MIX: A General Purpose Multiagent Architecture*

Carlos A. Iglesias, José C. González and Juan R. Velasco

Dep. Ing. Sistemas Telemáticos, E.T.S.I. Telecomunicación
Universidad Politécnica de Madrid, E–28040 Madrid, Spain
{cif,jcg,juanra}@gsi.dit.upm.es

Abstract. The MIX multiagent architecture has been conceived as a general purpose distributed framework for the cooperation of multiple heterogeneous agents. This architecture, starting from previous work in our group on multiagent systems, has been redesigned and implemented within a research project investigating a particular class of hybrid systems: those integrated by connectionist and symbolic components.
This paper describes in some detail the principal concepts of the architecture: the network model and the agent model. Around these models, a set of languages and tools have been developed. In particular, an Agent Description Language (MIX-ADL) has been designed to specify agents declaratively in a hierarchy of classes.

1 A multiagent architecture for hybridization

Distributed heterogeneous systems are receiving great attention from researchers in all the computer-related fields. These systems involve components with relevant differences regarding their nature (human/artificial), role inside an organisation, capabilities, structure, etc. In this work, we have developed a general framework for the interoperation of software systems [11]. This architecture, starting from a previous work in our group on multiagent systems [7], has been developed within a research project investigating a particular class of hybrid systems: those integrated by connectionist and symbolic components.

The obvious complementarity of the symbolic and connectionist approaches to problem solving has led to a growing interest in these hybrid systems involving the cooperation of both approaches [15]. This is the focus of the MIX project. An

* This research is funded in part by the Commission of the European Communities under the ESPRIT Basic Research Project *MIX: Modular Integration of Connectionist and Symbolic Processing in Knowledge Based Systems*, ESPRIT-9119. The MIX consortium is formed by the following institutions and companies: Institute National de Recherche en Informatique et en Automatique (INRIA–Lorraine/CRIN–CNRS, France), Centre Universitaire d'Informatique (Université de Genève, Switzerland), Institute d'Informatique et de Mathématiques Appliquées de Grenoble (France), Kratzer Automatisierung (Germany), Fakultät für Informatik (Technische Universität München, Germany) and Dep. Ingeniería de Sistemas Telemáticos (Universidad Politécnica de Madrid, Spain).

overview of the project can be found in [14], and a complete description of the MIX platform in [13]. To achieve the project goals, a first objective was conceived from the beginning: symbolic-connectionist integration would be pursued within a distributed framework for the cooperation of multiple heterogeneous agents.

Apart from the usual reasons for building a multiagent architecture, some conditions were specially taken into account:

- *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.)
- *Support for diverse integration schemes:* This architecture should support several integration schemes allowing different levels of coupling (ranging from loose to tight) among symbolic and connectionist processes.
- *Ease of integration:* The overload imposed on researchers to integrate pre-existing components in this architecture and to organise the interaction of these components should be reduced to a minimum. In any case, such effort should be fully justified in terms of the inherent benefits of this approach.

This presentation of the MIX architecture starts from the definition of two basic entities: the agents and the network through which they interact (see Figure 1). The basic functionalities and interfaces of the network constitutes our network model (Section 2). Although different agent models can be integrated, one has been adopted as standard for the current implementation (Section 3). Then, we introduce some features that make this platform specially adequate for symbolic/connectionist hybridization. Section 5, on applications development, shows how to build a simple cooperating hybrid system in the MIX framework. The relationship between this and other multiagent systems is covered in Section 6. Finally, we present some conclusions and our current research and development directions.

2 The network model

The network model serves to provide the agents with a uniform view of the network. A layered model has been defined, where two kinds of agents have been considered:

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

Three layers can be distinguished in the MIX network model (Figure 2):

- **Interface layer:** provides the abstraction and encapsulation mechanisms needed to isolate the agent itself from the net through which it interacts with other agents. In particular, it offers a C++ API for accessing the network and communicating via message-passing. This layer includes primitive

