumulated sum is set to the new value, the counter (**N**) is set to 1, and the process repeats itself.

Note that we assume in-order arrival of data items, and that each update is picked up for computation before new ones arrive. Ordering constraints could be imposed either through the use of the message IDs, or by embedding sequence data in the rate predicate.

It is important to observe the dual role played by the **SubscribeOne** conversation in this agent. It serves concurrently as a client to the **rate** source, and as a server to the **rmean** subscriber.

SCA The **SCA** (Figure 9.3.3) is outfitted just as the **PRA**, with some minor changes to its operational component (Figure 9.3.3). Transitions **T7**, **T8** and **T9** play the same roles, except that the service retained is **rmean** data, and the service provided is notification of above-threshold **rmean** values. When processing a subscription, **T8** caches not only the name and message ID, but also the threshold value in the **event** predicate. **T9** is signifi-

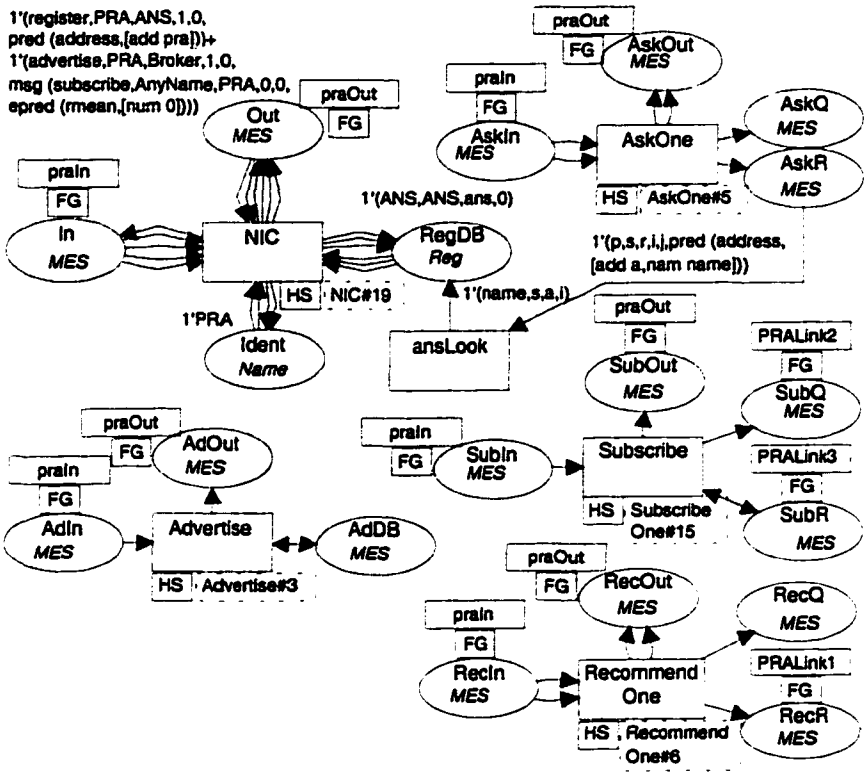


Figure 9.21: Process Rate Agent (**PRA**)

cantly less complex in the **SCA**; as **rmean** updates arrive, **T9** consumes them, and if the mean is greater than the value in **Thresh**, sends a notification to the subscribing agent, and updates the message ID index in **Idx**. This component is not guarded against sending repeat notifications.

Monitor The **Monitor** agent (Figure 9.25 is the final agent in the chain. Embedding the same conversations as the **PRA** and **SCA**, the action of the **Monitor** is implemented in only one transition, **Uplink**, which takes the name of a notification server, recommended by the **Broker**, and send a subscription request to that agent, for rate mean exception events of threshold 42. Any such notifications will end up in the **Response** place of the **SubscribeOne** conversation.

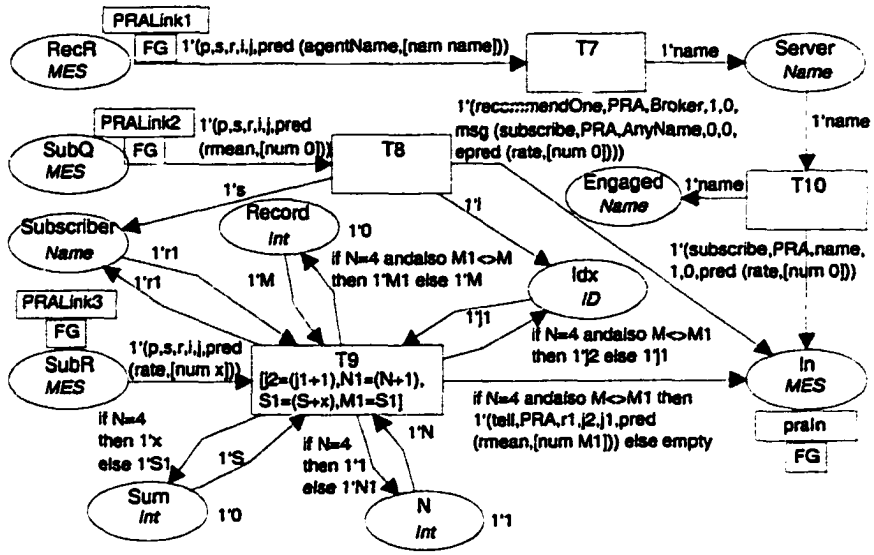


Figure 9.22: **PRA Factory**

9.3.4 Simulation

The model scenario as specified completes simulation in Design/CPN after 373 transition firings, using 54 messages (see Table 9.6), and results in the correct notification of the Monitor Agent of an exception event after processing the second half of the data stream. The non-interactive simulation (no intermediate visualization) runs on a Sun Sparc Ultra in

	Initial Message	Resultant Messages	Total
Register	5		5
ANS Query	13	13	26
Advertise	3		3
Recommend	3	3	6
Subscribe (Monitor)	1	1	2
Subscribe (SCA)	1	2	3
Subscribe (PRA)	1	8	9
Total	27	27	54

Table 9.6: Messages in the PRA Model

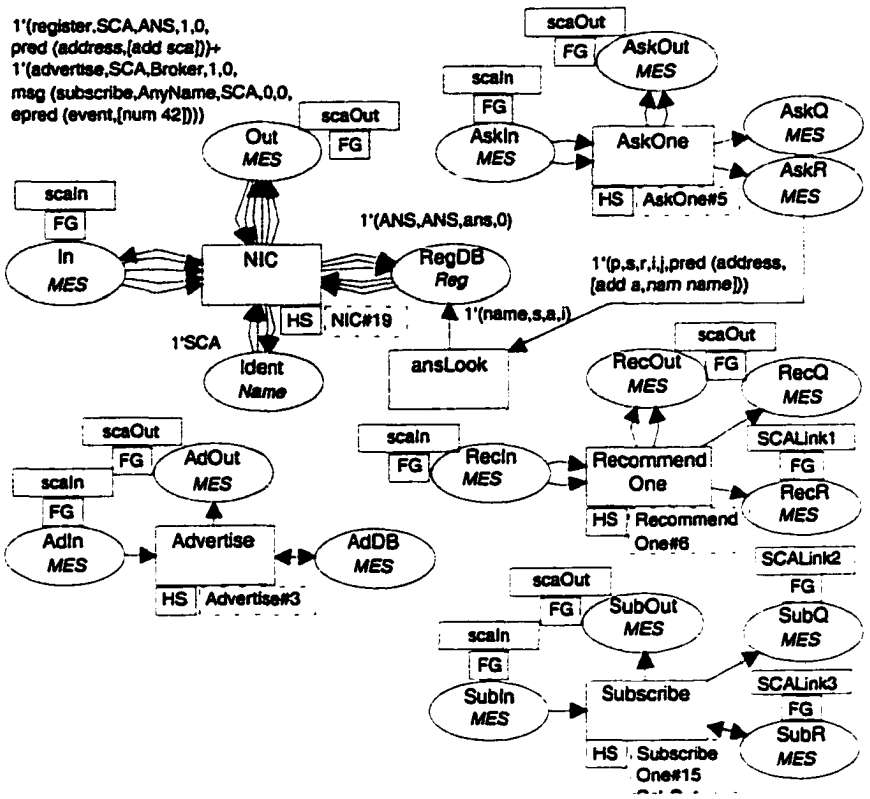


Figure 9.23: Scenario Coordination Agent (SCA)

roughly 10 seconds (wall clock).

9.3.5 Discussion

Our experience with this model has shown the following:

1. CPN models are relatively easy to build, especially with the help of Design/CPN. Both the formalism and the design tool have relatively steep learning curves. This model was constructed, including training time, in roughly 30 person/hours.

2. Fairly complex models can be constructed and executed. This model, while simplified in many respects, is quite complete, down to the actual details of computation.

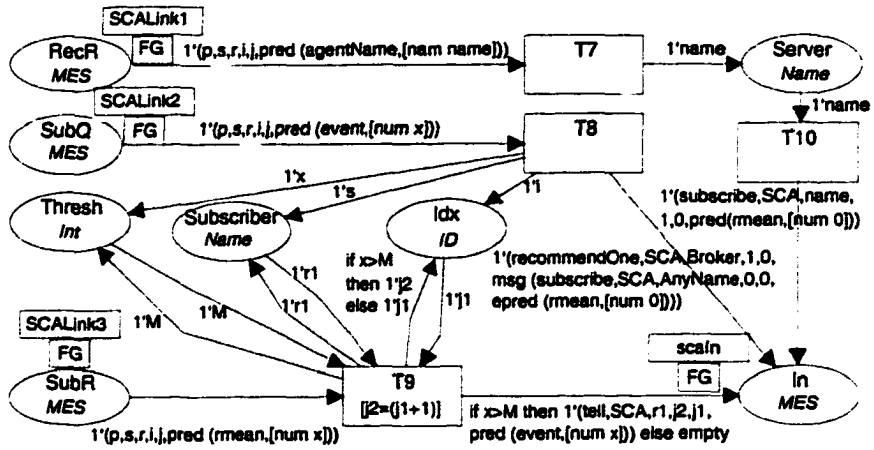


Figure 9.24: **SCA Factory**

3. While is actually possible to arrive at many 'clever' means of doing things (e.g. computations, search operations) in CPN-ML, the type constraints make representation somewhat cumbersome. For example, it is not simple to implement a recursive type, which is why for demonstration purposes we chose a bounded message structure for the PRA model.

4. The graphical nature of CPNs and the visual environment provided by Design/CPN facilitate debugging enormously. The distribution of tokens is easily inspected, and most problems are easily recognized by the presence of tokens in incorrect locations, or trapped in places from which they should have been consumed.

9.4 Verification

The ability to verify the properties of a specification is one of the important benefits of applying formal methods. These benefits can be derived in two ways: Verification of CPs or protocols directly, and verification of agents/MASs that are based on such protocols.

