A Multiagent Architecture for Symbolic-Connectionist Integration

José C. González, Juan R. Velasco, Carlos A. Iglesias, Jaime Álvarez de Toledo and Andrés Escobero
Dep. Ingeniería de Sistemas Telemáticos
Universidad Politécnica de Madrid

Workpackage: WP1
Document Type: Deliverable D1
Status: Final revised
Date of Issue: November 15, 1995
Classification: Public Document
Doc. Id.: MIX/WP1/UPM/3.2
Distribution: Free and Consortium

© Departamento de Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid, Spain, 1995
MIX (Modular Integration of Connectionist and Symbolic Processing in Knowledge-Based Systems) is an ESPRIT III Basic Research Project financed jointly by the European Union and the Swiss Federal Government from 1994 to 1996. Project partners are:

- INRIA-Lorraine, Coordinator (France)
- CUI - Université de Genève (Switzerland)
- Institut de Mathématiques Appliquées de Grenoble (France)
- Kratzer Automatisierung (Germany)
- Technische Universität München (Germany)
- Universidad Politécnica de Madrid (Spain)

The goals of the project are the following:

- To enhance the versatility of symbolic-connectionist hybrid systems (SCSs) by combining a variety of representation, inference and learning schemes from the symbolic and the connectionist paradigms.
- To augment the reasoning power of SCSs by overcoming their traditional limitation to propositional logic.
- To clarify the theoretical impact of symbolic-connectionist integration on fundamental machine learning issues such as the choice and combination of learning methods, credit assignment and knowledge assimilation.
- To ensure the scalability and real-world applicability of the resulting hybrid models by testing them on prediction, optimization and classification problems in industrial and other areas.

To achieve these goals, the first project objective, as stated in the Technical Annex to the contract, is:

To design and implement a distributed architecture for the cooperation of multiple agents that use symbolic and/or connectionist processing methods. Implement interfacing methods that will establish an effective dialogue between these heterogeneous agents.

This is the objective of Workpackage 1, that is further described in the same document as follows:
Symbolic-connectionist integration will be pursued within a distributed framework for the cooperation of multiple heterogeneous agents. This first package will consist of the design, implementation and integration of a multiagent architecture, generic agent structures and services, and communication protocols (classes of messages, ways of making a connection or activating/sleeping processes). On the basis of these protocols, generic functions will be specified which will allow any agent to create a communication channel, to send and receive messages, etc. More specific agents will then be created for symbolic and connectionist inference or learning tasks (for example, agents which will implement a fuzzy logic inference engine, a case-based reasoner or a combined rule and theory-based system).

This document, classified as deliverable D1, provides a full account of the multiagent architecture designed for the project. It also serves as a user guide for the current version of the MIX platform, the software package that we have implemented following the guidelines mentioned above. A companion document, deliverable S1, serves as the reference manual of the platform. The authors of this work can be reached by electronic mail at mix@gsi.dit.upm.es.

Madrid, November 15, 1995
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>The Service Object</td>
<td>26</td>
</tr>
<tr>
<td>3.6</td>
<td>Services</td>
<td>26</td>
</tr>
<tr>
<td>3.7</td>
<td>Hierarchies of Agents</td>
<td>27</td>
</tr>
<tr>
<td>3.8</td>
<td>The Agent Class: Default Services</td>
<td>27</td>
</tr>
<tr>
<td>3.9</td>
<td>The YP Agent Class: Default Services</td>
<td>27</td>
</tr>
<tr>
<td>3.10</td>
<td>The BaseAgent Class: Default Services</td>
<td>28</td>
</tr>
<tr>
<td>3.11</td>
<td>Related work</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>3.11.1 Generic agent model</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>3.11.2 Agent communication languages</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>3.11.3 Agent Description Languages</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>MIX Platform</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>The MSM Library</td>
<td>34</td>
</tr>
<tr>
<td>4.3</td>
<td>The CKRL Toolbox</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>4.3.1 The C++ CKRL Library</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>4.3.2 The static ckrl2c++ tool</td>
<td>35</td>
</tr>
<tr>
<td>4.4</td>
<td>The ADL Compiler</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>4.4.1 Interacting with CKRL Toolbox</td>
<td>37</td>
</tr>
<tr>
<td>5.1</td>
<td>The MIX Agent Programming Language: ADL</td>
<td>39</td>
</tr>
<tr>
<td>5.2</td>
<td>Agent hierarchy</td>
<td>40</td>
</tr>
<tr>
<td>5.3</td>
<td>Services</td>
<td>41</td>
</tr>
<tr>
<td>5.4</td>
<td>Goals</td>
<td>42</td>
</tr>
<tr>
<td>5.5</td>
<td>Resources</td>
<td>42</td>
</tr>
<tr>
<td>5.6</td>
<td>Internal objects</td>
<td>44</td>
</tr>
<tr>
<td>5.7</td>
<td>Control</td>
<td>45</td>
</tr>
<tr>
<td>5.8</td>
<td>Types of coupling</td>
<td>46</td>
</tr>
<tr>
<td>5.9</td>
<td>Directives</td>
<td>47</td>
</tr>
<tr>
<td>6.1</td>
<td>Knowledge Representation Languages</td>
<td>49</td>
</tr>
<tr>
<td>6.2</td>
<td>A MIX language for knowledge representation</td>
<td>49</td>
</tr>
<tr>
<td>6.3</td>
<td>CKRL 2.0</td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td>CKRL-MIX</td>
<td></td>
</tr>
<tr>
<td>6.5</td>
<td>Restrictions imposed on the CKRL Language</td>
<td></td>
</tr>
<tr>
<td>6.6</td>
<td>Extensions offered by the CKRL-MIX language</td>
<td></td>
</tr>
<tr>
<td>7.1</td>
<td>Getting started: the foo example</td>
<td></td>
</tr>
<tr>
<td>7.2</td>
<td>Beginning with service programming: the foo service</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.2.1 Header files needed to be included</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.2.2 How to declare a service function</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.2.3 Your first service</td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>Demanding a service: The ask-for-foo goal</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.3.1 Possible ways of demanding a service</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.3.2 How to create and compose a message</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.3.3 Functions for requesting a service</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.3.4 Implementing the ask-for-foo service</td>
<td></td>
</tr>
<tr>
<td>7.4</td>
<td>Running the Application</td>
<td></td>
</tr>
<tr>
<td>7.5</td>
<td>Sending and receiving answers: the echo service</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.5.1 How to read the data of the requesting message</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.5.2 How to send an answer to the agent requesting the service</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.5.3 How to get an answer in a synchronous service call</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.5.4 Implementing the echo service</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.5.5 Implementing the user-echo goal</td>
<td></td>
</tr>
<tr>
<td>7.6</td>
<td>Requesting deferred services: the user-echo-2 goal</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.6.1 How to get deferred answers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.6.2 Implementing the user-echo2 goal</td>
<td></td>
</tr>
<tr>
<td>7.7</td>
<td>The Contract Net Protocol</td>
<td></td>
</tr>
<tr>
<td>7.8</td>
<td>Dynamic calling</td>
<td></td>
</tr>
<tr>
<td>7.9</td>
<td>Calling external programs</td>
<td></td>
</tr>
<tr>
<td>7.10</td>
<td>The Knowledge level: Mixy and Santa Claus</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.10.1 Defining concepts</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.10.2 Declaring Knowledge structures in ADL</td>
<td></td>
</tr>
<tr>
<td>7.11</td>
<td>Sending Knowledge structures: the goal send_letter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.11.1 How to manage CKRL objects while service programming</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.11.2 How to obtain C++ objects from a message</td>
<td></td>
</tr>
<tr>
<td>7.12</td>
<td>Receiving Knowledge structures: the service dispenser_gift</td>
<td></td>
</tr>
<tr>
<td>7.13</td>
<td>Running an application with CKRL</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 1 A Common Platform for the MIX Project

1.1 Introduction

The obvious complementarity of the symbolic and connectionist approaches to problem solving has led to a growing interest in hybrid systems, those involving the cooperation of both approaches in a single, integrated system.

As necessarily happens with new research arenas, we can note some global problems that render the achievement of advances in the field difficult:

• First, most of the research effort until now has addressed specific applications, generating specific hybrid solutions. Such results rarely evolve toward general models of interaction.

• Moreover, implementations are not easy to reuse, as they are most times developed in a purely ad hoc manner. Portability, scalability and, in the last place, applicability to real-world problems, are seriously hampered.

• In this situation, comparing the plausibility, generality and performance of different approaches, models or implementations (in general, any kind of experimental research) becomes an impossible mission.

1.2 A software interoperation approach

To address the aforementioned problems, we propose to adopt a framework devised for the interoperation of heterogeneous systems. Such a framework should feature the following capabilities:

• Modularity: Hybrid systems/models should be developed from basic building blocks comprising symbolic and connectionist modules as well as other hardware/software systems (e.g., data acquisition systems, statistical modules, mechanical actuators, etc.)
1.2 A software interoperation approach

- Support for diverse integration schemes: This framework should support several integration schemes allowing different levels of coupling (ranging from loose to tight) among symbolic and connectionist processes.
- Encapsulation: These components should be encapsulated in order to offer homogeneous interfaces.
- Cooperation: Mechanisms suitable for intelligent cooperation should be supported to allow complex interactions among the components in the framework.
- Distribution: Components should be able to work in distant environments. This requirement can be due to external constraints (such as the execution of individual components in specialized hardware/software platforms), or to the very nature of the problem.
- Ease of use: Solutions imposing a strong discipline on researchers/developers to force the use of complex languages, software development methodologies and tools should be avoided. The overload imposed on researchers to integrate particular pre-existing components in this framework and to organize the interaction of these components should be reduced to a minimum. In any case, such an overload should be fully justified in terms of the inherent benefits of this approach.
- Openness: facilities have to be foreseen for the interoperation of these components with other distributed frameworks currently under development by companies and research institutions.

These considerations lead us to propose a multiagent architecture as the most adequate technology for building a common platform for the MIX project. The starting point for the development of the MIX platform has been the formal agent model proposed and implemented by Domínguez in [Dom92].

To summarize, we call agents autonomous entities capable of carrying out specific tasks by themselves or through cooperation with other agents. Multiagent systems offer a decentralized model of control, use the mechanisms of message-passing for communication purposes and are usually implemented from an object-oriented perspective.

1.3 Overview of the MIX architecture and platform

When talking about a multiagent architecture, two components first have to be described: the agent model(s) and the network model (Figure 1.1). Here, we will explain briefly the main aspects of both models.

From an external perspective, our agent model is structured as a set of elements:

- Services: functionality offered to other agents.
- Goals: functions that an agent carries out for self-interest (not as a result of a petition from another agent).
- Resources: information about external resources (services, libraries, ontologies, groups, etc.)
- Internal objects: data structures shared by all the processes that can be launched by the agent to carry out service requests or to achieve goals.
- Control: specification of how service requests are handled by the agent.

At the network level, coordination is achieved in the platform through a specialized agent YP_agent (Yellow_Pages_Agent). At the time of birth, agents register themselves to a particular YP_agent, giving their net address, the services they offer, and the services they require. They can also subscribe to particular groups. In this way, service petitions can be addressed to an individual agent, to the agents subscribed to a group or to all the agents offering the service. The YP_agent responds to a registration request providing all the information that the agent needs to know (e.g., addresses of the agents offering services that it will request). Such information is contained in...
updated by the YP_agent.

Several communication primitives are implemented, including different synchronization mechanisms (synchronous, asynchronous and deferred communications) and higher level protocols, such as Contract Net.

One important problem regarding the exchange of messages in a society of agents is how to facilitate the mutual understanding of the content of these messages. The adopted solution permits the inclusion of a parameter in message headers expressing the language used to codify its content. Moreover, the content can reference concepts in an ontology shared by the sender and the recipient agents. The current implementation includes tools for the automatic translation of messages written in a reduced version of the CKRL language (MIX-CKRL) to and from C++ code. CKRL (Common Knowledge Representation Language) [C’93] was designed by the MLT consortium (project ESPRIT-2154) as an interchange language for symbolic machine learning algorithms. Apart from this, it is always possible to compose the content of messages in a free format. But, in this case, both the sender and the recipient need to agree previously on how to generate and how to process this information.

The specification of agents is made using MIX-ADL, the Agent Description Language defined for this project. A MIX-ADL file consists, in brief, of the following elements:

• Net address of the machine where the YP_agent for all the agents in the file will be found.
• Default language that will be used to codify messages for all the agents in the file.
• Default ontologies for concepts in the application domain.
• C++ libraries used for low level service specification.
• Specification of the structure of a set of agents. These agents can be organized in a class hierarchy with inheritance of structure, in the style of the object-oriented paradigm.

1.4 The structure of this report

This report has been structured as follows:

Chapter 1. A Common Platform for the MIX Project
   (Current chapter).

Chapter 2. The Network Model
   The MIX platform proposes a layered network model. This chapter describes the different facilities offered by each layer of the model.

Chapter 3. The Agent Model
   The MIX platform has been built on a modified version of a library for multiagent systems support: the MSM library, developed at UPM by Tomás Domínguez. This library covers the communication and transport layers of the MIX platform.

Chapter 4. MIX platform
   High level description of the tools integrated in the MIX platform.

Chapter 5. The Agent Description Language: ADL
   ADL (Agent Description Language) has been designed to specify agents (according to Domínguez Agent Model) in the MIX architecture. A full account of the language is given in this chapter.

Chapter 6. The Knowledge Level: CKRL-MIX
   CKRL (Common Knowledge Representation Language), from the MLT consortium (ESPRIT project 2154), has been selected as the first language for knowledge interchange in the MIX architecture. What has been implemented is a reduced version of the language.

Chapter 7. Quick Guide to ADL/CKRL Programming
   It accounts for all the levels of the platform in great detail: agent, knowledge, communications, etc.

Chapter 8. Examples of Applications
   Some examples are programmed as sets of ADL agents. Examples include SYNHESYS and SETHEO systems, developed by Institut d’Informatique et de Mathématiques Appliquées de Grenoble and Technische Universität München respectively.

Chapter 9. Current Status of the MIX Platform and Future Work
   In spite of the fact that the platform provides full functionality for hybrid applications, some work remains to be done in the near future to improve performance and to offer new functionality.

References

Appendix A. Multiagent Architectures for Software Interoperation
   A general description of the state of the art on distributed software development and distributed artificial intelligence.

Appendix B. Syntax of the ADL Language
Appendix C. Syntax of the CKRL-MIX Language
Appendix D. Code of the Quick Guide
   Complete listing of the source code for the examples of the quick guide.
CHAPTER 2  The Network Model

2.1 Introduction

As has already been stressed, the MIX approach to symbolic-connectionist integration is based on a multiagent architecture. This chapter is devoted to the network model of this architecture (Figure 2.1).

The MIX network model considers two kinds of agents:

- **Network agents**: agents which offer services for maintaining the network.
- **Application agents**: agents intended to address a particular problem through cooperation with other agents.

The MIX network model is a layered model. The defined layers are:

- **Interface layer**: it provides a C++ API for accessing the network and communicating via message-passing.
- **Message layer**: it provides a set of facilities for message composition/decomposition. A message consists of:
  - Addresses of sender and addressee. The agent names should be translated into transport addresses.
  - Speech act of the message: this represents the primitive interactions available and understood by an agent according to a particular protocol: request a service, reply to a message, subscribe to the network,... The complete set of permitted interactions constitutes the Agent Communication Language, and can include both interactions between application agents and interactions between network agents and network.
  - Body: content of the message and language to encode/decode it. We will use a knowledge representation language, though other languages could be allowed.
- **Transport layer**: this deals with internetworking and is built on TCP/IP sockets.
2.2 The Interface layer

One of the characteristics requested from our network model is to allow heterogeneous agents to interoperate. This layer provides an API for sending and delivering messages.

This layer offers the following primitive functions:
- **Composition**: this allows the composition of a message
- **Sending**: this allows the sending of a message to the network
- **Receiving**: this allows the reception of a message from the network
- **Decomposition**: this allows the obtation of structures from the received message, i.e. to interpret the message.

This layer has been implemented as a C++ library and it is composed of two main modules (see Figure 2.2):
- **Network Communications Model**: this allows the maintenance of network connections.
- **Mailbox**: this allows the composition and storage of message objects.

These objects are described in depth in the Reference Manual.

FIGURE 2.1 Network model

FIGURE 2.2 Interface layer
2.3 The Message layer

This layer is the core of the network model and it is the layer defined for agent communication, while the other two layers are general telematic layers. It receives a message from an application and delivers the messages to the transport layer.

This level, which will be modelled as speech acts, offers a set of modules to normalize the interactions between agents:

- **Network facilities**: provide the means to log into the network and to receive information about the dynamic changes in the network (mainly connection and disconnection). This layer is managed by the YP agent, which is described below.

- **Coordination facilities**: this level deals with message structure and agent communication language. This layer provides a limited set of performatives for interagent communication. It also includes some of the coordination mechanisms of DAI, such as contract networks [Smi88].

- **Knowledge facilities**: provide facilities to support ontologies. The current implementation only deals with CKRL. An extension to KIF is also being developed.

2.3.1 Network facilities

We will distinguish between the services offered by this layer, which can be considered general for a multiagent architecture, and the particular implementation of these services in the MIX Network Model. So, we will start by describing which services are offered, and then how these services are offered in our architecture.

This level offers the following services:

- **logging in / logging out**: some primitives must be provided to subscribe/unsubscribe to the network. While an agent is not subscribed to the network, it remains unknown to the rest of agents.

- **register / deregister capabilities**: some primitives must be provided to register/unregister the capabilities of the agents, so that other agents can request these capabilities.

- **register / deregister required capabilities**: some primitives must be provided to register in which capabilities an agent is interested. This allows the filtering of the information an agent receives.

- **agent Name Server**: this is needed to maintain and to consult a database with agent names and the transport address of the agent.

- **notify logging in / logging out**: provides a means of notifying new subscriptions/unsubscriptions to the network.

- **trading**: a service such as trading (see Appendix A) seems suitable for multiagent architectures. It allows the communication between service providers and service consumers.

- **management of agent groups**: maintaining groups of agents can be useful to allow cast to a group of agents. If the groups are dynamically constructed, some services required to subscribe/unsubscribe to a group.

2.3.1.1 The MIX Network facilities

There are different possibilities in offering the facilities listed above:

- **No network agent**: each agent has capabilities of registering new agents. When an agent is born, it must have some initial addresses to connect with. An agent can inform the known agents of the birth/death of agents. Each agent must maintain locally agent and agent addresses. We could consider different strategies (all agents act as network agents, the agents compete with the rest of agents to be network agents, ...).

- **Centralized network agent**: there is one network agent which offers all the network services.

- **Distributed network agents**: instead of one network agent, there is a federation of network agents as in ODP-Trader, that is, each group of agents communicates with a local network agent, and the network agents can communicate with each other to provide information to the rest of the network. A protocol between network agents is needed.

The approach followed by the current MIX network model has been to have a centralized network agent, call YP Agent (Yellow-Pages Agent).

The advantages of this approach are:

- only one network address is needed as a priori knowledge
- the facility of maintaining consistency in the database of agent names

This approach has been taken under the following premise:

- **limited number of agents**: as our architecture will be used in concrete applications, there will not be a great number of agents, so there will be no bottleneck in the YP Agent.