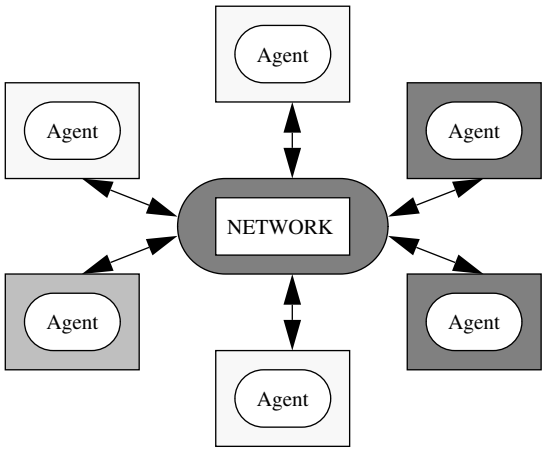

Fig. 1. The MIX multiagent architecture: network and agents

functions for message composition/extraction and delivery/reception with different synchronisation modes.

- **Message layer:** implements the facilities for message composition and content extraction. A message consists of:
  - *Addresses of sender and addressee:* At this level, agent names are translated into transport addresses.
  - *Speech act of the message:* This represents the primitive interactions available or understood by an agent according to a particular protocol: request a service, answer a message, subscribe to the network, etc. The complete set of permitted interactions involving any pair of (network or application) agents constitutes the *agent communication language*.
  - *Body:* This includes the arguments for the specified speech act. It is codified by using a particular knowledge representation language.
  - *Knowledge representation language:* The language used to codify the body of the message can be declared explicitly.
- **Transport layer:** is responsible for inter-networking, being implemented on TCP/IP sockets. Other networking protocols (http, etc.) are being considered for inclusion in future releases of the platform.

In the following, we will focus on the facilities offered by the message layer, which is the core of the network model, the layer specific for inter-agent communication. It interacts with the agent through the interface layer and delivers/receives messages to/from the network through the transport layer. This layer offers three groups of facilities (modelled as speech acts) to standardise the interactions between agents:

- **Network facilities**: provide the means to log into/out of the network and to receive information on the dynamic changes of the environment (mainly agent connections and disconnections). This layer is managed by a network agent, called the YP (Yellow Pages) agent, which is described below.

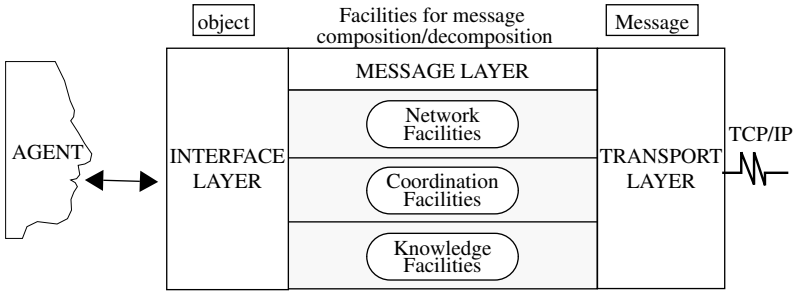

Fig. 2. The Network Model

- **Coordination facilities**: provide a limited set of speech acts for inter-agent communication. It also includes some of the typical coordination mechanisms of DAI, as Contract Net [20].
- **Knowledge facilities**: provide support for ontologies. The current implementation only deals with a modified version of CKRL [6].

2.1 Network facilities

We 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 current MIX Network Model. So, we start by enumerating which services can be offered; then, we outline how they are carried out in our architecture. This level offers the following services:

- *Logging-in/logging-out:* Primitives are provided to subscribe/unsubscribe to the network. An agent remains unknown to the rest of the agents, as long as it is not subscribed to the network.
- *Register/unregister capabilities:* These are intended for registering and unregistering agent capabilities, so that other agents can request them.
- *Register/unregister required capabilities:* Primitives are provided to state the capabilities an agent is interested in. This allows the filtering of the information that an agent will receive about its environment.
- *Agent name server:* This is needed in order to maintain and consult a database with agent names and transport addresses.
- *Notify logging-in/logging-out:* These provide a means of notifying the agents about the incorporation to or the withdrawal of agents from the net.
- *Trading:* This kind of service allows providers to advertise their services and consumers to match their needs against the available services.
- *Management of agent groups:* Maintaining groups of agents is useful for multicasting of messages. As the groups are dynamically constructed, some services are required to subscribe/unsubscribe to/from a group.
- *Security management:* Mechanisms are needed in general to encrypt/decrypt messages and to avoid the interference of intrusive agents.

