MACE

A Flexible Testbed for

Distributed AI Research

Les Gasser

Carl Braganza

Nava Herman

Distributed Artificial Intelligence Group
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0782

appeared in

M. Huhns, Ed. *Distributed Artificial Intelligence*, Pitman, 1987.

Abstract

Parallelism in AI problem-solving applications can be exploited at many different levels: in hardware, in the implementation language (e.g., a production system language), in a problem-solving paradigm, or directly in the application. MACE (Multi-Agent Computing Environment) is an instrumented testbed for building a wide range of experimental Distributed Artificial Intelligence systems at different levels of granularity. MACE computational units (called "agents") run in parallel, and communicate via messages. They provide optional facilities for knowledge representation (world knowledge, models of other agents, their goals and plans, their roles and capabilities, etc.) and reasoning capabilities. The MACE environment maps agents onto processors, handles inter-agent communication, and provides:

- *A language for describing agents,*
- *Tracing and instrumentation,*
- *A facility for remote demons,*
- *A collection of system-agents which construct user-agents from descriptions, monitor execution, handle errors, and interface to a user.*

We have used MACE to model lower-level parallelism (a distributed system of production rules without global database or inference engine, where each rule is an agent) and to build higher-level distributed problem-solving architectures (domain-independent distributed blackboard and contract-net schemes). MACE is implemented on a 16-node INTEL SYM-1 large-memory hypercube and in a LISP machine environment.

1 Introduction

1.1 MACE Goals

Distributed Artificial Intelligence (DAI) is a subfield of AI concerned with the problems of describing and constructing multiple "intelligent" systems which interact. MACE (for Multi-Agent Computing Environment) is a language, programming environment, and testbed for DAI systems [8, 9, 10, 17, 20, 27, 35]. Other experimental testbeds have been built for DAI research (E.g., [18, 21, 30]), but these have generally encompassed only a single problem-solving architecture or domain (e.g., air-traffic control or distributed sensor nets; rule-based, blackboard, or contract-net architectures, etc.). In MACE we have built on these experiences to construct a testbed which will be useful for experimenting with a variety of architectures and problem-solving paradigms, at varying levels of granularity. The goal of MACE is to support experimentation with different styles of distributed AI systems, at different levels of complexity.

An experimental system built in MACE is a collection of *agents* which run in parallel and interact in organized ways. The dominant metaphor of MACE is that of a DAI system as an organization of problem-solvers. MACE supplies the basic constructs for describing features of agents which allow for reasoning about their organization (see below). At a high level, a programmer can describe the role and authority structure of agents, their goals, plans, skills, etc. A cluster of agents can be viewed abstractly as a unit, so agents can reason about other parts of the system as organizations with which they interact. In this way, MACE provides a basis for research into comparative organizational forms for problem-solving.

But not all MACE agents need to be complex; we would also like to experiment with DAI systems which don't require large-grain agents or knowledge of organization. The MACE organizational paradigm suffices to model very simple agents (such as a collection of individual production rules built as agents, which

interact in highly structured ways). It can be used to describe collections of medium-granularity agents which interact more opportunistically (such as a blackboard system with multiple knowledge sources). It would be useful for describing large and complex collections of high granularity agents (such as a collection of higher-level problem-solvers with changing organizational structures).

Versions of MACE are now running on an Intel SYM-1 concurrent processor ("hypercube") and on TI Explorer LISP machines. Our emphasis is on experimentation with DAI techniques, not on efficient execution of MACE agent communities. MACE is an attempt to provide the tools for prototyping experimental systems which run in real parallel environments, but not for production-quality delivery systems.

This chapter describes the philosophy of MACE, the high-level constructs of MACE, and many of the functions which implement those high-level constructs.

1.2 General Description of MACE

A programming system should reflect the concepts and structures of the problems it is designed to express. The dominant metaphor of MACE is a collection of intelligent, semi-autonomous agents interacting in organized ways. In Wegner's terms [33], MACE is a *distributed, object-oriented system.* The MACE system is a collection of components, including:

- A collection of **agents**: Agents are the basic computational units of the MACE system. MACE agents are inherently "social" in nature: they know about some other agents in their environment, and expect to draw upon and coordinate with the expertise of others they know about. Thus they may need to know the identity and location of their acquaintances, something about their capabilities, etc. MACE agents represent this information in the form of "models of other agents in the world." In addition, agents may be organized into sub-units or coalitions which act in response to particular problems. In this sense they respond as organized groups or composites. The MACE Agent Description Language includes facilities for describing organized clusters of agents.
- A community of **system agents**: The pre-defined system agents provide MACE command interpretation, a standard user interface, the agent-builder cluster, error handling, tracing, execution monitoring, etc. Specific MACE agents serve as interactive tools for building MACE programs. Such tools include agents to build and edit the behaviors and structure of other agents, as well as to change the parameters of the execution environment or simulation.
- A collection of **facilities** which all agents may use: These include a pattern matcher, a simulator, several standard agent engines, handlers for standard errors, and standard messages understood by all agents. The MACE testbed is instrumented to allow for measurements of the characteristics of problem-solvers during experimental runs. Message traffic, queue and database sizes, work done by an agent (in terms of elapsed real time or number of invocations), and load on a processor node are common measurements. We expect distributed AI systems constructed from collections of agents to be very complex [22, 25, 26]. In addition, we would like to have the control provided by repeatable tests, especially for development purposes. For this reason the MACE system includes a parameterized simulator. Agents communities can be developed on the simulator and moved to MACE on parallel hardware. MACE incorporates extensive pattern-matching facilities for interpreting messages, for pattern-directed invocation of asynchronous events (e.g., demons and other event-monitors), and for associative database access within agents.

- A **description database**: Agent descriptions are maintained in a description database by a system-agent cluster which constructs new descriptions, verifies descriptions, and constructs executable agents from descriptions.

- A collection of **kernels**: MACE kernels collectively handle communication and message routing, perform I/O to terminals, files, or other devices, map agents onto processors, and schedule agents for execution.

1.3 Related Work

The conceptual roots of MACE as a programming system are closest to ROSS, ACTORS, and LISP FLAVORS [14, 21, 34]. In such languages, the high level encapsulation units for computation are abstract data-types sometimes called *objects* [31]. Objects incorporate local variables and procedures, and interact by "sending messages" to one another. However, existing implementations of similar object-oriented languages are inherently sequential. Messages are really procedure calls, often implemented using constructs such as **funcall** or **apply**; the sending object must wait until the receiving object has processed a message before resuming. If the receipt of a message triggers more messages, there may be a depth-first activation pattern before the original sender can continue. While the metaphor of object-oriented systems suggests parallelism, most such languages actually are sequential in nature. The good reasons for this are the ease of implementation, lack of need for synchronization, and the fact that objects retain an interpretation context for return results automatically (see, e.g. ROSS [21].). However, parallel execution of objects is lost.

As an inherently parallel system, MACE presents some execution-time differences from existing object-oriented systems. In MACE, agents exhibit truly parallel execution, and we face the problems of synchronization and retaining interpretation contexts. Explicit synchronization can be performed using mailbox-level synchronization primitives; since all communication is via messages, a programmer can create synchronization behaviors for an agent easily [28]. Retaining interpretation contexts is more difficult, but can be achieved by tagging messages with *computation identifiers* linked to *partial result frames* in the originating agent. The partial result frame contains enough information to continue the computation once the partial result (i.e. the work of another agent) is achieved. At a higher level, this concept can be expanded to maintaining *plans* for a computation. Outgoing and incoming messages are linked to *plan points*, which are tagged with instantiated variables as they are completed [16]. MACE provides a framework for storing an agent's plans, which can be used for this purpose.

There has been a tradition of experimental research and simulation in Distributed AI, which has provided a strong motivation for MACE. In early 1980's Smith and Davis reported on the Contract Net (CNET) system [5, 30]. CNET used a bidding-contracting task allocation scheme to divide labor in a problem-solving task. The Contract Net was implemented as a simulation in LISP.