The disadvantages of this approach are obvious, the same as that of a centralized system versus a distributed one: a fault in the YP Agent shuts down the complete multiagent architecture.

2.3.1.2 The YP Agent

The YP_Agent offers the following services (that is, it understands the following speech acts):

- **check_in**: this service is used to log into the network, register capabilities and requested capabilities in order to be notified of which agents are offering them.

- **check_out**: this service is used to log out of the network, and notifying this to the interested agents.
• **subscribe_to_group**: this service allows the subscription to a public group, and the reception of the messages sent to this group.
• **unsubscribe_to_group**: this service allows the cancellation of a subscription to a public group.

The YP_Agent employs a strategy of **informing without demand**. This is also a valid strategy when the number of agents is limited (as in our case), and has the advantage of making the dynamic changes of the agent society transparent to the agents. If the number of agents were high, there would be a saturation of messages from the YP_Agent.

The YP_Agent takes charge of testing if the agents are alive. When an agent crashes, it can have sent a **check_out** message or not, so YP_Agent should test the agents periodically to maintain the consistency of its own database. It can also use some heuristics:
• an agent is alive if it has requested something after a certain time period.
• an agent can be idle if it does not accept connections, etc.

### 2.3.2 Coordination facilities

This layer defines the set of interactions permitted between application agents. So, the Agent Communication Language is composed of speech-acts between application agents and speech-acts between network agents and the rest of the agents.

The following speech acts have been considered:
• **AskForService**: this primitive is used to request a service. There are three possible ways of synchronization: **asynchronous**, **synchronous** or **deferred** in which this primitive can be invoked.
• **Answer**: this primitive is used to answer a service and send the final results.
• **PartialResults**: this primitive is used to send partial results.
• **Failure**: this primitive notifies errors which do not allow the completion of the requested service.

There are also some primitives to implement a version of the Contract Net Protocol:

When an agent wants to request a service from several agents and to select the best offer(s) to perform the service, it sends a message to the group with the primitive **AskForServiceWithCost**. The agents interested in performing should send the estimated cost (via the primitive **AnswerCost**). The agent which sent the advertisement will select some agents to perform the service (via the primitive **Ack**) and will reject the other offers (via the primitive **Nack**).

The contracted agents send the results via the primitive **Answer**, and can also report error conditions via the primitive **Failure** or incremental/partial results via the primitive **PartialResults**.

### 2.3.3 Knowledge facilities

There are two main functions of these services:
• To provide a transparent interface between objects and knowledge descriptions. The MIX platform offers a set of tools to manage CKRL descriptions. It is described in chapters 4 and 6 of this document.
• To provide some primitives to access to knowledge object of other agents (assert/retract a fact, ask for an ontology, ...) These primitives have not yet been implemented. We are working on an integration of the platform with the expert system CLIPS developed by the software Technology Branch of the Lyndon B. Johnson Space Center, NASA, and when the integration has been completed, we will add primitives for these actions.

### 2.4 The Transport layer

This layer has been implemented over TCP/IP. The interface of this layer is provided by the Communications object, described in depth in the Reference Manual.

### 2.5 Related work

Several models have been proposed to allow interoperability between agents, here we review here some of them.

#### 2.5.1 SKTP

SKTP (Single Knowledge Transfer Protocol) [TFMc] “supports KQML interactions and is considered as a protocol stack with at least three layers: content, message and communication. These layers are built upon a reliable network transport communication”.

The protocol KQML is described in detail in Appendix A.

#### 2.5.2 HISML

Virdhagriswaran [Vir] proposes a stack protocol (see Table 2.1) which integrates some of the distributed processing environments presented in Appendix A. The supporting protocol is not considered a layer in the protocol. Virdhagriswaran also proposes an extension to KQML, the language and protocol HISML (Heterogeneous Information System Messaging Language).
2.5 Related work

MIX (ESPRIT 9119): A Multiagent Architecture for Symbolic-Connectionist Integration

2.5.3 EMMA

EMMA (Enterprise Modeling and Management Architecture) [Sycara92] is divided into six layers:

- **The network layer** provides the definition of the network. The services provided are: name server, registration of agents, message passing and synchronization primitives.
- **The data layer** provides distributed access to agents' databases via SQL-like queries.
- **The information layer** provides distributed access to agents' databases without knowing where the information is located.
- **The organization layer** provides primitives and elements for distributed problem solving: assignment of roles, goals, responsibilities and authorities inside the organization.
- **The coordination layer** provides protocols for group decisions and negotiations.
- **The market layer** provides protocols for coordination between organizations in a market environment.

<table>
<thead>
<tr>
<th>Layers</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
<td>APIs for applications</td>
</tr>
<tr>
<td>Domain</td>
<td>Support for ontology: KIF, LOOM, ...</td>
</tr>
<tr>
<td>Transaction</td>
<td>Controls dialogue (prevent deadlocks, …)</td>
</tr>
<tr>
<td>Facilitation</td>
<td>Semantic events (OpenDoc), Corba Msg, … - KQML with performatives</td>
</tr>
<tr>
<td>Network</td>
<td>Provide network addresses, registration of capabilities</td>
</tr>
<tr>
<td>Message</td>
<td>KQML messages</td>
</tr>
<tr>
<td>Transport</td>
<td>Packets (KQML---&gt; Semantic events (OpenDoc), Corba Msg, …)</td>
</tr>
<tr>
<td>Supporting</td>
<td>TCP/IP Sockets CORBA RPC-DCE HTTP</td>
</tr>
<tr>
<td></td>
<td>Ole OpenDoc SMTP</td>
</tr>
</tbody>
</table>

**TABLE 2.1 Layered Architecture**

2.5.4 I^3 Reference Architecture

The I^3 reference architecture [Gene95a, Gene94, Gene94a, Gene94b, Gen] defines the following services that an agent network should provide:

- Query processing services: services of distributed database access. Translation of the different SQL-dialects.
- Data/knowledge Interchange services: translation and management of the different languages, standards and protocols (KIF, KQML, CORBA, …).
- Active services: daemon services which can be programmed to be launched based on events.
- Security and Privacy Services.
- Repository Service.

The access to the network is via a special agent: the facilitator. The agents do not connect directly to the network. There is one facilitator per machine, and all the agents in the same machine interact only with the machine facilitator. The facilitator is a kind of service access point. This facilitator provides white pages, yellow pages, message passing, content-based routing, problem decomposition, translation and monitoring. A similar approach is followed by the SodaBot architecture [Coen94], in which a basic software agent (BSA) runs as a background process in each workstation, providing an “agent operation system” which handles the low-level tasks.

Agents register by specifying their interests and capabilities to their facilitator. The facilitators are federated.

The main difference of this approach is that the facilitator is the entity which provides the different network services.

2.5.5 Discussion

HISML and I^3 deal with software interoperability taking into account both message based integration and data flow based integration. As SKTP is more a reference model than a particular architecture, it is extended by HISML in several ways:

- the transport layer is explicitly considered.
- different gateways for data mapping and operation mapping between heterogeneous systems are considered. It supports different interactions (event-based, data stream-based, object stream-based,...) and different types of data (database tables, objects,...).
- it also extends the addressing of sender and receiver, the offered APIs (more diverse than the facilitator capabilities) and it adds some new performatives to KQML.

The first three layers of EMMA are covered by the HISML and I^3 architectures, but the upper ones, related to coordination are not considered in either. Some kind of coordination is covered in I^3: problem decomposition by the facilitator.

The network model defined by HISML and I^3 is quite different. While the network model of I^3 is just facilitators interconnected and agents connected to these facilitators, the network model of HISML is more complex: there gateways and bridges to support the different kinds of data services.
and operations. In the last model there is no such an intelligent agent as the facilitator, so different network elements are defined.

This different architecture proposes different “network agents”. There are different alternatives:

- yellow pages service: translation between agent names and internet addresses; registration of agents.
- white pages/broker service: register of services (such as a trader service).

These services can be centralized by one agent listening on a port, or distributed, which implies the establishment of a protocol between name servers (the same problems and solutions as those proposed to the trader service).

2.5.6 Coordination

There are different approaches to coordination:

- KQML should be extended to allow coordination and negotiation. This approach is, for instance, the approach of the COOL language [Barbu95], which extends KQML to allow conversations. As in Natural Language Processing, a discourse processing is needed to understand a sentence, so COOL allows the maintance of a conversation space in which each KQML-interaction is included.
- KQML should be general enough to support software interoperability. As not all the systems need to be coordinated, there must be a pragmatic level over the semantic one (KQML).
- This pragmatic level will manage the coordination issues. A coordination ontology shared by the agents should be needed, that is, coordination should be performed in the content level (with a language like KIF).

2.6 Discussion of the MIX Network Model

One of the main features of the MIX Network Model is that it provides a C++ API, in contrast to the use of specialized languages such as April or CUBL. This allows the easy integration of other applications.

A distributed naming has been followed, that is, an agent name is compound by “domain” and “name”, being not allowed to run two agents with the same name in the same domain. The domain can be used for describing a logical domain (e.g. an application, a module in an application,...) or a physical domain (e.g. dit.upm.es, loria.fr,...).

Another particular characteristic of the MIX is a special agent, called the YP agent. This agent provides the following services:

- registration: an agent can register in the network. It indicates the offered services, relevant services, known ontologies, groups it is subscribed to and groups it is interested in.
- deregistration: when an agent leaves the network, it notifies it to the YP_Agent. An automatic deregistration is performed by the YP_Agent, which tests periodically that the agents are alive.
- public groups management: agents can subscribe and unsubscribe to public groups, which are treated as mailing-lists. These groups are named public as opposed to private groups, in which each agent can group a set of agents for particular purposes.
- offered and requested services management: agents can modify dynamically the services they offer or they are interested in.
- broadcasting: the changes of the YP database are notified to the interested agents. This way, all the agents know the active agents they are interested in. This approach is only valid when the number of agents is limited (if not, the broadcasted messages would clog the agent mailbox). But if we view an application as a local network with a limited number of agents, this approach allows the maintance of the distributed database by the agents.

The YP approach is strongly influenced by the ODP reference model, and the performatives are now being adapted to the KQML language (see Appendix A).

The purpose of the platform has been to allow neural-symbolic integration, so the network model does not pay special attention to the integration of databases or event-based applications as contrast to the architecture HISLM.

This architecture provides performatives (and high level functions in the API) to perform a contract-net protocol. As these kinds of performatives should be included in KQML or not is under discussion, it is not clear if they will be maintained in future versions.
CHAPTER 3  The Agent Model

3.1 Foundations of the proposed agent model

In this chapter we describe the adopted agent model. This agent model is based on the agent model proposed by Tomás Domínguez [Dom92]. We also describe some enhancements to the model to interact with the network model. The proposed agent model is constructed along the following research lines from the Object Oriented Concurrent Programming and Distributed Artificial Intelligence [Dom92]:

• The use of mechanisms of encapsulation, isolation and local control: each agent is an independent entity endowed with autonomous control and distinct behaviour; its knowledge remains invisible to other agents unless communicated via messages.

• The integration of heterogeneous problem solvers where no assumptions are made regarding an agent’s knowledge (which can be either declarative or procedural), problem-solving methods, or granularity level: systems can be composed of simple entities such as network units or more sophisticated agents such as those involved in cooperative distributed problem solving.

• The methods which allow flexible organization and structuring of interaction, thus providing for dynamically evolving groups.

An agent’s high grain level has a disadvantage—that is, an agent is occupied by a single task whereas it could be doing several things at the same time. Hence the need for a lower grain level of processes associated with each agent, which could be running in parallel without necessarily occupying the agent. In this way, the work of N agents can be done by a single agent with N processes has the advantage of simplifying the communication problem which implies maintaining a routing table for each agent. The size of routing tables grows in an exponential way with the number of agents in a system. By associating N processes to agents, we reduce the number of agents, thus reducing the number of routing table entries.

This is precisely one of the main and original characteristics of Domínguez’s thesis [Dom92]: the concurrent execution of the agents and its services.
3.2 Communication mechanisms in concurrent models

We now present the basic mechanisms of communication in concurrent models and the MIX agent model.

3.2 Communication mechanisms in concurrent models

3.2.1 Introduction

There are two main communication mechanisms in concurrent models:

• **variable sharing** (plus some mechanism for mutual exclusion to ensure coherence) via shared memory, monitors, semaphores, active waiting and conditional critical regions.

• **message passing**, in particular, direct use of primitives such as SEND and RECEIVE, rendezvous, direct procedure calls or atomic transactions. Messages are sent by processes associated with agents and are received by the agents themselves. A message sender should know another agent’s address in order to send it a message directly. An address can be known initially (decided by the designer) or communicated via a message. However, messages can be send indirectly to an agent - either by broadcasting (the message is received by all system agents) or by sending a message to a group to which the agent belongs.

3.2.2 Synchronization mechanisms

Synchronization is a set of restrictions imposed to order events produced in an unpredictable manner by concurrent processes.

Synchronization mechanisms for communication via variable sharing:

• **mutual exclusion**: ensures the execution of a sequence of declarations, called a critical section, in an indivisible way.

• **conditional synchronization**: delays the action of a process on a data structure as long as its current state is inappropriate for the execution of a given operation (semaphores, monitors,...).

Synchronization mechanisms for communication via message passing (used in MSM):

• **asynchronous**: processes are not blocked while waiting for the answer.

• **synchronous**: the sender remains blocked until the peer agent answers.

• **deferred**: used in processes which do not require an immediate answer; between the moment a service request is sent and the time the response is needed (synchronization point), the process can do other things.

3.3 The MIX Agent model

The MIX platform allows the implementation of new agents according to the agent model proposed by Tomás Domínguez [Dom92]. This agent model has been implemented in C++ classes and it is possible to define hierarchies of agents. A declarative language, ADL (Agent Definition Language), has been defined to easily allow the definition of new agents, and a toolkit to translate these definitions has also been implemented.

3.3.1 Domínguez’s agent model

Domínguez’s agent model (see Figure 3.1) is a concurrent object oriented model.

The main characteristic of Domínguez’s model is the way in which an agent attends to a service when a service is requested from an agent and the agent decides to perform it, the agent creates a concurrent process which attends to the service. The agent only attends to the incoming messages and creates concurrent services to perform it if necessary. So, the main functions of the agent are:

• **Mailbox manager**: the agent takes charge of the incoming messages. Only the agent public address for white and yellow pages services. The agent also has a policy for attending to incoming messages and for fixing preferences when asking for a service.

• **Database manager**: the agent maintains its own database (both environment model and shared objects) which is accessed by its services.

This basic model has been extended in several ways:

• The addition of different policies to select agents in order to send a message or a request.

• The addition of two different groups of agents: public and private.

• The definition of a network agent, YP, which acts as an Agent Name Server and manages service interface descriptions. This agent will provide the Network Services of the Network Model.

• The modification of the basic control loop in order to interoperate with the YP_Agent.

• The explicitation of the goals, objects and services requested by the agent itself (autonomous).

• The inclusion of a version of the Contract Net Protocol in the speech acts.

• The definition of a declarative language to define agents, ADL. ADL allows the definition of classes of agents and particular agents. It does not allow the programming of the services, which should be programmed in C++.

3.4 The MIX Agent Model

The agent architecture is composed of the following blocks:
3.3 The MIX Agent model

MIX (ESPRIT 9119): A Multiagent Architecture for Symbolic-Connectionist Integration

- Required Groups
- Required Ontologies
  - Environment Model
    - Known Agents
    - Known Groups
    - Private Groups
  - Agent State
    - list of child processes
    - list of open communications
- Private Objects
- Global Data
- Network Communications Model
- Mailbox

3.4.1 Control
The agent’s control is the set of actions the agent performs. The basic control of a typical agent is shown in Figure 3.2.

Firstly, the agent sends a message to be connected to the network. That is performed by using one of the services (check_in) provided by the network model (specifically by the network level of this model).

There are some functionalities inside this control block:
  - **Network attention**: the agent must manage network connections, that is, it has to interact with the network model via the network interface object.
  - **Destination Policy**: the agent can establish default policies for selecting a subset of the possible agents which carry out a service in order to advertise an offer. That is, it can ask for a service from all the known agents which perform it or from a restricted set (e.g. the agents with lower communication costs,...).
  - **Service Policy**: the agent can establish default policies for accepting service petitions.
  - **Mailbox Management**: the agent can establish different strategies for determining the order in which the service petitions are processed.
  - **Agent Goals**: the agent can have a set of goals to perform when it is born. The agent grammar can establish what actions must be performed depending on the goal achievement.
  - **Offered services**: (concurrent and non concurrent): the agent offers a set of services to the rest of the agents. These offered services can be executed in a concurrent way (as an asynchronous call) or in a non concurrent way (as a synchronous call).
3.3 The MIX Agent model

3.4 Mix (ESPRIT 9119): A Multiagent Architecture for Symbolic-Connectionist Integration

pendent process) or in a non-concurrent way (that is, the agent itself performs the services and stops the basic control loop).

• Offered primitives: the agent offers a set of internal services which may modify some of its private objects. These services are performed by the agent itself, in a non-concurrent way.

• Required services: this a list with the names of the services the agent requires. The agent tells the YP-Agent which services it requires. In this way, the agent will be dynamically informed by the YP-Agent about the dynamic changes of the agent society relevant to the agent.

• Public Subscribed Groups: the agent can subscribe to different public groups, which work as the mailing lists. The name of those groups to inform the YP-Agent are stored here.

• Required Groups: this a list with the names of the public groups the agent requires. The agent informs the YP-Agent of what public groups it knows (and uses, for example, in its service policy).

• Required Ontologies: the agent can declare what ontologies it knows.

FIGURE 3.2 Basic Control of an agent

Send a "Check_in" message to "YP-Agent"
Wait for the answer
Update the internal data base
Start the goal functions (if any)
REPEAT FOREVER
Look at the mailbox for new messages (listen)
If there are pending messages
    Select one of them according to (mbox_manager function)
    CASE kind of message (speech act)
    AskForService:
        IF it has to be processed (according to serv_policy)
        Execute the appropriate service method
        ENDIF
    AskForServiceWithCost:
        IF it has to be processed (according to serv_policy)
        Execute the appropriate cost method
        ENDIF
    Answer: (goal or service achieved)
    Execute associated method
    ErrorReport: (goal or service not achieved)
    Execute associated method
    OTHERWISE
    Error
    ENDCASE
ENDIF
ENDREPEAT

3.4.2 Environment Model

• Known Agents: a collection of Agent Model objects. An Agent Model object stores information about another agent that may be interesting for the agent. This information is composed of services, required services, required groups, subscribed groups, required ontologies, name, domain and address of an agent.

• Known Groups: a collection of Group objects. A Group object has a name and a collection of objects with the name, domain and address of the alive agents which make up the group. The public groups the agent requires are stored here.

• Private Groups: a collection of Group objects. The private groups the agent defines for its own use are stored here.

3.4.3 Agent State

• List of child processes: the agent keeps information here about the child processes, such as concurrent services and goals, that it has started and are still alive.

• List of open communications: the agent keeps a collection of objects here for communicating with the agents that are requesting a service from this agent.

3.4.4 Private Objects

Data structures managed by the agent to service requests or to achieve goals. The current version of the library provides two primitives to access the private objects from the services: GetData and PutData (see the Reference Manual).

