Go! for multi-threaded deliberative agents

K. L. Clark[1] and F. G. McCabe[2]

[1] Dept. of Computing, Imperial College, London
[2] Fujitsu Labs of America, Sunnuvale, CA

Abstract. `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 memory stores.
In this paper we introduce the essential features of `Go!` illustrating 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. An agent's 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.

1 Introduction

`Go!` is a logic programming descendant of the multi-threaded symbolic programming language `April`[22][3] , with influences from IC-Prolog II [5] and L&O[21]. `April` was initially developed as the implementation language for the much higher level MAI^{2}L[15] 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[31], and the agent services running on that platform at Imperial College and Fujitsu. We are currently investigating the use of `Go!` as an ontology server within that agent platform.

A significant theme in the design of `Go!` is software engineering in the service of high-integrity intelligent 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 public Internet, security, transparency and integrity are critical to the adoption of logic programming technology.

[3] Go is the sound of a Japanese word for 5. April is the 4th month.

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 (in the functional programming sense) and strongly typed, using a modified Hindley/Milner style type inference technique[23].

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* [21] extension of Prolog. A labeled theory is a 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 given values to these label variables. Labeled theories are analogous to class definitions, and their instances are `Go!`'s objects. Objects can have state, recorded by *cell* and *dynamic relation* 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[24].

`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 is 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. Although an artificial example we believe it is representative of many multi-agent applications.

In section 2 we give a brief overview of `Go!` and its facilities for programming task orientated agents. In the limited space available we cannot give a comprehensive description of `Go!`. For a more complete description of the language see [7].

In section 3 we explore `Go!` in the context of the simulated ballroom. Each dancer agent is programmed using multiple concurrently executing threads that implement different aspects of its behaviour – coordinated by shared `belief`, `desire` and `intention` dynamic relation memory stores. This internal run-time architecture has *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].

The `belief`, `desire` and `intention` memory stores are used in a manner similar to Linda tuple stores[3]. For example, memory store updates are atomic, and a thread can suspend waiting for a belief to be added or deleted. Linda tuple stores have been used for inter-agent coordination [25]. 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 4 we briefly discuss related work before giving our concluding remarks.

2 Key Features of Go!

`Go!` is a multi-paradigm language with a declarative subset of function and relation definitions and an imperative subset comprising action procedure definitions.

2.1 Function, relation and action rules

Functions are defined using sequences of rewrite rules of the form:

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

where the guard *Test* is omitted if not required.

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. Once a function rule has been selected there is no backtracking to select an alternative rule.

Relation definitions comprise sequences of Prolog-style `:-` clauses ; with some modifications – such as permitting expressions as well as data terms, and no cut. We can also define relations using *iff* 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:

$a(A_1,..,A_k)::Test \rightarrow Action_1;...;Action_n$

As with equations, 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.

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

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. To place a message in a thread's buffer the sender has to have the threads unique handle identity.

The message send action:

```
Msg >> To
```

sends the message *Msg* to the thread identified by the handle *To*. It is a non-blocking asynchronous communication. Handles are terms of the form `hdl(Id,Group)` where `Id` and `Group` are symbols that together uniquely identify the thread. Typically, threads within the same agent share the same `Group` name, which can be the unique agent's name.

To look for and remove from the message buffer a message matching *Ptn* sent by a thread *From* the receive action:

```
Ptn << From
```

can be used.

To look for any one of several messages, and to act appropriately when one is found, the conditional receive:

```
( Ptn_1 << From_1 -> Actions_1
| ...
| Ptn_n << From_n -> Actions_n
)
```

can be used. When executed, the message buffer of the thread is searched to find the first message that will *fire* one of these alternate message receive rules. The matched message is removed from the message buffer and corresponding actions are executed. Messages that don't match are left in the message buffer for a later message receive to pick up.

Both forms of message receive suspend if no matching message is found, causing the thread to suspend. The thread resumes only when a matching message is received. This is the message receive semantics of `Erlang`[1] and `April`[22].

Communication daemons and a special external communications system module allow threads in different invocations of `Go!` to communicate using the same message send and receive actions as are used between threads of a single invocation, see [7]. This allows an application comprising several modules, developed and tested as one multi-threaded `Go!` invocation, to be converted into a distributed application with minimal re-programming.