More recently, Lesser and Corkill at UMASS have constructed a testbed for distributed problem-solving research in a distributed sensor net domain, called the Distributed Vehicle Monitoring Testbed (DVMT) [18]. The DVMT is a collection of very complex problem-solving nodes each resembling a HEARSAY-II blackboard system, which cooperate to interpret signals from a sensor array. The nodes communicate by sending messages to one another. The DVMT is a large simulation system built in LISP. The DVMT simulator incorporates a language called EFIGE for describing the organizational communication structure among nodes and the problem-solving roles each node will play [24].

Rand researchers have constructed simulations of distributed problem solving in air traffic control and remotely-piloted vehicle domains, using the ROSS rule-based object-oriented simulation system [21]. ROSS is a flexible simulation system but its execution is sequential, not parallel.

Hewitt and his associates have introduced a highly concurrent programming system for DAI research based upon the ACTORS model. Actors are independent objects which communicate via messages and which have three activities: they can change state, send messages, and create new actors. The ACTORS work introduced the idea of **acquaintances** as other actors known in the environment. MACE has expanded upon this notion by allowing agents to incorporate detailed models of the behavior and capabilities of its acquaintances, and by allowing agents to reason about the actions of their acquaintances. Hewitt et al. have also constructed a simulator for their low-grain parallel ACTORS system [19].

Other single-paradigm domain-independent problem-solving frameworks and description systems have been built. These include the blackboard systems BB1 [13], and GBB [2], and the Stanford Knowledge Systems Laboratory ensemble which includes AGE, CAGE, POLIGON, etc. [23].

These systems have shown the utility of experimentation and simulation in DAI in single paradigms and for single domains. The thrust of MACE is to supply a distributed testbed for systems with varying architectures and granularity.

2 What is a MACE Agent?

MACE agents are self-contained, active *objects* which communicate with one another through messages. Agents exist in an *environment*, and have three aspects: They *contain knowledge*, they *sense their environment*, and they *take actions*.

Agents contain some knowledge and a means to manipulate that knowledge. Agents have *attributes*, which are properties of agents that can be referenced by the agent. The attributes are the repositories of the agent's knowledge. Attributes are not directly visible to any other agent, but the contents of attributes may be sent in messages to other agents, as a way of transferring knowledge. Attributes may have the full power of an associative, deductive database, incorporating pattern-directed retrieval and inferencing. (E.g., the attribute *acquaintances* is such a database).

The environment of an agent comprises the MACE system, other agents, and the world outside MACE. Agents have two ways of sensing their environment. First, agents sense other agents and the outside world by receiving messages. Agents receive messages because other agents (including users or external processes) have sent them via the MACE kernel communication handlers and/or intermediary *interface agents*. Second, events in the kernel environment and internal to the agents themselves (e.g., alarm timers) can be sensed directly by the agent via its engine-shell (see below).

The active part of an agent is called its *engine*. Each agent's engine defines the agent's activities and how it interprets messages. The engine manipulates the knowledge stored by the agent's attributes in response to messages received by the agent. An agent's engine is its only active part. Agents can take 3 kinds of action:

- They can change their internal state by manipulating their attributes;

- They can send messages to other agents (by calling upon kernel communication handlers);
- They can send *monitoring requests* to the MACE kernel to monitor kernel-level events (e.g., timers), events elsewhere in the system, and their own internal state changes.

Only the last two types of action are visible outside the agent.

To carry out these actions, agents may contain locally-defined functions, which also cannot be referenced from outside the agent. These functions are a type of procedural knowledge which may be called upon by the agent's engine as it runs. Since these functions may give an agent specific skills, they may be sent in messages as a way of transferring skills to other agents.

Every agent has a *name* and a *class*. Agents are described by their *class*. There may be many *instances* of agents of the same class, but they will have unique names within that class. Uniqueness is determined by the combination of *name* and *class*.

3 What Agents Know

While the agent description framework is general, MACE provides facilities for representing particular kinds of knowledge. Frameworks for organizational and interactional knowledge are the primary higher-level knowledge structures built into MACE. Knowledge for organization and interaction is built into an agent attribute called its **acquaintances**. The acquaintances attribute provides an environmental model by representing explicit models of other agents in the world. This attribute is an associative database within an agent.

Agents may also have specialized local knowledge, represented in user-defined attributes. For example, some of our existing agents with rule interpreters for engines have a specialized **memory** attribute which stores is the production systems' working memory, and a specialized **behaviors** attribute which stores the procedural rules for operating on that memory.

3.1 Models of Other Agents in the World

Every agent has its own *model of the world*. This model encompasses knowledge of itself, other agents, their *roles*, their *skills*, their *addresses*, etc. This knowledge creates within the agent an image of its environment which it can interact with and change as its knowledge grows. Every agent holds its world model in an attribute called *acquaintances*. The agents it models are referred to as its acquaintances. Every agent's world model has representations of other agents in terms of the following qualifiers:

name The name of the agent being modeled.

class The name of the modeled agent's class.

address The location of the modeled agent. This address is used in communicating with the agent.

roles The *relationship(s)* the modeled agent bears to this agent, or a named role it plays. Currently, the role has several predefined values:

self The model is of the agent itself.

creator The model is of the agent's creator.

org-member The model is of a member of the organization which this agent manages (see below).

my-org-manager The model is of the top-level manager of the organization of which this agent is a part (see below).

co-worker The model is of a member of the organization of which this agent is also a part (see below).

This list of values may include additional user-declared roles.

skills Skills are what this agent knows to be the capabilities of the modeled agent. Skills are defined by *skill descriptors* which are pairs consisting of a set of patterns which match goals the skill will achieve, and a set of alternative methods for achieving the goals. The format of methods is left to the programmer (and they depend upon the engine used), but there are generally two types of methods: *Explicit procedures* can be invoked by the agent's engine. Their structure thus depends directly on the agent's engine definition. *Agent references* are the names of known agents which can possibly achieve the goals. Thus agents can do things by calling on other agents. Agent references are specified with an agent description and a message to be sent to that agent. One of our standard engines employs the descriptor USE to define explicit procedures, and REF to describe agent references. (See the CNET example below).

goals Goals are what this agent understands the modeled agent wants to achieve. Goals are represented by patterns which match programmer-specified descriptions of agent goals. Knowing about the goals of another agent is useful for mixed-initiative task allocation [5] and for reasoning about communication [3, 29]. For task allocation, a *goal-association database* (agent attribute) can be used to associate an agent's skills with the goals of another agent which they will help to achieve. The modeler agent can then reason about which acquaintances might provide a consumer for its services. For example, for *greedy robot* distributed task allocation in a multi-robot system [1] a robot task allocator must decide where to request more work. Acquaintance goals indicate where a greedy robot can market its production capabilities.

plans Plans are an agent's view of the way the modeled agent will achieve its goals. Plans are represented by a partially ordered collection of goals or skills, interspersed with *plan points*. Plan points are partial contexts from which the agent can resume computations when others have fulfilled their commitments to do some work [16].

3.2 Getting Acquainted

There must be a way for an agent to get to know about other agents. The MACE system provides two ways of instantiating an agent with the knowledge of other agents *built-in*. It also provides a system agent called the *directory* agent, which can furnish addresses of other agents in the system upon request.

An agent is instantiated with a model of its *creator* i.e. the agent or source which is responsible for its instantiation. The model contains only the name, class, address and role of the creator.

Each agent also carries a model of itself. This model can be set up in the class description of the agent, using the *plans, goals* and *skills* statements of the Agent Description Language (see Sec. 7.1). The name, class and address of the agent are provided by the MACE system; its role is set to *self.*

The user may specify some acquaintances which will be known at instantiation. These will be added to the default values for the acquaintance attribute in the class description of the agent. When the agent is instantiated, all its acquaintances' addresses will be present. This implies that the agent will not be instantiated until all its acquaintances exist. The roles of *self* and *creator* are reserved at instantiation.

There are means to *dynamically* alter an agent's concept of the world. The agent can add new acquaintances with the function add-acquaintance, as in

```
(add-acquaintance CNODE-9 CNODE [role CNODE-MANAGER]))
```

which would make the agent whose name and class is (CNODE-9 CNODE) and whose role with respect to the invoking agent is CNODE-MANAGER, an acquaintance. Unwanted acquaintances can be removed with delete-acquaintance. In addition, the value of the individual qualifiers associated with an acquaintance can be modified with change-qualifier, a function that replaces the previous value of the qualifier with the newly specified one, or add-to-qualifier, a function which will add information to a qualifier:

```
(add-to-qualifier CNODE-9 CNODE [role POTENTIAL-NQ-CONTRACTOR])
```

would change CNODE-9's role, adding POTENTIAL-NQ-CONTRACTOR to CNODE-MANAGER.