3.4.5 Global Data

Information on the agent's operational problem-solving state. This information is used by the control block with a view to defining their own behaviour. Global data also comprise domain knowledge which can be introduced in the arguments of services provided by an agent.

3.4.6 Network Communications Model

This block allows the agent to request services from other agents and to answer the agent that requested a service. It also allows the agent to request a service with different types of synchronization. Destination policy of the Control block determines to which agents a service must be requested. This model is provided by the Interface layer of the Network Model.

3.4.7 Mailbox

Here, the agent keeps a list of the messages received which are petitions of service execution. Mailbox Policy of the control block determines which of these service petitions will be first attended and which will be discarded. This object is also provided by the Interface Layer of the Network Model.
3.5 The Service Object

The Service object is used by both concurrent and non-concurrent services to carry out their work. This object allows a service to communicate with other agents, to communicate with its agent and to answer the originator agent. The architecture of the Service object is composed of the following blocks:

- **Network Communications Model:** This block allows the service to request services with different types of synchronization. It also allows the answering of service petitions.
- **Cost Evaluation:** Since services may be requested with costs, this block decides which offers are accepted. In this way, only the agents which returned a cost accepted by this block are selected to carry out the service.
- **Destination Policy:** This block decides which agents will be finally solicited to carry out a service request.

3.6 Services

In general, a service is a task accomplished by an agent; it can be implemented in procedural, declarative, or knowledge-based form.

From an internal point of view, the services can be classified by two properties:

- **Execution type:** it may be concurrent or non-concurrent:
  - Non Concurrent: the service is accomplished by the agent process. In this case, during its execution the agent doesn’t respond to any other service petition and itself remains blocked until all the actions associated with the requested service are terminated.
  - Concurrent: the service is accomplished by a child process of the agent process that executes concurrently with the agent process. The agent is thus discharged from executing the service and can attend to new requests, since it is not blocked. This mode of operation constitutes the basic difference between the proposed architecture and others. This way the agent process is free to attend more pending service petitions.
- **Used object:** this may a Service object or an Agent object:
  - Service object: this object allows the service to communicate with agents, but it does not allow the direct access to the agent database.
  - Agent object: this object allows the service to communicate with agents and to access directly the agent database for consulting and updating.

According to these properties, we classify the services in three types:

- Concurrent Services: they execute concurrently and use a Service object.
- Non Concurrent Services: they execute non concurrently and use a Service object.
- Primitives or Internal Services: they execute non concurrently and use an Agent object.

3.7 Hierarchies of Agents

The MSM platform has been implemented using the Object Oriented Paradigm. This allows implemention of a hierarchy of agents. In the present version of the MSM library, two classes of agents derived from the elemental Agent class are implemented: the BaseAgent class and the YPAgent class. These classes allow the derivation of new classes of agents, extending the hierarchy.

The YPAgent class has been designed to have an agent that will be in charge of informing the other agents, which will belong to the Base Agent class or a class derived from it, about the births and deaths of agents. The programmer should use the Base Agent class for deriving new classes, because this class has been designed with methods to communicate with the YP_Agent to inform it about their birth or death.

3.8 The Agent Class: Default Services

All the agents offer, by default, a service called **Suicide**. This service performs several actions:

- Closes the agent mailbox. This way the agent can not longer receive service requests.
- Discards all the pending service requests.
- Orders all its goals and concurrent services to die.

After performing these actions, the agent is dead.

Next, we will see the new default services offered by the two derived agents:

- **YP Agent:** CheckIn, CheckOut, ShutDown, KillAgents and KillDomain.
- **BaseAgent:** UpdateDBIn, UpdateDBOut and SuicideInOrder.

The default service **Suicide** is still offered by the derived classes.

3.9 The YPAgent Class: Default Services

The YPAgent is offers five default services:
• **CheckIn**: When an agent is born, it must request this service from the YP_Agent. This service adds the new agent to the YP_Agent database and informs the new agent about the living agents which offer the services the new agent requires and about the agents which make up the public groups it requires. The YP_Agent also informs, requesting the service `UpdateDBIn`, about the new agent to the agents that require services offered by the new agent and to the agents which require the public groups to which the new agent is subscribed.

• **CheckOut**: When an agent dies, it must request this service from the YP_Agent. This service deletes the agent from the YP_Agent database and informs, requesting the service `UpdateDBOut`, about its death to the agents that require services offered by the dead agent and to the agents which require the public groups to which the dead agent was subscribed.

• **ShutDown**: This service sends a message to all the agents the YP_Agent knows ordering them to die. Afterwards, the YP_Agent dies.

• **KillAgents**: This service sends a message to all the agents the YP_Agent knows ordering them to die. The YP_Agent remains alive.

• **KillDomain**: This service sends a message ordering to all the agents the YP_Agent knows which belong to the same domain to die.

### 3.10 The BaseAgent Class: Default Services

The BaseAgent offers three default services:

• **SuicideInOrder**: This service first requests the `CheckOut` service of the YP_Agent, and then performs the same actions as the service `Suicide`.

• **UpDateDBIn**: This service is offered to add new agents to the agent database. It should only be requested by the YP_Agent when it is informed about the birth of an agent.

• **UpDateDBOut**: This service is offered to delete agents from the agent database. It should be only requested by the YP_Agent when informed about the death of an agent.

### 3.11 Related work

Being heterogeneous means that different agents can gain access to the network and communicate to each other.

In this section we will review some of the different proposed agent models, and their relationship to ADL.

Müller [Mueller94] distinguishes between two kinds of agent models: *vertical agent architectures*, if all the layers have access to the perception and action modules, or *horizontal agent architectures*, if one layer has only communication with the adjacent layers.

### 3.11.1 Generic agent model

There some functional blocks shared by most of the agent models:

- Communication subsystem and Message handler subsystem
- Environment model
- Self model
- *(Optional) Knowledge-based subsystem
- *(Optional) Cooperation subsystem
- *(Optional) Belief subsystem

An agent programming language allows the definition of the agent model, the desired behaviour and the API to interact with other agents.

We will distinguish between:

- **Agent communication languages**: we will give the name Agent Communication Language (ACL) to the language for defining the interaction between agents. It is the language defined by the Coordination services of the Message layer in the MIX-Network model.

- **Agent description languages**: we will give the name Agent Description Language (ADL) to the language for defining the model and behaviour of an agent. In the MIX-Platform, this is the function of the defined ADL language.

### 3.11.2 Agent communication languages

#### 3.11.2.1 Procedural agent communication languages

These are based on scripting languages. Agent communication is based on the exchange of procedural directives. Some of these agent programming languages are:

- **Sodabot platform** [Coen94] includes an agent programming language, SodaBotL, based on Perl, extended Tcl/Tk and C. The main purpose of sodabot is to define software agents in the sense of *application agents* (or artificial secretaries), i.e. agents that automate simple, repetitive and time-consuming tasks (e.g. mail filters, ...)

- **Telescript** defines object oriented systems with a script-based programming language.

These languages are direct and easy to program. The main disadvantage of these languages [Gene95a] is that the scripts are difficult to mix, in contrast with the declarative approach. They nevertheless, be very useful for building interface agents.

These kinds of agent communication languages allows the definition of which procedure be executed when a certain message is received. For instance, the SodaBot agent model consists of global variables, the role of the agent (server/client), the “main agent” or main program of the
a yellow pages service; Monitor, which allows the analysis of the transferred messages, and Editor, which allows the access and modification of the agent model.

- DAISY agent model consists of three parts: knowledge base, set of actions to be performed in response to external stimuli and the engine which defines the agent behaviour.

• Application level: applications built using the agent programming language. (This level is only explicit in IMAGINE).

3.11.3.3 Agent oriented programming languages

Following the Shoham AOP paradigm [Sho93], PLACA [Thomas94] is an agent programming language whose agent model takes into account the mental state of the agent. An agent program is a compound of transition functions between these mental states. A mental state consists of beliefs, capabilities and obligations. The mental-change rules determine what messages will be sent and what mental-events will be performed when a particular message under a particular mental-condition is received.

3.11.3.4 Production rule languages

MAGSY [Weise92], DYNAACLIPS [Cenge94] and RTA [Wavish94] are production rule languages for building multiagent architectures.

MAGSY is a OPS5 based production language and Dynaclips is a blackboard architecture based on CLIPS.

RTA defines an agent model with three layers: roles, skills and behaviour. Each agent can adopt some role and can perform some skills according to the role, and they will have some basic behaviour according to the skills. RTA allows the implementation of this behaviour.

3.11.3.5 AgentSpeak

The AgentSpeak agent model [Weer94] consists of internal mental state, proactive or goal driven behaviour and reactive behaviour and communication through speech acts.

The agent programming language AgentSpeak allows the definition of agent families and agent instances of these families. An agent consists of services (public, if they are offered to the other agents; or private, if they satisfy their own goals); database (set of public and private objects/relations); and plans to perform the services.

There are three types of services: achieve a certain relation, query a relation or be told of a relation. These services can be invoked by three speech acts: inform, request with wait and request with no wait. The three speech acts can be requested from an agent, to an agent family or to an agent of an agent family.
CHAPTER 4  MIX Platform

4.1 MIX platform

The MSM Architecture provides a standard library for managing software agents from the point of view of Object-Oriented Concurrent Programming and Distributed Artificial Intelligence. With the project in mind, a specialized language (Agent Description Language, ADL) has been designed to specify the agents that are intended to cooperate in order to solve a problem in a hybrid work. Finally, some tools are being implemented to translate the description files to standard C++ programs and to compile them.

A scheme of the platform can be found in Figure 4.1. It is made up of four elements:

- MSM C++ library: this offers the basic low-level functionality of the platform, including the typical mechanisms of multiagent libraries and general purpose objects and functions.
- ADL compiler: this translates the ADL description of a particular set of agents into a set of independent executable programs (one program per agent). These agents can cooperate in the same application, possibly together with other agents compiled in the same or different machines in a distributed and heterogeneous environment. In this task, it will use the following two complementary sources:
  - CKRL toolbox: this translates both CKRL descriptions to C++ classes and objects, and objects to CKRL descriptions.
  - Standard ADL agent definitions and CKRL ontologies.

The user of the system has to write an ADL specification of his application and, eventually, a library containing some objects and functions that implement services, goals, etc. The result obtained from the platform using these elements is a set of executable programs. These are the agents.

This platform has been built and tested with Solaris 2.3 and SunOs 4.1, working on Sun Sparc Stations. However, the official delivery platform is for Solaris 2.3.
4.2 The MSM Library

The MSM Library is based on the architecture designed by Tomás Domínguez [Dom92]. This library is described in detail in the MSM-MIX chapter and in the Reference MSM Manual.

4.3 The CKRL Toolbox

The CKRL Toolbox is a set of tools designed for integrating CKRL descriptions in C++ programs. The designed tools are:

- The C++ CKRL Library
- The ckrl2c++ tool
- The runtime ckrl2obj tool

4.3.1 The C++ CKRL Library

A library of C++ classes has been built for mapping the basic elements of CKRL. There are a C++ class for each CKRL-sort and CKRL-property. The user-defined CKRL descriptions (sorts, properties and concepts) are derived from these basic classes, so they inherit all the methods. These basic classes are described in the CKRL Reference Manual.

4.3.2 The static ckrl2c++ tool

This tool (see Figure 4.2) translates a CKRL ontology into a set of classes. Every CKRL concept, sort and property is mapped into a C++ class. These classes allow the user to program with C++ objects which can be easily mapped into plain CKRL descriptions by invoking the method PrintCKRL. This tool also generate the code of several functions for managing the CKRL to C++ conversion.

This tool makes use of different generic sources and tools (see Figure 4.2):

- CKRL ontologies defined in the system for different applications
- CKRL ontologies defined by the user
- The C++ CKRL library described in 3.3.1
- A lexical analyser generator, the flex compiler ver. 2.4.6, by GNU.
- A parser generator, bison ver. 1.22 by GNU.
- A C++ compiler, the g++ ver. 2.6.0 by GNU.

4.3.2.1 The runtime ckrl2obj tool

This tool (see Figure 4.3) allows us to obtain dynamically C++ objects from a CRKL description contained in a message. This process is performed in two steps:

1. ckrl2struct: The CKRL descriptions are mapped onto an array of generic structs by the ReceiveCKRL function. Every element of the array contains a generic struct. These structs can be inspected.
2. struct2obj: The generic structs can be mapped into C++ objects with the CKRL2Obj function.
4.3 The CKRL Toolbox

4.3.2.2 From object to CKRL message

Objects belonging to a class generated by the ckrl2c++ tool can be mapped onto a CKRL description by invoking the method `PrintCKRL` (Figure 4.4), which returns a string. This string can be included, of course, in a message.

4.4 The ADL Compiler

The ADL Compiler (see Figure 4.5) receives an ADL description file and the C++ service functions as input and generates one file for each agent containing the C++ code of the agent. This C++ code is derived from the MSM C++ library and from user-defined C++ functions.

The ADL compiler can use different sources and tools (Figure 4.5):
- MSM C++ Library
- a C++ Service Library
- ADL User Agents Hierarchy
- ADL System Agents Hierarchy
- a lexical analyser generator, the flex compiler ver. 2.4.6, by GNU.
- a parser generator, bison ver. 1.22 by GNU.
- a C++ compiler, the g++ ver. 2.6.0 by GNU.

4.4.1 Interacting with CKRL Toolbox

The ADL compiler can invoke the ckrl2c++, and generates directly from the ADL description of the CKRL classes and the corresponding includes in the makefile. You just need to invoke the adl2c program with the `-e` option.
CHAPTER 5  The MIX Agent Programming Language: ADL

5.1 Description of MIX agents

From the point of view of their behaviour, we can see agents as autonomous entities capable of:

• pursuing some particular and inherent goals. These goals determine the typical behaviour of the agent.
• offering some specialized services to the rest of the world (its environment), allowing problem solving in societies of agents through cooperation.
• demanding services from other agents when they are needed.

The MIX multiagent architecture is an heterogeneous one: that is, different kinds of agents can interact and connect to the network. We consider two agents different when they follow different agent models.

In the MIX platform, we allow the easy implementation of one kind of agent which follows the agent model proposed by Tomás Domínguez [Dom92].

In this chapter we intend to characterize the agents of the MIX architecture from an external point of view. To describe the external structure of an agent, a specialized language has been designed: ADL (Agent Description Language). It is necessary to distinguish clearly between the external structure, that determines the external interfaces, and the internal structure of an agent, determined by the current implementation and built upon the MSM (Multiagent System Model) library. The internal structure has already been defined.

From the external perspective, agents are integrated in a class hierarchy and structured as a set of elements:

• **Services**: functionality offered to other agents.
5.2 Agent hierarchy

- **Goals**: functions that an agent carries out under its own initiative (not to serve a petition from another agent).
- **Resources**: information about external resources (services, libraries, ontologies, groups, etc.) that the ADL programmer needs to know in advance, (statically, before compilation of his ADL file).
- **Internal objects**: data structures that are shared by all the processes that can be launched by the agent to deliver service requests or to achieve goals.
- **Control**: specification of how service requests are handled by the agent.

### 5.2 Agent hierarchy

Agents are organized in a hierarchical structure using the concept of class. It permits an agent to inherit an external structure from the class it belongs to. Every agent has to belong to a parent class. All the elements defined in the parent class are inherited by its successor agents; for instance, its resources, its internal objects, the names of its services and the functions that implement them (unless they are explicitly declared as undefined). This does not mean that children share the same data; it just means that all children will have a similar structure.

Of course, every class (except for the root agent) belongs to another class, and the terminal agent instances (leaves of the hierarchy) will inherit all their ancestors properties.

So, agents and classes have the same structure. The syntactic specification of classes and agents have to comply with the following BNF expressions:

```
<Class_Declaration> ::= CLASS <Class_id> -> <Parent_Class>
                     END <Class_id>

<Agent_Declaration> ::= AGENT <Agent_id> -> <Parent_Class>
                      END <Agent_id>

<Parent_Class> ::= <Class_id> <Class/Agent_Body> ::= [ <Resources> ]
                 [ <Internal_Objects> ]
                 [ <Control> ]
                 [ <Goals> ]
                 [ <Services> ]
```

In the following, we will explore briefly some of the most important external features of our agents.

### 5.3 Services

A typical agent offers services to other agents. Each agent can execute a set of actions (by library functions) that may be requested by other agents. These may be executed in a concurrent or non-concurrent way. The first method implies starting a new process to carry out the task. So, it allows the agent to continue its internal working (taking care of new incoming messages and executing new service requests) while the other processes are running. The second method blocks the agent, stopping its control loop, as a way to assure a perfect control over global agent activity or as a means to improving efficiency.

If the execution of an external program is needed as part of a service, this program has to be encapsulated in a C++ library function.

As has already been mentioned in the previous chapter, at the implementation level, software agents are independent programs that can be executed at any single moment. Services are demanded and delivered through message passing protocols that, at the ADL level, are not visible to the users of the system. One of the protocols that can be used for service petitions is Contract Net. As this protocol, it is necessary for services to have an associated cost function. In this way, agents can receive a service petition according to this protocol, evaluate the cost of serving this petition by using a programmed criterion (function), and send back the result of this function to the agent that demanded the service. The petitioner decides, by analysing the received results who is the agent (or agents) that will get the contract to carry out the service.

The messages that are interchanged for sending a service petition (with previous estimation of cost or not) or for sending back a result (from the service function or from the cost function), require to be structured as a set of arguments according to entities in an ontology. An ontology, a description of concepts in a particular knowledge representation language, the body of these messages would be then composed by chaining instances of these concepts.

```
<Services> ::= SERVICES <Service_Decl> 
            { ; <Service_Decl> }

<Service_Decl> ::= <Service_id> : [ CONCURRENT] <Method>
                 [ COST <Method> ]

<Method> ::= <Function_id>
           [ REQ_MSG_STRUCT <Entity_Spec_List> ]
           [ ANS_MSG_STRUCT <Entity_Spec_List> ]
```

As we have already seen, there are several ways of asking for services. According to the communication mechanism, service petitions can be: synchronous, asynchronous or deferred. On the other hand, a service petition can be addressed to:

- an individual agent.
- a public group of agents, i.e. an alias whose composition (variable, depending on a subscription mechanism) is handled by the YP_Agent.
5.4 Goals

• a private group of agents, i.e. an alias whose composition is included as part of the agent declaration. Private groups are just shorthands useful for programming services that need to make reference on numerous occasions to restricted groups of agents in order to solicit a service from them.

• every agent offering the service in the architecture. In any case, the addressees of a service petition have to be agents registered to the same YP_Agent.

5.4 Goals

Every agent may have proper goals. Each goal can be expressed through any library function, that agents will start up at birth. Special functions can be included for execution upon success or failure of the goal.

Regarding the presence or not of particular goals, we can distinguish two different kinds of agents:

• Reactive agents
  They have no inherent goal. They just work on demand, offering their services to other agents.

• Active agents
  They have some inherent goal function(s) that the agents execute just before being born.

<Goals> ::= GOALS <Goal_Decl>
  { ; <Goal_Decl> }
<Goal_Decl> ::= <Goal_id> : CONCURRENT <Method>
  [ IF_SUCCESS <Function_Call> ]
  [ IF_FAIL <Function_Call> ]

5.5 Resources

It is appropriate to distinguish between two different concepts: environment and resources of an agent. We call environment the dynamic knowledge that any agent has about the services that are offered by other agents and their addresses. This environment is automatically handled by the agent, through communication with its Yellow Pages Agent, as described in the previous chapter.