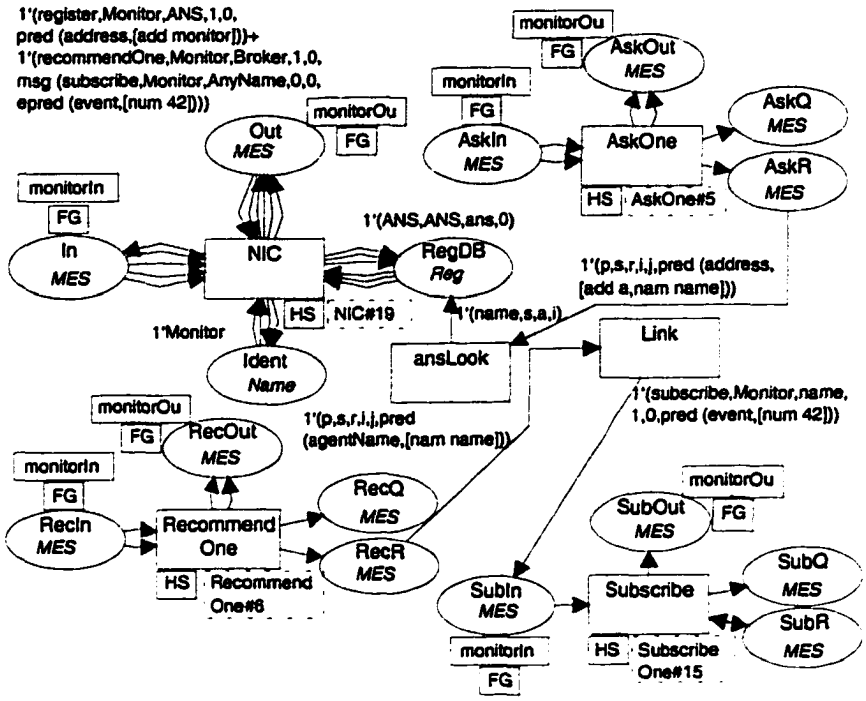


Figure 9.25: **Monitor Agent**

We will first consider the range of properties amenable to analysis, and then discuss their value in the two contexts described. The focus will be on the methods provided by Design/CPN and associated tools.

In addition to 'proof by execution', CPNs can be checked for a variety of properties, including liveness, fairness, boundedness, home properties, and reachability (described below). This is done by way of an Occurrence Graph (OG) [Dep96]. Each node in an OG consists of a possible marking for the net. If another marking (B) can be reached by the firing of a transition, the graph contains a directed arc from the node representing the initial marking to B. All nodes in an OG are therefore derived from the initial marking of the net.

We illustrate this with an example taken from the Design/CPN Occurrence Graph Manual [Dep96]. Figure 9.26 shows the classic dining philosophers problem. Briefly, each of the five philosophers may eat when there is a chopstick available to either side of him. This

system is modeled by the CP-net shown in Figure 9.27. Philosophers and chopsticks are represented by icks maps each philosopher in

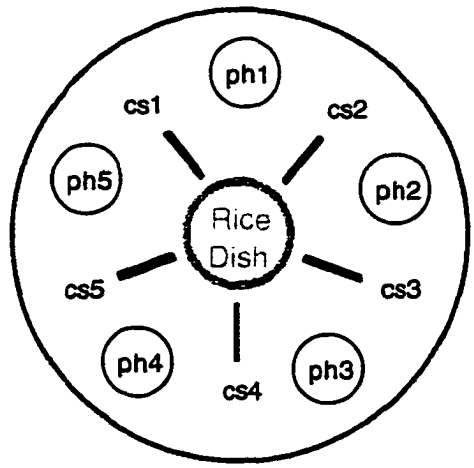


Figure 9.26: Dining Philosophers

The OG for this system is seen in Figure 9.28. For readability, some annotations have been omitted. Each node represents a reachable marking, and is annotated with a label, the degree in and the degree out. Some are annotated with the marking they represent, and some arcs are likewise annotated with the corresponding binding element.

The properties listed above are defined as follows (Given an OG G):

1. Reachability Properties: This relates to whether or not a path exists in G from node A to node B (i.e. the marking denoted by node B is reachable by some sequence of transition firings from node A). It is also possible to discover if B is reachable from A in a strongly connected component of G, or if G itself is strongly connected. Note that in Figure 9.28, all arcs are double arcs, so every marking is reachable from every other.

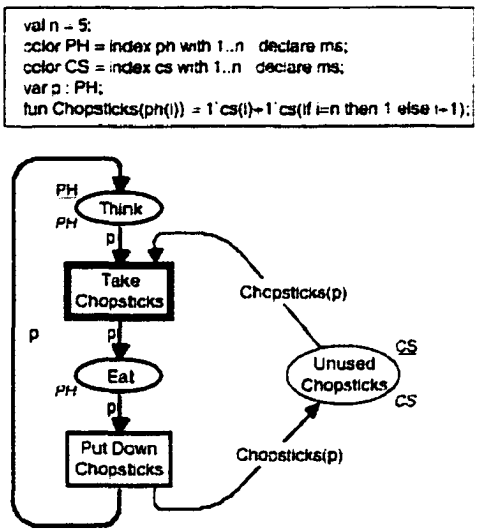


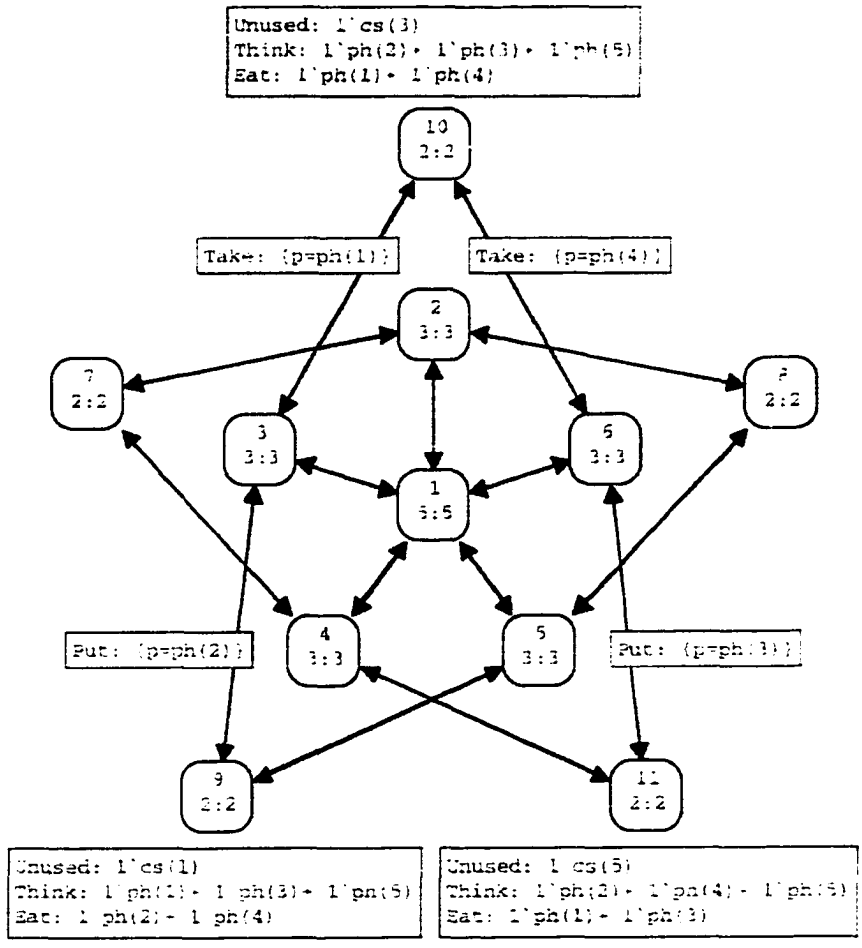
Figure 9.27: Dining Philosophers: CP-net

2. Boundedness Properties: Relate to the upper or lower bound on the contents of place X in the net, over all possible markings. This can be the integer cardinality of the multiset at node X, or the greatest or least multiset itself.

3. Home Properties: This relates to the marking or set of markings which are reachable from all other markings in the OG. One can verify that a marking or set of markings constitutes a homespace, or determine whether or not a home marking exits. It is also possible list all home markings in G and in G_S, to find the minimal home space, and to determine if the initial marking is also a home marking.

4. Liveness Properties: A marking from which no further markings can be derived is 'dead'. Liveness, then, relates to the possible progressions from a given node in the OG. One can verify that a marking is dead, or list dead markings in G or G_S.

5. Fairness Properties: Relates to the degree to which certain Transition Instances (TI) will be allowed with respect to other TIs.



TBA

Figure 9.28: Dining Philosophers: Occurrence Graph

Many of these properties have different value depending on whether we are regarding a CP or an MAS, and also on the complexity of the net. CPs describe/operate on a message stream, which in most cases is finite; CPs themselves are static. One can imagine analyzing a CP in the context of (1) a single message stream, or (2) in the presence of a generator for all or many representative streams. In that sense, we might be interested in boundedness or home properties, and possibly reachability or fairness, but not liveness. On the other hand, liveness and fairness will often be more important in the analysis of a system as a whole.

For example, consider a simple CP (e.g. **Register**). Given some sequence of messages, we might be interested in reachability; does this initial marking result in the correct behavior (e.g. a registration being placed)? If we were to construct a net which generated a broad range of messages or message sequences, we could combine this with our **Register** conversation, and analyze the behavior of that system. Home properties would be useful here; we could designate certain terminal markings (e.g. accept, reject) as members of a home space, and determine whether or not all test sequences resulted in one of the two acceptable markings.

In a MAS, we are concerned with more dynamic properties of the system, which is assumed to be engaged in some self-sustaining activity. Liveness tests will assure us that the system has not entered a state in which no further activity can occur. Fairness assures us that no elements of the system which are able to act will remain inactive indefinitely.

It is possible to verify properties even for very large and complex nets. The version of Design/CPN used in this research supports the computation and analysis of OGs of 20,000 - 200,000 nodes and 50,000 to 2,000,000 arcs. The CPs used in the PRA model are too simple to generate very meaningful analysis. The system as a whole, however, is likely to have interesting properties. Liveness is not one, since the MAS is designed to perform a task of finite duration. If we could identify a marking or markings which represented successful completion of the task, and verify them to be a homespace, we would have proven the model correct. There are a couple of approaches we could take:

1. Identify all acceptable terminal markings. In our case, this is not very practical. Correct behavior is defined by the presence of exactly one `tell` message in the Response place of the Monitor's **Subscribe** conversation. There is an enormous amount of irrelevant state, in address caches, counters, and so on.

2. Reduce the amount of irrelevant state. This could be done by 'shutting down' the agents, letting them purge caches and so forth. Then the previous approach could be applied more easily, since the number of terminal markings, or at least the complexity of the terminal marking, would be reduced. This is cleaner, but complicates the model with additional mechanisms, making analysis more difficult.

3. Apply marking equivalence. Design/CPN supports the use of Occurrence Graphs with Equivalence (OE) and Occurrence Graphs with Symmetry (OS) [Dep98]. Using these, it is possible to define equivalences/symmetries (in CPN ML) in markings, vastly reducing the graph size. We could define the homespace with respect to the marking of the one significant place, and reduce all acceptable markings to one or several through equivalence.

The choice of approach depends on the individual model, of course.

9.5 Summary

Through building a CPN-based multi-agent system, we have demonstrated the real value of CPNs in a very tangible way. They can be used to model systems at varying levels of abstraction (e.g. CPs and MASs), they have an intuitive visual representation which improves readability, and they are expressive enough to model a wide variety of computations, including logical and numeric computations and search operations. More importantly, we have shown that modular specifications, such as the CPs here for KQML and FIPA, can be built, shared and embedded in larger models. This supports the use of CPNs as a model for specifying agent conversations.

Chapter 10

Protolingua

While CPNs provide an excellent framework for modeling conversations, there are still a couple of issues that need to be addressed in order to allow their effective use in the field. The first is a notational issue, relating to the ability to easily compartmentalize and compose models. The second relates to CPNs close alliance with ML, perhaps not the preferred specification language for all developers. We will discuss each of these points briefly, and then propose a solution; a language independent specification framework based on the CPN model, called Protolingua.

