Design of a Concurrent Language for Distributed Artificial Intelligence

Published in Proc. of FGCS'88,
ICOT, Tokyo, Japan, Nov.-Dec. 1988, Vol. 2, pages 755-762

Jacques FERBER

LAFORIA, Tour 45
Université Paris VI
4, Place Jussieu
75252 Paris Cedex 05
FRANCE
litp!jf@inria.inria.fr.uucp

Jean Pierre BRIOT

Equipe mixte XEROX/LITP
Université Paris VI
4, Place Jussieu
75252 Paris Cedex 05
FRANCE
litp!briot@inria.inria.fr.uucp

Abstract:

This paper presents the design of a concurrent language for knowledge representation, MERING IV. It will be used as a support for Distributed Artificial Intelligence (DAI) applications, where problem solving is achieved by a society of "intelligent" agents. Its architecture is based on a multi-layered approach: the lowest layer is a multi-processor machine, and the highest layer is a multi-agent description language. The system incorporates reflective abilities, a description language, daemons facilities for declarative programming. All these features are unified into a general model of computation based on concurrent active objects and message passing. The model is fully distributed to reflect the underlying parallel machine on which it is currently under development.

Keywords:

Distributed AI, open systems, multi-layered architecture, object, actor, message, asynchronous, meta-object, reflection, daemon, sprite, multi-agent

1 Introduction

This paper presents a Distributed Artificial Intelligence (DAI) project, currently under progress at University of Paris 6. Our project consists in a multi-layered and open-ended [Ferber 84, Hewitt 86] actor language for knowledge representation, called MERING IV, which will be used for describing multi-agent systems.

In this □paper we will focus on the MERING IV architecture. The first language level is a very simple actor language based on the actor model of computation [Hewitt 76] [Agha 86]. The second level extends this basic model with higher level programming constructs and reflective capabilities. The third level provides rules and daemons to describe decision making process.

The hardware layer is a multiprocessor machine (currently a network of workstations is used for prototyping, this summer we start a multiprocessor implementation on a new generation hypercube).

The MERING IV language is to be used as an experimental basis for various kinds of organization models to manage distributed problem solving, such as the Contract Net [Smith 81]. We aim at providing a testbed for such experiments as does the MACE system [Gasser et al. 87]. In this paper we intend to describe the architecture of our model, and to point out our design decisions.

2 Design decisions

2.1 Modelling a social organization for solving problems

In order to model intelligent systems which could solve human tasks, the AI discipline has generally considered human activity as a starting point. Thus traditional AI systems intend to model a single processing agent with knowledge, expertise and reasonning abilities. The world mechanism, where several, possibly conflicting, viewpoints can coexist is one of the most common solution for managing large and complex knowledge bases. Parallelism is also introduced in such systems in order to speed up reasonning process. But these systems remain one-piece and sequential by nature.

A more recent approach, and currently very active field of research, considers intelligent activity as a cooperation of a collection of multiple active agents. An intelligent system is modelled through a community of experts cooperating to achieve a goal. The study of human societies gives us various models of such communities. We believe that it is easier to study and model the activity of a social community than the intelligent activity of one man. This field is currently known as Distributed Artificial Intelligence (DAI [DAI 87]), and many contributions have already been made in terms of models of architectures (hierarchical relations between the agents, communication protocols...) [Minsky 86], [Kornfeld and Hewitt 80], [Hewitt and DeJong 82], [Smith 81].

2.2 A system for social metaphors experiments

We started a project at University of Paris 6 to design and implement a system for DAI. This system is made of three parts:

- The **language component**, MERING IV, is an actor oriented model of both knowledge representation and concurrent computation.

- The **multi-agent component** is based on a system, called PALADIN, which provides for the definition of social organizations (e.g. the Contract Net [Smith 81]), using an intentional model of communications based on "speech acts" theory [Searle 69].

• □The **implementation component** is composed of a virtual machine for distributed computation, currently simulated in LISP and whose prototype is under development on multi-processor hardware.

2.3 Criterias

Here are the main requirements for our system:

- **simplicity and modularity:** the system should be as simple as possible in terms of concepts required. Therefore we rely on the object-oriented programming (OOP) metaphor which also provides modularity. Any kind of element of the system is called an **object**. The objects communicate through exchange of **messages**. Objects and messages are the two fundamental concepts of OOP.

- **concurrency and distribution**: concurrency is not only a speedup opportunity but the expression of concurrent activities in a community of processing agents. Every agent in the simulated community has a local knowledge and performs some local processing. The computation model and the implementation must match these requirements. Therefore we chose the actor model of computation as a basic model for its clean concurrency model. Concurrency will be implemented by real parallelism on a multiprocessor distributed machine (an hypercube configuration).

- **multi-agent testbed environment**: the main goal of the project is to design an environment to specify and simulate the activity of a community of agents which are autonomous objects communicating by message passing. Agents can be seen as mini problem solvers with their own local knowledge and a set of facilities for interacting with other agents. Each agent can only communicate with a subset of the whole agent family, called its acquaintances on which it has some representation (beliefs).

- **reflective ability**: the system should be able to model its structure and execution. This is achieved through distributed reflection, i.e. by associating a meta-object to every object. The meta-object represents both the structural part of the object and its execution mechanism. Reflection is used for describing implementation details, monitoring objects activities, defining higher behavioural constructs, etc...

2.4 Design decisions

We will now focus on the architecture of our system and our main design decisions.

2.4.1 Communicating active entities

It is now well known that object-oriented programming (OOP) is a good means to design high level and modular systems. However the use of this metaphor for parallelism is still a younger activity, although it is a very active research field [OOCP 87]. Our system relies on an OOCP model, where everything is an active object which can communicate with other objects through exchange of messages. We will stay with the **object** terminology along the paper even if the basic computation model is the actor model, and the system is aimed at describing active agents.

2.4.2 The three aspects of an object

Object oriented languages can be seen from three different points of view:

- The **structural** aspect: an object can be considered as a specific data structure related to a general model which is an abstract definition of the structure itself. We call it the structural aspect of objects.

- The **conceptual** aspect: objects can be used for representing external things and events, using frame-based representations.