2.2 Programming behaviour with action rules

As an example of the use of action rules let us consider programming the top level of an agent with a mission: this is to achieve some fixed goal by the repeated execution of an appropriate action. The two action rule procedure:

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

captures the essence of this goal directed activity. ({} is the empty action.) This procedure would be executed by one thread within an agent whilst another concurrently executing thread is monitoring its environment, constantly updating the agent's beliefs about the environment; these beliefs being queried by *Goal*, and by *doNextStep*. `performMission` is a tail recursive procedure and will be executed as an iteration by the `Go!` engine.

Some missions – such as survival – do not have a termination goal but rather one or more continuation actions:

```
survive()::detectDanger(D) -> hideFrom(D);survive().
survive()::detectFood(F) -> eat(F); survive().
survive() -> wanderFor(safeTime()); survive().
```

The order of the rules prioritises avoiding danger. `safeTime` is a function that queries the belief store to determine a 'safe' period to wander, given current knowledge about the environment, before re-checking for danger. Again we assume the belief store is being concurrently manipulated by an environment monitoring thread within the agent. `hideFrom(D)` would typically cause the survival thread to suspend until the monitoring thread deletes those beliefs that made `detectDanger(D)` true.

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.
– An action rule guard – $(A_1,..,A_k)::Q$ – can augment the argument matching test with a query *Q*.
– If *Q* is a query, {*Q*}, indicating a single solution to *Q*, can appear as an 'action' in an action rule body.
– We can use a set expression {*Trm* || *Q*} to find all solutions to some query. This is `Go!`'s `findall`. Since `Trm` can involve defined functions, it can also be used to map a function over the set of solutions to `Q`.
– We can use `Go!`'s *forall* action. (*Q* `*>` *A*) iterates the action *A* over all solutions to query *Q*.
– We can use a conditional action. (*Q* ? A_1 | A_2) executes A_1 if *Q* succeeds, else A_2.

As an example of the use of `*>`:

```
(is_a_task(Task), \+ Icando(Task), cando(Ag,Task)
  *>  request(Task) >> Ag)
```

might be used to send a `'request'` message, for each task that the agent cannot itself do, to some agent it believes can do the task. `\+` is `Go!`'s negation-as-failure operator.

2.3 Type definitions and type inference

`Go!` is a strongly typed language; using a form of Hindley/Milner's type inference system[23]. 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 previously introduced in a type definition.

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 a constructor functions `toDance` mapping a `dance` and a `number` into a `Desire` and `barWhen` mapping a `dance`.

`Go!` has primitives types such as `symbol`, `string` and `number` and the polymorphic recursive type `list[T] -` a list of elements of of type `T` of unbounded length. So:

```
[1,4,-8]
[('harry',23),('paul',12)]
```

are respectively of type `list[number]`, `list[(symbol,number)]`. Notice that `'harry'` and `'paul'` are quoted. This is because `Go!` does not have a variable name convention like Prolog. Variable names can begin with upper or lower case letters. So, unless a symbol has been declared as a term of an enumerated type, such as `dance`, it must be quoted.

2.4 Dynamic relations

In `Prolog` we can use `assert` and `retract` to change the definition of a dynamic relation whilst a program is executing. The most frequent use of this feature is to modify a definition comprising a sequence of unconditonal clauses. In `Go!`, such a dynamic relation is an object with updateable state. It is an instance of a polymorphic system class `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.

The dynamic relations class has methods: `add`, for adding an argument term to the end of the current extension of the relation, `del` for removing the first argument term that unifies with a given term, `delall` for removing all argument terms unifying with a given term, `mem`, for accessing the instantiation of each current argument term that unifies with a given term, 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 using:

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

`dynamic` takes two kinds of argument. The type of the argument terms to be stored, in this case `Desire`, and any initial extension given as a list of terms. This list could be empty. 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` 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.add(barWhen(quickstep))
```

and:

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

The second is analogous to the following sequence of `Prolog` calls:

```
retract(desire(toDance(jive,N)),NewN is N-1,
assert(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 facts as a single action:

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

will delete all current `barWhen` desires. `delall` is the similar to `prolog`'s `retractall`.

2.5 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 the beliefs of the agent, and we want to allow some or all these threads to be able to update these beliefs.

We can represent the relations for which we will have changing information as dynamic relations. A *linda* a polymorphic subclass of the dynamic relations class has extra methods to facilitate the sharing of dynamic relations across threads. Instances if this subclass are created using initializations such as:

```
LinRel = $linda[type]([...])
```

For example, it has a `replace` method allowing the deleting and adding of a shared linda relation term to be executed atomically, and it has a `memw` relation method. A call:

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

will suspend if no term unifying with `Trm` is currently contained in `LinRel` until such a term is added by *another thread.*

There is also a dual, `notw` such that:

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

will suspend if a term unifying with `Trm` is currently contained in `LinRel` until all such terms are deleted by *other threads.* It also has a suspending delete method, `delw`.