Different strategies can be considered for offering the facilities listed above:

- **No network agent:** Each agent has capabilities to register new agents. When an agent is born, it must have some initial addresses to connect with. An agent can inform the rest of the known agents of the birth/death of agents. Every agent has to maintain locally agent names and addresses. We could consider different strategies: all the agents act as network agents, all the agents compete against the others to become network agents, etc.
- **Centralised network agent:** There is only one network agent which offers all these facilities.
- **Distributed network agents:** Instead of one, there is a federation of network agents (as in ODP-Trader [1]). Groups of agents communicate with a local network agent, and network agents can communicate between themselves to obtain/provide information from/to the rest of the network. In this case, a protocol between network agents is needed.

The MIX network facilities. The approach followed in the current implementation of the MIX Network Model has been to define one centralised network agent, called the YP agent. The advantages of this approach are twofold. Firstly, only one network address is *a priori* needed. Secondly, maintaining consistency among the internal databases of the agents is easier.

The idea of a centralised network agent is not too restrictive in practice regarding the number of agents that it will be able to manage. The MIX architecture has been designed from a problem solving perspective: only the agents cooperating in solving the same problem should be registered in the same YP agent. On the other hand, the functions of this agent in the MIX architecture are limited to acting as an active information repository. YP informs the agents on relevant changes in their environment, but inter-agent communications are established directly between the agents involved, without YP mediation.

The two main consequences of this approach are the following:

- The possibility of a communication bottleneck in the YP agent is negligible for the range of applications developed until now (tens of agents).
- A running set of agents can continue working in case of the failure of the YP agent. In such a case, the application agents suppose that their environment will never change.

Nevertheless, a protocol between different YP agents is currently under study and will be added to a future release of the platform. In this line, thanks to the object oriented agent model, a first version of a federated YP agent has been derived from the corresponding YP class in an application on intelligent telecommunications network management.

The YP agent. The YP agent offers the following services (i.e., it understands the following speech acts):

- Check_in: This service is used to log into the network, register capabilities and register requested capabilities in order to be notified about the agents offering them.

- Check_out: Used to log out from the network, and to notify this fact to the interested agents.
- Subscribe_to_group/Unsubscribe_from_group: These services allows agents to subscribe to a public group, and to receive the messages sent to this group.

The YP agent employs a strategy of *informing without demand*. The YP agent takes charge of testing if the agents are alive. When an agent crashes, it might have sent a Check_out message or not, so YP should test periodically the agents to maintain the consistency of its own database. It can also use some heuristics. For instance, an agent could be considered alive if it has requested something within some time period. On the contrary, an agent might be idle if it does not accept connections.

2.2 Coordination facilities

This layer defines the set of permitted interactions (speech acts) between application agents.

- Ask_for_service is used to request a service. There are three possible ways of synchronisation in which this primitive can be invoked: *asynchronous*, *synchronous*, or *deferred*.
- Answer is used for answering service petitions and for sending final results.
- Partial_results is used for sending partial results about a service petition.
- Failure notifies errors which do not allow the completion of a service.

There are also some primitives that implement a version of the Contract Net protocol. An agent may be interested in requesting a service from several agents in order to select the best offer(s). To do so, it sends a message to the group with the primitive Ask_for_service_with_cost. The agents willing to perform it, must reply with a cost figure (via the primitive Answer_cost). Cost may have different interpretations: price charged to the petitioner, estimation of the error made when performing the service, estimation of resources consumed (e.g., completion time), etc. The agent which sends the advertisement will select (according to a predefined evaluation function) some agent(s) to perform the service (via Ack) and will reject the other offers (via Nack). Then, the contracted agents can send back the results by using the primitive Answer, or can report error conditions via the primitive Failure or incremental/partial results via Partial_results.

2.3 Knowledge facilities

Two main functions can be distinguished regarding knowledge facilities:

- Providing a transparent interface between objects and knowledge descriptions. The MIX platform offers a set of tools to manage CKRL descriptions.
- Providing some primitives to access knowledge bases of other agents (assert/retract a fact, send a query, etc.) These primitives have not been fully implemented yet. We are now working on the development of a knowledge based agent integrated in the architecture and implemented in CLIPS [12].

