Agents for WIN-PROLOG

Programming Guide and Technical Reference
Version 1.0

Brian D Steel

CHIMERA 1.0

The contents of this manual describe the product, Chimera 1.0, and are believed correct at time of going to press. They do not embody a commitment on the part of Logic Programming Associates Ltd (LPA), who may from time to time make changes to the specification of the product, in line with their policy of continual improvement. No part of this manual may be reproduced or transmitted in any form, electronic or mechanical, for any purpose without the prior written agreement of LPA.

Designed and Written by Brian D Steel

The "wallpaper" used in the screen shots in this publication is based on the Willow Boughs design by William Morris (1834-96)

Logic Programming Associates Ltd
Studio 30
The Royal Victoria Patriotic Building
Trinity Road
London SW18 3SX
England

phone:	**+44 (0) 20 8871 2016**
fax:	**+44 (0) 20 8874 0449**
email:	**support@lpa.co.uk**
web:	**http://www.lpa.co.uk**

Welcome to Chimera: Agents for WIN-PROLOG

Table of Contents

Introduction

Welcome to Chimera, the *WIN-PROLOG* Agent Development Toolkit. This toolkit is entirely new, having been written from scratch with the help of the Windows Sockets (Winsock) predicates that were introduced in *WIN-PROLOG* 4.600; Chimera replaces the previous TCP/IP and Agent Toolkits, at once greatly extending the functionality while also simplifying the programming interface considerably.

Chimera Heuristic Intelligence Modelling Enhanced Relational Agents

Chimera provides a conceptually clean, enhanced relational architecture for building sophisticated multiple agent systems, which inherits and builds upon the advanced heuristic intelligence modelling of *WIN-PROLOG*.

What's in Chimera?

Chimera comprises three primary components: first, a system file, *CHIMERA.PC*, provides a set of high-level predicates which handle connection management, agent communication and event scheduling; next, a *KQML* parser allows *WIN-PROLOG* agents to communicate with those written in other languages, and finally, a set of example programs demonstrates the ease with which agents can be written and manipulated using this toolkit.

About this Manual

This manual explains the background to agents in general terms, before going into further details about how various aspects of agents are represented in Chimera and *WIN-PROLOG*; next, the individual predicates are discussed, with examples of calls, and then the shipped example programs are dissected in detail.

We hope you will find Chimera both fun and easy to use, and will take a few moments to read this document to gain an appreciation of the sheer power and total flexibility of this beautifully simple package.

Brian D Steel, 05 Jul 07

Chimera - Agents for WIN-PROLOG

Chapter 1 - All About Agents

This chapter explores the background to agents and agent-based systems, providing a basis for the more specific discussions in later chapters.

What is an Agent?

Perhaps the most fundamental issue regarding agents, namely the question, "What is an Agent?", is also the most controversial: almost as many schools of thought exist as there are researchers working in this particular field of computing. Despite this last assertion, there are certain properties that are exhibited by all agent-based systems.

Essentially, an agent is a self-contained software component, which is able both to offer services to other agents and/or request and use services of other other agents. In some respects, an agent can be thought of as a chimera comprising part "server" and part "client": however, this alone is insufficient to define an agent. In addition, an agent should exhibit some or all of the following properties:

Property	Meaning
Autonomy	An agent should be capable of running itself, without need of human intervention, for indefinite periods of time
Awareness	An agent should be aware of its environment, with the ability to discover other agents and initiate interactions with them
Communicativity	An agent should be able to communicate with other agents, sharing knowledge and services
Proactivity	An agent should be able to initiate operations, rather than simply respond to changes of situation
Reactivity	An agent should be able to react to changes in its

Chimera H I M E R A

	environment, modifying its behaviour accordingly
Cooperation	An agent should be able to cooperate with other agents, to help achieve their goals
Knowledge	An agent should have knowledge, for example a database or a set of "beliefs", which it can maintain and represent
Purpose	An agent should have a long term goal which defines its reason for existing, and can be used as an indication of its eventual success

As can be seen from the above, these concepts are somewhat woolly, and open to multiple interpretations, but they do help set the background to what an agent is.

Servers, Clients and Agents

In some respects, an agent can be thought of as a simple Server; consider, for example, a traditional HTTP (Web) server:

Property	**Meaning**
Autonomy	A web server can run without human intervention
Awareness	A web server can discover and execute scripts and other resources
Communicativity	A web server can communicate with browsers as well as other services, such as database engines or media streamers
Proactivity	A web server can push certain information to some clients
Reactivity	A web server reacts to incoming HTTP requests, running scripts and uploading static resources

Cooperation	A web server can cooperate with a browser, reading or writing cookies to help it perform its function
Knowledge	A web server "knows" the contents of its website; it also maintains background information in its log files
Purpose	A web server exists to expose and publish a web site

Distributed Objects

So why isn't an HTTP (web) server considered to be an agent? The answer to this is subtle, to do with shades of colour and perception. Agents are, in many ways, just servers, but they have a slightly different bent: their purpose is to be a component of a distributed computational environment; intelligent nodes in a knowledge network, able to contribute and assimilate knowledge, perhaps from sources unknown at the time of their creation. Agents might be able to advertise their knowledge, offering services to other agents; consequently, they should be able to detect and seek out those agents who are touting for business.