`memw` and `delw` and the analogues of the Linda[3] `readw` and `inw` methods for manipulating a shared tuple store. There is no analogue of `notw` in Linda.

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 the type definition is the theta environment in which it appears.

A theta environment is of a set of definitions, each of which is either a

- A *Var*=*Expression* assignment definition
- A *Type*::=*TypeEpression* new type definition
- A *Type*:>*TypeEpression* renaming type definition
- A relation definition
- A function definition
- An action procedure definition
- A DCG grammar[26]
- A labeled theta environment - a class definition (see 2.7)
- A class rule - defining an inheritance relation (see 2.7)

grouped inside {} brackets. The rules and definitions are separated by the '`.␣`' operator[4].

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.

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

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

2.7 Classes and objects

Classes in `Go!` are labeled theta environments, which we can view as labeled theories as in L&O[21][5]. The labels can contain variables, which are global to all the definitions of the theory. The label variables *must* be explicitly typed, by attaching a type annotation.

Class definitions also double as type definitions - the functor of the class label is implicilty defined as a new type name that can be used to characterise the type of the object instances of the class.

We can create an instance of a labeled theory by giving values to the global variables of the theory label. The instance is an object characterised by these global variable values - they define its *static state*. Different object instances of the theory will generally have different values for these global variables.

Two system classes, the dynamic relations class and the cell class, have instances with mutable state. A new 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*.

Finally, inheritance can be used to define a new labeled theory. This is done using inheritance rules using the class labels.

The following set of definitions constitute a mini-theory of a person:

```
dateOfB :> (number,number,number).    -- type renaming def
sex::= male | female.                 -- new type def.
person(Nm:symbol,BrthDate:dateOfB,Sx:sex,Home:symbol){
    age()= __yearsBetween(time2date(now()),BrthDate).
    sex=Sx.
    name=Nm.
    lives(Home).
    __yearsBetween(....) => .....
}.
```

The label arguments `Nm`, `Brthdate`, `Sx`, `Home` are parameters to the theory which, when known, make it the theory of a specific person.

A person's age is computed by converting the difference between the current time returned by the primitive function `now`, converted to a date, and the person's date of birth. The conversion is done using a function `__yearsBetween` that is private to the theory. It is private since its name begins with `__`.

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

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

```
P1=$person('Bill',(1978,3,22),male,'London,England').
P2=$person('Jane',(1986,11,1),female,'Cardiff,Wales').

P1.name          -- returns name 'Bill' of P1
P2.age()           -- returns current age of P2
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, BrthDate,Sx, Hm, _,_)<=person(Nm, BrthDate,Sx,Hm).
student(_, _,_,_,Cge,Sbj){
  lives(Pl):-location_of(Cge,Pl).
  lives(Pl):-super.person.lives(Pl).
  studies_at(Sbj,Cge).
}.
```

The separate `<=` rule says that this theory inherits from the `person` theory with overriding inheritance. This means that any attribute defined in student with the same name as a person attribute automatically replaces the inherited definition. In this case, there is only one relation, `lives`, which is so redefined but its new definition explicitly extends the definition of the `parent` super class by virtue of its second clause.

`location_of` is defined outside the `student` theory. It has a normal definition such as:

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

We can create the *theory* of a specific student 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
```

Te above two labeled theories can be viewed as a small ontology about the `person` and `student` concepts. The use of `Go!`'s for ontology construction and querying is further explored in [6].

2.8 Modules

A module is a *where* expression that evaluates to a single higher order value, or to a tuple of values, some of which are higher order. A module that contains the class definitions for person and student given earlier, which exports both definitions, has the form:

```
(person,student)..{
person(Nm,Age,Sx,Hm){...}.
student(...)<=person(...).
student(_,_,_,_,Cge,Sbj){...}
}
```

The following is a module that exports the relation `ordered` and the function `reverse`. The definition of the auxiliary relation `ord` and the auxiliary function `rev` are local to the theta environment and not visible outside. `ordered` and `reverse` iare themselves defined using *where* expressions.