At any time, an agent can query the *directory agent* for the addresses of other agents in the system. (See Sec. 9.3) The agent could then ask the other agents for pertinent information to fill up its model of these agents.

The MACE system provides *selectors* to query the agent's world model database according to specified criteria. Selectors are pattern-directed associative retrieval functions. Acquaintance selections can be made on the basis of name, class, role, skills, goals, plans and in combinations of these. They provide the agent with the ability to abbreviate references to acquaintances in a meaningful way, and to reason about work organization at run-time. A database selector is specified by the character '@', followed by a list containing name of the selector and a sequence of patterns to be extracted from the specified acquaintance qualifier. To be extracted, a qualifier must have entries which satisfy all of the patterns specified in the query. The selector will return the matching items, or nil, if there is no match. Depending on the type of query, a single item, (i.e. an address, a plan, a role, etc.), or a list of items may be returned. For example, the selector

@(goals-of *agent class*)

returns the goal list of the acquaintance whose name and class matches (*agent class*), while the query

```
@(roles CNODE-MANAGER ORG-MEMBER)
```

returns a list of addresses of acquaintances known to fill both the roles *CNODE-MANAGER* and *ORG-MEMBER*.

4 How Agents Sense Their World

Agents sense their world by receiving messages and by being forcibly notified of internal and system-level events. Most sensing is passive sensing in the form of receiving messages from the outside world. Some sensing is active sensing in the form of requests for notification of events.

The MACE system delivers messages addressed to an agent in the agent's *incoming-messages* queue. Messages are generally queued in the order in which they arrive, but message receipt may be prioritized. Before receiving at least one message, an agent is dormant. When a message arrives for an agent, the agent is activated.

The message received by the agent contains the address of the sender of the message. The agent can use this address to reply to the sender, if it so wishes. This information is provided by the MACE system; the sender does not have to include its address as part of the message.

Messages received by an agent will always be in the format:

```
(source-address message-content)
```

The source address is the address that is stored in the address part of the source agent's model of itself unless explicitly coerced to be some other address (see Sec. 5.1). The message content is the actual message sent to the agent. It is a LISP list. Beyond this, the MACE system puts no restraints on the form or content of the message. It is up to the agent to interpret and respond to the messages it receives using its own particular engine.

4.1 Monitoring Events: Demons, Event-Monitors and Synchronization

Two kinds of events can be monitored by agents in MACE:

- Events which are effects of some agent's visible actions. Such events include sending or receiving specific messages, agent status changes, and the actual creation or destruction of agents (e.g., placing an agent into a *new* state or deleting it from the directory). These events are monitored by *demons*. Demons are mappings from event descriptions to messages. A demon is always active. When an agent requests a demon to monitor a particular type of event, the demon waits until an event matching the description occurs, and then issues a specified message informing the agent of the event.
- Events which are the result of kernel actions or state-changes in the agent doing the monitoring. (No agent can directly monitor state-changes internal to another agent.) These events are monitored by *imps*. Imps are predicates on internal attributes or certain predefined kernel events (e.g., alarms, pattern-triggers). Imps are very powerful; they can invoke any LISP function within the agent. Imps

are used for automatic interpretation of signals or error states, and for synchronization (e.g., for moving from a wait state to a running state when an alarm triggers.) They are evaluated after each execution cycle.

The engine shell provides a special imp which is a rudimentary alarm service within the event monitoring framework. The MACE engine-shell also provides an imp called a *pattern-trigger* which maps a message template to a procedure. When a message which matches a pattern trigger arrives at an agent, the procedure executes without the agent being activated. Pattern-triggers and demons together provide easy tracing facilities, and simple ways to implement guarded ports as in OIL [4]. Pattern-triggers are also sometimes more useful than engine-interpreted messages because their overhead can be ignored in experimental measurements of agent activations.

5 How Agents Act: Engines

MACE agents take actions when the kernel evaluates a function called the *engine* of the agent. An agent's engine is the only executable part of the agent, apart from its initialization code and the action parts of any imps. The engine and its associated attributes (e.g., a set of production rules which it references) provide the procedural knowledge incorporated in an agent.

The engine has many duties, including the responsibilities of handling all communication, error handling, acquiring and disseminating of knowledge about its environment, etc. What the engine does is the only outward manifestation of the agent's activities. It is the engine's responsibility to deactivate the agent when it has completed its current activities. The MACE system provides the function deactivate-me, which returns the agent to a dormant or waiting state where it will remain until it gets a new message. At that time it will be scheduled for execution once again. Unless it deactivates itself, an agent's engine is evaluated on efery scheduling cycle. Specialized agents which poll external world states can be designed to be always active. The MACE system provides several simple engines which can be used when building agents: two are simple production system interpreters, and a third is a basic user-interface engine.

Before executing an agent's engine, the MACE system wraps it in an *engine shell.* The engine shell is a part of the MACE kernel which provides standard error handling facilities, protects the MACE system from engine errors, prioritizes messages, and handles standard messages which all agents understand (e.g., certain debugging and examination messages) The engine shell serves as a link between the agent and the MACE system. One of the primary functions of the engine shell is to trap irrecoverable errors caused by the agent's engine. The engine shell monitors an agent's activities and is involved in the tracing and profiling of an agent's execution. It performs all the necessary initialization for the agent, getting the addresses of its initial acquaintances, and executing its *init-code.* Finally, the engine shell can prioritize messages in a partial order specified in a template provided by the agent. This is useful in prioritizing interactions among specific agents, and for prioritizing demon-activated messages.

5.1 Sending Messages

Agents communicate by sending messages. Messages can be addressed to individual agents, to groups of agents, or to all agents of a particular class. Messages can be addressed to agents on MACE systems on

other machines, as long as a communication path exists.

Every agent in the MACE system has a unique address. This address is comprised of the agent's name, its class, the node on which it resides, and the machine on which the agent is running. This information is coded as a list, as follows:

```
(agent-name agent-class node machine)
```

The MACE system places this address in the acquaintance with the *self* role of the agent, allowing an agent to look up its own address. At instantiation the *acquaintances* attribute has the addresses of the creator of the agent, and the other agents which were defined to be its initial acquaintances. The agent can query the directory agent for addresses of other agents. (See Sec. 9.3)

There are three basic ways an agent can communicate with other agents. It can send a message to an individual agent, to a group of agents, or to all agents of a particular class. All messages will contain the address of the sender (as is stored in the agent's model of itself). The action of

```
(send-to-agent '(CNODE-1 CNODE 1 ORPHEUS) '(DIRECTED-AWARD ...))
```

is to send the message "(DIRECTED-AWARD ...)" to the agent whose name and class is CNODE-1 CNODE, and which is located on processor node 1 on a machine called ORPHEUS. A message can be made to appear to have originated from a different agent if a return-address is explicitly provided. The message received by (CNODE-1 CNODE 1 ORPHEUS) after

```
(send-to-agent '(CNODE-1 CNODE 1 ORPHEUS) '(START)
'(USER-1 USER 1 CSEVAX))
```

will appear to have come from (USER-1 USER 1 CSEVAX), irrespective of who had actually sent it. In addition there is an option which causes messages that are sent to non-existing agents to be returned to the sender; the default is to discard the message.

In addition to the function send-to-agent, there are functions that send messages to all agents which are instances of a particular class (that the directory agent knows about), and to all agents in a particular *group* (called, appropriately, send-to-class and send-to-group, respectively). Groups may be set up by individual agents to collectively address a number of agents. These groups names are specific to each agent, and cannot be shared across agents. Functions such as make-group, delete-group, add-to-group and remove-from-group are provided to create and manipulate groups.