On the other hand, the resources group all the static knowledge that the programmer at the agent level (the ADL programmer) needs to know. Of course, information contained in these static resources is needed to compile an ADL file, before the execution of any agent included in the ADL specification. Resources can include the following items:

• Required services:
  If the agent has to send service petitions to other agents, they have to be specified along with:
  - the contract policy to follow in case of multiple bids (accepting answers) received from multiple agents when the Contract Net protocol is used.
  - the time-out that applies, after which no more answers are taken into account.
  - the number of retries of the service petition (when no answer had been received before time-out).

If all or some of these items do not appear in the ADL specification of an agent, defaults inherited from the class hierarchy.

• Required public groups:
  A public group is an alias for a restricted set of agents that are significant to the application. Aliases are managed by the YP_Agent, which distributes information dynamically on the composition of these public groups.

• Public groups subscription:
  Public groups are formed by subscription. The agent will inform the YP_Agent of the groups in which it wants to be included.

• Required private groups:
  Restricted groups of agents that are frequently referenced in services programming.

• Required ontologies:
  Ontologies are concept descriptions in a formalized knowledge representation language. These ontologies are provided as a common reference used for knowledge interchange among several agents. The body of messages interchanged by agents can have:
  - a completely free format; in this case sender and receiver have to agree in advance on the exact structure of what is sent or received.
  - a format compliant with any representation language; in this case it is enough to express the structure of an expected or sent message as concepts in a shared ontology.
  - Internally, ontologies are translated into C++ classes. Instances of the concepts in the ontology have to be handled as class instances when the agent services are programmed.

• Required libraries:
  Libraries needed to compile the agent (some of their functions being used for service programming).

<Resources> ::= RESOURCES
  [ REQ_SERVICES : <Required_Services> ]
  [ REQ_PUBLIC_GROUPS : <Public_Groups_List> ]
  [ REQ_PRIVATE_GROUPS : <Required_Private_Groups> ]
5.6 Internal objects

Every agent may have its own internal data (internal objects). Such data can be specified as
instances of a (C++) library classes or as instances of concepts in an ontology. Internal objects are “abstract
data types” implemented as C++ objects. If the specification has been made in form of concepts of an
ontology, the complete ontology is previously translated into C++. Agents will store any relevant information
in these internal objects. They will be used for information storing between successive calls for any
service or for data exchanging among several services.

5.7 Control

Three important components of the internal structure of an agent are its internal data base, its
mailbox and its control function:

• Internal database:
  Every agent has a small internal data base where it records its dynamic environment: YP_Agent address, addresses for those agents that offer services that it may need, addresses of agents in interested public groups, etc.

• Mailbox:
  The mailbox is the place where all the messages addressed to the agent are deposited.

• Control function:
  Determines the overall behaviour of the agent. This behaviour can be modified at the agent program by defining alternative functions to handle the following processes: mailbox management, service policy and destination policy.

The pseudocode of the standard control function of every agent in the MIX platform is:

Send a "Check_in" message to YP_Agent. *Wait for the answer*
Update the internal data base
Start the goal functions (if any)
REPEAT FOREVER
Look at the mailbox for new messages (listen)
IF there are pending messages
  Select one of them according to (mbox_manager function)
  CASE kind of message
    AskForService:
      IF it has to be processed (according to serv_policy)
        Execute the appropriate method
        ENDIF
    AskForServiceWithCost:
      IF it has to be processed (according to serv_policy)
        Execute the appropriate cost method
        ENDIF
    Answer: (goal achieved)
    Execute the method associated to the case of success
    ErrorReport: (goal not achieved)
    Execute the method associated to the case of success
    OTHERWISE
      Error
  ENDIF
ENDIF
ENDFOREVER
In this description we can observe the purpose of the first two aforementioned functions:

- **Mailbox manager:**
determines which message will be the first to be selected for processing. The standard manager use a FIFO criterion: first in, first out. This function can be used to implement a priority scheme.

- **Service policy:**
is used to implement different criteria regarding the attention to a service petition.
The standard service algorithm is an eager algorithm, that tries to serve every external petition. The following are two possible service policies:
  - not to serve the petition if the performance of the system is low (due, for example, to overload).
  - not to serve a petition from outside the local network where the agent is running.

- **Destination policy:**
the standard destination function may send a service petition to every agent known to be offering such a service. The destination function may be used to restrict this policy to select, for example, only the nearest agents or the agents that satisfy certain conditions.

## 5.8 Types of coupling

By default, agents are loosely coupled. This means that inter-agent communication is carried out via message passing. If two agents (Ag1, Ag2) need (for the sake of efficiency) to be strongly coupled, the following section of code should be included in the ADL specification file before the declaration of the agents:

```plaintext
STRONGLY_COUPLED_SOCIETY Ag1, Ag2, ...
EXPORT FROM Ag1: Serv11, Serv12, ... ; FROM Ag2: Serv21, Serv22, ... ;
...
COMMON_OBJECTS: Obj11 FROM Ag1, Obj21 FROM Ag2;
ObjX FROM Ag1, ObjX FROM Ag2, ... ;
END_SOCIETY
```

Only the exported services will be known from the outside. The remaining services will only be offered to the other agents in the strongly coupled society. The calls for internal services are only possible through common objects (that should belong to the very same class). The Common_Objects section in the declaration serves to identify the shared objects among agents. Such objects should have been declared separately for each agent. There is no need for additional modifications of the ADL code when some agents are declared strongly coupled.

Strong coupling forces the use of direct references to internal object instead of using service petitions via message passing.

## 5.9 Directives

The specification of a distributed application in the MIX platform can be sparse in several separate machines. Each one of these files contains an ADL program which is compiled separately.

The header of every specification file must contain mandatorily two directives:

- **YP directive:**
  this is used to specify the place (internet address) where the YP_agent for this application is located and the port that will be used for communications. All the agents declared in the same file are supposed to be attached to the same application and they share the same YP_agent.

- **Domain directive:**
  this is used to identify properly the agents that cooperate in the same application. Agent identifiers are supposed to be unique in the same ADL file. However, it would not be reasonable at all to force them to be unique in the whole net. In order to avoid name collision, agents are known in the MIX-net by the name domain/agent. Domains working together must be distinguished by different names.

Other directives are optional, but, if they exist, they have to be included in the program header:

- **Communication language:**
  All the agents declared in the same domain are supposed to speak the same language. Messages departing from these agents will include a field in their header indicating the language as everybody agrees with a particular communication language. This can be useful for the interpretation of the messages by their addresses.

This directive is optional. In case of absence, the content of messages does not need to be consistent with a particular syntax or semantics. In such case, senders and addresses are responsible for the interpretation of the messages that they receive.

STRONGLY_COUPLED_SOCIETY Ag1, Ag2, ...
EXPORT FROM Ag1: Serv11, Serv12, ... ; FROM Ag2: Serv21, Serv22, ... ;
...
COMMON_OBJECTS: Obj11 FROM Ag1, Obj21 FROM Ag2;
ObjX FROM Ag1, ObjX FROM Ag2, ... ;
END_SOCIETY

Other directives are optional, but, if they exist, they have to be included in the program header:

- **Communication language:**
  All the agents declared in the same domain are supposed to speak the same language. Messages departing from these agents will include a field in their header indicating the language that every agent agrees with a particular communication language. This can be useful for the interpretation of the messages by their addresses.

This directive is optional. In case of absence, the content of messages does not need to be consistent with a particular syntax or semantics. In such case, senders and addresses are responsible for the interpretation of the messages that they receive.
5.9 Directives

- **Include directive:** An indication for the ADL compiler to load an object library.
- **Input directive:** When processing an ADL file, this directive indicates that the input stream continues in another file. Once this second file has been processed, the input is taken from the line following the directive in the first file.
- **Ontology directive:** To indicate an ontology to be used by the agent.

```
<Comm_Lang_Directive> ::= #COMM_LANGUAGE <Language_id>
<Include_Directive> ::= #LIBRARY <File_Spec>
<Input_Directive> ::= #INPUT <File_Spec>
<Ontology_Directive> ::= #ONTOLOGY <File_Spec>
```

---

CHAPTER 6  The Knowledge Level: CKRL

6.1 Knowledge Representation Languages

Lacking standards for knowledge representation, it seems convenient to start from some language specially designed for the purpose of sharing knowledge. There are several possible well known alternatives:

- **KIF** (Knowledge Interchange Format): Designed by the DARPA Knowledge Sharing Initiative, KIF is a prefix version of the language of first order predicate calculus, with various extensions to enhance its expressiveness. Such expressive power goes far beyond the requirements of the MIX project.

- **CKRL** (Common Knowledge Representation Language): Designed by the MLT Consortium (ESPRIT-2154), CKRL was designed in the context of a system intended to offer a variety of symbolic machine learning programs. It is not presented as a general representation language, nor even as a general representation language for all learning systems. However, it has the advantage of greater simplicity.

6.2 A MIX language for knowledge representation

At the early stage of the MIX project during which this work on WP1 has been carried out, it was difficult to evaluate precisely the convenience of using a particular language for knowledge interchange or a language defined by us. As far as we have explored the literature in the field, the requirements on knowledge sharing for hybridation of symbolic and connectionist systems do not seem to be very demanding and do not require a great shift in this tendency in the near future. We have taken as a working hypothesis that the integration of symbolic and connectionist systems does not require the interchange (at least for the moment) of expressions with very complex semantic content. CKRL was finally selected for several reasons:

- It was considered that CKRL offered all the expressive power needed for our purposes, being less complex than other alternative languages.
• CKRL design criteria were very close to the problems we face in symbolic/connectionist hybridation, as learning is the main goal of the connectionist side in most hybrid systems. CKRL was designed to standardize input/output formats among different learning programs in order to offer an adequate platform for comparative purposes. The similarity with our project is clear.

There was, however, a serious drawback in selecting CKRL: it is not widespread in the scientific community. The members of the MLT consortium do not seem to be involved in the international efforts on the standardization of knowledge representation languages. So, we decided to select a reduced subset of the CKRL language (that can be roughly estimated around the 40% of the language) and to implement the translator that we needed into C++ classes.

The degree of commitment to CKRL has been kept to a minimum. The possible inclusion of any other representation language was considered a requirement for the design of the MIX platform. So, a specific field in the messages transmitted among MIX agents was reserved to indicate the language used for coding the body of a particular message. The possibility of transmitting messages in any format, not being constrained by any language, has even been foreseen. In this case, the addressee of the message has to know perfectly in advance the structure of the messages that it will receive from other MIX agents.

The level of coverage with respect to the original CKRL language will be augmented or adapted only on demand, to fulfil the needs from the MIX partners on their hybrid models and applications. Regarding other languages, UPM will study the feasibility of including a (considerably pruned version of) KIF in a future release of the MIX platform.

### 6.3 CKRL 2.0

The last version of CKRL (version 2.0) is described in the document “Final Discussion of the Common Knowledge Representation Language (CKRL)” [C‡93], deliverable D2.3 of the project MLT (Esprit-2154). The introduction to this document explains the methodology followed for the design of the language, and makes evident the objectives of the consortium.

"CKRL was defined in an interactive process involving all the developers of the [machine learning] algorithms. The process consisted of a close analysis of the different representation formalisms used and in the definition of a common terminology (D2.1, 1989) [MLT91]. Only when this terminology was agreed on, could we compare the representation languages and evaluate which structures fulfill the same role and, in this sense, have the same meaning. It is on this comparison and in the choice of essential features covering the diversity of structures present, that CKRL was built (D2.2, 1990) [CCMR90]."

A set of tools were built by the MLT consortium around CKRL:

• **Parsers:**
  A YACC/LEX parser and a PROLOG parser were developed to provide syntactic and semantic checking. The first parser -executable code- is distributed freely (on demand) by LRI (France). The second one has been made public along with the MOBAL system by GMD (Germany).

• **Translators:**
  They take as input a CKRL file and translate just the portion of source code of interest, particularly learning algorithm into a useful internal representation. Most of the translators were built using the aforementioned parsers (in the first case, for example, adding actions to the YACC grammar).

### 6.4 CKRL-MIX

The syntax of CKRL-MIX is described in appendix C. However, CKRL-MIX, being basically a subset of CKRL, will not be fully described here. The interested reader can consult the original language sources. However, a brief explanation will be provided on the CKRL entities covered. Finally, some indications will be given on the many restrictions and on the minimal extension made to CKRL.

A CKRL-MIX program consists of a set of declarations of epistemic entities. These entities have been reduced to the following:

• **Concepts:**
  A concept defines a set of objects. It can be defined by its relevant properties (intensionally) or by a set of instances (extensionally).

• **Instances:**
  An instance is an object of the world that belongs to a class (a concept) and that has values for the relevant properties of the concept (concept structure is inherited by instances).

• **Sorts:**
  A sort defines a range of values that can be used in properties. There are five basic sorts (booleans, integers, real, nominals and strings), and one structured sort (lists). Members of lists can belong to simple or structured sorts. Integers and real sorts can be restricted to intervals.

• **Properties:**
  A property relates a concept to a sort, or an instance to one or more values.

• **Facts:**
  A fact instantiates a property to express a state of the world. An instance of a concept is defined by a set of facts instantiating the relevant properties of the concept.
In the following, we account for the main restrictions imposed on the MIX version of the CKRL language:

- **File headers** have been suppressed. CKRL 2.0 makes use of a header for identification purposes. This header includes the following fields: file name, date, language version, user (knowledge destination), origin and algorithm (knowledge source). In the MIX project, the CKRL-MIX language is used to codify the body of messages, being the source, destination, time of issue and language version already specified in the message header.

- **The inclusion of some CKRL standard entities** has not been considered, in particular, relations, functions, modifiers, target concepts, rules and blocks.

- **Relations**: Relations link together concepts and/or sorts. They correspond to predicates in first order logic. Lacking general predicates, the CKRL-MIX language has the expressive power of the logic $0^+$ (object/attribute/value).

- **Functions**: CKRL permits the specification of function interfaces (not being a language intended for interpretation). This ability is partially included in the MIX environment. The specification of services as methods with a set of input and output arguments belonging to particular CKRL-MIX sorts.

- **Rules**: Rules define possible inferences that can be made when certain facts are known to be true. Their concrete interpretation depends on the addressee of the rule.

- **Blocks**: These are just packages of rules grouped for different purposes.

- **Targets**: Target concepts are necessary in the MLT project to properly identify the learning goals for different algorithms.

- **Modifiers**: Modifiers provide a way to express special features modifying CKRL entities and needed for particular learning algorithms (e.g., they can be used to attach belief values to facts).

- **In CKRL-MIX, strings are allowed as standard sorts. However, it is not possible to define, for example, strings ranging from “A” to “Z” (capital letters).**

- **Sort facets are not considered. These are used in CKRL for changing the default ordering of a basic type, for defining hierarchies of values and for classifying a particular sort according to a hierarchy.**

- **Among the property facets, just the default facet has been preserved. The cardinal facet, indicating that a property may take more than one value, is not necessary after having added lists to the language definition. Neither has the nonsense facet been included, being unnecessary for the MIX version of the CKRL to indicate when a particular property is no more relevant for a particular purpose.**

- **Concepts have also been pruned with respect to CKRL. As in CKRL, they can be defined with their relevant properties (intensionally) and by a set of instances (extensionally). However, they can not be defined in CKRL-MIX by characteristic properties (equivalent to defining the concept through a rule), nor by means of a prototype (a particular instance concept). On the other hand, neither partitions, nor superconcepts are allowed; hierarchies of concepts are therefore not supported.**

- **Facts are restricted to properties (as relations have not been considered).**

### 6.6 Extensions offered by the CKRL-MIX language.

CKRL 2.0 was extended to include lists among its sorts. CKRL allowed disjunction of values. However, the explicit inclusion of lists has been considered necessary for two reasons: syntactic convenience and to provide a more natural expression of the input and output vectors from/to neural networks.
CHAPTER 7  Quick Guide to ADL/CKRL Programming

7.1 Getting started: the foo example

This tutorial will show you in an incremental way how to program agents and services in the MIX platform.

The foo example is very easy: we have two agents Foo_Agent and Client_Foo_Agent.

- The Foo_Agent offers the service foo which only displays “I am the foo service” on the screen when it is requested.
- The Client_Foo_Agent requests the foo service to the Foo_Agent.

We should describe this problem in ADL:

```
#DOMAIN "foo_domain.ckrl"
#YP_SERVER "tcp://madrazo.gsi.dit.upm.es:6050"
AGENT YP_Agent -> YP_Agent
   REQ_LIBRARIES: "ypagent.H"
END YP_Agent

AGENT Foo_Agent -> Basic_Agent
   RESOURCES
      REQ_LIBRARIES: "foo-functions.C"
      SERVICES
         Foo: CONCURRENT foo_service
   END Foo_Agent

AGENT Client_Foo -> Basic_Agent
   RESOURCES
      REQ_LIBRARIES: "foo-functions.C"
      REQ_SERVICES: Foo
   GOALS
      AskForFoo: CONCURRENT AskForFoo
   END Client_Foo
```

We have to specify:
7.2 Beginning with service programming: the foo service

• the DOMAIN: the name of our application. An agent is identified by its name and its domain, so that names of agents from different applications do not conflict.

• YP_SERVER: the YP_Agent is a special agent that offers different services to the rest of the agents. It announces when an agent is born or dies, the network addresses of new agents, etc. You have to ask your system-administrator, where the YP_Agent is running.

• The agents of our “Foo Applications”. You specify an agent basically by the Agent_Class which it belongs to, the services it offers, and the services it demands and the goals. The goals are processes that the agent starts when it is born. We can see these characteristics in Table 7.1.

You don’t need to specify all the properties of an agent, it inherits many from its class. For example, the YP_Agent offers the “check_in” service, but you do not specify it because it is offered by all the agents belonging to the YP_Agent class.

One important thing is that at least one agent of your system must have a goal. If not, all the agents will be waiting for their services be demanded.

• You should implement the services of your agents and specify which function implements it and where the function is. In this description we have supposed that all the functions are in the “foo-functions.C” file. We will learn in the following sections how to implement the functions.

#### Table 7.1 The Foo Example

<table>
<thead>
<tr>
<th>Agent Class</th>
<th>Offered Services</th>
<th>Demanded Services</th>
<th>Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>YP_Agent</td>
<td>YP_Agent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Foo_Agent</td>
<td>Basic_Agent</td>
<td>Foo</td>
<td>AskForFoo</td>
</tr>
<tr>
<td>Client_Foo</td>
<td>Basic_Agent</td>
<td>Foo</td>
<td></td>
</tr>
</tbody>
</table>

7.2.1 Header files needed to be included

The following files, at least, must be included to compile with the MIX library:

• service.H
• message.H

7.2.2 How to declare a service function

You need to declare a function to implement this service, e.g. foo-service.

Your service function must have the following declaration:

```c
void <Function_name> (Service &<var_srv>, Message &<var_msg>)
```

7.2.3 Your first service

You don’t need to know anything else. Just let us implement it!

```c
#include <service.H>
#include <message.H>
void foo-service (Service &srv, Message &msg){
    cout << "I am the foo service" << endl;
}
```

7.3 Demanding a service: The ask-for-foo goal

Usually, your services need to call services from other agents. The ask-for-foo goal will ask foo service from the agent FOO-AGENT. Agents communicate with each other by message passing. Remember again, if at least one agent does not have a goal, your agent society will be passive, and all agents will be waiting for a request.