- The **agent oriented** aspect: objects can send and receive messages, and behave in a social way by interacting and cooperating with other agents.

Most object or actor oriented languages either cope with only one of the point of views, or try to find a middle-term between all aspects without considering them in an integrated way. For instance, actor languages are more concerned with the third aspect, most object-oriented languages are specialized in the first aspect (e.g. consider the way the CLOS language [CLOS 87] renounces the notion of message passing and local methods for the more abstract data type aspect of generic functions), and frame languages (like FRL, SRL+, KEE,...) only deal with the conceptual one.

On the contrary, the MERING IV language gives a unified approach of the three different aspects:

- each object is a **module** (see it either as an autonomous world or as an implementation of an abstract algebra) which contains other objects and local functions,

- each object is a **frame** that possesses a set of slots representing properties and binary relations. Slots are complete objects and can hold slots as well. In fact all attributes of an object are represented as slots.

- Each object is a small **agent** which communicates with other agents by message passing.

2.4.3 Layered architecture

The system architecture is based on a multi-layered approach.

2.4.4 Level 0: multiprocessor implementation

Our target machine is a multi-processor machine, either a 32 nodes Intel IPSC hypercube machine or a network of transputers. We are currently designing a virtual machine to support the execution of the level 1 assembly language. We are now using a prototype virtual machine implemented in CommonLisp on a workstation.

2.4.5 Level 1: an actor assembly language

We selected the actor model of computation [Hewitt 76] as the foundation of the level 1 language. This choice was dictated by the necessity of a simple and clean kernel model to express concurrent execution of active entities.

Level 1 is the lowest language level in the system. It is directly translated into virtual machine instructions. The primitive part of level 1 consists in a set of rock bottom objects (numbers, strings...) and the corresponding primitive operations on these low-level objects, i.e. simple arithmetics (+, *,..) predicates (=, >, <,..) and conditionals (if).

Level 1 supports definition of simple objects (they are called **executors**) and message passing operations among them. At this level every transmission is unidirectional, i.e. without any reply. Non trivial computations are expressed through the use of continuations (objects to whom the reply will be sent and which will carry on the computation) [Hewitt 76]. At level 1 every continuation is made explicit.

2.4.6 Level 2: a programming language

The level 2 language augments the underlying layer with higher level constructs such as functional application, description and abstraction capabilities, programming environment modules, and reflective abilities.

To define elaborate computation at level 1 the user should explicitly describe every step of computation in terms of a chain of continuation objects. Level 2 will give the user higher constructs which will be compiled in terms of explicit continuation handling. For instance, functional programming is made possible at that level.

At level 2 every object belongs to a (conceptual) class, which is also an object. A description system expresses every object as the instance of a class. Objects defined "by hand" at level 1 can be fully described at level 2.

Reflection [Reflection 88] is introduced in the language in a distributed (local) way by associating a meta-object to each object. A meta-object gives a complete structural representation of the object it represents (for instance, its mailqueue, its message handler...).

Finally as a programming language, programming environment modules (such as an exception mechanism) are defined at level 2. This last part won't be discussed in this paper.

2.4.7 Level 3: a multi-agent description system

The level 3 introduces declarative constructs such as rules and daemons. It gives the ability to describe intelligent agents and their behaviour as a set of rules which are fired either upon reception of messages or modification of the internal state. These rules work on a forward chaining basis. Rules and daemons are used for defining knowledge based agents whose behaviour is based on an intentional content (i.e. goals, plans) related to their local knowledge and specific competences. Such agents are part of the PALADIN system [Ferber 87] and defined using MERING IV.

3 The first layer: the basic execution model

3.1 The actor model of computation

The model we adopted for our kernel language is based on the actor model of computation [Hewitt 76], and whose first representative programming language was the PLASMA language [Smith and Hewitt 75].

3.1.1 The reaction principle

An object (called an **actor** in the model) represents a computational expertise. The meaning of "computational" is quite general and may cover arithmetic operations, knowledge processing, problem solving, etc... An object is requested to apply his expertise by sending him a request communication. This communication may include informations about the processing to be done.

The expertise of an object could be decomposed into a knowledge part and a behavioural part.

- The **knowledge part** is a set of other objects it knows about (they are called its **acquaintances**), i.e. with which it can communicate. Examples of acquaintances are personal data bases, representations of outside world entities, other objects he could distribute the work to, etc...

- The **behavioural part,** also called a **script**, is the description of what the object will do when receiving a **message**. The activity of an object starts upon reception of a message from another object (possibly himself) and triggers some actions.

A script consists in a collection of methods, each being associated to a message pattern, a symbol called a **selector**. One (and only one) method will be selected among them when accepting a message. Selection is done by pattern matching on the set of methods indexed by the same selector. Each method is defined as a set of **actions**. Possible actions are performing primitive operations (for instance arithmetics or tests), sending messages to known objects, and creating new objects in order to delegate some computations to them.

3.1.2 Concurrency aspects

Here are the main characteristics of the computation model:

- **asynchronicity**: the requests of computation are sent asynchronously from an object to another one, this means that none has to wait for both to be ready for communication. This model comes from the observation of a postal service. A **mail system** buffers the communications sent between objects. This includes the routing subsystem (using interprocessor communication in a multiprocessor implementation) and a distributed set of **mailboxes**, one per object. The merits of asynchronism and buffering are discussed in [Agha 86]. We chose asynchronicity as the kernel communication model between entities, although we may express synchronous calls in terms of asynchronicity and use of continuations at level 2.

- **concurrent processing of messages:** there are two kinds of objects in the language. Most objects are **unserialized**, i.e. their will never change their knowledge part (state). Thus unserialized objects may process several messages concurrently. **Serialized** ([Hewitt and Atkinson 79]) objects may change their knowledge part, thus they need to process one message at a time. As we will see later, **cell objects** are the only serialized objects in the language.

- **arrival of messages**: for serialized objects, incoming messages are ordered into the receiver's mailbox which behaves as a queue. This means that only one message at a time could reach the mailbox. The protector of this assumption is called an **arbiter** (between two concurrent messages).

- **concurrent execution of actions**: when accepting a message, a method is selected and then executed. A method could be executed sequentially as a sequence of instructions or as an unordered set of actions executed concurrently. The latter solution, which does not limit the use of parallelism, has been chosen. Sequentiality may be expressed in the model through causality (as seen below).

