
Annals of Mathematics and Artificial Intelligence **41:** 171–206, 2004.

Go! – A multi-paradigm programming language for implementing multi-threaded agents

K.L. Clark [a] and F.G. McCabe [b]

[a] *Department of Computing, Imperial College, London, UK*
E-mail: klc@doc.ic.ac.uk
[b] *Fujistu Labs of America, Sunnyvale, CA, USA*
E-mail: fgm@fla.fujitsu.com

`Go!` is a multi-paradigm programming language that is oriented to the needs of programming secure, production quality, agent based applications. It is multi-threaded, strongly typed and higher order (in the functional programming sense). It has relation, function and action procedure definitions. Threads execute action procedures, calling functions and querying relations as need be. Threads in different agents communicate and coordinate using asynchronous messages. Threads within the same agent can also use shared dynamic relations acting as Linda-style tuple stores. In this paper we introduce the essential features of `Go!`. We then illustrate them by programming a simple multi-agent application comprising hybrid reactive/deliberative agents interacting in a simulated ballroom. The dancer agents negotiate to enter into joint commitments to dance a particular dance (e.g., polka) they both desire. When the dance is announced, they dance together. The agents' reactive and deliberative components are concurrently executing threads which communicate and coordinate using belief, desire and intention memory stores. We believe such a multi-threaded agent architecture represents a powerful and natural style of agent implementation, for which `Go!` is well suited.

Keywords: agent programming, multi-paradigm programming, threads

1. Introduction

`Go!` is a logic programming descendant of the multi-threaded symbolic programming language `April` [25], with influences from IC-Prolog II [6] and L&O [23]. Crudely `Go!` is `April` enriched with the knowledge representation features of `Prolog` and L&O.

`April` was initially developed as the implementation language for the much higher level MAI^{2}L [17] agent programming of the EU Imagine project. It has more recently been used to implement one of the FIPA compliant agent platforms of the EU AgentCities project [38], and the agent services running on that platform at Imperial College and Fujitsu. `Go!` was used to implement a query server for knowledge and queries expressed in the `SL1` subset of the FIPA query content language SL [13], for that platform.

A significant theme in the design of Go! is software engineering in the service of high-integrity systems. To bring the benefits of logic programming to applications developers requires fitting the language into current best-practice; and, especially since applications are increasingly operating in the Internet, security, transparency and integrity are critical to the adoption of logic programming technology.

Although Go! has many features in common with Prolog, particularly multi-threaded Prolog's, there are significant differences related to transparency of code and security. Features of Prolog that mitigate against transparency, such as the infamous *cut* (!) primitive, are absent from Go!. Instead, its main uses are supported by higher level programming constructs, such as single solution calls, *iff* rules, and the ability to define 'functional' relations as functions.

In Prolog, the same clause syntax is used both for defining relations, with a declarative semantics, and for defining procedures, say that read and write to files, which really only have an operational semantics. In Go!, behaviours are described using action rules which have a different syntax.

While Prolog is a *meta-order* language, Go! is higher-order as in higher order functional programming. Go! is not higher order in the higher order logic sense. There is no unification of function or relation definitions, as there is in lambda-Prolog [26]. Code can only be passed as an opaque value with a specific type.

A key feature of Go! is the ability to group a set of definitions into a lexical unit by surrounding them with {} braces. We call such a unit a *theta environment*. Theta environments are Go!'s program structuring mechanism. Two key uses of theta environments are *where* expressions, analogous to the *let* ... *in* ... construct of some functional programming languages, and labeled theories, which are labeled theta environments.

Labeled theories are based on McCabe's *L&O* [23] extension of Prolog. A labeled theory is theta environment labeled by a term where variables of the label term are global variables of the theory. Instances of the theory are created by giving values to these label variables. Labeled theories are analogous to class definitions, and their instances are Go!'s objects. New labeled theories can be defined in terms of one or more existing theories using inheritance rules. Labeled theories provide a rich knowledge representation notation akin to that of frame systems [28].

Go! is strongly typed – using a modified Hindley/Milner style type inference technique [18], [27] to reduce the programmer's burden. The programmer can declare new types and thereby introduce new data constructors. Indeed, all uses of functors as data constructors need to be declared by being introduced in some type definition. Compile time type checking improves code safety.

Go! is also multi-threaded, a feature which we consider essential for building sophisticated agents. Threads primarily communicate through asynchronous message passing as in Erlang [1], April [25], Qu-Prolog [8]. Threads, executing action rules, react to received messages using pattern matching and pattern based message reaction rules. A communications daemon enables threads in different Go!

processes to communicate transparently over a network. Typically, each agent will comprise several threads, each of which can directly communicate with threads in other agents.

Threads in the same agent can also communicate by manipulating shared cell or dynamic relation objects. Updates of these objects are atomic. Moreover, threads can be made to suspend until a term unifying with some given term is added to a shared dynamic relation by *some other thread*. Thus dynamic relations can be used like Linda tuple stores [4] to coordinate the activities of different threads within an agent. This is a powerful implementation abstraction for building multi-threaded agents, which we will illustrate with our example application of Go!.

Go! does not directly support any specific agent architecture or agent programming methodology, although this could be done using library modules. It is a language in which different architectures and methodologies can be quickly prototyped and explored. We illustrate this by developing a simple multi-agent application comprising hybrid reactive/deliberative agents interacting at a simulated ball. In this ballroom there are male and female dancers. The dancers have desires that reflect – somewhat – a typical ballroom situation with the different desires corresponding to individual preferences. Dancers negotiate with each other over up-coming dances and also make counter proposals – such as going to the bar instead of having a dance.

Each dancer agent is programmed using multiple concurrently executing threads that implement different aspects of its behaviour. The threads communicate and coordinate using shared belief, desire and intention dynamic relations. This internal run-time architecture allows us to use *implicit* interleaving of the various activities of the agent. This contrasts with the *explicit* interleaving of observation, short deliberation and partial execution of the classic single threaded BDI (*B*eliefs, *D*esires, *I*ntentions) architecture [2].

Linda tuple stores have been used for inter-agent coordination [29]. For scalability and other reasons, we prefer to use asynchronous point-to-point messages between agents, as in KQML [11]. However, we strongly advocate concurrency and Linda style shared memory co-ordination for internal agent design.

In section 2 we give an overview of Go!. In section 3 we introduce dynamic relations. In section 4 we give the program for a simple directory server that can be used for agent registration and discovery. This is used in our ballroom application to enable a dancer to discover other dancers with similar desires. In section 5 we develop the ballroom application. In section 6 we discuss related work before giving our concluding remarks in section 7.

2. Key features of Go!

Go! is a multi-paradigm language with a declarative subset of function and relation definitions, an imperative subset comprising action procedure definitions and rich program structuring mechanisms.

2.1. Function, relation and action rules

Function rules. Functions are defined using sequences of conditional rewrite rules of the form:

$$f(A_1,..,A_k)::Test \Rightarrow Exp$$

The guard *Test* is optional. The `::` can be read as *such that*.

As in most functional programming languages, the testing of whether a function rule can be used to evaluate a function call uses *matching* not unification. A rule is used if the head matches the call and any *Test* succeeds. Once a function rule has been selected there is no backtracking to try an alternative rule. The sequence of rules defining a function should cover *all the cases*. It is an error if no rule can be used to evaluate a function call. Example function definitions are:

```
nrev()=>[].
nrev([E,..X]) => nrev(X)<>R.

age(DateOfBirth) => yearsBetween(now(),DateOfBirth).

number_of_maths_courses(S)=>
  listlen({C || takes(S,C),subject(C,'maths')}).
```

The operator `,..` is Go!'s list constructor, equivalent to the LISP `cons` and the Prolog `|`. It can be read as *followed by*. `listlen` is Go!'s primitive for finding the length of a list, <> is a library function for appending lists, and an expression of the form:

$$\{T \mid\mid Cond\}$$

is a set expression. It is Go!'s equivalent to the Prolog `findall`. `now` is a system function for returning the Unix time from which the current date can be computed. `yearsBetween` is another defined function.

Notice that `'maths'` is singly quoted. Go! does not use the Prolog convention that all alphanumeric names beginning with a lower case letter are symbols and all those beginning with an upper case letter are variables. Symbols have to be explicitly quoted and alphanumeric names beginning with a lower case letter can be used as variable names.

if relation definitions. These comprise sequences of Prolog-style `:-` (*if*) clauses; with some modifications – such as permitting expressions as well as data terms, and no cut. An example is:

```
only_sons(P:person) :- (P.child(C:person)*>C.sex='male')
```

This defines a property that holds of a person (object) if all their children (objects) have sex attribute `'male'`. The first occurrences of `P` and `C` are both type annotated as instances of the `person` class. Object attributes are accessed using `'.'`, as in `C.sex`.

*> is Go!'s *forall*. A condition:

```
(Cond1 *> Cond2).
```

holds if for every solution to *Cond1*, there exists a solution to *Cond2*.

Instead of the cut, Go! has negation-as-failure (\+)[1], single solution calls, and a conditional test as primitives.

iff relation definitions. In addition, relations can be defined using sequences of :-- (*iff*) rules. These have the syntax:

$$p(A_1,..,A_k)::Test \text{ :-- } Condition$$

where *Test* is optional, but not *Condition*. Once an *iff* rule has been found that unifies with a relation call, such that any associated *Test* succeeds, there is a commitment to use just that rule. There is no backtracking on the choice of the rule, just possible backtracking in the evaluation of the body conditions of the rule when some of these are defined using Prolog style :- rules. *iff* rules allow relations to be defined using disjoint cases, each case being represented as a different head pattern/test of a rule. An example of a definition using *iff* rules is:

```
ordered([]) :-- true.
ordered([_]):-- true.
ordered([X,Y,..L])::X=<Y :-- ordered([Y,..L]).
```

This is the definition of a relation for testing if a list of numbers is ordered.

