Distributed Object Oriented Logic Programming

Keith L. Clark Tzone I Wang
Logic Programming Section,
Department of Computing,
Imperial College
London SW7 2BZ
klc@doc.ic.ac.uk tiw@doc.ic.ac.uk

November 19, 1994

Abstract

In this paper we introduce a programming language/system DK_Parlog^{++} which is an experimental testbed for distributed applications, particularly distributed AI and distributed knowledge base applications. The language is designed to have the concurrent computation ability of the Concurrent Logic Programming(CLP) languages, the knowledge representation and problem solving ability of Prolog, both enriched with the program and knowledge structuring capabilities of Object Oriented programming.

The basic building blocks of an application are a collection of *classes* and *servers* which can be distributed over a network of machines. The location of a class or server is specified in its definition. Classes and servers have unique public names, such as `student`, `department` and `db_manager`. Classes can be linked using single inheritance. Servers are like classes except they have no instances (which is why we don not call them classes) and no inheritance links. They are used to implement publically named interface processes; for example, an interface process to a dialogue handler, or to an external data base.

This paper introduces the key features of the language illustrated through the progressive development of part of a distributed knowledge base for a University. Finally we show how a server could be used as a query manager for the knowledge base, and how the system could be extended to interface with an external object oriented data base. Via such servers the language could be used as a harness for distributed heterogeneous knowledge base systems.

1 Introduction

DK_ Parlog^{++} is a programming language/system built on top of IC-Prolog II [Chu and Clark 1993], which is a combined Parlog and multi-threaded Prolog system. The new language is an experimental testbed for distributed applications, particularly distributed AI and distributed knowledge base applications. The language is designed to have the concurrent computation ability of the Concurrent Logic Programming(CLP) languages, the knowledge representation and problem solving ability of Prolog, both enriched with the program and knowledge structuring capabilities of Object Oriented programming. DK_Parlog^{++} was inspired by Davis's Polka [Davison 1989] OO extension of Parlog [Clark and Gregory 1986] and McCabe's L&O [McCabe 1992] extension of Prolog. Crudely characterized, it is a significantly modified and distributed version of Polka relatively seemlessly integrated with an extended and distributed version of L&O.

Classes and servers The basic building blocks of an application are a collection of *classes* and *servers* which can be distributed over a network of machines. The location of a class or server is specified in its definition. Classes and servers have unique public names, such as `student`, `department` and `db_manager`. Classes can be linked using single inheritance. Both state variables and methods are inherited. That is, an instance of a subclass automatically has all the state components and methods of the instances of its super classes. Servers are like classes except they have no instances (which is why we do not call them classes) and no inheritance links. They are used to implement publically named interface processes; for example, an interface process to a dialogue handler, or to an external data base.

Default methods and state components A class must have a *create* method for generating instances. If none is given in the class definition, the DK_ Parlog^{++} system adds a default one. It also adds an extra class state variable which holds the current list of unique identifiers for all the created instances of the class.

Unique object identifiers A newly created instance can have a system generated identifier, or a program assigned identifier, such as `doc105`. Both class and instance methods are invoked by sending a message to the class or instance identifier. A system name server automatically routes the message to the machine on which the object resides. Every object, be it class, instance or server, has a unique identifier. A program assigned name that is not unique will be rejected.

Reply mechanisms Messages sent between objects can contain variables. Typically answers are given by binding these variables. A network wide distributed unification scheme implements this answer return method. Alternatively, a reply to a message `M` can be given by explicitly sending a reply message to the object that sent `M`. The sending object's identity is automatically attached to every message as it is sent and can be accessed using the *sender* keyword inside the method that handles the message. This reply mechanism does not need distributed unification.

Two kinds of methods That DK_Parlog^{++} has classes which have state variables and methods, and the ability to reply to a message by sending a message to the object identified as the source of the message, makes it significantly different from Polka and the other OO extensions of concurrent LP languages such as Vulcan[Kahn et al. 1987] and A'UM[Yoshida and Chikayama 1988]. However, the feature of DK_ Parlog^{++} which makes it significantly from these other languages, is that both class and instance methods can be of two kinds: *procedural methods* and *knowledge methods*.

Procedural methods A procedural method is a rule that links a Parlog conjunction of calls with a pattern for a received message, a message sent to the object using the message send operator `=>`. Each Parlog call in the method definition only has one solution and it is evaluated using the committed choice operational semantics [Clark and Gregory 1986]. The method can comprise any mixture of sequential and parallel computation, specified using a combination of the Parlog ',' and '&' conjunction operators. As soon as a procedural method is invoked, another message invoking a procedural method can be accepted. But that method will automatically suspend if it needs to access a state component that is changed by any previously invoked procedural method and for which the new value has not yet been computed. A message containing variables sent to an object invoking a procedural method can have *at most one* solution binding returned for its variables.