Agents are, in point of fact, conceptually similar to the "objects" of *OOPS* (*Object Oriented Programming Systems*), with the added property that they might exist anywhere on the network, rather than inside of some locall *DLL* (*Dynamic Link Library*) or somesuch.

A Chimera?

So what do we have at the end of this discussion? In a nutshell, an Agent can be thought of as a *chimera* that is part server, part client, with some of the conceptual properties of an object, and designed to be a component in a distributed computing network. This eclectic collection of properties is inspiration for the name of the present toolkit, somewhat more catchy than "***WIN-PROLOG*** Agent Development Toolkit". In *Chapter 2*, we'll look in more detail how these concepts are realised in Chimera.

Chapter 2 - Components of Chimera

In this chapter, we'll look in general terms at what is built into Chimera, and explore how they fulfill or enable the general requirements of agents as set out in *Chapter 1*.

Code, Comms and Events

In Chimera, an agent is comprised of three primary components:

Component	Explanation
Code	The core component of any agent is the program code which defines its function and behaviour, and in the case of Chimera, this is simply standard ***WIN-PROLOG*** code (or flex, Flint, etc).
Comms	In order for the code to be of use to other agents, "comms" (or communications) are an essential component of any agent, and Chimera makes this very simple
Events	Borrowing from the excellent message handling model used by ***WIN-PROLOG***'s window and dialog handling subsystem, Chimera reacts to events using an asynchronous *Event Queue* and user-defined *Event Handler*

The first of these components, Code, needs no introduction: any suitable ***WIN-PROLOG*** program can be used as the basis of an agent. The next two headings look in some more detail at the other components.

C Heuristic I M E R A

Comms: Communication Between Agents

Traditionally, comms (or communications) between any two software programs has been fraught with difficulty and endless unforseen problems; add to this the wish to communicate between multiple entities across networks, and most programmers would run a mile. Thankfully, in Chimera, communications are so simple that they can more or less be forgotten about.

When an agent is created, it creates a server which sits quietly in the background, monitoring for activity. If another agent attempts a connection, this is acknowledged and completed transparently, without any need for programming or user interaction. And if it is desired to connect proactively to some other agent, the only information required to complete the operation is the machine *Domain Name* (or *IP address*) where the intended correspondent is running, together with something called a *Port Number*.

Once a communication channel has been established, sending information between agents is as simple as constructing a Prolog term, and calling a predicate to send this term to the other agent. When all transactions are completed, connections can be closed equally easily.

Events: The Program Driving Force

There are effectively two ways in which a comms-based program can be written. In the first, it is implemented as a tight loop which continuously monitors some form of input port: when data arrives, it is processed, before the loop continues. The problem with this style of programming is that an application is generally wasting its time in between the arrival of data packets, when it could more usefully be doing something else (such as calculating the next thousand digits of Pi, or searching for signs of extra-terrestrial life in radio telescope data).

Which brings us to the second approach: Events. An *event-driven* program is just that: a program which is driven by events. When an event occurs, the program automatically freezes whatever it was doing, services the event, and then picks up where it left off as if nothing (other than the loss of a short timeslice) had happened. This is the model used by Chimera, and it greatly simplifies nearly all aspects of writing agents.

In the previous section about comms, we briefly mentioned that arbitrary Prolog terms could easily be sent to other agents: however, we did not explain how they could be received. The answer is simple: whenever such a term arrives, an event occurs, and no matter what else was happening at that point in time, control is passed to an *Event Handler*, defined by the user, which decides what to do. The event handler might, for example, update a local database, or perhaps initiate a calculation; it may schedule further events, either locally (in the same agent) or globally (in the agent which sent the original term, or in any other agent or agents that are currently connected).

Events, Terms and Messages

Digging a little deeper into Chimera, we discover that sending a term to another agent is synonymous with scheduling an event in that agent; likewise, scheduling an event in the local agent is simply a matter of "sending" a term to oneself. Because self-sending is a curious concept, Chimera refers to the sending of messages to anywhere as *Event Posting*, rather than event sending.

Suppose we wanted to raise an event called "hello", and include our name, say, "Fred Bloggs", as its arguments: effectively, all we need to do is post the Prolog term, hello(fred,bloggs) or, if we prefer, hello(`Fred Bloggs`) - it's up to us - to the desired agent. That agent will then receive the event, calling its event handler with this term as one of its arguments.