- **non determinism**: we reject assumptions such as sequential execution of a method, we also reject assumptions on ordering arrival of messages or actions because we intend to keep the model as much opened as possible. However we won't leave an unbounded non determinism, here are a few restrictions.
  - We assume the garanty of delivery, i.e. every message sent to an object will be accepted by it in a finite time (we don't consider here a possible rejection of a message in case of unrecognized message pattern, leaving error management out of the scope of this paper). This means that the arbiter must be fair [Clinger 81].
  - We restrict non determinism for method selection. This means that in contrast to parallel logic programming languages, there is no concurrent selection of clauses for a given goal. We assume the set of patterns of messages (selectors) to be ordered. Pattern matching process occurs sequentially among these patterns. This restriction is motivated by the necessity of having only one method selected. We could have chosen a commitment discipline such as in and-stream parallel logic programming languages [Takeuchi and Furukawa 86] but we prefer here a more conservative approach. This selection issue is still an open problem. We are waiting for significant conclusions before drawing a firmer approach.

- We assume causality, i.e. an event created by another event cannot occur before its cause. This will be used at level 2 to express sequentiality as a chain of continuations representing the steps of the computation.

3.1.3 Advantages

- **simple and clean semantics**: the model combines the advantages of functional programming (purity) and object-oriented programming (i.e. it could model changeable state with special mutable objects, called **cells**, see below).

- **inherent concurrency**: there is no construct for adding parallelism to a sequentially based language. Concurrency is implicit as a kind of default mechanism.

- **concurrency of message processing**: most objects, except cells, are unserialized, i.e. their state and script never change. As a consequence they may process several messages concurrently and may be freely copied among the processors. Serialized objects may process only one message at a time.

- **fluidity**: computation is fully distributed among objects, who in turn distribute it among their acquaintances. The use of reply destinations enhances fluidity of objects because an object computes only a single step of a computation and then defines the object which will carry on the remaining computation.

- **disponibility**: because of this fluidity and the absence of waiting states, objects keep being ready for communications. This principle is fundamental when we will define metaobjects at level 2.

3.1.4 Executor objects

Executor objects are the primitive kind of objects at level 1. They perform a simple and unique kind of processing. At level 2 we will define more complex objects that could perform several kinds of processing requests (for instance deposit or withdraw for a bank account) but at level 1, executors may process only one pattern of message.

Here is a first example of a printer object which outputs (by calling a primitive format function) the object (expression) it receives. This argument is prefixed by the PRINT selector:

```
(define print-executor
    (=>> (print ?expr)
       (format t "=> ~S~%" ?expr)))
```

(=>> ...) is an expression defining an executor object. It could be activated by sending a message:

```
(print-executor <= (print "hello world"))
```

3.1.5 Continuation objects

Continuation objects are a special kind of executor objects. Their selector is always REPLY. They are so much used in the system that there is a special definition expression for them. Here is the definition of the previous printer as a continuation object:

```
(define printer
    (=> (?expr)
       (format t "=> ~S~%" ?expr)))
```

About the syntax change, the previous =>> arrow is replaced by a => arrow, and the REPLY selector is implicit.

Here is the traditional example of a factorial computation in the continuation style:

```
(define factorial
    (=>> (call ?n) (reply-to ?c)
       (if (= ?n 0)
             (?c <= (reply 1))
             (self <= (call (1- n))
                        (reply-to (=> (v) (?c <= (* n v))))))))
```

The computation of (factorial n) is performed when n > 0 by computing (factorial (1- n)) whose result is sent to a continuation which will multiply it with n and send this product to the current continuation. (The values of ?n and ?c will be closured in the continuation object, due to lexical scope discipline).

```
(factorial <= (call 10) (reply-to printer))
```

will print the value of factorial(10). The REPLY-TO selector indicates the continuation (or reply-destination [Yonezawa et al. 86]) in a message.

3.1.6 Primitive objects and operations

The level 1 language includes low level objects such as numbers and primitive classes. Level 1 also defines primitive operations on low level objects, and conditionals (=, <, >, if, cond ...). They will be directly implemented in the virtual machine for efficiency reasons.

3.1.7 Cells for side effects

With executors and continuations, the model is side effect free and may take advantage of parallelism. However it should be able to model changeable states. Cell objects are introduced for this purpose. Cells are the only objects where side effect may take place. A cell object is serialized because its state may change during time. A cell object recognizes two message patterns, SET and GET:

```
(a-cell <= (set 10))

(a-cell <= (get) (reply-to printer))
```

To ensure the sequentiality of these two requests on the cell (we want to consult the contents of the cell AFTER its updating), we will use continuations:

```
(a-cell <= (set 10)
   (reply-to
      (=> (?dummy)
            (a-cell <= (get) (reply-to printer)))))
```

Note that the DUMMY parameter indicates that the value received is not significant (and is not used).

A higher level mechanism for concurrent transactions with a cell is proposed at level 2. Cells will be used as the foundation for slots introduced also at level 2.

3.1.8 From level 1 to level 2

Level 1 is not a language intended for end users because of its low level descriptive and programming constructs, but its main purpose is to serve as a target for the compiler from level 2 downto level 1. The architecture of levels 1 and 2 is, in that respect, analog to usual actor architectures, for instance the Acore / Pract duality [Manning 87].

4 The second layer: the programming language

Level 2 gives to programmer a higher level programming language. Level 2 is somehow analog to the ABCL/1 language [Yonezawa et al. 86] [OOCP 87] in terms of level constructs.

4.1 Implicit continuations and functional constructs

At level 1 every continuation needs to be explicitely given. Level 2 introduces implicit continuations. The <<= message passing form indicates an implicit continuation. For instance:

```
(printer <= (reply (+ (a-cell <<= (get)) 2)))
```

is equivalent to:

```
(a-cell <= (get)
    (reply-to
       (=> (?val) (printer <= (reply (+ ?val 2))))))
```

There is a compilation of those constructs into level 1, with a mechanism analog to the Scripter compiler [Lieberman 83] [DeJong 86] or Acore compiler [Manning 87].

