How Agents Do It In Stream Logic Programming

Matthew M Huntbach, Nick R Jennings and Graem A Ringwood

Queen Mary and Westfield College
LONDON E1 4NS, UK
(mmh@dcs.|N.R.Jennings@|gar@dcs.)qmw.ac.uk

Abstract

The key factor that will determine the speed and depth to which multi-agent systems penetrate the commercial marketplace is the ease with which applications can be developed. One approach is to use general purpose languages to construct layers of agent level constructs. Object-oriented languages have been advocated as appropriate for the complexity of distributed systems. According to Gasser and Briot [1992], the key problem with the common forms of object based concurrent programming is the fixed boundaries they give to agents are too inflexible. They do not reflect either the theoretical positions emerging in Multi-agent systems, MAS, nor the reality of multilevel aggregations of actions and knowledge. This paper advocates the use of a rather different type of object based concurrent language, stream logic programming, SLP, that does not have this drawback.

1 Object Based Concurrent Programming

The key factor that will determine the speed and depth to which multi-agent systems, MAS, penetrate the commercial marketplace is the ease with which applications can be developed. The few agent applications that, presently, exist are hand-crafted from scratch for each new problem, [e.g. Jennings, 1994]. This means they have a high overhead as the relatively complex infrastructure for agent computing needs to be put in place before the rest of the application can be constructed. A further impediment to the take up of MAS is many of the tools and techniques that are currently being deployed appear ill-suited to the problem - they are not designed for easy and concise expression of the problem, nor the concomitant complexity, of agents and agent interactions.

One way of alleviating these problems is to provide high level constructs and primitives that facilitate the implementation of agents and multi-agent systems. There are, essentially, two ways of doing this. Firstly, new languages are proposed in which agent-level features and interactions are primitives. For example, Shoham's [1990] agent-oriented paradigm involves programming agents directly with intentional notions like beliefs and commitments. Other languages of this genre include PLACA [Thomas, 1994] and MAIL [Steiner, 1995]. (This paper will perpetuate the annoying convention of writing language names in upper case even when they are not acronyms.) The second approach is to use more general purpose languages to build higher layers that provide agent level constructs. Within this category, object based concurrent programming, OBCP, has emerged as the major contender [Gasser and Briot, 1992]. In OBCP, a number of self-contained concurrently executing objects communicate by message passing.

The main reason for the rise of OBCP in MAS is the prospect of better coping with the complexity of running large programs on distributed systems [Colouris et al., 1993]. Well-known OBCP languages for programming distributed artificial intelligence, DAI, systems include ACTORS [Agha, 1990] and ABCL [Yonezawa, 1990]. ORG [Hewitt and Inman, 1991], is an agent organisational layer above ACTORS. While admitting that OBCP has some complementary aims with DAI, Gasser and Briot [1992] point to some deficiencies. The key problem the authors identify with the common forms of OBCP is object identity.

According to Gasser and Briot, the fixed boundaries for agents which OBCP gives are too inflexible. They do not reflect either the theoretical positions emerging in DAI nor the reality of multilevel aggregations of actions and knowledge. The present paper advocates a rather different type of OBCP, stream logic programming, SLP, for MAS, which does not have this drawback. A stream [Kahn, 1974] can be thought of as a message queue [Abelson et al, 1985].

SLP developed from attempts parallelise PROLOG [Ringwood, 1988; 1989; 1994]. This endeavour was given impetus by the Japanese Fifth Generation Initiative, FGI. The FGI realised that the critical application for a kernel language of any machine is the operating system [Chikayama, 1993a]. Without an operating system, parallel hardware is virtually useless. In his thesis, Durfee [1988] also recognises the converging concerns of distributed operating systems and DAI. The appropriateness of streams for systems programming was a principle justification for their introduction [Kahn, 1974].

At an early stage, SLP was interpreted as OBCP [Shapiro and Takeuchi, 1983]. The objects of SLP, though, are ermergent and do not have identitiy. This paper introduces an exemplar of SLP, Reactive, Guarded, Definite Clauses, RGDC [Cohen et al, 1992], by means of simple examples. More complex examples pertinent to DAI and MAS are gradually developed. Because of the limits on space, the introduction assumes a little familiarity with ACTORS and PROLOG.

2 Clauses as Behaviours

A superficial difference between ACTORS [Agha, 1990] and SLP is the syntax. Actor like languages generally use LISP syntax [Tomlinson and Scheevel, 1989] - the LISP syntax reveals the language in which the interpreters are written and the ancestor, PLANNER [Hewitt, 1969, Sussman et al 1971]. McCarthy [1988] claims that PLANNER can also be recognised as a precursor of PROLOG. Stream logic languages, naturally, have a PROLOG-like syntax.

Like ACTORS, a stream logic program consists of a fixed set of named behaviours. Standard object-oriented terminology uses the word "class" in place of behaviour. In logic programming, this is a set of definite clauses with the same predicate (name) at the head of a clause.