Action rules. The locus of action in Go! is a *thread*; each Go! thread executes a procedure. Procedures are defined using non-declarative *action* rules of the form:

$$a(A_1,..,A_k)::Test \text{ -> } Action_1;...;Action_n$$

As with function rules, the first action rule that matches some call, and whose test is satisfied, is used. Once an action rule has been selected there is no backtracking on the choice of rule *and* there is no backtracking within the action sequence of the body of the rule. Every action call must succeed, it is a run-time error if it does not.

An example action rule is:

```
runMe(Count) -> stdout.outLine(nrev(iota(1,Count))^0)
```

This defines a procedure runMe with number argument. The body of the rule is the single action:

```
stdout.outLine(nrev(iota(1,Count))^0)
```

This uses the Go! primitive function iota to construct a list of numbers from 1 to Count, uses nrev to reverse this list, and uses the standard operator ^ which will

[1] This is the normal Prolog operator for negation.

convert any Go! data value into a string. This string, followed by a newline, is written to the standard ouput channel (usually the terminal window from which the program was invoked) using the procedure outLine of the stdout file object.

More generally, the permissible actions of an action rule include: message dispatch and receipt, I/O, updating of dynamic relations and cells, the calling of a procedure, and the spawning of any action, or sequence of actions, to create a new thread.

2.2. *Invoking queries from actions*

The declarative part of a Go! program can be accessed from action rules in a number of ways:

- Any expression can invoke functions.
- The head of an action rule – a(*A*$_1$,..,*A*$_k$)::*Q* – can extend the argument matching with a query *Q*.
- If *Q* is a query, {*Q*}, indicating a single solution call to *Q*, can be used as an 'action'.
- We can use a set expression {*Trm* || *Q*} to find all solutions to some query.
- We can use Go!'s *forall* action. (*Q* *> *A*) executes the action *A* for each solution to query *Q*. Typically, *Q* and *A* share variables.
- We can use a conditional action. (*Q* ? *A*$_1$ | *A*$_2$) executes *A*$_1$ if *Q* succeeds, else *A*$_2$. *Q* and *A1* can share variables.

Invoking actions from queries. Occasionally it is necessary to execute an action inside a query or relation definition although this style of programming is discouraged in Go!. If it is done, then the action being called must be prefixed by the action operator.

2.3. *Programming behaviour with action rules*

As an example of the use of action rules let us consider programming the action thread of an agent with a mission: this is to achieve some fixed goal by the repeated execution of an appropriate action given the current situation. The procedure:

```
performMission()::Goal -> {}.
performMission() ->
    {choose_action(ActProc)};
    ActProc();
    performMission().
```

captures the essence of this goal directed activity. ({} is the empty action.) It assumes that a concurrently executing thread is monitoring the agent's environment, constantly updating the agent's beliefs about its state. These beliefs are queried by *Goal*, and by the choose_action relation which selects an action procedure to ex-

ecute based on current beliefs. Note this is a higher order relation. `performMission` is a tail recursive procedure and will be executed as an iteration by the `Go!` engine.

2.4. Message communication

Threads in a single `Go!` invocation can communicate either by thread-to-thread message communication or by synchronisable access and update of shared data, such as dynamic relations. Threads in different `Go!` invocations can only communicate using messages. To support thread-to-thread communication, each thread has its own buffer of messages it has not yet read, which are ordered in the buffer by time of arrival.

Sending messages. The non-blocking message send action is:

```
Exp >> Rec
```

where *Exp* is an expression denoting the message to be sent and *Rec* is a `handle`-valued expression. `handles` are the type used by `Go!` for the identifiers of threads. Any `Go!` data value can be sent as a message, including higher order values and data terms containing variables. Any variables in a message are replaced by entirely new variables when it is received. Variables are not shared across threads.

A handle is a term of the form:

```
hdl(Th,Rt)
```

The second argument *Rt* is a symbol name of a family of threads, usually all the threads in a given `Go!` invocation. We will refer to this as the *root* name. The first argument *Th* is a symbol that identifies the thread within the family. When a `Go!` program is started we can set the both the root and thread names of the initial thread with command line arguments.

`Go!`'s message send is asynchronous. Successful dispatch means neither that the recipient thread has 'accepted' the message nor that it actually received it. Message acceptance is an active process on the part of the receiver. Also, the destination task may be no longer executing. Every thread has a single associated message queue into which all sent messages are placed in time order of *arrival*.

Multi-casting. The message send primitive sends to just one other thread, there is no primitive multicast or broadcast facility in `Go!`. However, a message can be sent to all the threads whose handles are in some list *Handles* using `Go!`'s *forall* operator:

```
(H in Handles*> Exp  >> H)
```

`in` is `Go!`'s list membership primitive. More generally, we can use *>> to send a message *M* to thread *H* for each *M*, *H* satisfying some arbitrary query condition *Cond(M,H)*:

```
(Cond(M,H)*> M  >> H)
```

For example:

```
(current_sub_task(Task),cannot_do(Task),cando(Ag,Task)*>
  ('will_you_do',Task) >> Ag)
```

Receiving messages. A thread can search for and remove a message from its message queue using a simple << message receive action, or by a executing a choice message receive action. A << action has the form:

MsgPtn << *Sender*

It removes from the message queue the first message that matches *MsgPtn* which comes from the thread identified by *Sender*. Typically *Sender* is an unbound variable which will be bound to the handle of the sender. If not, it serves as a check on the identity of the sender, and only messages from that sender will be accepted by this receive action. For example:

MsgPtn << hdl(ThNm,*RootNm*)

will only accept a message matching *MsgPtn* from a thread within the collection of threads with root name *RootNm*. Variable ThNm will be bound to its thread name.

If there is no message in the thread's message queue that matches *Msg*, and is from the specified sender thread, then the << call suspends until such a message is received. The message receive is a match operation, not a unification, because no variable in the message term can be bound by the message receive. Any attempt to bind a variable in a message results in the match failure, and the next message in the queue is tested for a match. Once a message has been accepted, variables can be bound in the message term.

Note that it is often useful to use a test pattern, with an associated condition. For example, the message receive action:

...;N::N<10 << *S*;...

looks for a number message from thread *S* whose value is less than 10. Numeric messages that are greater than 10, or messages from threads whose handles do not unify with *S*, will *not* be picked up by this message receive.

Choice message receive. Go! also has a more powerful mechanism for receiving messages: the *choice message receive*. This allows a thread to search for one of several different forms of message in a single sweep of its message queue and to execute a different action depending on which form of message it finds.

The recursive semaphore procedure:

```
semaphore(F) ->
  ( 'req'::F>0 << H -> 'ok'>>H; semaphore(F-1)
  | 'free' << H -> semaphore(F+1)
  ).
```

uses a choice message receive comprising a disjunction of two *message rules* of the

form:

```
MsgPtn::Test << Sender -> Action
```

where *Test* is optional. Each (recursive) execution of the choice receive results in a new search of the thread's message buffer looking for either a `'req'` message, or a `'free'` message. If a `'req'` from a some sender `H` is found first – and the value of `F` is greater than zero – the message is removed from the queue and the procedure sends an `'ok'` acknowledgement back to the sender of the message. The `semaphore` procedure then recurses with a decremented value for `F`.

Otherwise, if a `'free'` message is found first, or if `F` is not currently positive, causing all `'req'` messages to be skipped over until a `'free'` is found, the `'free'` message is removed and we enter a tail recursive call with an incremented value of `F`.

More generally, a choice message receive action operates by examining each of the messages in the message queue; applying each of the message patterns of its disjunction of message rules to each message in turn. As soon as a message is found that satisfies a message pattern of some rule *R*, the message is removed from the queue and the action sequence of rule *R* is executed – none of the other actions in the choice message receive will be executed. Execution of the choice message receive will suspend if there is currently no message satisfying any of the alternative message patterns. It is automatically resumed each time a new message arrives.

A choice message receive normally exits only when a message is received that can be accepted by one of its message rules. However, a timeout can be associated with the choice receive as:

```
(Choice message receive) timeout (T -> default_action)
```

After finding no acceptable message in its buffer, the thread executing this will wait for time *T* for an acceptable message to arrive. At that time it will quit the message receive and execute *default_action*.

2.5. *Spawning threads*

Any `Go!` thread can spawn new threads. The evaluation of all the threads is time shared by the `Go!` run-time system. Each spawned thread also has its own message buffer and handle identity.

As an example:

```
sem = spawn{semaphore(3)}
```

launches a new thread executing the `semaphore` procedure to control three resources. The identity of the thread, its *handle*, is returned by the `spawn` primitive and assigned to variable `sem`. The spawning thread, and any other thread given access to the handle `sem`, can send messages to the new thread using:

```
...; 'req' >> sem; 'ok' << sem; ...
```

The `'ok'` message receive action will suspend until a reply from `sem` is received.

As an alternative to holding the handle of a spawned thread in a variable, the programmer can assign a handle to the thread when it is launched, thus giving a 'public' name to the thread which can be used throughout the program.

```
spawn {semaphore(3)} as hdl('listener','semaphore');
...;
'req' >> hdl('listener','semaphore');
```

Such an assigned handle can even be used to send messages to the semaphore thread from threads in other `Go!` processes, see section 4.1.

2.6. Theta environments

In many ways, theta environments form the 'heart' of `Go!` programs: they are where most programs are actually defined; they are also the only place where new types may be defined. The scope of a type definition is the theta environment in which it appears.