Knowledge methods In contrast, a knowledge method is a Prolog program for some predicate `p/k`[1]. It is evaluated as a sequential Prolog back-tracking search for one or more solutions to a call, `p(t1,..,tk)`, sent to the object using the message send operator `:>`. This call can have many solutions. As with procedural methods, the next call to a knowledge method can be accepted and executed by an object whilst the previous call is still being processed. This is because the Prolog component of the underlying IC-Prolog II system is multi-threaded[Chu and Clark 1993]. There are exceptions to this immediate acceptance of the next call. Certain knowledge method calls will be delayed until all existing calls have terminated. We discuss this more fully below.

Sending messages in methods Both kinds of methods can send messages to other objects (using either message send operator), or, using the *self* keyword, back to themselves.

Atomic Transactions Both message send operators can also send a sequence of messages as in:

```
(msk,...,ms2,ms1) => O
```

This guarantees that the sequence will be preserved on arrival at the object, i.e. that the messages will not get out order and that no other message sent from elsewhere will get interleaved with the sequence. This is useful for guaranteeing atomicity of a sequence of access and update messages sent to an object.

Distributed backtracking A message send

```
p(X) :> O
```

will invoke a Prolog computation which may have multiple solution paths, each resulting in a different answer binding for `X`. If this message send is executed in a procedural method, only one solution binding will be returned, that given by the first solution path. However, if it is executed in a knowledge method, the backtracking search for a solution to that method may cause multiple alternative solutions for `p(X)` to be requested and returned. When O is an object on another machine, this fetch of the alternative solutions for `p(X)` is implemented by DK_ Parlog^{++}'s support for distributed backtracking.

Finally, let us note that a parallel conjunction of knowledge method calls

```
p(X) :> O, q(Y) :> O'
```

executed in a Parlog defined procedural method, will cause concurrent execution of two threads of Prolog computation to find the first solution to each call. Where O and O' reside on the same machine, these will be time shared. Where the reside on different machines, they will be executed concurrently.

[1]We use the Prolog /k convention for indicating the number k of arguments that a message/method call should have.

Restricted access to state variables State variables have their scope restricted to the procedural methods. Therefore, only a procedural method can directly access a state variable. This is because the procedural methods are implemented as the clauses of a Parlog recursive process definition[2] and the values of state variables of an object are held as arguments of this process. The Prolog and Parlog systems are coupled [Chu and Clark 1993], but only loosely via a message passing interface. A knowledge method `M` that needs to access the value of a state variable `S` must either do this indirectly by `=>` sending a message to *self* to invoke a procedural method to access `S`, or, if the knowledge method is always invoked by a message send `M :> self` from one of the procedural methods of the object, the value of `S` can be passed as an argument in the message `M`.

State information represented as knowledge Knowledge methods can directly access state information only if it is represented as Prolog clauses. DK_ Parlog^{++} has a mechanism for allowing state information for an object to be represented in this way. Special system defined knowledge methods, that are automatically added by DK_ Parlog^{++} as extra methods to every object, allow the addition, deletion or update of a special category of *dynamic* knowledge methods for the object[3]. In addition, when a create message is sent to a class, a sequence of dynamic knowledge methods can be attached to the message to become the initial dynamic knowledge of the new instance. The knowledge methods given in a class definition are the *static* knowledge methods. They cannot be changed.

Dynamic knowledge versus instance variables The dynamic knowledge methods of an object can be used to record specific data about the object that would otherwise be held as the values of the instance state variables. For example, we can record the name `smith` of a student instance `S` in an instance state variable, `Name`, to be accessed via a call

```
        name(N) => S
```

to procedural method with the definition

```
        name(N) -> N=Name
```

Alternatively, we can record it as a dynamic knowledge method/fact

```
        name(smith)
```

to be accessed via a knowledge call

```
        name(N) :> S
```

To the other objects that need to access this instance specific data there is virtually no difference. They just need to know that access is via a procedural method or a knowledge method, in order to invoke the method with the appropriate message send.

Maintaining consistent dynamic knowledge state When we record state information using state variables we do have a guarantee of consistent access and update of the state. A procedural method executed on receipt of a message M sees the set of values for the state variables that result from the execution of *all* and *only* the procedural method messages that have reached the object before it, even though it can start executing before these other methods have terminated. Because knowledge methods of an object can also execute concurrently, we need a similar guarantee with respect to the dynamic knowledge of the object. We need to know that a knowledge method, executed on receipt of some

[2]We use a modification [Wang and Clark], of the concurrent OO implementation method of Shapiro and Takeuchi [Shapiro and Takeuchi 1983]. The major difference is that all the instance methods, whether inherited or not, become the clauses of a single Parlog process definition. Shapiro and Takeuchi, and Davison [Davison 1989], have a separate process definition for the instance methods at each level in the hierarchy.