Unlike conventional OBCP, there is no implicit message queue associated with each object. Rather, there is an array of message reception slots. The following clause fragment gives the head of one behaviour for an **act** object

```
  act(X, Y) :- ...
```

with two message slots (variables) X and Y. As a syntactic convention of PROLOG, message slot names start with upper case letters. The scope of the message slot name is the clause. The symbol, :-, separates the class name from constraints on the message. A distinction is made between the same behaviour name with differing numbers of message slots (arity). For example, **act/2** with two message slots is considered different from **act/3** with three message slots.

In conventional OBCP, there is a queue of message slots associated with each object. As message slots are explicit in SLP, there is no implicit merging of incoming messages. The fixed number of message slots may seem debilitating but as will be shown it is much more flexible than a message queue.

2.1 Message selection as condition synchronisation

The general form of a clause is <guard> <- <behaviour> where the guard is further decomposed <head> :- <boolean constraint>. The <- symbol separates constraints from the behaviour which can be enacted if the constraints are satisfied.

```
  act(X, Y) :- X=m   <- ...
  + act(X, Y) :- X=n <- ...
```

The + symbol is used as the separator between alternative condition-behaviour pairs. As with OBCP, each method specifies the behaviour of an object if it receives a message with a particular selector. A message in SLP is a term in the sense of predicate logic. The principle functor of the term is the selector name. Again selectors are identified by name and arity.

The above program fragment illustrates the heads of methods for two behaviours for **act/2**. If an object of the class **act/2** receives a message m/0 on its first message slot it adopts the behaviour specified by the first clause. Alternatively, if it receives a message n/0 it adopts the behaviour specified by the second clause. As a syntactic convention of PROLOG, selector and behaviour names begin with lower case letters. If no other clauses are given for **act/2**, an object of the class remains suspended until a message is received on the first message slot. This is blocking receive. In this simple example, if a message is received which is not m/0 nor n/0, the object remains suspended forever.

Pattern matching is used as a shorthand for equality constraints. The message selection described for the class **act/2** is:

```
  act(m, Y)  <- ...
  + act(n, Y) <- ...
```

This pattern matching is not restricted to zero arity messages, but applies to terms of arbitrary depth (Note this is pattern matching and not unification as in PROLOG.)

The constraint X=/Y can be used to describe behaviour if particular messages are not received:

```
  + act(X, Y) :- X=/m & X=/n <- ...
```

The ampersand denotes conjunction. If an object of the class **act/2** receives a message that is neither m/0 nor n/0, the object will enact the behaviour specified by the third clause. If more than one clause guard is satisfied the first in textual order is chosen. This feature can be used to program priority and fairness as will be shown in Section 3.1.

An object that fails to receive the messages it is expecting will remain inactive. This feature can be used to implement time-outs as illustrated in Section 3.3. A nonblocking receive can be specified using priority of clause ordering:

```
  act(m, Y)  <- ...
  + act(n, Y) <- ...
  + act(X, Y) <- ...
```

Clauses have priority given by top to bottom ordering. If an m/0 or n/0 message is received it will be dealt with by appropriate clause. If no message or some message other than m/0 or n/0 has been received, the behaviour of the last clause will be used.

In SLP, message selection is more refined than is usual for OBCP. It is possible to invoke an object behaviour only when it receives a message on two or more message slots. A class behaviour can be further restrained to be enacted only if messages satisfy certain primitive test conditions, called *guard* constraints. The following guards for **max/2**, denote that it requires two numerical messages to be received on different slots:

```
  max(I, J)   :- I>J  <- ...
  + max(I, J) :- I=<J <- ...
```

If no other clauses are specified, an object of the class will remain suspended until messages are received on both message slots (the constraints cannot be evaluated until messages have been received). This condition synchronisation was inspired by the guarded commands of

Dijkstra [1975] but it has an earlier manifestation in decision tables [Cohen and Ringwood, 1994]. In ACTORS [Agha, 1990] a conditional expression is used instead of guards.

2.2 Goals as short-lived objects

An object is a "goal" in logic programming and is an instance of a behaviour, with a number of named message slots. The object itself is not named and thus may not be referred to directly. In Actor-like languages, objects have names and these can be passed as arguments to newly formed objects. Message slots are implicit and messages are sent to named objects. It is not possible to view or otherwise manipulate the message slots on which messages are received.

In SLP, message slots are explicit - they have local names that can be manipulated by the programmer. Message slots can be sent as messages to newly created objects. The body of a clause is interpreted as a network of concurrent objects connected by communication message slots. This is represented as a list of objects (separated by |).

```
<- A<<m | act(A, B) | bct(B, C) | ...
```

These objects are either new instances of defined classes or a primitive message-send object. The message-send object is indicated by the syntax Message-slot<<Message, similar to C++. Message sending is asynchronous.