5.2 Issuing Monitor Requests

MACE agents can act to invoke a demon by supplying event descriptions and notification messages to the demon. In the case of demons which monitor specific messages, message template patterns are supplied. Demons are invoked by

```
(demon event-description notification-message)
```

where *event-description* identifies the demon, and *notification-message* is the message to be sent to the invoking agent on the occurrence of the event.

Imps provide number of useful features to agents which help reduce the complexity of their engines. Powerful pattern triggered behavior can be created with the function pattern-trigger which accepts a pattern template to match incoming messages against, and an action which will be carried out when the message matches. Pattern variables can be used, and matching is done by a unification pattern matcher; bindings of the pattern variables can be used in the action specification. For example, our *allocation* agent (which maps and creates agents on nodes, on request from other agents) has initialization code:

```
(pattern-trigger '((KERNEL SYS ?n ORPH-SYS) (AGENT ?addr CREATED))
'(reply-to-pending-allocation-request ?n ?addr))
```

which causes its internal function reply-to-pending-allocation-request to be invoked whenever it gets a message from a MACE kernel saying that the kernel has created an agent (in response to an earlier request from the allocation agent). This imp has additional options that provide the ability to discard the incoming message, change the agent's state on occurrence of the message (i.e. wake it up if it was inactive), and to perform this match repeatedly.

The allocation agent also uses an *alarm* imp that periodically polls all the MACE kernels that it knows off, requesting load data, so that it can update its internal tables. The alarm imp is similar to the pattern-trigger imp, except that a time period (in seconds) is specified instead of a pattern-template. The action is carried out after a specified time has elapsed since the invocation of the imp. The code looks like:

```
(set-alarm PERIOD '(send-to-group KERNELS '(RETURN-LOAD-FACTOR)))
```

The same options are provided for this imp too, except that here there is no message involved. Both imps return a request number that can be used to disable their actions.

6 Organizations

To save resources and attention when they act collectively, agents must organize their activities. Organizing conserves resources by providing a basis for expectations of how others will behave. We view an organization as a structure of expectations and commitments about behavior. The actions of organizational participants are patterned, often governed by routines, and based on expectations of others. An organization only really exists indirectly in the commitments and expectations of its members. In MACE, the term *organization* refers to the abstraction which allows agents to treat a collection of activities as being part of a known concerted effort or a known role. In this sense, a *role* is an accepted shorthand for a set of expectations. An organization is an abstract object about which agents can reason, and to which they can ascribe expectations and commitments. The abstraction can be simplified by naming the role of the organization,

in the same way they may abstract their knowledge of the skills, goals, plans, etc. of other agents by naming their roles.

This view of organizations contrasts with the view presented by Pattison et al. in this volume [24]. From their perspective, an organization is a system of constraints on behavior; our view is that an organization is a set of commitments and expectations held by its members, and not only imposed structurally from outside. This allows us to model *negotiated order* properties of multi-agent systems more accurately [6, 7, 12, 15, 32].

MACE represents an organization as an abstract shell and a communication agent called a *manager*. The communication agent is named for the organization, and its primary organizational function is to disseminate messages which have been sent to the organization to the appropriate members of the organization. This abstract shell of an organization is represented as the world model of the manager agent. Each organizational member is represented in the world model with an *org-member* role entry. Thus the organization abstraction is the set of all acquaintances who are org-members. This set can be extracted using the @roles selector.

The manager of a MACE organization is primarily a bright secretary. The manager's engine maps most incoming messages to addresses of agents which handle them. The basic work of the manager, then, is *task allocation* for work arriving from outside the organization. Unlike most other agents, the manager's effect on a message may be invisible; when it arrives at its destination within the organization, the source of the message is most often the original sender.

Organization members can send messages to anyone they choose, without going through the manager. But they may also send a message which is representative of the organization, not themselves (like a form letter on organization letterhead). In this case the message is routed through the manager where the source address is converted to the name of the organization.

Organizations may be instantiated with staff members, or empty. The organization is a shell only, and if it has not been initialized with staff, it must issue staffing commands at the time it is initialized, to build member agents, to "hire" existing agents as members, etc. "Hiring" means establishing a mapping from messages received by the organization to agents which will handle the messages.

7 Describing Agents: The ADL and Description Database

MACE provides a language called the *MACE Agent Description Language* or *ADL* for describing agents. MACE differentiates the description of an agent from its executable form. Agent descriptions are stored in a database which is used by an agent cluster which constructs both new executable agents and new descriptions. New descriptions of agents are built using queries and assertions into the description database. The ADL is a procedural language for describing the database operations necessary to construct a new agent description.

We would often like to make or describe MACE agents which are have attributes in common with other MACE agents (e.g., engines, local functions, behavioral rules, etc.) Hierarchical classification and attribute inheritance are the schemes commonly used for related descriptions in object-oriented languages. In MACE, we often want to selectively import some but not all attributes of related agent-descriptions. Moreover,

a distributed-memory parallel environment presents difficulties for inheritance-based schemes at execution time. In a distributed-memory parallel environment inheritance can be handled by sending messages to inherit superclass attributes. This introduces several problems with inheritance:

- Speed: If message delay is significant, if inherited attributes reside several levels away, or if the amount of data to be transmitted in an inheritance message is large, the overhead for inheritance may be great.
- Reliability: If the superclass object disappears or the machine upon which it resides is disabled, the subclass object cannot inherit attributes.
- Reference Problems: If objects are arbitrarily spread across machines, knowing the exact whereabouts of parents or children several levels away depends upon totally consistent address tables at the moment of inheritance.
- Conceptual Problems Conceptually, classification is a different problem than execution. Classification of agents and attributes ought to be done as they are specified, but not necessarily as they are executed.

Inheritance and classification schemes are important for knowledge representation and there are important research issues which center around how to implement inheritance in a distributed environment[1]. For the moment, however, we consider execution-time inheritance unworkable for MACE. Each executable MACE agent is a self-contained object which inherits nothing from other agents at run time. Descriptions of MACE agents involve inheritance, primarily for establishing default values for agent attributes. Any run-time attribute changes must be handled explicitly by updating agents.

The MACE *Agent Description Language* uses *class descriptions* to describe agents. Class descriptions have unique *class-names*, or may be unnamed. Named class descriptions are stored by the MACE system for future reference, either for building executable agents of the named class, or for serving as a template on which to draw to describe new classes. The MACE system provides a method for describing new agents from existing class descriptions. Unnamed class descriptions, however, cannot be referenced later.

Attributes are properties of agents which contain the agent's knowledge. They are only accessible to the agent to which they belong, and not to any other agent. Certain attributes are pre-defined by the MACE system. These attributes will always be present in an executable agent, and never appear in a class description. They may be referenced by the agent.

- The attribute *incoming-messages* has been defined to contain all the messages received by the agent. The messages are put there by the MACE system in the order in which they are received, and it is the agent's responsibility to remove them. (see Section 5)
- The *status* attribute is used by the MACE system to reflect the current status of the agent. The attribute takes fixed values: *running, inactive, waiting, stopped.* The status attribute is visible and monitorable by other agents with demons.

[1]For example, we are currently pursuing another line of thought which treats inheritance as a distributed database problem. Inheritance is only a problem when attributes change - otherwise inherited attributes can be compiled into distributed agents with the only cost being duplicate copies of information. Changes in inherited attributes can be announced via message. The system must have ways, then, of dealing with anomalies caused by action taken before an inheritance-change message is received. The associated costs benefits, and mechanisms are open research questions at the moment.

Certain other attributes always appear in executable agents, but may be defined by a MACE programmer:

- The agent's *acquaintances* attribute will always appear, but may be defined by the programmer. This attribute at minimum contains the agent's model of itself and its creator.
- The agent's *processor* attribute defines the (logical) processor on which the agent will be loaded. If left unset by the programmer, the MACE kernel will load the agent and assign this attribute.

Besides these attributes, the user may define any attribute he wants, along with its initial values. The initial values will be set at instantiation.

7.1 Agent Description Language

The Agent Description Language, or ADL for short, is a language for describing agents, (i.e. *class descriptions*), either in terms of new or of pre-defined class descriptions. A new class may be defined by *importing* features of other *named* classes. Such features could be the *engine* or the *attributes*, of existing class descriptions, or even a complete class description itself. Some of the imported features may be *deleted* if not required. Additional features may be specified if so desired. The ADL also provides the facility to specify initialization procedures for the agent. This code is executed upon the instantiation of an agent, in the processor environment where the agent has been loaded.

