24

Distributed Object Oriented Logic Programming as a tool for Enterprise Modelling

K. L. Clark, N. Skarmeas, T. I. Wang
Imperial College
Department of Computing
London
{klc,ns4,tiw}@doc.ic.ac.uk

Abstract

In this paper we propose a distributed object oriented logic programming language, called DK_Parlog^{++}, that we have developed at Imperial College, as a powerful tool for enterprise modelling and for prototyping an enterprise integration framework. We have used the language to develop a model of our own department and to prototype a generic enterprise integration framework based on role theory.

We adopt an agent based approach, the agents being essentially distributed active objects containing both procedural methods and public knowledge, the knowledge being expressed as Prolog facts and rules. Each agent is an instance of some agent class. The agents interact with each other by sending messages to request services, and by remote queries to access each others public knowledge. The agents can either be used as a coordination layer for an enterprise integration prototype or to provide an active model of the activities of the enterprise. Each agent inherits all the methods and knowledge of its class and super classes. It has a 'state' partly encoded in its state variables and partly in a dynamic component of its public knowledge. The methods of the agents will be the activity rules, the procedures of the enterprise. The public knowledge of each agent can be used to express the constraints and preconditions of these procedures, or to describe the effects of the procedures. The knowledge, distributed over the the agents and their classes, comprise a distributed OO knowledge base for the activities of the organisation.

This paper assumes acquaintance with the concepts of concurrent object oriented programming and logic programming, particularly Prolog.

Keywords

Distributed Object Oriented Logic Programming, Enterprise Modelling, Multi Agent Systems, Organization Modelling

P. Bernus et al. (eds.), *Modelling and Methodologies for Enterprise Integration*
© Springer Science+Business Media Dordrecht 1996

1 INTRODUCTION

1.1 The programming language DK_Parlog^{++}

A program in this language (Clark and Wang, 1994) comprises a set of classes, and their instances, linked via single inheritance and distributed over a local area network. Different classes can be on different machines and the class and all its instances have separate computation threads, i.e. they process messages independently and concurrently. They are *active* objects.

All objects (i.e. both the classes and their instances) have a unique identifier. In the case of a class, this is the class name such as **student** or **department** given in the program. In the case of an instance, it is either a system generated or a programmer assigned symbol. All instance objects are generated by the *create* method of their class. Classes maintain a list of the identities of all their current instances. This can be used for multicasting a message to all the instances of the class. Methods are concurrent actor style computations (Agha and Hewitt, 1987) invoked by messages, which are just Prolog terms sent to the object identified by its unique identifier. Replies to messages can either be explicitly sent, as in actor programs, by a message send action in the invoked method, or they can be implicitly returned by a unification action in the invoked method that binds a variable in the received message term. This is the powerful reply method *via answer variables in messages* of concurrent logic programming (Shapiro and Takeuchi, 1983).

Methods can also access and update state variables of the object. (The *create* method of a class updates a class state variable maintaining the list of instances, for example.) A method execution is forked as a separate thread, so as soon as its execution starts the object is ready to accept the next message. The processing of a sequence of received messages can therefore proceed in parallel. The implementation ensures that if a method invoked by a later message wants to access a state component that is updated by an earlier method, which is still executing, then the execution of later method will automatically suspend until the new value for that state component is computed. Similarly, if the computation of a later method thread updates a state component that is accessed but not updated by an earlier method invocation, the earlier method will always see only the old value. This integrity of state is a consequence of the fact that methods are compiled into clauses of concurrent logic programs. They are compiled into Parlog programs (Clark and Gregory, 1986) using a modification of the method of representing objects as processes of Shapiro and Takeuchi, 1983 and Davison, 1988.

If objects only had methods, DK_ Parlog^{++} would just be an actor style language with inheritance and an extra reply mechanism using answer variables in messages. What makes the language much more powerful, and much more suited to the task of enterprise modelling, is that in addition to methods, every object can have a publicly accessible knowledge component comprising a set of Prolog clauses. The public knowledge of any object O can be queried from *any other object* using a query of the form `O?Q`. Like a method invocation, each query evaluation also becomes a separate thread of computation, so an object can be servicing several queries at the same time. The knowledge used to answer the query `O?Q` is the knowledge of the object O together with all the knowledge that O inherits from its super classes. Thus the knowledge of the objects is, in effect, a

distributed OO deductive data base. The Prolog component is built on top of Imperial's multi-threaded Prolog system (Cosmadopoulos, 1993).

In addition to the knowledge an object derives from its class definition, which will be common to all instances of the class, an object can acquire *new* knowledge during its lifetime. It does this by asserting and retracting clauses for a special class of *dynamic* predicates. This allows the knowledge of an object to change, in the way that the values held in the state variables of the object can change. It allows objects to learn.