We can store absolutely any ***WIN-PROLOG*** data structure in one of these event terms, using all data types, including numbers, variables, Unicode text, lists, and so forth. Exactly the same term will arrive at the remote end (or locally in the case of a self-post), with the only observation being that any variables within the term will be new, and unconnected to the original ones. The reason for this limitation should be self-evident: it would be massively complicated (and horrendously slow) to allow Prolog unification pointers to connect across the Internet!

System Events

In addition to user events, the names and data components of which are entirely up to the programmer, Chimera defines a number of *System Events*. These occur when, for example, an incoming connection is automatically accepted, or an incoming event

has been received: in the latter case, this notification is in addition to the incoming event itself. There are eight distinct system events, and none need be processed by the user, unless it is specifically desired to do so. For example, an application might want to perform some local housekeeping, such as updating a "connection count" on a dialog, if the remote end of a connection disappears without warning: in order to do so, all the programmer needs to do is include a case for the "close" system event in the event handler.

Transport Syntax

A final component of Chimera is its provision of a *Transport Syntax* for ***WIN-PROLOG*** terms: effectively, such terms are written into a special binary form which can be reassembled into an exact copy of the term at the other end. This is all very well when the agents at both end of a link are written in ***WIN-PROLOG***; however, we may be connecting to and working with agents written in Java, C++ and other programming languages.

Chimera provides the necessary "escape" from Prolog, by allowing the reader/writer routine to be replaced on a per-agent basis. One such example, *KQML.PL*, is shipped as source code. This provides a simple reader/writer for the *KQML* (*Knowledge Query Manipulation Language*) system, which represents communications as S-Prolog expressions, and which has been used in agent research for a number of years. This reader/writer can be used in place of the default one simply by calling a predicate to notify Chimera of the user's preference.

In Summary

Chimera comprises a small, easily understood set of predicate which handles comms and events with simplicity, allowing users to define their own event regimes and protocols, and to utilise different transport syntaxes according to their needs. In *Chapter 3*, we'll look at the predicates themselves, and see how they work.

Chapter 3 - Chimera Predicates

In this chapter, we'll look in some detail at the predicates that are built into Chimera, refining the concepts discussed in *Chapter 2*, and exploring how they interact with reference to some simple examples of their use.

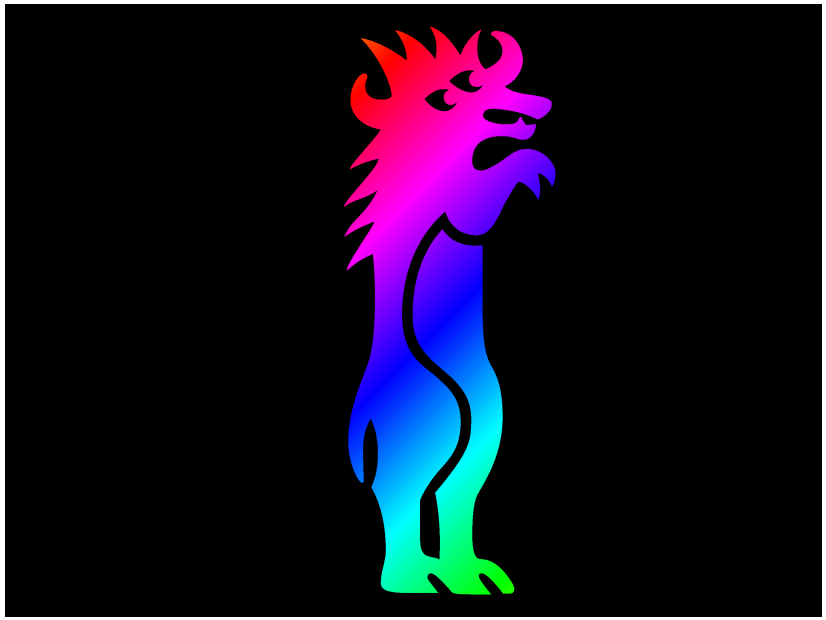
What is a Chimera Agent?

We asked a similar question in *Chapter 1*, but in that instance the discussion was more of a philosophical nature than practical. Here we will get down to the nuts an bolts, and start to work on some simple examples of how Chimera works in practice.

Strictly speaking, a Chimera agent comprises two components:

Component	Explanation
Socket	The comms part of an agent is handled by a *Windows Sockets* (*Winsock*) "*Socket*", which is a named entity relating to the Winsock predicates in ***WIN-PROLOG***, that supports *TCP/IP* network communications
Handler	The event part of an agent is handled by a user program, of arity 3, which is called automatically whenever an event occurs; in all other respects, this *Event Handler* is just an ordinary ***WIN-PROLOG*** program

When an agent is first created, the program must specify three items: the agent name, its handler predicate, and (optionally) a *TCP/IP Port*. The first name becomes that of the agent, and is also used to name the Winsock socket that handles that agent's incoming connections; the second name is simply that of the user's choice of event handler predicate. The final component, the TCP/IP Port, needs a little more explanation.