A class description is composed of an optional class-name and a description body. The name is any acceptable LISP symbol. The class name, if specified, must be unique. If named, the new class description will be saved for future use. (See Section 9.3.)

Class descriptions can be built up from other existing class descriptions by using the *copy-of* statement. The new class will then be an identical copy of the class specified in the copy-of statement, including all default values and initialization code, which can be modified with the *delete*, *new* or *import* statements. Optionally, the user can describe a new class without copying an existing class. The copy-of statement, if used, can only occur once and must be the first statement of the description body.

The *import* statement provides a means to selectively pick desirable features of other class descriptions. Whatever initialization values were associated with the feature in the other class description, will be in effect in the new class description. Importable features are *engine*, *attributes*, and *functions*.

The *new* statement defines a new feature for the agent. The feature can be an attribute or part of an attribute (as in the case of the *plans*, *roles*, *goals* and *skills* statements), a function, or an engine. A feature may take an optional initial value, or, in the case of a function, must have a function definition which must be a valid LISP function. The *plans*, *roles*, *goals*, and *skills* statements specify agent qualifiers. The features specified by the *import* and *new* statements take precedence over the same features if described earlier in the description.

An initial value for any attribute may be specified. The initial value can be any LISP s-expression that can serve as an argument to the LISP function `eval`. The *plans*, *roles*, *goals* and *skills* statements and the attribute *acquaintances* take special initial values.

To remove features from the class description, the *delete* statement has to be used. This is useful in removing unwanted features when using a copy of another class description. It is not an error to delete a non-existent feature.

The description can contain arbitrary LISP code that will be executed when an agent of this class is instantiated. This code is placed in the *init-code* statement. The code forms no part of the agent, and cannot be referenced within the agent. However, it may refer to functions defined within the agent. Refer to Section 8 for an example of the ADL code for describing some CNET agents.

7.2 Accessing Attributes and Functions

Attributes are bound to their values and may referenced in the same way as any LISP variable, with one exception. If the attribute was not previously assigned a value by a programmer or by the MACE system, it will always have been bound to `nil`. The function `set-attribute-value` is used to assign values to the attributes. Local functions, accessible only within the agent, are used like any other LISP functions. However, they can only be defined with the `NEW-DEFUN` clause of the Agent Definition Language, as they may be inheritable.

8 An Example: the Contract Net in MACE

As one of several proof-of-concept studies for MACE, we implemented a general version of Reid Smith's Contract Net (CNET) problem-solver [5, 30] in MACE, and applied it (as did Smith) to the N-Queens problem [35]. In this implementation, each CNET node is composed of an organization of four MACE agents, including a communication manager, a contract manager, a task processor, and a knowledge base. Each of the 4 agents comprising one CNET node may be mapped onto any MACE processor, depending on decisions of the allocator. Our CNET N-Queens experiment includes a manager agent called NQ which creates and initializes CNET agents and monitors the experimental runs. NQ issues building and allocation requests, and responds to requests from users via the MACE user-interface and command-interpreter agents. Using MACE on our 16-node Intel Hypercube we have run experiments using either four or six CNET nodes to solve problems of allocating 4 or 6 queens. Running a 6-Node by 6-Queen experiment requires 28 MACE agents, including 24 for the CNET, one NQ experiment manager, a user-interface, a monitor, and a MACE command interpreter. All of these agents execute concurrently.

Tables 1 and 2 present a selection of code from the MACE 6.0 description language to illustrate its use for describing some agents from our reimplementation of the Contract Net. These fragments illustrate the following points:

- Pattern descriptors.
- The use of pattern triggers.
- The use of acquaintances and acquaintance attribute retrieval.
- The use of organizations.

- Reasoning about allocation of work using models of other agents.
- Description and instantiation of individual classes and of organizations.
- The use of REF and USE skill descriptors.

9 An Environment for Building and Experimenting with Agents

We are working toward constructing MACE DAI systems which appear to be *organizations which solve problems.* Programming such a system means creating and managing an organization of problem-solvers of varying complexity and power. The programmer of such a system should develop some of the specifications and technologies for solving the organization's problems, much as the founders of a human organization begin with some initial product ideas and (perhaps innovative) ways of producing them. In this task, the programmer should have the help of a group of specialists who are experts in the known techniques - information management, quality control, domain knowledge for particular applications, etc.

Specialized system agents should take on themselves much of the burden of actual system construction and operations - the programmer's task should be to establish initial forms, directions and policy. Thus eventually programmers should become managers of interdependent processes, rather than specifiers and builders of entire systems in detail. They specify staffing requirements with *job descriptions* - high-level descriptions of the special and general requirements of agents, described in terms of *roles*, *goals*, or *skills.* Some agents come with basic skills in their domain of expertise. For example, a *Quality Control* agent might have some general knowledge about testing techniques, statistical quality control, error-discovery procedures, etc. The quality-control agent should be able to accommodate new goals by learning new procedures, or by forming alliances with other agents who can achieve those goals (E.g., hiring or building new associates).

Programmers direct and organize agents by specifying their relationships to one another and by issuing *policies* and *goals* - high level directives and constraints on behavior. Agents carry out and interpret the policies and goals in the light of their own local circumstances. In this way, agents are problem-solvers, and systems are adaptive without explicit detailed control by a central actor or overseer.

Many organizations have similar components (sub-units). Programming should begin with a set of existing agents. One set of starting agents might be:

- *Garbage collectors* which clean up space and recover "used" agents;
- *Fault-control specialists* which handle errors and recognize anomalies;
- *Quality control specialists* which verify procedures or agent descriptions;
- *Organization specialists* which establish structures of communication, roles, authority, etc.
- *Personnel specialists* which associate existing agents with appropriate goals, and which order the construction of new agents;
- *Communications specialists* which display information to users and interact with them, which manage global address directories, etc.

Table 1: Partial **Node Manager** Definition in **MACE**

; **Node Manager** is an organization with 3 members:

```
((name CNODE-1)
 (import ENGINE from CNET-DEF)
 (acquaintances
  (CNODE-1
   [roles (SELF CNODE-MANAGER)]
   [goals ((TASK-ANNOUNCEMENT $)
           (ANNOUNCE-TASK $)
           (ANNOUNCED-AWARD $)
           (DIRECTED-AWARD $)
           (ACCEPTANCE $)
           (REFUSAL $)
           (FINAL-REPORT $)
           (INFORMATION $)...)]
   [skills ([(DIRECTED-AWARD $spec)
             (use (send-to-agent
                     @(roles CONTRACT-MANAGER ORG-MEMBER)
                     '(DIRECTED-AWARD $spec)))]
            [(ANNOUNCE-TASK $task-spec)
             (ref (acquaintance with
                   [roles (CNODE-MANAGER POTENTIAL-NQ-CONTRACTOR)]
                   [goals (TASK-ANNOUNCEMENT $)])
              (by-sending '(TASK-ANNOUNCEMENT $task-spec)))]...)] ...)

  (CNODE-2 [roles (CNODE-MANAGER POTENTIAL-NQ-CONTRACTOR)]
           [goals ((TASK-ANNOUNCEMENT $)...)] ... )
  (CNODE-3 [roles (CNODE-MANAGER POTENTIAL-NQ-CONTRACTOR)...])

  (CONTRACT-MGR-1
   [roles (ORG-MEMBER CONTRACT-MANAGER)]
   [goals ((TASK-ANNOUNCEMENT $)
           (ANNOUNCED-AWARD $)
           (ELIGIBILITY-REPORT $)
           (DIRECTED-AWARD $)
           (ANNOUNCE-TASKS $)...)]...)

  (KBASE-MGR-1 [roles (KBASE-MANAGER ORG-MEMBER)])
  (TASK-MGR-1 [roles (TASK-MANAGER ORG-MEMBER)]) ) )
```