A theta environment is of a sequence of statements grouped inside {} brackets, each of which is either:

- a *Var* = *Expression* assignment,
- a *Type* ::= *TypeEpression* new type definition,
- a *Type* :> *TypeEpression* renaming type definition,
- a relation rule,
- a function rule,
- an action rule,
- a DCG grammar [30] rule,
- a labeled theta environment – a class definition (see section 2.7),
- a class rule – defining an inheritance relation (see section 2.7).

The statements are separated by the '.␣' operator[2].

where expressions. A common use of a theta environment is a *`where`* expression, which is an expression of the form:

`Exp..ThetaEnvironment`

The `..` is read as *where*. *`Exp`* is evaluated relative to the definitions inside *`ThetaEnvironment`* which otherwise are local the environment.

where calls. As well as expressions, calls can be evaluated relative to a theta environment. The call, whether relation or action call, is written:

`Call..ThetaEnvironment`

[2] Where '.␣' means a period followed at least one whitespace character.

2.7. Classes and objects

Classes in Go! are labeled theta environments[3], which we can view as labeled theories as in L&O [23]. The labels can contain variables, which are global to all the definitions of the theory.

Class definitions also double as type definitions – the functor of the class label is impliclty defined as a new type name. It can be used to declare the type of variables that will be bound to instances of the class.

If *L* is a class label, `$L'`, where *L'* is an instance of *L* in which any variables are given values, denotes an instance of the class. The instance is an object characterised by these values – they define its *static state*.

Two system classes, the dynamic relations class and the cell class, have instances with mutable state. A labeled theory can contain variables bound to instances of these mutable state classes. If so, instances of the theory will be objects with *mutable state*.

The following labeled theory is a mini-theory of a person.

```
person(Nm:symbol,Age:number,Sx:symbol,Hm:string){
  include "sys:go/cell.gof".
  __Age=$cell[number](Age).
  age() => __Age.get().
  birthday() -> __Age.set(__Age.get()+1).
  sex=Sx.
  name=Nm.
  lives(Hm)
}.
```

The person's age is recorded in a variable `__Age` bound to a mutable cell object that is private to the theory. It is private since its name begins with a double underscore. `__Age` holds a `cell[number]` object with initial value `Age`. The `__Age` cell is updated by the `birthday` action method of the theory. An object of type `cell[T]` is an instance of the polymorphic `cell` class exported from the included system module. It can be used to hold an updateable value of type `T`. When an instance of the class is created, this type has to be specified.

`person(Nm,Age,Sx,Hm)` is the theory label and `Nm`, `Age`, `Sx`, `Hm` are the theory parameters characterizing each instance – the *theory* of a particular person. The class definition implicitly introduces a new object type `person`. We type annotate the class parameters since, except for `Age`, the method definitions do not constrain their types and we do not need the class definition to be polymorphic with respect to the types of `Nm`, `Sx`, `Hm`. We need not have typed `Age`.

[3] We shall use the terms *labeled theory* and *class* interchangeably.

We can create two instances of the theory, i.e. two person objects, and query them as follows:

```
P1=$person('Bill',23,'male',"London,England").
P2=$person('Jane',20,'female',"Cardiff,Wales").
P1.name           -- returns name 'Bill' of P1
P2.age()          -- returns age 20 of P2
P1.birthday()     -- adds one to age of P1
P1.age()          -- returns 24 as current age of P1
P2.lives(Place)   -- gives solution: Place="Cardiff,Wales"
```

Inheritance. The following is a labeled theory for a student. It inherits from the person theory.

```
student(Nm,Age,Sx,Hm,_,_)<=person(Nm,Age,Sx,Hm).
student(_,_,_,_,Cge:string,Sbj:string){
  lives(Pl) :- location_of(Cge,Pl).
  lives(Pl) :- person.lives(Pl).
  studies_at(Sbj,Cge)}.
```

The separate <= rule says that this theory inherits from the `person` theory. Inheritance is overriding. Any relation, function or action procedure defined in a super class is replaced by any new definition in the inheriting class. In this case, there is a relation, `lives`, which is so redefined. However, the second clause for this relation explicitly invokes the over-ridden definition in the person class, which means that in this case the new definition actually extends the old definition. Note that the student theory makes use of a global relation `location_of` defined outside the theory. It has a normal definition such as:

```
location_of("Imperial","London,England").
location_of("Caltec","Pasadena,CA").
...
```

We can create a specific student *theory* and query it as follows:

```
S=$student('mary',19,'female',"Bath,England","Imperial",
                        "computing")
S.lives(Place) -- has two answers:
          -- Place="Bath,England",Place="London,England"
S.age()        -- returns 19
```

Analogously to `student`, we can define an `employee` class, and then a new class of `employed_student` that inherits from both of these:

```
employee(Nm,Age,Sx,Hm, _)<=person(Nm, Age,Sx,Hm).
employee(_,_,_,_,WorkPl:string,Role:symbol){
  employed_as(WorkPl,Role)}.
employed_student(Nm,Age,Sx,Hm, Cge,Sbj,_,_)<=
     student(Nm,Age,Sx,Hm,Cge,Sbj).
employed_student(Nm,Age,Sx,Hm, _,_,WorkPl,Role)<=
     employee(Nm,Age,Sx,Hm, WorkPl,Role).
```

Note that `employed_student` is defined solely by two class rules. We can query an instance of the class using any of the methods and attributes of `person`, `student` and `employee`. Those from the shared `person` super class of `student` and `employee` are inherited only once.

The use of `Go!` classes for knowledge representation is more fully explored in [7].

2.8. Modules

The following is a module that exports the single function `sort`. The definitions of the auxiliary relations `srt` and `partition` are local to the theta environment and not visible outside. `<>` is exported from the included system `stdlib` module.

```
sort ..  {
  include "sys:go/stdlib.gof".
  sort(list,lesseq) => srt(list) ..  {
    srt([]) => [].
    srt([E,..X]) ::  partition(X,E,A,B) =>
      srt(A)<>[E,..srt(B)].
  partition([],_,[],[]):-- true.
  partition([E,..X],Piv,[E,..A],B)::lessq(E,Piv):--
     partition(X,Piv,A,B).
  partition([E,..X],Piv,A,[E,..B])::\+lesseq(E,Piv)[4]:--
     partition(X,Piv,A,B).
  }}
```

The `sort` function is *parameterized* with respect to the ordering relation used to compare elements of the list. It is defined in terms of the auxilliary definitions for `srt` and `partition`, themselves introduced in a subsiduary *where* expression. This illustrates how *where* expressions may be used at many levels – not just the top-level of a program. Note that the `lesseq` variable – which holds the ordering relation – is only mentioned

[4] Given the operations semantics of *iff* rules, this `\+` test could be dropped.

where it is important: where it is introduced as a parameter of `sort` and where it is used in `partition`. This is an example of variables having a somewhat extended scope compared to `Prolog`. In `Prolog`, to achieve the same effect, we would have had to 'pass down' the `lesseq` relation through all the intermediate programs from the top-level to where it is needed.

Executable programs. An executable program is a module that exports a single action procedure value, this is the procedure that will be called when the program is run. The execution of the procedure will be the initial thread of the program.

2.9. Higher order values

A call to `sort` must supply the list to be sorted and the `lesseq` relation. In many cases this relation is given as the value of a variable with a higher order value[5]; however, it is also possible to use a lambda rule, or a disjunction of such rules, to give an on-the-fly definition of the relation. For example, in the call:

```
sort([Some list of number lists],
      ( (L,L):--true
      | (L1,L2)::listlen(L1) =< listlen(L2):--true
      | (L1,L2)::listlen(L1) = listlen(L2):--
           less_at(L1,L2)..{
             less_at([N1,..._],[N2,..._])::N1<N2:--true.
             less_at([N,..L1],[N,..L2]):--
                          less_at(L1,L2)}
      ))
```

the second argument is a disjunction of lambda relation rules that uses the standard number ordering relation to define an ordering for lists of numbers. The first rule says that a pair of identical lists are in the relation. The second includes each pair where the first list is shorter than the second. The third covers the case of non-identical lists of the same length. It includes a pair of lists `L1`,`L2` if they are identical up to some elements `N1` in `L1`, `N2` in `L2`, such that `N1<N2`. It uses an auxiliary relation `less_at` defined in a *where* call.

`Go!` has lambda forms of all of its rule types: relation rules, function rules, action rules and grammar rules.

[5] Note the contrast with `Prolog`. In `Prolog` a relation is passed as argument by passing in its name – which is an atom. `Prolog`'s meta-level `call` is then used to map the name to the value at run-time by accessing a run-time dictionary linking atom names with code values. In `Go!` the code value is passed, not the name.

2.10. Types

Go! is a strongly typed language; using a form of Hindley/Milner's type inference system [27]. For the most part it is not necessary for programers to associate types with variables or other expressions. However, all constructors and *unquoted* symbols are required to be introduced using type definitions. If an identifier is used as a function symbol in an expression it is assumed to refer to an 'evaluable' function unless it has been introduced in a type definition as a data constructor.

The type of a non-polymorphic relation, such as a relation whose extension is a set of pairs, each pair comprising a `symbol` and a lists of `number`s, is denoted by the type expression:

```
(symbol,list[number]){}
```

Here, *`list[number]`* is the Go! type expression for list of numbers, and the postfix `{}` signals a set of pairs, i.e. a binary relation.

In Go! the type expressions for *polymorphic* relation, function and procedure types contain type variables instead of specific type names, and they are quantified with respect to these type variables. Type quantification is denoted by the *varlist-type* form, where all the variables in *varlist* are implicitly universally quantified.

The type of `sort` is

```
[t]-((list[t],(t,t){})=>list[t])
```

i.e., it is a polymorphic function – it can sort lists of any type. `sort` takes a list and a relation as arguments and returns a list of the same type.[6]

Any values. Heterogeneous types of values can be included in a list of type `list[any]`. The list:

```
[??1,??'a',??"hello"]
```

is such a list. The `??` prefix operator maps *any* data value into the Go! standard type `any`.