The class of objects **act/2** introduced above which can receive different messages on its first message slot can send appropriate responses to its second message slot as indicated below:

```
act(m, Y)  <- Y<<q
+ act(n, Y) <- Y<<r
```

Behaviour is specified by object replacement in the same way as ACTORS [Agha, 1990]. Object termination is specified by a **done/0** replacement.

```
bct(q, Y) <- done
```

A **bct/2** class object can terminate if it receives a q/0 message where the behaviour **done** signifies no replacement.

With the behaviours given, **act/2** and **bct/2** can only suspend, waiting to receive messages. The message send primitive never blocks. A possible scenario of message passing and object replacement is as follows:

```
<- A<<m | act(A, B) | bct(B, C) | ...
<- act(m, B) | bct(B, C) | ...
<- B<<q | bct(B, C) | ...
<- bct(q, C) | ...
<- ...
```

The underscore indicates objects that are not suspended. The reception of the message m/0 activates the first clause of **act/2** . The object **act/2** is replaced by its behaviour, another message send primitive.

2.3 Emergent long-lived objects and streams

Goal objects are, like Actor objects, ephemeral. Once a method (clause) is selected, an object is replaced by the network of objects specified in the behaviour. Once a message has been received in a message slot it cannot be used to receive another message (nondestructive). However, messages are terms and terms may themselves have message slots - a sort of reply paid envelope.

```
act(m(X), Y)  <- Y<<q(Z) | cct(X, Z)
+ act(n(X), Y) <- Y<<r(Z) | dct(X, Z)
```

Here, the **act/2** object waits to receive an m/1 or n/1 message. These messages contain a message slot within them. Rather than deal with the message on slot X itself, the act/2 object creates an **cct/2** or **dct/2** object to deal with it. The message slot names used in the body of a clause may be those from the head of the method, or new ones that only appear in the body. This is similar to the way an Actor object handles subsequent messages, by creating new ACTORS [Agha, 1990]. In combination with guard priority, this may be used to implement delegation [Liebermann, 1986]. Delegation is an alternative to inheritance in OBCP that is better suited to distributed systems. Instead of passing incomprehensible messages to a higher class, delegation passes the message onto other objects to deal with it.

Messages may be multicast by using multiple occurrences of message slots in the body of clauses.

```
<- X<<m | act(X, ...) | bct(..., X, ...) | cct(..., X)
```

An object may reincarnate itself by replacing itself with a recursive copy:

```
bct(q(X), Y) <- bct(X, Y)
```

effectively creating a long lived object that can receive a stream of q/1 messages. A stream is a potentially infinite message buffer. Tail recursion can be implemented as efficiently as repetition. A recursively reincarnated object can send itself a message:

```
bct(q(X), Y) <- Z<<r(X) | bct(Z, Y)
```

This is similar to the capacity offered by languages such as SMALLTALK [Goldberg and Robson, 1983] for an object to send a message to itself using the special message queue self. A more efficient specification having the same effect is:

```
bct(q(X), Y) <- bct(r(X), Y).
```

The class **max/2**, above, illustrates how objects may wait for messages on two message slots simultaneously. The pattern matching ability of guards can be used to select methods that are only invoked by objects that receive consecutive messages on the same stream:

```
bct(s(t(X)), Y) <- bct(X, Y)
```

This can be used to program atomic transactions for which several messages are required [Cohen et al, 1992].

Binary messages can be used to create, a perhaps, more readable syntax for channels or streams. For example, using a right associative, binary infix functor :/2 the necessary receipt of two unary messages can be specified as:

```
bct(s:t:X, Y) <- bct(X, Y)
```

Parentheses are avoided by taking :/2 to be right associative. PROLOG-like list syntax may also be used where [X|Y] is a binary message:

```
bct([s, t |X], Y) <- bct(X, Y)
```

The notation [X,Y|Z] is shorthand for [X|[Y|Z]]. Streams in SLP are often explained as PROLOG lists [Shapiro and

Takeuchi, 1983]. This exposition has deliberately not done this to illustrate that this is far too narrow a view of the capability afforded by messages with embedded message slots.

3 Naive Agents

SLP has now been sufficiently explicated to give an example of negotiation between naive agents. Below is a RGDC program for the Winograd and Flores [1986] haggling protocol:

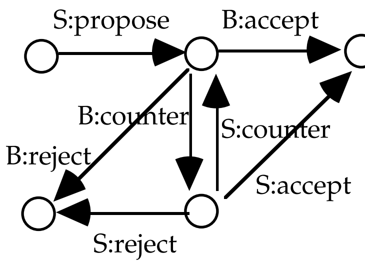

Figure 1: A Winograd Flores labelled digraph

The buyer and seller start concurrently with a message for the buyer, the seller's asking price, already waiting:

```
<- seller(100, Counter, 50) | buyer(30, 100:Counter, 60)
```

As with C++, comments are indicated by lines beginning with a pair of slashes

```
//seller(Current_Asking_Price, Offers, Lower_Limit)
```

Comments are usually limited to explaining the type of messages the message slots are expecting.