[3]The dynamic knowledge methods are the analogue of the dynamic clauses of Prolog. They are represented as such.

message M', sees the dynamic knowledge that results from *all* and *only* the knowledge update methods whose execution was started before the receipt of the message M'. DK_ Parlog^{++} guarantees this by handling calls to the knowledge update methods in a special way. A knowledge update method is only executed by an object if all previously invoked knowledge methods have completely terminated. (This means that all the needed different solutions have been found.) In addition, no new message invoking a knowledge method will be accepted by the object until the knowledge update method has terminated.

This consistency of access to state information represented as dynamic knowledge guaranteed by DK_ Parlog^{++} is stronger than that provided by the DLP language [Eliens 1992], another distributed logic programming language in which all the methods are what we call knowledge methods. DLP only delays a state update until a solution to each currently executing method has been found. So, on backtracking to find the next solution, the method may see a different state of dynamic knowledge to that seen when the previous solution was found. We consider this to be too weak.

Active classes and DOOD classes Classes in which instances have procedural methods we call *active* classes. There instances are represented as forked Parlog processes. Active classes can be used to program distributed object oriented applications. Classes with instances that only have knowledge methods, are rather like classes of an object oriented data base. Their instances are not processes. They are named sets of Prolog facts and rules that can be deductively accessed. We call these *DOOD* (*D*eductive *O*bject *O*riented *D*ata) classes. A collection of DOOD classes can be used to build a small OO knowledge base, with a server process used to implement a query and update manager for the data base. The server can enforce atomicity of transactions that access and update knowledge held in several instances, even instances from different classes. It can also ensure that consistency constraints that relate dynamic knowledge held in instances of different classes are maintained.

Interface with other applications The message send and variable binding communication across the network use TCP/IP based communication. There are even TCP/IP communication primitives that can be called directly from a method, inherited from the underlying IC-Prolog II [Chu and Clark 1993]. This makes it easy to link in with other heterogeneous applications.

2 Active classes in DK_Parlog^{++}

We shall introduce programming in DK_Parlog^{++} by giving a small example of the definition of an active class. This comprises the two class definitions of Program 1. The second class, *field_course*, inherits from the first *course*. It adds an extra state component, `MaxStudents` to its instances, in addition to the state components `Title`, `Lecturer`, `NumStudents`, `Students` and `Level` that it inherits. Note that all but `Lecturer` and `Title` have default initial values specified in the class definitions. The `field_course` class also has an extra instance method `add_an_enrollment/2` that makes use of the `add_an_enrollment/1` method defined in the `course` class. Notice that the inherited `course` instance method is directly called, as usual in a OO languages, via a message sent to `super`.

Default class methods and state variables Neither class has any explicitly given class methods or class state variables. In the absence of a program defined method for creating instances every class has one automatically added. It is a method invoked with a message of the form:

```
        create(Oid,I1,I2,..,Ik)
```

where Oid is either a given identifier for the instance to be created, or a variable to which the system generated identifier will be bound. Each Ii is given value, or the Prolog anonymous variable _. If it is value, it is the initial value for the i'th state variable of the created instance, overriding any default value for that state variable. If it is _, the default value of that state component is given to the variable if one was specified. If not, the instance variable is left uninitialized. It must be given a value by a procedural method call.

```
class course with
{
 instance_definition
   states
     Title, Lecturer, NumStudents := 0, Students := [], Level := 1.
   methods
     run_by_lecturer(LECTURER) -> Lecturer := LECTURER.
     add_an_enrollment(STUDENT) -> Students := [STUDENT | Students],
        NumStudents := NumStudents + 1.
     current_numbers(NUMBERS) -> NUMBERS = NumStudents.
     title(T) -> T=Title.
     course_done  -> suicide.
}.
%------------------------------------------------------------
class field_course isa course with
{
 instance_definition
   states
     MaxStudents := 20.
   methods
     add_an_enrollment(Student, Ans) : NumStudents < MaxStudents ->
         Ans=ok, add_an_enrollment(Student) => super.
     add_an_enrollment(Student, Ans) : NumStudents >=MaxStudents ->
         Ans='The course is over crowded!'.
}.
instance create(doc105, 'Assembler Programming', _, _, _, 1) => field_course.
```

Program 1. A two class active object DK_Parlog^{++} program

Tha last line of Program 1 is an example of an instance creation using the system provided `create` method. It creates an instance of the `field_course` class with object identifier `doc105` and with default values for its `NumStudents` and `Students` state variables. The `Lecturer` state variable will be initially unbound and will have to be bound by sending a message to instance `doc105` such as:

    run_by_lecturer(clark) => doc105

The `run_by_lecturer/1` method can also be used to later update the identity of the lecturer. The operator `:=` used in a procedural method will change the current value of a state variable. But the new value will only be visible by the *next* procedural method to be invoked. The current method sees only the value it had at the start of execution of the method. So `:=` is not conventional assignment. It is just the way that a procedural method indicates any change of values for the state variables to be passed to the next method invocation. If there is no use of `:=` in a method, there is no change to the values of the state variables for the next method invocation. Access to the value of a state variable is via use of the unification operator =.