The type of a value `V` in a `??V` term is not visible outside the term, but its type is remembered as a hidden second argument of the term. In that an `any` value hides an encapsulated type it is similar to what is called an existential type [3].

When unifying two `??` terms, the hidden types are also compared – at run-time if necessary. Thus the unification of two `??` terms succeeds only if the hidden types are identical *and* the encapsulated values unify.

As an example, suppose `A` is a variable of type `any` holding some `??` term. The unification:

```
A=??(W:list[number])
```

[6] Note that while `sort` is polymorphic, its argument relation is not. This must be a relation over the values on its argument list. If `sort` required a polymorphic relation in its second argument, its type would be: *`[t]-((list[t],[s]-(s,s){})=>list[t])`*.

where `W` is an unbound variable, will succeed if and only if `A` encapsulates a list of numbers. Thereafter, `W`, with type `list[number]`, can be used in any context where a list of numbers is required, and we have effectively extracted the list of numbers hidden inside `A`.

Messages sent as `any` values. Messages are passed between threads as `any` values. The primitive message send and receive actions of `Go!` are both of type `(any,handle)*`, the `*` indicating an action type. They are denoted, respectively, by the operators `?>>` and `<<?`. A message send `Msg >> H` is actually shorthand for `??Msg ?>> H` and is expanded to this by the compiler. Similarly, `MsgPtn << H` is shorthand for `??(MsgPtn:T) <<? H` where `T` is the inferred type of `MsgPtn`.

Programmer defined types. The pair of type definitions:

```
dance::= polka | jive | waltz | tango | quickstep | samba.
Desire::= toDance(dance,number) | barWhen(dance).
```

introduce two new types. An enumerated type `dance`, which has 6 literal values:

```
polka, jive, waltz, tango, quickstep, samba
```

and a `Desire` type that has two constructor functions `toDance` and `barWhen`. Each symbol literal and constructor name can only be used in a single type definition within the same theta environment.

These types will be used in our ballroom application. The `Desire` type defines the terms that will be stored in the dynamic relation representing the agents desires. The compiler will check that only terms such as:

```
toDance(polka, 3), barWhen(samba)
```

are used to represent an agent's desires.

Object types and object interfaces. As mentioned earlier, a class definition implicitly defines a new type – the type of the instances of the class. `Go!` can usually infer the type of a variable from its use, however, since methods and attributes are not unique to a class, it cannot infer the object type of a variable from method invocation use.

For example, the procedure definition:

```
display_person_details(P) ->
  stdout.outLine("Name is:  "<>P.name^0);
  stdout.outLine("Age is:  "<>P.age()^0);
  ...
```

will generate a compiler error since the object type of `P` cannot be determined. To make it into a procedure for displaying the key attribute values of any `person` object we need

to use:

```
display_person_details(P:person) -> ...
```

To make it a procedure for displaying the details of any person object, or any object 'declared' to include all the attributes and methods of a person, i.e. to implement the `person` interface, we use:

```
display_person_details(P<~person) -> ...
```

Automatically, all subclasses of the `person` class implement its interface so this is now a procedure for displaying the `person` level details of `students`, `workers`, etc.

3. Dynamic relations

In `Prolog` we can use `assert` and `retract` to change the definition of a dynamic relation whilst a program is executing. In `Go!`, a dynamic relation is an object with updateable state. It is an object of type `dynamic[T]`, `T` being the type of the argument of the dynamic relation. All `Go!` dynamic relations are unary, but the unary argument can be a tuple of terms.

A dynamic relation object has methods: `add(Trm)`, for adding `Trm` to the end of the current extension of the relation, `del(TrmPtn)` for removing the first term in the relation that unifies with `TrmPtn`, `delall` for removing all unifying terms, `mem(TrmPtn)`, for accessing each current term that unifies with `TrmPtn`, and finally `ext()` for retrieving the current extension as a list of terms.

Creating a new dynamic relation. A *dynamic relation* object can be created and initialised as in:

```
desire = $dynamic[Desire]([toDance(jive,2),barWhen(polka),
                           ...,toDance(waltz,1)])
```

The above initialisation is equivalent to giving the following sequence of clauses for a Prolog dynamic relation:

```
desire(toDance(jive,2)).
desire(toDance(waltz,1)).
...
desire(barWhen(polka)).
```

Querying a dynamic relation. If we want to query such a dynamic relation we use the `mem` relation method as in:

```
desire.mem(todance(D,N)),N>2
```

Modifying a dynamic relation. To modify a dynamic relation we can use the `add`, and `del` action methods. For example:

```
desire.del(toDance(jive,N));desire.add(toDance(jive,N-1))
```

This is analogous to the following sequence of `Prolog` calls:

```
retract(desire(toDance(jive,N)),NewN is N-1,
assert(desire(toDance(jive,NewN)))
```

One difference is that we cannot backtrack on a `del` call to delete further matching facts. This is because it is an action, and all `Go!` actions are deterministic. A `del` call always succeeds, even if there is no matching term. The `delall` method deletes all unifying terms as a single action:

```
desire.delall(barWhen(_))
```

will delete all current `barWhen` desires.

Dynamic relations with rules. To allow run-time manipulation of dynamic relations defined by rules we can store a pair of values – a term representing the head of the rule, and a relation lambda rule of type `(){}`, which is its body. Variables in the head term can appear as global variables in the body of the lambda. As an example:

```
Likes=$dynamic[((symbol,symbol), (){})](
                          [(('peter','mary'), ():-true)),
                           ((X,Y), ():-parent_of(X,Y)))]).
```

initialises `Likes` to a dynamic relation over pairs of symbols defined by two 'clauses'. The first tells us that `('peter','mary')` is in the relation, the second that any pair `(X,Y)` is in the relation providing a call to:

```
():-parent_of(X,Y)
```

succeeds – i.e. providing `parent_of(X,Y)` is true.

We can add to the relation using:

```
Likes.add(((X,Y),():-same_family(X,Y),\+ hates(X,Y))))
```

and query it as in:

```
{Y || Likes.mem((('peter',Y),Body)),Body()}.
```

This will find all the `Y`'s whom `'peter'` likes given suitable definitions of the other three relations. To make querying easier, we can define:

```
likes(X,Y) :- Likes.mem(((X,Y),Body)),Body().
```

and use:

```
{Y || likes('peter',Y)}.
```

3.1. Multi-threaded applications and data sharing

It is often the case, in a multi-threaded `Go!` application, that we want the different threads to be able to share information. For example, in a multi-threaded agent we often want all the threads to be able to access and perhaps update the beliefs of the agent.

We can represent the relations for which we will have changing information as dynamic relations. A *linda* dynamic relation is an instance of a subclass of the dynamic relations class with extra methods to facilitate the sharing of dynamic relations across threads. It has a `replace` method, allowing the deleting and adding of a term to be executed atomically, and, as well as the inherited `mem`, is has a `memw` relation method. A call:

```
LinRel.memw(Trm)
```

will suspend if no term unifying with `Trm` is currently contained in `LinRel`. It will resume as soon as such a term is added by *another thread*. There is also a dual, `notw`.

```
LinRel.notw(Trm)
```

will suspend if a term unifying with `Trm` is currently contained in `LinRel`. It will resume when all such terms are deleted by *other threads*. There is also a suspending delete method, `delw`. If need be, it will wait until some unfying term is added before immediately deleting it. `delw` calls are queued.

`memw` and `delw` and the analogues of the Linda [4] `readw` and `inw` methods for manipulating a shared tuple store. There is no analogue of `notw` in Linda.

4. A directory server example

Agent applications often make use of a directory server or match-maker which enables agents to discover one another using descriptions. Our directory server given below stores descriptions as arbitrary length lists of terms of type `dscrTrm`, which is an application specific data type.

The `DSmessage` type defines the type of the message protocol of the directory server. This is a common use of new type. The `register`, `subscribe` and `inform` message constructors each take an argument of type `descrip`, which is a list of `dscrTrm` terms.

The `description` dynamic relation holds the lists of `dscrTrm` terms contained in each received registration message. `subcription` holds pairs – handle of a subscriber, and a list of `dscrTrm` terms.

```
directory_server..{
  include "sys:go/dynamic.gof".
  ...-- type defs defining dscrTrm
  descrip :> list[dscrTrm].
  DSmessage ::= register(descrip) |
                subscribe(descrip) |
                inform(descrip).
description=$dynamic[descrip]([]).
subscription=$dynamic[(handle, descrip)]([]).

unifying_part_descr(PrtDsrc, Dscr) :-
  (Term in PrtDsrc*> Term in Dscr).

inform_if_unify(PrtDsrc,Dscr,S) ->
  (unifying_part_descr(PrtDsrc,Dscr) ?
     inform(PrtDscr) >> S
     | {}).
directory_server() ->

  ( register(RegDescr) >> _ ->
     description.add(RegDescr);
     (subscription.mem((S,PrtDscr))*>
        inform_if_unify(PrtDscr,RegDscr,S)
  | subscribe(PrtDsrc) >> S ->
     subscription.add((S,PrtDsrc));
     (description.mem(RegDescr)*>
        inform_if_unify(PrtDsrc,RegDscr,S))
  );
  directory_server()
}
```

Using the directory server. The directory server is designed to be executed in a separate thread, perhaps in a separate `Go!` process, and agents communicate with it using message send and message receive actions.

Given the type definition for the descriptive terms that will be used, such as:

```
Sex::= male | female.
dscrTrm::= name(symbol) | sex(Sex) | ...
```

an agent can register with the directory with a registration message, such as:

```
register([name('sally'),sex(female),...]) >> DS
```

This will record a list of `dscrTrm` terms in the directory server `DS`.