– Providing mechanisms for the management of ontologies. Agents can interact because they share ontologies. An ontology is a description of concepts in a particular knowledge representation language. The body of the messages exchanged by agents is composed by chaining instances of these concepts.

3 The agent model

The MIX multiagent architecture can be considered heterogeneous in the sense that agents built according to different models can interact whenever they use the interface layer of the network model for communication purposes. MIX agents are autonomous entities capable of pursuing particular goals, offering specialised services to the rest of the world (its environment), and demanding services from other agents.

The MIX platform allows the easy implementation of agents following a modified version of the concurrent, object-oriented agent model proposed by Domínguez [7]. The main feature of this model is the way in which agents carry out services: when an agent receives a service petition and decides to perform it, the agent creates a concurrent process which executes the service. The agent only checks the incoming messages and creates the respective processes if needed. The internal components of an agent, depicted in Figure 3, are the following:

– **Internal database**:
  - *Environment Model*: stores the dynamic environment of the agent: YP agent address, addresses of agents offering services that it may need, or included in interesting public groups, etc.
  - *Self-Model*: stores the information about the agent itself (services offered and required, public and private groups, etc.)
  - *Process Management*: stores low-level information on attached processes.
  - *Private Objects* (described in 3.4).
– **Mailbox**: is the place where all the incoming messages are deposited.
– **Control**: determines the overall behaviour of the agent. (See Section 3.5.)

To specify the structure of these agents, a specialised language has been designed: ADL (Agent Description Language). The ADL specification of an agent includes these elements: services, goals, resources, internal objects and control. Besides this, ADL allows the organisation of agents in a hierarchy of classes.

3.1 Services

A typical agent offers services to other agents. Each agent can execute a set of actions (by calling library functions or external programs) that may be requested by other agents. These actions may be executed in a *concurrent* or *non-concurrent* way. The first method implies starting a new process for carrying out the task. So, it allows the agent to continue its internal working (taking care of new incoming messages and executing new services). The second method blocks the agent,

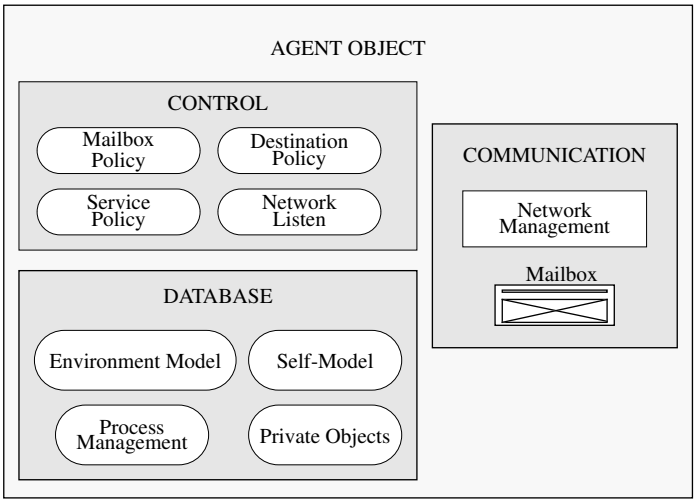

Fig. 3. The agent model

stopping its control loop, as a way to assure a perfect control over the global agent activity or as a means to improving efficiency.

Services are demanded and delivered through message passing protocols. One of the available protocols is Contract Net. To use this protocol, it is necessary that services have an associated cost function. In this way, agents that 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 to the agent that demanded the service. The petitioner decides, by analysing these results, the agent or agents that will get the contract to carry out the service (contract policy).

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.
- a private group of agents, i.e. a private alias. Private groups are shorthands useful for services programming.
- every agent offering the service in the application (broadcasting).

3.2 Goals

Every agent may have their own goals: tasks that are carried out under its own initiative, not to serve a petition made by another agent. Goals can be implemented through library functions or external programs but, unlike services, they are started up at birth. Special functions can be included for execution upon success

or failure of a goal. Regarding the presence or not of particular goals, we can talk about reactive and proactive agents. Reactive agents have no inherent goal. They just work on demand, offering their services to others. On the contrary, proactive agents have at least one goal.

3.3 Resources

It is appropriate to distinguish between two different concepts: the environment and the 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 YP agent, as described above.

