Agents as Multi-threaded Logical Objects

Keith Clark[1] and Peter J. Robinson[2]

[1] Department of Computing, Imperial College, London, England
klc@doc.ic.ac.uk
[2] School of Computer Science and Electrical Engineering, The University of Queensland, Australia
pjr@csee.uq.edu.au

Abstract. In this paper we describe a distributed object oriented logic programming language in which an object is a collection of threads deductively accessing and updating a shared logic program. The key features of the language, such as static and dynamic object methods and multiple inheritance, are illustrated through a series of small examples. We show how we can implement object servers, allowing remote spawning of objects, which we can use as staging posts for mobile agents. We give as an example an information gathering mobile agent that can be queried about the information it has so far gathered whilst it is gathering new information. Finally we define a class of co-operative reasoning agents that can do resource bounded inference for full first order predicate logic, handling multiple queries and information updates concurrently.
We believe that the combination of the concurrent OO and the LP programming paradigms produces a powerful tool for quickly implementing rational multi-agent applications on the internet.

1 Introduction

In this paper we describe an object oriented extension of the multi-threaded Qu-Prolog described in [7]. We show how this can be used to quickly implement multi-agent applications on the internet in which agents have both reactive and pro-active behaviours that utilize quite rich inference systems. The different behaviours execute concurrently, as separated threads of an active object that implements the agent.

The original Qu-Prolog [12] was developed as an implementation and tactic language for interactive theorem provers, particularly those that carry out schematic proofs. It has built-in support for the kinds of data values typically needed when writing a theorem prover in Prolog: object variables - the variables of the logical formulae being manipulated, substitutions for these object variables, and quantified terms, terms denoting object level formulae with explicit quantifiers over the object level variables. As further support, the unification algorithm of Qu-Prolog unifies such quantified terms up to alpha-equivalence, that is it knows about equivalence up to changes of quantifier bound object variables. It also carries out the occurs checks before binding a variable. This is essential for implementing sound inference systems. Qu-Prolog is the implementation

A.C. Kakas, F. Sadri (Eds.): Computat. Logic (Kowalski Festschrift), LNAI 2407, pp. 33–65, 2002.

language of the Ergo theorem prover [1], which has seen substantial use in the development of verified software.

Motivated by a desire to implement a multi-threaded, multi-user version of Ergo, we then added multi-threading and high-level inter-thread communication between Qu-Prolog threads running anywhere on the internet [7]. Each thread has a internet wide unique identity similar to an email address. It also has a message buffer of received but unprocessed messages which it can periodically search for messages of interest. Communication between threads in different Qu-Prolog processes makes use of the store and forward ICM communications system [17] developed for the April language [18]. This offers robust middleware for distributed symbolic applications. As an example, it can be configured to automatically store messages for threads running on hosts, such as laptops, that are temporarly disconnected, delivering them when the laptop reconnects.

In [7] we describe the multi-threading and the inter-thread communication facilities in detail and show how they can be used to implement a distributed deductive data base in which each data base comprises the clauses of a program being executed by a multi-threaded Qu-Prolog process. The clauses in each data base can contain remote calls that are queries for relations defined in other data bases. Such a remote call takes the form `DB?Call`, where `DB` is the global identity of the query interface thread for the other Qu-Prolog process. `DB` typically has a value such as `interface:qupDB@'zeus.doc.ic.ac.uk'`. The interface thread can fork a new query thread for each received remote query. Moreover, although we did not illustrate this in [7], different deductive data bases can do inference using a different logic, a non-resolution inference system or even a modal logic. Since each can have rules that call for sub-proofs in other deductive data bases, we can easily implement distributed hybrid reasoning systems.

Threads in different invocations of Qu-Prolog can only communicate using messages, but threads within the same invocation can also communiciate via the dynamic clause data base. Asserting or retracting a clause is an atomic operation with respect to the multi-threading. In [7] we showed how we can use the shared dymamic clauses to implement a Linda-style tuple space manager in Qu-Prolog. In addition, threads can be made to suspend waiting for a particular clause to be asserted. Suspension waiting for a clause of a certain form to be asserted enables one to implement daemons. A daemon is a thread that is launched but which immediately suspends until the trigger clause is asserted.

In [8] we sketched how multi-threaded Qu-Prolog could be used to implement DAI applications. With this type of application in mind, we have recently added a concurrent object oriented layer to Qu-Prolog. This OO layer, which in this paper we shall refer to as QuP^{++}, is transformed into the base Qu-Prolog using the term expansion pre-processing facilities of Qu-Prolog. It allows OO software engineering methodology to be used to construct distributed Qu-Prolog applications, in particular multi-agent applications.

In the next section we give a brief overview of the main features of QuP^{++}. This is followed by section 3 which is an example based introduction to programming in QuP^{++}. In section 4 we show how object servers allowing remote

spawning of objects can be defined and used to create and manage mobile objects and agents. In section 5 we introduce the features of Qu-Prolog that allow the implementation of non-resolution inference. We show how they can be used to define a reasoning agent that can do resource bounded inference for full first order predicate logic both to answer questions about what it believes and to check for possible inconsistency before it adds new information to its belief store. We then elaborate the agent to a co-operative reasoning agent that can ask other agents to engage in sub-prrofs on its behalf. In section 6 we conclude with mention of related research.

2 Overview of QuP^{++}

QuP^{++} is a class based OO language with multiple inheritance. A class is a named collection of static Qu-Prolog clauses with an optional state component comprising a collection of dynamic predicates and state variables, the latter being Qu-Prolog atoms. The stucture of a class definition is:

```
class C isa [S1,.Si-[r/2]..,Sn]  % optional inheritance
state [d/3,a:=9,b,{k(g,h). k(j,l)},...] % state components
clauses{                          % sequence of static clauses
p(...):-  ...
...
p(...):-super?p(...).
....
}private [d/3,..]  % preds that are private
```

The dynamic predicates (of the object state) must be disjoint from the static predicates of the class and any of its super-classes. Instances of the class share the static clauses but *do not* share clauses for their dynamic predicates and *do not* share state variable values.

A class definition with a state component is the template for an object. An object is an instance of the class. The static clauses of the class are the fixed methods of the object. Objects are *active*, each is implemented as one or more independently executing threads. The clauses for the dynamic predicates and the values associated with the state variables are the state of the object. Default initial clauses for the dynamic predicates can be given in the class definition, e.g. the clauses for `k/2` above, as can default initial values for the state variables, e.g. `a:=9`. A default value for a state component given in a class `C` over-rides any default value given for the same state component in a super-class of `C`. A state variable value can only be accessed and updated from the methods of the class, and clauses for a dynamic predicate can only be asserted and retracted by a class method. However, the dynamic predicates of an object can be queried in the same way as the static predicates. Externally they look like extra method names. They are methods with dynamic definitions unique to each object.

Static predicate names and state component names can be re-used in different classes, they are treated as distinct names. Inheritance, by default, makes

all the static predicates of the super-classes of a class C static predicates of C. If an inherited predicate is redefined in a class, the new definition over-rides the inherited definition. However, the combined superclass definition for a predicate p/n can always be accessed from inside C with a call super?p(...). Using super?p(...) we can make the new definition extend what would have been the inherited definition, as in:

```
p(...):-  ...
...
p(...):-super?p(...).
```

More precisely, the definition for p/n given in a specific super-class S can also be accessed with a call super(S)?p(...). If the predicate p/n is not redefined in C, the definition that is inherited in C is exactly the same as if it were redefined in C as:

```
p(X1,..,Xn):-  super(S1)?p(X1,..,Xn);
               super(S2)?p(X1,..,Xn);
               ...
               super(Sj)?p(X1,..,Xn).
```

Here S1,..,Sj are all the superclasses of C from which inheritance of p/n has not been explicitly suppressed. Inheritance of the clauses for p/n, from a specific super-class S is suppressed by using of S-[p/n], rather than S in the isa list of super-classes.

A call p(...) in a static clause of a class C always denotes a call to the definition for p/n of the class C, even if the call is executed inside an object O that is an instance of a sub-class SubC of C that has redefined p/n. In contrast, a call self?p(...) in a static method of C executed by O will be evaluated using the definition for p/n of SubC.

Inheritance unions the state components of a class C with the state components of all its superclasses. That is, all state variables of a super-class are automatically state variables of C, and all dynamic predicates of a super-class are automatically dynamic predicates of C.

By default, all the static and dynamic predicates of a class are visible, that is they can be used in queries to the object instances of the class. Both static and dynamic predicates can also be declared as private, in which case they can only be called from methods of the class and its sub-classes[1]. Queries to instances of the class cannot access the clauses for the private predicates. Such a call to a private predicate of an object will fail.