```
(ordered,reverse) .. {
   ordered(list,less) :- ord(list) .. {
     ord([]).
     ord([_]).
     ord([E1,E2,..L]):- less(E1,E2),ord([E2,..L]).
    }.
    reverse(L) => rev(L,[])..{
      rev([],R) => R.
      rev([E,..L],R) => rev(L,[E,..R]).
    }
}.
```

Incidentally, `ordered` is a higher order polymorphic relation of type:

```
[T]-(list[T], (T,T){}){}
```

which says that for any T ([T]- inidicates the quantification), it is a binary relation (signaled by the postfix {}), taking as first argument a list of elements of type T (the type expression list(T)) , and as second argument a binary relation over elements of type T (the type expression (T,T){}).

2.9 Higher order values

The `ordered` relation is *parameterized* with respect to the ordering relation used to compare elements of the list. `ordered` is further defined in terms of the auxilliary relation `ord`, itself defined 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 `less` variable – which holds the ordering relation – is only mentioned where it is important: where it is introduced as a parameter of `ordered` and where it is used in `ord`. 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 `less` relation through all the intermediate programs from the top-level to where it is needed; this is a significant source of irritation in `Prolog` programming.

A call to `ordered` must supply the list to be checked and an ordering relation. In many cases the ordering relation is given as the value of a variable with a higher order value[6]; 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, the call:

```
ordered([(3,5),(3,8),(10,12),...],
        ( ((X1,_),(X2,_)):-X1=<X2  | ((X,Y1), (X,Y2)) :-Y1=<Y2 ) )
```

The relation argument is given as a disjunction of lambda relation rules that uses the standard =< relation to define an ordering on pairs of numbers.

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

3 Multi-threaded dancer agents at a ball

In our agents' ball, we have male and female dancer agents that are attempting to dance with each other and a band that 'plays' music for different kinds of dances. The two kinds of dancer agent are required to discover like-minded agents and to negotiate over possible dance engagements. In addition to dancing, dancer agents may have additional goals – such as getting refreshed at 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 a BDI model[2][28], each agent has a `belief`, a `desire` and an `intention` relation. The `belief` relation contains beliefs about what other dancers there currently are and what dances they like to do. The `desire` relation contains the goals each dancer would like to achieve, for example, which dances it would like to dance. The `intention` relation holds its current intentions – these normally represent the agent's commitments to perform some particular dance with some partner agent; however, it can also be an intention to go to the bar when a dance is announced.

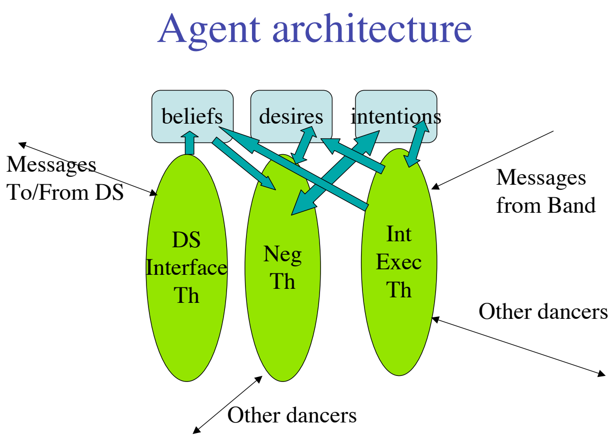
The dancers use a directory server to discover one another. As each dancer agent 'arrives' at the dance in some random and phased order, it registers with the directory server. The dancers also subscribe in order to be informed about other dancers that are already 'at the dance', and those that will arrive later.

The internal execution architecture of each dancer agent comprises three threads – corresponding to the three key activities of the agent: a directory server interface thread, a negotiations thread and an intention execution thread. The directory server interface interacts with the directory server to publish its own description and to subscribe for the descriptions of other dancer agents. The negotiations thread communicates with other dancer agents in order to

[6] Note the contrast with `Prolog`. In `Prolog` a relation is passed as argument by passing in its name - which is an atom. `Prolog`'s metal 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. Moreover, its type is checked at compile time to make sure it is consistent with its intended use. In Prolog, a type inconsistency will generally result in a runtime error or failure.

agree joint intentions to dance the next dance of a particular kind. The intentions execution thread coordinates the actual dance activities and any ‘drinking’ activities. The architecture is depicted in figure below.

These threads communicate using the shared linda dynamic relations: `belief`, `desire` and `intention`. Note that while all the dancers could be executed in a single invocation of the `Go!` engine, they will *not* have direct access to each others’ beliefs, desires and intentions. Furthermore, it is a simple task to distribute the progam across multiple invocations and machines, making each dancer a separate `Go!` process. The internal architecture of each agent is depicted in the figure above. The fat arrows indicate the internal agent communication through the shared dynamic relations, and the thin arrows indicate the external message communciation.



3.1 A 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 later stops playing each dance ‘number’.

The procedures for the intention execution threads of the male and female dancers are very similar with respect to how they ’listen’ for announcements from the band. They differ in what happens when a dance is starting and there is an intention to do that dance. We present here only the male case – as the male dancer is expected to take the initiative during the dance.[7]

[7] This symmetry is an aspect of the ballroom scenario; one that we would not expect for general agent systems.