We may now define the factorial function in the following way (by using descriptions which will be introduced later):

```
(define fact
    ($a Function
       (methods
           ((=>> (apply [?n])
             (if (= n 0)
                   (reply (* ?n (fact (1- ?n))))))
```

We consider that `(fact 10)` is equivalent to `(fact <= (apply [10]))`. Remark that a function object, may accept other combinations than just reply requests.

4.2 Transactions with cells

Cell objects are dedicated at isolating and encapsulating side effects. Actually, the serialization mechanism is not sufficient for encapsulating transactions with cells. Suppose that we define the incrementation from outside the cell (typically in an object owning a cell as one of its acquaintances):

```
(a-cell <= (get)
   (reply-to
         (=> (?val) (a-cell <= (set (1+ ?val))))))
```

But two INCR requests sent to the same cell may overlap and yield an invalid value. Thus, we need to lock A-CELL until the transaction (computation requests) is finished, to avoid any interference with another transaction. This is done by sending a CLAIM message to the cell.

The idea is that a locked cell will still accept any messages except another claim request. In such a case, the second CLAIM request has to wait (the message is kept in a queue belonging to the cell and will be processed after unlocking). Here is an example of such incrementation:

```
(a-cell <= (claim)
    (reply-to
       (=> (?dummy)
         (a-cell <= (get)
            (reply-to
              (=> (?val)
                 (a-cell <= (set (1+ ?val)))
                    (reply-to
                        (=> (?dummy)
                            (a-cell <= (release-claim)))))))))
```

The level 2 provides for a primitive construct called LET-CLAIM inspired from the claim techniques used in concurrent accesses of data bases [Deen 77], which generates the above chain of continuation. Then the previous code can be written at the level 2 as:

```
(let-claim (a-cell)
    (a-cell <= (set (1+ @a-cell))))
```

where `@a-cell` is equivalent to `(a-cell <<= (get))`.

Here is the definition with descriptions (see below for the definition of descriptions) of a counter to illustrate this new construct:

```
(define counter
    ($a class
       (super Object)
       (with
          (contents ~ Number))
       (methods
```

```
(=>> (incr)
   (let-claim (contents)
         (contents <= (set (+ @contents 1))))
(=>> (reset)
   (let-claim (contents)
         (contents <= (set 0)))
(=>> (consult) (reply-to c)
    (c <= (reply @contents)))))
```

4.3 Classes and descriptions

4.3.1 Classes and Instances

The MERING IV model is based on a class/instance distinction, with subtle differences. Whereas a class in the standard model (for instance the ObjVlisp model [Briot and Cointe 87], which is the purest for defining classes and metaclasses) is the structural model of an instance, a class in MERING IV is an intensional abstraction of a set of objects, i.e. the **conceptual model** of an object.

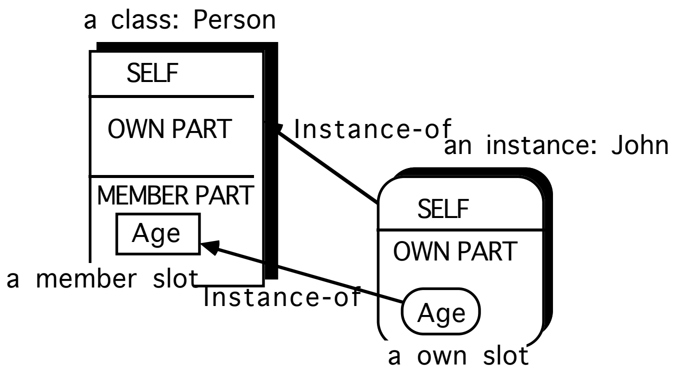
Classes are also objects and are instances of higher order classes, called **Hyperclasses**. We have not chosen the word "metaclasses" for such entities, because the metaclass construct in OOP is more concerned with the structural definition of classes, where hyperclasses are used for representing general properties of classes. A hyperclass is no more than an intensional definition of a set of classes, i.e. a class of classes. These classes are conceptual instances of hyperclasses. The class and instance terminology is only used within the conceptual aspect of objects, which is in contrast with the usual abstract data type conception.

Objects representation capabilities consist in a collection of slots that describe the characteristics of real entities. There are two different kinds of slots. Following the KEE terminology, we will call them **own slots** and **member slots**. Own slots represent individual attributes, whereas member slots represent general properties of individuals. For instance the age of John can be described by an own slot which is specific to the JOHN object. But the fact that all persons (and more generally any concrete thing) can have an age, can be represented by a member slots associated to the class PERSON (or more generally the class CONCRETETHING). But classes are objects by themselves, and can have own slots as well. For instance the fact that the cat species is a numerous species is a property of the CAT class, not of its individuals.

Thus, member slots are described at the class level, and instantiated as own slots at the instance level. In order to remain homogeneous, in MERING IV slots are complete objects: member slots are considered as subclasses of the SLOT class, and own slots are instances of member slots.

For practical reasons, member slots and own slots are located in different zones of an object: member slots are situated in a zone called **member part** and own slots in a zone called **own part**. In a given class C, all elements of the member part are classes, and are instanciated as instances in the own part of C instances. For instance, the Age attribute of persons is represented as a member slot PERSON.AGE (i.e. a subclass of SLOT) in the class PERSON, and the Age attribute of John is represented as an instance JOHN.AGE of

PERSON.AGE and situated in the own part of the JOHN instance. The following diagram shows the relationship between member slots and own slots:



4.3.2 Descriptions

Descriptions are expressions which are used for the definition and retrieval of objects in the system. Thus descriptions denote objects.
Syntactically, descriptions are expressions of the form:

$$(\$a\ C\ \ (Key_1\ E_{11}\ ..\ E_{1k})\ ...\ (Key_n\ E_{n1}\ ..\ E_{nl}))$$

where C is an expression which reduces to a class, $Key_1..Key_n$ are keywords (e.g. With, Methods, Def, etc..), and E_{ij} are general expressions. For instance:

```
(define John
  ($a Person
    (with
       (age = 23)
       (job = ($a Job
                 (with
                    (earnings = 20000
                         (with (Unit = Dollar)))))))))
```

is a description of somebody whose age is 23 and earns 20000 dollars. By dereferencing this description, it is possible to create an object corresponding to this description:

```
($a person (with ... )) = #<a person : obj-234>
```