An object instance of a calls C is created with a call of the form:

```
new(C,...,O)
```

where O is an unbound variable which will be assigned a system generated globally unique identity for the new object. O is actually the identity of the object's

[1] Private predicates are inheritable and can be redefined in sub-classes.

default execution thread. This thread will immediately call the `init` method of class `C`, if this is defined. This can be used to launch sub-threads of object `O` using the QuP^{++} `object_thread_fork` primitive. The object sub-threads can communicate with one another either by explicit messages using the inter-thread message primitives of Qu-Prolog, or by updating `O`'s dynamic clauses or state variables. Special QuP^{++} `self_assert` and `self_retract` primitives enable any thread within an object to update the dynamic clauses of the object. The QuP^{++} primitives `*=` and `:=` enable any object thread to access and update the value of one of the object's state variables[2]. The `init` method can also be used to announce the object's presence by remote calls to other objects, for example a call to a directory server registering some description of the object. On termination of the `init` method, the default thread enters a loop in which it repeatedly accepts and evaluates remote calls for `O`. It suspends if there are no pending remote calls. It becomes the object's external interface thread - its reactive component.

A remote call is a query `Q` sent to `O` from another concurrently executing object, anywhere on the internet. The query can be sent as a call `O?Q`, or a call `O^^Q`[3]. (The differences between the two forms of call will be explained shortly.) `Q` can be an arbitrary Prolog query using any of the visible predicates of the class of `O` or any Qu-Prolog primitive[4]. Multiple remote calls, whether synchronous or asynchronous, are queued at an object in time order of arrival. The object will respond to them in this order.

A `?` call is a *synchronous* communication, the client querying thread `Cl` suspends until an answer is returned, which may be a fail message. Backtracking in the client thread will generate all solutions of the remote call[5].

A call `O^^Q` is an *asynchronous* remote call. `Q` is executed by `O` as a single solution call. There is no automatic answer response from `O` to such a query, no client variables in `Q` will be bound as a result of the call, and on the client side the call always *immediately succeeds*. Usually `Q` will cause some update of the state of `O`, or cause `O` to execute a remote call. This remote call could be either a synchronous or an asynchronous call back to the object from which the query was sent. The architecture of a QuP^{++} object is depicted in figure 1.

During the evaluation of any remote call received by an object `O`, the global identity of the object `QO` from which the query came can be found by executing a call `caller(QO)`. This will unify `QO` with the global identity of the querying

[2] Execution of the dynamic clause and state variable update and access primitives is an atomic action. However it is a useful discipline to restrict update of a particular dynamic predicate or state variable to a particular sub-thread and have other threads only access the value.

[3] There is also a `O??Q` form of call with the semantics as given in [7]. We shall not use this form of call in this paper.

[4] In addition, any predicate of a Qu-Prolog program can be used in `Q` if we know that it will have been loaded by the Qu-Prolog process in which `O` is running. To the QuP^{++} application these are seen as extra Qu-Prolog primitives.

[5] For a call `O?Q` all solutions to `Q` are immediately found by `O` using a `findall` call and returned by `O` to `Cl` as a list. There is then local backtracking in `Cl` over the different solutions in the returned list.

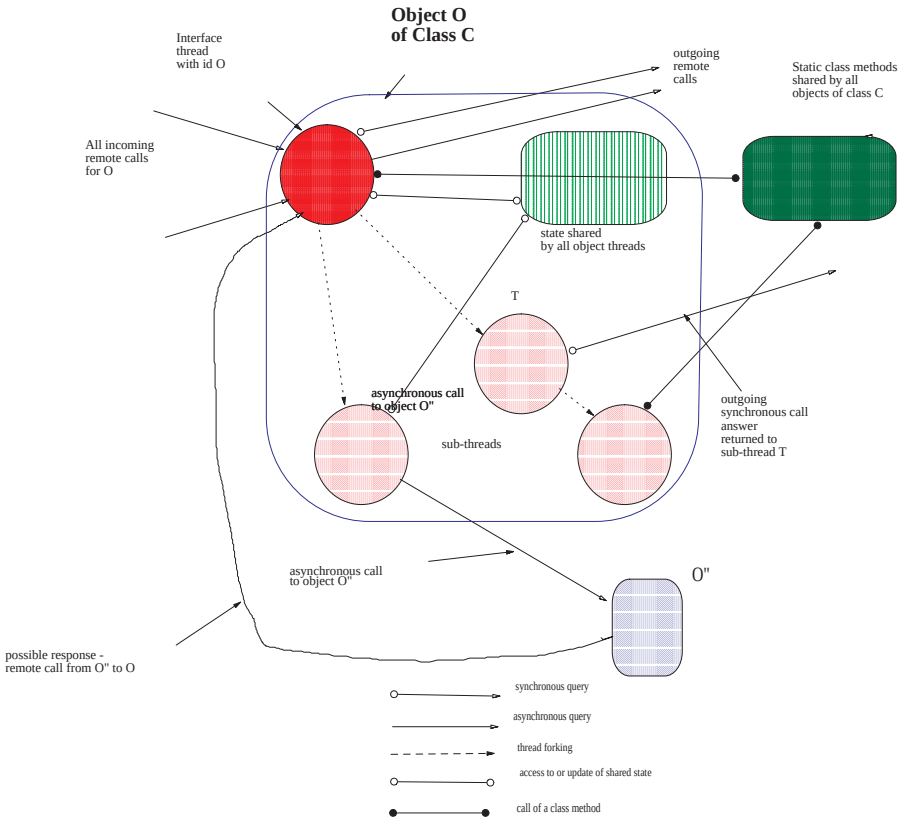

Fig. 1. A QuP^{++} object

object, which remember is the global identity of its interface thread. This will be the case even if the query came from another sub-thread of `QO`. The pair of calls, `caller(QO),QO^^RCall`, thus sends an asynchronous query `RCall` to the object `QO` which sent `O` the remote call it is currently evaluating. If `O` executes this pair of calls whilst evaluating an asynchronous call, `O^^Q`, from `QO`, the return call `QO^^RCall` is effectively a response to `QO` for the query `Q`. Use of `^^` remote calls and `caller/1` enables objects to have asynchronous conversations as well as client server interactions. This is particularly useful when the objects implement agents.

Tests on the value returned by a `caller/1` call can also be used to restrict use of certain methods to known objects, or objects satisfying certain properties. For example, a method:

```
p(...):- caller(QO), allowed_to_call_p(QO), ...
```

causes a remote call to `p` to fail if the querying object is not allowed to call `p`. `allowed_to_call_p/1` can be a dynamic predicate initialised when the object is created, and perhaps updated by calls to a method:

```
allow_to_call_p(NewO):-caller(QO),allowed_to_call_p(QO),
                       self_assert(allowed_to_call_p(NewO)).
```

from objects already allowed to call `p/k`.

3 QuP^{++} by Example

Let us begin with a simple example program. This is a class definition for a person object. In this case there is no inheritance, except from a default system class that defines a set of standard method predicates for all objects. One of these is the reflective method `predicate/1` which can be used to query an object to find its visible predicates. A call `O?predicate(V)`, where V is a variable, will return one at a time the names and arities of `O`'s visible predicates. Another system class predicate is `class/1`. A call `O?class(C)` will unify `C` with the class name of `O`. There are two other reflective predicates: `myid/1` and `mystate/1` which are actually used in the above class definition. They can only be called from a method. `myid/1` unifies its argument with global identity of the object that calls it. `mystate/1` returns the entire current state of the object that executes the call as a list.

```
class person
state [firstname/1,surname/1,age:=0,sex/1,child/1,parent/1]
clauses{
adult :- age*=A,A>18.
family_name(N):-surname(N).
likes(O):-child(O).

new_child(Fn,Sx,O):-
    nonvar(O),!,
    self_assert(child(O)).
new_child(Fn,Sx,O):-
    surname(Sn),
    myid(Me),
    new(person,
        [firstname(Fn),surname(Sn),sex(Sx),{parent(Me). }],
        O),
    self_assert(child(O)).

get_married_to(Sp):-
      myid(Me),
      Sp?(class(person);class(married_person),spouse(Me)),
      mystate(St),
      become(married_person,
                       [spouse(Sp)|St]).

} private {surname/1}.
```

Let us now look more closely at the above class definition. The state declaration:

```
state [firstname/1,surname/1,age:=0,sex/1,child/1,parent/1]
```

tells us that instances of this class will record the state of the object using clauses for five dynamic predicates and one state variable `age`. The state variable has a default initial value of `0`. When we create an instance of the class we can give values for the dynamic predicates and we can override the default value `0` for `age`. For example, the call:

```
new(person,[firstname(bill),surname(smith),sex(male),age:=23],O1)
```

will create a new instance of the `person` class, with the clauses given in the state list second argument as initial definitions for its dynamic predicates, and the value `23` for its `age` state variable. The clauses for the dynamic predicates and the state variable initialisations can be given in any order. Notice that this `person` object does not have clauses defining `parent/1` and `child/1`.

When an object is created it can be given a set of clauses for some or all of its dynamic predicates and values for some or all of its state variables. For a dynamic predicate these either over-ride or add to any default clauses given for the predicate of the class definition. The choice is signalled by the way the clauses are given in the object creation call. For a state variable any value given in the object creation call always over-rides any default value it might have in the class definition.