1.2 Integrating the activities of an enterprise using agents/active objects

Today's enterprise environments consist of a collection of workers, physically distributed, performing a variety of tasks. In addition, there is usually a networked collection of machines performing tasks complementary to these workers, and holding information that they must access and manipulate.

The problem of increased performance in those environments is not due to limited labor or capital but due to limited access to information and difficulty in achieving coordination (Pan and Tenenbaum, 1991). Information is scattered throughout the organization and workers have no direct and easy way of accessing it. Furthermore, decisions and activities are highly interdependent, and the coordination between them is a major problem in today's enterprises.

Information systems could be used to improve the performance of organizations in these matters. They can make communication and access to information much easier. They can also help to integrate the several processing components (machines, software, people) of the organisation into a unified system, which should improve coordination between those components.

One way of achieving both increased coordination and integration over on existing organisation structure is too super-impose a coordination layer comprising a network of agents. In such an agent network, some agents will autonomously perform activities assigned to them, others will be attached to workers and assist them, others will control machines, others will gather and maintain information, or control access to the enterprise data bases, and so on.

A problem that might arise with this, is that huge Knowledge Bases have to be built (Pan and Tenenbaum, 1991). These are difficult to build and even more difficult to maintain them. Far better, if each agent has access to its own local knowledge, which can inherit from more general knowledge about the organisation, and if each agent can, when needed, explicitly query publicly accessible knowledge of another agent whose identity it knows. In this approach, we distribute the knowledge around the organisation, whilst maintaining global access.

A language such as DK_ Parlog^{++} allows us to implement the agents as active objects, distributed over some computer network. It also allows us to break up and distribute the organisational knowledge as public knowledge held in these active objects.

We have investigated two approaches to enterprise modelling using this language. One, which we will introduce in some detail in this paper, is closer to a simulation. For each entity in the organisation we will have an active object with the Prolog rules of the knowledge components used, primarily, to encode more powerful methods. For example,

we use Prolog rules to represent methods that search for other agents with whom to communicate, or to collect information from several other agents. In this approach, the agents have little self knowledge and very little *explicit* knowledge about organisational structure. That an agent must report to some other agent will only be *implicit* in the fact that it has a method that sends messages to that other agent.

In the second approach, which will briefly describe at the end of the paper, we have an explicit representation of the organisation structure, the roles of the organisation and the contractual and reporting relationships between the roles. Agents do not have preassigned roles. Instead, when the agent is created, it is given the roles and methods it must perform Thus, we do not have a class for each type of agent, with the methods for that agent class 'wired in'. We instead have a generic agent class, instances of which are specialised to particular roles as they are created. This gives us a much more high level, and more easily modifiable, model of an organisation. It is also an active model, but it will execute more slowly that the more direct 'simulation' model of the first approach.

2 AGENT BASED SIMULATION

Figure 1 illustrates this first approach. We have several different classes of agents specialised to certain tasks and certain patterns of communication. We summarise the activities of some of these classes.

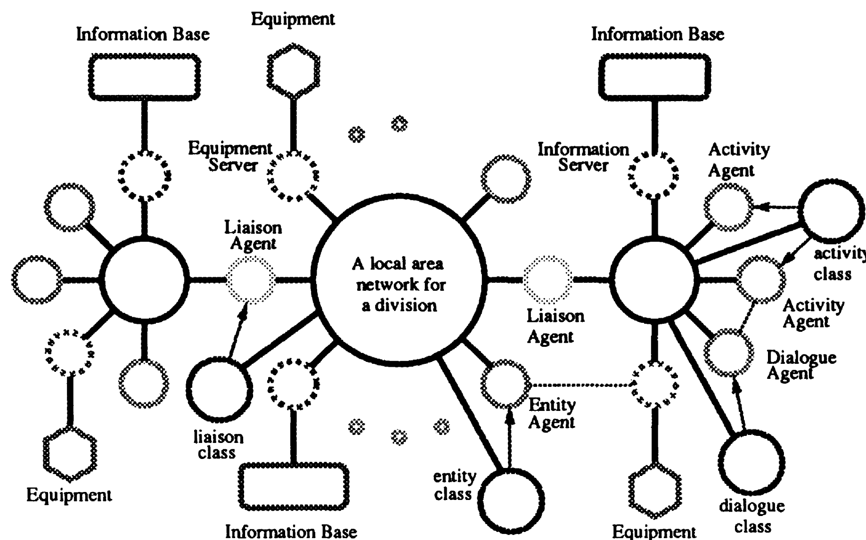

Figure 1 A framework for the enterprise integration.