C H Intelligence M E R A

Winsock, Sockets and All That

Windows Sockets, or Winsock, is a potentially massive subject area in its entirety, and has evolved over more than two decades from the original *Berkeley Sockets* concepts that originated at the University of California at Berkeley in the early 1980s. It is beyond the scope of the current document to go deeply into the details of sockets, but a little background should prove helpful.

In short, Sockets (whether Berkeley or Winsock) provides a model wherein machines can set up connections across a network, and then pass data between themselves using a protocol called *TCP/IP* (*Transmission Control Protocol/Internet Protocol*). Sockets supports other protocols, such as *UDP* (*User Datagram Protocol*), but these do not concern us here.

Any given machine on a TCP/IP network, which includes any machine directly connected to the Internet, has a globally unique IP address, made up of four 8-bit numbers, and written in the widely seen "dot" notation. For example, the LPA web server currently sits at the address:

```
217.207.1.226
```

This number uniquely identifies the LPA web server, but has two problems associated with it. The first is human: it is hard to remember this type of number. The second is practical: should LPA decide to change its *ISP* (*Internet Service Provider*), its web server will move to a new IP address; moreover, the existing address might end up being allocated to someone else.

This is where a second method of identifying machines comes into play: the *DNS* (*Domain Name System*) provides a method of assigning a *Domain Name* to identify a given address. For example, the LPA web server is named:

```
www.lpa.co.uk
```

This name is again unique to LPA's web server, and when the name it is used to locate this machine, a "DNS lookup" is performed, effectively to convert the domain name into its corresponding IP address. An Internet-wide set of *DNS Servers* performs this task, and should LPA decide to change its machine's physical address, for example

as a result of changing its ISP, a simple notification to the domain name registration authorities is enough to get the DNS system to redirect the name, "www.lpa.co.uk" to the new address.

The Domain Name, leading to an IP Address, is just one part of addressing a machine. Because machines may want to run multiple services (for example, the LPA server supports Web, file transfer and email services), there needs to be a way of burrowing down to just one part of a machine. This is where *TCP/IP Ports* come in: each IP Address uniquely identifies a computer, but that computer can be connected to at any one of 65536 distinct "ports". Think of ports rather like TV channels: you might have several tuned into, say, the BBC, but each one picks up a different programme service.

All of which get us back to the point: when a Chimera agent is created, its third and final parameter is the number of the port where the agent should "listen" for incoming traffic. For a more detailed look at Windows Sockets (Winsock) and TCP/IP, see *Appendix P* in the ***WIN-PROLOG*** Technical Reference).

Time to Experiment

Enough background: let's start some experiments. To begin with, to avoid worrying about networking issues, we will run two agents side-by-side on a single machine: the only thing we need to know before we start is the local IP address. If you are running Windows 2000 or XP, you can find this out by starting a DOS box and typing the following command at the "C>" prompt:

```
C> ipconfig                                                    <enter>
```

In this and all other examples, only type the letters that have been highlighted in **bold**, and follow this by pressing any key or key combination bracketed in *<italics>*. The "ipconfig" command will display your local IP address as shown in *Fig 3.1*. If you are running Windows 98 or ME, use explorer to locate and run the program "winipcfg", which will display a dialog containing you local IP address, as shown in *Fig 3.2*. In both these figures, the local IP address is shown as:

```
192.168.1.2
```