`new/3` is one of two QuP^{++} primitives for creating new objects. The above call to `new/3` returns the global indentity of the person object as the binding for `O1`. We can access `O1`'s state as recorded by its visible dynamic predicates by queries such as:

```
O1?sex(S)
```

which binds `S` to `male`. We cannot directly access the `age` of `O1` since this is recorded as the value of a state variable. However we can use the `adult` method to indirectly access its value. For example,

```
O1?adult
```

will succeed. The call `age*=A` in the `adult` clause uses the QuP^{++} primitive `*=/2` to access the current value of the `age` state variable. This call can only be used in a method. An attempt to use it in a remote call such as `O1?age*=A` will fail.

A call:

```
O1?predicate(P)
```

will in turn bind `P` to each of:

```
new_child/3, adult/0, family_name/1, get_married_to/1,
likes/1, firstname/1,  sex/1, child/1, parent/1
```

`surname` will not be returned as it was declared as private to the class. Its definition can be accessed indirectly via the `family_name` method. We have a separate `family_name` definition because, when we define the `married_person` subclass, we shall redefine this predicate.

```
O1?class(C)
```

will bind `C` to `person`.

The `person` class has a method `new_child/3` that both updates the state of the object that executes it and may create a new instance of the person clause, which is the object representing the recorded child. The asserted `child/1` clause records the child object's global identity. A new person object is created if the third argument of the `new_child/3` method call, the object identity of the child, is given as an unbound variable. Thus, a call:

```
O1?new_child(mary,female,O2)
```

will result in a new person object with the global identity the returned binding for `O2` being created with state:

```
[surname(smith),firstname(mary),age:=0,sex(female),parent(O1)]
```

The `new_child/3` second clause is used and this calls the dynamic predicate `surname/1` to access the surname for object `O1` in order to define the `surname/1` dynamic predicate of the new person object that it creates. It also calls the QuP^{++} primitive `myid` to find the global identity of the object executing the method[6]. This is in order to give an initial clause for the `parent/1` dynamic predicate of the new person object, which is deemed to be a child of the object executing the `new_child` method. Finally the `new_child/3` method adds the clause `child(O2)` to the state of `O1` using the QuP^{++} primitive `self_assert`. `self_assert` rather than `assert` is used to ensure that the dynamic clauses for the same predicate in different objects are kept distinct.

Now a query:

```
O2?firstname(F)
```

or the equivalent queries:

```
O1?child(C),C?firstname(F)
```

```
O1?(child(C),C?firstname(F))
```

[6] In many OO languages the returned binding for `Me` is denoted by use of the term `self`. In QuP^{++} `self` can only be used as the object identity of a call, as in `self?p(..)`. If we want to embed its value as an argument of a remote call, as here, we must find its value using `myid/1`. As we remarked earlier, a `self?p(...)` call can be used within a method of a class `C` to signal that the latest definition of `p` should be called in case the method is being executed by an instance of a subclass of `C` which redefines `p`. This is a standard use of `self` in OO languages.

can be used to find the first name of the new child object. The last two queries differ with respect to where the call `C?firstname(F)` is executed. In the first query it is executed in the object that executes the call `O1?child(C)`, and in the second it is executed in the object `O1`. The second is a remote call containing a remote call. Remember all the objects are executing as separate threads which repeatedly accept and execute remote calls. The differences between the evaluations of the two queries is depicted in figure 2.

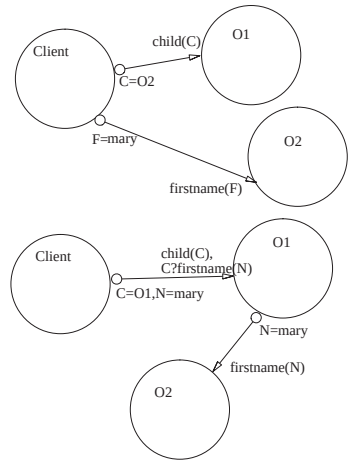

Fig. 2. Remote calls

Let us now look at the method `get_married_to/1`. This does not create a new object but metamorphises the person object that executes it into an instance of the `married_person` class. This is as a result of the call to the the QuP^{++} primitive `become/2`. This can be called by a static method of any object `O` and when the method that calls it terminates the object `O` becomes an instance of a new class. Importantly, it retains the same global identity. The first argument of the `become/2` call is the name of the new class, the second is a list, just like the list argument of a `new/3` call, giving values for some or all the state components for the object as an instance of the new class. In the case of the `become/2` call of the `get_married_to/1` method the new state list is the state list returned by executing the system class method `mystate` with the clause `spouse(Sp)` added as a new component. Notice that the method only succeeds if `Sp` is an instance of the person class (i.e. as yet unmarried), or `Sp` is an instance of the `married_person` call that has the person being told to get married (the `Me` returned by the call `myid(Me)`) as its recorded spouse. A call to `mystate/1` unifies its argument with a list giving the current complete state of the object `O` that executes the call. The state of an object `O` as a married person is its state as a person object with an extra clause for a new dynamic predicate `spouse/1`.

This clause records the identity of the object to whom the married person is married.

As one can imagine, the `married_person` class is best defined as a sub-class of the person class. Its definition is given below. The `isa person-[get_married_to/1]` of the class declaration means that all the static clauses and state components of the `person` class, except the clauses for `get_married_to/1` which is not inherited and `family_name/1` and `likes` which are redefined, are automatically included in the `married_person` class. Note that the sub-class redefines the `likes/1` predicate as:

```
likes(O):- spouse(O);super?likes(O).
```

This redefinition calls the definition that would be inherited so it just extends the `person` definition for `likes/1`. Note that `get_married_to/1` is removed from the methods of the `married_person` class.

The sub-class also has a clause for the predicate `init`. When a class contains a definition for `init`, which is always deemed as private to the class, it is called immediately after any instance of the class is created, either by a `new` call, or a `becomes` call. Only when the `init` method terminates will the object accept external queries.

```
class married_person isa person-[get_married_to]
state [spouse/1]
clauses {
init:-  spouse(Sp),
        myid(Me),
        Sp?spouse(Me) -> true;
                         Sp^^get_married_to(Me).

likes(O):- spouse(O);super?likes(O).

family_name(N):- sex(male) -> surname(N) ;
                              spouse(Sp),Sp?surname(N).

get_divorced:-
        mystate(St),
        remove(spouse(Sp),St,NSt),
        myid(Me),
        (Sp?spouse(Me)->Sp^^get_divorced),
        become(person,NSt).
}.
```

Let us see what the effect of the `init` is if we execute the conjunction:

```
new(person,[firstname(june),surname(jones),
            sex(female),age:=20],O3),
O3^^get_married_to(O1)
```

where O1 is the previously created male instance of the person class. The call O3^^get_married_to(O1) is an asynchronous call. It always immediately succeeds whether or not the call get_married_to(O1) succeeds in the object O3. No answer bindings are ever directly returned from an asynchronous call and so the query of the call usually contains no unbound variables, as here.

When O3 receives the query it will eventually execute:

```
become(married_person,
       [spouse(O1),firstname(june),surname(jones),
        sex(female),age:=20])
```

and this causes O3 to become an instance of the married_person class. This in turn, will cause the automatic execution of the init method of this class by O3. This will query O1, the recorded spouse of the metamorphised O3, to see if O1 'knows' that its spouse is the object executing the init method, i.e. O3. The init method finds the global identity O3 by executing the call myid(Me). Since O1 is at this time an instance of the person class, it will have no clauses for spouse, and the call Sp?spouse(Me) will fail. This will result in the execution by O3 of the asynchronous remote call:

```
O1^^get_married_to(O3)
```

and this will cause O1 to metamorphise into an instance of the married_person class, with recorded spouse O3. Now the init call executed when O1 becomes a married person will find that its spouse O3 does 'know' that it is married to O1 and the distributed activity started by the init executed by O3 will terminate. The init method ensures consistency between the state components of the two married person objects.

Note that it is essential that the remote call to get_married_to/1 of the init method is executed asynchronously. Before the remote call terminates, the object that executes the call will itself be queried. The interaction between O1 and O3 is as depicted in the figure 3. If O1 executed the remote get_married_to(O1) query to O3 synchronously, that is if it suspended until the remote query successfully terminated, it would not be able to respond to the synchronous query spouse(O3) from O3. The two objects would deadlock, and neither would be able to complete their init methods.

Finally let us look at the get_divorced method for a married person. This causes a married_person object O to metamorphise back into a person object and ensures that the recorded spouse, if it 'believes' it is still married to O, similarly reverts to being a person.

4 Object Servers and Mobile Agent Objects

Below is a definition of an object server class. Instances of this class can be sent messages to remotely spawn objects and can be used as stepping stones by mobile agent objects.