On the other hand, *resources* refer to the static, *a priori* knowledge that the programmer at the agent level (the ADL programmer) has to specify. Resources include, among others, the following items:

- Required services: If an agent has to send service petitions to others, they have to be declared, along with:
  - the contract policy to follow in case of multiple bids (positive answers) when the Contract Net protocol is used.
  - a time-out, after which no more answers are taken into account.
  - the number of retries (if no answer has been received before time-out).
- Required/subscribed groups.
- Required ontologies.

3.4 Internal objects

Every agent may have its own internal data (internal objects). Such data can be specified as instances of library classes or as instances of concepts in an ontology. They are used for information storing between successive calls for any service or for data exchanging among several services.

3.5 Control

Three aspects of the control component of an agent can be modified in ADL:

- *Mailbox management policy*: determines which message will be the first one selected for processing. The standard manager uses a FIFO criterion. This function can be used to implement a priority scheme.
- *Service policy*: is used to implement different criteria regarding the attention to service petitions. The standard service policy follows an eager algorithm, that tries to serve every external petition. However, alternative policies are also possible; for instance, rejecting petitions when the agent's performance is low (due, for example, to overload) or when they come from outside the local network where the agent is running.

– *Destination policy*: the standard destination function may broadcast a service petition to every agent known to be offering it. The destination function may be used to restrict this policy to select, for instance, only the nearest agents or those satisfying special conditions.

4 The MIX platform: Tools for hybridization

Regarding hybridization, this platform offers some special facilities to implement a particular hybrid system by means of loose or tight coupling techniques.

For instance, for prototyping purposes, it can be useful to implement a first version of a hybrid system by using a set of loosely coupled agents. Each one of them may encapsulate a symbolic or connectionist system offering some services. In this way, we can encapsulate, for instance, a neural network as an agent offering the following services: Training, Set_weights, Give_output, Give_error, etc.

Let's suppose now that, due to reasons of inefficiency, a tighter level of integration is needed. To do this, ADL permits the definition of a *strongly coupled society*, which merges a set of separate agents into just one, thus replacing the initial communication mechanism between the agents (message-passing) with direct access to variables in memory. This change is transparent to the service programmer, thus facilitating experimentation with different integration levels.

5 Study case: a control application

In the following, the ADL specification of a simple cooperative control application is developed. Let's suppose that the data acquisition system of an industrial plant filters and aggregates signals coming from sensors to deliver a vector of normalised variables (context vector). Taking this vector as input, a control manager has to propose values for operation variables (operation vector), based on the suggestions received from different modules that implement different optimisation techniques (neural, statistical, fuzzy, etc.) These suggestions are previously sent to a prediction system (a more or less accurate model of the plant), and evaluated.

An agent-oriented model of this application is shown in figure 4. The problem is modelled by using the following set of agents. Collector provides filtered and normalised context vectors to the Manager upon demand. This agent asks for control actions to a set of controllers Controller_1, ..., Controller_n according to the Contract Net protocol. The contracted controllers elaborate an operation vector that is sent, along with the context vector, to Predictor for calculating the context vector at instant $t+1$. With this information, the controllers ask Evaluator for an estimation of the effectiveness of their respective suggestions according to a predefined criterion. Depending on the results available, the Manager displays the best operation plan to a human operator. The ADL code of this application is shown in Appendix A.