An agent can also send a `subscribe` message such as:

```
subscribe([sex(female),name(_),...]) >> DS
```

which contains a partial description, including terms with unbound variables.

Such a message is understood as a request for a stream of `inform` messages to be sent as response, all of which contain an instantiation of the partial description. For example:

```
inform([sex(female),name('mary'),...])
```

The instantiation is done by the call to `unifying_part_descr`.

The directory server module can be imported into another program, and its exported procedure spawned as a directory server to be used by all the other threads spawned in the program. It can also be executed as a separate stand-alone `Go!` process as described in the next section. `Go!` agents executing in separate processes, even on other hosts, can use it to advertise their attributes and to discover one another.

This directory server is by no means complete; a more elaborate directory server would also accept de-registration messages and un-subscribe messages. If intended for a public environment it would also require a technique for authenticating clients. However, this program is sufficient for the dancer agents in our 'ballroom' scenario.

4.1. Distributed use

Full discussion of how a `Go!` application can be distributed over a network is beyond the scope of this paper and is described in [24]. In brief, to distribute a `Go!` application we make use of a separately launched message router, which is an April program, a `Go!` system module that communicates with the router using TCP/IP, and the ability we have to launch a `Go!` program with command line options that determine the handle of its initial thread.

Using this last feature, we can launch the directory server program so that its initial and only thread has a public handle, such as:

```
hdl('receiver','directory')
```

In addition, we slightly modify its program by importing the router communications module and by making its initially executed procedure one that executes the `directory_server` procedure indirectly, by being passed as argument to a higher order procedure `scsconnect`. `scsconnect` is exported from the communications module. It has another argument which is the identity of the host on which the router is running. It opens a connection with the router, registers the assigned handle with it, and then executes `directory_server` in a special mode that allows external thread communication via the router.

Any messages sent by the directory server to external threads now get automatically sent to the router for forwarding. Conversely, any messages with destination

```
hdl('receiver','directory')
```

sent by threads in other Go! processes, similarly connected to the router, get sent to the router for forwarding to the directory server.

5. Multi-threaded dancer agents at a ball

In our agents' ball, we have male and female dancer agents, a directory server, and a band that 'plays' music for different kinds of dances, with intervals between each dance. The directory server and the band 'are' the ball. The dancers 'arive' in some random phased order. They use the directory server to discover the identities and dance desires of dancers of opposite gender, and negotiate with one another to do a dance, or go to the bar. This scenario is a compact use case that demonstrates many of the aspects of building intelligent agents and of coordinating their activities.

Following the BDI model [2,32], each agent has a belief, a desire and an intention store each implemented as a linda dynamic relation. The belief store contains beliefs about the desires of other dancers, what dance tune is currently being played, if any, and what intentions have been successfully executed. The desire store contains the currently unfulfilled desires. The intention relation holds current intentions. An intention is a commitment to a joint activity with another dancer the next time the band announces a particular dance, such as a polka. It is either to dance, or to go to the bar.

5.1. The male dancer agent

Below we give the overall structure of the male dancer class. Each instance of the class will have its own state represented by the three linda dynamic relations.

A call to the start method is spawned as a new thread immediately after an instance is created. To create a new male dancer we execute:

```
spawn {($maleDancer(Name,Desires)).start()}
```

start adds each desire of the given *Desires* parameter to the dancer's desire relation. It then spawns three threads: one to execute intentions when the triggering dance announcement occurs, one to negotiate with other dancers to generate new joint intentions, one to interface with the directory server.

```
maleDancer(MyNm,MyDesires){
  Belief::= hasDesires(symbol, list[Desire]) |
            haveDanced(dance,symbol) |
            bandPlaying(dance) |
            ballOver.
  dance::= polka | jive | waltz | ...
  Desire::= toDance(dance,number) | barWhen(dance).
  Intention ::= toDanceWith(dance,symbol) |
             toBarWith(dance,symbol).
```

```
Sex::= male | female.

belief=$linda[Belief]([]).
believe(B):-belief.mem(B).
desire=$linda[Desire]([]).
want(D):-desire.mem(D).
intention=$linda[Intention]([]).
intend(I):-intention.mem(I).

maleIntention=maleIntention..{
  -- inner module to be given}.

DSinterface= DSinterface..{
  -- inner module to be given}.

satisfyDesires=satisfyDesires..{
 -- inner module to be given}.

start() ->
  (Des on MyDesires *> desire.add(Des));
  spawn {maleIntention()} as hdl('execTh',MyNm);
  spawn {satisfyDesires()} as hdl('negTh',MyNm);
  spawn {DSinterface()};
  waitfor(hdl('execTh',MyNm)).
}
```

The first two are assigned handle identities with root name the name of the dancer and standard thread names. This facilitates the inter-thread communication between the dancers. The threads within a dancer communicate with each other using the shared linda dynamic relations: `belief`, `desire` and `intention` accessed as global variables of the inner modules defining their procedures. After spawning the three threads the `start` procedure waits for the intention execution thread to terminate, which it does when the band announces the ball is over. At this point the `start` procedure will terminate, automatically terminating the spawned negotiation and directory interface threads.

All three sub-threads execute concurrently. The data flow between the threads is depicted in the architecture figure 1. Each update and access to one of the shared stores is atomic, and a thread may suspend waiting for an item to be added to a store by another thread.

Note that while all the dancers can be executed in a single invocation of the `Go!` engine, they will *not* have direct access to each others' beliefs, desires and intentions since each instance of the class has its own copies of the three linda relations. Furthermore, it is a simple task to distribute the progam across multiple invocations and hosts, making each dancer a separate `Go!` process.

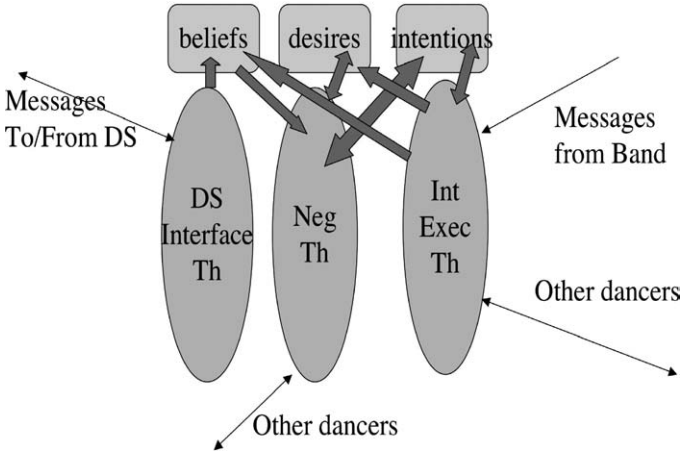

Figure 1. Agent architecture.

5.2. *A dancer's directory interface thread*

Each dancer registers with the directory its name, gender, and its initial desires. It then subscribes for details of all other registrations of opposite gender.

```
DSinterface..{
  DS=hdl('main','dir')[7].

  dscrTrm::=name(symbol)|sex(Sex)|desires(list[Desire]).
  DSmessage ::= ...-- as in directory server.

   DSinterface() ->
     register([name(MyNm),sex(male),desires(MyDesires)])
                    >> DS;
     subscribe([sex(female),name(_),desires(_)])
                    >> DS;
     acceptDSinforms().

  acceptDSinforms() ->
    inform([_,name(Nm),desires(Dsrs)]) << DS;
    belief.add(hasDesires(Nm,Dsrs));
    acceptDSinforms().
}
```

The `acceptDSinforms` procedure – which is the continuation of the interface thread after the initial registration – iterates, processing one `inform` message on each iteration. Initially these will be messages giving details of female dancers already at the

[7] We assume that the directory server has been spawned with this handle, perhaps as a separate process.

dance. Thereafter, they will be messages sent when new females arrive and register. The procedure waits for each of these messages, processing it as soon as the message arrives. The added belief about the new dancer can then be used by the negotiation thread.

5.3. A male dancer's intention execution thread

A dancer's intention execution thread handles the execution of intentions when they are triggered by dance announcements. We assume a band agent which sends an announcement message to every currently registered dancer when it starts, and when it stops playing each dance 'number'.

```
maleIntention= maleIntention..{
  bandMessage::=starting(dance) | stopping |
                ball_over.
  danceMessage::=shallWeDance(dance) | myPleasure(dance).
  Band=hdl('main','band')[8].
  maleIntention() ->
    ( starting(D) << Band ->
      belief.add(bandPlaying(D));
      check_intents(D);
      maleIntention()
    | stopping << Band ->
       belief.del(bandPlaying(_));
       maleIntention()
    | ball_over << Band -> belief.add(ballOver)
    ).
  check_intents(D) ->
    ( intend(toDanceWith(D,FNm)) ?
        intention.del(toDanceWith(D,FNm));
        maleDance(D,FNm)
    | (intend(toBarWith(D,...)) ? ...
    ).
  ...-- definition of maleDance }
```

The `maleIntention` action procedure is a recursive loop that listens for messages from the band that signal: the starting of a new dance, the stopping of the current dance, and that the ball is over. A `ball_over` message terminates the loop. It also maintains an appropriate belief recording when and what the band is playing, and when the ball is over. These beliefs are accessed by the negotiation thread.

When it receives a `starting(D)` message, and there is an intention to do dance `D`, the `maleDance` procedure is executed. The intended partner will similarly have called its corresponding `femaleDance` procedure and the interaction between the

[8] As with the directory server, we assume the band thread is separately launched and given this handle.

two intention threads, respectively executing their dance procedures, is the joint dancing activity. The dance procedures terminate when the `stopping` message is received.