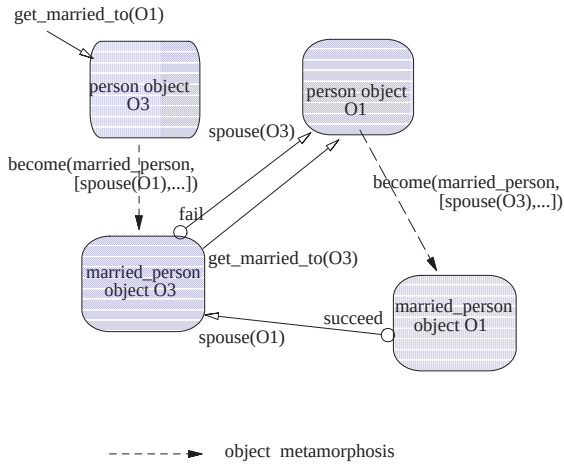

Fig. 3. Object state synchronisation

An `object_server` accepts requests to create new objects for a particular class keeping track of which objects it has created, in which class, in a dynamic predicate `class_of/2`. It also allows objects to be created with given public names, as we shall describe below. It keeps track of these public names in a dynamic relation `used_names`. The two dynamic predicates are not private, so both can be queried by other objects. Use of such an object server assumes that the class definitions for all the classes for which it may need to create instances have been loaded by the Qu-Prolog process in which the object server is running.

```
class object_server
state [class_of/2,used_name/1]
clauses {
  newob(C,Inits,O) :-
       var(O),
       new(C,Inits,O),
       self_assert(class_of(C,O)).
  newob(C,Inits,N,O) :-
       atom(N),
       var(O),
       \+ used_name(N),
       new(C,Inits,N,O),
       self_assert(used_name(N)),
       self_assert(class(C,O)).
}.
```

The class has two methods, one for `newob/3` and one for `newob/4`. The first takes the name of the class and the state components and creates a new object with a system generated identity `O` that will be returned to the client providing

the method was invoked as a synchronous query. The method for `newob/4` has an extra argument, `N`, which must be an atom. It then calls the four argument `new` primitive passing in this symbol `N`. This will use `N` to construct the global identity `O`. For example, suppose we have an instance of the `object_server` class running on a machine `'zeus.doc.ic.ac'` within a Qu-Prolog process with the name `objects`. The Qu-Prolog process can be given this name by a command line option when it is started. If we send it the remote synchronous query:

```
newob{person,[firstname(bill),...],billS,O)
```

then `O` will be bound to:

```
billS:objects@'zeus.doc.ic.ac.uk'
```

providing `billS` is not already a used name for an object already created by the object server. (The already used names can be found by querying its `used_names` dynamic relation.) This is a public global identity that can be used to refer to this particular `person` object in any QuP^{++} application. A call:

```
billS:objects@'zeus.doc.ic.ac.uk'?family_name(N)
```

from any QuP^{++} object, anywhere on the internet, will be routed to the object via the ICM[17] message transport system[7].

More usefully, we can give such a public identity to the object servers running on each internet host. We can do this by launching each object server, in a Qu-Prolog process with the name `objects`, with a call:

```
?-new(object_server,[],server,_).
```

If we do this on the host `zeus.doc.ic.ac.uk`, we can remotely launch an object on this host with a remote call:

```
server:objects@'zeus.doc.ic.ac.uk'?newob(person,[...],O).
```

or, if we want the launched object to have a public name, with a query:

```
server:objects@'zeus.doc.ic.ac.uk'?newob(person,[...],billS,O).
```

As we remarked earlier, such a remote launch requires that the class definition for `person` has been loaded on `zeus.doc.ic.ac.uk`. We could, however, elaborate the object server so that it keeps track of which class definitions have been loaded, loading new ones as required. Then all that we need to assume is that we only use a given object server to create objects for classes to which it has access to the class definition.

[7] This typically requires ICM processes to be running on each host on which we have a QuP^{++} process running.

Consider now the class definitions:

```
class mobile_object
clauses {
move_to(Host,O):-
  mystate(St),
  class(C),
  server:objects@Host?newob(C,St,O),
  die.
}.

mobile_person isa [person,mobile_object].
```

The `mobile_object` class is an abstract class. It will have no direct instances but can be used as a super-class whenever we want some class of objects to be re-locatable. The `mobile_person` class inherits from this class, and the `person` class.

The single method of the `mobile_object` class takes the name of a host machine, `Host` and relocates the object by sending a remote `newob/3` query to the publically named object server on that host. Executed by a `mobile_person` object, the call `mystate(St)` will bind `St` to the person state component and the call `class(C)` will bind `C` to `mobile_person`. The last action of the method, executed if the remote `newob` call succeeds, is `die`. This terminates all the threads executing within the object on the current host.

Suppose `O1` is mobile person object initially created by a `newob/3` query to some object server. If we then execute[8]:

```
O1?move_to('pine.doc.ic.ac.uk',O2)
```

then, providing there is an object server running on that host, the object `O1` will relocate to become the object with global identity `O2`. This safely relocates an object that only has the default interface thread executing at the time it is relocating and the `move_to` is executed by this thread. If we want to relocate a multi-threaded object we should program it so that all threads but the interface thread have terminated, perhaps after recording information about their execution state in the state of the object, before `move_to` is executed. The object's class should then have an `init` method that will re-launch the additional threads when the object is re-launched on the new host.

Of course, if we are to have objects moving from object server to object server, we should augment the object servers so that they can be informed when an object moves. We should add a new method to the object server class:

```
moved_to(NewHost):-
  caller(O),
  self_retract(class_of(O,C)),
  (O=N:_@_,atom(N)->self_retract(used_name(N));true).
```

[8] We can also identify the host using its IP number

and the move_to/1 method of a mobile object should be:

```
move_to(Host,O):-
  mystate(St),
  class(C),
  server:objects@Host?newob(C,St,O),
  myid(_:objects@CurrHost),
  server:objects@CurrHost^^moved_to(Host),
  die.
```

Notice that the new moved_to/2 method of the object server uses caller /1 to find the identity of the local object that is moving, and the move_to method finds the identity of the object server that should be informed of the move by massaging the term that is its own global identity. It makes the assumption that all these moving objects are created by newob messages to object servers and hence have global identities of the form:

```
Name:objects@CurrHost
```

This is the case even if the object is not given a public name, Name is then an atom such as object234.

To many, a mobile agent is a mobile object with a purpose. The purpose manifests itself in proactive behaviour when the agent object arrives at a new site. Below is a class definition for a two threaded generic mobile agent object.

```
class mobile_agent isa mobile_object
state [name,hostlist,script/1,report_to]
clauses {
init:-
   hostlist*=[CH|Hosts], % find where I am -- head of hostlist
   hostlist:=Hosts,      % update hostlist
   report_to*=R,         % find agent to report to
   name*=N,              % find my name
   myid(Me),             % find my current global id
   R^^i_am_now(N,Me),    % inform report_to agent of new id
   object_thread_fork(_,script(CH)). % execute script for CH
                                     % as a separate thread
} private [move_to].
```

It has a state component which is a list of hosts to visit, and a script of what to do as it arrives at each host. The script is given by clauses for the dynamic relation script/1. It has another state component, report_to, which is the global identity of an agent to which it should report, and one called name which is some name by which it can be recognised. Each time it arrives at a host it executes the init method. This sends an asynchronous call to the report_to agent object giving its current global identity. This is so that the report_to agent can send remote queries accessing its current state.

The `init` method of this class also calls the `script` progam passing in the name of the current host which is assumed to be the first host on `hostlist`. The script is executed as a separate object thread so that the main thread of the object can become the default interface thread responding to remote calls, in particular calls from the `report_to` agent that will have been informed of its current identity. It also updates `hostlist` by removing the current host name. The called `script/1` program will typically end by executing a `move_to/1` call on the inherited method of the `mobile_object` class. To implement a mobile agent we only need to assume that this generic class definition is available on each host that the agent will visit. The actual `script` for the mobile agent will be passed as part of the state component of the agent and will be agent specific.

```
server:objects@H1?newob(mobile_agent,
                  [hostlist:=[H1,...,'zeus.doc...'],report_to:=R,
                  {script('zeus....'):- % script for home base
                    make_visible(found_pair/2),
                    !.                % terminate script thread
                   script(H):-          % script for elsewhere
                    make_visible(found_pair/2),
                    forall(server:objects@H?
                              class_of(Mp,married_person),
                           (Mp?(sex(male),spouse(Sp)),
                            self_assert(found_pair(Mp,Sp)))),
                    hostlist*=[H|_],
                    self^^move_to(H,_).}],_)
```

The above call creates a mobile agent that moves to each of the list of hosts `[H1,...'zeus.doc...']` reporting to an agent object $\widehat{\mathrm{R}}$. It is initially created on `H1`. In all but the last host `'zeus.doc...'`, which is its home base, perhaps the host on which `R` resides, it queries all the local `married_person` objects to create a list of the `married_person` pairs on that host. It finds the identities of the `married_person` objects by querying the `class_of` relation of the local object server. The found married person pairs, if any, are cached in a new dynamic relation `found_pair`. `self_assert` can be used to add clauses for dynamic relations that are not declared in the state component of an object's class. By default they become additional private dynamic relations of the object and are automatically collected as part of the state list constructed by `mystate/1`. So the clauses for these additional dynamic relations will move with the mobile agent. Any private dynamic predicate can be made visible if the object executes a call to `make_visible/1`. This is what our mobile agent script does at each host, allowing the `report_to` agent to query the `found_pair/2` relation each time the mobile agent reports its new identity. Finally note that the last action of the script, at other than the home host, is an asynchronous call `self^^move_to(H,_)` to itself. This is instead of of a direct method call `move_to(H,_)`. The direct call would result in the inherited `move_to` method being executed in the script thread, whereas the asynchronous `self` call results in its being sent as an asynchronous

remote call to the interface thread of the mobile agent. Sending it for execution in the interface thread is cleaner. It means that when it is executed the script thread on the current host will have terminated because it immediately terminates after sending the `self` call. It also means that any remote synchronous call currently being executed by the interface thread, and any such calls that are pending, will be completed before the `move_to` method is executed by this thread. (Remember that remote calls are queued and executed by the interface thread in time order of arrival.)