Recall the HCPN specification (Section 8.2). While this provides a useful means of describing large nets in a simplified, compartmentalized way, it does not allow for the recursive definitions of conversations. Thus, the definition for a complex conversation or net contains all the details of construction from smaller nets on up. This is a significant problem with respect to the arbitrary composition of conversations into new and larger conversations. For this reason, we will define an Recursive Colored Petri Net (RCPN). The RCPN is an extension to CPNs which allows the expression of HCPNs in a recursive, modular manner; an RCPN has a corresponding HCPN representation. The set of RCPNs

Model	Description Language	Motivation
PN		
CPN	CPN-ML	More expressive; formally equivalent to PN.
HCPN		
RCPN	*none*	Allows for the encapsulated expression of individual elements of an HCPN; describes the same set of models.
RCPN'	Protolingua	Extends RCPN by providing for explicit decomposition and labeling of multiset elements, and of arc expressions which act on these sub-elements.

Table 10.1: Models

is equivalent to that of HCPNs.

Table 10.1 describes the steps we will take at deriving Protolingua from the CPN model.

We extend the CPN to RCPN directly by adding the ability to fuse other RCPNs into the net, as is done with HCPNs. We also add fusion sets, defined recursively in terms of the fusion sets of sub-RCPNs.

For simplicity, we will observe that every CPN has an immediate HCPN representation, and restrict discussion to HCPNs. An RCPN is then an HCPN, along with possible sub-RCPNs and the requisite binding features. Intuitively, we say that an RCPN is an HCPN, and we define RCPNs as sets of other RCPNs. As such:

18 DEFINITION (RECURSIVE CP-NET) *A tuple* $RCPN = (\Sigma, P, T, A, N, C, G, E, I, SA, PA, FS, F$ *satisfying the requirements below:*

1. P, T, A, N, C, G, E, I *as with CPN.*
2. SA *is a* **substitution assignment** *function. It is defined from* T *into the set of all named conversations, such that no conversation is a conversation of itself.*
3. PA *is a* **port assignment** *function. It is defined from* P *into* P_i*, nodes in sub-conversations.*
4. FS *is defined as in HCPN as a set of fusion sets, where fusion sets consist of places (local) and other fusion sets.*

5. *FT is a* **fusion type** *function. It is defined as in HCPNs.*

We will use a notation which describes RCPNs to describe conversations.

10.1 Ontology

Ontologies for both the problem domain (discourse) and the model are required in order to define Protolingua. We will treat discourse first.

The discourse domain will require entities related primarily to message composition (e.g. message, sender, receiver) and message sequencing (e.g. conversation, transaction, thread).

In order to model discourse, we will need to identify the objects in the modeling domain. This will consist primarily of analogues to CPN components, including: place, action, port, function, tuple, multiset, schema, and arc.

10.2 Protolingua Interface Definition Language

In order to associate actions with protocols, an Interface Definition Language (IDL) is required. For this purpose, we define Protolingua Interface Definition Language (PIDL). The decision to write a lightweight IDL, rather than use an existing one, was primarily motivated by complexity. CORBA IDL itself [Gro98], the strongest candidate because of its broad acceptance, is not suitable for our task for two reasons: First, CORBA IDL is somewhat more complex than is necessary for Protolingua. Second, CORBA IDL does not permit sufficient freedom of abstract type manipulation. We will discuss both of these points in more detail. CORBA is geared toward the interaction of distributed objects, and incurs overhead in the form of brokers and additional name services. Conversations of

the form we propose will, once loaded, may be used many times, and must be very light. An interface definition in this scheme is a means of declaring a function, a signature, for which executable code can be downloaded once and then used and reused. This requires that for each function, code for many different platforms is available, but involves much less execution overhead. The IDL required to specify objects in this scheme can be made simpler than in CORBA.

Because we would like to work with native types, for efficient operation, we extend IDL to contain arbitrary named types. For example, a Message type to denote message objects. This will allow functions to exchange and operate on the high level, complex objects already in play in the host system, without the overhead of type inter-conversion. Note, however, that because Protolingua should run on any platform which supports an interpreter, we must restrict type construction in such a way that types can be represented on various platforms. For example, while it would be desirable to include object types for message in Java, conversations employing these types would be difficult to support under Tcl. However, if equivalent representations exist, there is nothing problematic about the use of a Message type which has an object representation on some platforms, and a string-based representation on others.

Thus, we will define PIDL, a description language encompassing functions and function implementations, as described by type signatures and descriptions, where the types and descriptions are names from a shared ontology. These signatures will be used to talk about, locate and obtain functions. For an initial definition, PIDL will not accommodate inheritance or type polymorphism. As an example, Figure 10.1 shows the PIDL description of a function which returns the Domain portion of a FQAN:

It is conceivable that a combination of PIDL and CORBA could be used - acquiring local methods when they are available in the required form, and accessing them remotely

```
(PIDL-function-description
  (name domain_from_name)
  (description ``Returns Domain portion of a \ac{KNS} \ac{FQAN}.'')
  (return kns_fqan)
  (parameters kns_fqan))
```

Figure 10.1: PIDL description of a function to return the Domain portion of a FQAN.

when they are not. For this project, we consider only the former.

Note that there are two issues involved in the level of sophistication of the modeling language. The first is one of efficiency. Supporting object types allows for greater efficiency if, for example, message objects must be reduced to string form in order to be processed. The second is one of extension of the communication language. Some systems support internally and even exchange messages which are more complex than specified by the ACL, and rely on features and methods specific to these representations. While no one language could cover the nuances of every system, extensions to Protolingua could be created to accommodate the features of various ACLs and platforms. We leave the issue for future work.

Formally, PIDL is a KIF-like language with two top-level objects, represented by the predicates *PIDL-function-description* and *PIDL-implementation-description*

$$
\begin{aligned}
&(PIDL-function-description \\
&\quad (name < function\_name >) \\
&\quad (description < descriptive\_text >) \\
&\quad (returns < type >) \\
&\quad (parameters < type >_1 \ldots < type >_n))
\end{aligned}
\tag{10.1}
$$

$$(PIDL - implementation - description \quad (10.2)$$
$$(name < module\_name >)$$
$$(platform < platform\_descriptor >)$$
$$(phase < phase\_name >)$$
$$(implements < function_name >))$$

All slot filling elements (see Tables 10.2, 10.5, 10.3, 10.4), except for descriptive text, are found in the PIDL ontology, and are available to all agents.

Type	Description
Message	Standard unit of communication; includes addressing information
Agent-Name	KNS standard Fully Qualified Agent Name

Table 10.2: PIDL Types.

Platform	Description
jvm2	Sun Java Virtual Machine Version 2
x86	Intel x86 Processor

Table 10.3: PIDL Platforms.

Phase	Description
Bytecode	Compiled bytecode for interpretation
Source	Uncompiled source code

Table 10.4: PIDL Phases.

10.3 Protolingua

Having laid the groundwork, we are now ready to define the language itself. Protolingua is a description language for CPNs which model conversations. It supports the description

Function	Description
domain_from_name	Return the domain portion of a KNS FQAN

Table 10.5: PIDL Functions.

of hierarchical CPNs as described in [Jen97b]. Protolingua is a KIF-based description language which completely specifies CPNs using a standard ontology of functions and types. This language effectively describes the structure of a CPN and names the associated conditions and actions in an interface definition language PIDL. In specifying Protolingua, we have opted to prefer clarity to brevity, so the notation is very explicit.

There are good reasons for and against including many standard features in Protolingua (e.g. types, variables, expressions). We have opted to take the approach of extreme simplicity in order to optimize interaction with existing datastructures. The use of Protolingua types mirroring the standard CPN color sets, for examples, would simplify use somewhat, but would require that objects being handled by the CP interpreter either conform to or translate to special types; for many applications, this presents an unacceptable bottleneck in the message stream. In our 'wire-frame' approach, objects are handled at all times in their native form, and with access methods provided by the agent/application. We will introduce variables as a notational convenience.

CPNs are typically described by three components: the net structure, the (function and colorset) declarations, and the net inscriptions, which describe assignments and initialization of the components of the net. Since we rely on named sets of known types and functions, Protolingua does not include a declaration. Net inscription is explicitly included in the description of the net structure.

We associate a *page* with a conversation, and so will only name conversations explicitly. Multiple instances of identical sub-conversations (substitution transitions) occupy indexed

subpages. That is, the 'query_77' conversation might contain three sub-instances of the 'ask' conversation, identified as ask(1), ask(2) and ask(3).

Formally, a conversation is defined as (Equation 10.4):

$$
\begin{aligned}
&(conversation \\
&\quad (name\ Name) \\
&\quad (places\ Place_1 \ldots Place_n) \\
&\quad (actions\ Action_1 \ldots Action_n) \\
&\quad (arcs\ Arc_1 \ldots Arc_n) \\
&\quad [(fusionSets\ FusionSet_1 \ldots FusionSet_n)])
\end{aligned}
\tag{10.3}
$$

Places bear the schema of their associated multiset:

$$
\begin{aligned}
&(place \\
&\quad (name\ Name) \\
&\quad (type\ Type))
\end{aligned}
\tag{10.4}
$$

where Type denotes the name of a known datatype.

An Action is essentially a stateless Place, or rather, a Place with a maximum Multiset cardinality of one. Because the rules of simulation state that all of the Arcs emanating from an Actions must fire immediately following an input, and there are no loop Arcs on an Action, the Action's multiset is empty except during execution.

$$
\begin{aligned}
&(action \\
&\quad (name\ Name) \\
&\quad (substitution\ Name))
\end{aligned}
\tag{10.5}
$$

(*substitution* (10.6)
(*name Name*)
(*bindings* $B_1 \ldots B_n$))

(*binding* (10.7)
(*insert Name*)
(*attach Name*)

The Arc applies transformations to elements in the input view, and binds them to the output view. The input view is defined by the set of inputs to the Arc Functions, and the output view by the corresponding functional mapping.

The intersection of an Arc with a Node uniquely identifies a Port. While Ports are not explicitly represented in the specification of a conversation, they serve as a focal point for local views of the Multiset with respect to individual Arcs. The concept will be important in defining the interface to the interpreter.

(*arc* (10.8)
(*name Name*)
(*arc Expression*)
(*attach NamePlace*)
(*insert NameAction*)

$$(function \quad (name\ Name) \quad (result\ Variable) \quad (parameters\ Variable_1 \ldots Variable_n)) \tag{10.9}$$

$$(variable \quad (name\ Name)) \tag{10.10}$$

$$(fusionSet \quad (name\ Name) \quad (places\ Place_1 \ldots Place_n) \quad (imports\ I_1 \ldots I_n)) \tag{10.11}$$

where:

$$(import \quad (conversation\ Name) \quad (fusionSet\ Name)) \tag{10.12}$$

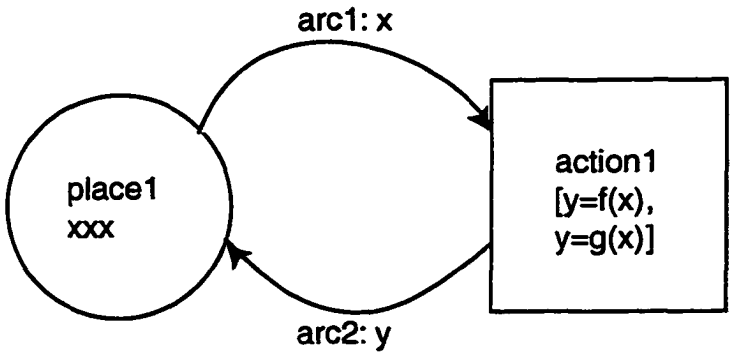
An example of Protolingua usage for a simple interaction is shown in Figure 10.2, and graphically in Figure 10.3 (extended examples are given later).

