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. 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.

1 Introduction

`Go!` is a logic programming descendant of the multi-threaded symbolic programming language `April`[15]. `April` was initially developed as the implementation language for the much higher level MAI^{2}L[10] 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[21], and the agent services running on that platform at Imperial College and Fujitsu.

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 public 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 such as Qu-Prolog[6], 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[16].

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 in which an expression is evaluated relative to a theta environment, and labeled theories, which are labeled theta environments.

Labeled theories are based on McCabe's *L&O* [14] 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 primitive *cell* and *dynamic relation* objects. New labeled theories can be defined in terms of existing theories using inheritance rules. Labeled theories provide a rich knowledge representation notation akin to that of frame systems.

This paper introduces the key features of `Go!` and illustrates its power and succinctness 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!`. In particular, space does not allow us to fully illustrate the OO features. For a more complete description see [5].

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 [17]. For scalability and other reasons, we prefer to use asynchronous point-to-point messages between agents, as in KQML[8]. 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:

$f(A_1,..,A_k)::Test \texttt{ => } Exp$

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:

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

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*. 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`[15].

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 [5]. 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 extend the argument matching 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 $\{Trm \texttt{ || } Q\}$ to find all solutions to some query. This is `Go!`'s `findall`.
- We can use `Go!`'s *forall* action. $(Q \texttt{ *> } A)$ iterates the action A over all solutions to query Q.
- We can use a conditional action. $(Q \texttt{ ? } A_1 \texttt{ | } A_2)$ executes A_1 if Q succeeds, else A_2.

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

```
(is_a_task(Task), I_cant_do(Task), cando(Ag,Task)
  *> ('request',Task) >> Ag)
```

might be used to send a `'request'` message for each current sub-task that the agent cannot itself do to some agent it believes can do the task. `'request'` is a *quoted symbol*, which is a primitive data type of `Go!`.

2.3 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. `Go!` has a system class that can be imported and used to create, update and call dynamic

relations defined using unconditional clauses. In addition, lists of higher-order values, such as relations of the same type, can be stored in cells. This allows us to store, modify and call dynamic relations defined by sequences of rules.

The dynamic relations class definition is imported by using:

```
include "sys:go/dynamic.gof"
```

A dynamic relation is an object with methods: `add`, for adding a fact term to the end of the current sequence of facts in the relation object, `del` for removing the first fact term in a dynamic relation object that unifies with some pattern, `delall` for removing all unifying fact terms, `mem`, for accessing the current fact terms in some dynamic relation component, and finally `ext` for retrieving the current extension as a list of fact terms.

Creating a new dynamic relation A dynamic relation object can be created and initialised using:

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

`dynamic` takes a list of terms that are the initial extension of the relations. This list could be empty. The above initialisation is equivalent to giving the following the 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.

2.4 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* 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([...])
```

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.

2.5 Type definitions and type inference

`Go!` is a strongly typed language; using a form of Hindley/Milner's type inference system[16]. 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 two constructor functions `toDance` and `barWhen`.

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][19], 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 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.

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.

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.[3]

```
maleIntention..{
 ... -- include statements
 dance::= polka | jive | waltz | tango | quickstep | samba.
 Belief::= hasDesires(symbol,Desire[])
          | bandNotPlaying
          | bandPlaying(dance)
          | ballOver
          | haveDanced(dance,symbol).
 Desire::= toDance(dance,number) | barWhen(dance).
 Intention ::= toDance(dance,symbol) | ...
 bandMessage::= starting(dance) | stopping(dance) | ball_over.