This is a very simple mobile agent program but the agent, in its ability to concurrently accept queries about the information it has gathered, whilst it is gathering new information, is quite sophisticated. Its activity is as depicted in figure 4. We can use the same program to launch mobile agents with scripts that find out new hosts to visit, adding the host name to `hostnames`. We can also define other mobile agent classes, inheriting from this class, or directly from `mobile_object`, that allow agents to be recalled or given new scripts on their journey.

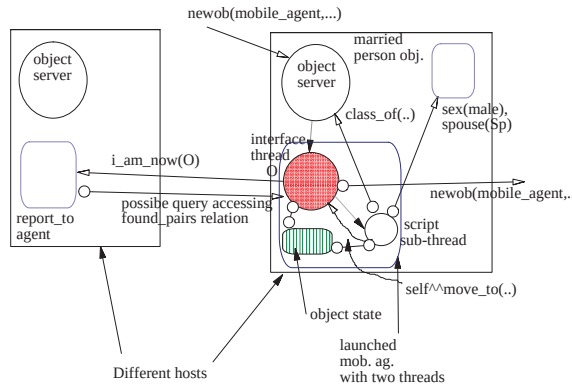

Fig. 4. Simple mobile agent

5 Advanced Reasoning Agents

In this section we show how the Qu-Prolog support for quantifiers, substitutions and object variables can be used to implement powerful reasoning agents that go beyond Prolog inference.

In order to support the programming of such reasoning agents the Herbrand Universe (or object-level) of Qu-Prolog extends that of normal Prolog. Qu-Prolog's Herbrand universe has quantified terms and object level variables. Correspondingly the meta-level of Qu-Prolog includes terms to represent the

object-level quantified terms and variables. Object variables (or more strictly object-variable variables) are meta-level variables that range over variables at the object-level. This means that one object variable may be bound to another during unification, but cannot be bound to any other kind of term.

Qu-Prolog also supports a notation for substitution application. Such a meta-level term represents the application of a substitution to a term at the object-level with change of bound variables as required.

Unification in up to alpha-equivalence. In other words, the unification algorithm attempts to find instantiations of variables that make two terms equal up to change of bound variables. We present some example unification problems shortly to illustrate the unification of quantified terms.

Note that, in Qu-Prolog, there is a distinction between substituition and instantiation. When talking about *substitution* we mean variable substitution at the object-level and consequently change of bound variables is required when 'pushing' a substitution into a quantified term (at the object-level). On the other hand, *instantiation* (often called substitution when discussing standard Prolog) is really substitution at the meta-level. Instantiations therefore 'move through' terms representing object-level quantified terms without requiring change of bound variables.

Object variables use the same syntax as Prolog atoms but are distinguished from atoms by declaration. The declaration

```
?- obvar_prefix([x,y]).
```

declares `x` and `y`, as well as `x` and `y` followed by numbers or underscores and numbers, as object variables. So, for example, `x0`, `y_1` are also object variables.

Quantifier symbols are declared using the same method as declaring operators. So, for example,

```
?- op(500, quant, q).
```

declares `q` to be a quantifier symbol with precedence `500`. Note, however, that this declaration does not give any semantics to the quantifer symbols (other than as an object variable binder) – the semantics are defined by the predicates of the program.

Assuming the declarations above, the following interaction with the interpreter shows Qu-Prolog unification in action.

```
| ?- x = y.
x = y
y = y

| ?- x = a.
no

| ?- q x f(x) = q y f(y).
x = x
y = y
```

```
| ?- q x A = q y B.
x = x
A = [x/y]B
y = y
B = B
provided:
x not_free_in [$/y]B

| ?- [A/x]B = 3.
A = A
x = x
B = B
provided:
[A/x]B = 3
```

The first example shows that object variables can be unified with each other. The second example shows that object variables don't unify with other terms. The third example shows that unification of quantified terms is up to alpha-equivalence – neither `x` nor `y` is instantiated by the unification.

The forth example extends the third example – to make the two terms alpha equivalent all free occurrences of `y` in `B` are replaced by `x`. The notation `[x/y]B` is the application of a substitution to `B` with this property. Note that, without more information about `B`, the substitution cannot be evaluated. Also note that the unification adds the constraint `x not_free_in [$/y]B` (where `$` is an atom). This constraint is also required in order to make the terms alpha-equivalent. If `x` and `y` represent different object variables then the constraint reduces to `x not_free_in B` – which says that since the left hand side of the unification has no free `x`'s then neither can the right hand side. On the other hand if `x` and `y` represent the same object variable then the constraint becomes true since there are no free `x`'s in `[$/x]B`. Also, in this case there are no free `x`'s on either side of the unification.

The final example shows a unification problem that delays, that is, becomes a constraint. This is because the unification problem has two solutions: `B = 3` and `B = x, A = 3`. Unification problems that have more than one solution or problems for which it is hard to prove there is only one solution, delay in the hope that some future computation will simplify the problem. The Qu-Prolog release comes with an example program, `incomplete_retry_delays` that attempts to find solutions to delayed unification problems. This program is used in the Ergo prover to eliminate such delays on request and is used in our example below to eliminate any remaining delayed unification problems.

Let us now look at the implementation in QuP^{++} of a reasoning agent whose inference engine is a tableau style prover for full first order predicate logic. The inference engine is given a list of sentences in first order logic and tries to find a contradiction – in other words it tries to show the collection of sentences is unsatisfiable. The inference engine is supplied with a resource bound that limits the number of inference steps.

We begin with a discussion of the inconsistency checker class (the inference engine) and later look at the reasoning agent class.

The inconsistency checker and the reasoning agent and its clients need to represent logical formulae as Qu-Prolog terms and this is aided with the following declarations.

```
?- obvar_prefix([x,y]).
?- op(860, quant, all).  % The universal quantifier
?- op(860, quant, ex).   % The existential quantifier
?- op(810, fx, ~).       % negation
?- op(820, xfy, and).    % conjunction
?- op(830, xfy, or).     % disjunction
?- op(840, xfy, =>).     % implication
?- op(850, xfy, <=>).    % equivalence
```

Following the declarations, the Qu-Prolog parser will then recognize the terms below (for example).

```
all x p(x)
[A/x]B
all x_1 ex x_2 (p(x_1) => q(x_2))
```

The first term represents the quantified term whose quantifier symbol is `all`, whose bound variable is `x` and whose body is `p(x)`. The seond term represents a substitution application where all free `x`'s in `B` are to be replaced by `A`.

The header for the inconsistency checker class is given below. The state variable `simplifier` is the address of a simplifier agent that the inconsistency checker uses to simplify the formulae.

```
class inconsistency_checker
state [simplifier]
inconsistent(Fs,R,RR):-
   find_contradiction(Fs,R,RR,not_simplified_yet).

% ... clauses for find_contradiction/4 and make_instances/5
} private [find_contradiction/4,make_instances/5]
```

In the only public method of this class, `inconsistent(Fs,R,RR)`, `Fs` is a list of formulae and `R` is a resource bound – the maximum number of inference steps allowed in trying to reduce `Fs` to an obviously inconsistent list of formulae. `RR` is the remaining sumber of inference steps after an inconsistency is found. The state variable `simplifier` holds the identity of a simplifier agent that can be used, at most once, to do auxilary simplification reductions.

```
find_contradiction(_,0,_,_):- !,fail. %  resource bound exceeded
find_contradiction(Fs,R,RR,STag) :-
    member(~true, Fs),!,
    RR is R-1.
```

```
find_contradiction(Fs,R,RR,STag) :-
    member(~(X=X), Fs),
    incomplete_retry_delays,
    !,
    RR is R-1.
find_contradiction(Fs,R,RR,STag) :-
    member(X, Fs),
    member(~X, Fs),
    incomplete_retry_delays,
    !,
    RR is R-1.
find_contradiction(Fs,R,RR,STag) :- %  Split conjunct.
    member_and_rest(A and B, Fs, Rst),
    !,
    NR is R-1,
    find_contradiction([A,B|Rst],NR,RR,STag).
find_contradiction(Fs,R,RR,STag) :- %  Remove an ex quantifier.
    member_and_rest(ex x A, Fs, Rst),
    x not_free_in Rst,
    !,
    NR is R-1,
    find_contradiction([A|Rst],NR,RR,STag).
find_contradiction(Fs,R,RR,STag) :- %  Branch on disjunct.
    member_and_rest(A or B, Fs, Rst),
    !,
    NR is R-1,
    find_contradiction([A|R],NR,IRR,STag),
    find_contradiction([B|R],IRR,RR,STag).
find_contradiction(Fs,R,RR,STag) :- %  Branch on implication.
    member_and_rest(A => B, Fs, Rst),
    !,
    NR is R-1,
    find_contradiction([~A|R],NR,IRR,STag),
    find_contradiction([B|R],IRR,RR,STag).
find_contradiction(Fs,R,RR,STag) :- % Do  univ. instantiations.
    make_instances(Fs, Fs, NewFs, R, NR),
    NR < R,       % made at least one univ. instantiation
    !,
    find_contradiction(NewFs,NR,RR,STag).

%  Call the simplifier - only if not been called before.
find_contradiction(Fs,R,RR,not_simplified_yet) :-
    NR is R-1,
    simplifier*=S,
    S?simplify(Fs,SFs),      % remote call to simplifier agent
    find_contradiction(SFs,NR,RR,simplified).
```

```
% Make instances of all universal and
% negated existential formulae.
make_instances([], New, New, R, R).
make_instances([H|T], Fs, NewFs, R, NR) :-
    ( H = all x A
      ->
        IFs = [[_/x]A|Fs],
        IR is R - 1
      ;
      H = ~ex x A
      ->
        IFs = [~[_/x]A|Fs],
        IR is R - 1
      ;
        IFs = Fs,
        IR = R
    ),
    make_instances(T, IFs, NewFs, IR, NR).
```