Conversation 'example' describes the Protolingua conversation depicted in Figure 10.3. Action 'action1' is enabled when the multiset at place1 contains a tuple t_i such that $f(t_{i,j})$ is defined. That tuple is removed by action1, and replace with the tuple $(c, g(f(t_{i,j})), c)$, where c is some constant. Note that the specification does not declare the multiset schema

```
(conversation
  (name example)
  (places
    (place (name place1) (type xxx)) )
  (actions
    (action (name action1)) )
  (arcs
    (arc (name arc1) (functions
      ( function (name f) (parameters A) (result B))))
    (arc (name arc2) (functions
      ( function (name g) (parameters C) (result D))) ) ) )
```

Figure 10.2: Protolingua description of the conversation 'example'.

for action1; it is implicit in the designations of the input arcs.



TBA

Figure 10.3: The conversation corresponding to the Protolingua description above.

Note that ports definitions are implicit in the definition of nodes and arcs. They are made explicit in this representation to provide for future work.

Protolingua only encodes the minimum amount of information required to specify the conversation. The rest will be derived by the interpreter from the specification and the function and type ontologies. For example, the types of the variables are implicitly determined by the type signatures of the functions in which they are used. Similarly, the interpreter

identifies port and socket nodes by their use in substitution assignments. The amount of type and structure checking done by the is up to the implementor. Details of name syntax and other minor technical issues are not addressed here.

10.4 Implementation

We will discuss only high level issues of implementation here. Individual systems can make use of stylized interpreters which conform to the standards defined.

One of the benefits of using CPNs for modeling conversations is that the CPN specification itself can be run on interpreters ranging from only a few lines to many thousand, with a difference only in efficiency of operation. This means that an agent using only simple conversations can do just as well with a fairly simple-minded interpreter, if code size is an issue. For the purposes of this research, we will be using an interpreter which is only sophisticated enough to effectively handle the conversations discussed.

An interface to the Protolingua interpreter should provide access at a minimum to the following information:

1. The cardinality of a named Multiset

2. The data element at position x of tuple y in a named Multiset.

10.5 Implementation in Jackal

For every platform that will use Protolingua, an interpreter (or interpreters) and local implementations of common types and functions must be created, along with the mappings to their abstract counterparts. In this section, we describe the initial integration of Protolingua

in Java for use with the Jackal System. The steps for implementation of Protolingua are fairly standard:

1. Construct an interpreter for the target language, if one does not exist. This interpreter must correctly render a CPN from a Protolingua description, and simulate it faithfully according to the model definition. It must also correctly implement the Jackal Protolingua Interpreter Interface (JPII).

2. Specify any desired conversations which are not currently available, and publish them as appropriate. This step is actually independent of a port to a target language, but is included for completeness.

3. In support of any (new or existing) conversations to be used, construct those necessary native types and functions which are not already available, and publish them as appropriate.

4. Construct/extend a mapping from PIDL to the target language, to include new types and functions introduced.

Tables and mappings in support of PIDL should reside with a known Protolingua server, which can be queried and updated as appropriate. Unpublished information can, however, be shared directly. For example, two agents engaging in a private conversation may decide to exchange local conversation specifications, private mappings or functions.

An agent which finds the need to engage in some unfamiliar conversation may query either the engaged agent or some common server (likely identified via a broker) for the conversation specification ("What is the form of this conversation?"). Through similar queries, the agent could find, based on the conversation specification, what local types and functions were required ("What module implements type/function X for my native

language?"). Finally, the agent could download the components from the engaged agent or a server, and begin using the conversation.

10.5.1 Example: Sketching a Java Protolingua Interpreter

As mentioned earlier, the form of the Protolingua interpreter, effectively a CPN simulator, depends largely on the sophistication of the simulation engine. As conversations grow, their simulation increases exponentially in complexity. Thus, for extended conversations, a fairly advanced mechanism is required for effective online use. However, for use with conversations that are bounded in complexity to a fairly low level, a relatively inefficient but very lightweight simulation engine may be used, making Protolingua accessible to virtually all agents.

Since the purpose of this research is the demonstration of the viability of Protolingua for the given task, we implement a minimally sophisticated simulation engine for Jackal, and will focus only on its interface to Jackal and the agent. For convenience, we will refer to this engine as the Jackal Protolingua Interpreter (JPI).

Interface

We construct a layered JPII. At the core, this interface provides access to the basic state of the CPN; that is:

1. The number of places and actions.
2. The names of places and actions, by iteration.
3. Multiset cardinality, by place name.
4. Multiset width, by place name.

5. Type designators for multiset columns, by place name, tuple and column.

6. Data elements in the multisets, by place name, tuple and column.

Of course, since conversations should be dynamic, we include methods used to construct and de-construct the conversation:

1. Add or remove place, action or arc.

2. Add, remove or reorder functions on an arc.

Note that changing colorset form is *not* permitted. All actions are permitted on conversations in play, but only while execution is paused.

We would also like to provide some control over running conversations:

1. Pause simulation.

2. Unpause simulation.

3. Insert tuple into named multiset, at a given position.

4. Remove a tuple from a multiset, from a given position.

Given JPII, we can construct, simulate, and modify conversations on the fly. Using the interface as defined above, we can implement more advance options, such as conversation merging, and message threading.

10.5.2 Java PIDL Mapping

For simplicity, we make use of the class implementation of native types in Java, so all PIDL types map directly to Java classes. Likewise, all functions map to Java static methods. We

present below a partial mapping of PIDL types (Table 10.6) and functions (Table 10.7) to Java; since classes can be user defined, it is not possible to represent a complete mapping. Only those entries required for scenarios in this work are included.

Type	Java Class
FQAN	J3.FQAN
Integer	java.lang.Integer
Message	J3.Message
String	java.lang.String

Table 10.6: PIDL to Java Mapping: Types.

Function	Java Static Method
domain_from_name	J3.FQAN.domain()

Table 10.7: PIDL to Java Mapping: Functions.

10.5.3 Advanced Topics

Given what we have constructed so far, it is possible to construct and reason about the form of conversations, and even to attach actions which are beyond scope of the reasoning process. But what if we desired that two unrelated conversations interact? For example, the actions of registration and unregistration, which take place within two separate conversations, involve actions which affect one another. This poses two distinct problems. First, how do we allow actions of the former (**Register**) to affect the behavior of the latter? It is not obvious how to generally provide for their interaction. Second, if we would like to reason about such interaction, how do we do this, since the actions take place outside the model? Both problems can be solved by drawing the interaction into the model. We consider two possible solutions.

We have assumed to this point that the current state known to a conversation is the state of the corresponding CPN. Changes in state are madc explicit by introduction of data (e.g. messages) into the model. If the interface was extended to allow access to a common store, such a store could be used to keep such common information as agent registries. The ramifications of such an extension on the model, however, are unclear. Thus, this solution would make it possible to conduct registrations and unregistrations entirely through conversation actions, but might not allow reasoning about series of such interactions.

Another solution is to combine the conversations. Instead of a common external store, we make the registration and unregistration action part of the same larger conversation. Registrations can then be stored easily as part of the internal state of the conversation, and reasoning is unaffected. The problem with this approach is that conversations must be bound together with other related components. This makes it difficult to implement almost any, small, independent conversations.

By introducing a common tuple space, we can provide for the interaction of conversations that are not directly or hierarchically connected. This tuple space has the form of a CPN place, and is common, though only implicitly represented, in every conversation. Its behavior is the same as any ordinary place, and it is managed by the JPII mechanisms. This is left for future work. Without this addition, it is still possible to use and reason about small conversations in isolation, and to reason about composite conversations by binding their extra-model interactions in some way similar to what was described above.

Chapter 11

Results and Conclusion

Over the course of three years, Jackal has changed dramatically, and has become a valuable enough tool to win the attention and respect of the agents community. Somewhat more limited in scope than many agent development packages currently available, it does well the task for which it was designed; facilitating communication among distributed agents.

11.1 Jackal

The decision to focus on communication and integration led Jackal down a different path than many other similar systems. Design for easy integration meant that Jackal had to be a 'tool' used by the agent, not a shell or environment. As a result, it does not address issues relating to the construction or functioning of the agent itself; only communication and related issues. This is in contrast to more comprehensive systems which provide support for persistence, mobility, resource control, and so forth. Often this is in the form of libraries and an AEE (especially in the case of mobility). Jackal-based agents do not require a specialized AEE; only the standard JVM.

The introduction of the Distributor, which occurred very early on in the project, was motivated by a desire to separate the conversation specifications from the associated actions. Early versions (Lewak and Jackal 2.x) allowed actions which were bound directly to the arcs of the conversation DFAs. This required the agent programmer to interact directly with the conversation management code, and made it more difficult to maintain 'like' conversations across systems of agents. The distributor provides a flexible interface to the messages, allowing the conversation mechanisms to be protected from the user. The cost, unfortunately, has been a restriction in the expressiveness of conversation templates. While attachments are still allowed, they are discouraged because of the potential conflicts they can present with the messaging system.

11.1.1 Future work

- **Security:** Security is an issue which has only been casually addressed in Jackal. The potential exists for multiple levels of security, and plans are underway to implement them. For example, the (socket-based) transport modules, designed to be customized, can easily be fitted with the code to implement SSL. At a higher level, the KNS framework provides an ideal framework for storing and retrieving common agent information. This could be used to facilitate the distribution of certificates, which could be used in combination with messages for secrecy or authentication.

- **Intelligence:** Many features of Jackal would benefit from the presence of a reasoning engine. Jackal components could be enhanced with rule-based systems, and such a framework could also be leveraged by the agent developer, who could use the engine to construct rule-based Jackal agent programs. While this idea seems simple, some practical constraints need to be resolved. Traditionally, Jackal has consisted entirely

of in-house code, in order to maximize control over development. Writing a Java-based reasoning engine broadens the scope of the project considerably. There are few Java-based rule engines, and most have significant limitations. Currently, we are experimenting with the use of an external Prolog engine.

- **Concurrency:** While Jackal has progressed through a number of changes, its underlying architecture still follows the same design given in its initial rewrite from Lewak. There are a number of bottlenecks in the message path, and a few components that should be rewritten in order to support higher internal bandwidth.

- **Language Independence:** Since the beginning of this project, we have witnessed the birth of an alternative to KQML, the FIPA ACL. It will be important in the future for packages which facilitate communication to support a range of possible ACLs. While Jackal was written to handle KQML communications, the internal message path is largely language neutral, and should be adaptable to multilanguage use. There are three issues in handling multiple languages: (1) The message format of each language must map, at least partially, to some generic message format. Messages can then either be translated, or preferably, wrapped. (2) Conversation specifications must exist for each language, and ideally should be somewhat compatible. In the strong sense, this would mean that a single conversation could involve messages of different languages. In a weaker sense, data associated with the conversations should be compatible. (3) Management issues must be addressed; this will be the most difficult task. Each language typically specifies a support infrastructure (e.g. KNS), and Jackal will have to be facile with the infrastructure for each language it supports.

11.2 Conversations

It is clear that finding the right model for CPs is a challenging issue. Most likely, the best solution will be a mix of formalisms (and in-formalisms) tailored to specific needs.

CPNs appear to be a promising model for specifying conversations; they have certainly served well as models for agents and agents systems for some time. The benefits of using CPNs include concise, verifiable, executable specifications.[1] They are, however, not without practical limitations. CPN inscriptions are given in ML, and at least a rudimentary knowledge of functional programming is required for most models. While the amount of knowledge required is not great, there is a startup cost which makes CPNs less accessible than DFAs, for instance.

Note that many of the models used for specifying protocols loose their power because of their limited expressiveness; modelers attempting to specify complex patterns of interaction find it necessary to extend the model in various ways. The expressiveness of CPNs should allow developers to specify interactions without resorting to degrading the model.