A class also always has an undeclared state component, initialized to `[]` which holds a list of all the names of its instances. The value of this state variable is automatically updated by the system as class instances are created and destroyed. Inside a class procedural method the current value of this variable can be accessed using the keyword *members*. This membership list can also be accessed from outside the class by sending a `members/1` message to the class. This is another method automatically added to every class. Thus, if we now execute:

    members(L) => field_course

with L an unbound variable, the answer binding returned would be `L=[doc105]`.

The class `course` also has the `members/1` method and the same message sent to it will give the same answer. Superclasses have on their membership lists all current members of their subclasses. This is useful if we want to send a message to every course. We do not have to separately retrieve the membership lists for `course` and `field_course`. We do not even have to know about the subclasses of `course`.

Instances also have some automatically added state components and methods. They have a state component that holds their own identity, accessed in any method via the keyword *self.* Another holds the identity of their class, accessed in a method using the keyword *class*, for example, as in a message send:

```
msg => class
```

It can be also be accessed by sending a `class/1` message to the instance.

```
class(C) => doc105
```

will bind `C` to `field_course`. Thus, by sending it a `class` message the 'type' of any instance to be determined. Another default method of a class is the `destroy(Oid)` method which kills an created instance with identity `Oid`. It also deletes it from the membership list of its class and all that class's superclasses.

To get rid of the instance `doc105` we would execute:

```
destroy(doc105) => field_course.
```

This will also delete `doc105` from the members lists of `course` and `field_course`. Notice that `course` also has an instance method:

```
course_done => suicide
```

This is an alternative, less abrupt way to terminate an instance. `suicide` is another keyword. When used in a method it causes the object to terminate after the method terminates. The object will accept no more messages. If the object is an instance, it also causes a `destroy(self)` message to be sent to its class.

Procedural method rule The general form of a procedural method rule is:

Pattern [: *Guard*] -> *Method*

where *:Guard* is optional. (The -> is not optional.) If present, any message matching the pattern invokes *Method* only if *Guard* succeeds. The `field_course` instance definition has two method definitions for `add_an_enrollment/2`. One for the case that the current value of instance variable `NumStudents` is less than the allowed maximum number of students for this course, held as the value of instance variable `MaxStudents`. The other method is for the case where this maximum has been reached.

Default parallel evaluation The `Method` code is a Parlog conjunction. In this, the ',' separator indicates parallel execution. To force a sequential execution we can use '&' to conjoin the calls of the method. Finally, by default, as soon as `Method` starts to execute the instance 'recurses' to accept the next method invocation method. To stop this, we can use '&.' instead of '.' as the method terminator. This convention, derived from Polka, means that the next message will only be accepted when all the calls of the method has terminated. Note that this does not necessarily mean when all the activities set off by the method have terminated. Suppose that the method causes activity to be started in another object via the sending of a message to that object. The '&.' will not cause the object to suspend message processing until this activity has terminated. It only waits for the message send and the other calls of the method to terminate. The message send will terminate almost immediately. What we can do, and this is often the only delay that we need, is make the method suspend (hence not terminate) until some variable in the sent message has been bound by the activity that the message initiates. We do this by using the Parlog `data/1` primitive. We give an example below.

3 Dynamic knowledge versus state variables

To illustrate the trade of between using state variables and using dynamic knowledge consider Program 2. It uses instance variables. After briefly discussing certain aspects of the program we will replace two of these with dynamic knowledge (actually Prolog facts) attached to instances.

Program 2 is a two class program with the `undergraduate` class inheriting from the `student class`. Note also that the student class header specifies a machine location, `laotzu` for the class. This is the name of the workstation in our department at Imperial College where the `student` class process will reside. By default, its subclass, and all the instances of both classes, will also reside on that machine. However, we could locate the subclass, and hence all its instances, on another machine.

The program assumes that there is a `department` class with instances that are college departments, and that the class has an procedural instance method, `enroll_for_a_course(CourseId,Ans)`. The method will try to enroll the sender of the message, some instance of the `undergraduate` class, in course `CourseId`, an object identifier, such as `doc105`. Remember that the sender identifier does not need to be included in the message. DK_Parlog^{++} automatically adds the sender identifier to every message and this can be accessed in the receiving method using the `sender` keyword.