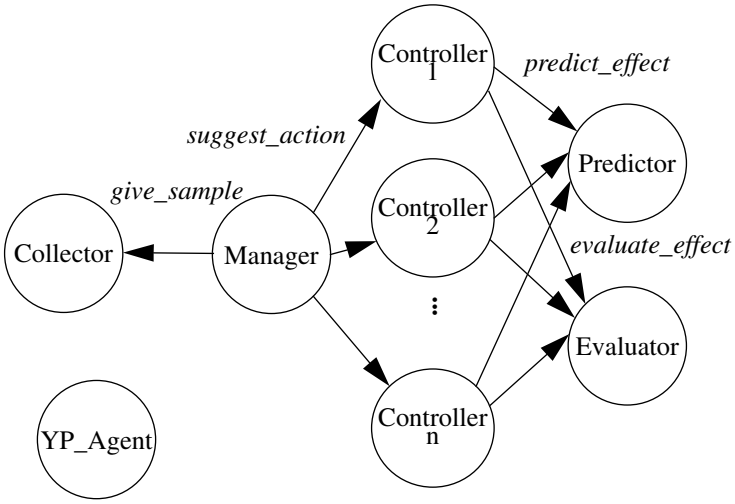

Fig. 4. An open-loop control application

6 Comparison with other approaches

One of the main contributions of the MIX architecture is the explicit distinction between agent model and network model. This distinction allows the integration of different agent models in a uniform way (i.e., our multiagent architecture is open). Our approach distinguishes between network agents, that provide the network facilities, and application agents, that can be implemented according to different models (i.e. agent models such as [3, 4, 16, 17, 19]). In spite of this, the agent models in many architectures include most of the layers defined in our network model [3].

Our architecture is also related with the architecture in [17], though our model is not based on cognitive psychology:

- Both architectures share the usage of a network agent (YP/FA, Facilitator Agent). In both cases, agent interaction is based on service requests, and a contract net protocol is offered.
- Our agent model distinguishes between the agent-object and the processes that are executed to serve petitions and that share some of the knowledge of the agent. There are some similarities between our agent-object and the SIAs (Senior Intelligent Agents) and between our service-object and the JIAs (Junior Intelligent Agents). However, in our architecture only agents are registered. In this way, the communication load is decreased.
- The competence energy level of the SIAs could be interpreted as a particular "destination policy": an agent (FA or JIA) requests a service from the JIA having the highest energy level. The concept is global in this architecture (registered in the FAs) but local in our architecture (different agents can have

different destination policies according to other criteria: select the agents in the same organisation, the cheapest ones, etc.).
- The degree of parallelism obtained via agent-cloning is lower than the parallelism obtained between an agent and its concurrent services (which can access common databases), specially if services are implemented with threads.

The agent model proposed by Domínguez has a direct relationship with works on object oriented concurrent programming. In particular, it employs the same synchronisation mechanisms that ABCL [28].

Some agent programming languages such as MAIL [21] and MAPL [18] use a process-oriented concurrent language between a general purpose language (C, Lisp,...) and the agent programming language. In contrast, ADL is translated directly to C++, and all the procedures associated to an agent are also written in C++. An API of the agents library is provided, and several tools to make transparent the use of knowledge structures (in CKRL).

ADL is also related with AgentSpeak [25]. Though the agent models are quite different, the agent programming language is quite similar. The main differences are in the absence of plans in ADL, and in the absence of a network model and agent policies in AgentSpeak.

Regarding the MIX network model, it is related to EMMA [22], I^3 [10] and HISML[24], though it does not deal with database access nor event-based interactions.

The message format is similar to the KQML [8] one, facilitating a future integration of KQML.

7 Conclusions and future work

The multiagent architecture presented in this paper is built around a network model and an agent model. This separation helps to clarify the relationship between network agents, existent in most of the multiagent architectures, and application agents. The network model allows the integration of different agent models in an open framework.

The agent model of the architecture provides a great degree of parallelism in task execution. The agent can be seen as the monitor of a blackboard system where the different services share the internal objects of the agent, and the monitor of the blackboard takes charge of creating these services. The agent can handle two kinds of groups of agents: public groups, managed via subscription, and private groups, particular to each agent.

The MIX multiagent toolbox consists of several tools for building multiagent architectures:

- *MSM C++ library*: This offers the basic low-level functionality of the platform, including all 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 other machines in a distributed and heterogeneous environment. In this task, it will use the following two complementary tools listed below.

- *CKRL toolbox*: This translates both CKRL descriptions to C++ classes and objects, and C++ objects to CKRL descriptions.
- Standard ADL agent definitions and CKRL ontologies.

We are currently working on the coordination facilities, extending the cooperative performatives, and studying the integration of emergent standards as KQML. Another line of development is the integration of other knowledge representation languages such as KIF [9] and COOL[12].

The platform is currently being used by the MIX consortium to test different hybrid models for three applications. The first one is the optimisation of a motor/gear-box combination for a turbo-charged engine. The second one is regarding of the control of a roll-mill for a steel company. The last application pertains to the medical domain: a monitoring system for an intensive care unit. The hybrid models considered include the integration of neural and symbolic components. From the symbolic part, the following reasoning paradigms are being applied: fuzzy, probabilistic, case-based, rule-based and model-based.