Classes can be defined by descriptions. For instance, the definition of the class PERSON can be defined this way:

```
(define Person
   ($a Class
       (super Object)
       (with-member-slots
          (age ($a NumericalSlot))
          (job ($a Slot
                   (with
                      (range = Job)))))))
```

A description is a level 1 construct, which is translated into level 0 message passing. For instance, the description of the above class PERSON is transformed into the level 0 expression:

```
(Object <= (subclass)
  (reply-to
    (=> (?o1)
       (GlobalWorld <= (HasOwn 'Person ?o1))
       (NumericalSlot <= (subclass)
         (reply-to
            (=> (?s1)
               (?o1 <= (HasMemberSlot 'age ?s1)))))
       (Slot <= subclass
         (reply-to
            (=> (?s2)
               (?o1 <= (HasMemberSlot 'job ?s2))
               (?s2 <= (Setvalue 'range Job)))))))
```

Note that the construction of the age and job slots are concurrently handled.

4.4 Reflection

Reflection is the ability for a system to model itself in terms of static description (by accessing a representation of itself) and dynamic execution (by controlling the context of its execution). The goal of reflection, as defined by [Smith 82], is to define intelligent programs capable of reasoning about and act upon themselves. Reflection is not restricted to static (structural) description as in some description systems but extended to dynamic control by allowing meta-entities to monitor the activity of entities. This simple mechanism is intensively used at the second and third level of the language as a basic mechanism for dynamically redefine object structure and system organization, planning object activities, etc...

As in 3-KRS [Maes 87], every object has a meta-object which is its representation at the meta-level. As meta-objects are objects, they can themselves be represented at the meta-meta-level by a meta-meta objects. This leads to a virtually infinite tower of meta-objects. In order to remain finite, meta-objects are created in a lazy way, i.e. by creating them only when needed. Meta-objects and objects are causally related in such a way that all operations made at the meta-level have an impact to the structure and the behaviour of the basic-level objects.

The basic concept of object reflection developped by P.Maes for 3-KRS has been extended in two directions:

- in the direction of the class/instance model of object oriented languages (note: 3-KRS does not distinguish between classes and instances. Thus there is just a derivation link which is used for both inheritance and instanciation). This extension has yielded a new notion of metaclasses, which is different from the standard notion of metaclasses developped in Smalltalk-like object oriented languages.

- in the direction of concurrent programming in order to cope with the definition of parallel message interpretation at the meta-level.

We will use the Smith's notation [Smith 82]: for any object O, $\uparrow$O gives the meta-object of O, and $\downarrow$O yields its referent. When O is a meta-object, then $\downarrow$O gives the basic level object O represents. Objects and meta-objects are related in such a way that for any object O and O', and any transformation T from $\mathbb{O}$ to $\mathbb{O}$ (where $\mathbb{O}$ is the set of all objects),the following formulas holds:

1) $\downarrow\uparrow O = O$
2) $\uparrow O = O' \Rightarrow O = \downarrow O'$
3) $\uparrow(T(O)) = (\uparrow T)(\uparrow O)$

The 1) and 2) formulas show that reification (the process of going from objects to meta-objects) and denotation (the process of going from the representation to the referent) are inverse operations; the 3) formula shows that all operations done at the meta-level are causally connected to operations done at the basic level. Furthermore, it shows that reification is a morphism between basic level elements to meta-level elements.

4.4.1 Static reflection

In MERING IV, as in 3-KRS, all objects have a related meta-object. For instance, the object JOHN:

```
(define John
    ($a Person
       (with
         (age = 23)
         (job = ComputerScientist))))
```

has a corresponding meta-object, which is an instance of the class METAINSTANCE which is the class that describes the basic structure of all the normal instances:

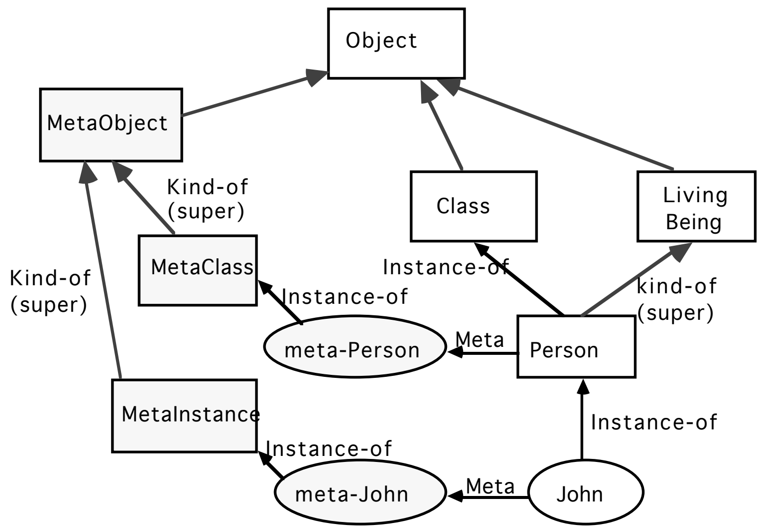
```
↑John =
    ($a MetaInstance
       (with
          (name = 'John)
          (isa = Person)
          (meta = Self)
          (ref = #!John) ;; the referent of John
          (context = GlobalMetaWorld)
          (OwnSlots = [($a Person.age
                          (with
                             (Source = John)
                             (Target = 23)
                             (localname = 'age)))
                        ($a Person.job
                          (with
                             (Source = John)
                             (Target = ComputerScientist)
                             (localname = 'job))) ])
            (MessageBuffer  =  [...]  );;  the  list  of  incoming
messages
       .... ))
```

Slots of the meta-object ↑JOHN are descriptions of the internal structure of the object JOHN. For instance, the 'ref' and 'meta' slots of JOHN describes the Referent and Meta link of JOHN, not of ↑JOHN.