What do we need to know about requesting services?

• Possible ways of demanding a service
• How to create and compose a message
• How to send a message asking for a service

7.3.1 Possible ways of demanding a service

When you request a service you may want:

• to request a service, stop and wait for an answer. We will call this kind of service synchronous service.
• to request a service and receive no answer. This service will be called asynchronous service.

The foo service only displays “I am the foo service” on the screen when it is requested.

Let start to implement it. You need to know:

• What header files should be included.
• How to declare a service function.
7.3 Demanding a service: The ask-for-foo goal

• to request a service, execute some code, and wait for an answer. This will be called delayed service.

7.3.2 How to create and compose a message

Messages are modelled as a class, the Message class. So, you only need to create an instance of the Message class. At the moment, you only need to know three slots of a message:

• To: recipient agents you deliver the message to.
• ServName: the name of the demanded service.
• Body: the content of the message. Usually, you will send data in the messages.

So, let us create the message:

```
Message m;
m.To = "FOO-AGENT";
m.ServName = "Foo";
m.Body = "";  // This line is not necessary because // the foo service does not wait for any input data
```

7.3.3 Functions for requesting a service

The service is modelled on the Service class. Depending on the kind of service call you perform, you will use different class methods:

• Synchronous service: AskForSyncService
  ```
  int Service::AskForSyncService (Message &m, List<Message> &answers)
  ```
• Asynchronous service: AskForAsyncService
  ```
  int Service::AskForAsyncService (Message &m)
  ```
• Deferred service: AskForDeferredService
  ```
  int Service::AskForDeferredService (Message &m, int &ServReq_id)
  ```

All these methods return SUCCESS if the message is delivered without errors, and, in the synchronous case, if at least one answer is received.

7.3.4 Implementing the ask-for-foo service

The ask-for-foo service is, evidently, an asynchronous service. Here is the complete code of the function:

```
#include <service.H>
#include <message.H>

void AskForFoo (Service &s, Message &msg) {
  Message m;
  // msg is the message requesting the service
  // ask-for-foo received by the agent
  // m is the message we send to request the foo service
```
7.5 Sending and receiving answers: the echo service

Now you can run them. It is usually better to run agents in different windows to watch the output of each agent separately.

What is the right order to run them?

1. The YP_Agent must be operating when you run an agent, therefore execute the YP_Agent first. Every agent sends a message to the YP_Agent at least when they are born, in order to receive the addresses of other agents (the ones which provided its requested services). There is only one YP_Agent for all the applications. Don’t worry if you get the message below. It may mean that there is a YP_Agent running (then go to step 2) or that the YP has just died and the operating system has not yet released the resources, wait a moment and retry it.

```
YP_AgentAG
FATAL ERROR***********MODULE : Channel
FUNCTION: InitListenSocket: bind
Error : Address already in use
```

2. Now you can run the Foo_Agent, because the Client_Agent needs services from this service. If you run the Client_Agent first, it will execute the Error function, and will say that it couldn’t deliver the message to the Foo_Agent.

3. Run the Client_Agent.

You will see different information on the screen and the outputs:

I request the foo service
I am the foo service

To kill the agents, type Control-Z on the YP window or in each window. (You could also program this into your services, via the SuicideInOrder service, but it is too soon to try it!).

7.5 Sending and receiving answers: the echo service

So far, we know how to ask for a service, but we usually need to get a result from a service too.

Now we will implement a useful service: the echo service.

One agent, called MOUNTAIN, will offer this service. This service will display the received data and send them back.

Our agent, MIXY, will have the user-echo goal. This goal will ask the user for data, ask for the service echo to MOUNTAIN, and display the data received from MOUNTAIN.

First, we should write the ADL description:

```
#DOMAIN "echo_domain"
#YP_SERVER "madrazo.gsi.dit.upm.es:6050"
AGENT YP_Agent -> YP_Agent
REQ_LIBRARIES: "ypagent.H"
END YP_Agent
AGENT Mountain -> Basic_Agent
RESOURCES
REQ_LIBRARIES: "echo-functions.C"
SERVICES
echo: CONCURRENT Echo
END Mountain
AGENT Mixy -> Basic_Agent
RESOURCES
REQ_LIBRARIES: "echo-functions.C"
REQ_SERVICES: echo
GOALS
user_echo: CONCURRENT user_echo
END Mixy
```

Now we have to implement the service functions (in the echo-functions.C file).

What should we know before implementing?

- To implement the echo service
  - How to read the content of the requesting message
  - How to send an answer to the agent requesting the service
- To implement the user-echo goal
  - How to get an answer in a synchronous service call

7.5.1 How to read the data of the requesting message

You can directly access the body field of the requesting message. The body data will be interpreted in different ways depending on the knowledge representation language (CKRL, KIF,...) we use. To make the example easier, we will employ no knowledge representation language (the default language, ASCII).

7.5.2 How to send an answer to the agent requesting the service

The class Service offers us a method of sending an answer: SendAnswer, which receives as parameter the message we want to send as an answer.

7.5.3 How to get an answer in a synchronous service call

We receive a list of answers in the second parameter of the AskForSyncService method. We just receive an answer? This is because we can receive more than one answer, as we will see later. But now we need to learn how to manage a list.
Let \( \text{MsgList} \) be a list of messages and \( \text{msg} \) be a message:

\[
\text{List<Message>} \text{MsgList};
\text{Message} \text{msg};
\]

Some useful operations on a list are:

- detect an empty list: \( \text{MsgList}.Is\text{IsEmpty}() \);
- iterate over the elements of the list: \( \text{MsgList}.\text{GoNext}() \);
- get an element of the list: \( \text{msg} = \text{MsgList}.\text{GiveCursor}() \);
- add an element to the list: \( \text{MsgList}.\text{AddElement}() \);
- remove an element from the list: \( \text{MsgList}.\text{DeleteElement}() \);
- number of elements of the list: \( \text{MsgList}.\text{Length}() \);
- destroy a list: \( \text{MsgList}.\text{DestroyList}() \);

### 7.5.4 Implementing the echo service

The \texttt{Echo} function:

```cpp
#include <service.H>
#include <message.H>
void Echo(Service &s, Message &msg){
    Message m;
    cout << "ECHOOOO:" << msg.Body << endl;
    m.Body = msg.Body; // Returns the input message
    s.SendAnswer(m); // Sends the msg to the caller
}
```

### 7.5.5 Implementing the user-echo goal

The \texttt{user-echo} function:

```cpp
#include <service.H>
#include <message.H>
void user-echo(Service &s, Message &msg){
    Message m, m_aux;
    char input_str[80];
    List<Message> answers;
    // Gets the user name
    cin >> input_str;
    // Asks MOUNTAIN for the echo service
    m.To = "MOUNTAIN";
    m.ServName = "echo";
    m.Body = input_str;
    if(s.AskForSyncService(m, answers)== SUCCESS){
        do {
            m_aux = answers.GiveCursor(); // each answer
            // Process the message
        } while(answers.GoNext());
    }
    else
        Error(); // Invoke the standard error function
}
```

Now just run the application as we saw in 7.4.

### 7.6 Requesting deferred services: the user-echo-2 goal

Now we will implement another client of the \texttt{echo} service, the \texttt{user-echo-2} goal, which requests the \texttt{echo} service in a deferred way.

We just need to learn how to get deferred answers.

#### 7.6.1 How to get deferred answers

The \texttt{Service} class provides us a function to get deferred answers:

```cpp
int GetDeferredAnswers (List<Message> &answers, int ServReq_id);
```

The first parameter is the same as the one used to get a synchronous answer. The second is needed because you may have requested several services in a deferred way without getting answers, so you need to specify which service answers you want to get. This parameter is the one the method \texttt{AskForDelayedService} returned to you.

We may add another agent to our ADL description which asks for the \texttt{echo} service but in a delayed way. We need just to add to \texttt{echo.adl}:

```adl
AGENT Mixie -> Basic_Agent
RESOURCES
REQ_LIBRARIES: "echo-functions.C"
REQ_SERVICES: echo
GOALS
user_echo2: CONCURRENT user_echo_2
END Mixie
```

#### 7.6.2 Implementing the user_echo2 goal

Now we can implement the \texttt{use\_echo\_2} function:

```cpp
#include <service.H>
#include <message.H>
void user-echo-2(Service &s, Message &msg){
    Message m, m_aux;
    char input_str[80];
    List<Message> answers;
    // Asks MOUNTAIN for the echo service
    m.To = "MOUNTAIN";
    m.ServName = "echo";
    m.Body = input_str;
    if(s.AskForSyncService(m, answers)== SUCCESS){
        do {
            m_aux = answers.GiveCursor(); // each answer
            // Process the message
        } while(answers.GoNext());
    }
    else
        Error(); // Invoke the standard error function
}
```
7.7 The Contract Net Protocol

The MIX library lets you request a service following the contract net protocol. That is:
1. You (as contractor) demand a service from different agents.
2. The agents return an estimated cost (a bid).
3. The contractor selects some bids and rejects the rest. The selected agents perform the service and return the answers.

There are two methods you should implement (after specification in ADL): cost calculation and costs evaluation:

- **Cost calculation**: how a service estimates its cost. By default, the cost method returns -1.
- **Costs evaluation**: which bids you select. By default, the strategy is select-all.

When you want to request a service, following this protocol, you can use another method of the Service class:

```c++
int Service::AskForSSwithCost (Message &m, List<Message> &answers)
```

Its usage is analogous to `Service::AskForSyncService`.

7.8 Dynamic calling

So far we have requested services from known agents; that is, we specify in our code the name of the server agents. We can also request a service without specifying the recipient agents in the code\(^1\). Our agent will send the message to the known agents which offer this service.

This is possible in MIX architecture thanks to the YP agent, which controls the running of the system.

When an agent is born, it sends YP “personal data”: its name, network address, the offered services, the requested services and some other data. YP answers it with the set of agents that offer some of its requested services as an answer to this **CheckIn** service. YP takes charge of informing there are new agents offering these requested services or if one of these agents has died.

When an agent is going to die, it also informs YP, requesting the **CheckOut** service.

7.9 Calling external programs

MIX library provides you a function `execute` for calling external programs, the **Execute** function.

You are responsible for encoding and decoding the data the external program uses. The parameters are a string with the command you want to execute (e.g. “ls”, “c4.5 -f names”...), and it returns the standard output of the program.

7.10 The Knowledge level: Mixy and Santa Claus

Suppose we now have another problem. Christmas is coming and Mixy wants Santa to deliver a Teddy bear. How can we model this in ADL?

Mixy and Santa are, of course, agents. We can say that Santa offers a service to all the children: if they send a letter containing a wished-toys-list and they have been good children during the year, Santa will send them the toys they want. In ADL, Santa offers a service, which we will call **pensador-gift**. This service requires a **wished-for-gift-letter** as input, and returns a **toy** (or coal according to a Spanish tradition, if the boy was really bad). In this section we will learn how to model concepts (like “toy” and **wished-for-gift-letter**) which cannot be mapped into agents nor services.

ADL agents model the objects of the world and their relationships as concepts. Concepts

1. The field **To** of the message object must be empty. If you are reusing a message object, you can clean this field with the method **DestroyList**: `m.To.DestroyList();`
be defined from their properties, which will define attributes of the concepts. For instance, the wished-for-gift-list concept could have the following properties: the name of the child, his age, the behaviour of this child during the year and the toy he wants to receive. We will distinguish between a concept, which defines a structure, and an instance, belonging to a concept. For example, there will be one concept wished-gift-list, and every child will have his own list, that is, we will model each particular list as an instance of the concept. In our example, Mixy will sent a Mixy-letter, which will be an instance of the concept wished-for-gift-list. We will define an ontology as a group of concepts and instances of these concepts. Ontologies are very useful because they allow the inference of properties and relationships. For example, if Mixy says that he wants a Teddy bear, he does not need to say that this bear have 2 legs, 2 ears, ... because these are common properties to all the Teddy-bears. Agents will only be able to communicate if they share the same ontology. You cannot say to Santa "give me a toy", if Santa doesn’t understand what a toy is.

There are Knowledge Representation Languages to define ontologies, that is, sets of concepts and instances. In ADL, we will use one of these languages: CKRL.

Now, we will learn how to represent concepts and instances in CKRL.

### 7.10.1 Defining concepts
We said before that concepts are defined from their properties. What are the properties of the concepts wished-for-toys-list and toys? We can define a property defining its type (integer, real, string or boolean) and some restrictions (allowed and default values).

The concept “Christmas letter” has the properties listed in Table 7.2.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Type</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>child_name</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>age</td>
<td>integer</td>
<td>0-99</td>
</tr>
<tr>
<td>good_behaviour</td>
<td>boolean</td>
<td>default true</td>
</tr>
<tr>
<td>toy_name</td>
<td>string</td>
<td>default ball</td>
</tr>
</tbody>
</table>

**Table 7.2 Properties of the concept “Christmas letter”**

The concept “Toy” has only one property: the kind of toy (bear, ball, ...).

<table>
<thead>
<tr>
<th>Property name</th>
<th>Type</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>toy_kind</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>

**Table 7.3 Properties of the concept “Toy”**

Now we will learn how to express these properties and concepts in CKRL.

Here we can see the direct mapping from the tables to CKRL:

```ckrl
defproperty boy_name
range (string);
defproperty age
range (integer (0:99));
defproperty good_behaviour
range (boolean)default (true);
defproperty toy_name
range (string)default ("ball");
defconcept Christmas_letter
relevant boy_name, age, good_behaviour, toy_name;
defproperty toy_kind
range (string);
defconcept Toy
relevant toy_kind;
```

### 7.10.2 Declaring Knowledge structures in ADL
We have just defined an ontology which both Santa and Children should share. Here follows the ADL description of the problem.
7.11 Sending Knowledge structures: the goal send_letter

There are some new features:

- **ONTOMETRY:** the name of the ontology and the file where it is defined.
- **REQ_MSG_STRUCT and ANS_MSG_STRUCT:** the concepts you expect to receive in the service request and you will send in the service answer, respectively. Concepts are referred to by the ontology name followed by "::" and the concept name.

Now, as you will have guessed, we have to program the goal send-letter and the service dispenser-gift.

### 7.11 Sending Knowledge structures: the goal send_letter

Mixy should only fill in the letter and send it. We learnt before how to request a service, now we will learn:

- how to manage CKRL objects while service programming
- how to obtain C++ objects from a message

#### 7.11.1 How to manage CKRL objects while service programming.

The CKRL Toolbox generates C++ classes from CKRL descriptions with the same name as the CKRL concept definitions. This translator is invoked by the ADL compiler (you can also execute it independently). So, you can define C++ objects as instances of a class with the CKRL concept name. These objects have a method (PrintCKRL) which can be invoked in order to include the CKRL description of the objects in a message. For example, if you want to create an object by mapping it to a CKRL instance of the concept *Christmas_letter*, it should be created in the following way:

```c++
Christmas_letter myletter(mname.GetString(), 1, SMA_TRUE, "good", "'Teddy_bear'");
```

We have created the object and initialized it at the same time. You could also do this in several steps, using the method `SetValue`:

```c++
Christmas_letter myletter;
myletter.SetValue("child_name", "MIXY");
myletter.SetValue("age", "1");
myletter.SetValue("good_behaviour", "true");
myletter.SetValue("toy_kind", "good");
myletter.SetValue("toy_name", "Teddy_bear");
```

(Note that strings must go enclosed between "'????'" characters, because of CKRL translator.) Once you have created the object in your own way, you should indicate the name of the instance that will be produced when you invoke the method PrintCKRL. We set the instance name with the method `SetID`:

```c++
myletter.SetID("Mixy_letter");
```

#### 7.11.2 How to obtain C++ objects from a message

Now you know how to send an object in a message. In this section you will learn how to get C++ objects from a message you have received.

Receiving objects should be performed in two steps:

1. You read the content message, and obtain a "generic struct" for all kinds of CKRL objects (in one message you can receive more than one instance, from different classes). This is performed by the function `ReceiveCKRL`, which receives as input argument a string with the body of the message with the CKRL description. The number of "generic structs" is stored in the global variable `total_instances` and the "generic structs" are stored in an array `messCKRL`.
2. You inspect this struct and decide which "generic objects" you want to translate into C++ objects. You can inspect the "class" of the struct by accessing the `concept` field of the struct. This translation is performed by the function `CKRL2Obj`, which receives as input argument a generic struct, and returns a pointer to the desired object. You can find out the class of a generic struct through the field `concept`.

Here follows the code of the `send_letter` function. Note that we have indicated that the knowledge representation language of the message is CKRL.

We have also used the method `GetFatherName` of the Service class to obtain the name of the agent. Why not directly write "MIXY"? Because we would have to rewrite this service for every agent (or all the agents would send the letter as Mixy). In this way, this service can be reused.

```c++
void send_letter(Service &s, Message &msg)
{
    Message m, m_aux;
    List<Message> answers;
    Toy *toyptr;
    Chain myname;

    myname = s.GetFatherName();
    myname = m"'"+s.GetFatherName()+m"'"; // CRKL strings format "'xxx'

    System mysys = s.GetSysInformation();
    System msys;
    msys = m"'"+s.GetSysInformation()+m"'"; // CRKL strings format "'xxx'

    m.Body = myletter.PrintCKRL;

    cout << myletter TValue.GetValue();
```

Including the CKRL of your object is very easy: just invoke the PrintCKRL method:

```c++
    m.Body = myletter.PrintCKRL;
```

There is another useful method of the CKRL objects: `GetValue`, which returns the value of a property (depending on the type of the property, this will be the type of the returned value). For example, for showing the value of the property `child_name` of the object `myletter` just type:

```c++
    cout << myletter.boy_name.GetValue();
```

CHAPTER 7 . Quick Guide to ADL/CKRL Programming
7.11 Sending Knowledge structures: the goal send_letter

```cpp
Christmas_letter myletter = myname.GetString(), 1, SMA_TRUE, "good", "Teddy_bear");
myletter.SetID("MixyLetter");
if (myname="BadMixy@santa_domain"){
    myletter.SetValue("good_behaviour", "false");
    myletter.SetID("BadMixyLetter");
    //Asks Santa for the dispenser_gift service
    m.To = "Santa";
    m.FirstName = "dispenser_gift";
    m.Body = myletter.PrintCKRL();
    cout << "I send my christmas letter to Santa: " << endl;
    m.Show();
    cout << endl;
    if (s.AskForSyncService(m, answers) == SMA_SUCCESS){
        m_aux.Show(); //Show a method of the message class
        //To display the message structure (here, the ckrl description)
        ReceiveCKRL(m_aux.Body.GetString());
        cout << "Wow! I have received " << total_instances
            << " gifts from Santa!!" << endl;
        for (int i=0; i < total_instances; i++)
            //total_instances global variable
            if (!(strcmp(messCKRL[i].concept,"Toy")))
                { toyptr = CKRL2Obj(messCKRL[i]);
                    cout << "I receive a gift: " << toyptr->toy_name << "is a ";
                    cout << toyptr->toy_kind.GetValue() << endl;
                }
    } while(answers.GoNext());
    else Error(); //Invoke the standard error function
}
```

7.12 Receiving Knowledge structures: the service dispenser_gift

Santa will fulfill your wishes if you were a good boy but if you were really bad, you will receive coal. You don’t need to learn anymore! Here is the service:

```cpp
void disp_gift(Service &s, Message &msg){
    Message m;
    char buffer[2000];
    Christmas_letter *letterptr;
    Toy gift, gift_extra;
    cout << endl << "Ho, ho, ho!! This is my answer (plus a bonus toy :-))!!" << endl;
    s.SendAnswer(m); //Sends the msg to the caller
}
```
7.13 Running an application with CKRL

If you have a CKRL ontology, the adl compiler will call the CKRL translator. If you do not want this automatic call, you should just call the adl compiler with the \texttt{-n} option. (Remember that the environment variable \texttt{CKRLPATH} must contain the path to the \texttt{ckrl2c} program. Example: if you have installed the translator at \texttt{/usr/local/bin}, then do \texttt{setenv CKRLPATH /usr/local/bin/} under a \texttt{tcsh}.

\begin{verbatim}
> adl2c santa.adl
\end{verbatim}

Now you only need type

\begin{verbatim}
> make -f Makefile.AG
\end{verbatim}

That's all, folks! See the reference manual if you need deeper information.