The seller is specified as long lived object with a message stream of offers to buy. If the seller receives an offer greater than its lower limit, a message agreeing the price is sent.

```
seller(Ask, Offer:Counter, Lower_Limit)
    :- Offer>Lower_Limit
    <- Counter<<accept(Offer)
        | seller(Ask, Counter, Lower_Limit)
```

If the seller receives an offer less than the lower limit it makes a counterbid that is half of the previous asking price and the buyers offer.

```
+ seller(Ask, Offer:Counter, Lower_Limit)
    :- Offer<Lower_Limit & New_Ask:=(Ask+Offer)/2
        & New_Ask>Lower_Limit
    <- Counter<<New_Ask:New_Counter
        | seller(New_Ask, New_Counter, Lower_Limit)
```

The previous asking price is a self message. Simple arithmetic, such as NewAsk:=(Ask+Offer)/2, is performed in the guard.

If the seller receives an offer less than its lower limit, it sends a message denying agreement is possible.

```
+ seller(Ask, Offer:Counter, Lower_Limit)
    :- Offer<Lower_Limit & New_Ask:=(Ask+Offer)/2
        & New_Ask<Lower_Limit
    <- Counter<<reject(Offer)
        | seller(Ask, Counter, Lower_Limit).
```

The code for the buyer is similar.

```
//buyer(Current_Offer, Asks, Upper_Limit)
buyer(Offer, Ask:Counter, Upper_Limit)
    :- Ask<Upper_Limit
    <- Counter<<accept(Ask)
        | buyer(Offer, Counter, Upper_Limit)
+ buyer(Offer, Ask:Counter, Upper_Limit)
    :- Ask>Upper_Limit & New_Offer:=(Ask+Offer)/2
        & New_Offer<Upper_Limit
    <- Counter<<New_Offer:New_Counter
        | buyer(New_Offer, New_Counter, Upper_Limit)
+ buyer(Offer, Ask:Counter, Upper_Limit)
    :- Ask>Upper_Limit & New_Offer:=(Ask+Offer)/2
        &  New_Offer>Upper_Limit
    <- Counter<<reject(Ask)
        | buyer(Offer, Counter, Upper_Limit).
```

The clauses for the behaviours when the seller and buyer receive accept or reject messages have been omitted. (Presumably they proceed to the exchange.)

The clauses of a program can be interpreted as decision tables [Cohen and Ringwood, 1994]. Each clause is a rule of a table and each guard the condition of a rule. The body of a clause gives the action to be performed if the condition is satisfied. Decision tables are often used to specify finite state machines. A connection between the Winograd Flores network diagram and finite state machines is readily apparent.

The given initial object network results in the following message exchange scenario:

```
<- seller(100, Counter, 75)
    | buyer(40, 100:Counter, 80)
<- seller(100, Counter, 50) | Counter<<70:Counter1
    | buyer(70, Counter1, 80)
<- seller(100, 70:Counter1, 75)
    | buyer(70, Counter1, 80)
<- seller(85, Counter2, 75) | Counter1<< 85:Counter2
    | buyer(70, Counter1, 80)
<- seller(85, Counter2, 75)
    | buyer(70, 85:Counter2, 80)
<- seller(85, Counter2, 75) | Counter2<<77.5:Counter3
    | buyer(77.5, Counter3, 80)
<- seller(85, 77.5:Counter3, 75)
    | buyer(77.5, Counter3, 80)
<- seller(85, Counter3, 75) | Counter3<<accept(77.5)
    | buyer(77.5, Counter3, 80)
<- seller(85, accept(77.5), 75)
    | buyer(77.5, accept(77.5), 80)
```

3.1 Fair merge

In the haggling protocol above the objects communicate directly with one another. More usually, it is convenient (particularly with input/output) to determine or change at runtime the number of objects communicating. In usual object languages, messages to an object may come from several different other objects and are implicitly merged into one input message queue. There is no implicit merging of messages in SLP but fair merges can easily be programmed:

```
//merge(In_Stream1, In_Stream2, Merged_Stream)
merge(Item:S1s, S2s, Ms)
    <- Ms<<Item:M1s
    | merge(S2s, S1s, M1s)
+ merge(S1s, Item:S2s, Ms)
    <- Ms<<Item:M1s
    | merge(S2s, S1s, M1s)
```

In usual message passing systems, the order of arrival of messages is not necessarily the order in which they were sent. However, with the **merge/3** object, above, the relative order of terms produced by sources is preserved.

When there are messages on the two input streams of the **merge/3** object the arguments are reversed in recursive incarnations. This prevents starvation. With a fixed order of testing of alternative guards, changing the order of arguments causes the list from which an element is taken to alternate. This serves to effect fairness. A biased merge (without interchanging the streams) gives a way of programming priority to a stream.

If the message done/0 is interpreted as closing the stream, then adding the methods

```
  + merge(done, Ss, Ms) <- Ms<<Ss
  + merge(Ss, done, Ms) <- Ms<<Ss.
```

causes message redirection. Here a message slot is sent as a message.