Activity Agents As the name suggests, an Activity Class(AC) of agents will be used to encapsulate a group of very similar activities which are either human behaviors or predefined service procedures. Note that these activities are not necessarily identical, they are just very similar. There might be several secretaries in a division working for different managers. They might have some identical responsibilities, such as arranging appointments for the manager, and some individual behaviors, such as making different recommendations when asked to suggest a birthday gift for their manager's wife, They

will, of course, have different salaries. In the proposed framework, agents with slight differences in their behaviors will be grouped into a same activity class. So we would have a secretary agent class which would partially automate the routine activities of secretaries of the division. Common procedures and knowledge for each agent are defined in the class or some super class. Typical examples are the minimum salary of a new secretary and the rules for deciding the annual bonus of secretaries for the division. These are the basic guidelines for the behaviors of all the agents created from this class. An Activity Agent(AA) is created when the AC receives a *create* request sent out from another agent, say, in the personnel division. When it is being created, unique characteristics can be given to it. For example, a new secretary agent can be assigned an initial salary overriding the default salary. It can also be given extra knowledge and problem solving capabilities, via Prolog clauses for its dynamic predicates. These are added to, and hence modify, the public knowledge it shares with all the other secretary agents of the class.

Dialogue Agents When an AA is a surrogate for part of a persons activities, it will sometimes need to interact with that person, for example, for displaying received information or for input on crucial decisions. A special Dialogue Class(DC) is set up for such interactions. When an interaction is needed by an AA or a human, a Dialogue Agent(DA) will be created and, after security checks, linked to the associated AA. The dialogue agent can be destroyed when it is no longer needed. A dialogue class can also allow a person to modify the responsibilities entrusted to their associated AA. This modification might be achieved by making changes to the AA's dynamic knowledge, or by the more radical instruction, sent via the DA, that the AA should metamorphise into an AA of another activity class. (DK_ Parlog^{++} has a special action which allows an instance object to transforms itself into an instance of another class, whilst retaining its unique object identity.)

Entity Agents A very common interaction between activities is the exchange of information, including both documents and messages. To serve such sharing of data, entity agents can be used. The internal state of an entity agent will include the location of physical storage of the data, if it does not directly hold the data. The entity agent will also handle all requests to access and update the data, or to move its physical location. When an activity agent has finished with the data and wants to pass it on, or it wants to allow concurrent working with the data, it just passes the identity of the entity agent to some other agent. Thus, an entity agent does not actually migrate between machines in a local area network, only its identity is passed around and its internal state is perhaps modified to register the new owners. The progress of an entity agent through a sequence of activity agents is a workflow.

Information Agents The main functionality of an information agent is to translate messages into proper inputs for an associated external information system and to customize the generated outputs for the client agents. For example, the information agent interfacing to an external object oriented database will contain methods for translating messages into appropriate queries of the database system. An information agent is similar to an entity agent. The difference is that entity agents are more light weight, are generally temporary, and manage a smaller quantity of data. Information agents manage access to large amounts of data, such as enterprise data bases, and are generally permanent.

3 AN EXAMPLE AGENT BASED ENTERPRISE SIMULATION

The following example is part of a system which models some of the activities of an Imperial College department. We illustrate the part dealing with the creating and managing of a new course, and the subsequent inquiries about the course, and possible enrollment requests, from student agents.

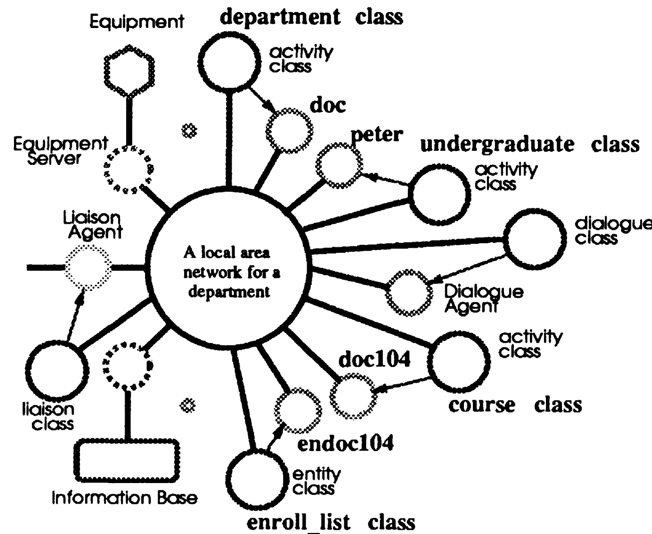

Figure 2 The interaction between objects.

3.1 Components of The Example System

Four agent classes are needed for the example system, three of them are **Activity classes**, the remaining one is an **Entity class**. These classes are all shown in Figure 2 along with other possible classes and agents in the same network.