The private method `find_contradiction/4` attempts to reduce its `Fs` argument to a contradictary list and succeeds if it can do this within the resource bound of `R` steps. The last argument is a symbol flag that switches to `simplified` when the `simplifier` agent has been used in a particular inference, preventing another use. The third argument will return the final resource count when a contradiction is found. It is not of interest for a top level call, but it must be used when an inference splits into two sub-proofs to ensure that the second sub-proof uses only the resource left after the first sub-proof succeeds.

The first clause for `find_contradictition/4` causes the call to fail when the resource bound has been reduced to 0. The next three clauses deal with direct contradictions in its list of formulae first argument. The remainder deal with the logical operators and simplification. We only give representitive examples of this last group of clauses. The predicate `member_and_rest(E,L,R)` succeeds if `E` is somewhere on `L` and `R` is `L` with `E` removed.

The sixth clause eliminates existential quantifiers. The call to the built-in predicate `not_free_in/2` constrains `x` to be not-free-in `R` as required.

The universal instantiation rule makes an instance of each universal and negated existential formula and `adds` this to the list of formulae. For example, the formula `all x A` is instantiated to `A` with all free `x`'s in `A` replaced by a new meta-variable representing a yet-to-be-determined instance and this is added as a new formula. Since the universally quantified formulae remain, the rule can be re-applied any number of times providing there is at least one new formula added by its application. Repeated application of the rule to the same formulae is needed because sometimes a proof requires several different instantiations of a universally quantified formula. After each application we can expect that earlier rules will apply to the augmented list of formulae and these will be exhaustively

applied before it is re-used. The earlier rules always remove the formula to which they apply.

The universal instantiation rule is made to fail if no universal instantiation is found by the call to the auxiliary predicate `make_instances/5` to prevent repeated, pointless application to lists of formulae which contain no universally quantified formulae. In this case, when the universal instantiation rule is first called and fails, only the simplification rule can be used, as a last resort. After this has been used once, when all the earlier rules have been exhaustively applied and the universal instantiation rule is recalled and again fails, the entire proof fails.

The last clause sends a message to a simplifier agent that attempts to simplify the formula list according to its own simplification rules. The prover agent waits until the simplifier returns a simplified list. This clause demonstrates how one reasoning agent can take advantage of the skills of other reasoning agents in solving its problems. The simplifier might, for example, be a rewrite system for arithmetic subexpressions.

We now give an example of the inference engine in action by showing the sequence of transformations that `find_contradictition` would generate given a list of formulae.

```
(initial list)
[ ~ex x r(x), p(a) or ex x1 q(x1), all y1 ~q(y1),
                                   all z1 p(z1) => r(z1)]

(or rule on: p(a) or ..)
[ ~ex x r(x), p(a), all y1 ~q(y1), all z1 p(z1) => r(z1)],

[~ex x r(x), ex x1 q(x1), all y1 ~q(y1), all z1 p(z1) => r(z1)]

(univ. instant. rule on: ~ ex x .., all y1 .., all z1 ..
          of first list)
[~r(X1), p(a), ~q(Y1), p(Z1) => r(Z1), ~ex x r(x),
                  all y1 ~q(y1), all z1 p(z1) => r(z1)],

[~ex x r(x), ex x1 q(x1), all y1 ~q(y1), all z1 p(z1) => r(z1)]

(implies rule on: p(Z1)=>r(Z1) of first list)
[~r(X1), p(a), ~q(Y1), ~p(Z1), ...],

[~r(X1), ~q(Y1), r(Z1), ...],

[~ex x r(x), ex x1 q(x1), all y1 ~q(y1), all z1 p(z1) => r(z1)]

(contradiction rule appied to: p(a),~p(Z1) of first list
                       and to: ~r(X1),r(Z1) of second list)
[~ex x r(x), ex x1 q(x1), all y1 ~q(y1), all z1 p(z1) => r(z1)]
```

```
(ex rule applied to: ex x1 q(x1))
[~ex x r(x), q(x2), all y1 ~q(y1), all z1 p(z1) => r(z1)]

(univ. instant. rule applied to:
        ~ex x r(x), all y1 ~q(y1), all z1 p(z1) => r(z1))
[~r(X2), ~q(Y2), p(Z2) => r(Z2), ~ex x r(x), q(x2), ...]

(contradiction rule applied to: ~q(Y2),q(x2))
success
```

When the `ex rule` is applied the new object variable (which comes from the rule instance) is set to be not free in all the other formulae in the list.

Note that we can use `find_contradiction` to attempt answer extraction during the proof. If, for example, we have the formula `~r(X)`, instead of the formula `~ex x r(x)` in the list of formulas at the start of the above contradiction derivation, a contradiction will also be found generating the binding `X=a`. In fact, if the formulae in the knowledge base are essentially horn clauses and the 'query' formula is of the right form then `find_contradiction` behaves as a Prolog goal evaluator.

However, answer extraction is not always possible. If we take `~ex y r(y)` as the query formula and if the knowledge base consists of the formula `ex x r(x)` or the formula `r(a) or r(b)` then `find_contradiction` will succeed. If, however, the query formula is `~r(X)` then a contradiction cannot be found. In the first case, the use of the rule for existential quantification causes a not-free-in condition to be generated that prevents `X` from being instantiated to `x`. In the second case, two different instantiations are required during the proof.

We now turn our attention to an example of a reasoning agent class. This is the class definition for a reasoning agent. Each reasoning agent object contains a knowledge base of `believes` facts that can be initialised when the agent is created and added to whilst it is alive. Clients of the reasoning agent can use the `ask` method to see if the agent believes the supplied formula. The agent believes the formula if it is in the knowledge base or can be deduced from the knowledge base within the supplied inference step resource bound.

```
class reasoner isa inconsistency_checker
state [believes/1,  told/1, mentor/1]
clauses{
init :- object_thread_fork(_,absorb_told_info).

absorb_told_info:-
   thread_wait_on_goal(self_retract(told(F)),
   findall(S, believes(S), Fs),
   ( inconsistent([F|Fs],200,_) ->
       true
    ;
```

```
        self_assert(believes(F))
   ),
   absorb_told_info.
tell(B) :-
  caller(M),
  mentor(M),
  self_assertz(told(B)).
ask(F,_) :-
    believes(F),
    !,
    caller(Cl),
    Cl^^proved(F).
ask(F,R):-
    nonvar(F),
    integer(R),
    R>0,
    caller(Cl),
    object_thread_fork(_,try_to_prove(F, R, Cl)).
try_to_prove(F, R, Cl) :-
    findall(S, believes(S), Fs),
    ( inconsistent([~F|Fs],R,RR) ->
        Cl^^proved(F,RR)
      ;
        Cl^^not_proved(F,RR)
    ).
}
private [try_to_prove/2, absorb_told_info/0, inconsistent/2,
         told/1].
```

As an example use of this program, suppose we execute:

```
new(reasoner,[{believes(p(a) or ex x1 q(x1)). ..},..],Ag)
```

where the agent is given the formulas:

```
 p(a) or ex x1 q(x1), all y1 ~q(y1), all z1 p(z1) => r(z1)
```

as its initial beliefs. If some other agent `Cl` then sends the query:

```
Ag^^ask(r(X),100)
```

`Ag` will spawn a contradiction sub-proof trying to reduce:

```
[ ~r(X), p(a) or ex x1 q(x1), all y1 ~q(y1),
                                   all z1 p(z1) => r(z1)]
```

to a contradiction. Since this will succeed, the reply:

```
Cl^^proved(r(a))
```

will be sent to the client agent.

The `mentor/1` dynamic predicate is used to tell the agent which other agents are allowed to give it new information by calling its `tell` method. Notice that the method does not immediately add a `believes/1` fact. Instead a `told/1` fact is asserted and it is the responsibility of the `absorb_told_info` 'daemon', that runs as a separate thread launched by the `init` method, to check if the told sentence `F` is inconsistent with the sentences already in the knowledge base. If it can prove inconsistency within a resource limit of 200 inference steps then the told sentence is ignored. Otherwise the told sentence is added to the knowledge base. This is potentially dangerous since it could produce a knowledge base with 'deep' contradictions, but it is pragmatic. That the agent will not accept `tell/1` calls except from its mentors is another safeguard.