 maleIntention(belief,desire,intention,band) -> loop()..{
  loop() ->
    ( starting(D) << band ->
      belief.replace(bandNotPlaying,bandPlaying(D));
      check_intents(D,belief,desire,intention);
      loop()
   | stopping(D) << band ->
      belief.replace(bandPlaying(D),bandNotPlaying);
      loop()
   | ball_over << band -> belief.add(ballOver)
   ).
  check_intents(D)::intention.mem(toDance(D,FNm)) ->
      intention.del(toDance(D,FNm));
      maleDance(D,FNm)).
  ...
  } -- end of inner environment defining loop etc
}
```

The above is a module that exports the `maleIntention` action procedure. The `..` can be read as *where* and the definitions enclosed inside the {} braces following `..` are a *theta environment*. The procedure calls `loop`, which is itself defined using a *where* with an inner theta environment[4]. `loop` 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. This is given access to the dancer's beliefs and desires as it may need to modify them. The intended partner

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

[4] The parameters `belief`,..,`band` of the `maleIntention` procedure are in scope throughout the inner environment in which `loop` etc are defined.

should similarly have called its corresponding `femaleDance` procedure and the interaction between the dance threads 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 intentions using current beliefs. 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),\+[5]intention.mem(toDance(D,_),
     still_ok_to_negotiate()) *>
              negotiateOver(Des,FNm));  -- negotiation forall
  {belief.memw(bandPlaying(_))}; -- wait until band playing
  satisfyDesires().
still_ok_to_negotiate():-
  \+ believe.mem(bandPlaying(_)),\+ believe.mem(ballOver).
```

The `satisfyDesires` procedure terminates when there is a belief[6] 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.

[5] `\+` is `Go!` 's negation as failure operator.

[6] 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.

There is then an attempt to convert into commitments to dance as many unsatisfied dance desires as possible, before the band restarts. This is done by negotiation with a named female whom the male dancer believes shares that dance desire. When that iterative action terminates, the dancer checks that the band has restarted and waits if not. This 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(Dance(D,N),FNm) :-
   uncmtdFeasibleDesire(Dance(D,N),FNm),
   (desire.mem(Dance(OthrD,OthrN)),OthrD\=D *> OthrN < N).
chooseDesire(Dance(D,N),FNm) :-
   uncmtdFeasibleDesire(Dance(D,N),FNm),
   \+ belief.mem(haveDanced(D,_)).
chooseDesire(Dance(D,N),FNm) :-
   uncmtdFeasibleDesire(Dance(D,N),FNm),
   \+ belief.mem(haveDanced(D,FNm)).
chooseDesire(Dance(D,N),FNm) :-
   uncmtdFeasibleDesire(Dance(D,N),FNm).
uncmtdFeasibleDesire(Dance(D,N),FNm) :-
   uncmtdDesire(Dance(D,N)),
   belief.mem(hasDesires(FNm,FDesires)),
   Dance(D,_) in FDesires.
...
uncmtdDesire(Dance(D,N)):-
   desire.mem(Dance(D,N)), N>0,
   \+ intention.mem(toDance(D,_)),
...
```

The above clauses are tried in order, which reflects priorities. All the clauses return a dance desire only if it is currently uncommitted and the male believes some female desires to do that dance. 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.)

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. 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 desired, uncommitted feasible dance. 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 for a simple negotiation strategy:

```
ngtMess::= willYouDance(dance) | okDance(dance) |
           sorry | barWhen(dance) | okBar(dance).
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(Dance(D,N),Dance(D,N-1));
      intention.add(toDance(D,FNm))
  | sorry << FNgtTh -> {}  -- female has declined
  | willYouDance(D2)::uncmtdDesire(Dance(D2,N2))) << FNgtTh ->
       -- she has counter-proposed to dance D2, which is ok
       intention.add(toDance(D2,FNm));
       desire.replace(Dance(D2,N2),Dance(D2,N2-1));
       okDance(D2) >> FNgtTh
  | willYouDance(D2) << FNgtTh -> -- dance counter prop. not ok
       counterP(FNm,FNgtTh,[D,D2,..PrevDs])
  | barWhen(D2)::uncmtdDesire(BarWhen(D2)) << FNgtTh ->
       intention.add(toBarWhen(D2,FNm));
       desire.del(BarWhen(D2));
       okBar(D2) >> FNgtTh
  | barWhen(D2) << FNgtTh ->      -- bar counter prop. not ok
      counterP(FNm,FNgtTh,[D,D2,..PrevDs])
  ).
counterP(FNm,FNgtTh,PrevDs)::
  (chooseDesire(Dance(D,N),FNm),\+(D in PrevDs))->
     ngtOverDance(D,N,FNm,FNgtTh,PrevDs).
counterP(_,FNgtTh,_) -> -- cannot find a new dance desire
     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.