The `enroll_for_a_course` message might have been sent to the `undergraduate` instance by an enrollments server that interfaces with real students. They would perhaps use dialogues displayed by the server to enroll in courses, find their grades etc.

```
class student at laotzu with
{
 instance_definition
   states
     Name, Dept.
   methods
     department(DEPT) -> DEPT = Dept.
     name(N) -> N=Name.
}.
%----------------------------------------------------------------------
class undergraduate isa student with
{
 instance_definition
   states
     Year := 1, Subject, CurrentCourses := [], PreviousCourses := [].
   methods
     enroll_for_a_course(CourseId, Ans) ->
         enroll_for_a_course(CourseId, Reply) => Dept,
         (Reply == yes ->
            Ans = yes, CurrentCourses := [CourseNo | CurrentCourses] ;
            Ans = no
         ).
     final_grade(CourseId, Grade) ->
         remove(CourseId,CurrentCourses,NCurrentCourses),
         CurrentCourses := NCurrentCourses,
         CoursesTaken := [(CourseId,Grade)|CourseTaken].
     year(Y) -> Y = Year.
     next_year -> Year := Year +1.
     :       %other methods
}.
```

Program 2. Student active classes

An undergraduate instance is created with a message send:

```
      create(StId,2,ai,_,_,Bill Smith','Computing') => undergraduate
```

`StId` will be bound to a unique name for the new instance.

Let us now consider a modification of this program in which we use dynamic knowledge in instances of the `undergraduate` class in lieu of the instance variables, `Year` and `CurrentCourses`. In this case, there is not much advantage. We do it to exemplify the knowledge manipulation facilities.

Let us suppose that these two state components of an undergraduate student instance will be recorded by dynamic knowledge methods, actually Prolog facts. The year will be recorded by a fact of the form, yr(N) and the CurrentCourses by a sequence of facts `course(Id1).` `course(Id2).` ... `course(Idk).`

The create message now takes the form:

```
create(StId,'Bill Smith',computing,_,'Computing') with {yr(1)} => undergraduate.
```

where {yr(1)} is the dynamic knowledge/data attached to the new instance in place of the value for the instance variable `Year`.

The new class definition for undergraduate is given in Program 3.

```
class undergraduate isa student with
{
 instance_definition
   states
     Name:=unknown,Subject := unknown, PreviousCourses := [].
   methods
     enroll_for_a_course(CourseId, Ans) ->
         enroll_for_a_course(CourseId, Reply) => Dept,
         (Reply == yes) ->
            acquire_knowledge(course(CourseId)) :> self ;
            Ans = no
         ).
     final_grade(CourseId, Grade) ->
         remove_knowledge(course(CourseId)) :> self,
         CoursesTaken := [(CourseId,Grade)|CourseTaken].
     year(Y) -> (yr(Z) :> self -> Y=Z ; Y=1).
     next_year -> year(Y) :> self & NY is Y +1 &
         (add_knowledge(yr(NY)), remove_knowledge(yr(Y))) :> self.
     .
     .     %other procedural methods
     .
     .     %other knowledge methods
     .
}.
```

Program. 3 Undergraduate class with knowledge methods

The definition of the knowledge method for `year/1` is worth examining.

```
year(Y) -> (yr(Z) :> self -> Y=Z ; Y=1).
```

It first checks to see if a yr/1 fact has been added to the dynamic knowledge of the instance by the knowledge query:

```
yr(Z) :> self
```

If this succeeds, the year has been explicitly recorded and its value is returned. If this fails, it returns the default value 1. So, just has when we used a state variable to record the year, if we create a student instance without specifying the year, and then we query it for its year, we will get the answer 1. We can be a little more sophisticated than this. Instead of just returning the default value 1, we could try to 'infer' the year by looking at the courses that the student is currently taking. We could have a definition:

```
year(Y) -> (yr(Z) :> self -> Y=Z ; findyear(Y) :> self.
```

where findyear/1 has a private Prolog definition in the code section for the class. A possible definition is:

```
findyear(Oid,Y) :- findall(C-L, (course(C):>Oid, level(L)=>C), Cs),
                   most_frequent_level(Cs,Y).
```

We assume that `most_frequent_level(Cs,Y)` would bind `Y` to the most frequently occurring level number appearing on the list of C-L returned in Cs. This default computation uses the university 'rule' that a student must take the majority of their courses with a level that is their year of study.

Let us also examine the method:

```
    next_year -> year(Y) :> self & NY is Y +1 &
                  (add_knowledge(yr(NY)),remove_knowledge(yr(Y))) :> self.
```

acquire_knowledge/1 and remove_knowledge/1 are the dynamic knowledge manipulation methods automatically added as instance knowledge methods to every class definition by the DK_Parlog^{++} system. [4]

The procedural method rule uses the Parlog sequential connective '&' to sequentialize the execution of its calls. First it uses a self enquiry to find the students current year in order to add 1. Then, it sends the knowledge modification sequence

```
                  (add_knowledge(yr(NY)),remove_knowledge(yr(Y)))
```

to itself. The two updates are put into a communication sequence in order to guarantee that they will be executed atomically by the object.