11.2.1 Protolingua

Protolingua endeavors to be an inscription-language-independent representation of CPNs. By keeping the language simple, we minimize the difficulty of porting interpreters to a wide variety of languages/platforms. This is done at the expense of requiring the existence of code attachments in each of these languages.

The successful development of Protolingua, or an equivalent, will be important in determining the value of CPNs to this endeavor, Without such a language, their will be a

[1] By executable specification, we mean a representation which can be be run directly in an interpreter, such as Design/CPN, and which has a straightforward, if not direct, mapping into a form which can be used to drive real systems.

gap between the conversation specifications themselves and the representations that can be manipulated and used by agents.

There are alternative approaches. One would be the use of CPN-ML and the tuple representation directly. This does not address the problem completely, since Protolingua is a representation language for RCPN, not HCPN. However, RCPN has a corresponding tuple representation. Even so, this would require that every agent platform be able to execute ML code, which is a strong requirement. Another approach would be significantly restricting CPN-ML. This would limit the expression of conversations, but would allow for a much lighter implementation. A possible compromise is the addition of some more sophisticated linguistic features to Protolingua, easing the reliance on code attachments.

11.3 Conclusion

While the term 'agent' may mean vastly different things to different people, most would agree that some common framework supporting agent interaction and communication is essential. The goal of agent interoperability is defeated if agents are only able to interact within their own specialized communities. We have developed Jackal, a package which will attempt to meet this need by providing agent designers with a flexible, portable agent communication infrastructure, support for abstractly specified agent behavior.

Jackal is a streamlined harness around the conversation mechanisms, to which capabilities may be added dynamically, and which supports an advanced agent interface. Jackal provides extensive support for interagent communication, but is independent of the features of any one ACL, so various languages may be used, even concurrently. Jackal facilitates the abstraction of agent behaviors by implementing a Protolingua interpreter. Agent behavior are declaratively specified, freely exchanged or manipulated, and run in the interpreter.

The CP mechanisms allows agent behaviors to be specified in a language which is independent of the agent development platform. Thus, agents specified with Protolingua can be ported to any platform for which there is an interpreter, and simple behavior can easily be exchanged among agents as needed. Additionally, as Protolingua has a basis in a well-known formal model of computation, behaviors are amenable to formal analysis. This model supports agents in a wide range of complexity. Simple, reactive agents may have a fixed set of protocols, or may chose to load and exchange them on demand. More sophisticated agents may reason about what protocols might be best to employ for a given situation. High-level agents with planning capabilities could translate goals into plans, and decompose them into units which could be retrieved from a library or created on the fly.

KNS provides a set of protocols for advanced address resolution and agent information tracking. Communication is a central problem in distributed systems. Without the ability to communicate, a set of agents would be a merely a collection of isolated systems. All classic distributed systems problems, such as coordination and distributed reasoning, depend on an underlying communication framework Jackal is designed to support multiple concurrent agent communication languages, but implements the KQML agent communication language as the default. Still, before communication can take place, there must be a known destination. KNS, like the KQML ANS scheme [FPT+95], adds a communication layer in which symbolic names are mapped to actual Internet addresses. In addition, however, it offers more advanced support for dynamic group formation and disbanding, and a more abstract notion of agent identity, independent of any particular name or globally unique identifier.

Jackal differentiates itself from other existing agent development platforms in its extensive support for high level agent communication, its powerful abstract protocol specification, and its internal message distribution architecture. While many other systems have

focused on agent mobility, they have often neglected remote communication or omitted it entirely, intending agents to collocate and communicate directly. Jackal is committed to supporting inter-agent communication at a universal, symbolic level, and as a basis for all other interaction. This aspect makes Jackal-based MAS more flexible, and also enhances the likelihood of compatibility between Jackal and other systems.

Many systems, including Jackal, have made use of conversation models which, although very expressive, have not lent themselves to structured analysis and design. This is the shortcoming that we hope to address with the use of CPNs. CPNs are more sophisticated than many models currently employed (e.g. automata), allowing for greater expression within the bounds of the model, but are also relatively simple, and have an intuitive graphical representation. Models can then be constructed which are amenable to analysis, exchange, and immediate simulation or execution. This last point is facilitated by a cross-platform modeling language; Protolingua. Our initial view of Protolingua, that of a wire-frame structure for binding well-known native methods and types, is only one of many possible approach, which emphasizes portability over ease of expression.

Jackal is well suited to enhancing interoperability among agents systems, a necessary step in developing a global community of software agents which can be truly useful to their creators and to each other.

Bibliography

[Age79] Tilak Agerwala. Putting Petri Nets to work. *Computer*, pages 85–94, December 1979.

[AL98] Yariv Aridor and Danny B. Lange. Agent design patterns: Elements of agent application design. In *Proceedings of the Second International Conference on Autonomous Agents (Agents '98)*, Minneapolis, May 1998. ACM Press.

[Bar93] Mihai Barbuceanu. Models: Toward integrated knowledge modeling environments. *Knowledge Acquisition*, 5:245–304, 1993.

[Bar96] Mihai Barbuceanu. COOL: A language for representing and executing coordinated behavior in multi-agent systems, April 1996.

[Bar99] Mihai Barbuceanu. The agent building shell: Programming cooperative enterprise agents. http://www.ie.utoronto.ca/EIL/ABS-page/ABS-overview.html, 1999.

[BBC+97] R. J. Bayardo, Jr., W. Bohrer, A. Cichocki, J. Fowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. InfoSleuth: Agent-based semantic integration of information in open and dynamic systems. In *Proceedings of (SigMod 97)*, 1997.

[BDBW98] Jeffrey M. Bradshaw, Stuart Dutfield, Pete Benoit, and John D. Woolley. KAoS: Toward an industrial-strength open agent architecture. In Jeffrey M. Bradshaw, editor, *Software Agents*. AAAI/MIT Press, 1998.

[BE98] Jon Barwise and John Etchemendy. Computers, visualization, and the nature of reasoning. In James Moor, editor, *The Impact of Computers in Philosophy (Running Title)*. Blackwell Publishers, 1998.

[Ber96] J. Bermudez. Advanced planning and scheduling systems: Just a fad or a breakthrough in manufacturing and supply chain management? Technical report, Advanced Manufacturing Research, Boston, Massachusetts, December 1996.

[BF94a] M. Barbuceanu and M. S. Fox. The information agent: An infrastructure agent supporting collaborative enterprise architectures. In *Proceedings of the Third Workshop on Enabling Technologies, Infrastructure for Collaborative Enterprises*, Morgantown, WV, June 1994. IEEE Computer Society Press.

[BF94b] Mihai Barbuceanu and Mark S. Fox. The information agent: An infrastructure for collaboration in the integrated enterprise. In M. Deen, editor, *International Conference on Collaborative Knowledge Based Systems (CKBS '94)*, DAKE Centre, University of Keele, UK, June 1994.

[BF95] Mihai Barbuceanu and Mark S. Fox. COOL: A language for describing coordination in multiagent systems. In Victor Lesser, editor, *Proceedings of the First International Conference on Multi–Agent Systems*, pages 17–25, San Francisco, CA, 1995. MIT Press.

[BF96a] Mihai Barbuceanu and Mark S. Fox. The architecture of an agent building shell. In Michael Wooldridge, Jörg P. Müller, and Milind Tambe, editors, *Proceedings on the IJCAI Workshop on Intelligent Agents II : Agent Theories, Architectures, and Languages*, volume 1037 of *Lecture Notes in Artificial Intelligence*, pages 235–250. Springer-Verlag, Berlin, 19–20 1996.

[BF96b] Mihai Barbuceanu and Mark S. Fox. Capturing and modeling coordination knowledge for multi-agent systems. *International Journal on Cooperative Information Systems*, 5(2 & 3):275–314, 1996.

[BF96c] Mihai Barbuceanu and Mark S. Fox. Coordinating multiple agents in the supply chain. In *Proceedings of the Fifth Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 134–142. IEEE Computer Society Press, 1996.

[BF96d] Mihai Barbuceanu and Mark S. Fox. Integrating communicative action, conversations and decision theory in a coordination language for multi–agent systems, 1996.

[BF97a] M. Barbuceanu and M. S. Fox. The design of a coordination language for multi-agent systems. In J. P. Muller, M. J. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III: Agent Theories, Architectures and Languages, Lecture Notes in Artificial Intelligence 1193*, pages 341–357. Springer-Verlag, 1997.

[BF97b] Mihai Barbuceanu and Mark S. Fox. Integrating communicative action, conversations and decision theory to coordinate agents. In *Proceedings of the First International Conference on Autonomous Agents (Agents '97)*, pages 49–58, Marina del Rey, February 1997. ACM Press.

[BFD98] J. Billington, M. Farrington, and B. B. Du. Modelling and analysis of multi-agent communication protocols using CP-nets. In *Proceedings of the third Biennial Engineering Mathematics and Applications Conference (EMAC'98)*, pages 119–122, Adelaide, Australia, July 1998.

[BFG+95] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. MetateM: An imperative approach to temporal logic programming. *Formal Aspects of Computing*, 7(E):111–154, 1995.

[BFSH+98] S. Boussetta, A. El Fallah-Seghrouchni, S. Haddad, P. Moraïtis, and M. Taghelit. Coordination d'agents rationnels par planification distributée. *Revue d'Intelligence Artificielle*, January 1998.

[BGH+98] J. M. Bradshaw, M. Greaves, H. Holmback, R. Carpenter, R. Cranfill, R. Jeffers, L. Poblete, T. Robinson, A. Sun, Y. Gawdiak, A. Canas, N. Suri, B. Silverman, M. Brooks, A. Wong, I. Bichindaritz, and K. Sullivan. Mediating representations for an agent design toolkit. In *Proceedings of the Eleventh Annual Knowledge Acquisition Workshop*, volume I, pages 1–21, Banff, Canada, April 1998.

[BGR95] E. Börger, Y. Gurevich, and D. Rosenzweig. The Bakery Algorithm: Yet Another Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.

[BH98] E. Börger and J. Huggins. Abstract State Machines 1988-1998: Commented ASM Bibliography. *Bulletin of EATCS*, 64:105–127, February 1998.

[BHR97] Joachim Baumann, Fritz Hohl, and Kurt Rothermel. Mole - concepts of a mobile agent system. Technical Report TR-1997-15, Universistät Stuttgart, Fakultät Informatik, Germany, August 1997.

[BHRS97] J. Baumann, F. Hohl, K. Rothermel, and M. Straßer. Communication concepts for mobile agent systems. *Lecture Notes in Computer Science*, 1219, 1997. 123–??

[BKJT95] Frances Brazier, Barbara Dunin Keplicz, Nick R. Jennings, and Jan Treur. Formal specification of multi-agent systems: a real-world case. In V. Lesser, editor, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS '95)*, pages 25–32. MIT Press, 1995.

[BKR97] Jonathan Bredin, David Kotz, and Daniela Rus. Market-based Resource Control for Mobile Agents. Technical Report PCS-TR97-326, Dartmouth College, Computer Science, Hanover, NH, November 1997.

[Bör95] E. Börger. Why Use Evolving Algebras for Hardware and Software Engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *Proceedings of SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *Lecture Notes in Computer Science*, pages 236–271. Springer, 1995.

[Bra96] Jeffrey M. Bradshaw. KAoS: An open agent architecture supporting reuse, interoperability, and extensibility. In *Tenth Knowledge Acquisition for Knowledge-Based Systems Workshop*, 1996.

[Bra97] Jeffrey M. Bradshaw. An introduction to software agents. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 3–46. AAAI/MIT Press, 1997.

[BS98] E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, Lecture Notes in Computer Science. Springer, 1998.

[BTF97] M. Barbuceanu, R. Teigen, and M. S. Fox. Agent based design and simulation of supply chain systems. *IEEE Computer Society Press*, 1997.