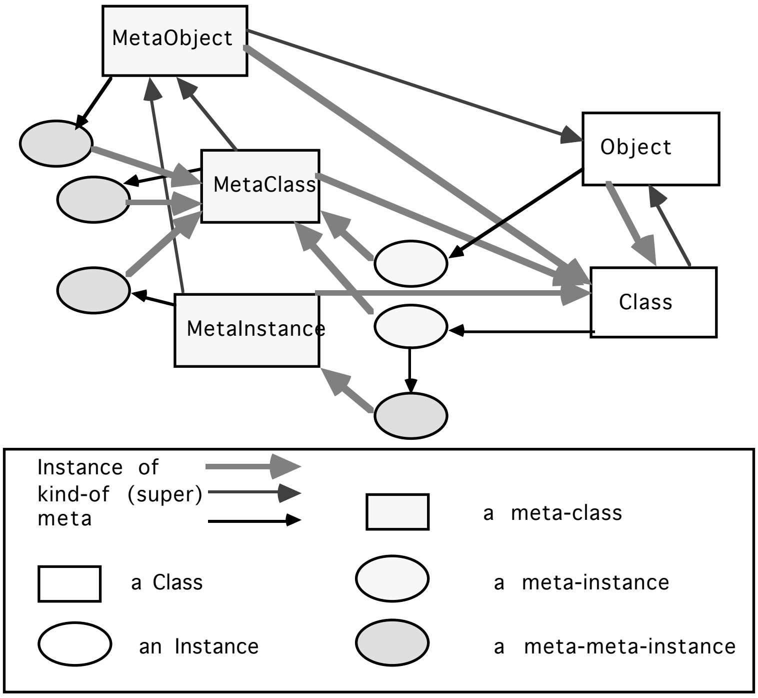
In order to implement the reflection principle into a class/instance model, the basic model of instance/class/metaclass analysed in [Briot and Cointe 87] had to be modified in the following way:

- instances meta-objects are instances of the class METAINSTANCE.
- classes meta-objects are instances of the class METACLASS.

For instance, if JOHN is an instance of the class PERSON, ↑JOHN is an instance of the class METAINSTANCE, and ↑PERSON is an instance of the class METACLASS. The following diagram illustrates the situation:



One can see that CLASS is the class of all classes, whereas METACLASS is the class that describes all the meta-objects that represent classes. In order to give a brief account of the underlying structure, the following figure shows the relations between the root objects: OBJECT, CLASS, METAOBJECT, METAINSTANCE and METACLASS. Most meta-instances are only virtually present, i.e. they are not created by the system, but the general behaviour takes them into account, and they can be created at any time if the system needs it:



4.4.2 Dynamic reflection

Dynamic reflection is the ability for a process to represent its own behaviour at the meta-level. In MERING IV, transmissions can be represented at the meta-level as the sending of a specific message (HANDLEMESSAGE) to the receiver's meta-object, where the argument of the message is an instance of the class COMMUNICATION:

```
↑(A <= (f a1 .. an) C) =
     (↑A <= (HandleMessage
                  ($a Communication
                     (with
                        (MessageSelector = f)
                        (MessageComponents = [a1 .. an]))))
                        (Sender = Self)
                        (Continuation = ↑C)))))
```

The overall mechanism can therefore be represented in the language itself at the meta-level. For instance, the class METAINSTANCE contains the methods HANDLEMESSAGE and INTERPRETMESSAGE which is the definition, in the language itself of the real computational methods that are used for managing communications (the character '~' is used for constraining the type of variables):

```
(define MetaInstance
    ($a Class
       (super MetaObject)
       ...
       (methods
          (=>> (handlemessage ?c~($a Communication))
               (MessageBuffer <= (Add ?c)))
```

```
(=>>(InterpretMessage
        ?c~($a Communication
              (with
                 (MessageSelector = ?sel)
                 (MessageComponents = ?l))))
                 (sender = ?exp)
                 (continuation = ?cont))))
   (let ((meth (self <<= (lookup ?sel))))
      (if meth
         (meth <= (apply ?l ?c))
         (errorHandler <= (InvalidMethod ?sel ?c))))))))
```

5 The third layer: rules as daemons

The third level consists in a multi-agent description language, adding reasoning capabilities to the basic actor model of computation. This level is not actually separated from the second level: it merely adds new classes and entities to the system, in order to accomodate for declarative reasoning using special kind of **daemons**. Their interest in problem solving with actors has been shown for "open systems" [Kornfeld and Hewitt 80]. Daemons modelizes the actual effect of a rule, by declaring the conditions for which a daemon has to be triggered and its effect on the object it looks upon.

5.1 Definition of daemons

A daemon is a tuple,

<InitialState, NextState, Cont>

where InitialState and NextState are descriptions of two different states of the contextual entity, i.e. the entity where an instance of a daemon has been defined (i.e. the states of the SELF entity), and Cont is a continuation. One can see a daemon as a kind of a state transition describing an entity transformation. Whenever the contextual entity E reaches the initial state IS, the daemon is fired. It modifies the entity E into another state NS, the next state, and replies to the continuation, with the instance of the daemon itself as a value.

The external syntax of a daemon is:

```
($when
       (name <name>)
       (in <entity>)
       (with <state-description>)
       (and <expressions>)
       (then <state-description>)
       (activate <continuation-body>))
```

where <state-description> is a list of triple:

```
(<attribute-name> = <value>)
```

where <value> can either be a variable or an expression which returns a value, and <expressions> is a list of expression which returns a boolean value. The <name> of the daemon is not used by the system, and serves only for explanations and debugging purposes. Here is an example of a daemon with no continuation,

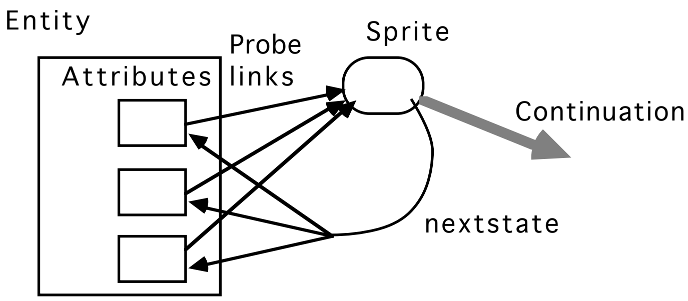
```
($when
    (name adultp)
    (in Person)
    (with
       (age = ?x))
    (and
       (>= ?x 18))
    (then
       (adult = #t)))
```

which says that all persons having more than 18 years old are adult. Its semantic can be informally described by the following sentence:

If I am a person and my age is greater or equal to 18, then I am truly an adult.