```
maleIntention..{
 ... -- type defs
 maleIntention(belief,desire,intention,band) ->
    ( starting(D) << band ->
      belief.replace(bandNotPlaying,bandPlaying(D));
      check_intents(D);
       maleIntention(belief,desire,intention,band)
   | stopping(D) << band ->
      belief.replace(bandPlaying(D),bandNotPlaying);
       maleIntention(belief,desire,intention,band)
   | ball_over << band -> belief.add(ballOver)
   ).
  check_intents(D)::intention.mem(toDanceWith(D,FNm)) ->
      intention.del(toDanceWith(D,FNm));
      maleDance(D,FNm)).
  ... }
```

The above is a module that exports the `maleIntention` action procedure. The procedure iterates 'listening' for messages; in this case messages from the band. It terminates when it receives a `ball_over` message.

When it receives a `starting(D)` message, and there is an intention to do that dance, the `maleDance` procedure is executed. The intended partner should similarly have called its corresponding `femaleDance` procedure and the interaction between the dance procedures of the two dancers is the joint dancing activity.

Notice that the `maleIntention` procedure reflects its environment by maintaining an appropriate belief regarding what the band is currently doing and when the ball is over. `replace` is an atomic update action on a linda dynamic relation.

3.2 A 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 agent for they convert desires into new intentions using current beliefs and intentions. In contrast, the intentions execution and directory interface threads are essentially reactive activities.

A male dancer's negotiation thread must decide which uncommitted desire to try to convert into an intention, and, if this is to do some dance the next time it is announced, which female dancer to invite to do the dance. This may result in negotiation over which dance they will do together, for the female who is invited may have a higher priority desire. Remember that each dancer has a partial model of the other dancer in that it has beliefs that tell it the desires the other dancer registered with the directory server on arrival. But it does not know the priorities, or which have already been fully or partially satisfied.

The overall negotiation procedure is `satisfyDesires`:

```
 satisfyDesires()::belief.mem(ballOver) -> {}.
 satisfyDesires() ->
   {belief.memw(bandNotPlaying)}; -- wait until band not playing
   (chooseDesire(Des,FNm),\+ intention.mem(toDanceWith(D,_),
      still_ok_to_negotiate()) *>
               negotiateOver(Des,FNm));  -- negotiation loop
   {belief.memw(bandPlaying(_))};
       -- wait, if need be,  until band playing
   satisfyDesires().
 still_ok_to_negotiate():-
   belief.mem(bandNotPlaying),\+ belief.mem(ballOver).
```

The `satisfyDesires` procedure terminates when there is a belief[8] that the band has finished – a belief that will be added by the intentions execution thread when it receives the message `ball_over`. If not, the first action of `satisfyDesires` is the `memw` call. This is a query action to the `belief` relation that will suspend, if need be, until `bandNotPlaying` is believed. For our dancers we only allow negotiations when the band is not playing. This is not a mandatory aspect of all scenarios – other situations may permit uninterrupted negotiations over desires.

There is then an attempt to convert into commitments to dance as many unsatisfied desires as possible, before the band restarts or announces that the ball is over. This is done by negotiating over each such desire with a female `FNm` whom the male dancer believes shares the desire. When the negotiation *forall* loop terminates, either because there are no more solutions to `chooseDesire`, or the dancer no longer believes it is appropriate to continue negotiating, the action procedure waits, if need be, until the dancer believes the band has restarted[9] The possible wait is to ensure there is only one round of negotiation in each dance interval. The next time the band stops playing, the answers returned by `chooseDesire` will almost certainly be different because the beliefs, desires and intentions of the dancer will have changed. (Other female dancers may have arrived, and the dancer may have executed an intention during the last dance.) Even if one of the answers is the same, a re-negotiation with the same female may now have a different outcome because of changes in her mental state.

```
chooseDesire(toDance(D,N),FNm) :-
   uncmtdFeasibleDesire(toDance(D,N),FNm),
   (desire.mem(toDance(OthrD,OthrN)),OthrD\=D *> OthrN < N).
chooseDesire(toDance(D,N),FNm) :-
   uncmtdFeasibleDesire(toDance(D,N),FNm),
   \+ belief.mem(haveDanced(D,_)).
...
```

[8] All the procedures for this thread access the linda dynamic relations as global variables since the procedures will be defined in the environment where these relations are introduced.

[9] The `bandPlaying` belief will be added by its intention execution thread. If the band does not restart, the negotiation thread never resumes.