3.2 Subservient agents

An important form of interaction between agents is client-server co-operation in which several agents ask another agent to perform a service on its behalf. In such interactions, it is common that several agents require the same service. Using a fair merge to communicate requests for service, the archetype server objects can be specified:

```
//server(Request:More_Requests, Database)
server(R:Rs, D)
   <- transaction(R, D, New_D) | server(Rs, New_D)
```

The first argument of the **server/2** is a stream of transactions from clients. The second message slot is a self message that represents the state of the server. The self message is to be generated according to the form of transaction stipulated in the transaction message. Because the task and the server run concurrently, the server can respond to new requests before a prior one is completed. Concurrency rather than sequencing is the default in SLP.

In procedural languages, remote procedure calls are often used to implement servers. With a remote procedure call, the client has to wait for a reply from the server before continuing. The client can be made to wait until the transaction is complete by supplying a reply message slot together with the task:

```
//client(State, Requests, Reply)
client(N, Rs, done) :- N1:=N+1
   <- Rs<<do(N, Reply):Ts | client(N1, Ts, Reply)
```

Here the client state is a simple count of the transactions. The **client/3** is forced to wait until it receives a done/0 message before it can continue. The server can be forced to wait until the transaction completes in a similar way:

```
  server1(R:Rs, data(D))
     <- transaction(R, D, New_D) | server1(Rs, New_D)
```

For simplicity assume that the state of the server is just a record of the requests processed:

```
transaction(do(Job, Reply_to), D, D1)
   <- D1<<data(Job:D) | Reply_to<<done
```

If the transaction is not as simple as the one here there is no guarantee that it will complete before the reply is sent. Even as it stands, there is no guarantee that the data/1 message will be sent to itself before the done/0 message is sent to the client. There is limited sequencing in SLP due to the guard but more than this has to be programmed with continuations. Continuations arose in denotational semantics to define the meaning of sequencing [Strachey and Wadsworth, 1974]. As message send is a primitive a three message slot version is needed to specify a continuation. The disfix primitive object X<<M;Cont sends a message M to a message slot X and asynchronously continues as the object specified by Cont. Here Cont is a metamessage that ranges over networks of objects (goals).

```
transaction1(do(Job, Reply_to), D, D1)
   <- Reply_to<<done;(D1<<data(Job:D))
```

In this simple example, a more sophisticated alternative to making the server suspend waiting for a data/1 message is to send a more complex message containing a continuation:

```
  server2(do(Job, Reply_to):Rs, D)
     <- Reply_to<<done;server2(Rs, data(Job:D))
```

3.3 Market forces

Another much vaunted co-operation protocol is the contract net [Smith, 1980]. In this, politically correct example of market forces, a customer requests contractors to tender for a specified job.

```
  //contract_net(Specific_Job, Set_of_Contractors)
  contract_net(Job, Contractors)
     <- tender(Job, Contractors, done, Tenders)
     | time(5, Time) | wait(Time, Tenders)
```

The **tender/4** distributes the Job description to the set of contractors with a unique reply message slot (as in the **client/2** example above).

```
//tender(Job, Contractors, Replies, Tender_Reply_Pairs)
tender(Job, Contractor:Cs, Replies, Tenders)
   <- Contractor<<offer(Job, Reply)
      | tender(Job, Cs, Reply:Replies, Tender)
+ tender(Job, end, Replies, Tenders)
   <- Tenders<<Replies
```

Reply message slot pairs are accumulated in the self message Tenders.

The object **time/2** is a primitive object which immediately it is spawned initialises a realtime clock. After the designated 5 time periods have elapsed, **time/2** sends a message time_up on the message slot Time.

```
//wait(Time, Set_of__Tender_Reply_Pairs)
wait(time_up, Tenders)
   <- select(Tenders, Best_offer)
```

Man [1993] describes analysis techniques that, he claims, makes SLP suitable for hard realtime systems.

4 More Sophisticated Agents

Reflexive or metalevel architectures, where some part of an agent reasons about and influences some other part, are often advocated for MAS [Maes, 1988; Ferber and Carle, 1992]. REACTTALK [Giroux and Sentini, 1991] is a reflective OBCP based on ACTTALK, itself an extension of SMALLTALK. REACTTALK uses reflection to decompose the behaviour of an aggregate agent into organisations of objects to produce higher level adaptive agents. SLP shares with PROLOG and LISP an affinity for meta-interpretation. The simplest form of meta-interpreter, the *vanilla interpreter*, emulates the underlying computational model:

```
//exec(Queue_of_goals)
exec(done)<- done
+ exec(user(Goal) | Rest_Goals)
    <- clause(user(Goal), Body) | exec(Body)
        | exec(Rest_Goals).
```

where a clause of the form

```
head : -c1 & ... cn <- b1 |...| bm
```

is represented in the meta-interpreter by

```
clause(user(Head), Tail)
    :- c1 & ... cn
    <- Tail<<(user(b1) |... | user(bm) | done)
```

Tanaka and Matono [1992] describe a simple reflective extension of GHC (Guarded Horn Clauses, an FGI manifestation of an SLP) where a reflective tower can be constructed and collapsed dynamically.