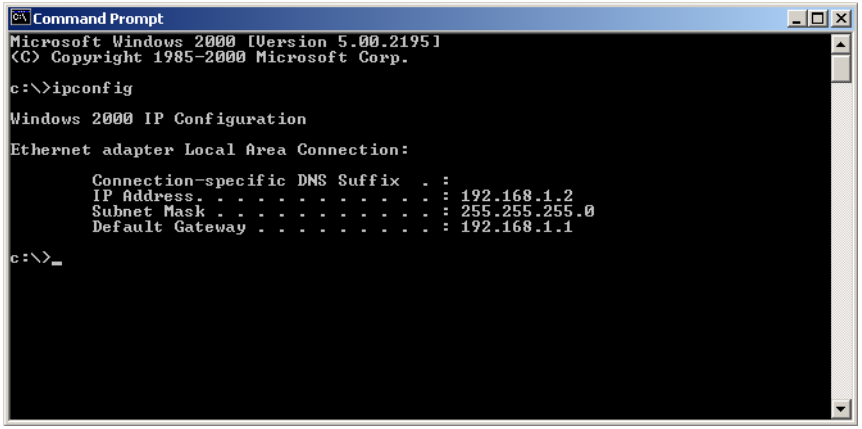


Fig 3.1 - The IPCONFIG command in Windows 2000

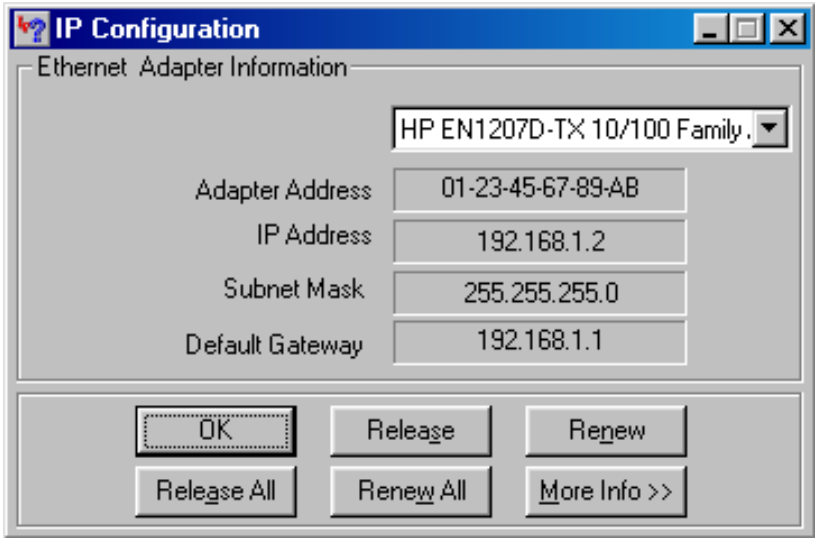


Fig 3.2 - The WINIPCFG command in Windows 98

This is the IP address we will be using in the examples that follow: you should substitute your own address when you try out these commands.

The Local Host Name: localhost

If your computer is not connected to a network, your IP address may be reported as "0.0.0.0": if this is the case, you should use the special host name, *localhost*, in place of an IP address when running the examples that follow. This name is active at all times, and serves as a convenient shortcut to your own machine's network address: however, it cannot be used to access or publicise services across a network, so it is best to get into the habit of using your actual IP address wherever possible.

To get a good feel for Chimera and its agents, start up two copies of ***WIN-PROLOG*** on a single computer, and arrange them above and below, as shown in *Fig 3.3*. Next, enter the following command into each one to ensure that Chimera system file is loaded, as shown in *Fig 3.4*:

```
| ?- ensure_loaded( system(chimera) ).                    <enter>
yes
```

Next, we'll define a very simple event handler which does nothing other then echo each event, as it occurs, to the console window. In the upper copy of ***WIN-PROLOG***, select "*File/New*" to create a new program window, and type in the following definition:

```
handler( Name, Link, Event ) :-
   writeq( Name - Link = Event ),
   nl.
```

The three arguments to our handler are the *Name* of the agent for which an event has occurred, a *Link* number which identifies an individual connection, and the *Event* itself. As we will see in a few moments, events can take two forms: *System Events* are generated automatically whenever connections are made or broken, or when terms are sent or received; *User Events* are generated specifically by calls to a predicate, *agent_post/3*, which is called explicitly. The two types of event can be distinguished by their *data type*:

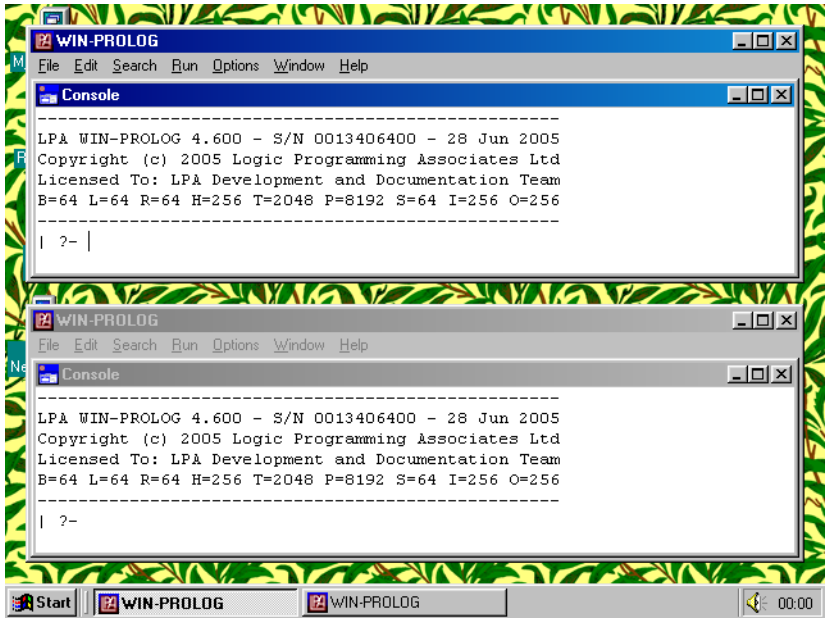


Fig 3.3 - Two copies of WIN-PROLOG running side by side

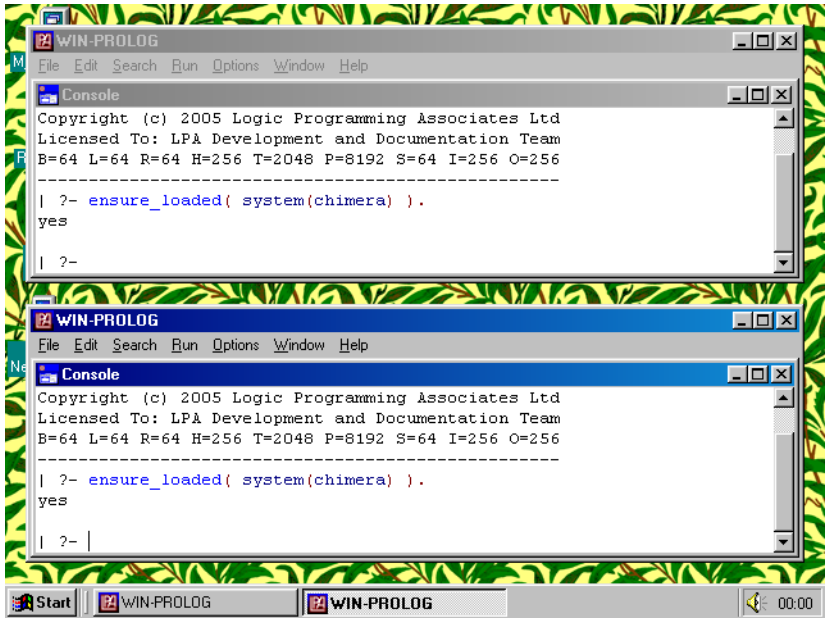


Fig 3.4 - Loading the Chimera system file into both copies

Data Type	Syntax	Meaning
Conjunction	(foo,...)	System Event
Tuple	foo(...)	User Event

System events can, by and large, be ignored by applications: Chimera functions perfectly well without any additional processing of such events. User events, on the other hand, are the fundamental building blocks of a Chimera application: there is simply no point in scheduling such an event if it cannot be handled by the recipient!

Our simple handler currently makes no distinctions between system and user events: it just displays a line of text for each event that occurs. We want this handler in both copies of ***WIN-PROLOG***, so save it from the upper copy by selecting "*File/Save*", and choosing (say) the file name, "HANDLER.PL", as shown in *Fig 3.5*. Next, go to the lower copy of ***WIN-PROLOG***, and open the file we've just saved. In both copies, select "*Run/Compile*" to compile the handler, and then use the "*Window*" menu to select the console window in each.

Creating the Agents

We are going to create an agent in each of the two copies of ***WIN-PROLOG***; first, in the upper copy, type the command:

```
| ?- agent_create( foo, handler, 2000 ).                    <enter>
yes
```

The first argument, "foo", is the name of our new agent; the second, "handler", is the name of the arity 3 program we've just written and compiled (ie, handler/3). The third argument is a *TCP Port* number: unless we are running this example on a machine that is already operating as a server, we can choose almost any number in the range 0..65535; having said that, it's usually best to avoid numbers in the lower range, 0..1023, which correspond to the so-called "*Well Known Ports*" used by the majority of TCP/IP services, such as HTTP (web, on port 80), FTP (file transfers, on port 21), SMTP (mail sending, on port 25), and so forth.