The **department** class is for creating agents that represent a college department and its administrative procedures. Only the methods and knowledge needed for managing a course and checking the enrollment requests from students are shown. Student representative agents are created by the **undergraduate** class, which inherits from a **student** class. Major activities of an undergraduate agent include enrolling in a course, keeping a record of courses which have been taken, and recording the grades achieved. A student agent will interface with the student it represents via a **dialogue** agent. Course agents come from the **course** class, this handles the activities of announcing a new course to all undergraduate students, administering enrollment requests from them, and keeping the record of those who successfully enrolled. The enrollment list is actually maintained by an entity agent, an instance of the enroll_list class. This entity agent is shared by the course agent and the lecturer agent for the lecturer for the course.

The Department Activity Class Part of the instance definition of the **department** class is shown in program 31. It includes just two method and two knowledge rules, representing two procedures and two rules common to all departments. Two predicates

are also declared as dynamic. Facts and rules about these dynamic predicates are used to record the department specific information, which can change. They record which courses are compulsory for students of specific years, and the information that a course is a prerequisite for another course. The method *new_prereqs_for(CourseId,ListofCourseIds)* actually updates the dynamic knowledge encoded in the facts about the dynamic predicate *prerequisite/2*. This method can also be used for entering the prerequisite knowledge for an entirely new course. There is an analogous method (not shown) for adding the knowledge that a new course is compulsory.

```
class department with
{
 class_definition
    :
    :
 instance_definition
   methods
     states Name,Head......

     will_run_a_course(CourseId, Title, EnrollList) with Descrip ->
        create(CourseId,Title,self,sender,EnrollList) with Descrip => course,
        self? assert(course(CourseId)).
     new_prereqs_for(CourseId, ListOfCourseIds) ->
        self ? retractall(prerequisite(CourseId,_),
        forall( on(CId,ListOfCourseIds),
                self ? assert(prerequisite(CourseId,CId))).
     .
     .

  knowledge
     dynamic course/1, compulsory_course/2, prerequisite/2.

     satisfies_prerequisites(Student, CourseId) :-
        forall(self ? prerequisite(CourseId,Pre), Student ? passed(CourseId)).
}.
```

Program 31 A fragment of the department class

The other displayed method, *will_run_a_course(CourseId,Title,EnrollList) with Descrip*, is for requesting that a new **course** agent be created for managing a new course run by a lecturer. The agent which sends the message will usually supply a public name, `CourseId`, which identifies the course, together with a `Title` and a `Descrip` of the course. The `Descrip` component comprises a set of Prolog clauses for the dynamic predicates, `topic/1` and `level/1` of a course instance. These give details of the new course. `Enrolllist` will generally be an unbound variable to be bound to the identity of the entity agent which will maintain the enrollment list for the new course. On receipt of the *will_run_a_course/3* message the department object sends a *create(CoursId,Title,self,sender,EnrollList)* message to the **course** class. (=> is the message send operator.) This will create the new course instance object with identity `CourseId`, and, indirectly, will create the entity agent for the enrollment list of the course binding the variable `Enrolllist` to the system generated identity of this entity agent. The agent which sent the original *will_run_a_course* message will have access to the value of this variable as soon as it is bound. (This is the reply mechanism via answer variables that we mentioned in the introduction). Suppose

this is the personal agent of the lecturer for the course. That lecturer, via this agent, can now directly interact with the enrollment list entity agent. The enrollment agent might let the lecturer peruse the list, set an enrollment limit, etc.

Notice that the *create* message sent to the **course** class has two special arguments, *self* and *sender*. The *self* argument identifies the department agent which is sending the create, and the *sender* argument identifies the agent which sent the *will_run_a_course* message to this department agent. The keyword *sender* used in any method always denotes the identity of the sender of the message that invoked the method.

Notice that the *will_run_a_course* also asserts a new `course/1` fact into its knowledge component recording the new course's identity.

Let us now look at the knowledge rule which determines when a student satisfies the prerequisites for a course. It says that all the prerequisites for the course (if there are any) must be passed by the student. This is tested by the Prolog `forall`, which for every prerequisite of the course in question (found by the `self` enquiry), checks by a query `Student?passed(CourseID)` to the public knowledge of the student, that they have passed the course. This latter is a remote call to a student agents's public knowledge from a rule being executed in a query thread (for a `satisfies_prerequisites/2` query) executing within some instance of the `course` class.

Spawning a department instance To spawn an agent instance of the **department** class, with the public identity `doc`, and with a suitable description comprising a set of *course/1*, *prerequisite/2* and *compulsory_course/2* facts, the following *create* message can be used:

```
create(doc,computing,maibaum) with
  {
   course(doc102).
   course(doc104).
   .
   .
   prerequisite(doc104,doc102).
   prerequisite(doc107,doc106).
   .
   .
   compulsory_course(doc104,2).
   compulsory_course(doc107,3).
   .
   .
  } => department.
```

The initial dynamic knowledge could also contain rules, for example, a rule specifying that all level 1 math topic courses are prerequisites for any level 2 theory topic course. (As we shall see, course agents maintain level and topic information about the courses they represent as facts in their knowledge component.) This would be an enterprise rule specific to the computing department. It would be expressed as:

```
prerequisite(CId1,CId2) :-
    self?(course(CId1),course(CId2)),
          CId1?(level(1),topic(math)),Cid2?(level(2),topic(theory)).
```

This rule would simply be given as an extra clause about the predicate `prerequisite/2` in the dynamic knowledge component. Notice that it has two remote calls, to the knowledge

components of the two courses that are potentially in the prequisite relation according to this rule.