The platform is also being applied by our group in other areas, as natural language processing, intelligent network management and process control in fossil power plants [23].

8 Acknowledgements

This work would have never been done without the experience accumulated during years and the tools developed by Mercedes Garijo and Tomás Domínguez in their Multiagent System Model, that constitutes the basis for the MIX agent model. We are also indebted to Jaime Alvarez and Andrés Escobero (from our group) and Marc Vuilleumier (from Université de Genève, Switzerland) for their contribution to the implementation of the platform. Finally, we thank Michael Wooldridge and two anonymous referees for their careful review of an earlier version of this chapter.

A Appendix: ADL code for the control application

Here follows the ADL specification of the control application explained in 5.

```
#DOMAIN "atal-example"
#YP_SERVER "tcp://madrazo.gsi.dit.upm.es:6050"
#COMM_LANGUAGE ckrl
#ONTOLOGY "example.ckrl"

AGENT YP_Agent -> YP_Class
END YP_Agent
```

```
AGENT Manager -> Basic_Class
    RESOURCES
        REQ_LIBRARIES: "manager_funct.C"
        REQ_SERVICES:
            give_sample;
            give_suggestion
                CONTRACT_POLICY Eval_Function
                    REQ_MSG_STRUCT example::error_estim
    GOALS
        start_control: CONCURRENT Start_Function
END Manager

AGENT Collector -> Basic_Class
    RESOURCES
        REQ_LIBRARIES: "collec_funct.C"
    GOALS
        get_data: CONCURRENT Get_Data_Function
    SERVICES
        give_sample: Give_Sample_Function
                ANS_MSG_STRUCT example::Vector
END Collector

AGENT Predictor -> Basic_Class
    RESOURCES
        REQ_LIBRARIES: "predict_funct.C"
    SERVICES
        predict_effect: Predict_Effect_Function
                REQ_MES_STRUCT example::context_vector;
                               example::operation_vector
                ANS_MES_STRUCT example::context_vector
END Predictor

AGENT Evaluator -> Basic_Class
    RESOURCES
        REQ_LIBRARIES: "eval_funct.C"
    SERVICES
        evaluate_effect: Evaulate_Effect_Function
                REQ_MES_STRUCT example::context_vector;
                               example::context_vector
                ANS_MES_STRUCT example::evaluation_result
END Evaluator

CLASS Controller_Class -> Basic_Class
    RESOURCES
        REQ_SERVICES:   predict_effect,evaluate_effect
        SUBSCRIBE_TO:   Learning_Group
    SERVICES
        give_suggestion: Give_Sugg_Function
                REQ_MES_STRUCT example::context_vector
                ANS_MES_STRUCT example::operation_vector;
                               example::evaluation_result
             COST Error_Estimation_Function
                ANS_MES_STRUCT example::error-estim
END Controller_Class
```

```
AGENT Controller_1 -> Controller_Class
    RESOURCES
        REQ_LIBRARIES: "fuzzy_control.C"
END Controller_1

AGENT Controller_2 -> Controller_Class
    RESOURCES
        REQ_LIBRARIES: "neural_control.C"
END Controller_2
.....
```