4.1 Intelligent agents

The top level for a Shoham agent [1990] in a RGDC would be similar to the subservient agent described previously:

```
//agent0(Message_Stream, Mental_State)
agent0(Message:Stream, Mental_State)
    :- <conditions on messages and mental state>
    <-exec(Message, Mental_State, New_Mental_State)
    | agent0(Stream, New_Mental_State)
```

Simple commitment conditions on the messages received can be represented in the guard of a clause. Indeed, choosing which clause to reduce a goal is often called commitment in SLP. More complex commitment conditions on the message and the mental state are represented in the **exec/3** object (which is not defined here). It is similar the **transaction/3** object of the subservient agent above.

The **exec/3** behaviour may involve sending a message and or updating the mental state. The mental state could consist of a pair {Beliefs, Common_Beliefs} of the agents own beliefs and what it believes about other agents. The messages a Shoham agent can send are inspired by speech-act theory [Searle, 1969]. Sending an *inform* message could be coded as the object:

```
//inform(Time, Agent, Fact)
inform(time_up, Agent, Fact) <-
    Agent<<inform(Fact).
```

Here Agent is a message slot known to the recipient agent, possibly a reply slot. Rather than a realtime clock driving an agent, time is measured by events and events are the sending of messages. The message reception slots are monotonic and thus behave like a clock. A realtime clock is a concept, like inheritance, that is not well suited to distributed systems. Rather than regarding history as the passage of time, time is considered as the passage of history. If no events have taken place no time has passed. This is the philosophy of discrete event simulation. The Time message could be connected to a clock primitive **time/2**, described previously, but the present interpretation seems preferable.

5 Conclusions

The paper has attempted to illustrate that SLP has all the power of OBCP for MAS. The arguments in favour of SLP are the same as for OBCP [Gasser and Briot, 1992]. As with ACTORS, the process structure is small and therefore scalable. SLP can model fine entities such as neurones [Kozato and Ringwood, 1992]. This paper illustrates *buddin*g [Hewitt and Inman, 1992] in which an SLP object can transform itself into a network of objects, in this case a neural net. An agent can be built, not from a single long-lived object, but from a network of distributed objects that are created and destroyed dynamically. This accords with symbolic interactionist sociology [Mead, 1934] where agents are perceived as dynamic and evolving with many components.

There have been a number of papers comparing SLP and actor languages [eg Hewitt and Agha, 1988] but these have not recognised the ontological differnce. Rather than a message queue, SLP offers an array of message reception slots. The vital ingredient that leads to stream capability is that messages may contain message slots. Some actor languages such as ABCL [Yonezawa, 1990] provide two message queues with each object, but this doesn't compare with the flexibility afforded by message slots.

Rather than the conditional expression of ACTORS, SLP provides guards to discriminate received messages and choose between alternative behaviours. SLP guards provide a complex form of pattern matching. Guards are also found in ABCL [Yonezawa, 1990]. But, unlike SLP, ABCL can only specify constraints on a single message.

It can be the case that judging between languages is a matter of taste. Those trained or experienced in one paradigm feel that it is the most “natural” way to program or even write a specification. Others not used to that paradigm find it awkward and unnatural. The superficial difference in syntax between OBCP and SLP has the danger of degenerating into the LISP-PROLOG wars.

Proposals have been made for Actor-like syntaxes for SLP: Vulcan [Kahn et al, 86], Mandala [Ohki et al, 1987] and Polka [Davison, 1992]. However, one of the authors of Vulcan [Kahn, 1989], later recants on this tendency because of the loss of simplicity and flexibility. These criticisms are remarkably similar to the short comings of OBCP for DAI reported by Gasser and Briot [1992].

The essential distinction between ACTORS and SLP is the ontological status of objects. In ACTORS, objects are first-class entities. They may be assigned to variables and passed as parameters. In SLP, message slots are the first class entities that can be passed as parameters. However, with continuations, SLP objects and networks of objects can also be passed as messages.

Stream logic objects are only identified by the message slots on which they can receive messages. In this and other respects, SLP is similar to process algebras such as the pi calculus [Milner et al, 1989]. This again accords with symbolic-interactionist sociology [Mead, 1934] which proposes that interactions and not individuals are the primary units of analysis. Objects and message streams in SLP emerge from interaction and not vice-versa.

With OBCP it is generally recognised that the objects are too fine grained to be agents Aggregates of objects have been proposed as agents as in ORG [Hewitt and Inman, 1991]. The key problem with the common forms of OBCP is directing messages to agents which are aggregates [Giroux and Sentini, 1991]. Aggregates of SLP objects do not have this problem as the interface is determined by the message slots visible outside a network of objects.