The Course Class In program 32, the **course** class is defined. This contains a definition of the class *create* method. If no definition of this method is given, a default *create* method can be used. We used the default method for the `course` class when creating the `doc` instance of the class.

```
class course with
{
 class_definition
   methods
     create(CourseId,Title,Department,Lecturer,EnrollListEnt) with Knowledge ->
        create(EnrollListEnt, Lecturer) => enroll_list,
        fork_instance(course,course_notification,(CourseId,Department,Title,
                        Topics,Lecturer, _, EnrollListEnt),Knowledge).

 instance_definition
   methods
     states Title,Department,Topics,Lecturer,NoOfStudents := 0,EnrollListEnt.

     course_notification ->
        announce(self) => undergraduate.

     title(T) -> T=Title.

     add_topic(Tp) -> self?assert(topic(Tp)).

     current_numbers(NUMBERS) ->
        NUMBERS = NoOfStudents.

     enroll(Student,Ans) ->
         (Department?satisfies_prerequisites(Student,CourseId),
          EnrolllistEnt?course_still_open)
          ->
           Ans=ok, NoOfStudents := NoOfStudents + 1,
           add_an_enrollment(Student) => EnrollListEnt
         ; Ans='Prerequisites are not satisfied!'.
  knowledge
    dynamic topic/1, level/1.
}.
```

Program 32 An active class of DK_Parlog^{++}

For this class, we need to define a special *create* method since an instance of another class will be created as a side effect. An instance of the **enroll_list** class is created and its identity is held in a state variable of the created course agent. The identity of this entity is also sent to the lecturer agent for the lecturer of the course, so that it can access the enrollment list. Since we are defining our own *create* method for the **course** class, we cannot simply send a *create* message to the class to actually fork the new instance. Instead we must use a primitive *fork_instance/4* procedure, which forks an instance of any class. This primitive also allows us to send an initial message to the newly created object (given as the second argument of the *fork_instance/4* call). In this case, we will send an initial *course_notification* message to the newly created course agent. The tuple of initial

values for all the state variables of the forked instance are given as the third argument of the *fork_instance* call.

An example course agent creation message is:

```
create(doc104,'Declarative Programming',doc,klc,EnListE) with
     {topic(prolog). topic(unification). topic(programming). level(1)}
```

This also gives some Prolog facts for the dynamic predicates `topic/1` and `level/1`. Since we are just using facts, we could have held the topic and level information as values of state variables. We could have had a state variable holding a list of the topic names, and another holding the level. We quite often have this sort of choice regarding the representation of 'state' information for an object.

Triggered by the initial `course_notification` message, a **course** agent will immediately send a message announcing the new course to the **undergraduate** class. As we shall see, the receipt of this message by the undergraduate class will cause the announcement to be forwarded to all the current undergraduate agents of the class. Note that all but the first method of a course agent makes use of a reply variable to respond to the message invoking the method (because each method has a = unification against one of the arguments of the received message, which will instantiate the argument if it is an unbound variable). The first two simply access the value of two of the state variables. The last one deals with an enrollment request for some student in the course.

This has a Prolog style conditional as its action (the '->' is the 'then', the ';' is the 'else'). If the `Student` satisfies all the prerequisites for the course, tested by querying the public knowledge of the `Department` agent, and the course is still open, tested by querying the entity agent, `EnrollListEnt`, managing the enrollment list, then the student is enrolled. An 'ok' answer to the enrollment request is sent back via the answer variable `Ans` of the *enroll* request. Two state components are also updated. One is the state variable `NoOfStudents`, updated by a simple assignment. The other is the enrollment list, updated by a message sent to the `EnrollListEnt` agent that manages the list. If the student does not satisfy the prerequisites (the 'else' branch), a suitable reply is sent via the answer variable and no enrollment is made. Notice that method invocations, unlike calls to the Prolog rules of the knowledge component of an object, never fail. So, even though the enrollment does not take effect, the enrollment request, as a method call, succeeds. It just has a different response sent back via the answer variable indicating that the hoped for action has not been taken.