4 User defined create and class level knowledge

As an example of a class with a create that overrides the system default, and which has class level knowledge rules, consider Program 3.

```
class research_student is_a student at confucius with
{
 class_definition
   methods
    create(Id, Interests, Supervisor, Name, Dept) : nonvar(Supervisor) ->
        fork_instance(research_student,Id,(Interests, Supervisor, Name, Dept)).
    create(Id, Interests, Supervisor, Name, Dept) : var(Supervisor) ->
        find_a_supervisor(Interests, Dept, Supervisor) :> self,
        data(Supervisor) &
        fork_instance(research_student,Id,(Interests, Supervisor, Name, Dept)).
    %---------------------------------------------------------------------
    find_a_supervisor(Interests, Dept, Name) :-
        interests(Name, Interests) :> Dept,
        will_accept_student(Student,Interests,Ans) => Name, Ans=yes.
    find_a_supervisor(Interests, Dept, Name) :-
        head_of_the_department(Name) => Dept,
 instance_definition
   states
     Interests, Supervisor
   methods
      :
}.
```

Program 3. Knowledge methods at the class level

There are two procedural methods for create messages. The first is for the case where the identity of the supervisor is given in the create message. This essentially does what the system provided create does. The user method calls a system primitive, fork_instance/3. This will bind Id if it is was not supplied. It also informs the class research_student that Id is a new member using one of the system methods inserted in every class.

An attempt to create an instance of the *research_student* class without giving the supervisor will cause a use of a class level knowledge method for finding a supervisor. The class knowledge method will be invoked with the call:

[4] We can also view them as automatically inherited from a special *system* class.

```
        find_a_supervisor(Interests, Dept, Supervisor) :> self
```

The create method then waits until an answer binding for Supervisor is returned (the Parlog primitive data/1 will suspend until its argument is a non_variable). When the answer is returned the instance will be created. Let us look at the execution of the *find_a_supervisor* call. Notice that the class research_student is declared as residing on machine confucius. Suppose that the department class, which we have not defined, resides on another machine with all its instances. The invocation of the knowledge method for interests/3, assumed to be defined in the department class, with the call

```
        interests(Name, Interests) :> Dept
```

is a request to find a faculty member of instance Dept with some overlap of interests with the new research student. Generally this will have several solutions. As each solution to the call is returned, the enquiry and test

```
        will_accept_student(Student,Interests,Ans) => Name,
        Ans=yes
```

will check that the identified potential supervisor will accept the student. If they will not, the next solution to the call

```
        interests(Name, Interests) :> Dept
```

will be found. This involves distributed backtracking. A query thread for the suspended Prolog call interests(Name, Interests), on the department class machine, will be suspended waiting for a request for the next solution. The second clause for find_a_supervisor/3 is used to give a default supervisor if the first rule fails to find one. Because find_a_supervisor/3 is invoked from a procedural rule, as soon a solution to the call is found, its execution will terminate. This will also terminate the suspended thread for interests(Name, Interests) on the remote machine, should a supervisor who will accept the student be found before all solutions to the call interests(Name, Interests) have been generated. It is part of the housekeeping of the distributed backtracking implementation.

5 Using DOOD classes and a query server

We will now further change our example so that research_student class has no procedural instances methods or instance state variables. It no longer inherits from the student class because that had instance variable holding the name and department of the student. For a research student this information will now be held as a dynamic fact. All state information about an instance of the research_student class is now recorded as dynamic knowledge attached to the created instances. The class is what we have referred to in the introduction as a DOOD class. In addition, we add another DOOD class, supervisor. The dynamic knowledge that will be held about the instances of the two classes will conform to the schemas:

```
student:{                          supervisor:{
        id(ObjectId).                      id(ObjectId).
        name(Name).                        name((Name).
        department(Dept).                  interest(Interest1).
        interest(Interest).                interest(Interest2).
        supervisor(SUP).                   ........
        }.                                 student(Student1).
                                           student(Student2).
                                           ........ }.
```

However, not every instance need have facts for each predicate. The new class definitions are given in program 4. The user defined create methods are used to create instances according to the schema. In contrast to the *fork_instance* primitive used in creating an active instance, which forks a Parlog process, the *make_passive_instance* adds an L&O object definition [McCabe 1992] containing the clauses given in the third argument. It will generate a unique identifier for the object if need be, just as the *fork_instance* primitive.