```
uncmtdDesire(toDance(D,N)):-
   desire.mem(toDance(D,N)), N>0,
   \+ intention.mem(toDanceWith(D,_)).
...
```

The above clauses are a partial definition of a `chooseDesire` that might be used by one of the male dancers. The two given clauses both return a dance desire only if it is currently uncommitted and feasible. It is an uncommitted desire if it is still desired to perform the dance at least once, and there is not a current intention to do that dance. (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 with the identified partner the *next* time *that* dance is announced.) It is feasible if the male believes some female still desires to do that dance. The first rule selects a dance if, additionally, it is desired more times than any other dance. The second selects a dance if it has not so far been danced with *any* partner. Each male dancer can have a different `chooseDesire` definition.

Below is a `negotiateOver` procedure for a simple male dancer negotiation strategy that starts with a dance proposal:

```
negotiateOver(Dance(D,N),FNm) ->
  ngtOverDance(D,N,FNm,hdl('neg',FNm),[]).
ngtOverDance(D,N,FNm,FNgtTh,PrevDs) ->
  willYouDance(D) >> FNgtTh; -- invite female to dance D
  ( okDance(D) << FNgtTh ->  -- female has accepted
      desire.replace(toDance(D,N),toDance(D,N-1));
      intention.add(toDanceWith(D,FNm))
  | sorry << FNgtTh -> {}  -- female has declined
  | willYouDance(D2)::uncmtdDesire(toDance(D2,N2))) << FNgtTh ->
       -- a counter-proposal to dance D2 accepted since uncom. des.
       intention.add(toDanceWith(D2,FNm));
       desire.replace(toDance(D2,N2),toDance(D2,N2-1));
       okDance(D2) >> FNgtTh
  | willYouDance(D2) << FNgtTh ->
       -- to dance D2 not an uncom. des., must counter-propose
       counterP(FNm,FNgtTh,[D,D2,..PrevDs])
  | barWhen(D2)::uncmtdDesire(BarWhen(D2)) << FNgtTh ->
       -- a counter-proposal to go to the bar D2 accepted
       intention.add(toBarWhen(D2,FNm));
       desire.del(BarWhen(D2));
       okBar(D2) >> FNgtTh
  | barWhen(D2) << FNgtTh ->
       -- to go to the bar when D2 not an uncom. des., counter-propose
      counterP(FNm,FNgtTh,[D,D2,..PrevDs])
  ).
```

```
counterP(FNm,FNgtTh,PrevDs)::
  (chooseDesire(toDance(D,N),FNm),\+(D in PrevDs))->
     -- continue with a proposal to do a new dance
    ngtOverDance(D,N,FNm,FNgtTh,PrevDs).
counterP(_,FNgtTh,_) -> -- terminate the negotiation
     sorry >> FNgtTh)).
```

The negotiation is with the negotiation thread, `hdl('neg',FNm)`, in the female dancer with name `FNm`.

The negotiation to fulfill a dance desire with a named female 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, or a counter proposal to do another dance, or to go to the bar when some dance is played. A counter proposal is accepted if it is currently an uncommitted desire. Otherwise, the `counterP` procedure is called to suggest an alternative dance. This calls `chooseDesire` to try find another feasible dance `D` for female `FNm`, different from all previous dances already mentioned in this negotiation (the `PrevDs` argument). If this succeeds, the dance negotiation procedure is re-called with `D` as the new dance to propose. If not, a `sorry` message is sent and the negotiation with this female ends. Each negotiation could be spawned as a new thread providing we use another dynamic relation to keep track of the current desire being considered in each negotiation to ensure they do not result in conflicting commitments.

3.3 The male dancer agent

Below we give the overall structure of the male dancer class definition. It uses modules defining the `maleIntention` and `DSinterface` procedures and it spawns them as separate threads.