Table 2: Partial **Contract Manager** Definition in **MACE**

```
((name CONTRACT-MGR-1)
 (import ENGINE from CNET-DEF)
 (acquaintances
  (CONTRACT-MGR-1
   [roles (SELF)]
   [goals ((TASK-ANNOUNCEMENT $)
           (ANNOUNCED-AWARD $)
           (ELIGIBILITY-REPORT $)
           (DIRECTED-AWARD $)
           (ANNOUNCE-TASKS $)...)]
   [skills ([(TASK-ANNOUNCEMENT ?contract ?eligibility-spec)
              (ref (acquaintance with
                       [roles KBASE-MANAGER CO-WORKER])
                (by-sending
                    '(CHECK-ELIGIBILITY ?contract
                       ?eligibility 'task-announcement)))]
            [(ANNOUNCED-AWARD ?contract ?sender ?task-spec)
              (use [(ADD-CONTRACT ?contract)
                    (ref (acquaintance with
                          [roles TASK-MANAGER CO-WORKER])
                     (by-sending
                       '(EXECUTE-TASK ?contract
                          ?sender ?task-spec)))])])] ...)

  (CNODE-1 [roles (CNODE-MANAGER MY-ORG-MANAGER)])
  (KBASE-MGR-1 [roles (KBASE-MANAGER CO-WORKER)])
  (TASK-MGR-1 [roles (TASK-MANAGER CO-WORKER)]))

 (new  RANKED-TASK-LIST)
 (new  AWARDED-CONTRACT-LIST)
 (new  RESULT)
 (new  STRATEGY smallest-first)

 (new-defun STORE-BID-IF-ELIGIBLE (REPORT)...))
 (new-defun ACCEPT-OR-REFUSE-DIRECTED-AWARD ...))

 (init-code
      (pattern-trigger '(ELIGIBILITY-REPORT ?report)
       (cond ((TASK-ANNOUNCEMENT ?report)
              (STORE-BID-IF-ELIGIBLE ?report))
           (t (ACCEPT-OR-REFUSE-DIRECTED-AWARD ?report))))))
```

- *Domain specialists* which handle particular parts of particular domain-related problems.

We already have incorporated simple versions of some of these agents (domain, communication and personnel specialists) into MACE.

9.1 Building Agents

The MACE system provides a *Class Description Database* of stored class descriptions which can be used in the creation of new class descriptions. Class descriptions are created and modified by issuing *design specifications* to the MACE system's *builder* agent, which is responsible for accepting commands to add, delete and modify class descriptions into or from the class database. Design specifications are messages, and they may be issued by any MACE agent. Typically they are issued by the user via a user-interface agent tied to the builder agent.

For example, when the *builder* receives a message

```
(design-spec ((NAME CNODE) ...) )
```

it describes and installs into the class database the class called *CNODE*. Other messages including delete-class and modify-class are provided to manipulate the class database. These messages have an option whereby the result status is sent as a message to the invoking agent.

Agent class-descriptions are not executable. From class descriptions, executable instance agents must be created. The builder agent also accepts *construction orders*, which are messages which cause it to construct agents, given their class description. The class descriptions can be given directly, or a reference can be made to a named, class description stored in the class-database. For example

```
(construct CNODE-7 CNODE)
```

creates an instance of the agent class *CNODE*, called *CNODE-7*. The builder agent uses the allocation agent mentioned earlier to map the agent to a node.

At any time after instantiation any user-created agent can be destroyed, by itself or by any other agent. There is no restriction on who kills whom; a valid address suffices. To kill (CNODE-7 CNODE 1 ORPHEUS) any agent (including the agent in question) can call the kernel function:

```
(kill-agent (CNODE-9 CNODE 1 ORPHEUS))
```

The agent's status is maintained by the MACE system in the *status* attribute. The status attribute can take any one of the following values: *new, inactive, active, waiting* and *stopped*. These values have meaning to the engine shell, which is an extension of the MACE system into the agent.

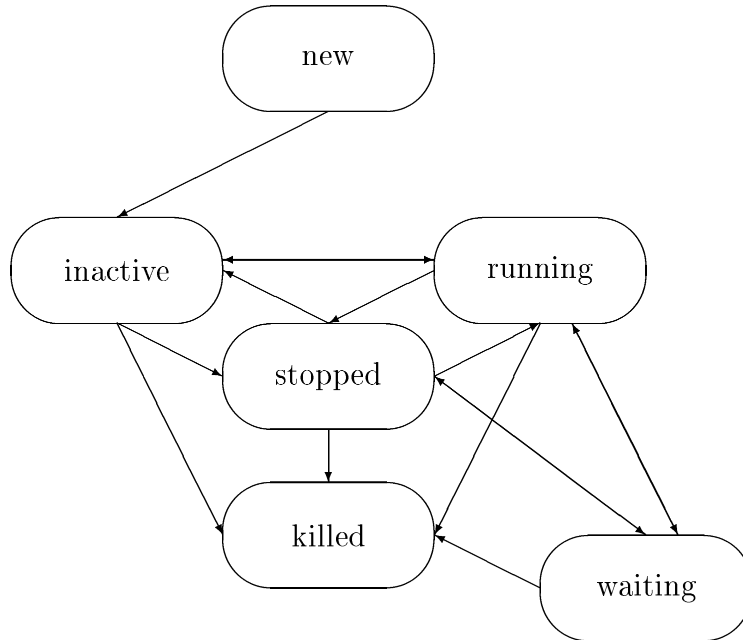

Figure 1: Agents' State Diagram

When an agent is first created it is mapped onto a (logical) processor by the Mace kernel, its status is set to *new*, and the engine shell starts initializing it. If the agent has been defined with pre-defined acquaintances, their addresses must be filled in by the engine-shell before the agent is free to send messages to them. At creation time, the pre-defined acquaintances may not yet exist. The agent's status remains as *new* until all its pre-defined acquaintances have also been created. The agent's engine is not activated during this period. When the addresses of all its acquaintances have been found, the engine shell will eval the *init-code* of the agent, if any, and change the status to *inactive*.

If the pre-defined acquaintances do not exist, the agent will never become active. One of the reasons for this status value is to be able to resolve circular acquaintance dependencies without a deadlock. By introducing the *new* state, the instantiation of agents no longer depends upon the existence of their acquaintances; only the actual activation of the agent does so. The instantiation of agents only depends upon the builder agent. This provides an elegant solution to the problem of agents 'waking up' knowing the addresses of their acquaintances.

The agent remains in the *inactive* state until it receives a message. Its status then changes to *active*. When the agent is active, its *engine* will be eval'ed. It remains in the active state until it deactivates itself.

An active agent may request the engine shell for some event to be monitored, and then block itself. In this case, it does not become *inactive*, rather, it goes to a *waiting* state. In the waiting state, an agent does not get activated by incoming messages; only when the event, or events, that were being monitored

occur, will the agent be activated once more. In a waiting state, messages are held for the agent; they are delivered when it leaves the waiting state.

The *stopped* state temporarily prevents an agent from being executed. None of its messages are lost; they are saved and delivered to the agent when it is returns to whatever state it was in prior to being stopped. An agent can be made to enter the stopped state through the *command-interpreter* agent. (See 9.3) This state is useful when debugging and tracing agents.

9.2 Instrumentation and Control

MACE is instrumented to allow for measurements of the characteristics of its agents during experimental runs. Message traffic, queue and database sizes, work done by an agent (in terms of amount of real time the agent is active, or number of agent evaluations), and load on a processor node (in terms of number of agents or total scheduler cycle time) are common measurements. Load factors are used by the allocator agent in determining where to load new agents. Instrumentation is carried out through the system's uses of the basic demon and imp mechanisms of MACE. Trace and instrumentation agents issue demon and imp requests, and collect and analyze the results.

The MACE simulator gives an added measure of control for debugging and repeatable experiments. While the standard (un-simulated) MACE executions involve random execution and message delivery delays, the simulator's behavior is deterministic. The simulator will model arbitrary logical processor speeds and interconnection topologies. It will also simulate errors in the communications and processing in the network, and agent breakdowns, with a specified probability rate. Like the un-simulated MACE system, the simulator will gather execution-time behavioral statistics, including message routing and traffic loads, agent and processor breakdowns, timing and synchronization information, and behavioral trace information for behavioral interpretation.