5.2 Installation of a daemon

A daemon is installed into an entity by setting special links, called **probe links**, into its slots, in order to propagate the values into the daemons. Whenever a new value is assigned to one of the slots, the daemon is fired. If the initial state condition is satisfied, the new state is propagated along the nextstate line, and the customer is activated, as shown in the following figure:



Thus the two important messages a daemon can respond to are FIRE, that fires the daemon, and INSTALL that set up a daemon. Here is the (simplified) version of the DAEMON class:

```
(define Daemon
    ($a Class
       (super Object)
       (with-member-slots
          (InsideOf ~ Object)
          (InitialState ~ StateDescription)
          (Conditions ~ Expression)
          (NextState ~ StateDescription)
          (Cont ~ Continuation)
          (SlotsIn ~ ($aList (of Slots)))
          (SlotsOut ~ ($aList (of Slots)))
          (Variables ~ ($aList (of LocalVariables)))
          (State ~ ($oneOf 'Active 'Inactive)))
       (methods
```

```
              (=>> (fire)
                   (let ((env (@InitialState <<= (Check))))
                       (if (and env (@Conditions <<= (Check))))
                           then
                             (@NextState <= (Perform env))
                             (@Customer <= (reply self))))))
              (=>> (Install ?ent)
                 (Broadcast ?x @attributesIn
                       (x   <=   (PropagateTo   (=>   (?v)   (self   <=
(Fire)))))))
           ... )))
```

Daemons can be installed at any time, in order to describe dynamic activities. For instance, if an entity E receives a message from an entity A asking to perform a specific task (for instance to recall A of something), when the entity E has a specific status:

```
...
    (methods
       (=>> (remind ?me ?something ?status)
          ($when
             (with
                (status = ?status))
             (do
                (?me <= (recall ?something)))))
```

Thus when the object E receives the message (Remind A), the daemon is installed on the slot STATUS, checking for equality with the value of ?STATUS. It the condition is satisfied, then the message (RECALL S) where S is the something to be recalled, is sent to E.

6 Implementation issues

The architecture of our system is fully distributed to support this model. We will use a hypercube as a physical realization of this distribution. We are currently designing a prototype of the system on a workstation. A prototype virtual machine written in LISP defines the support for level 1 which is completed. We have gained experience of a prototype implementation of ABCL/1 as a model for level 2. A previous sequential model of level 3 has also already been implemented. This is now the time to unify all pieces of the puzzle in a uniform manner (analog to Act2 [Theriault 83] as a first step towards a Prelude system incorporating Act1, Ether and Omega). All examples presented in this paper have been tested on a prototype implementation.

The debugging issue is a very important issue which has not been touched yet in this paper. There is a growing activity in the domain of debugging concurrent object programs [Manning 87], [Honda and Yonezawa 88]. We except our layered approach to improve the debugging scheme (a program should be debugged at the definition level, not at the underlying execution level) and the use of reflection to help fighting this huge task.

7 Related Work

We chose the actor metaphor for open systems initiated by C. Hewitt [Hewitt and DeJong 82] as a major source of influence and inspiration, and specially the design of actor systems along the Apiary architecture [Hewitt 80] with Act1 [Lieberman 1983] [OOCP 87], Act2 [Theriault 83] and Act3 [Agha 86] [Manning 87].

The ABCL project [Yonezawa et al. 86] [OOCP 87] gave us a major source of experience and ideas for the level 2 language.

Major works on reflection by B. Smith [Smith 82] and P. Maes [Maes 87] gave us the foundations to incorporate reflection in level 2. Works by [Briot and Cointe 87], [Ferber 87] [Ferber 88] gave us a good experience on structural, procedural and conceptual reflection and its application to knowledge programming. Following the work of P. Maes on reflection in sequential object-oriented computation, there is now a growing activity in the field of concurrent execution. Mandala [Furukawa et al. 84] was one of the first concurrent languages to incorporate metaprogramming facilities. [Tanaka 87] and [Watanabe and Yonezawa 88] are some representatives of the current activity. The latter paper develops a model very close to ours. We thank A. Yonezawa for communicating us the latest results of his project.

The Omega language [Attardi and Simi 81] was a reference for our description system. The idea of sprites proposed by W. Kornfeld [Kornfeld and Hewitt 80] has inspired us the notion of daemons in objects at level 3.

The MACE system is our main reference as a DAI system, we thank Les Gasser for some discussions about evolution of DAI systems.

The Orient84/K system [Ishikawa and Tokoro 86] proposes a model for concurrent knowledge programming. Although not directly a source of inspiration, it has shown us the feasibility of a concurrent language dedicated to knowledge representation.

8 Conclusion

We presented the model of a system aimed at experimenting Distributed Artificial Intelligence (DAI). Our model is based on a notion of active objects communicating through exchanges of messages. We focused on the important design decisions we made for our model: actor style parallelism, importance of slots, reflection in class/instance systems and in concurrent object oriented systems, etc... A layered architecture was presented in a bottom-up manner and the status of the currently ongoing implementation was reported.

9 References

[Agha 86] G. Agha, "Actors - A Model of Concurrent Computation for Distributed Systems," MIT Press, 1986.

[Attardi and Simi 81] G. Attardi and M. Simi, "Semantics of Inheritance and Attribution in the Description System

Omega," AI Lab Memo N°642, MIT, Cambridge MA, USA, 1981.

[Briot and Cointe 87] J-P. Briot and P. Cointe, "A Uniform Model for Object-Oriented Languages Using the Class Abstraction," IJCAI'87, Milano, Italy, August 1987.

[Briot and Yonezawa 87] J-P. Briot and A. Yonezawa, "Inheritance and Synchronization in Concurrent OOP," ECOOP'87, LNCS N° 276, Springer Verlag, 1987.

[Clinger 81] W. Clinger, "Foundations of Actor Semantics," Phd Thesis, AI-TR-633, MIT, Cambridge MA, USA, May 1981.

[CLOS 87] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. Keene, G. Kiczales and D. A. Moon, Common Lisp Object System Specification, ANSI document X3J13, March - July - November 1987.