Another advantage of SLP is the extreme simplicity of the language. It is even simpler than Actors. This simplicity it partly inherits from PROLOG. But SLP is much simpler than PROLOG: there is no backtracking, no unification and no negation as failure. Although the computational model is simple, condition synchronisation gives the ability to model all other forms of synchronisation. Even higher order functions can be specified in SLP [Reddy, 1994].

More recent languages, such as AKL [Janson and Haridi, 1991] or OZ [Henz et al., 1993], claim to offer all the advantages of SLP, plus those of PROLOG and constraint solving. Such languages have three or more computational mechanisms that make deadlock detection a nightmare. They do not offer anything that cannot be programmed in SLP. The unbounded automatic backtracking of PROLOG is of doubtful use in MAS. Distributed unification and constraint solving require substantial overhead in locking and deadlock avoidance and none of this is accessible to the application programmer.

Durfee [1988] recognises the common concerns of distributed operating systems and DAI and claims that they are merging technologies. As evidence of its capability, SLP has been used to implement a large distributed operating system, PIMOS, Parallel Inference Machine Operating System of the Japanese Fifth Generation Initiative. This machine has hundreds of processors.

Until recently, SLP has not been considered for building DAI systems. Linney [1993] uses a commercial SLP for multi-agent planning in a biomedical imaging system. Cohen and Ringwood [1993; 1994] show how to use SLP to combine independent databases into a distributed database. Apart from these examples, SLP has received no significant exposure in DAI. The authors contend that SLP is a potentially natural and useful basis for implementing MAS. The principle advantage over OBCP is the principle difference between them. Because message slots are the first class entities, there is no difficulty with referencing adaptive, dynamically evolving, multi-object, agents.

A commercial version of a distributed SLP language is STRAND88 [Strand Software Technologies Ltd, 1988]. The implementation offers an abstract machine that is interpreted in C. It runs on iPSC Hypercube, Sequent Symmetry and networks of Unix workstations. A public domain SLP is, KL1C, from the FGI. This version is not interpreted but compiles directly to C. This proves to be a real advantage for speed, portability and the interfacing of other software. KL1C [Chikayama, 1993b] is available by anonymous ftp from ICOT.

6 References

Abelson H, Sussman GY and Sussman J (1985) *Structure and Interpretation of Computer Programs*, MIT Press

Agha G (1990) Concurrent object-oriented programming, CACM **33**(9)125-41

Chikayama T (1993a) The Fifth Generation Project: personal perspectives, CACM **36**(3)82-90

Chikayama T (1993b) A KL1 Implementation for Unix Systems, New Generation Computing **12**, 123-4.

Cohen D, Huntbach MM and Ringwood GA (1992) Logical Occam, in Kacsuk P and Wise MJ (eds), *Implementations of Distributed PROLOG,* Wiley.

Cohen D and Ringwood GA (1993), Distributed databases tied with StrIng, *Advances in Databases*, LNCS **696**, Springer Verlag, 76-92

Cohen D and Ringwood GA (1994) New wine in old bottles, submitted IJIS

Colouris G, Dollimore J and Kindberg T (1993) *Distributed Systems: Concepts and Design* (2nd ed), Addison-Wesley

Davison A (1992) Object oriented databases in Polka, in Valduriez P (ed), *Parallel Processing and Data Management*, Chapman Hall, 207-23.

Dijkstra EW (1975) Guarded commands, nondeterminacy and formal derivation of programs, CACM **18**(8)453-7

Durfee EH (1988) *Co-ordination of Distributed Problem Solvers*, Kluwer Academic Publishers,

Ferber J and Carle P (1992) Actors and agents as reflective concurrent objects: a Mering IV perspective, IEEE Transaction on Systems, Man and Cybernetics, **21**(6)1420-36

Gasser L, Brazganza C and Herman N (1987) MACE: A flexible testbed for distributed AI research, in Huhns MN (ed) *Distributed Artificial Intelligence*, Pitman-Morgan Kaufmann, 119-52

Gasser L (1991) Social conceptions of knowledge and action: DAI foundations and open systems semantics, Artificial Intelligence, Special issue on the foundations of Artificial Intelligence

Gasser L (1992) Boundaries, aggregation and identity: Plurality issues for multi-agent systems, in Demazeau Y and Werner E (eds) *Decentralized Artificial Intelligence 3*, Elsevier

Gasser L and Briot J-P (1992) Object based concurrent programming in DAI, in Avouris NA and Gasser L (eds), *Distributed Artificial Intelligence: Theory and Praxis*, Kluwer, 81-107

Giroux S and Sentini A (1991) A distributed artificial intelligence approach to behavioural simulation, *Proc 1991 European Simulation Multiconference*, Copenhagen

Goldberg A and Robson D (1983) *Smalltalk-80, the Language and its Implementation*, Addison Wesley

Henz M, Smolka G and Wuertz J (1993) Oz - a Programming Language for Multi-Agent Systems, in IJCAI-93,.404-9.