Note that the `maleDance` procedure below increments the dancer's desires to dance `D` if the `stopping` message is received before the `myPleasure` reply from female `FNm`; and he 'forgets' about `FNm` – the belief about her desires is deleted as are any other intentions involving her. Getting no reply is interpreted to mean she has 'left' the ball. A late arriving reply from `FNm` will now just be ignored. It increases its desires to do `D` because they will have been decreased by 1 when the commitment to dance was made.

```
maleDance(D,FNm)->
 shallWeDance(D) >> hdl('execTh',FNm);
 ( myPleasure(D) << hdl('execTh',FNm) ->
     stopping << _; -- 'dance' till end of D
     belief.add(haveDanced(D,FNm))
 | stopping << _ -> -- FNm has probably left the ball
     forget_about(FNm);
     {want(toDance(D,N))};
     desire.replace(toDance(D,N),toDance(D,N+1))
 );
 belief.del(bandPlaying(D)).
```

5.4. A male dancer's negotiation thread

The procedures executed by the negotiation threads of our dancers are the most complex. They represent the rational and pro-active activity of the agents for they convert desires into intentions taking into account current beliefs, desires and intentions. In contrast, the intention execution and directory interface threads are essentially reactive.

A male dancer's negotiation thread must decide which uncommitted desire to try to convert into an intention and which female to invite to participate. This may result in negotiation over which dance they will do together, for the female who is invited may no longer want to do that dance, may prefer to do another, or may already be committed to doing it. (We allow a dancer to enter into at most one joint commitment to do a particular type of dance since this is understood as a commitment to do the dance the *next* time *that* dance is announced.)

The overall negotiation procedure is `satisfyDesires`:

```
satisfyDesires() ->
  {belief.notw(bandPlaying(_))};
  (chooseDesire(Des,FNm),still_ok_to_negotiate() *>
                    negotiateOver(Des,FNm));
  {belief.memw(bandPlaying(_))};
   satisfyDesires().
still_ok_to_negotiate():-
 \+(believe(bandPlaying(_)) | believe(ballOver)).
```

The first action of `satisfyDesires` is the `notw` call. This is a query action to the `belief` relation that will suspend, if need be, until any `bandPlaying(_)` belief is removed by the intention execution thread that is 'listening' to the band. As an 'etiquette', our dancer agents can start a negotiation only if the band is not playing.

As soon as that belief query succeeds, there is an attempt to convert into commitments as many unsatisfied desires as possible during the dance interval. This is the `*>` forall loop of the procedure. Each negotiation is with a named female `FNm` whom the male dancer believes shares that desire. Before starting each negotiation the negotiation thread checks that it is still 'ok' to negotiate – it checks that it does not believe the band has started playing, or announced that the ball is over. When the negotiation loop terminates, in case this is before the band has restarted, the thread checks if that there is a belief that the band is playing and waits if not. (It does this to ensure there is only one round of negotiations in any dance interval.) The thread's procedure then recurses for a new round of negotiation when the band next stops playing providing the ball is not over. If so, it suspends.

At the next dance interval, the answers returned by `chooseDesire` will almost certainly be different because the beliefs, desires and intentions of the dancer will have changed and new female dancers may have arrived. Even if one answer is the same, a re-negotiation may have a different outcome because of changes in the female's mental state.

```
chooseDesire(toDance(D,N),FNm) :-
  uncmtdFeasibleDesire(toDance(D,N),FNm),
  (want(toDance(OthrD,OthrN)),OthrD\=D *> OthrN<N).
chooseDesire(toDance(D,N),FNm) :-
   uncmtdFeasibleDesire(toDance(D,N),FNm),
   \+ believe(haveDanced(D,_)).
chooseDesire(toDance(D,N),FNm) :-
  uncmtdFeasibleDesire(toDance(D,N),FNm),
  \+ believe(haveDanced(D,FNm)).
chooseDesire(toDance(D,N),FNm) :-
  uncmtdFeasibleDesire(toDance(D,N),FNm).
uncmtdFeasibleDesire(toDance(D,N),FNm) :-
  uncmtdDesire(toDance(D,N)),
  believe(hasDesires(FNm,FDesires)),
  toDance(D,_) in FDesires.
uncmtdDesire(toDance(D,N)):-
  want(toDance(D,N)), N>0,
  \+ intend(toDanceWith(D,_)).
uncmtdDesire(barWhen(D)):-
  want(barWhen(D)),
  \+ intend(toBarWith(D,_)).
```

The above clauses for `chooseDesire` are tried in order, which reflects priorities.

The clauses only select a dance desire. The minimal requirement for choosing a desire `toDance(D,N)` and a female `FNm` is: `N>0`, there is no current commitment to do the dance `D`, and `FNm` is believed to want to do `D`.

The first rule selects a dance if, additionally, the dancer at that point desires to do the dance more times than any other dance. The second selects a dance if it has not so far been danced with *any* partner. The third rule selects it if it has not so far been danced with the female who will now be asked. The last, default rule, selects any dance desire satisfying the minimal requirement. A male with this `chooseDesire` definition only actively tries to satisfy dance desires, but it could still end up with an intention of going to the bar as a result of negotiation with a female dancer.

Below is a `negotiateOver` procedure embodying a simple negotiation strategy.

```
ngtMess::= willYouDance(dance) | okDance(dance) |
           sorry | goBarWhen(dance) | okBarWhen(dance).
negotiateOver(toDance(D,N),FNm) ->
  ngtOverDance(D,N,FNm,[]).

ngtOverDance(D,N,FNm,PrevDs) ->
  willYouDance(D) >> hdl('negTh', FNm);
  ( okDance(D) << hdl('negTh', FNm) ->
      desire.replace(toDance(D,N),toDance(D,N-1));
      intention.add(toDanceWith(D,FNm))
  | sorry << hdl('negTh', FNm) -> {}
  | willYouDance(D2)::
      uncmtdDesire(toDance(D2,N2)) << hdl('negTh',FNm)->
        intention.add(toDanceWith(D2,FNm));
        desire.replace(toDance(D2,N2),toDance(D2,N2-1));
        okDance(D2) >> hdl('negTh', FNm)
  | willYouDance(D2) << hdl('negTh', FNm) ->
      counterP(FNm,[D,D2,..PrevDs])
  | goBarWhen(D2)::
      uncmtdDesire(barWhen(D2)) << hdl('negTh',FNm) ->
        intention.add(toBarWith(D2,FNm));
        desire.del(barWhen(D2));
        okBarWhen(D2) >> hdl('negTh', FNm)
  | goBarWhen(D2) << hdl('negTh', FNm) ->
      counterP(FNm,[D,D2,..PrevDs])).

  counterP(FNm,PrevDs)::
    chooseDesire(toDance(D,N),FNm),\+(D in PrevDs)->
      ngtOverDance(D,N,FNm,PrevDs).
  counterP(FNm,_) ->
      sorry >> hdl('negTh',FNm).
```

The negotiation is with the negotiation thread `hdl('negTh', FNm)` in the female dancer with name `FNm`. We assume, as with the male dancers, that this thread will have been spawned with this assigned handle.

The negotiation starts with the male sending a `willYouDance(D)` message to her negotiation thread. There are four possible responses: an `okDance(D)` accepting the invitation, a `sorry` message declining, a counter proposal to do another dance, or one to go to the bar when some dance is played. A counter proposal is accepted if it is currently an uncommitted desire. (This is when a male dancer may enter a commitment to go to the bar.) Otherwise, the male's `counterP` procedure is called to find an alternative dance to suggest. An alternative is suitable if it is a dance that satisfies the `chooseDesire` constraint for female `FNm` and it is different from all previous dances already mentioned in this negotiation (the `PrevDs` argument). If there is such a dance, the dance negotiation procedure is re-called to invite `FNm` to do this new dance. If no alternative new dance can be found, a `sorry` message is sent and the negotiation with this female ends.

5.5. A female dancer

The female dancer is similar to the male dancer; we assume that the female never takes the initiative. The female negotiation thread must wait for an initial proposal from a male but thereafter it can make counter proposals. It might immediately counter propose a different dance or to go to the bar, depending on its current desires and commitments.

5.6. The band

The band agent is a two threaded agent with one shared linda dynamic relation containing the names of dancers at the ball. One thread subscribes to the directory server in order to be sent the name of each dancer as it arrives at the ball and registers. The other thread recurses over a list of dances, which are its schedule of dances for the ball. For each dance `D` on the schedule it sends a `starting(D)` message to the intention execution thread of every dancer it currently knows about. It then pauses for a time dependent upon `D` before sending the `stopping` message to each dancer. After a delay for the inter-dance interval, when the dancers will negotiate, it announces the next dance on its schedule. When it reaches the end of the schedule, it sends the `dance_over` message.

6. Related work

6.1. Logic based programming languages

Qu-Prolog [8], BinProlog [34], CIAO Prolog [5], SICStus-MT Prolog [10], IC-Prolog II [6] are all multi-threaded Prolog systems. The closest to `Go!` are Qu-Prolog and IC-Prolog II.

Threads in Qu-Prolog communicate using messages or via the dynamic data base. As in `Go!`, threads can suspend waiting for another thread to update some dynamic relation. Threads in IC-Prolog II communicate either using unidirectional pipes, shared data base, or mailboxes. Mailboxes must be used for communication between threads in different invocations of IC-Prolog II. The language also has the L&O extension [23]. Neither language has higher order features or type checking support, and all threads in the same invocation share the same global dynamic data base. In `Go!`, a dynamic relation is the value of a variable. Only threads whose procedures access the variable as a global variable, or which are given access to the dynamic relation via a call argument or a message, can access it.

SICStus-MT [10] Prolog threads also each have a single message buffer, and threads can scan the buffer looking for a message of a certain form. But this buffered communication only applies to communication between threads in the same Prolog invocation. Threads running on different hosts must use lower level TCP/IP communication primitives.