Strictly, we do not need to give create definitions. With them, we create a research_student instance with a call:

```
create(Id,'bill smith',computing,distributed_logic_ programming) => research_student
```

Without it we can do this using the default create and the message:

```
create(Id) with {name('bill smith',department(computing),
                 interest(distributed_logic_programming)} => research_student
```

Our create allows a shorthand for the latter.

```
class research_student with
{
 class_definition
   methods
     create(OID, Name, Dept, Interest) ->
         make_passive_instance(research_student,OID,
                               {name(Name),department(Dept),interest(Interest)}).
}
%-------------------------------------------------------------------------------
class supervisor with
{
 class_definition
   methods
     create(OID, Name, Interest) ->
         make_passive_instance(supervisor,OID,{name(Name),interest(Interest)} ).
}
%-------------------------------------------------------------------------------
server db_manager with
{
 methods
   add_knowledge(ObjectID, Knowledge) ->
       acquire_knowledge(Knowledge) :> ObjectID.
   query(QueryExpression) :-
       query_converter(QueryExpression, ActualQuery),
       QueryExpression.
 code
   query_converter(Class-Query, (on(X, MemberList), Query) :> X) :-
     members(MemberList) => Class.
   query_converter( (Class-Query,RestQs),((on(X,MemberList), Query):>X),Erest) :-
     members(MemberList) => Class,
     query_converter( RestQs, Erest ).
}.
```

Program 4. A DOOD classes and query server

Whether or not we have a collection of DOOD classes or active classes, it is useful to provide a data base manager interface. This can handle queries that refer to more than one class, and it can ensure that a sequence of updates to several classes are performed atomically. Program 4 also contains a definition of a server, called db_manager, which acts as a simple manager for queries and updates to our two DOOD classes, and to any other DOOD or active knowledge class. It just assumes that the queries are to be processed by knowledge methods - Prolog rules and facts - that belong to, or are inherited by, instances of the classes referenced by the query.

The query format The query method in the **db_manager** assumes the following format for the query expression is used:

```
(Class1-Query1, Class2-Query2,...)
```

where **Classi** specifier confines the **Queryi** expression to the instances of **Classi**. Each `Class-Query` component is actually converted into the DK_Parlog^{++} query:

```
( on(X, MemberList), Query :> X )
```

where MemberList is the current membership list of `Class` acquired by sending a

```
members(MemberList) => Class
```

message. Note that the `db_manager` server definition has a code section containing the definition for `query_converter`. This is private to the server and cannot be called, via a message, from outside the server.

The following `forall` loop embeds a query to `db_manager` will find each research_student who does not yet have a supervisor for whom a supervisor can be found who shares the recorded interest of the student. In addition, for each found pair, it uses an the `add_knowledge` method of `db_manager` to atomically update the knowledge attached to the student and the supervisor, assuming that all queries and updates go via the `db_manager`.

```
forall( query( student-(interest(I),id(ObjectID),name(Name)),(not supervisor(_)),
              supervisor-(interest(I),name(Sname),id(SObjectID),(not student(_)))
             ) :> db_manager,
       (add_knowledge(ObjectID, supervisor(Sname),add_knowledge(SObjectID, student(Name)
       ) :> db_manager
      ).
```

6 Extension to an Heterogeneous System

The link with other systems is best effected using the TCP/IP communication facilities mentioned in the introduction and interface servers.

6.1 Interface Servers

The main functionality of the interface server is to translate messages to the proper inputs of the external systems and to maintain the outputs generated. For example, the interface serverto an external object oriented database may include methods for translating messages to the queries of the database system. The translation could range from simple literal rearrangement to complex query synthesis. For instance, a simple interface to the **ORION**[Kim et al. 1990] database system would comprise methods which translate messages into one of the **select**, **select-any**, **delete**, **delete-object**, and **change** associative access messages or to the methods which manipulate the returned *set object* containing selected instances. The interface to the **Jasmine** object-oriented DBMS[Ishikawa 1993], on the other hand, is more complicated. Proper messages must be provided and translated into its set-oriented access query of the following form:

<target part> where <condition part>

Take an example:

```
[DOCTOR.Name, DOCTOR.Address] where DOCTOR.Dept="pediatrics" and DOCTOR.Age > 45
```

An SQL-like language synthesizer might be required for the methods to translate a message like:

```
all( 'DOCTOR'.(Name, Address), (Dept = pediatrics & Age > 45))
```

to the above Query.

7 Concluding remarks

DK_Parlog^{++} is an experimental system that is still evolving. Most of what we have described is implemented or is very nearly implemented. We intend to explore several different kinds of applications, including ones that require an interface to other, heterogeneous, applications.

There are two other heterogeneous applications, both of which accept and send Prolog style terms using TCP/IP, that we definitely intend to link with. One is a Motif based dialogue server, called Dialox, the other is a small OO data base system implemented in C. Both of these have been designed and implemented by our colleague, Frank McCabe, and his students at Imperial. The interface will be via a server process.