[DAI 87] "Distributed Artificial Intelligence," edited by M. N. Huhns, Pitman - Morgan Kaufman, 1987.

[Deen 77] S.M. Deen, "Fundamentals of Data Base Systems," Mc Millian Press, London, England, 1977.

[DeJong 86] P. DeJong, "Compilation into Actors," draft, AI Lab Memo, MIT, Cambridge MA, USA, 1986.

[Denis 88] C. Denis, "An Object-Oriented Exception Handling System for an Object-Oriented Language," to be presented to ECOOP'88, Oslo, Norway, August 1988.

[Ferber 84] J. Ferber, "Mering: An Open-Ended object oriented language for knowledge representation" ECAI'84, Italy, September 1984.

[Ferber 87] J.Ferber, "Des Objets aux Agents: une Architecture Stratifiée," Actes du 6ème colloque RFIA, Antibes, France, November 1987.

[Ferber 88] J. Ferber, "Conceptual Reflection and Actor Language," in [Reflection 88].

[Furukawa et al. 84] K. Furukawa et al., "Mandala: A Logic Based Knowledge Programming System," FGCS'84, ICOT, Tokyo, Japan, 1984.

[Gasser et al. 87] L. Gasser, C. Braganza and N. Herman, "MACE: A Flexible Testbed for Distributed AI Research," Chapter 5 of [DAI 87].

[Hewitt 76] C. Hewitt, "Viewing Control Structures as Patterns of Message Passing," AI Lab Memo N°410, MIT, Cambridge MA, USA, December 1976.

[Hewitt and Atkinson 79] C. Hewitt and R. Atkinson, "Specification and Proof Techniques for Serializers," IEEE Transactions on Software Engineering, Vol. SE-5 N° 1, 1979.

[Hewitt 80] C. Hewitt, "The Apiary Network Architecture for Knowledgeable Systems," Lisp Conference'80 , Stanford U., Palo Alto CA, USA, August 1980.

[Hewitt and DeJong 82] C. Hewitt and P. DeJong, "Open Systems," AI Lab Memo, MIT, Cambridge MA, USA, 1982.

[Hewitt and Lieberman 83] C. Hewitt and H. Lieberman, "Design Issues in Parallel Architectures for Artificial Intelligence," AI Lab Memo N°750, MIT, Cambridge MA, USA, 1982.

[Hewitt 86] C. Hewitt, "Offices are Open Systems," Transactions on OIS, Vol. 4 N° 3, July 1986.

[Honda and Yonezawa 88] Y. Honda and A. Yonezawa, "Debugging Concurrent Systems Based on Object Groups," to be presented to ECOOP'88, Oslo, Norway, August 1988.

[Ishikawa and Tokoro 86] Y. Ishikawa and M. Tokoro, "A Concurrent Object-Oriented Knowledge Representation Language Orient84/K: Its Features and Implementation," OOPSLA'86, Sigplan Notices, Vol. 21 N°11, November 1986.

[Kornfeld and Hewitt 80] W. Kornfeld and C. Hewitt, "The Scientific Community Metaphor," AI Lab Memo N°641, MIT, Cambridge MA, USA, 1980.

[Lieberman 83] H. Lieberman, "An Object-Oriented Simulator for the Apiary," AAI'83, Washington DC, USA, August 1983.

[Maes 87] P. Maes, "Concepts and Experiments in Computational Reflection," OOPSLA'87, Sigplan Notices, Vol. 22 N°12, December 1987.

[Maes 88] P. Maes, "Issues in Computational Reflection," in [Reflection 88].

[Manning 87] C. Manning, "Acore: The Design of a Core Actor Language and its Compiler," Master Thesis, AI Lab, MIT, Cambridge MA, USA, August 1987.

[Minsky 86] M. Minsky, The Society of Mind," Basic Books, 1986.

[OOCP 87] "Object-Oriented Concurrent Programming," edited by A. Yonezawa and M. Tokoro, MIT Press, 1987.

[Reflection 88] "Meta-Level Architectures and Reflection," edited by P. Maes and D. Nardi, North Holland, 1988.

[Searle 69] J.R. Searle "Speech Acts: an Essay in the Philosophy of Language," Cambridge University Press, London, 1969.

[Smith 81] R. G. Smith, "A Framework for Distributed Problem Solving," UMI Rearch Press, Ann Arbor MI, USA, 1981.

[Smith and Hewitt 75] B. C. Smith and C. Hewitt, "A Plasma Primer," draft, AI Lab, MIT, Cambridge MA, USA, September 1975.

[Smith 82] B.C. Smith, "Reflection and Semantics in a Procedural Language," TR 272, CS Lab, MIT, Cambridge MA, USA, 1982.

[Stefik and Bobrow 86] M. Stefik and D. G. Bobrow, "Object-Oriented Programming: Themes and Variations," AI Magazine, Vol. 6 N°4, Winter 1986.

[Takeuchi and Furukawa 86] A. Takeuchi and K. Furukawa, "Parallel Logic Programming Languages," ICLP'86, London, England, July 1986.

[Tanaka 87] J. Tanaka, "Architecture Abstraction in GHC," ICOT report, ICOT, Tokyo, Japan, 1987.

[Theriault 83] D. G. Theriault, "Issues in the Design and Implementation of Act2," Master Thesis, TR 728, AI Lab, MIT, Cambridge MA, USA, 1983.

[Watanabe and Yonezawa 88] T. Watanabe and A. Yonezawa, "Reflection in an Object-Oriented Concurrent Language," draft, TIT, Tokyo, Japan, April 1988.

[Yonezawa 86] A. Yonezawa, "AI Parallelism and Programming," IFIP'86, Dublin, Ireland, 1986.

[Yonezawa et al. 86] "Object-Oriented Concurrent Programming in ABCL/1," A.

Yonezawa, J-P. Briot and E. Shibayama, OOPSLA'86, Sigplan Notices, Vol. 21 N°11, November 1986.

[Yoshida and Chikayama 87] K. Yoshida and T. Chikayama, "A'UM - Parallel Object-Oriented Language upon KL1," ICOT report, ICOT, Tokyo, Japan, December1987.