9.3 System Agents and Tools: The Predefined Community

The MACE system has several predefined system agents which work with a user to display information and interpret commands. This community includes:

- **User-Interfaces** A collection of *user-interface* agents control all terminal I/O. They are responsible for all communication to and from terminal-like output devices. Agents can request input or send output to a terminal or graphics device through these agents. When an agent requests input from a user, it send a message to the user-interface agent which, prompts the user. The input is sent to the requesting agent in a message. For example, this is the way the command-interpreter agent gets commands from a user. Multiple input requests from different agents can be explicitly serialized by synchronization with the user-interface, or agents can open different I/O windows for simultaneous communication.
- **Monitor**: Agents can send communications which are intended for transmission outside the MACE system, and which require no responses, to *monitor* agents. Monitor agents open trace and log files, display trace and error information on user-interface windows, and selectively filter instrumentation information for experimental tests.

- **Command-Interpreter**: System-related requests or queries can be directed to the *command-interpreter*. In addition, this agent serves as a link between the user and the user-defined agents; the user can communicate with all other agents through this agent. The command-interpreter accepts a number of commands from the user, including queries about system status, requests to monitor agent activities by tracing their actions, and requests to schedule agents on the system. Messages to the command interpreter may come from any MACE agent; typically they are issued by a user via a user-interface agent.
- **Builder**: The *builder* agent accepts agent design or construction orders. Given a design specification for an agent, it adds it to the class database it maintains. A construction order results in the creation of a new agent of a specified class (that may exist in its class database, or may be specified directly in the order using the ADL). In either case the requesting agent can be notified of the result of its request via a message.
- **Directory**: The *directory* agent maintains the address list of all the agents in the MACE system. Any agent in the system can query the directory agent for the address of other agents in the system. The directory agent provides a means for agents to get additional information about their environment, and to expand their model of the world.

During typical MACE experiments we have run, the user is communicating through the user-interface to the command interpreter agent, while an experiment manager agent is creating agents and making agent allocation requests, domain-related agents are solving the domain-related problems, and monitor agents are collecting and displaying data on the experiment.

10 Experiences and Conclusions

We have validated MACE by building several communities of interacting agents, based on existing Distributed AI paradigms. We have found MACE to be useful in coping with the diverse requirements and specifications of various problem solving domains.

Our re-implementation of the Contract Net has been described above. In addition, we have built three distributed production systems, based on the rule-agents concept - a dataflow-like approach to production systems [11]. Under this approach, each rule is an agent, and there is no global database or inference engine. Rules fire in parallel and send tokens to other rule-agents. Rule-agents themselves may query a user via the system user-interface and monitor agents for messages (i.e. data) which are not produced by other rules.

We have also built a simple distributed blackboard system, which implements a parallel arithmetic calculator. In the distributed blackboard, knowledge sources and blackboard levels are MACE agents. Different knowledge sources reduce subexpressions, perform simple arithmetic operations, and synthesize results [9].

Other experimental systems built using MACE to date include a system for simple visual scene analysis using schemas [17], a prototype multiple-agent diagnosis system [10], and a distributed planner for two cooperating robots [20].

Based on our experiences building these systems, we believe that several features of MACE have proven their worth or feasibility:

- *The creation of multi-grain systems.* We used MACE to build distributed systems of different granularity, ranging from the large grained contract net, a medium grained distributed blackboard, to the small grained rule agents. The description language of MACE is powerful enough to capture all of these; it permits the description of agents of varied complexities through the same language, providing the ability to use stepwise refinement in the construction of large and complex systems. Within a system, there are no constraints on all agents having the same level of granularity.

- *The construction of Multi-Mode sytems.* In MACE the user can control the interpretation of messages by providing specific engines to implement problem-specific syntax or message passing paradigms. This flexibility has been demonstrated in the implementation of MACE system agents which interpret messages using different syntactic structures by using different engines.

- *Abbreviated references.* Acquaintance selectors provide the agent with the ability to abbreviate references in a meaningful way that does not detract from the problem at hand. Reasoning about the division of work can be done at a high level without having to know about how the mapping between agents and their individual skills is performed. This bore fruit in the idea of high level organizations, permitting reference to a group of agents with shared goals, as a single entity. Organizations, used successfully in the implementation of the distributed blackboard and CNET, were proved to be a simple and effective solution to tasks which require the complex distribution of effort among a group of related agents.

- *Control through simulation.* The MACE system has been implemented in a simulated as well as in an actual parallel environment. The simulator, by its sequential nature, provides an alternative, easier to grasp, environment for the less experienced MACE programmer to familiarize himself with the MACE system and MACE programming. The initial versions of both the distributed blackboard and the contract net were developed in the simulated environment and then ported to the parallel implementation of MACE.

- *Debugging tools.* One of the goals of the MACE system is to provide an environment conducive to the development of distributed systems. We found the tracing and instrumentation features to be invaluable while debugging and testing the various modules which were developed. The most useful, at that time, was the ability of the tracing system to maintain a temporal link across concurrently executing (on separate processors), yet highly interdependent agents.

- *Description database.* MACE provides a database containing the description and specification of existing agent classes. The agent description language permits selective retrieval of these descriptions in the specification of new agent classes. Programmers using MACE tend to remake agents by example rather than explicitly coding a new one every time.

- *Inter-machine flexibility.* The flexibility of MACE is demonstrated by the fact that MACE recognizes and is able to communicate with MACE systems on other machines, as well as support a multi-user environment. Communication with other machines is provided by different drivers that are invoked by the use of a specific machine name in the destination address of a message. The advantage of this system is that the machines themselves may be *virtual machines* and thus communications can be tailored to practically any interface. This approach was used to extend, with minimal effort, the linkage of the tracing of the contract-net to an external graphic display. (This was used at a demonstration of the CNET running on MACE at the 1986 AAAI conference.) Extending MACE to a multi-user system merely called for a new user-interface agent for each user.

- *Multiple user interfaces.* We have built MACE experiments which included simultaneous line-oriented, window-based, and color graphic user-interfaces, each controlled by different agents, and

running on different processors. This flexibility and multiplicity provides many compelling possibilities for research into multi-modality human-machine interaction.

Despite our successful use of MACE, we have found difficulties with certain aspects. We have also discovered several pressing research needs. These are being addressed for future versions of MACE.

- *Understanding DAI system execution.* Perhaps the most salient research issue to emerge from our MACE experiments is the pressing need for display and analytic techniques for understanding and debugging the parallel execution of heterogeneous DAI systems. Concurrency, problem-domain uncertainty, and non-determinism in execution together conspire to make it very difficult to comprehend the activity in a distributed intelligent system. Debugging and control become extremely difficult. While our tracing and display systems were very helpful, we urgently need graphic displays of system activity linked to intelligent model-based tools which help a developer reason about expected and observed behavior. We expect to apply the fruits of our research on multi-agent diagnosis here.

- *Inflexible mapping to nodes.* MACE provides no dynamic transfer of agents from one processing node another. Once an agent has been instantiated, it is impossible to reinstantiate it on another node or machine while maintaining local state. Though we have devised an dynamic load-balancing algorithm [27] and implemented the load measurements it needs, dynamic load balancing is currently impossible. This restriction will be removed in the next version of MACE.

- *Isolating basic operations for standard engines.* What are the high-level primitives for expressing computations in distributed AI systems? We would like to incrementally increase the expressive power of MACE by providing a better set of standard engines which incorporate actions which we have found repeatedly useful. These include asynchronous context maintainance using result-frames, and a language structure for expressing prospective reasoning primitives using *plan points*. We have found that much of the agent activity in the systems we have built involves combining asynchronously-arriving information from several sources with fairly limited computation (usually data extraction and insertion). We call this *form-filling*. MACE should have form-filling primitives.

- *Speed.* We have been only marginally concerned with speed of execution; our focus has been on parallel problem-solving architectures. Still, speed has emerged as a growing concern. Our experiments have been carried out using MACE on TI Explorer LISP machines and a 16-node Intel Hyprecube running Gold Hill Computers' CCLISP. All MACE code on both machines is compiled Common LISP. In the 6-node, 6-Queens CNET experiments, parallel MACE on the Hypercube gave us a 7-fold speedup (real time) over the MACE simulator on the Explorer. This comparison does not account for the added parallel simulation overhead on the Explorer, and so is of limited utility. Still, we need much greater speed per processor in a parallel architecture to provide adequate performance.