If you happen to choose a port number that's already in use on your machine, you

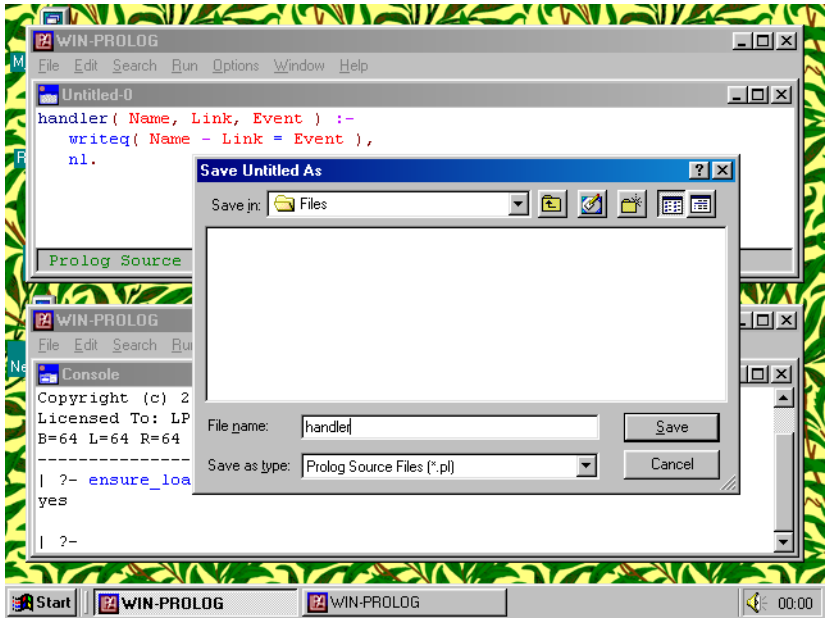


Fig 3.5 - Saving handler/3 as HANDLE.PL

will receive an error message: simply try other port numbers until one works. Assuming the above command worked first time, you will now see some additional output in the console:

```
| ?- foo - [] = (create,2000)
```

This output has been generated by handler/3, in response the system event, "*(create,Port)*", which occurs whenever an agent has been successfully created, returning its port number as shown in *Fig 3.6*.

If you are not too bothered about which port you want to use, and want to avoid the risk of clashes with existing ports, you can get *agent_create/3* to select a port number for you, simply by specifying a variable as the third argument. For example, go to the lower copy of ***WIN-PROLOG***, and enter this command:

```
| ?- agent_create( bar, handler, Port ).                    <enter>
Port = 49152
```

One of the so-called *Dynamic Ports*, which are those in the range 49152..65535, will be allocated on a first-available basis, and returned in the given variable, as shown above and in *Fig 3.7*. As before, our handler will generate some output at the console in response to the "*(create,Port)*" system event:

```
| ?- bar - [] = (create,49152)
```

Our two agents, "foo" in the upper copy of ***WIN-PROLOG***, and "bar" in the lower, are ready to communicate.

Establishing Communications

Staying for the moment with "bar", in the lower copy of ***WIN-PROLOG***, we are going to use *agent_create/4* to initiate a connection. Type the following command:

```
| ?- agent_create( bar, 0, `192.168.1.2`, 2000 ).           <enter>
yes
```

The first argument, "bar", is simply the name of the local agent which we want to