The Undergraduate Activity Class Part of the declarations of an **undergraduate** activity class, which inherits from the **student** class, are shown in program 33. In the program, a class method *announce(CourseId)* is used to broadcast a message *course(CourseId)* to all the student agents that are current instances of the class. This list of current instances for any class, is given by the value of *members*. Like *self*, the value of *members* is dependent upon the context of its use. They both refer to the values of hidden state variables. `self` is a hidden state variable of every object, `members` is a hidden state variable of every class object.

```
class student with
{
 instance_definition
   methods
     states  Department, Name.

     department(DEPT) -> DEPT = Department.
     name(N) -> N = Name.
}.
%-----------------------------------------------------------------
class undergraduate isa student with
{
 class_definition
   methods
     announce(CourseId) ->
        new_course(CourseId) => members.

 instance_definition
   methods
     states  MyYear:=1.

     new_course(CourseId,Name) ->
       ( Department?compulsory(CourseId,MyYear); self?interested(CourseId) ) ->
         enroll(Name, Reply) => CourseId,
         (Reply == ok -> self?assert(taking(CourseId)) ).

     final_grade(CourseId,Grade) ->
        self?( retract(taking(CourseId)), assert(grade(CourseId,Grade)) ).
      .
      .    %other methods

   knowledge
     dynamic  taking/1, grade/2, interested/1.

     passed(CourseId) :-
        self?grade(CourseId, Gr),
        Gr>50.

      .
      .    %other knowledge rules and facts
      .    %that pertain to all students
      .

}.
```

Program 33 Undergraduate class with knowledge rules

On receiving a course announcement message, an undergraduate student agent makes a decision based on a disjunctive test

```
(Department?compulsory(CourseId,MyYear); self?interested(CourseId))
```

(here the ';' is Prolog's 'or'). The agent queries the department to which the student belongs, held in the state variable `Department` inherited from the **student** class, to see if the course is compulsory. If not, there is a self inquiry, which will access the student agents dynamic knowledge, to find out whether the student would be interested in the course. An example creation of a student agent, with suitable dynamic knowledge, is:

```
create(bj,doc,'Bill Jones',_) with
   { interested(CId) :- CId?topic(logic).
     interested(CId) :- CId?topic(algorithms). }
```

3.2 A sample run of the example

To provide a much clearer picture, a snapshot of the interactions between all the agents and classes of a sample run of the example system is shown in figure 3. The snapshot captures the interactions of the activities involved after a message

```
will_run_course(doc104,'Declarative Programming',EnListE) with
    {topic(prolog).topic(unification). topic(programming). level(1)}
```

is sent to the **department** agent, **doc**, in the presence of the undergraduate student agent **bj**. Interactions are represented by dotted lines in the figure. The sequence of the interactions are numbered, with interactions which may happen concurrently having the same sequence number.

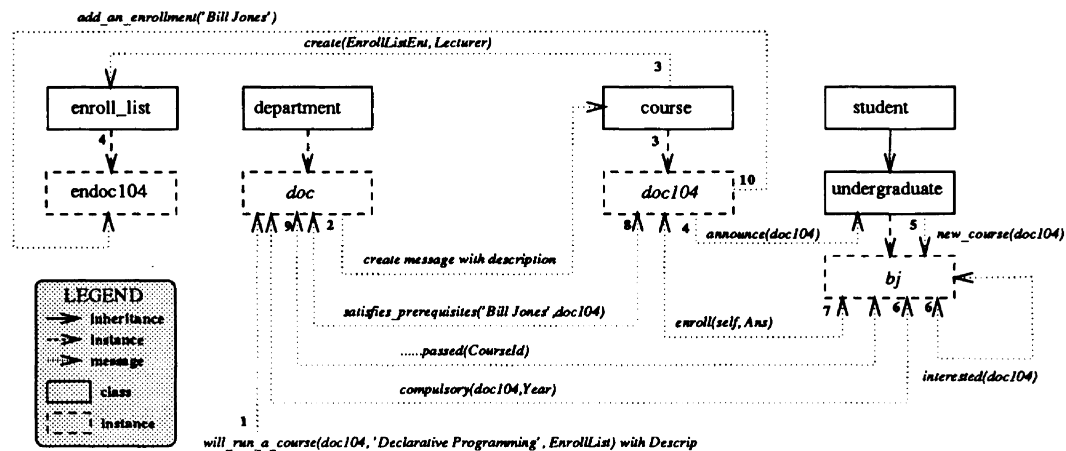

Figure 3 The interaction between objects.

4 EXPLICITLY REPRESENTING ORGANIZATIONAL STRUCTURE

Although the example enterprise model described above includes such things as enterprise rules, encoded in the knowledge components of the instance objects, it has no explicit representation of the organisation that is being modelled. For example, it has no explicit representation, via a fact or rule, that new courses can only be launched by department agents. This is implicit in that only the department class has a method for creating a new course agent. 'Hardwiring' aspects such as this into the model makes it inflexible. It is partly a consequence of having multiple agent classes, with each class endowed with different, specific methods.