References

1. Nicholas M. Avouris and Les Gasser, editors. *Distributed Artificial Intelligence: Theory and Praxis*, volume 5 of *Computer and Information Science*. Kluwer Academic Publishers, 1992.
2. M. Barbuceanu and M. S. Fox. The architecture of an agent building shell. In Wooldridge et al. [26]. (In this volume).
3. R. P. Bonasso, D. Kortenkamp, D. P. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. In Wooldridge et al. [26]. (In this volume).
4. Alan H. Bond and Les Gasser, editors. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann, 1988.
5. K. Causse et al. Final discussion of the Common Knowledge Representation Language (CKRL). Deliverable D2.3, MLT Consortium, ESPRIT project 2154, May 1993.
6. Tomás Domínguez. *Definición de un Modelo Concurrente Orientado a Objetos para Sistemas Multiagente*. PhD thesis, Dep. Ingeniería de Sistemas Telemáticos, E.T.S.I. Telecomunicación, Universidad Politécnica de Madrid, 1992.
7. T. Finin and R. Fritzson. KQML — a language and protocol for knowledge and information exchange. In *Proceedings of the Thirteenth International Workshop on Distributed Artificial Intelligence*, pages 126–136, Lake Quinalt, WA, July 1994.
8. M. Genesereth, R. Fikes, et al. Knowledge Interchange Format, version 3.0. Reference manual. Technical report, Computer Science Department, Stanford University, 1992.
9. Michael R. Genesereth, Dave Gunning, Rick Hull, Larry Kerschberg, Roger King, Bob Neches, and Gio Wiederhold. Reference architecture I^3 intelligent integration of information program. Draft, January 1995.
10. Michael R. Genesereth, Narinder P. Singh, and Mustafa A. Syed. A distributed and anonymous knowledge sharing approach to software interoperation. In *Proceedings of the Third International Conference on Information and Knowledge Management, CIKM'94*, November 1994.
11. Giarratano and Riley. *Clips Manuals version 6.0*. Software Technology Branch - Lyndon B. Johnson Space Center, 1993.
12. José C. González, Juan R. Velasco, Carlos A. Iglesias, Jaime Alvarez, and Andrés Escobero. A multiagent architecture for symbolic-connectionist integration. Technical Report MIX/WP1/UPM/3.0, Dep. Ingeniería de Sistemas Telemáticos, E.T.S.I. Telecomunicación, Universidad Politécnica de Madrid, December 1994.

13. Melanie Hilario, Christian Pellegrini, and Frédéric Alexandre. Modular integration of connectionist and symbolic processing in knowledge-based systems. In *Proceedings of the International Symposium on Integrating Knowledge and Neural Heuristics*, pages 123–132, Pensacola, Florida, May 1994.
14. ISO/IEC JTC1/SC 21. Draft recommendation X.9tr: ODP trading function ISO/IEC DIS 13235. Technical report, May 1995.
15. Larry R. Medsker. *Hybrid Intelligent Systems*. Kluwer Academic Publishers, 1995.
16. J. P. Müller. A markovian model for interaction among behavior-based agents. In Wooldridge et al. [26]. (In this volume).
17. S.-J. Pelletier and J.-F. Arcand. Cognitive based multiagent architecture. In Wooldridge et al. [26]. (In this volume).
18. Agostino Poggi. DAISY: an object-oriented system for distributed artificial intelligence. In Wooldridge and Jennings [27], pages 297–306.
19. A. Sloman and R. Poli. SIM_AGENT: A toolkit for exploring agent designs. In Wooldridge et al. [26]. (In this volume).
20. Reid G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. In Bond and Gasser [5], pages 357–366.
21. D. D. Steiner, A. Burt, M. Kolb, and Ch. Lerin. The conceptual framework of MAI^{2}L. In *Pre-Proceedings of MAAMAW'93*, Neuchâtel, Switzerland, August 1993.
22. Katia Sycara and Michael Roboam. *Distributed Artificial Intelligence: Theory and Praxis*, chapter 2. EMMA: An Architecture for Enterprise Modeling and Integration, pages 197–213. Volume 5 of Avouris and Gasser [2], 1992.
23. Juan R. Velasco, José C. González, Carlos A. Iglesias, and Luis Magdalena. Multiagent based control systems: a hybrid approach to distributed process control. In A.E.K. Sahraoui and J.A. de la Puente, editors, *Preprints of the 13th IFAC Workshop on Distributed Computer Control Systems, DCCS-95*, pages 7–12, Toulouse, France, September 1995.
24. Sankar Virdhagriswaran. Heterogeneous information systems integration - an agent messaging based approach. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94)*, November 1994.
25. D. Weerasooriya, A. Rao, and K. Ramamohanarao. Design of a concurrent agent-oriented language. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages (LNAI Volume 890)*, pages 386–402. Springer-Verlag: Heidelberg, Germany, January 1995.
26. M. Wooldridge, J. P. Müller, and M. Tambe, editors. *Intelligent Agents Volume II — Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages (ATAL-95)*, volume ??? of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1996. (In this volume).
27. Michael Wooldridge and Nicholas Jennings, editors. *Agent theories, architectures, and languages*, Amsterdam, The Netherlands, August 1994. ECAI.
28. Akinori Yonezawa and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In Bond and Gasser [5], pages 434–445.

This article was processed using the LaTeX macro package with LLNCS style