- *Adherence to Common LISP.* To insure the portability of MACE, we adhered to the use of only standard Common LISP. However, Common LISP does not provide any standard error trapping mechanism. We experienced the inconvenience of restarting the entire system due to some agent having an error in its code. We decided to compromise on the portability of MACE and use the error trapping facility provided by our Common LISP environment. However, we see this more as a reflection on Common LISP than on MACE.

With new generations of more powerful parallel hardware, distributed AI will become more possible and more attractive. We believe MACE will provide a useful step toward productive experimental DAI research.

11 Acknowledgements

We are grateful to Janet Coates, Eric Ho, Lunze Liu, Matt Staker, and Dit-An Yeung for their help building and documenting the experimental application systems and/or earlier versions of MACE. We are also grateful to Gary Doney and Fernando Tenorio for helpful comments on earlier drafts. We thank the Intel Corporation and Prof. Kai Hwang for their support. Part of this research was funded under Lawrence Livermore Laboratory contract number 86-5987.

References

[1] G. Bekey and L. Gasser. Task Allocation Among Multiple Robots. Paper presented at 1987 JPL Conference on Space Telerobotics, Pasadena, CA., Jan 20-22, 1987.

[2] D. D. Corkill, K. Q. Gallagher, and K. E. Murray. GBB: A Generic Blackboard Development System. *Proceedings AAAI-86*.

[3] P. Cohen and H. Levesque. Speech Acts and Rationality. *Proc. 23rd Meeting or the Association for Computational Linguistics*. Chicago, 1985.

[4] S. Cohen, J. Conery, A. Davis, and S. Robinson, *OIL PRogramming Language Reference Manual*, Technical Report, Schlumberger Palo Alto Research, 1985.

[5] R. Davis and R.G. Smith. Negotiation as a Metaphor for Distributed Problem-Solving. *Artificial Intelligence* Vol. 20, pgs. 63-109, 1983.

[6] L. Gasser. The Integration of Computing and Routine Work. *ACM Transactions on Office Information Systems*, 4:3, pgs. 205-225, July, 1986.

[7] L. Gasser. Negotiated Order: Concerted Action in Multi-Agent Systems. DAI Group Research Note 3, Distributed AI Group, Dept. of Computer Science, USC, 1986.

[8] L. Gasser, C. Braganza, and N. Herman. MACE: Experimental DAI Research on the Intel Hypercube. In *Proceedings of the Second Conference on Hypercube Multiprocessors*, Knoxville, Tn, September, 1986.

[9] L. Gasser, C. Braganza, and N. Herman. Implementing Distributed AI Systems Using MACE. *Proc. IEEE Third Conference on AI Applications*, February, 1987.

[10] L. Gasser, M. Staker, and E. Ho. A Prototype Multi-Agent Diagnosis System. DAI Group Research Note 19, USC Distributed AI Group, Dept. of Computer Science, USC, 1986.

[11] L. Gasser and M.F. Tenorio. *Rule-Agents: A Distributed, Object-Oriented Approach to Production Systems Using MACE.* DAI Group Research Note 4, Distributed AI Group, Dept. of Computer Science, USC, 1986.

[12] E.M. Gerson and S.L. Star. Analyzing Due Process in the Workplace. *ACM Transactions on Office Information Systems*, 4:3, pgs. 257-270, July, 1986.

[13] B. Hayes-Roth. A Blackboard Architecture for Control. *Artificial Intelligence,* 26(2): 251-321, March 1985.

[14] C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Artificial Intelligence,* pp. 323-364, 1977.

[15] C. Hewitt. Offices are Open Systems. *ACM Transactions on Office Information Systems*, 4:3, pgs. 271-287, July, 1986.

[16] R. Hill and L. Gasser. Notes on Prospective Reasoning in Multi-Agent Systems. DAI Group Research Note 22, USC Distributed AI Group, Dept. of Computer Science, USC, 1986.

[17] E. Ho, E. Kiralay, and V. Natarajan. Schema-Based Recognition of Environments. DAI Group Research Note 24, USC Distributed AI Group, Dept. of Computer Science, USC, 1986.

[18] V. R. Lesser and D. D. Corkill. The Distributed Vehicle Monitoring Testbed: A Tool for Investigating Distributed Problem Solving Networks. *AI Magazine,* pp. 15-33, 1983.

[19] H. Lieberman. An Object-Oriented Simulator for the Apiary. *Proc. 1980 Lisp Conference*, 1980.

[20] L. Liu. A Distributed Planner for Two Cooperating Robots. DAI Group Research Note 21, USC Distributed AI Group, Dept. of Computer Science, USC, 1986.

[21] D. McArthur and P. Klahr. *The ROSS Language Manual.* Rand, 1982.

[22] S. Narain, D. McArthur and P. Klahr. Large-Scale System Development in Several LISP Environments. In *Proceedings IJCAI-83,* pp. 859-861. International Joint Conference on Artificial Intelligence, 1983.

[23] H. P. Nii. CAGE and POLIGON: Two Frameworks for Blackboard-based Concurrent Problem Solving. Report No. KSL 86-41, Knowledge Systems Laboratory, Computer Science Department, Stanford University, Stanford, April 1986.

[24] E.H. Pattison, D.D. Corkhill, and V.R. Lesser. *Instantiating Descriptions of Organizational Structures.* Technical Report 85-45, Computer Science Department, UMASS, Amherst, MA. , November, 1985.

[25] J. Pavlin. Predicting the Performance of Distributed Knowledge-Based Systems: A Modeling Approach. In *Proceedings AAAI-83,* pp. 314-319. American Association for Artificial Intelligence, 1983.

[26] J. Pavlin and D. D. Corkill. Selective Abstraction of AI System Activity. In *Proceedings AAAI-84,* pp. 264-268. American Association for Artificial Intelligence, 1984.

[27] D. Persinger. A Distributed Load Balancing Algorithm for MACE. DAI Group Research Note 23, USC Distributed AI Group, Dept. of Computer Science, USC, 1986.

[28] J. Peterson and A. Silberschatz. *Operating System Principles.* Addison-Wesley, 1983.

[29] R. Schank and K.M.Colby. *Scripts, Plans Goals, and Understanding.*selec Lawrence Erlbaum, 1977.

[30] R. G. Smith. *A Framework for Distributed Problem Solving.* UMI Research Press, Ann Arbor, Michigan, 1981.

[31] M. Stefik and D. Bobrow. Object-Oriented Programming: Themes and Variations. *AI Magazine,* 6(4), Winter, 1986.

[32] A. Strauss. *Negotiations.* San Francisco: Jossy-Bass, 1978.

[33] P. Wegner. Language Paradigms for Programming in the Large. 1985. Working Paper, Dept. of Computer Science, Brown University.

[34] D. Weinreb, D. Moon, and R. Stallman. *LISP Machine Manual.* Massachusetts Institute of Technology, 1983.

[35] D. Yeung. Using CNET on the iPSC. DAI Group Research Note 20, USC Distributed AI Group, Dept. of Computer Science, USC, 1986.

Contents

1 Introduction **1**

1.1 MACE Goals 1

1.2 General Description of MACE 2

1.3 Related Work 3

2 What is a MACE Agent? **4**

3 What Agents Know **5**

3.1 Models of Other Agents in the World 5

3.2 Getting Acquainted 6

4 How Agents Sense Their World **8**

4.1 Monitoring Events: Demons, Event-Monitors and Synchronization 8

5 How Agents Act: Engines **9**

5.1 Sending Messages 9

5.2 Issuing Monitor Requests 10

6 Organizations **11**

7 Describing Agents: The ADL and Description Database **12**

7.1 Agent Description Language 14

7.2 Accessing Attributes and Functions 15

8 An Example: the Contract Net in MACE **15**

9 An Environment for Building and Experimenting with Agents **16**

9.1 Building Agents 19

9.2 Instrumentation and Control 21

9.3 System Agents and Tools: The Predefined Community 21

10 Experiences and Conclusions **22**

11 Acknowledgements **25**