A more flexible approach is to factor out the methods and group them into explicitly represented *tasks* which can themselves be combined into explicitly described *roles*. Instead of having different agent classes, we can then have a generic agent class which can spawn agents that are given roles when they are created.

Now, agents have an explicit representation of what they can and must do. More generally, they can also refer to explicit models of the organisation (Fraser 1994, Fox 1993, Fox and Gruninger 1994, Fox, Barbuceanu and Gruninger 1995) whilst performing their tasks. The models describe all the entities in the Enterprise, the relationships between them, the resources they are using, the constraints that might be imposed, the organizational structure etc (Fraser 1994, Fox, Barbuceanu and Gruninger 1995, Fox and Gruninger 1994, Fox 1993, Barbuceanu and Fox 1994), even the workflow.

This alternative approach, which we have also investigated using DK_Parlog^{++}, is based on *role theory*. It is described in more detail in Skarmeas, 1995 and Skarmeas, 1996. Instead of different agent classes, we have *social_position*, *role*, *task*, *contract* and *workflow* classes. The *social_position* class roughly corresponds to a generic agent class.

A specific role is created from the role class by passing to it the identities of the tasks it has to perform. These are held as the value of a state variable within the role object. In this way we can easily change tasks corresponding to a role, we simply send the role object a message to update this state component. Task instances are created by passing to them a detailed description of the plan of action that has to be followed by that task. The detailed plan is held in a state variable of the task object, whilst other information, such as needed resources, is encoded in the public knowledge component of the task.

Specific positions in the organizational hierarchy correspond to instances of the *social_position* class. As instances are created they are assigned an initial set of roles (responsibilities) associated with that social position. These are also stored in a state component of the created object, so can be dynamically updated if need be to reflect changes in the organisational environment. For example, a change that requires a particular social position, say a departmental administrator, to take on new roles.

An example of a social position and its roles is depicted in Figure 4. In this example, we are considering the social position of a secretary of the department of computing, which has been assigned a number of responsibilities. Those involve the *handling of payment forms*, the *handling of stationary* and *the maintenance of departmental information* (Figure 4).

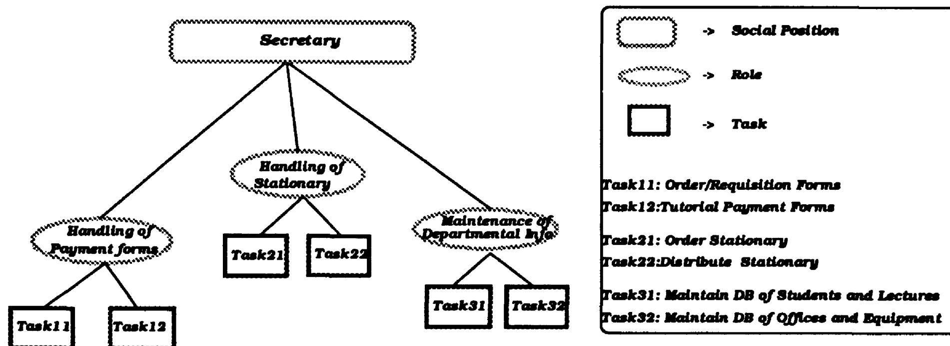

Figure 4 The Secretary Role Structure

Each role contains a number of tasks that the secretary needs to perform upon request. The handling of payment forms for example involve cases like handling of *order/requisition forms* and *tutorial payment forms*. The other roles have also a number of tasks. The tasks, as we mentioned before, are also implemented as objects which are instances of the task class.

Contractual relationships for a social position, which correspond to the hierarchical or reporting relationships between individuals in an organisation, are explicit recorded as contracts, which are instances of the *contract* class. For example, a contract will record the fact that the social position of being the departmental secretary is sub-ordinate to that of departmental manager (Figure 5). The contracts are the main repository of the

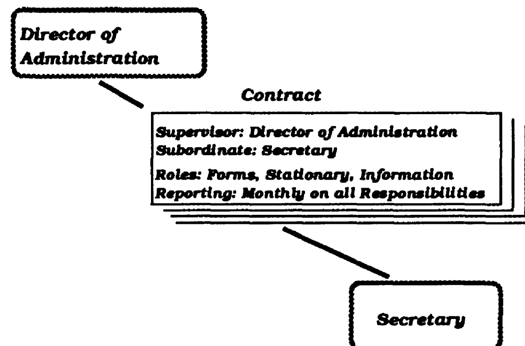

Figure 5 The Contract between Director and Secretary

structural model of the organization.

Contracts record the hierarchical structure of the organization. We also need an entity for describing the lateral relationships between social positions. Those correspond to the workflow activities in the organization, which are a sequence of tasks spanning different social positions (individuals), departments and even organizations. These are *workflow* objects. They hold descriptions of the sequence of tasks that need to be executed, the role or social position location of the tasks, and the resources that are used during the execution of the workflow.