[CCF+99] R. Scott Cost, Ye Chen, Tim Finin, Yannis Labrou, and Yun Peng. Modeling agent conversations with colored petri nets. In *Working Notes of the Workshop on Specifying and Implementing Conversation Policies*, pages 59–66, Seattle, Washington, May 1999.

[CCWB94] Philip R. Cohen, Adam J. Cheyer, Michelle Wang, and Soon Choel Baeg. An open agent architecture. In *AAAI Spring Symposium*, pages 1–8, March 1994.

[CFL+98] R. Scott Cost, Tim Finin, Yannis Labrou, Xiaocheng Luan, Yun Peng, Ian Soboroff, James Mayfield, and Akram Boughannam. Jackal: A Java-based tool for agent development. In Jeremy Baxter and Chairs Brian Logan, editors, *Working Notes of the Workshop on Tools for Developing Agents, AAAI '98*, number WS-98-10 in AAAI Technical Reports, pages 73–82, Minneapolis, Minnesota, July 1998. AAAI, AAAI Press.

[CG89] N. Carriero and D. Gelertner. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[CH92] Søren Christensen and Niels Damgaard Hansen. Coloured petri nets extended with place capacities, test arcs and inhibitor arcs. Technical Report DAIMI PB-398, Computer Science Department, Aarhus University, Aarhus C, Denmark, May 1992.

[Cha97] Deepika Chauhan. JAFMAS: A Java-based agent framework for multiagent systems development and implementation. Master's thesis, ECECS Department, University of Cincinnati, 1997.

[CJ95] Adam Cheyer and Luc Julia. Multimodal maps: An agent-based approach. In *Proc. of the International Conference on Cooperative Multimodal Communication (CMC/95)*, Eindhoven, The Netherlands, May 1995.

[CL90] Philip R. Cohen and Hector J. Levesque. Rational interaction as a basis for communication. In P. R. Cohen, J. Morgan, and M. E. Pollack, editors, *Intentions in Communication*, SDF Benchmark Series, pages 221–255. MIT Press, 1990.

[CLM+98] B. Chu, J. Long, M. Matthews, J. Barnes, J. Sims, M. Hamilton, and R. Lambert. FAIME: An object-oriented methodology for application plug-and-play. *Jounal of Object Oriented Programming*, 1998.

[Com96] Compositional Research Group. Caltech Infospheres Project. http://www.infospheres.caltech.edu, 1996.

[CSL+97] R. Scott Cost, Ian Soboroff, Jeegar Lakhani, Tim Finin, Ethan Miller, and Charles Nicholas. TKQML: A scripting tool for building agents. In Michael Wooldridge, Munindar Singh, and Anand Rao, editors, *Intelligent Agents Volume IV – Proceedings of the 1997 Workshop on Agent Theories, Architectures and Languages*, volume 1365 of *Lecture Notes in Artificial Intelligence*, pages 336–340. Springer-Verlag, Berlin, 1997.

[CTW+96] B. Chu, W. J. Tolone, R. Wilhelm, M. Hegedus, J. Fesko, T. Finin. Y. Peng, C. Jones, J. Long, M. Matthes, J. Mayfield, J. Shimp, and S. Su. Integrating manufacturing softwares for intelligent planning-execution: A CIIMPLEX perspective. In *Plug and Play Software for Agile Manufacturing, SPIE International Symposium of Intelligent Systems and Advanced Manufacturing*, Boston, MA, 1996.

[DB92] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the ACM 1992 Conference on Computer-Supported Cooperative Work: Sharing Perspectives (CSCW '92)*, pages 107–114, Toronto, November 1992.

[Dec95] Keith S. Decker. *Environment Centered Analysis and Design of Coordination Mechanisms*. PhD thesis, University of Massachusetts, Amherst, 1995.

[Dec97] Keith S. Decker. Task environment centered simulation. In M.L Prietula, K. Carley, and L. Gasser, editors, *Simulating Organizations: Computational Models of Institutions and Groups*. AAAI Press/MIT Press, 1997.

[Dep96] Department of Computer Science, University of Aarhus, Denmark. *Design/CPN Occurrence Graph Manual*, version 3.0 edition, 1996.

[Dep98] Department of Computer Science, University of Aarhus, Denmark. *Design/CPN OE/OS Graph Manual*, version 1.1 edition, 1998.

[Dic97] Ian Dickenson. Agent standards. Technical report, Foundation for Intelligent Physical Agents, October 1997.

[DL94] K. Decker and V. Lesser. Designing a family of coordination algorithms. In *Proceedings of the First International Conference on Multi-Agent Systems*, 1994.

[DOSW] Jack J. Dongarra, Steve W. Otto, Marc Snir, and David Walker. An introduction to the MPI standard.

[DPSW97] Keith Decker, Anandeep Pannu, Katia Sycara, and Mike Williamson. Designing behaviors for information agents. In *Proceedings of the First International Conference on Autonomous Agents*, Marina del Rey, February 1997.

[DS97] Keith Decker and Katia Sycara. Intelligent adaptive information agents. *Journal of Intelligent Information Systems*, 1997.

[DSW97] Keith Decker, Katia Sycara, and Mike Williamson. Middle-agents for the Internet. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, Nagoya, Japan, August 1997.

[DWJ95] Tzvetan T. Drashansky, Sanjiva Weerawarana, and Anupam Joshi. Software architecture of ubiquitous scientific computing environment for mobile platforms. Technical Report CSD-TR-95-032, Department of Computer Sciences, Purdue University, 1995.

[EH99] Renée Elio and Afsaneh Haddadi. On abstract task models and conversation policies. In *Working Notes of the Workshop on Specifying and Implementing Conversation Policies*, pages 89–98, Seattle, Washington, May 1999.

[FBG95] Mark S. Fox, Mihai Barbuceanu, and Michael Gruninger. An organisation ontology for enterprise modelling: Preliminary concepts for linking structure and behavior. *First International Conference on Multi-Agent Systems*, pages 17–25, June 1995.

[Fer96] Jaques Ferber. *Les Système Multi-Agents*. InterEditions, 1996.

[FFR96] Adam Farquhar, Richard Fikes, and James Rice. The Ontolingua server: A tool for collaborative ontology construction. In *KAW96*, November 1996.

[FIP97] FIPA. FIPA 97 specification part 2: Agent communication language. Technical report, FIPA - Foundation for Intelligent Physical Agents, October 1997.

[Fis94] Michael Fisher. A survey of concurrent METATEM - the language and its applications. In *Proceedings of the First International Conference on Temporal Logic (ICTL '94)*, volume 827 of *Lecture Notes in Computer Science*, Bonn, Germany, July 1994. Springer-Verlag.

[FLM97] Tim Finin, Yannis Labrou, and James Mayfield. KQML as an agent communication language. In Jeff Bradshaw, editor, *Software Agents*. MIT Press, 1997.

[FO95] Michael Fisher and Richard Owens. An introduction to executable modal and temporal logics. In Michael Fisher and Richard Owens, editors, *Executable Modal and Temporal Logics*, volume 897 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, 1995. Springer-Verlag.

[For95] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1995.

[FPT+95] Tim Finin, Anupama Potluri, Chelliah Thirunavukkarasu, Don McKay, and Robin McEntire. On agent domains, agent names and proxy agents. In Tim Finin and James Mayfield, editors, *Proceedings of the CIKM '95 Workshop on Intelligent Information Agents*, Baltimore, Maryland, 1995.

[Fro99] H. Robert Frost. Java Agent Template. Online Documentation: http://cdr.stanford.edu/ABE/JavaAgent.html, 1999.

[FSHM99] A. El Fallah-Seghrouchni, S. Haddad, and H. Mazouzi. A formal study of interactions in multi-agent systems. In *Proceedins of ISCA International Conference in Computer and their Applications (CATA '99)*, April 1999.

[FSM98] A. El Fallah-Seghrouchni and S. Haddad H. Mazouzi. Etude des interactions basée sur l'observation reépartie dans un systéme multi-agents. In Hermés, editor, *Proceedings of JFIADSMA '98*, Nancy, France, November 1998.

[FSM99] Amal El Fallah-Seghrouchni and Hamza Mazouzi. A hierarchial model for interactions in multi-agent systems. In *Working Notes of the Workshop on Agent Communication Languages, IJCAI '99*, August 1999.

[GB99a] Alan Galan and Albert Baker. Multi-agent communications in JAFMAS. In *Working Notes of the Workshop on Specifying and Implementing Conversation Policies*, pages 67–70, Seattle, Washington, May 1999.

[GB99b] Mark Greaves and Jeff Bradshaw, editors. *Working Notes of the Workshop on Specifying and Implementing Conversation Policies, Third International Conference on Autonomous Agents (Agents '99)*, Seattle, Washington, May 1999.

[GCG94] Nicola Guarino, Massimiliano Carrara, and Pierdaniele Giaretta. An ontology of meta-level categories. In Pietro Torasso Jon Doyle, Erik Sandewall, editor, *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning*, pages 270–280, Bonn, FRG, 1994. Morgan Kaufmann.

[Gen98] General Magic. *Introduction to the Odyssey API*, 1998.

[GHB98] Mark Greaves, Heather Holmback, and Jeff Bradshaw. CDT: A tool for agent conversation design. In *Working Notes of the AAAI-98 Workshop on Software Tools for Developing Agents*. AAAI, 1998.

[GHN+97] Shaw Green, Leon Hurst, Benda Nangle, Padraig Cunningham, Fergal Somers, and Richard Evans. Software agents: A review. Technical report, Intelligent Agents Group (IAG) of the Computer Science Department, Trinity College Dublin, 1997.

[GKCR97] Robert Gray, David Kotz, George Cybenko, and Daniela Rus. Agent Tcl. In William Cockayne and Michael Zyda, editors, *Itinerant Agents: Explanations and Examples with CD-ROM*. Manning Publishing, 1997.

[GKN+96] Robert Gray, David Kotz, Saurab Nog, Daniela Rus, and George Cybenko. Mobile agents for mobile computing. Technical Report PCS-TR96-285, Dartmouth College, Computer Science, Hanover, NH, May 1996.

[GL81] H. J. Genrich and K. Lautenbach. System modelling with high-level petri nets. *Theoretical Computer Science*, 13:109–136, 1981.

[Gla98] G. Glass. ObjectSpace Voyager — the agent ORB for Java. *Lecture Notes in Computer Science*, 1368, 1998.

[GR93] Paola Glavan and Dean Rosenzweig. Communicating evolving algebras. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Selected papers from CSL'92 (Computer Science Logic)*, volume 702 of *Lecture Notes in Computer Science*, pages 182–215. Springer-Verlag, 1993.

[Gra96] Robert S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In Mark Diekhans and Mark Roseman, editors, *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96)*. USENIX, July 1996.

[Gra98] Robert S. Gray. Agent Tcl: A flexible and secure mobile-agent system. Technical Report PCS-TR98-327, Dartmouth College, Computer Science, Hanover, NH, January 1998.

[Gro98] Object Management Group. CORBA/IIOP 2.2 specification, July 1998.

[Gru92] Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. In Nicola Guarino, editor, *Working Notes of the International Workshop on Formal Ontology*, Padova, Italy, 1992.

[Gru93] Thomas R. Gruber. A translation approach to portable ontology specifications. Technical Report KSL 92-71, Computer Science Department, Stanford University, California, 1993.

[GTW92] Thomas R. Gruber, Jay M. Tenenbaum, and Jay C. Weber. Toward a knowledge medium for collaborative product development. In J. S. Gero, editor, *Artificial Intelligence in Design '92*, pages 413–432. Kluwer Academic Publishers, Boston, 1992.