```
maleDancer(MyNm,MyDesires,DS,band){
  .. -- type defs
  belief=$linda[Belief]([]).
  desire=$linda[Desire]([]).
  intention=$linda[Intention]([]).
  init() ->
    (Des on MyDesires *> desire.add(Des));
    spawn DSinterface(MyNm,male,belief,MyDesires,DS);
    spawn maleIntention(belief,desire,intention,band)
                          as hdl('exec',MyNm);
    spawn satisfyDesires() as hdl('neg',MyNm);
    waitfor(hdl('exec',MyNm)).
  .. -- defs of satisfyDesires etc
}
```

The init method of this class is the one called to activate an instance of the class: `$maleDancer(MyNm,MyDesires,DS,band).init()`. If we launch several dancers inside one `Go!` process this call would be spawned as a new thread.

An instance is specified by four parameters: a unique symbol name `MyNm` of the dancer agent, such as `'bill l. smith'`, a list `MyDsires` of its initial desires such as `[toDance(jive,2),barWhen(polka),..]`, and the handles `DS`, `band` of the directory server and band agent of the ball it is to attend. Each instance will have its own three linda dynamic relations encoding the dynamic state of the dancer.

The `init` action method adds each desire of `MyDesires` parameter to the dancer's `desire` linda relation. It then `spawns` the directory server interface, the intention execution and the negotiation threads for the dancer. The latter are assigned standard handle identities based on the agents symbol name. The `init` procedure then waits for the intention execution thread to terminate (when the ball is over). Termination of `init` terminates the other two spawned threads.

The negotiation thread executes concurrently with the other two threads. The directory interface thread will be adding beliefs about other agents to the shared `belief` relation as it receives `inform` messages from the directory server, and the execute intentions thread will be concurrently accessing and updating all three shared relations.

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.

4 Related Work

4.1 Other Logic Based Programming Languages

Qu-Prolog[8], BinProlog[29], CIAO Prolog [4], SICStus-MT Prolog[10], IC-Prolog II[5] 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. IC-Prolog also supports the L&O class notation[21]. 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 explicitly given access to the dynamic relation as a call argument or in 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.

Mozart/Oz[30] is a higher order, untyped, concurrent constraint language with logic, functional and object oriented programming components. Threads are explicitly forked and can communicate either via shared variable 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 public names for ports, which ascii strings called tickets. Tickets can be used to share any data value across a network.

In BinProlog[29], threads in the same invocation communicate through the use of Linda tuple spaces[3] acting as shared information managers. BinProlog also supports the migration of threads, with the state of execution remembered and moved with the thread[10]. The CIAO Prolog system [4] uses just the global dynamic Prolog database for communicating between thread's in the same process. Through front end compilers, the system also supports functional syntax and modules.

Mercury[32] 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 [20], and Curry [14] 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 of functions and Curry uses narrowing - function evaluations can instantiate variables in the calls. Escher has a proposed [19] explicit fork primitive `ensemble` 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 and its threads can communicate as in concurrent logic programming via incremental binding of shared variables, or via Oz style ports.

Concurrent MetateM [12] is based on temporal logic. Each agent executes a program comprising a set 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. A broadcast communication to all other agents is one such event. Receipt of a message of a certain form is a possible 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

[10] A `Go!` thread executing a recursive procedure can also be migrated by sending a closure containing a 'continuation' call to this procedure in a message. The recipient then spawns the closure allowing the threads computation to continue in a new location. The original thread can even continue executing, allowing cloning.

frequency that can be specified by *try G ...* statements. The periodic 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 [12].

4.2 Logic and action agent languages

Vip[17], AgentSpeak(L)[27], 3APL[16], Minerva[18] and ConGolog[13] are all proposals for higher level agent programming languages with declarative and action components. We are currently investigating whether the implied architectures of some of these languages can be readily realised in `Go!`. Vip and 3APL have internal agent concurrency.

These languages typically have plan libraries indexed by desire or event descriptors with belief pre-conditions of applicability. Such a plan library can be encoded in `Go!` as a set of `planFor` and `reactTo` action rules of the form:

```
planFor(Desire)::beliefCond -> Actions
reactTo(Event)::beliefCond -> Actions
```

The actions can include updates of the belief store, or the generation of new desires whose fulfillment will complete the plan. Calls to `planFor` or `reactTo` can be spawned as new threads, allowing concurrent execution of plans.

5 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. There are many important software engineering features of the language that we have not had the space to explore – for example the I/O model, permission and resource constrained execution and the techniques for linking modules together in a safe and scalable fashion. We have also omitted any discussion of how `Go!` applications are distributed and of how `Go!` programs interoperate with standard technologies such as DAML, SOAP and so on. Some of these topics are covered in [7].

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 and responsibility. 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 deliberative multi-threaded agents.

6 Acknowledgments

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*, 4:349–355, 1988.
3. N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.
4. M. Carro and M. Hermenegildo. Concurrency in Prolog using Threads and a Shared Database. In D. D. Schreye, editor, *Proceedings of ICLP99*, pages 320–334. MIT Press, 1999.
5. D. Chu and K. L. Clark. IC-Prolog II: a multi-threaded Prolog system. In G. Succi and G. Colla, editors, *Proceedings of the ICLP'93 Workshop on Concurrent & Parallel Implementations of Logic Programming Systems*, pages 115–141, 1993.
6. K. Clark and F. McCabe. Ontology representation and inference in Go! Technical report, Dept. of Computing, Imperial College, London, 2003.
7. K. Clark and F. McCabe. Go! – a multi-paradigm programming language for implementing multi-threaded agents. *Annals of Mathematics and Artificial Intelligence*, 2004, to appear.
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):283–301, 2001.
9. S. Constantini and A. Tocchio. A logic programming language for multi-agent systems. In *Proc. JELIA02 - 8th European Conf. on Logics in AI*, pages 1–13. Springer-Verlag, LNAI, Vol 2424, 2002.
10. J. Eskilson and M. Carlsson. Sicstus MT - a multithreaded execution environment for SICStus Prolog. In K. M. Catuscia Palamidessi, Hugh Glaser, editor, *Principles of Declarative Programming*, LNCS 1490, pages 36–53. Springer-Verlag, 1998.
11. T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an agent communication language. In *Proceedings 3rd International Conference on Information and Knowledge Management*, 1994.
12. M. Fisher. A survey of concurrent MetateM- the language and its applications. In D. Gabbay and H. Ohlbach, editors, *Temporal Logic*, pages 480–505. Springer-Verlag, LNAI, Vol 827, 1994.
13. G. D. Giacomo, Y. Lesperance, and H. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 1–2(121):109–169, 2000.
14. M. Hanus. A unified computation model for functional and logic programming. In *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 80–93, 1997.
15. H. Haugeneder and D. Steiner. Co-operative agents: Concepts and applications. In N. R. Jennings and M. J. Wooldridge, editors, *Agent Technology*, pages 175–202. Springer-Verlag, 1998.

16. 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 Singh, Rao, and Wooldridge, editors, *Intelligent Agents IV*, LNAI, pages 215–230. Springer-Verlag, 1997.
17. D. Kinny. VIP:A visual programming language for plan execution systems. In *1st International Joint Conf. Autonomous Agents and Multi-agent Systems*, pages 721–728. ACM Press, 2002.
18. J. A. Leite, J. J. Alferes, and L. M. Pereira. Minerva-A dynamic logic prorgamming agent architecture. In *Intelligent Agents VIII, LNAI 2333*, pages 141–157, 2001.
19. J. Lloyd. Interaction and concurrency in a declarative programming language. Unpublished report, Dept. of Computer Science, Bristol University, London, 1988.
20. J. W. LLoyd. Programming in an integrated functional and logic programming language. *Journal of Functional and Logic Programming*, pages 1–49, March 1999.
21. F. McCabe. *L&O: Logic and Objects*. Prentice-Hall International, 1992.
22. F. McCabe and K. Clark. April - Agent PRocess Interaction Language. In N. Jennings and M. Wooldridge, editors, *Intelligent Agents, LNAI, 890*, pages 324–340. Springer-Verlag, 1995.
23. R. Milner. A theory of type polymorphism in programming. *Computer and System Sciences*, 17(3):348–375, 1978.
24. M. Minsky. A framework for representing knowledge. In P. Winston, editor, *Psychology of Computer Vision*, pages 211–277. MIT Press, 1975.
25. A. Omnicini and F. Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-agent systems*, 2(3):251–269, 1999.
26. F. Pereira and D. H. Warren. Definite clause grammars compared with augmented transition network. *Artificial Intelligence*, 13(3):231–278, 1980.
27. A. S. Roa. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Agents Breaking Away*, LNAI 1038, pages 42–55. Springer-Verlag, 1996.
28. A. S. Roa and M. P. Georgeff. An abstract architecture for rational agents. In *Proceedings of Knowledge Representation and Reasoning (KR&R92)*, pages 349–349, 1992.
29. P. Tarau and V. Dahl. Mobile Threads through First Order Continuations. In *Proceedings of APPAI-GULP-PRODE'98*, Coruna, Spain, 1998.
30. 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*. http://www.mozart-oz.org/papers/abstracts/diplcl99.html, 1999. Part of International Conference on Logic Programming (ICLP 99).
31. S. N. Willmott, J. Dale, B. Burg, C. Charlton, and P. O'Brien. Agentcities: A Worldwide Open Agent Network. *Agentlink News*, (8):13–15, November 2001.
32. F. H. Zoltan Somogyi and T. Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, 1995.