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){
  .. -- include statements
  belief=$linda([]).
  desire=$linda([]).
  intention=$linda([]).
  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 procedure of this module is the one called to activate an instance `$maleDancer(MyNm,MyDesires,DS,band)` of the class. An instance is specified by four parameters: the unique symbol name of the dancer agent, such as `'bill l. smith'`, a list of its desires expressed as `Desire` terms, and the handles 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 instance.

The `init` procedure 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. It can handle negotiations with male proposers one at a time, or have simultaneous negotiations by spawning auxiliary negotiation threads. This requires another dynamic relation to keep track of

the latest proposal of each negotiation so that they do not result in conflicting commitments.

4 Related Work

4.1 Logic Programming Languages

Qu-Prolog[6], BinProlog[20], CIAO Prolog [4], SICStus-MT Prolog[7] are all multi-threaded Prolog systems. The closest to `Go!` is Qu-Prolog. 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. However, Qu-Prolog has no higher order features or type checking support, and all threads in the same Qu-Prolog invocation share the same global dynamic data base. In `Go!`, using modules, dynamic relations can be restricted to a specified set of threads.

SICStus-MT[7] Prolog threads also each have a single message buffer, which they call a port, 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.

In BinProlog[20], 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[7]. 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[22] is a pure logic programming language with polymorphic types and modes. The modes are used to aid efficient compilation. It is not multi-threaded.

4.2 Logic and action agent languages

Vip[12], AgentSpeak(L)[18], 3APL[11], Minerva[13] and ConGolog[9] 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:

[7] 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.

```
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. Although many AI practitioners find the restrictions imposed by strong typing and other SE-oriented disciplines to be irksome, we find that we are not significantly hindered. In part this has been because we had the requirements for agent programming in mind in the design of `Go!`.

There are many other important qualities of a production environment 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. For a more complete description of some of these topics see [5].

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. K. Clark and F. McCabe. Go! – a logic programming language for implementing multi-threaded agents. Technical report, Downloadable from www.doc.ic.ac.uk/~klc, 2003.
6. 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.
7. 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.
8. 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.
9. 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.
10. H. Haugeneder and D. Steiner. Co-operative agents: Concepts and applications. In N. R. Jennings and M. J. Wooldridge, editors, *Agent Technology*. Springer-Verlag, 1998.
11. 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, Roa, and Wooldridge, editors, *Intelligent Agents IV*, LNAI. Springer-Verlag, 1997.
12. 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.
13. 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.
14. F. McCabe. *L&O: Logic and Objects*. Prentice-Hall International, 1992.
15. F. McCabe and K. Clark. April - Agent PRocess Interaction Language. In N. Jennings and M. Wooldridge, editors, *Intelligent Agents, LNAI, 890*. Springer-Verlag, 1995.
16. R. Milner. A theory of type polymorphism in programming. *Computer and System Sciences*, 17(3):348–375, 1978.
17. A. Omnicini and F. Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-agent systems*, 2(3):251–269, 1999.
18. A. S. Roa. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Agents Breaking Away*, LNAI 1038. Springer-Verlag, 1996.
19. 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.
20. P. Tarau and V. Dahl. Mobile Threads through First Order Continuations. In *Proceedings of APPAI-GULP-PRODE'98*, Coruna, Spain, 1998.
21. 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.
22. 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.