In BinProlog [34], threads in the same invocation communicate using Linda tuple spaces [4] acting as shared information managers. BinProlog also supports the migration of threads, with the state of execution remembered and moved with the thread which can be used to implement mobile agents[9]. The CIAO Prolog system [5] uses just the global dynamic database for communicating between thread's in the same process. Through front end compilers, the system also supports functional syntax and modules.

Mozart/Oz [36] is a higher order, untyped, concurrent constraint language with logic, functional and object oriented programming components. Threads must be explicitly forked and can communicate either via shared variables bindings in the constraint store, which acts as a shared memory, or ports which are multiple writer/single reader communication channels similar to `Go!` message queues. Threads in different hosts can communicate using a ticket, which is an ascii string public name for a port.

Mercury [39] is a pure logic programming language with polymorphic types and modes. The modes are used to aid efficient compilation. It is not multi-threaded.

Escher [22], and Curry [16] are both hybrid logic/functional programming languages with types, the latter with type inference similar to `Go!`. They differ from `Go!` in using lazy evaluation for functions and Curry uses narrowing – function evaluations can instantiate variables in the calls. In [21] an `ensemble` primitive for Escher is proposed that forks a set of threads. The threads communicate via a shared global blackboard that contains mutable variables has well as I/O channels and files. Curry also has concurrent execution. Its threads communicate, as in concurrent logic programming, via incremental binding of shared variables, or via Oz style ports.

Concurrent MetateM [14] is based on temporal logic. Each agent executes a program comprising a set of rules with preconditions that refer to past or current events. The rules specify future events that may or must occur, that are in the control of the agent.

[9] `Go!` also supports mobile agents. They are transported as objects with updateable state and activated by spawning a call to an action method. Details are given in [37].

A broadcast communication to all other agents is one such event. Receipt of a message of a certain form might be a current or past event. The agent uses the rules to determine its behaviour by endeavoring to make the description of the future implied by the rules and events come true.

Dali [9] is an extension of Prolog which has reactive rules as well as normal clauses. It is untyped and not explicitly multi-threaded. The reactive rules define how a Dali agent reacts to external and internal events. The arrival of a message sent by another agent is an external event, as is a signal, such as `alarm_clock_ring`, sent by the environment. An internal event is a goal G that can be inferred from the history of past events. Internal events are generated as a result of the agent automatically attempting to prove certain goals at a frequency that can be specified by *try G* ... statements. The automatic retrying of these goals gives the agent implicit multi-threading. In that a Dali program determines future actions based on the history of past events, it is similar to Concurrent MetateM [14].

6.2. Agent programming languages

Agent0 [33], the related Placa [35], AgentSpeak(L) [31], 3APL [19], Vip [20] and ConGolog [15] are all proposals for higher level agent programming languages.

Agent0 was the first language to be proposed for agent programming. A defined communication protocol for inter agent communication is a key component. Agents can transfer information and send requests to one another to perform actions at a specified future time. There are three message types: `inform`, `request` and `unrequest`. The agents are purely reactive, they only do things on request. If a request is accepted, this generates a commitment to do the requested action when its time is due providing the agent remains capable of doing it.

An Agent0 agent is programmed by being given a repertoire of primitive actions, an initial set of beliefs, a set of commitment rules that are used to respond to messages, and a set of capability rules. A commitment rule could constrain an agent to only accept `inform` or `request` messages from specified agents. The commitment rules are like `Go!` message receive rules. When the commitment rules allow acceptance of some request for a private action, the agent must still query its capability rules to determine whether or not it is capable of performing the action at the requested time taking into account its current commitments and beliefs. (Our dancer agents similarly check against current commitments, and take into account their current dance desires – which can be viewed as a reflection on their dancing capabilities – before accepted a dance request.)

An Agent0 agent executes in a cycle: process all incoming messages updating beliefs and adding new commitments as necessary; prune commitments that it no longer believes it is capable of doing; execute all commitments that must be done at this time; repeat. The Agent0 commitments are similar to our dancer's intentions except that execution of an Agent0 commitment is triggered by a time rather than an event, and they are not commitments to joint action.

Placa [35] is an extension of Agent0 to allow agents to request achievement of a goal as well as a basic action. To handle a goal request the agent's have a plan library with conditions of applicability that are checked against the mental state. Plans are sequences of actions and sub-goals. As part of its cycle, a Placa agent does plan selection, and then plan refinement under a time constraint. Plan refinement is the expansion of sub-goals of a selected plan using other plans in the plan library. This contrasts with the use of plans in classic BDI architectures such as AgentSpeak described below. There subgoals are only expanded when they are reached during execution of the plan so that the selection of the plan for the sub-goal can take into account changes in the beliefs of the agent at that later time. A Placa agent uses spare cycle time to expand its plans, thereby saving delays in plan execution, but the early expansions may need to be abandoned due to updated beliefs when the sub-plan is reached. This is similar to an Agent0 agent dropping a commitment if it comes to believe it is no longer capable of perfoming the action.

AgentSpeak(L) [31] assumes an agent state comprising a set of events `E`, a set of beliefs `B`, and a set of intentions `I`. As in Dali [9], events are either external or internal. The external events correspond to inputs from the agent's environment. Internal events are generated during execution of a plan.

Events are terms of the form: `+B`, `-B`, which are events to add or remove a belief `B`, or of the form: `+!B`, `-!B`, which are events to add or drop an achievement goal `!B`. All internal events are `+!B` events. `!B` can be understood as the goal to achieve a state of the environment which will reported to the agent by an external `+B` event.

A plan is a rule of the form:

$$\texttt{e:}b_1\texttt{,..,}b_n\texttt{ -> }a_1\texttt{;...;}a_k$$

where `e` is the event term head of the rule, the b_i are belief terms comprising the guard, and a_i are action terms or goals to be sequentially executed if the plan is invoked. It is invocable only if `e` matches some event in `E` and the b_i all match some belief in `B`. A plan action term denotes an external action on the environment. A plan goal is either an achievement goal `!B`, or a test goal `?B`. When plan execution reaches an `!B` goal this results in a `+!B` internal event being generated. A test goal `?B` is a query to the agents belief store.

The plan rules are very similar to `Go!`'s action rules especially if we interpret an event as a message – an internal event being a message sent by the agent to itself – and actions are messages sent to the environment. The belief store corresponds to a `Go!` dynamic relation, however the AgentSpeak(L) paper does not indicate how the agents beliefs get updated. A `+B` external event/message can only be handled by invocation of a plan rule with head `+B'` where `B` and `B'` match. As described in [31], plan bodies can query the belief store but not update it. However, a belief update action could be added to the language – the equivalent of the dynamic relation update action of `Go!` action rules.

An AgentSpeak(L) agent executes in a cycle: select some event `e` from the pending set of events `E` using an event selection function S_E; from all the applicable plans for `e`

choose one plan p using a plan selector function S_P; if e is an external event generate a new intention (e,p), if e is an achieve goal event generated by an existing intention i' extend i' with p; select an intention i from the current set of intentions I using an intention selector S_I; execute the next goal or action of i; repeat. Asynchronously the 'environment' is adding events to the agents event store, and it is asynchronously 'absorbing' the agents actions. Each i in I is an execution thread and corresponds to the stack of plans currently being used to respond to some external event. However, the agent explicitly time shares between them using S_I, it has no internal concurrency. There is also no inter-agent communication model in the language – no special communication actions. The three selection functions S_E, S_P, S_I are black boxes, there is no means of programming them within the AgentSpeak(L) language. In Go! one could program them using relation definitions similar to the chooseDesire relation of the dancer agents. AgentSpeak(L) belief store querying is also quite weak – there is no inferencing using rules.

3APL [19] is similar to AgentSpeak(L) in having beliefs, plans and intentions, although it calls the intentions goals. However it lacks events. Indeed the language does not appear to have the notion of an agent environment. Actions directly update the agent's own belief store! But 3APL plans are more complex, they can branch and fork, and intentions can be executed concurrently. A plan may also modify an existing intention as well as extend it, say by replacing one sub-plan by another. But, as it stands, the language cannot be used for programming agents due to its lack of any interface to other agents or the environment.

Vip [20] agents also have events, beliefs and intentions in their internal state and plans are activated in response to events taking into account current beliefs. The major difference between it and AgentSpeak(L) is that Vip plans, like those of 3APL, can branch and fork, and intentions can be executed concurrently. Plans have a graphical representation as trees. Belief updates occur because epistemic effects can be associated with events and actions; although how these epistemic effects are specified is not described. (In Go! the epistemic effects of events and actions are explicitly programmed as updates of a dynamic relation. For example, a dancer responds to the event of receiving a message from the directory server or the band by updating its belief store, whereas it responds to the event of receiving an okDance message from a dancer by updating its intention store. It also records the successful execution of a dance by a new belief.) VIP allows a programmer to express priorities and preferences on plans and branches in plans, thus allowing a means to program the applicable plan selection function, which is not possible in AgentSpeak(L).

ConGolog [15] is a concurrent agent programming language rooted in the situation calculus representation of action and change in predicate logic. Beliefs have an environment state argument. This is either a special symbol S_0 denoting the initial state, about which the agent has a set of beliefs, or a term $do(a_n, do(a_{n-1}, \ldots, do(a_1, S_0)\ldots))$, denoting a state reachable from this initial state by some sequence of actions. Successor state axioms and applicability axioms for each action a enable the agent to check whether a can be used in a state S and what its effect

will be – what beliefs will characterise the state `do(a,S)`. The agent environment is represented by the notion of exogenous actions, and an assumption that there is a concurrent process executing these actions outside the control of the agent. Notification of an exogenous action is equivalent to an external event. What distinguishes ConColog from normal situation calculus is that it has a rich language for describing complex actions – effectively plans. It allows action sequencing, non-deterministic choice, iteration, concurrent execution with priorities, and belief querying. The agents execute such plans, and can infer the consequences of both plan and exogenous actions, using the successor state axioms. A query in the plan is implicitly against the inferable beliefs about the state of the environment resulting from the sequence of actions executed up to that point, including the exogenous actions.