[Gua94] Nicola Guarino. The ontological level. In R. Casati, B. Smith, and G. White, editors, *Philosophy and the Cognitive Sciences*. Hölder-Pichler-Tempsky, Vienna, 1994.

[Gur93] Yuri Gurevich. Evolving algebras: An attempt to discover semantics. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 266–292. World Scientific, River Edge, NJ, 1993.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Ham96] M. Hammer. *Beyond Reengineering: How the Process-Centered Organization Is Changing Our Work and Our Lives*. Harper Collins, 1996.

[HJ95] M.G. Hinchey and S.A. Jarvis. *Concurrent Systems: Formal Development in CSP*. International Series in Software Engineering. McGraw-Hill, 1995.

[HK97] Tom Holvoet and Thilo Keilmann. Behavior specification of active objects in open generative communication environments. In Hesham El-Rewini and

Yale N. Patt, editors, *Proceedings of the HICSS-30 Conference, Track on Coordination Models, Languages and Systems*, pages 349–358. IEEE Computer Society Press, January, 7–10 1997.

[HK98] Tom Holvoet and Thilo Kielmann. Behaviour specification of parallel active objects. *Journal on Parallel Computing*, 24(7):1107–1135, 1998.

[HM] Jim Huggins and Raghu Mani. The evolving algebra interpreter version 2.0.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.

[Hol95] T. Holvoet. Agents and petri nets. *The Petri Net Newsletter*, (49):3–8, 1995.

[HV96] T. Holvoet and P. Verbaeten. Synchronization specifications for agents with net-based behavior descriptions. In *Proceedings of CESA '96 IMACS Conference, Symposium on Discrete Events and Manufacturing Systems*, pages 613–618, Lille, France, July 1996.

[HV97] T. Holvoet and P. Verbaeten. Using agents for simulating and implementing petri nets. In *Proceedings of the Eleventh Workshop on Parallel and Distributed Simulation*, Vienna, Austria, June 1997. ACM/IEEE/SCS, IEEE Computer Society Press.

[HV98] Tom Holvoet and Pierre Verbaeten. Using petri nets for specifying active objects and generative communication. In G. Agha and F. DeCindio, editors, *Advances in Petri Nets on Object-Orientation*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[ITN96] Michiaki Iwazume, Hideaki Takeda, and Toyoaki Nishida. Ontology-based information gathering and text categorization from the internet. In Takushi Tanaka, Setsuo Ohsuga, and Moonis Ali, editors, *Proceedings of the Ninth International Conference in Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE-96)*, pages 305–314, 1996.

[Jen81] K. Jensen. Coloured petri nets and the invariant method. *Theoretical Computer Science*, 14:317–336, 1981.

[Jen92] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume Volume 1, Basic Concepts of *Monographs in Theoretical Computer Science*. Springer-Verlag, 1992.

[Jen94a] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume Volume 2, Analysis Methods of *Monographs in Theoretical Computer Science*. Springer-Verlag, 1994.

[Jen94b] K. Jensen. An introduction to the theoretical aspects of coloured petri nets. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency*, volume 803 of *Lecture Notes in Computer Science*, pages 230–272. Springer-Verlag, 1994.

[Jen96] K. Jensen. An introduction to the practical use of coloured petri nets. In W. Reisig and G. Rozenberg, editors, *Advanced Course on Petri Nets, Bagstuhl 1996*, Lecture Notes in Computer Science. Springer-Verlag, 1996.

[Jen97a] K. Jensen. A brief introduction to coloured petri nets. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems. Proceedings of the TACAS '97 Workshop, Enschede, The Netherlands, 1997*, volume 1217 of *Lecture Notes in Computer Science*, pages 203–208. Springer-Verlag, 1997.

[Jen97b] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume Volume 3, Practical Use of *Monographs in Theoretical Computer Science*. Springer-Verlag, 1997.

[JFN+96] N. R. Jennings, P. Faratin, T. J. Norman, P. O'Brien, M. E. Wiegand, C. Voudouris, J. L. Alty, T. Miah, and E. H. Mamdani. Adept: Managing business processes using intelligent agents. In *Proceedings of BCS Expert Systems Conference (ISP Track)*, Cambridge, UK, 1996.

[Joh98] Thomas Johns. Walk the talk: An agent generator, 1998. http://www.bitpix.com/business/main/walktalk/agentgen/agentgen.htm.

[JS96] Nigel Jacobs and Ray Shea. The role of Java in InfoSleuth: Agent-based exploitation of heterogeneous information resources. Technical Report MCC-INSL-018-96, MCC, 1996. Presented at the IntraNet96 Java Developers Converence.

[JSvR97] Dag Johansen, Nils P. Sudmann, and Robbert van Renesse. Performance issues in TACOMA. In *3rd. Workshop on Mobile Object Systems, 11th Europeean Conference on Object-Oriented Programming*, Jyvaskyla, Finland, June 1997.

[JSvR98] Dag Johansen, Fred B. Schneider, and Robbert van Renesse. What TACOMA taught us. In Dejan Milojicic, Frederick Douglis, and Richard Wheeler, editors, *Mobility, Mobile Agents and Process Migration - An edited Collection.* Addison Wesley Publishing Company, 1998.

[JvRS95a] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. An introduction to the TACOMA distributed system. Technical report, University of Tromso, June 1995.

[JvRS95b] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Operating system support for mobile agents. In *Proceedings of the 5th. IEEE Workshop on Hot Topics in Operating Systems*, Orcas Island, Wa, USA, May 1995. IEEE Computer Society. Also available as Technical Report TR94-1468, Department of Computer Science, Cornell University, USA, November 1994. This paper has also been republished in "Readings in Agents", eds. M.N. Huhns and M.P. Singh, Morgan Kaufmann Publishers, USA, October 1997, ISBN 1-55860-495-2. This paper will also be republished in *Mobility, Mobile Agents and Process Migration - An edited Collection*, Dejan Milojicic, Frederick Douglis, and Richard Wheeler eds., Addison Wesley Publishing Company, 1998.

[JvRS96] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Supporting broad internet access to TACOMA. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages pp. 55–58, Connemara, Ireland, September 1996.

[JW98] N. R. Jennings and M. J. Wooldridge. Applications of intelligent agents. In N. R. Jennings and M. J. Wooldridge, editors, *Agent Technologies: Foundations, Applications, and Markets*. 1998.

[KF97] Adam Kellett and Michael Fisher. Concurrent METATEM as a coordination language. In *Coordination Languages and Models*, volume 1282 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[KGR96] David Kotz, Robert Gray, and Daniela Rus. Transportable agents support worldwide applications. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 41–48, September 1996.

[Kie96] Thilo Kielmann. Designing a coordination model for open systems. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models: Proceedings of COORDINATION '96*, number 1061 in Lecture Notes in Computer Science, pages 267–284. Springer, Cesena, Italy, 1996.

[KIO95] Kazuhiro Kuwabara, Toru Ishida, and Nobuyasu Osato. AgenTalk: Describing multiagent coordination protocols with inheritance. In *Proceedings of the 7th IEEE International Conference on Tools with Artificial Intelligence (ICTAI '95)*, pages 460–465, 1995.

[KIO96] Kazuhiro Kuwabara, Toru Ishida, and Nobuyasu Osato. AgenTalk: Describing multiagent coordination protocols. *Transactions of The Institute of Electronics, Information and Communication Engineers*, 1996.

[KMW98] Olaf Kummer, Daniel Moldt, and Frank Wienberg. A framework for interacting design/cpn- and Java-processes. In *Proceedings of the First Workshop on Practical Use of Coloured Petri Nets and Design/CPN*, Aarhus, Denmark, June 1998.

[KP97] P. Kutter and A. Pierantonio. Montages: Specifications of Realistic Programming Languages. *Journal of Universal Computer Science*, 3(5):416–442, 1997.

[KU58] A. N. Kolmogorov and V. A. Uspensky. To the definition of an algorithm. *Uspekhi Mat. Nauk*, 13(4):3–28, 1958. Russian; English translation in AMS Translations, ser. 2, vol. 21 (1963), 217–245.

[Kuw95] K. Kuwabara. AgenTalk: Coordination protocol description for multi-agent systems. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS '95)*. AAAI/MIT Press, 1995.

[Lab96a] Laboratoire Lip6, Université Pierre & Marie Curie (Paris 6). *The CON-AMI Environment Version 2.0 Beta Release*, 1996.

[Lab96b] Yannis Labrou. *Semantics for an Agent Communication Language*. PhD thesis, University of Maryland Baltimore County, 1996.

[LC93] C. Lakos and Søren Christensen. A general systematic approach to arc extensions for coloured petri nets. Technical Report R93-7, Department of Computer Science, University of Tasmania, Hobart, Tasmania, August 1993.

[LF97a] Yannis Labrou and Tim Finin. Comments on the specification for FIPA '97 AGENT COMMUNICATION LANGUAGE. Internet document, 1997.

[LF97b] Yannis Labrou and Tim Finin. A proposal for a new KQML specification. Technical report, UMBC, 1997.

[LF97c] Yannis Labrou and Tim Finin. Semantics and conversations for an agent communication language. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI '97)*. Morgan Kaufman, August 1997.

[LNSK99] Fuhua Lin, Douglas H. Norrie, Weiming Shen, and Rob Kremer. Schema-based approach to specifying conversation policies. In *Working Notes of the Workshop on Specifying and Implementing Conversation Policies, Third International Conference on Autonomous Agents*, pages 71–78, Seattle, Washington, May 1999.

[LO98a] D. B. Lange and M. Oshima. *Programming and Deploying Agents with Java*. Addison-Wesley, Reading, MA, 1998.

[LO98b] Danny B. Lange and Mitsuru Oshima. Mobile agents with Java: The Aglet API. *To appear in World Wide Web*, 1998.

[MAC+98] D. Milojicic, G. Agha, D. Chauhan, S. Guday, and N. Jamali. Composing agent-based applications and systems. Submitted to SIGOPS Workshop, 1998.

[MC95] Douglas B. Moran and Adam J. Cheyer. Intelligent agent-based user interfaces. In *Proceedings of International Workshop on Human Interface Technology 95 (IWHIT'95)*, pages 7–10, Aizu-Wakamatsu, Fukushima, Japan, 12-13 October 1995. The University of Aizu.

[MCJ+97] Douglas B. Moran, Adam J. Cheyer, Luc E. Julia, David L. Martin, and Sangkyu Park. Multimodal user interfaces in the Open Agent Architecture. In *Proc. of the 1997 International Conference on Intelligent User Interfaces (IUI97)*, pages 61–68, Orlando, Florida, 6-9 January 1997.

[MCL96] David L. Martin, Adam Cheyer, and Gowang Lo Lee. Agent development tools for the open agent architecture. In *Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 387–404, London, April 1996. The Practical Application Company Ltd.

[MCM98] David L. Martin, Adam J. Cheyer, and Douglas B. Moran. Building distributed software systems with the open agent architecture. In *Proc. of the Third International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, Blackpool, Lancashire, UK, March 1998. The Practical Application Company Ltd.

[Mer95] Stephan Merz. Efficiently executable temporal logic programs. In Michael Fisher and Richard Owens, editors, *Executable Modal and Temporal Logics*, volume 897 of *Lecture Notes in Computer Science*, pages 69–85, Berlin, Heidelberg, 1995. Springer-Verlag.

[Mil80] R. Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[Mil91] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. Appeared in *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer-Verlag, 1993.

[ML96] M. Merz and W. Lamersdorf. Agents, services, and electronic markets: How do they integrate? In *Proceedings of the IFIP/IEEE International Conference on Distributed Platforms*, Dresden, Germany, 1996.

[Moc87a] P. Mockapetris. RFC 1034: Domain names - concepts and facilities, 1987.