The meta-call predicate `thread_wait_on_goal`, used in the `reasoner` class definition, causes the thread to suspend until the goal which is its argument succeeds. That is, the argument goal is tried. If it succeeds, the meta-call succeeds and no further solutions of the argument goal are sought on back-tracking. If it fails, the thread executing the meta-call suspends until there is some update to the dynamic clause data base, or the record date base. The argument call is then retried. This try, fail, retry, continues indefinitely until the argument goal succeeds. In this case it will cause the `absorb_told_info` object thread to suspend until some `told(F)` fact is asserted by the interface thread. The thread deletes the asserted fact and asserts a `believes(F)` fact if `F` cannot be shown to be inconsistent with the agent's current beliefs within 200 inference steps. If it can be shown to be inconsistent with the current beliefs no belief fact is asserted. The `absorb_told_info` thread then recurses to handle the next asserted `told/1` fact.

This is one simple example of a reasoning agent. Another possibility is to define a cooperative reasoning agent that can be used to implement a distributed knowledge base. The system would contain a collection of agents, each with their own local knowledge base, that would cooperate to produce proofs based on the combined knowledge of the group. Each agent could have meta knowledge about which other agents 'know about' particular predicates and hence can be asked to prove or disprove predications (or their negations) containing these predicates.

To achieve this we can define a sub-class `coop_reasoner` of the `reasoner` class. This is given below.

It has an extra dynamic predicate:

```
has_proved_false(L,Ag,RR)
```

which is used by the agent to record answers to `isfalse/2` queries it has sent out to other agents. It also has extra methods for accepting asynchronous calls `isfalse(L,R)`, that cause the agent to try to contradict `L` within `R` inference steps, and for accepting asynchronous `proved_false(L,RR)` replies to such calls that it has sent to other agents. Here `RR` is the number of inference steps left from the resource `R` given in the `isfalse/2` request.

The three new clauses for `find_contradiction/4` add a new way for terminating a contradiction proof. When a literal `L` is found in the current list of

formulas with a predicate P, and the agent believes that some other agent Ag knows about P, providing the complement literal to L is not in the currentl list, Ag is sent an asynchronous isfalse(L,RforAg) call. The proof then continues with asked(L,A) replacing L in the list of formulas. (For this reason we need the second new clause for find_contradiction/4 that terminates a proof when a literal is found for which there is an asked/2 formula mentioning its complement.) RforAg is a number of inference steps that Ag should use in trying to contradict L. It is got by dividing up the remaining inference steps in a manner dependent upon L. We leave this undefined. A suitable default definition would just halve the remaining inference steps, no matter what L is. Notice that when a sub-contracted proof is achieved inside the given resource bound, signalled by the eventual self asserting of a has_proved_false(Ag,L,RR) dynamic clause by the concurrently executing interface thread as a result of a proved_false(L,RR) call, the unused inference steps RR of the sub-contracted proof are added to the still unused inference steps of the main proof to give a more accurate value for the unused the inference steps of the main proof.

The agent's interface thread will concurrently be responding to queries from other agents, including any proved_false(L) reply sent back from Ag. The interface thread will respond to this by self asserting a has_proved_false(L,Ag). These dynamic facts are handled by the second new clause. This second clause looks for asked(L,Ag) reminders left in the current list of formulas. For each such reminder it checks to see if has_proved_false(L,Ag) holds, i.e. if such a fact has been asserted by the concurrently executing interface thread. If any such replies have been received to the sub-contracted proofs, the main contradiction proof immediately terminates.

```
class coop_reasoner isa reasoner
state [has_proved_false/3]
clauses {
find_contradiction(Fs,R,RR,STag) :-
    member_and_rest(L, Fs, Rst),
    literal(L),
    predicate_of(L,P),           % perhaps should sub-contract L
    believes(knows_about(P,Ag)), % to Ag but should not if
    complement(L,CompL),         % Fs contains complement of L
    \+ member(CompL,Rst),        % or a note that Ag has been
    \+ member(asked(CompL,Ag)),  % asked about its complement
    divide_up(L,R,RforAg,NR),
    !,
    isfalse(L,RforAg)^^Ag,
    find_contradiction([asked(L,Ag)|Rst],NR,RR,STag).
find_contradiction(Fs,R,RR,STag) :-
    member(L, Fs),
    literal(L),
    complement(L,CompL),        % find complement to L
    member(asked(CompL,_), Fs), % equiv. to having CompL
```

```
    incomplete_retry_delays,
    !,
    RR is R-1.
find_contradiction(Fs,R,CRR,STag) :-
    member_and_rest(asked(L,A),R),
    has_proved_false(L,A,RR),  % reply has come from A about L
    incomplete_retry_delays,
    CRR is R + RR,
    !.
find_contradiction(Fs,R,RR,STag):-
    super?find_contradiction(Fs,R,RR,STag).

proved_false(L,RR):-
   caller(Ag),
   self_assert(has_proved_false(L,Ag,RR).

isfalse(L,R):-
   caller(Ag),
   findall(S,believes(S),Fs),
   object_thread_fork(_,try_to_contradict(L, R, Ag)).

try_to_contradict(F, R, Ag) :-
    findall(S, believes(S), Fs),
    inconsistent([F|Fs],R) -> proved_false(L)^^Ag ; true.
} private [has_proved_false].
```

6 Related Work

With repect to its OO features the design of QuP^{++} has been much influenced by L&O [16] and DK-Parlog++ [9]. L&O is an OO extension for a single threaded Prolog and the objects are not active. However, QuP^{++} borrows its inheritance semantics from L&O. DK-Parlog++ is an OO extension of a distributed hybrid of Parlog[6] and the multi-threaded IC-Prolog II[10]. DK-Parlog++ classes have both procedural methods (Parlog clauses) and knowledge methods (Prolog clauses). Object state, as in QuP^{++}, is represented by both state variables and dynamic clauses. QuP^{++} methods are the equivalent of the DK-Parlog++ knowledge methods. However, DK-Parlog++ has only single inheritance and does not have built in support for multi-threaded objects where all the threads can access and update the object's state with atomic operations. It is also restricted to a local area network, whereas QuP^{++} objects can be distributed over the internet.

DLP [11] is perhaps the closest distributed OO LP language to QuP^{++}. DLP has classes with multi-inheritance and class instances run as separate threads. Object state can only be recorded as state variables, not as clauses. Method invocation is a remote synchronous call. The default is that such a call spawns

a query sub-thread in the target object. This is similar to the `O??Q` remote call of QuP^{++} that we have not discussed in this paper. For a query `O??Q` the different solutions are returned by `O` to the caller `Cl`, one at a time, as required by backtracking within `Cl`. This is distributed backtracking and its QuP^{++} implementation is sketched in [7]. For a `O?Q` call all its solutions are returned to `Cl` in a list with local backtracking within `Cl`. DLP does not have the equivalent of the `?` and `^^` remote calls. In addition, it appears objects can only be single threaded. An object can have the equivalent of an `init` method but this cannot spawn sub-threads, it can only spawn new objects that have a separate state. Because of this the DLP `init` method must periodically expiclitly interrupt its pro-active execution to accept remote calls. One cannot have QuP^{++} style multi-threaded objects, with one thread accepting remote calls whilst the other threads concurrently engage in their own specific activities interacting, if need be via the shared object state. In addition, neither DLP and DK-Parlog++ have reflective methods such as `class/1` and `mystate/1` and consequently do not allow easy programming of mobile agents. Both are also OO extensions of normal Prolog, with no special support for writing inference systems.

CIAO Prolog is a rich Prolog systems that also has multi-threading[4], with inter-thread communication via atomic updates of the dynamic data base, and a module system which has been used to implement an OO extenssion O'CIAO[5]. O'CIAO supports multiple inheritance between classes with class methods being static clauses and object state being represented as dynamic clauses. Dynamic clauses for the different object instances are distinguished in the same way as in QuP^{++} by adding the object identity as an extra argument to the predicate they define. The objects of O'CIAO are passive objects, the instances do not run as separate threads, however CIAO itself has active modules which can also have state, repesented as dynamic clauses local to the module. These active modules can be given global identities that can be stored in files and client modules can make use of the active module by referencing this file and declaring which predicates it is using from amongst those that are exported by the module. These exported predicates are then called in the normal way within the client module, but the implementation will do a remote call to the active module. The concept of an active module/class could be added to O'CIAO to give it active objects. Also, the multi-threading of CIAO could be used to allow multi-threaded objects sharing the same dynamic clause object state, but this integration of all the features of CIAO has apparently not yet been done. CIAO Prolog also has constraint handling but has no built in support for programming non-clausal theorem provers.