The instances of active classes of the language, especially if all their instance methods are defined by purely parallel Parlog code, are very like actors [Agha 1986]. The addition of our inheritance mechanism gives extra programmability to the actor model, similar to that of [Kafura and Lee 1989].

At the other extreme, a program comprising only DOOD classes, is quite like a DLP [Eliens 1992] program. This also allows concurrent execution of methods in an object and has distributed backtracking. However, as we mentioned in the introduction, DLP has a weaker constraint regarding when dynamic information can be changed. Also it lacks our Parlog procedural methods, which allow parallelism within the method. DLP only has parallelism of alternative methods.

Orient84/K [Tokoro and Ishikawa 1984] is perhaps the closest in design goals to DK_Parlog^{++}. This language has procedural methods implemented in a distributed version of SmallTalk with knowledge methods implemented in Prolog. But the SmallTalk procedural methods are sequential, and the fit between SmallTalk and Prolog is much less homogeneous than that between Parlog and Prolog. For example, messages sent between SmallTalk objects cannot be Prolog queries containing variables. In DK_Parlog^{++} such messages can be sent, like those that we can send to the `db_manager` server, and evaluated by Prolog. The answer bindings are then communicated back to the sender object via the network wide distributed unification scheme.

We believe that DK_Parlog^{++} is a very expressive language ideally suited to prototyping distributed knowledge base, or distributed AI applications. When its implementation is completed, we will be happy to make it available to other researchers, to test its potential.

References

[Agha 1986] Agha, G., *Actors*. MIT Press, 1986.

[Agha and Hewitt 1988] Agha, G., Hewitt, C., "Concurrent programming using actors". In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, Page 37-53. MIT press, 1988.

[Chu and Clark 1993] Chu, D., Clark, K. L., "IC-PrologII: a Multi-threaded Prolog System", In *Proceedings of the ICLP'93 Post Conference Workshop on Concurrent, Distributed & Parallel Implementations of Logic Programming Systems*, Budapest, 1993.

[Clark and Gregory 1986] Clark, K. L., Gregory, S., " PARLOG : Parallel Programming in Logic", In *ACM transactions on Programming Languages and Systems* , Vol 8, No. 1, page 1-49, 1986.

[Clark 1990] Clark, K. L.,"Parallel Logic Programming", The Computer Journal, 33, No.6, 482-493, 1990.

[Davison 1989] Davison, A., "Polka:A Parlog Object Oriented Language", PhD thesis, Imperial College, 1989.

[Davison 1990b] Davison, A., "Parlog++ : A Parlog Object Oriented Language", Technical Report, DOC, Imperial College, March, 1990b.

[Eliens 1992] Eliens, A.,*DLP A Language For Distributed Logic programming-Design, Semantics and Implementation*, John Wiley & Sons, England, 1992

[Ishikawa 1993] Ishikawa, H. *Object-Oriented Database System : Design and Implementation for Advanced Applications*, Springer-Verlag, 1993.

[Kahn et al. 1987] Kahn, Kenneth M., Tribble, Eric D., Miller, Mark S., and Bobrow, Daniel G. "Vulcan: Logical concurrent objects", In P. Shriver and P. Wegner, editor, *Research Directions in Object-Oriented Programming*, MIT Press, 1987.

[Kafura and Lee 1989] Kafura, Dennis G., Lee, Keung H., "Inheritance in Actor Based Concurrent Object-Oriented Languages", In *Proceeding of ECOOP'89*, page 131-145. Cambridge University Press, 1989.

[Kim et al. 1990] Kim, W., Ballou, N., Banerjee, J., Chou, H. T., Garza, J. F., and Woelk, D., "Integrating an Object-Oriented Programming System with a Database System", In *Research Foundations in Object-Oriented and Semantic Database Systems*, edited by Alfonso F. Cardenas and Dennis Mcleod, Prentice Hall, 1990.

[McCabe 1992] McCabe, F. G., *L&O: Logic and Objects.* International series in Computer Science. Prentice-Hall International, 1992.

[Shapiro and Takeuchi 1983] Shapiro, E. Y. and Takeuchi, A., "Object Oriented Programming in Concurrent Prolog", In *New Generation Computing* 1(1), page 25-48, 1983.

[Tokoro and Ishikawa 1984] Tokoro, M., Ishikawa, Y., *Orient84/K: A Language with Multiple Paradigms in the Object Framework*, Proceeding of the International Conference on Fifth Generation Computer System, ICOT, Novemer 1984.

[Wang and Clark] Wang, T. I, Clark, K. L., "The Implementation of Distributed Object-oriented Logic Programming Language DK_Parlog^{++} ", Internal Report, Department Of Computing, Imperial College, in preparation.

[Yoshida and Chikayama 1988] Yoshida, K., and Chikayama, T., "A'UM-a stream-based concurrent object-oriented language", In *Proceeding of FGCS'88*, ICOT, Tokyo, 1988.