CHAPTER 8 EXAMPLES

8.1 Examples

This section includes three different examples of hybrid systems written in ADL. The first one shows a hybrid model of an industrial process: Two learning systems (neural and symbolic) are used to predict the value of one variable of the process. CKRL has been used to interchange data among the two learning systems.

The second example implements one of the working modes of the SYNTHESYS system [GAG92, OARG93]. This system trains a connectionist module by comparing its suggestions with those given by a logic module. The third one shows an ADL implementation of the SETHEO system [SE90LS]. SETHEO is a theorem prover which uses a neural module to guide the search for a solution along the problem space search tree.

8.1.1 A Simple Prediction System

This first example (see Figure 8.1) represents a possible model of an intelligent process that predicts the value of one distinguished process variable from the values of a set of observed variables (e.g., acquired through sensors). We use two learning systems (a neural net and an algorithm for the induction of decision trees) to build up this example.

As we can see in Figure 8.1, we have 6 agents and each one provides different services to others. As we are going to use two different learning systems, a learning group is defined. The learning agents are compiled, and result separate executable programs, which could be running in different computers.

- The \texttt{interface} agent allows the user to carry out some actions, such as starting and ending the process.
- The \texttt{collector} agent supplies samples from the process we are modeling, and stores a data history to enable the \texttt{intelligent} systems to learn.
- The \texttt{selector} is the agent who gets new samples from the Collector and asks the learning agents (as a group) for a suggestion.
8.1 Examples

- SNNS and C4.5 (neural and symbolic agents) provide suggestions. From time to time they ask the Collector for the last history, in order to update their knowledge.

- **YP_Agent** performs all check_in and check_out operations.

The next lines describe this application in the ADL language.

```
#DOMAIN "example.ckrl"
#YP_SERVER "tcp://madrazo.gti.dit.upm.es:6050"
#COMM_LANGUAGE ckrl
#ONTOGY example.ckrl

AGENT YP_Agent -> YPAgent
RESOURCES
REQ_LIBRARIES: "ypagent.H"
END YP_Agent

AGENT Interface -> BaseAgent
RESOURCES
REQ_LIBRARIES: "inter_funct.C"
REQ_SERVICES:start_control;
GOALS
Interface: CONCURRENT Inter
END Interface

AGENT Selector -> BaseAgent
RESOURCES
REQ_LIBRARIES: "selec_funct.C"
REQ_SERVICES:give_sample;
suggest
CONTRACT_POLICY eval_suggestion
REQ_MSG_STRUCT example::cost
SERVICES
start_control: CONCURRENT start_control
END Selector

AGENT Collector -> BaseAgent
RESOURCES
REQ_LIBRARIES: "collec_funct.C"
INTERNAL_OBJECTS
history -> History
GOALS
give_history: CONCURRENT GiveHistory
SERVICES
give_sample : GiveSample
ANS_MSG_STRUCT example::Vector;
give_history : GiveHistory
REQ_MSG_STRUCT example::Depth
ANS_MSG_STRUCT example::Vector
END Collector

// *********************
// Learning Agents
// *********************

AGENT SNNS -> BaseAgent
RESOURCES
REQ_LIBRARIES: "learn_funct.C"
REQ_SERVICES: Give_History
SUBSCRIBE_TO: Learning_Group
GOALS
demon : CONCURRENT LearningNeural
SERVICES
suggest: CONCURRENT SuggestionNeural
REQ_MSG_STRUCT example::Vector
ANS_MSG_STRUCT example::Suggestion
COST SNNS_cost_function
REQ_MSG_STRUCT example::Struct_File
ANS_MSG_STRUCT example::Cost
END SNNS

AGENT C45 -> BaseAgent
RESOURCES
REQ_LIBRARIES: "learn_funct.C"
REQ_SERVICES: Give_History
SUBSCRIBE_TO: Learning_Group
GOALS
demon: CONCURRENT LearningSymbolic
SERVICES
suggest: CONCURRENT SuggestionSymbolic
REQ_MSG_STRUCT example::Vector
ANS_MSG_STRUCT example::Suggestion
COST C45_cost_function
REQ_MSG_STRUCT example::Struct_File
```
This example uses some concepts which must be previously defined in CKRL: Cost, Vector, etc. The CKRL file is:

```ckrl
defsort intpos range (integer (0:*));
defproperty h_depth sortref intpos;
defconcept Depth
  relevant h_depth;
// How many data is the system going to use to learn

defsort data range (real (0.0:1.0));
defproperty d1 sortref data;
defproperty d2 sortref data;
defproperty d3 sortref data;
defconcept Vector
  relevant d1,d2,d3;
// Input data

defproperty service_cost sortref intpos;
defconcept Cost
  relevant service_cost;
// Cost of the learning system

defsort identif range (string);
defproperty file_name sortref identif;
defconcept Struct_File
  relevant file_name;
// For file names

defproperty dr sortref data;
defproperty sender sortref identif;
defconcept Suggestion
  relevant sender,dr;
// Suggestion made
```

8.1.2 SYNHESYS

SYNHESYS is a tightly-coupled hybrid system shell comprising of two modules (connectionist and logic modules). The first one is used to speed-up the logic module’s reasoning process. Knowledge is acquired through interaction between both modules. The agent structure for the MSM architecture is shown in Figure 8.2.

When a situation is given to the system, the connectionist module tries to obtain an appropriate decision. If it gets no decision, the logic module is activated in forward-chaining mode to obtain it. This output is used to make the connectionist module learn. Otherwise, if a decision is obtained from the connectionist module, the logic module is also activated, but this time it will work in backward-chaining mode in order to validate the connectionist decision. If a contradiction is found, the right decision (suggested by the logic module in forward mode) will be used for training the connectionist module.

The ADL file for SYNHESYS might be:

```adl
#DOMAIN "synhesys.ckrl"
#YP_SERVER "http://madrazo.gsi.dit.upm.es:6050"
#COMM_LANGUAGE ckrl
#ONTOLOGY synhesys.ckrl
AGENT YP_Agent -> YPAgent
  RESOURCES
  REQ_LIBRARIES: "ypagent.H"
END YP_Agent
AGENT Logic_Module -> BaseAgent
  RESOURCES
  REQ_LIBRARIES: "synhesys.C"
  SERVICES
  Get_Dec_B: CONCURRENT get_dec_b
    REQ_MSG_STRUCT synhesys::decision,
    ANS_MSG_STRUCT synhesys::ok;
  Get_Dec_F: CONCURRENT get_dec_f
    REQ_MSG_STRUCT synhesys::situation
    ANS_MSG_STRUCT synhesys::decision
END Logic_Module

AGENT Connect_Module -> BaseAgent
  RESOURCES
  REQ_LIBRARIES: "synhesys.C"
```

**FIGURE 8.2** Agents structure for SYNHESYS
To make the understanding of how SYNHESYS works easier, we have included the pseudocode for the goal function of the Interaction_Adm agent:

```plaintext
Tune_Conn_Module ()
BEGIN
  REPEAT
    Situation := Call_Service (Get_Sit)
    Net_Dec := Call_Service (Get_Dec, Situation)
    IF Net_Dec = NULL
      Sym_Dec := Call_Service (Get_Dec_F, Situation)
      Call_Service (Train_Net, Sym_Dec)
    ELSE
      OK := Call_Service (Get_Dec_B, Net_Dec, Situation)
      IF NOT OK
        Sym_Dec := Call_Service (Get_Dec_F, Situation)
        Call_Service (Train_Net, Sym_Dec)
      ENDIF
    ENDIF
  UNTIL tuned
END
```

8.1.3 SETHEO

SETHEO (for SEquential THEoremprover) is a Model Elimination theorem prover, both sound and complete for full first-order logic. A neural module is trained to select the shortest path to establish the proof of a theorem. When the prover arrives at a situation where several paths could be tried, the neural module suggests the most promising branch to take.

In SETHEO, the neural module is trained by using some simple well-known theorems. This training phase is finished, the resulting neural net will always be used for proving.

In the particular case that we are going to implement in this section (its agent structure shown in Figure 8.3), let's suppose that we have an Interaction_Manager which gives the theorem to the prover and gets the answer back. From time to time, when this manager detects that better solutions could be found (for instance, when the search trees of several proofs are too deep), it will order the neural module to learn.

The ADL file for this modified SETHEO system is:

```plaintext
#DOMAIN "setheo.ckrl"
#YP_SERVER "tcp://madrazo.gsi.dit.upm.es:6050"
#COMM_LANGUAGE ckrl
#ONTOLGY setheo.ckrl

AGENT YP_Agent -> YPAgent
RESOURCES
  REQ_LIBRARIES: "ypagent.H"
END YP_Agent
```

FIGURE 8.3 Agent society for SETHEO

The ADL file for this modified SETHEO system is:
CHAPTER 9  Current Status of the MIX Platform and Future Work

9.1 MIX Platform 1.1

Release 1.1 of the MIX platform is already available to the public. This platform is currently used by the MIX consortium to test different hybrid models for three applications. The first one is the optimization of a motor/gearbox combination for a turbo-charged engine. The second one consists of the control of a roll-mill in a steel company. The last application pertains to the medical domain: a monitoring system for an intensive care unit.

Apart from the uses foreseen in the workplan, the MIX platform is being used by UPM to develop two additional applications:

- A platform for Natural Language Processing.
- The control of a chemical process in an industrial plant.

The current implementation has some particularities that, although not affecting the overall functionality of the platform, deserve a mention here:

- Regarding data sharing mechanisms between an agent and its services: These are currently implemented through message passing. Threads (lightweight processes) have been considered for the implementation of services and goals instead of UNIX processes.
- Regarding the current implementation of CKRL-MIX: Facts and intensionally defined concepts cannot be interpreted properly unless our agents are able to handle knowledge (Our agents are not in fact knowledge agents, but they can be used to implement them) would be useless, the translation of these entities has not been implemented. (The corresponding CKRL-MIX expressions are parsed correctly, but they are not translated into C++).

As stated in the chapter on CKRL, the platform’s degree of commitment to CKRL has been kept to a minimum. Apart from the improvement mentioned, the level of coverage with respect to the original CKRL language will be augmented or adapted on demand, to

8.1 Examples

AGENT Interaction_Manager -> BaseAgent
RESOURCES
REQ_LIBRARIES: "setheo.C"
REQ_SERVICES: Proving;
Learn
INTERNAL_OBJECTS
history -> History
GOALS
Interface: CONCURRENT interface;
Daemon: CONCURRENT proof_eval
END Interaction_Manager

AGENT NNET -> BaseAgent
RESOURCES
REQ_LIBRARIES: "setheo.C"
SERVICES
Learn : learn
Select_Node: neural
REQ_MSG_STRUCT setheo::history;
REQ_MSG_STRUCT setheo::situation
ANS_MSG_STRUCT setheo::node
END NNET

AGENT TPS -> BaseAgent
RESOURCES
REQ_LIBRARIES: "setheo.C"
REQ_SERVICES: Select_Node
SERVICES
Proving: CONCURRENT proving
REQ_MSG_STRUCT setheo::theorem,
setheo::axiom
ANS_MSG_STRUCT setheo::ok
END TPS
the needs of MIX partners on their hybrid models and applications, whenever the effort needed for the changes is reasonable.

9.2 Future work

Further improvements are being included or considered for inclusion in the next version of the platform:

- Other knowledge representation languages are being considered for inclusion as built-in languages of the platform. In particular, the integration of KIF (Knowledge Interchange Format) language [Gen92], developed also in the framework of the Knowledge Sharing Effort, depends on the availability of a public API for KIF. The second language under consideration is COOL. This object oriented language is integrated in CLIPS (a tool for expert systems development).
- A KQML-compatible set of performatives could be offered. KQML (Knowledge Query and Manipulation Language) [FWW+93] is both, a language and a protocol designed for interchanging knowledge. It has been developed as part of the ARPA Knowledge Sharing Effort. A KQML API would be used to allow the interoperation of MIX and KQML agents.
- A knowledge agent is being implemented in CLIPS capable of interpreting knowledge-oriented performatives.
- A hierarchy of MIX agents could be developed for hybridation purposes.

REFERENCES


REFERENCES


APPENDIX A  Multiagent Architectures for Software Interoperability

A.1 Introduction

There are different choices [Gas] with regard to underlying platforms for implementing a DAI system. We can distinguish the following categories of implementation techniques (see Figure A.1):

- **Distributed Environments**
  A Distributed Artificial Intelligence (DAI) system is, mainly, a distributed system. The distributed processing technologies provide us with a reference model for distributed situations. We will examine some of the emerging international standards in this field. These include ODP, OMA, UI-ATLAS, DCE, Ansaware and Comandos.

- **Object Oriented Concurrent Programming (OOCP)**
  There is a symbiotic relationship between OOCP and DAI [GB]. OOCP has traditionally been applied to developing DAI systems, as they provide some of the basis for encapsulation, object composition, and message-based communication. We will examine some OOCP systems developed with an explicit orientation towards DAI systems. These include Actors, ABCL, Maruichi and Orient-84/K.

- **Multiagent systems**
  We will study agents from a software interoperation point of view. In this sense, we encapsulate software which interoperate. In particular, we will address the systems AOP [Sho] and KQML [HCW].

- **Blackboard systems**
  Blackboard systems [CL92] allow different entities (called knowledge sources) to communicate and synchronize through a shared knowledge structure called a blackboard. Blackboard systems have been applied to many DAI systems. There are also some generic shell blackboard systems, including BB1, GBB or CASSANDRA.

  In the following, we will study with some detail the first three approaches. Blackboard systems, being a distinct approach to DAI from the one selected for the project (multiagent systems...
Distributed Processing provides the basic facilities for communicating two processes in different machines, such as sockets, net addresses, ... Different reference models and protocols (e.g. OSI, TCP/IP) have been developed to achieve this goal. Although many of the problems of physical interconnection have been solved, there is a lack of standards for software interoperability in distributed computing systems.

We will examine some emerging standards:

- **ODP (Open Distributed Processing)**, emergent standard by ISO and ITU, provides an object-oriented reference model for solving the software interoperability problem.

- **OMA (Object Management Architecture)**, developed by Object Management Group (OMG), which provides an architecture for object distributed processing, allowing software interoperability based on object interchanging.

- **UI-ATLAS Framework**, developed by Unix International Consortium (UI). From the architectural point of view, this framework is very similar to OMG’s CORBA. For this reason, this framework will not be presented.

We will also review some platforms for distributed computing:

- **DCE (Distributed Computing Environment)**
- **Ansaware**
- **Comando**
- **Tooltalk**

### A.2.1 ODP (Open Distributed Processing)

ODP [JTCa, ABRa] is an emerging international (ISO and ITU) standard that defines a reference model for open distributed processing (RM-ODP). The RM-ODP standard will be known as both the CCITT/ITU X.900 series of recommendations and the ISO International Standard 10746. It is conceived as positioned as a part of the Applications (7th) Layer of the OSI model, though it includes non-OSI communications to support the model. Indeed Ansaware, the most RM-ODP conformant system currently available, uses TCP/IP communications. The main goals of ODP are:

- portability of applications across heterogeneous platforms
- interworking between ODP systems
- distribution transparency

The RM-ODP standard consists of four parts:

- Part 1: Overview and Guide to Use (ISO 10746-1, CCITT X.901)
- Part 2: Descriptive Model (ISO 10746-2, CCITT X.902)
- Part 3: Prescriptive Model (ISO 10746-3, CCITT X.903)
- Part 4: Architectural Semantics (ISO 10746-4, CCITT X.904)

A system conformant to RM-ODP can be described from five viewpoints:

- **Enterprise Viewpoint**: This point of view describes the social and organizational policies applicable to the ODP systems. It identifies agents (active objects), artifacts (passive objects) and communities (groupings of agents and artifacts).

- **Information Viewpoint**: This describes data and interrelationships of information schemes, relations, constraints and integrity rules.
A.2 Distributed environments

- **Computational Viewpoint:** This is an object based-model for describing interfaces and interactions between distributed applications. Objects can be application objects or ODP infrastructure objects (e.g. trader or type-manager). Interactions between objects can be operational (RPC - remote procedure call) or stream (producer/consumer) oriented. Interfaces in ODP are strongly typed, and inheritance of types leads to subtypes.

- **Engineering Viewpoint:** This defines a model for distributed systems infrastructure, consisting of objects and channels.

- **Technology Viewpoint:** This is used to select particular technologies for implementation, maintenance and testing of ODP systems.

ODP defines different functions and languages for each viewpoint. It defines different functions for management, coordination, repository and security. We now present an overview of two of the repository functions: Type Repository and Trading.

**A.2.1.1 Type Repository Function**

The ODP Type Repository function [JTCb, ABRa] is a registry for interface types. It maintains a type hierarchy and relationships between interface types.

Some of the requirements for an ODP-based Type Repository [BI94] are:

- Dynamic selection of services: the set of available services changes in a dynamic system. There must be a uniform mechanism to select a service and offer a service in a dynamic environment.
- Dynamic binding: since services are selected dynamically, they cannot be bound at the time of compilation and linking.
- Run-time type checking: run-time binding implies a need for run-time interface type checking.

The Type Manager is responsible for providing the type repository function, that is, the means describing and relating the types of operations, interfaces and services among others. A Type Repository should provide two distinct functions:

- **Type description:** [BI94] provides the means to describe and compare types. An ODP type description consists of:
  - *data* types (basic or constructed types), allowing opaque types and polymorphism (generic types).
  - *operations*: syntactic and semantic description of the permitted operations by asking for a type, announcing a new type, ...
  - *interfaces* for accessing objects. There will be operational and stream interfaces.

- **Type management:** the operations supported by the Type Manager can be divided into two basic categories:
  - *repository operations* for the addition and deletion of types and relationships between types.
  - *query operations* for browsing the stored types and for type matching.

All these operations [BI94] are designed to be accessed by both user programs and infrastructure objects (in particular, the Trader, Figure A.2).