Mozart-Oz[19] is a multi-paradigm distributed symbolic programming language with support for logic programming, functional programming and constraint handling. It is being used for distributed agent applications[22]. It also has passive objects, essentially records of functions which can access and update state local to the record. Mozart-Oz is multi-threaded with the threads sharing a common store of values and constraints. The store is used for inter-thread communication. Constraints are posted to the store and the store can be queried

as to whether some particular constraint is entailed by the current constraint store. A thread executing such a query will suspend until the store entails the constraint. This is a generalisation of our use of `thread_wait_on_goal/1` in QuP^{++}.

In Mozart-Oz any data value, including on object or an unbound variable of the constraint store, can be shared across different Mozart-Oz processes by creating a ticket for the data value using a special primitive. The ticket is an ASCII string and is similar to the global identity of an active object in QuP^{++}, which is a term constructed from three symbols. This ticket string can then be used in another Mozart-Oz process to access the value associated with the ticket, even if it is held in a non-local store, by calling another ticket value access primitive.

Active objects can be programmed by using a Mozart-Oz port which can be sent a message from any thread that has access to the port, perhaps via a ticket. A port is rather like an object's message queue in QuP^{++}. Another thread then accesses the messages sent to the port as elements of an incrementally generated list, calling an appropriate method of some local passive object for each accessed message. Such a port/thread/object combination behaves very like a QuP^{++} active object, but the calling of the object's methods and the sending of replies has to be achieved in Mozart-Oz using explict asynchronous message sends to ports and explicit accesses of messages from the port message streams. That is, what we have referred to as the interface thread has to be explictly programmed as a wrapper for an object to make it active. This is how the remote calls of QuP^{++} are implemented, using the inter-thread communication primitives of Qu-Prolog[7], but QuP^{++} presents to a programmer the higher level abstraction of synchronous and asynchronous remote calls directly to an object's methods.

Gaea[20] is a multi-threaded OO Prolog system with active objects which have dynamic methods and modifiable inheritance trees. Gaea is not a class based OO system. Instead each active object, which in Gaea is just a thread with an associated `cell` of clauses, executes in an environment of a list of parent cells for its cell. These parent cells have the role of super-classes, but the list of parent cells can be dynmically constructed as the object is created. Each of these parent cells can itself have an associated list of parent cells. So an object executes in an tree structured environment of ancestor cells rooted at its cell. This is similar to a QuP^{++} object executing in tree structured environment of the static methods of its super classes (the parent hierarchy of Gaea cells) with its own state component of dynamic clauses and state variables (the root cell directly linked with the Gaea object/thread). The difference is that in Gaea, the inheritance structure is created dynamically, as the active object is forked, *and* it can be modified whilst the object is executing. Any parent cell of a cell can be removed and new ones can be added. So the entire inheritance hierarchy for an object is dynamic. These modifications to the inheritance structure can be made by the object itself, or by another object executing in the same Gaea process.

Cells can contain cell variables as well as clauses. The cell variables are similar to the state variables of a QuP^{++} object. The cell clauses can be updated using

special cell assert and retract primitives, similar to the self assert and retract of QuP^{++}, as can the cell variables. Objects communicate via the clauses and cell variables of the cells they both have access to. In addition, a call can be evaluated relative to a named cell. When this is the root cell linked with an object, this is equivalent to a call to the methods of that object, even though the call is executed in the caller, rather than the callee. Clearly this is only possible when the different objects execute in the same Gaea process, for only then will each have access to the cell clauses of the other objects. Gaea is not a distributed system.

The ability to modify the inheritance structure of an object is a much more dymamic way of changing an object's behaviour than the `become/2` primitive of QuP^{++}. However, the flexibility may come at a cost of program transparency. Gaea has no special support for writing theorem provers.

λProlog, see for example [2], is a logic programming language with built-in support for λ-terms and consequently can be used as an implementation language for theorem provers in much the same way as is done in Qu-Prolog. λProlog does not, however, appear to provide as much support as Qu-Prolog does for implementing interactive theorem provers, nor does it appear to have support for multiple threads or even high-level communication bewteen different λProlog processes.

In this paper we have shown how simple multi-threaded agents can readily be implemented in QuP^{++}. Since our main concern was illustrating the features of the language we have not developed any complex agent architectures. However, it would be no great effort to implement logic based agent architectures such as those described in [3], [21], [23]. Implementing more complex architectures, with both sophisticated reasoning and reactive capabilities, is the subject of our on-going research.

Bob Kowalski wrote a short paper in 1985 [13] which anticipated many of the ideas now being discussed with respect to logic based agents. In particular, the paper discusses the need for information assimilation by resource bounded reasoning agents, interacting with one another and the world. Our co-operative reasoning agents are a partial realisation of the ideas expressed in that paper. His ideas have since been elaborated in [14] and [15] to allow interleaving of action and reasoning within an agent, in order to reconcile the need for rationality and reactivity. The agent architectures sketched in these more recent papers could also easily be implemented in QuP^{++}.

References

1. Holger Becht, Anthony Bloesch, Ray Nickson and Mark Utting, Ergo 4.1 Reference Manual, Technical Report No. 96-31, Software Verification Research Centre, Department of Computer Science, University of Queensland, 1996.
2. C. Belleannée, P. Brisset, O. Ridoux, A pragmatic reconstruction of λProlog, *Journal of Logic Programming*, 41(1), 1999, pp 67-102

3. M. Bozzano, G. Delzanno, M. Mattelli, V. Mascardi, F. Zini, Logic Programming and Multi-Agent Systems: A synergic combination for applications and semantics, in *The Logic Programming Paradigm*, (eds K. Apt et al), Springer-Verlag, 1999.
4. M. Carro and M. Hermenegildo, Concurrency in Prolog Using Threads and a Shared Database. *Proceedings of ICLP99*, (ed. D. De Schreye), MIT Press, 1999, pp 320-334.
5. A. Pineda and M. Hermenegildo, O'Ciao: An Object Oriented Programming Model for (CIAO) Prolog, Research Report CLIP 5/99.0, (accessible from http://www.clip.dia.fi.upm.es/), Facultad de Informatica, UPM, Madrid, 1999.
6. K. L. Clark, S. Gregory, Parlog: Parallel Programming in Logic, *ACM Toplas* 8(1), 1-49 pp, 1986.
7. Keith Clark, Peter Robinson and Richard Hagen. Multi-threading and Message Communication in Qu-Prolog *Theory and Practice of Logic Programming*, **1**(3), 2001, pp 283-301.
8. Keith Clark, Peter J. Robinson and Richard Hagen, Programming Internet Based DAI Applications in Qu-Prolog, *Multi-agent systems*, (eds. C. Zhang, D. Lukose), Springer-Verlag, LNAI 1544, 1998.
9. K.L. Clark, T.I. Wang, Distributed Object Oriented Logic Programming, *Proceedings of FGCS94 Workshop on Co-operating Heterogeneous Information Systems*, Tokyo, 1994.
10. D. Chu, K. L. Clark, IC-Prolog II: A multi-threaded Prolog system *Proceedings of ICLP93 Post Conf. Workshop on Concurrent, Distributed and Parallel implementations of Logic Programming Systems*, 1993
11. A. Eliens, *DLP, A language for distributed logic programming* Wiley, 1992
12. Richard Hagen and Peter J. Robinson. Qu-Prolog 4.3 User Guide. Technical Report No. 97-12, Software Verification Research Centre, University of Queensland, 1999.
13. R. A. Kowalski, Logic Based Open Systems, *Representation and Reasoning*, Jakob ph. Hoepelmann (Hg.) Max Niemeyer Verlag, Tubingen, 1985, pp125-134.
14. R. Kowalski and F. Sadri, Towards a unified agent architecture that combines rationality with reactivity, *Proc. International Workshop on Logic in Databases*, Springer-Verlag, LNCS 1154, 1996.
15. R. A. Kowalski and F. Sadri, From Logic Programming to Multi-Agent Systems, *Annals of Mathematics and Artificial Intelligence* 25, 1999, pp391-419.
16. F.G. McCabe, *Logic and Objects* Prentice-Hall, 1992.
17. F.G. McCabe, The Inter-Agent Communication Model (ICM), http://www.nar.fla.com/icm/, Fujitsu Laboratories of America Inc, 2000.
18. F. G. McCabe and K. L. Clark. April:Agent Process Interaction Language. *Intelligent Agents*, (ed. N. Jennings, M. Wooldridge), Springer-Verlag LNCS 890, 1995.
19. Mozart-Oz Homepage: www.mozart-oz.org
20. I. Noda, H. Nakashima, K. Handa, Programming language GAEA and its application for multi-agent systems, *Proc. of Workshop on Multi-agent systems and Logic programming*, In conjunction with ICLP'99, 1999.
21. A. Roa, AgentSpeak(L): BDI Agents speak out in a logical computable language, *Agents Breaking Away*, (eds. W. van de Velde and J. W. Perram), Springer-Verlag LNCS 1038, 1996.
22. Peter Van Roy and Seif Haridi, Mozart: A Programming System for Agent Applications, *International Workshop on Distributed and Internet Programming with Logic and Constraint Languages*, 1999. Accessible from http://www.mozart-oz.org/papers/
23. G. Wagner, Artificial Agents and Logic Programming, in *Proc. of ICLP'97 Post Conference Workshop on Logic Programming and Multi-Agent Systems*, 1997.