7. Conclusions

`Go!` is a multi-paradigm programming language – with a strong logic programming aspect – that has been designed to make it easier to build intelligent agents while still meeting strict software engineering best practice. It is at a lower level than the agent programming languages just described, but can be used to quickly build agent applications using concepts from these languages. In addition, aspects of these languages, such as the selection functions of AgentSpeak(L), can be declaratively programmed. `Go!` agent's can have internal concurrency, can deductively query their state components, can communicate with other `Go!` agents using application specific symbolic messages, can negotiate, and can communicate with external processes using TCP/IP based communication. Using its DCG grammars [30], parsers for KQML [11] or FIPA ACL [12] message strings are straightforward to program. A parser for XML strings and files is distributed with the language.

Although not described in this paper, `Go!` also has a powerful mechanism for code re-use. It is based on the use of URIs to access compile-time inclusion of text and run-time loading of compiled program fragments. `Go!`'s mechanisms for code re-use are significant for two reasons: they support type safe code re-use in a convenient manner; and they permit `Go!` programs to be executed out of 'non standard devices' – devices which may have an execution environment but no file system.

In addition to these language level features, the `Go!` run-time system offers additional support for safe programming in the form of a code verifier. Verification is used to ensure that programs do not attempt to harm either the core engine or other legal `Go!` programs.

The ballroom scenario is an interesting use case for multi-agent programming. Although the agents are quite simple, it encompasses key *behavioural* features of agents: autonomy, adaptability, negotiation and commitment. Our implementation features inter-agent communication and co-ordination via messages, multi-threaded agents, intra-agent communication and co-ordination via shared memory stores. We believe these features, which are so easily implemented in `Go!`, are firm foundations on which to explore the development of much more sophisticated multi-threaded agents.

The Go! language is available under a GNU general public licence from Sourceforge (http://sourceforge.net/projects/networkagent/).

Acknowledgements

The first named author wishes to thank Fujitsu Labs of America for a research contract that supported the collaboration between the authors on the design of Go! and the writing of this paper.

References

[1] J. Armstrong, R. Virding and M. Williams, *Concurrent Programming in Erlang* (Prentice-Hall International, 1993).
[2] M.E. Bratman, D.J. Israel and M.E. Pollack, Plans and resource bounded practical reasoning, Computational Intelligence 4 (1988) 349–355.
[3] L. Cardelli and P. Wegner, On understanding types, data abstraction, and polymorphism, Computing Surveys 17 (1985) 471–522.
[4] N. Carriero and D. Gelernter, Linda in context, Communications of the ACM 32(4) (1989) 444–458.
[5] M. Carro and M. Hermenegildo, Concurrency in prolog using threads and a shared database, in: *Proceedings of ICLP99*, ed. D.D. Schreye (MIT Press, 1999) pp. 320–334.
[6] D. Chu and K.L. Clark, IC-Prolog II: A multi-threaded Prolog system, in: *Proc. of the ICLP'93 Workshop on Concurrent & Parallel Implementations of Logic Programming Systems*, eds. G. Succi and G. Colla (1993) pp. 115–141.
[7] K. Clark and F. McCabe, Ontology representation and inference in Go!, Technical Report, www.doc.ic.ac.uk/~klc/ontology.html (2003).
[8] K.L. Clark, P.J. Robinson and R. Hagen, Multi-threading and message communication in Qu-Prolog, Theory and Practice of Logic Programming 1(3) (2001) 283–301.
[9] S. Constantini and A. Tocchio, A logic programming language for multi-agent systems, in: *Proc. of JELIA02 – 8th European Conf. on Logics in AI*, Lecture Notes in Artificial Intelligence, Vol. 2424, (Springer, 2002) pp. 1–13.
[10] J. Eskilson and M. Carlsson, Sicstus MT – a multithreaded execution environment for SICStus Prolog, in: *Principles of Declarative Programming*, eds. K.M. Catuscia Palamidessi and H. Glaser, Lecture Notes in Computer Science, Vol. 1490 (Springer, 1998) pp. 36–53.
[11] T. Finin, R. Fritzson, D. McKay and R. McEntire, KQML as an agent communication language, in: *Proc. of 3rd International Conference on Information and Knowledge Management* (1994).
[12] FIPA, Fipa SL content language specification, Technical Report, Foundation for Intelligent Physical Agents, www.fipa.org (2001).
[13] FIPA, Fipa communicative act library specification, Technical Report, Foundation for Intelligent Physical Agents, www.fipa.org (2002).
[14] M. Fisher, A survey of concurrent MetateM – the language and its applications, in: *Temporal Logic*, eds. D. Gabbay and H. Ohlbach, Lecture Notes in Artificial Intelligence, Vol. 827 (Springer, 1994) pp. 480–505.
[15] G. D. Giacomo, Y. Lesperance and H. Levesque, Congolog, a concurrent programming language based on the situation calculus, Artificial Intelligence 1–2(121) (2000) 109–169.
[16] M. Hanus, A unified computation model for functional and logic programming, in: *Proc. of 24th ACM Symposium on Principles of Programming Languages (POPL'97)* (1997) pp. 80–93.
[17] H. Haugeneder and D. Steiner, Co-operative agents: Concepts and applications, in: *Agent Technology*, eds. N.R. Jennings and M.J. Wooldridge (Springer, 1998) pp. 175–202.

[18] R. Hindley, The principal type scheme of an object in combinatory logic, Trans. AMS 146 (1969) 29–60.
[19] K.V. Hindriks, F.S. de Boer, W. van der Hoek and J.-J.C. Meyer, Formal semantics for an abstract agent programming language, in: *Intelligent Agents IV*, eds. Singh, A.S. Rao and M. Wooldridge, Lecture Notes in Artificial Intelligence, Springer (1997) pp. 215–230.
[20] D. Kinny, VIP: A visual programming language for plan execution systems, in: *Proc. of 1st International Joint Conf. on Autonomous Agents and Multi-Agent Systems* (ACM Press, 2002) pp. 721–728.
[21] J. Lloyd, Interaction and concurrency in a declarative programming language, Unpublished report, Department of Computer Science, Bristol University, London (1988).
[22] J. Lloyd, Programming in an integrated functional and logic programming language, Journal of Functional and Logic Programming, (March 1999) 1–49.
[23] F. McCabe, *L&O: Logic and Objects* (Prentice-Hall International, 1992).
[24] F. McCabe, `Go!` Reference Manual, Technical Report, Available from author, or `http://sourceforge.net/projects/networkagent/` (2003).
[25] F. McCabe and K. Clark, April – Agent PRocess Interaction Language, in: *Intelligent Agents*, eds. N. Jennings and M. Wooldridge, Lecture Notes in Artificial Intelligence, Vol. 890 (Springer, 1995) pp. 324–340.
[26] D. Miller and G. Nadathur, An overview of λ-prolog, in: *Proc. of 5th International Conference and Symposium on Logic Programming*, Seattle (1988).
[27] R. Milner, A theory of type polymorphism in programming, Computer and System Sciences 17(3) (1978) 348–375.
[28] M. Minsky, A framework for representing knowledge, in: *Psychology of Computer Vision*, ed. P. Winston (MIT Press, 1975) pp. 211–277.
[29] A. Omnicini and F. Zambonelli, Coordination for internet application development, Autonomous Agents and Multi-Agent Systems 2(3) (1999) 251–269.
[30] F. Pereira and D.H. Warren, Definite clause grammars compared with augmented transition network, Artificial Intelligence 13(3) (1980) 231–278.
[31] A.S. Rao, AgentSpeak(L): BDI agents speak out in a logical computable language, in: *Agents Breaking Away*, Lecture Notes in Artificial Intelligence, Vol. 1038 (Springer, 1996) pp. 42–55.
[32] A.S. Rao and M.P. Georgeff, An abstract architecture for rational agents, in: *Proceedings of Knowledge Representation and Reasoning (KR&R92)* (1992) pp. 349–349.
[33] Y. Shoham, Agent0: An agent-oriented language and its interpreter, in: *Proceedings of the National Conference on AI (AAAI-91)* (1991) pp. 704–709.
[34] P. Tarau and V. Dahl, Mobile threads through first order continuations, in: *Proceedings of APPAI-GULP-PRODE'98*, Coruna, Spain (1998).
[35] S.R. Thomas, PLACA, an agent oriented programming language, PhD thesis, Department of Computer Science, Stanford University, Stanford (1993).
[36] P. Van Roy and S. Haridi, Mozart: A programming system for agent applications, in: *International Workshop on Distributed and Internet Programming with Logic and Constraint Languages*, Part of *International Conference on Logic Programming (ICLP 99)*, `http://www.mozart-oz.org/papers/abstracts/diplcl99.html` (1999).
[37] T.I. Wang and K.L. Clark, Distributed logic porgramming using mobile agents, in: *Proc. of 18th International Conference on Advanced Information Networking and Applications (AINA-2004)* (IEEE Press, 2004).
[38] S.N. Willmott, J. Dale, B. Burg, C. Charlton and P. O'Brien, Agentcities: A worldwide open agent network, Agentlink News (8) (November 2001) 13–15.
[39] F.H. Zoltan Somogyi and T. Conway, Mercury: an efficient purely declarative logic programming language, in: *Proc. of the Australian Computer Science Conference* (1995) pp. 499–512.