![Figure A.2](image-url)

FIGURE A.2 Main clients for the ODP-based Type Manager

Comparing [BI94] some of the existing standardized types models (ASN.1 [oBN], ACT...
[dMR] and GDMO [oMIPGftDoMO], to ODP Type Manager, the last one is more general than the previous ones, being possible to map them into ODP types.

### A.2.1.2 Trader

The trading function [ISO94, JTCb] allows the advertisement and discovery of interfaces. It provides a dynamic selection of service providers. A trader is an object that acts as a mediator between service providers and service consumers (Figure A.3), enabling exporters to advertise their services and importers to match their needs against the available services. According to this figure, we can distinguish the following sequence of interactions:

1. The exporter gives the trader information on the available service. These services contain a description of the service together with the location of an interface at which the service is available and are stored by the trader in a database.
2. The importer asks the trader for a service with certain characteristics.
3. The trader will match this service request against the service offers contained in its database, and answers the importer with the locations of the most appropriate service offers (if one exists).
4. The importer interacts with the service provider of the best offer.

Both importers and providers will eventually define policies to select their partners, reducing the set of potential providers or requesters of a service, respectively. The trader will also have its own policies for accounting, acceptance of service offers and service requests, storage of the offers, formal search of the offers and resource consumption, among others.

It is important to distinguish between a service that a provider can facilitate, a service request and a service offer:

- A **service** is an instance of a service type, which determines the service interface (that is, the set of possible operations at the interface). However, instances of the same service type may have different properties (quality of the service, constraints, service cost, response time, etc.).
- A **service offer** will contain information about a service being traded. In addition to some service type and property values, a service offer can specify property values (e.g., expiration date of the offer, ...). The service property values can be static or dynamic, having to be updated in the latter case. This service offer will be classified by the trader in groups (texts) attending to common characteristics, in order to speed the matching process.
- A **service request** will contain information on a requested service, including requested service types, required service properties and service offer values, and some selection preferences.

Trading is a dynamic activity: the exporter can retract, modify or replace offers, the trader can modify dynamically the trading properties, ...

Since concentrating all the information in a single element inside a distributed system (the trader in this case) should be avoided, and the offer space can be very large, hierarchical federations of traders [ISO94, BRa] have been established. Each trader can handle both local and remote services, maintaining addresses of the adjacent traders. Some additional policies of the trader will allow the importers and exporters to perform local and remote search, linking to other traders, etc.

### A.2.2 OMA

The Object Management Group (OMG) is an international consortium, created in 1990, whose primary objective is the promotion of object-oriented technology. It has defined an Object Management Architecture (OMA) [Sol90], which describes the basic components of a distributed object-oriented environment and its relationships. The basic components are:

- A core Object Model for defining the objects.
- An Object Request Broker (ORB) for communicating the objects, via message-passing.
- A set of Object Services, which extend the functionality of the ORB.
- A set of design tools and facilities.

We will describe here mainly the ORB and some of the Objects Services.

The specification CORBA (Common Object Request Broker Architecture) [OMG91] of
A.2 Distributed environments

MIX (ESPRIT 9119): A Multiagent Architecture for Symbolic-Connectionist Integration

by the Object Management Group (OMG) provides the mechanisms by which objects make requests and receive responses transparently. The ORB (Figure A.4) provides interoperability between applications on different machines in heterogeneous distributed environments and interconnects multiple object systems.

A CORBA object system is defined as a collection of objects providing services to clients, the clients being any entity capable of asking for a service. An object is an identifiable, encapsulated entity that provides one or more services that can be requested by a client.

Figure A.4 Structure of Object Broker Interfaces

IDL (Interface Definition Language) defines the types of objects by specifying their interfaces. An interface consists of the set of permitted operations and the parameters of these operations. In addition to the interfaces defined statically in IDL, there is an Interface Repository Service that provides persistent objects that represent the IDL information in a form available at runtime. The client can invoke a dynamic invocation interface, common to all the objects, or the specific object interface defined in IDL.

Relationship to ODP: Some OMA services and ODP services are related:

• ODP Type Manager [Tho92]:
  - OMA Object Properties Service: Object Properties are arbitrary named values associated with an object. An object using the properties interface can associate information with its state, for example, a title or a date. Properties are the dynamic equivalent of attributes; attributes are defined in CORBA.
  - OMA Object Interface Repository: supports runtime access to IDL. It supports adding, locating and searching, retrieving, and dynamically updating definitions; (type) checking object invocation; and dynamic construction of object invocations.
  - OMA Object Relationships Service: The Object Relationships Service is provided for creating, deleting, navigating, and managing relationships between objects. Relationships support modelling object associations and semantics.
  - OMA Object Query Service: Queries are operations on sets or collections of instances that have a predicate-based, declarative specification, and that may be sets or collections of objects. The emphasis is on fine-grained object access.

• ODP Trader Service: the ORB Core is in some aspects very similar to trader, connecting clients and servers.

Since significant portions of RM-ODP overlap with OMG specifications, both organizations have agreed to work cooperatively [OMG]. ODP could adopt OMG’s CORBA, IDL, and Object Services, while OMG could adopt ODP’s Trader standard.

There are several implementations of CORBA [Hat]. We can point out versions from DEC and Orbeline, on different platforms (Unix, Windows, Macintosh).

A.2.3 DCE

The Open Software Foundation’s Distributed Computing Environment (DCE) is a set of software components providing a middleware platform for the construction and use of distributed applications.

DCE [Mau, ABRb] consists of multiple components (Figure A.5), originally from different vendors, which have been integrated to work closely together. They are the Remote Procedure Call (RPC), the Cell and Global Directory Services (CDS and GDS, white pages), the Security Service,
DCE Threads, Distributed Time Service (DTS) and Distributed File Service (DFS).

**Relationship to ODP:** this relationship is analysed in [ABRc]. The conclusion is “broadly, DCE and RM-ODP have a similar computational model. Although DCE does have the concept of an interface, it is not directly represented at the application development level. DCE diverges from the RM-ODP engineering model as a result of its lack of node managers, factories and clusters. While DCE provides a distributed environment, it is not a support environment which is conformant to RM-ODP, as it does not provide many of the RM-ODP functions”. Despite these objections (such as the lack of a yellow pages service instead of the DCE white pages service), some functions such as the Trader have been built on DCE [BB].

**Relationship to CORBA:** [Mau]: DCE provides a lower-level programming model than CORBA’s. Instead of interoperability based on objects, DCE provides interoperability based on RPCs. DCE may be used as the base communications mechanism in future CORBA products. Actually, DCE also defines an IDL for interface definition, different from CORBA’s. There is also a relationship [AS] between CORBA and DCE IDLs.

**A.2.4 ANSAware**

ANSAware [BRb, ABRb] is the realization of the Advanced Networked Systems Architecture (ANSA), an architecture for networked computer systems to support distributed applications, and has been developed by the research organisation Architecture Projects Management (APM) Limited.

ANSAware consists of multiple components, originally from different vendors, which have been integrated to work closely together. They are the Remote Procedure Call (RPC), ANSA Threads, Notification Service, Factory and Node Manager and Trader (yellow pages).

**Relationship to ODP:** This issue is analysed in [BRb]. Although ANSAware is the most ODP conformant system currently available, it provides a very limited set of RM-ODP functions and has no security mechanisms.

**Relationship to DCE:** This relationship is studied in [ABRb]. The results of this analysis is that, although both distributed environments share common layers (see Figure A.5 and Figure A.6), they are very different. These differences come from the different starting points: integrating existing technology (DCE) and designing an innovative architecture (ANSAware).

**A.2.5 The Comandos Distributed Application Platform**

The Comandos Distributed Application Platform [VCdPE92] provides a conceptual model of a distributed environment, encompassing both computation and data management. Comandos has been developed in the framework of two ESPRIT projects (Comandos-1 (1986-1989) and Comandos-2 (1989-1992))

The architecture of the Comandos platform is shown in Figure A.7. It consists of...
A.2 Distributed environments

- The Comandos system is structured in two layers: The virtual machine, which is independent of any supported programming language, and a set of language-specific run-time systems.
- A set of application services and management tools.

Relationship to DCE: there is an ESPRIT project, HARNESS [Con91] which has, as one of its objectives, the integration of the Comandos platform within the overall DCE framework. The integration depends on the evolution of DCE towards an object-oriented base.

FIGURE A.7 The COMANDOS Architecture

APPENDIX A Multiagent Architectures for Software Interoperability

MIX (ESPRIT 9119): A Multiagent Architecture for Symbolic-Connectionist Integration

Relationship to OMA: Both architectures are very close, although they diverge [VCdPE92] in the object services.

Relationship to ODP: Some of the Applications Services (Type Manager and Object Data Management System) are related to the ODP's Type Manager, and some of the management tools (DDS, Distributed Directory Service, X.500 based) are related to the ODP's Trader.

A.2.6 Sun's Tooltalk

The ToolTalk [SM] service enables independent applications to communicate with each other without having direct knowledge of each other. Applications create and send ToolTalk messages to communicate with each other. The ToolTalk service receives these messages, determines the recipient then delivers the messages to the appropriate applications. ToolTalk is designed to make it easy to add a messaging interface on any application, regardless of the application.

DOE is SunSoft's next-generation application environment, and includes SunSoft's implementation of OMG standards such as CORBA[OMG91]. DOMF is SunSoft's CORBA-compliant ORB. ToolTalk was designed and delivered in 1991, before the OMG's CORBA spec was defined. Thus, ToolTalk itself is not a CORBA-compliant Object Request Broker.

A.3 Object Oriented Concurrent Programming (OOCP)

OOCP [Bri] is a methodology for integrating Object-Oriented Programming (OOP) with concurrency. It has been successfully applied to different fields: distributed operating systems, distributed artificial intelligence, distributed simulation, distributed databases, office information services, real-time systems and distributed process control.

A comparative summary [Dom92] of several object-oriented concurrent models is shown in Table 9.1.

A.4 Intelligent multiagent platforms

The term agent has been widely used [Fon, BW, WJ] with different meanings. We will use the this section from a software interoperability point of view [Gen, GK94, GS94]. Agents will encapsulate software which interoperates.

Why do we need “intelligent agents” for communication? Is it not enough with the interoperability provided by distributed processing? We can answer that distributed processing techniques give us a support for communication, but these do not standardize the message semantics: how can interchange facts or rules, how a conversation can be held between two or more agents, etc. is the goal of, for instance, KQML (Knowledge Query and Manipulation Language). For its
A.5 Agent Oriented Programming (AOP)

Agent Oriented Programming (AOP) [Sho] is a new programming paradigm which can be viewed as a specialization of object-oriented programming. Instead of using the objects as the basic unit, it uses agents, which have associated states (beliefs, agreements, etc.). Both, objects and agents, are processed via message-passing and invoke methods to answer these messages, but the agents will have a set of preestablished messages (inform, provide, request, etc.).

A complete AOP system should include the following basic components:

- a formal language to describe in a straightforward way the mental state of the agent. Shoham defines this state from three basic parameters: beliefs, capabilities and commitments.

**TABLE 9.1 Comparative summary of object-oriented concurrent models**

<table>
<thead>
<tr>
<th>Message processing</th>
<th>MSM</th>
<th>ORIENT/84</th>
<th>ABCL/I</th>
<th>MARUICHI</th>
<th>ACTORS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message manager</td>
<td>By priorities</td>
<td>By restrictions</td>
<td>Message interpreter</td>
<td>Order of receipt</td>
<td></td>
</tr>
<tr>
<td>Communication mode</td>
<td>Point to point Groups Broadcast</td>
<td>Point to point Broadcast</td>
<td>Point to point</td>
<td>Point to point Broadcast</td>
<td></td>
</tr>
<tr>
<td>Synchronization</td>
<td>Synchronous Asynchronous Deferred</td>
<td>Synchronous Asynchronous Deferred</td>
<td>Synchronous Asynchronous Deferred</td>
<td>Asynchronous</td>
<td></td>
</tr>
<tr>
<td>Topology</td>
<td>Dynamic</td>
<td>---</td>
<td>---</td>
<td>Dynamic</td>
<td>---</td>
</tr>
<tr>
<td>No. of services</td>
<td>Several</td>
<td>Several</td>
<td>Several</td>
<td>Several</td>
<td>One</td>
</tr>
<tr>
<td>Blocked processes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

AOP (Agent Oriented Programming) proposes a reduced set of speech acts, and supplies a way to model the mental states of an intelligent agent.

Distributed Artificial Intelligence will study not only individual interactions but also social interaction: how an agent society can be coordinated to reach a common goal [BG88, Jen].

We will examine two well-known agent paradigms: Agent Oriented Programming (AOP) [Sho] and Knowledge Query and Manipulation Language (KQML) [HCW, TFMc, TFMb, TFMa, Gen, Gene94LF94] which are classified as knowledge coordination languages [GM94].

A.5.1 Knowledge Query and Manipulation Language (KQML)

KQML (Knowledge Query and Manipulation Language) [HCW, FWW93, TFMb, TFMa] is both a language and a protocol designed for interchanging information and knowledge.

KQML is one of the main results of the ARPA Knowledge Sharing Effort (KSE). This is a consortium dedicated to developing standards in order to facilitate the sharing and reuse of knowledge bases and knowledge based systems. The working groups of KSE are:

- The Interlingua Group, concerned with the translation of different representation languages.
- The KRSS Group (Knowledge Representation System Specification), concerned with developing common constructs with families of representation languages.
- The SRKB Group (Shared, Reusable Knowledge Bases), concerned with facilitating consensus on the contents of sharable knowledge bases.
- The External Interfaces Group, concerned with run-time interactions between knowledge based systems and the environment (e.g. databases).

A.5.1.1 KQML Protocols

KQML allows different communication protocols on a client-server basis:

- **Synchronous communication with single answer**: (Figure A.8) the client requests a service and becomes blocked waiting for the answer.
- **Synchronous communication with multiple answer**: (Figure A.9) the client requests a service obtaining more than one answer; it waits until an answer is received, and then demands each additional answer in the same way. The received answers can be different or partial answers which are improved incrementally, etc.
A.5 Agent Oriented Programming (AOP)

Asynchronous communication: (Figure A.10) The client requests a service and does not know how many answers will be sent by the server, nor when. After sending the request, the client will not be blocked, thus being able to carry out other tasks.

A.5.1.2 The KQML language

KQML is a communication language organized in three layers: content, message and communication.

- The Communication level: deals with the lower levels of communication (transport protocol, recipient and sender, etc.).
- The Message level: is the main level of KQML and deals with the logic of the communication, that is, with the speech acts. This level codifies the kind of messages which are sent. It determines the possible interactions between agents, allowing the specification of the protocol for delivering the message (see 2.5.1.1) and the set of specified speech acts (command, assert, ask, ...). Since the content is opaque in KQML, this level should specify some characteristics of the content: its language, ontology, ... in order to be able to analyze and deliver the message without accessing the content. The last version of KQML [TFMa] distinguishes the following performatives (or standardized speech acts):
  - Basic query performatives: evaluate, ask-if, ask-in, ask-one, ask-all
  - Multi-response query performatives: stream-in, stream-all
  - Response performatives: sorry, reply
  - Generic informational performatives: tell, achieve, cancel, untell, unachieve
  - Generator performatives: standby, ready, next, rest, discard, generator
  - Capability-definition performatives: advertise, monitor, export, import
  - Networking performatives: subscribe, unregistration, forward, broadcast, route
- The Content level: KQML does not make any assumption about the content of the messages, which can be in any knowledge language or format (KIF, LOOM, ASCII, Binary)

When two agents communicate, they have to agree upon the communication language that is used.

A.5.1.3 Specific Agents

Although KQML does not define a particular architecture, it has considered two particular architectures from the beginning. It is necessary for the running agents to know when an agent is born and how a message to an agent can be delivered. These agents are routers and facilitators.
• **The Router:** (Figure A.11) its task is to connect a KQML agent with the network. It is only concerned with the addresses of sender and receiver. Since agent and router are different processes, an interface between both is provided, KRIL (KQML Router Interface Library).

![KQML Router](image)

**FIGURE A.11** KQML Router

• **The Facilitator:** (Figure A.12) provides different services to the rest of agents (in fact, to the routers of the agents):
  - delivery of messages with incomplete addresses
  - registry of offered services
  - content based routing
  - brokering of information in the Contract Net Protocol

Davies [DE] has integrated AOP with KQML, by modifying the interpreter of AGENT-0 in order to allow commitment rules referring to KQML messages.
APPENDIX B  Syntax of the
ADL Language

B.1  ADL header

<ADL_Program> ::= <Program_Header> <Program_Body>

<Program_Header> ::= <Global_Directives> [ <File_Directives_List> ]


<Domain_Directive> ::= #DOMAIN <Domain_id>

Semantic: All the domains (ADL files) cooperating in a particular application (being served by the same YP_Agent) have to be declared with different identifiers. The domain identifier has been provided to avoid a possible conflict of agent names (thus all agents declared by different authors in their respective ADL files to have the same name).

<YP_Directive> ::= #YP_SERVER <YP_address>

Semantic: YP_address must be an existing DNS or IP address. If a port number is not present, port 6050 is assumed.

<Comm_Lang_Directive> ::= #COMM_LANGUAGE <Language_id>

<Include_Directive> ::= #LIBRARY <File_Spec>

Semantic: source libraries have to be compiled into object libraries before compiling an ADL file.

<Input_Directive> ::= #INPUT <File_Spec>

Semantic: When processing an ADL file, this directive indicates that the input stream continues in another file. Once this second file has been processed, the input is taken from the line following the directive in the first file. Directive nesting is handled.

<Ontology_Directive> ::= #ONTOLOGY <File_Spec>
B.2 Program body

Semantic: If just an identifier is given, the filename is composed adding the name of the knowledge representation language to the identifier.

B.2 Program body

<Program_Body> ::= <Program_Section> | { <Program_Section> }
<Program_Section> ::= <Strongly_Coupled_Soc_Decl>
                  | <Class_Declaration>
                  | <Agent_Declaration>

B.3 Strongly coupled societies

<Strongly_Coupled_Soc_Decl> ::= STRONGLY_COUPLED_SOCIETY <Agents_List>
                               [ <Export_List> ] [ <Common_Objects_List> ] END_SOCIETY

Semantic: Strongly coupled societies have to be declared before the agents they include. To avoid conflict names, their services have to be given different names.

<Export_List> ::= EXPORT FROM <Agent_id> : <Services_List>
                 { FROM <Agent_id> : <Services_List> }
<Common_Objects_List> ::= COMMON_OBJECTS :
                          <Objects_Identity_List>
                          { ; <Objects_Identity_List> }
<Object_Identity> ::= <Object_id> FROM <Agent_id>

Semantic: Internal objects declared as identical are treated as one single object of the society. These objects have to belong to the same class.

B.4 Agents and classes

<Class_Declaration> ::= CLASS <Class_id> -> <Parent_Class>
                       <Class/Agent_Body>
                     END <Class_id>

Semantic: Class identifiers in the same ADL file have to be different.

<Agent_Declaration> ::= AGENT <Agent_id> -> <Parent_Class>
                       <Class/Agent_Body>
                     END <Agent_id>

Semantic: Agent identifiers in the same ADL file have to be different.