Hewitt C (1969) PLANNER: a language for manipulating models and proving theorems in robots, Proc IJCAI 1, 295-301

Hewitt C and Agha G (1988) Guarded Horn clauses: are they deductive and logical? *Proc of 1988 Int Conf on Fifth Generation Computer Systems*, ICOT, 650-7

Hewitt C and Inman J (1991) DAI betwixt and between: from intelligent agents to open systems science, IEEE Transactions on Systems, Man And Cybernetics **21**(6)1409-19

Janson S and Haridi S (1991) Programming Paradigms of the Andorra Kernel Language, in Proc 1991 International Logic Programming Symposium, MIT Press.

Jennings NR (1994) *Cooperation in Industrial Multi-Agent Systems*, World Scientific Press.

Kahn G (1974) The semantics of a simple language for parallel programming, in Information Processing 74: Proc IFIP Congress **74**, 471-5

Kahn K (1989) Objects – a Fresh Look, in Cook S (ed), Proc ECOOP **89**, Cambridge University Press

Kahn K, Tribble ED, Miller MS and Bobrow DG (1986) Vulcan: Logical Concurrent Objects, OOPSLA-86 , SIGPLAN Notices **21**(11) - reprinted in Shapiro E (ed) *Concurrent PROLOG Volume 2*, MIT Press 1987, 274-303.

Kozato F and Ringwood GA (1992) How slow processes can think fast in concurrent logic, in Soucek B (ed) *Fast Learning and Invariant Object Recognition: the Sixth-Generation Breakthrough*, Wiley, 47-60

Lieberman H (1986) Using prototypical objects to implement shared behaviour in object-oriented systems, OOPSLA-86 , SIGPLAN Notices **21**(11)214-23

Linney J (1993) *Agent Architecture for Biomedical Imaging Systems*, MPhil Thesis, U. London

Maes P (1988) Computational reflection, Knowledge Engineering Review **3**(1)1-19

McCarthy J (1988) Mathematical logic in Artificial Intelligence, in Graubard SR (ed) *The Artificial Intelligence Debate*, MIT Press

Man T-L (1993) Real-time concurrent logic programming, PhD Thesis, Imperial College

Mead GH (1934) *Mind, Self and Society*, U Chicago Press

Metzner JR and Barnes BH (1977) *Decision Table Languages and Systems*, ACM monograph Series, Academic Press

Milner R, Parrow J and Walker D (1989) A calculus of mobile processes, TRs ECS-LFCS-89-85 and 86, LFCS, U of Edinburgh.

Ohki M, Takeuchi A and Furukawa K (1987) An object-oriented language based on the parallel logic language KL1, in Proc Fourth Int Conf on Logic Programming. MIT Press, 894-909.

Reddy (1994) Higher Order Aspects of Logic Programming, in Van Hentenryck P (ed) Logic Programming: Proc 11th Int Conf on LP, MIT Press

Ringwood GA (1988) Parlog86 and the Dining Logicians, CACM **31**, 10-25

Ringwood GA (1989) A comparative exploration of concurrent logic languages, Knowledge Engineering Review, **4**, 305-32

Ringwood GA (1994) A brief history of stream parallel logic programming, Logic Programming Newsletter **7**(2), 2-4

Searle JR (1969) *Speech Acts: An Essay in the Philosophy of Language*, CUP

Shapiro E and Takeuchi A (1983) Object-oriented programming in Concurrent Prolog, New Generation Computing **1**, 25-48

Shoham Y (1990) Agent-oriented programming, TR STAN-CS-1335-90, Standford University, Cal.

Smith RG (1980) The contract net protocol: high-level communication and control in a distributed problem solver, IEEE Transactions on Computers, 29:1104-113

Steiner DD (1995) IMAGINE: an integrated framework for constructing multi-agent systems, in O'Hare GMP and N. R. Jennings NR (eds) *Foundations of Distributed Artificial Intelligence* , (to appear) Wiley Interscience

Strachey C and Wadworth CP (1974) Continuations: a mathematical semantics for handling full jumps, TR PRG-11, Oxford University.

Strand Software Technologies Ltd (1988) *STRAND88 Reference Manual*

Sussman GJ, Winograd T and Charniak (1971) MicroPlanner Reference Manual, TR AIM-203A, MIT

Tanaka J and Matono F (1992) constructing and collapsing a reflective tower in Reflective Guarded Horn Clauses, Proc Int Conf Fifth Generation Computer Systems, ICOT

Thomas SR (1994) The PLACA agent programming language, in Wooldridge MJ and Jennings NR (eds) Proc ECAI Workshop on Agent Theories, Architectures and Languages , Amsterdam, The Netherlands, 307-20.

Tomlinson C and Scheevel M (1989) Concurrent object-oriented programming languages, in Kim W and Lochovsky FH (eds) *Object-Oriented Concepts, Databases, and Applications*, Addison Wesley

Winograd T and Flores F (1986) *Understanding Computers and Cognition*, Addison-Wesley

Yonezawa A (ed) (1990) *ABCL: An Object-Oriented Concurrent System*, MIT Press.