[Moc87b] P. Mockapetris. RFC 1035: Domain names - implementation and specification, 1987.

[MOMC97] David L. Martin, Hiroki Oohama, Douglas Moran, and Adam Cheyer. Information brokering in an agent architecture. In *Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, London, April 1997. The Practical Application Company Ltd.

[Moo99] Scott Moore. On conversation policies and the need for exceptions. In *Working Notes of the Workshop on Specifying and Implementing Conversation Policies*, pages 19–28, Seattle, Washington, May 1999.

[MPRA99] Francisco Martin, Enric Plaza, and Juan Rodríguez-Aguilar. Conversation protocols: Modeling and implementing conversations in agent-based systems. In *Working Notes of the Workshop on Specifying and Implementing Conversation Policies*, pages 49–58, Seattle, Washington, May 1999.

[MPW89a] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part II. LFCS Report Series ECS-LFCS-89-86, University of Edinburgh, June 1989. Republished in Journal of Information and Computation, 1992.

[MPW89b] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part I. LFCS Report Series ECS-LFCS-89-85, University of Edinburgh, June 1989. Republished in Journal of Information and Computation, 1992.

[MvRSS96] Yaron Minsky, Robbert van Renesse, Fred B. Schneider, and Scott D. Stoller. Cryptographic support for fault-tolerant distributed computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages pp. 109–114, Connemara, Ireland, September 1996.

[MW97] Daniel Moldt and Frank Wienberg. Multi-agent-systems based on coloured petri nets. In *Proceedings of the 18th International Conference on Application and Theory of Petri Nets (ICATPN '97)*, number 1248 in Lecture Notes in Computer Science, pages 82–101, Toulouse, France, June 1997.

[NCK96] Saurab Nog, Sumit Chawla, and David Kotz. An RPC Mechanism for Transportable Agents. Technical Report PCS-TR96-280, Dartmouth College, Computer Science, Hanover, NH, March 1996.

[NNLC99] Hyacinth S. Nwana, Divine T. Ndumu, Lyndon C. Lee, and Jaron C. Collis. ZEUS: A toolkit for builing distributed multi-agent systems. *Applied Artificial Intelligence*, 13(1):129–186, 1999.

[NU97a] M. H. Nodine and A. Unruh. Facilitating open communication in agent systems: the InfoSleuth infrastructure. In Michael Wooldridge, Munindar Singh, and Anand Rao, editors, *Intelligent Agents Volume IV – Proceedings of the 1997 Workshop on Agent Theories, Architectures and Languages*, volume 1365 of *Lecture Notes in Artificial Intelligence*, pages 281–295. Springer-Verlag, Berlin, 1997.

[NU97b] M. H. Nodine and A. Unruh. Facilitating open communication in agent systems: The InfoSleuth infrastructure. Technical Report MCC-INSL-056-97, MCC, April 1997.

[Nwa96] Hyacinth S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 1996.

[Obj97a] ObjectSpace Inc. *ObjectSpace Voyager Core Package: Technical Overview*, December 1997. Version 1.0.

[Obj97b] ObjectSpace Inc. *Voyager and Agent Platforms: Comparison*, December 1997. Version 1.0.

[Obj97c] ObjectSpace Inc. *Voyager and RMI: Comparison*, December 1997. Version 1.0.

[Obj97d] ObjectSpace Inc. *Voyager CORBA Integration: Technical Overview*, December 1997. Version 2.0 Beta 1.

[Obj98] ObjectSpace Inc. *Voyager Transaction Service: Technical Overview*, January 1998.

[OLW96] John K. Ousterhout, Jacob Y. Levy, and Brent B. Welch. The Safe-Tcl security model, 1996.

[OPLD95] Tim Oates, M. V. Nagendra Prasad, Victor R. Lesser, and Keith Decker. A distributed problem solving approach to cooperative information gathering. In *Proceedings of the AAAI Spring Symposium on Information Gathering from Heterogeneous, Distributed Environments.*, 1995.

[Par96] H. Van Dyke Parunak. Visualizing agent conversations: Using enhanced dooley graphs for agent design and analysis. In *Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS '96)*, 1996.

[PBC97] H. V. D. Parunak, A. Baker, and S. Clark. Aaria agent architecture: An example of requirements-driven agent-based system design. http://www.aaria.uc.edu, 1997.

[PC96] M. Purvis and S. Cranefield. Agent modelling with petri nets. In *Proceedings of the CESA '96 (Computational Engineering in Systems Applications) Symposium on Discrete Events and Manufacturing Systems*, pages 602–607, Lille, France, July 1996. IMACS, IEEE-SMC.

[Pei97] H. Peine. An introduction to mobile agent programming and the Ara system. Technical Report ZRI report 1/97, Dept. of Computer Science, University of Kaiserslautern, Germany, January 1997.

[Pet77] J. Peterson. Petri nets. *Computing Surveys*, 9(3):223–252, September 1977.

[Pet98] Charles Petrie. JATLite. Online Documentation: http://java.stanford.edu/, 1998.

[PM99] Jeremy Pitt and Abe Mamdani. Communication protocols in multi-agent systems. In *Working Notes of the Workshop on Specifying and Implementing Conversation Policies*, pages 39–48, Seattle, Washington, May 1999.

[PS97a] H. Peine and T. Stolpmann. Ara - agents for remote action. In William R. Cockayne and Michael Zyda, editors, *Mobile Agents: Explanations and Examples, with CD-ROM*. Manning/Prentice Hall, 1997.

[PS97b] H. Peine and T. Stolpmann. The architecture of the Ara platform for mobile agents. In Kurt Rothermel and Radu Popescu-Zeletin, editors, *Proc. of the First International Workshop on Mobile Agents MA'97*, volume 1219 of *Lecture Notes in Computer Science*, Berlin, Germany, April 1997. Springer-Verlag.

[RGK96] Daniela Rus, Robert Gray, and David Kotz. Autonomous and adaptive agents that gather information. In Ibrahim Imam, editor, *Working Notes of the AAAI-96 Workshop on Intelligent Adaptive Agents*, Portland, OR, August 1996.

[RGK97] Daniela Rus, Robert Gray, and David Kotz. Transportable information agents. In W. Lewis Johnson and Barbara Hayes-Roth, editors, *Proceedings of the 1st International Conference on Autonomous Agents*, pages 228–236, New York, February5–8 1997. ACM Press.

[Ros97] A.W. Roscoe. *The Theory and Practice of Concurrency*. International Series in Computer Science. Prentice Hall, 1997.

[RS99] Inc. Reticular Systems. *AgentBuilder: An Integrated Toolkit for Constructing Intelligent Software Agents*, revision 1.3 edition, February 1999.

[SBH96] M. Straßer, J. Baumann, and F. Hohl. Mole - a Java based mobile agent system. In *Working Notes of the Workshop on Mobile Object Systems, ECOOP '96*, dpunkt, 1996.

[SCB+98] I. A. Smith, P. R. Cohen, J. M. Bradshaw, M. Greaves, and H. Holmback. Designing conversation policies using joint intention theory. In *Proceedings of the Fourth International Conference on Multi-Agent Systems (ICMAS '98)*, Paris, France, July 1998.

[Sch80] Arnold Schönhage. Storage modification machines. *SIAM Journal on Computing*, 9(3):490–508, 1980.

[Sch97] Fred B. Schneider. Towards fault-tolerant and secure agentry. In *Proceedings 11th International Workshop on Distributed Algorithms*, Saarbucken, Germany,, sep 1997. Invited paper.

[SDP+96] Katia Sycara, Keith Decker, Anandeep Pannu, Mike Williamson, and Dajun Zeng. Distributed intelligent agents. *IEEE Expert*, 11(6):36–46, December 1996.

[Sho91] Y. Shoham. AGENT-0: A simple agent language and its interpreter. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, volume 2, pages 704–709, Anaheim, California, 1991.

[Sho93] Yoav Shoham. Agent–oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

[Tho94] S. R. Thomas. The PLACA agent programming language. In M. J. Wooldridge and N. R. Jennings, editors, *Proceedings of the ECAI '94 Workshop on Agent Theories, Architectures and Languages: Intelligent Agents I*, Lecture Notes in Artificial Intelligence, pages 355–370. Springer-Verlag, Berlin, 1994.

[TIN95] Hideaki Takeda, Kenji Iino, and Toyoaki Nishida. Agent organization and communication with multiple ontologies. *International Journal of Cooperative Information Systems*, 4(4), December 1995.

[TIT+96] Hideaki Takeda, Koji Iwata, Motoyuki Takaai, Atsushi Sawada, and Toyoaki Nishida. An ontology-based cooperative environment for real-world agents. In *Proceedings of Second International Conference on Multiagent Systems*, pages 353–360, 1996.

[Tol98] Robert Tolksdorf. Laura – a service-based coordination language. *Science of Computer Programing*, 31:359–381, 1998.

[Tro99] Monika Trompedeller. http://www.dsi.unimi.it/users/tesi/trompede/petri/home.html, 1999.

[TTN97] Motoyuki Takaai, Hideaki Takeda, and Toyoaki Nishida. Distributed ontology development environment for multi-agent systems. In *Working Notes for AAAI-97 Spring Symposium Series on Ontological Engineering*, pages 149–153, 1997.

[TWG93] M. Tennenbaum, J. Weber, and T. Gruber. Enterprise integration: Lessions from shade and pact. In C. Peter, editor, *Enterprise Integration Modeling*. MIT Press, 1993.

[VBW92] T. Vollmann, W. Berry, and D. Whybark. *Manufacturing Planning and Control Systems*. Irwin, New York, 1992.

[Vic95] Björn Victor. *The Mobility Workbench User's Guide, Polyadic version 3.122*. Uppsala University, Uppsala, Sweden, oct 1995.

[VM94] Björn Victor and Faron Moller. The Mobility Workbench — A tool for the π-calculus. In David Dill, editor, *Computer Aided Verification (Proc. of CAV'94)*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.

[Wal97] Charles Wallace. The semantics of the Java programming language: Preliminary version. Technical Report CSE-TR-355-97, University of Michigan Department of Electrical Engineering and Computer Science, 1997.

[WBLX99] Thomas Wagner, Brett Benyo, Victor Lesser, and Ping Xuan. Investigating interactions between agent conversations and agent control components. In *Working Notes of the Workshop on Specifying and Implementing Conversation Policies*, pages 79–88, Seattle, Washington, May 1999.

[WDS96] Mark Williamson, Keith Decker, and Katia Sycara. Unified information and control flow in hierarchical task networks. In *Working Notes of the AAAI-96 Workshop "Theories of Action, Planning and Control"*, August 1996.

[WF86] Terry Winograd and Fernando Flores. *Understanding Computers and Cognition*. Addison-Wesley, 1986.

[Whi95] James White. Mobile agents. In Jeffery M. Bradshaw, editor, *Software Agents*. MIT Press, 1995.

[Wie96] Frank Wienberg. *Multiagentensysteme auf def Basis gefärbter Petri-Netze*. PhD thesis, Universität Hamburg Fachbereich Informatik, 1996.

[WJ94] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 1994.

[WJHC95] Sanjiva Weerawarana, Anupam Joshi, Elias N. Houstis, and Ann C. Catlin. Using ncsa mosaic for building notebook interfaces for cs&e applications. Technical Report CSD-TR-95-006, Department of Computer Sciences, Purdue University, 1995.

[YMB98] Min-Jung Yoo, Walter Merlat, and Jean-Pierre Briot. Modeling and validation of mobile agents on the web. In *Proceedings of the International Conference on Web-Based Modeling & Simulation (SCS Western MultiConference on Computer Simulation)*, San Diego, California, January 1998.