B.5 Resources

<Resources> ::= RESOURCES
              [ REQ_LIBRARIES : <Library_Files_List> ]
              [ REQ_SERVICES : <Required_Services> ]
              [ SUBSCRIBE_TO : <Public_Groups_List> ]
              [ REQ_PUBLIC_GROUPS : <Required_Public_Groups> ]
              [ REQ_PRIVATE_GROUPS : <Required_Private_Groups> ]
              [ REQ_ONTOLOGIES : <Ontology_Files_List> ]

<Required_Services> ::= <Required_Service>
                      { ; <Required_Service> }
<Required_Service> ::= <Service_id>
                    [ CONTRACT_POLICY <Contract_Policy_Spec> ]
                    [ TIMEOUT <Timeout_Spec> ]
                    [ RETRIES <Retries_Spec> ]

<Contract_Policy_Spec> ::= <Method_id>
<Timeout_Spec> ::= REAL_N
<Retries_Spec> ::= INT_N
<Required_Private_Groups> ::= <Required_Private_Group>
                            { ; <Required_Private_Group> }
<Required_Private_Group> ::= <Group_id> : <Agents_Address_List>

Semantic: Private group names in the same agent/class can not be duplicated. Agents can be defined by name and optionally the domain (at agent level, name@domain). If a domain is not supplied the system supposes the agent’s domain.

<Ontology_Files_List> ::= <File_Specs_List>

Semantic: Ontology files included in an ontology directive need not be mentioned.

B.6 Internal objects

<Internal_Objects> ::= INTERNAL_OBJECTS
                     <Entities/Objects_Decl>
                     { ; <Entities/Objects_Decl> }

Semantic: Internal objects can be declared inside an agent definition, using a class definition is recommended instead, and then use inheritance to build up an instance of the
B.7 Control

<Entities/Objects_Decl> ::= <Objects_List> -> <Entity/Object_Spec>

Semantic: Objects are members (instances) of C++ classes (in a declared library) or entities (in a declared ontology). Object identifiers in the same agent should have different names.

<Entity/Object_Spec> ::= <Entity_Spec> | <Object_Spec>

<Entity_Spec> ::= <Ontology_id> :: <Entity_id>

Semantic: The entity identifier has to be included in the specified ontology. This ontology has to be declared in a directive or required in the agent environment.

<Object_Spec> ::= <Object_Class_id>

Semantic: The entity identifier has to be included in the specified ontology. This ontology has to be declared in a directive or required in the agent environment.

B.7 Control

<Control> ::= CONTROL

[<Mailbox_Manager_Declaration>]

[<Service_Policy_Declaration>]

[<Destination_Policy_Declaration>]

-Mailbox_Manager_Declaration> ::= MBOX_MANAGER : <Function_id>

<Service_Policy_Declaration> ::= SERV_POLICY : <Function_id>

<Destination_Policy_Declaration> ::= DEST_POLICY : <Function_id>

Semantic: All the control functions should exist in library files (included through directives or required in the agent environment). Standard functions are provided for control when these functions are not declared.

B.8 Goals

<Goals> ::= GOALS <Goal_Decl>

{ ; <Goal_Decl> }

<Goal_Decl> ::= <Goal_id> : CONCURRENT <Method>

[ IF_SUCCESS <Function_Call> ]

[ IF_FAIL <Function_Call> ]

Semantic: Referenced functions should exist in library files.

B.9 Services

<Services> ::= SERVICES <Service_Decl>

{ ; <Service_Decl> }

B.10 Lists

<Appendix B : Syntax of the ADL Language

B.11 Files

<Files> ::= STRING

| <File_id>

Semantic: If specified as a string, relative or absolute paths can be given. If an identifier is used, the current directory is assumed as default directory. A file extension is added depending on the context: the name of the communication language ".<Language_Id>" for ontologies and ".C" for libraries.
B.12 Addresses

<YP_address> ::= YP_ADDRESS

Semantic: DNS or IP address of the YP_agent. If a full address is not given, the address of the node where the ADL file is being compiled is assumed. By default, port 6050 is used.

<Agent_Address> ::= AG_ADDRESS

Semantic: By default, the domain of the current ADL file is assumed.

B.13 Identifiers

<Agent_id> ::= IDENTIFIER
<Class_id> ::= IDENTIFIER
<Domain_id> ::= IDENTIFIER
<Entity_id> ::= IDENTIFIER
<File_id> ::= IDENTIFIER
<Function_id> ::= IDENTIFIER
<Goal_id> ::= IDENTIFIER
<Group_id> ::= IDENTIFIER
<Language_id> ::= IDENTIFIER
<Object_id> ::= IDENTIFIER
<Object_Class_id> ::= IDENTIFIER
<Ontology_id> ::= IDENTIFIER
<Ontology_Element_id> ::= IDENTIFIER
<Service_id> ::= IDENTIFIER

Semantic: Identifiers for the same non-terminal symbol should be different when used in the same context (agent, class of file).

B.14 Lexical tokens

alpha ::= [a-zA-Z]
digit ::= [0-9]
alphanum ::= ( {alpha} | {digit} )
symbol ::= ( {alphabet} ( alphanum | _ ) )* digits ::= ( {digit}* )

B.15 Keywords

The following identifiers are reserved (keywords), and should not be used as user-defined identifiers:

AGENT  ANS_MSG_STRUCT  CLASS
COMMON_OBJECTS  CONCURRENT  CONTRACT_POLICY
CONTROL  COST  DEST_POLICY
END  END_SOCIETY  EXPORT
FROM  GOALS  IF_FAIL
IF_SUCCESS  INTERNAL_OBJECTS  MBOX_MANAGER
REQ_ONTOLOGIES  REQ_LIBRARIES  REQ_MSG_STRUCT
APPENDIX C Syntax of the CKRL-MIX Language

In the following, the syntax (and partially the semantics) of the CKRL-MIX language is described in BNF expressions. The reader should be aware of the following: while pruning the original CKRL syntactic description, special care was taken to keep description changes as few as possible. This criterion forces the maintaining of some non-terminal symbols that are redundant.

C.1 Declarations

```
<CKRL file> ::= { <CKRL declaration> ; }+
<CKRL declaration> ::= <sort definition>
  | <property definition>
  | <concept definition>
  | <instance definition>
  | <fact declaration>
```

C.2 Identifiers

```
{id dec} ::= IDENTIFIER

Semantic: Identifiers in a CKRL-MIX file have to be unique. A declaration has to precede any reference to an identifier.

{id ref} ::= IDENTIFIER

Semantic: The reference to an identifier has to belong to the same type appearing in its declaration. It means that an identifier for <sort id ref> has to be previously declared in <sort id dec>.
```

C.3 Sorts

```
<sort definition> ::= defsort <id dec> <sort descr>
```
C.3 Sorts

\[
\begin{align*}
\text{<sort id dec> ::= <id dec>}
\text{<sort id ref> ::= <id ref>}
\text{<sort> ::= sortref <sort id ref>}
\text{| <sort descr>}
\text{<sort descr> ::= <range descr>}
\text{| <list descr> <list sort facets>}
\text{<list descr> ::= list ( <standard sort> )}
\text{<list sort facets> ::= [ <list size facet> ]}
\text{<list size facet> ::= size ( <size descr> )}
\text{<size descr> ::= INT_N}
\text{| *}
\text{<range descr> ::= range ( <standard sort> )}
\text{<standard sort> ::= <nominal range> | <integer range> | <real range>}
\text{| <boolean range> | <string range>}
\text{<nominal range> ::= nominal ( <nominal value list> )}
\text{<nominal value list> ::= <nominal value list> , <nominal value>}
\text{| <nominal value>}
\text{<nominal value> ::= <value id dec>}
\text{<value id dec> ::= <id dec>}
\text{<value id ref> ::= <id ref>}
\text{<integer range> ::= integer [ <integer interval> ]}
\text{<integer interval> ::= [ INT_N : <max integer> ]}
\text{| [ * : INT_N ]}
\text{<max integer> ::= INT_N | *}
\text{<real range> ::= real [ <real interval> ]}
\text{<real interval> ::= [ REAL_N : <max real> ]}
\text{| [ * : REAL_N ]}
\text{<max real> ::= REAL_N | *}
\text{<string range> ::= string}
\text{<boolean range> ::= boolean}
\text{<interval> ::= <integer interval> | <real interval>}
\end{align*}
\]

C.4 Values

\[
\begin{align*}
\text{<simple value> ::= INT_N | REAL_N | <value id ref>}
\text{| STRING_N | <boolean value>}
\text{<boolean value> ::= true | false}
\text{<value> ::= <simple value>}
\text{| ["* <list value> "]}
\text{Semantics: A referenced value has to be a legal value of the specified sort.}
\text{<list value> ::= <list value> , <simple value>}
\text{| <simple value>}
\text{<d_value> ::= <value>}
\text{| ( interval <interval> )}
\text{Semantics: Allowed values are simple values, lists and intervals (integer or real).}
\text{<concept value> ::= <d_value>}
\end{align*}
\]

C.5 Properties

\[
\begin{align*}
\text{<property definition> ::= defproperty <property decl>}
\text{<property decl> ::= <property id dec> <property descr>}
\text{<property id dec> ::= <id dec>}
\text{<property id ref> ::= <id ref>}
\text{<property descr> ::= <sort> <property facets>}
\text{<property facets> ::= [ <default facet poss> ]}
\text{Semantics: Any instance of a concept, for which this property is relevant, and receives no explicit value, will automatically inherit this default value.}
\text{<default facet poss> ::= default ( <value> )}
\text{<property fact> ::= [ not ] <instance id ref> . <property id ref> <value>}
\text{<instance property fact> ::= [ not ] <property id ref> <value>}
\end{align*}
\]

C.6 Concepts

\[
\begin{align*}
\text{<concept definition> ::= defconcept <concept id dec> [ <concept descr> ]}
\text{<concept id dec> ::= <id dec>}
\text{<concept id ref> ::= <id ref>}
\text{<concept descr> ::= [ <relevant property list> ]}
\text{| [ set of instances ]}
\end{align*}
\]
C.7 Instances

<relevant property list> ::= [ relevant ] <relevant prop list>
<relevant prop list> ::= <relevant prop list> , <relevant property>
| <relevant property>
<relevant property> ::= <property id ref> [ <concept value> ]

Semantic: The concept value has to be a valid value for the property. All the instances of this concept must have compatible values (equal, within the required range or unknown).

C.7 Instances

<instance definition> ::= definstance <instance decl>
<instance decl> ::= <instance id dec> <concept id ref> ( <instance fact list> )
<instance id dec> ::= <id dec>
<instance id ref> ::= <id ref>
<instance fact list> ::= <instance fact list> , <instance property fact>
| <instance property fact>
<set of instances> ::= [ instanceref ( <instance ref list> ) ]
| [ instance ( <instance dec list> ) ]

Semantic: It is possible to reference instances defined previously to define new ones, or both simultaneously.

C.8 Facts

<fact declaration> ::= deffacts <fact list>
<fact list> ::= <fact list> , <fact>
| <fact>
<fact> ::= <property fact>

C.9 Lexical tokens

alpha ::= [a-zA-Z]
digit ::= [0-9]
alphanum ::= ( {alpha} | {digit} )

symbol ::= {alpha} ( alphanum | _ )* digits ::= ( {digit}+ )
sign ::= [+][-]? INT_N ::= ( {sign} {digits} )
dblquote ::= [*]
space ::= [ ]
stringcarac ::= ["*]
point ::= [.]
decimal ::= ( {integer} {point} {digits}? | {sign} {point} {digits} )
exponent ::= [ eE ] {integer} REAL_N ::= (decimal) {exponent}? | {integer} {exponent}
IDENTIFIER ::= symbol
STRING_N ::= {dblquote} (stringcarac)* {dblquote}

C.10 Keywords

The following identifiers are reserved (keywords), and should not be used as user-defined identifiers:

boolean default defconcept
deffacts definstance defproperty
defsort false instance
instanceref integer interval
list nominal not
range real relevant
size sortref string
ttrue
APPENDIX D  Code of the Quick Guide

D.1 The Foo example

D.1.1 foo.adl

```adl
#DOMAIN "foo_domain"
#YP_SERVER "madrazo.gsi.dit.upm.es:6050"
AGENT YP-Agent -> YPAgent
  REQ_LIBRARIES: "ypagent.H"
END YP-Agent

AGENT Foo_Agent -> BaseAgent
  RESOURCES
  REQ_LIBRARIES: "foo-functions.C"
  SERVICES
  Foo: CONCURRENT foo_service
END Foo_Agent

AGENT Client_Foo -> BaseAgent
  RESOURCES
  REQ_LIBRARIES: "foo-functions.C"
  REQ_SERVICES: Foo
  GOALS
  AskForFoo: CONCURRENT AskForFoo
END Client_Foo
```

D.1.2 foo-functions.C

```c
#include <service.H>
#include <message.H>
#include <MASError.H>

void foo_service (Service &srv, Message &msg){
  cout << "I am the foo service" << endl;
}

void AskForFoo (Service &s, Message &msg){
  Message m;
```
D.2 The Echo example

D.2.1 echo.adl

```adl
#DOMAIN "echo_domain"
#YP_SERVER "madrazo.gsi.dit.upm.es:6050"
AGENT YP_Agent -> YPAgent
   REQ_LIBRARIES: "ypagent.H"
END YP_Agent

AGENT Mountain -> BaseAgent
   RESOURCES
      REQ_LIBRARIES: "echo-functions.C"
      SERVICES
         echo: CONCURRENT Echo
END Mountain

AGENT Mixy -> BaseAgent
   RESOURCES
      REQ_LIBRARIES: "echo-functions.C"
      REQ_SERVICES: echo
      GOALS
         user_echo: CONCURRENT user_echo
END Mixy

AGENT Mixie -> BaseAgent
   RESOURCES
      REQ_LIBRARIES: "echo-functions.C"
      REQ_SERVICES: echo
      GOALS
         user_echo2: CONCURRENT user_echo_2
END Mixie
```

D.2.2 echo-functions.C

```c
#include <service.H>
#include <message.H>
#include <MASError.H>

void Echo(Service &s, Message &msg)
   Message m;
   if (s.AskForAsyncService(m) != SUCCESS)
      Error();
   cout << "I request the foo service" << endl;
   cout << "ECHOOOOO: " << msg.Body << endl;
   m.Body = msg.Body; //Returns the input message
   s.SendAnswer(m); //Sends the msg to the caller
   }

void user_echo(Service &s, Message &msg)
   Message m, m_aux;
   char input_str[80];
   List<Message> answers;
   //Gets the user name
   cout << "Type your name and the Mountain will make an echo ";
   cin >> input_str;
   //Asks Mountain for the echo service
   m.To = "Mountain";
   m.ServName = "echo";
   m.Body = input_str;
   if (s.AskForSyncService(m, answers) == SUCCESS){
      do {
         m_aux = answers.GiveCursor(); //each answer
         //Process the message
         m_aux.Show(); //Show is a method of the message class
         //to display the message structure
      } while(answers.GoNext());
   }
   else
      Error(); //Invoke the standard error function
   }

void user_echo_2(Service &s, Message &msg)
   Message m, m_aux;
   char input_str[80];
   List<Message> answers;
   int ServReq_id;
   //Gets the user name
   cout << "Type your name and the Mountain will make an echo ";
   cin >> input_str;
   //Asks Mountain for the echo service
   m.To = "Mountain";
   m.ServName = "echo";
   m.Body = input_str;
   if (s.AskForDeferredService(m, ServReq_id) == SUCCESS){
      //Now execute some actions before getting answers
      cout << "Service echo just requested" << endl;
      //Getting answers
      if (s.GetDeferredAnswers(answers, ServReq_id) == SUCCESS){
         do {
            m_aux = answers.GiveCursor(); //each answer
            //Process the message
            m_aux.Show();
         } while(answers.GoNext());
      }
   }
```

APPENDIX D. Code of the Quick Guide
D.3 The Santa example

D.3.1 santa.ckr1

defproperty boy_name
    range (string);

defproperty age
    range (integer (0:99));

defproperty good_behaviour
    range (boolean) default (true);

defproperty toy_name
    range (string) default ("ball");

defconcept Christmas_letter
    relevant boy_name, age, good_behaviour, toy_name;

defproperty toy_kind
    range (string);

defconcept Toy
    relevant toy_kind;

D.3.2 santa.adl

#DOMAIN "santa_domain"
#YP_SERVER "madrazo.gsi.dit.upm.es:6050"
#ONTOSTY santa.ckr1

AGENT YP_Agent -> YPAgent
    RESOURCES
        REQ_LIBRARIES: "ypagent.H"
    END YP_Agent

AGENT Santa -> BaseAgent
    RESOURCES
        REQ_LIBRARIES: "santa-functions.C"
        SERVICES
            dispenser_gift: CONCURRENT disp_gift
            REQ_MSG_STRUCT SANTA::Christmas_letter
            ANS_MSG_STRUCT SANTA::Toy
    END Santa

AGENT Mixy -> BaseAgent
    RESOURCES
        REQ_LIBRARIES: "santa-functions.C"

D.3.3 santa-functions.C

#include <service.H>
#include <message.H>
#include <MASError.H>

void disp_gift(Service &s, Message &msg)
{
    Message m;
    char buffer[2000];
    Christmas_letter *letterptr;
    Toy coal("coal");
    Toy gift, gift_extra;
    cout << endl << "Ho, ho, ho!! NEW LETTER" << endl;
    ReceiveCKRL(msg.Body.GetString());
    cout << "Ho, ho, ho!! I have received" << total_instances
        " in this letter!!" << endl;
    if (!(strcmp(messCKRL[0].concept, "Christmas_letter"))){
        letterptr = CKRL2Obj(messCKRL[0]); //messCKRL
gift.SetID("gift");
gift.toy_kind = letterptr->toy_name;
m.Body = gift.PrintCKRL();
    }
    cout << "Ho, ho, ho!! This is my answer (plus a bonus toy :-))!!"
        << endl;
    cout << m.Body << endl;
    s.SendAnswer(m); //Sends the msg to the caller
}

void send_letter(Service &s, Message &msg)
{
    Message m, m_aux;
    List<Message> answers;
    Toy *toyptr;
    Chain myname;
    myname = s.GetFatherName();
    Christmas_letter myletter(myname.GetString(), 1, TRUE,
        "Teddy_bear");

    requ_services: dispenser_gift
    goals
        send_letter: CONCURRENT send_letter
        ans_msg_struct SANTA::Toy
    end Mixy
myletter.SetID("MixyLetter");
//Asks Santa for the dispenser_gif service
m.To = "Santa";
m.ServName = "dispenser_gift";
m.Body = myletter.PrintCKRL();
cout << "I send my christmas letter to Santa: " << endl;
m.Show();
cout << endl;
if (s.AskForSyncService(m, answers)== SUCCESS){
  do {
    m_aux = answers.GiveCursor(); //each answer
    //Process the message
    m_aux.Show(); //Show is a method of the message class
    //to display the message structure
    //here, the ckrl description
    ReceiveCKRL(m_aux.Body.GetString());
    cout << "Wow! I have received " << total_instances << " gifts from Santa!!" << endl;
    for (int i=0;i < total_instances; i++){
      //total_instances global variable
      if (!(strcmp(messCKRL[i].concept,"Toy"))){
        toyptr = CKRL2Obj(messCKRL[i]);
        cout << "GIFT : " << toyptr->toy_kind.GetValue() << endl;
      }
    }
  } while(answers.GoNext());
else
  Error(); //Invoke the standard error function