In the following figure (Figure 6) we can see how the task of handling order/requisition forms relates to the activities of the director of administration and the department of finance. This set of activities consists a workflow that is activated whenever a new order takes place.

Notice that, the last participant of the workflow, is not a specific social position but a group (the Department of Finance). As far as the workflow is concerned, the task T4 can be executed by any member of the department of finance.

5 CONCLUSION

In this paper we have proposed a distributed object-oriented programming language that we have developed at Imperial College, as a tool for enterprise modelling and for the prototyping of agent based enterprise integration systems. A small example was presented in order to demonstrate the main features of this language and to illustrate an agent based

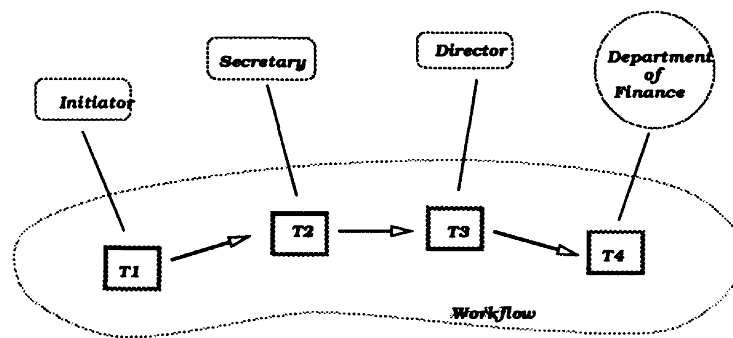

Figure 6 Workflow and Social Positions

approach to active enterprise modelling. We have also briefly described an alternative approach to enterprise modelling based on role theory.

DK_Parlog^{++} is part of our effort to develop tools and techniques that can be used for enterprise modelling. Another language that we have developed, and which we are also using for building agent based applications is April (McCabe, 1995). We have tested both languages on a number of applications related to Enterprise Modeling and Office Automation with encouraging results. Developing applications using these languages is fast and easy once the expertise in using them has been acquired.

ACKNOWLEDGMENTS

The paper was written whilst this first author was visiting the SVRC, Department of Computer Science, University of Queensland supported by funds from the SVRC and the department.

REFERENCES

G. Agha and C. Hewitt. Concurrent programming using actors. In A. Yonezawa and M. Tokoro, editors, *Object Oriented Concurrent Programming.* MIT Press, 1987.

M. Barbuceanu and M.S. Fox. *The Information Agent: An Infrastructure Agent Supporting Collabortive Enterprise Architectures.* IEEE Computer Science Press, Morgantown, WV., 1994.

K. L. Clark and T. I. Wang. Distributed Object Oriented Logic Programming. *ICOT Fifth Generation Computer System Workshop on Heterogeneous Cooperative Knowledge Bases*, 1994.

K.L. Clark and S. Gregory. Parlog:parallel programming in logic. *ACM Toplas*, 8(1):1–49, 1986.

Y. Cosmadopoulos and D. A. Chu. IC prolog II reference manual. Technical report, Logic Programming Section, Dept of Computing, Imperial College, London, 1993.

A. Davison. Polka: a parlog object oriented language. Internal report, Dept. of Computing, Imperial College, London, London, 1988.

M.S. Fox. Issues in enterprise modelling. *Proceedings of the IEEE Conference on Systems, Man and Cybernetics*, 1993.

M.S. Fox, M. Barbuceanu, and M. Gruninger. An organisation ontology for enterprise modelling: Preliminary concepts for linking structure and behaviour. In *Fourth Workshop on Enabling Technologies - Infrastructures for Collaborative Enterprises*, West Virginia University, 1995.

M.S. Fox and M. Gruninger. Ontologies for enterprise integration. In *Proceedings of the 2nd Conference on Cooperative Informatio Syst*, Toronto, Ontario, 1994.

John Fraser. Managing change through enterprise models. In R Milner and A. Montgomery, editors, *Applications and Innovations in Expert Systems II.* SGES Publications, 1994.

F. G. McCabe and Keith L. Clark. Programming in April:An Agent PRocesss Interaction Language. In *Intelligent Agents.* Springer Verlag, 1995.

Nikolaos Skarmeas. Organizations Through Roles and Agents. *COOP'95: International Workshop on the Design of Cooperative Systems*, pages 385–404, January 1995.

Nikolaos Skarmeas. Process Based Support for Offices. *PhD Thesis, forthcoming*, 1996.

Jeff Y. C. Pan and Jay M. Tenenbaum. An intelligent agent framework for enterprise integration. *IEEE Transaction on Systems, Man and Cybernetics*, 21(6):1391–1407, November 1991.

E. Shapiro and A. Takeuchi. Objected oriented programming in concurrent prolog. *New Generation Computing*, 1(1), 1983.