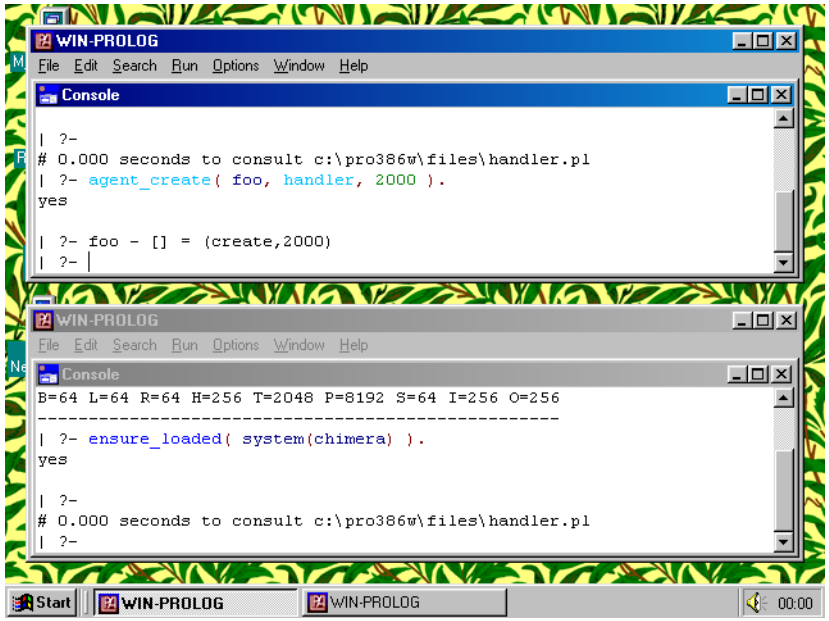


Fig 3.6 - A system event displayed by handler/3

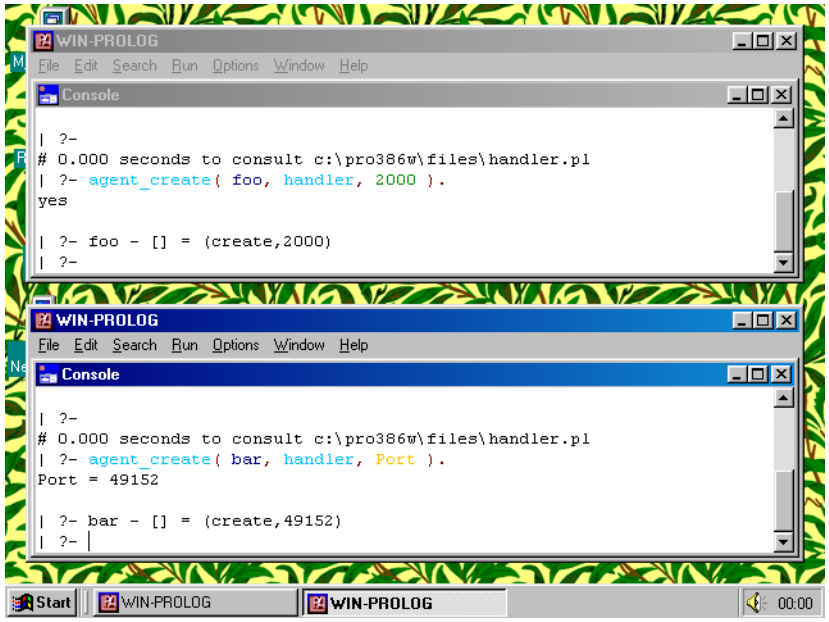


Fig 3.7 - Automatic assignment of a dynamic port

connect, and the second, "0" (zero) is a link number: this can be any integer, and is simply used together with the agent name (bar) to identify this particular connection. The third argument is a *WIN-PROLOG* string which contains the IP address of the remote agent, and the forth is the port number where we believe that agent is running. Please remember to substitute "192.168.1.2" with whatever address your "ipconfig" or "winipcfg" command returned (see above).

As the connection is made, another system event, "*(create,Addr,Port)*", occurs, as shown in the following output in the lower *WIN-PROLOG*'s console window:

```
| ?- bar - 0 = (create,`192.168.1.2`,2000)
```

Now for the interesting bit: the upper copy if *WIN-PROLOG* will have received another system event, "*(open,Addr,Port)*", resulting to the following output:

```
| ?- foo - 0 = (open,`192.168.1.2`,1025)
```

The port number, "1025", is selected automatically from the *Registered Ports*, the third and final set of port numbers, this time in the range 1024..49151. The two agents are now connected, ready to talk to each other, as shown in *Fig 3.8*.

Scheduling Events

The process of scheduling events is handled by a single predicate, *agent_post/3*; this predicate can be used to post any tuple onto the event queue of any connected agent as well as to a local agent, whether or not it is connected to others. From the upper copy of *WIN-PROLOG*, type the command:

```
| ?- agent_post( foo, 0, hello(world) ).                    <enter>
yes
```

The first argument is the local agent name, "foo", and the second is the link number ("0"): note that this was automatically assigned when the incoming connection, from "bar" in the lower copy of *WIN-PROLOG*, was accepted, and this was notified to our handler predicate, handler/3, as its second argument when the "*(open,Addr,Port)*" event occurred. As ever, our handler will by now have displayed some more output:

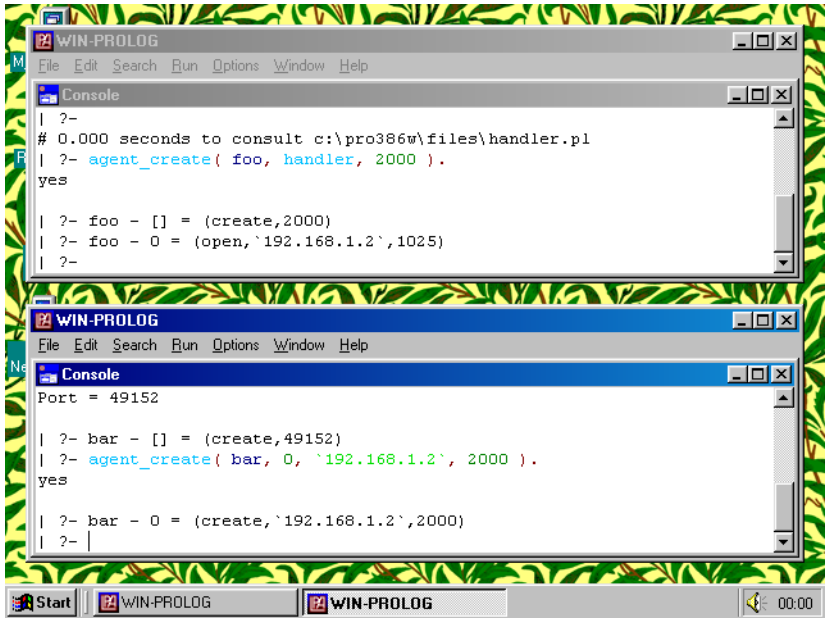


Fig 3.8 - Agents "foo" and "bar" connected

```
| ?- foo - 0 = (write,`192.168.1.2`,1025,hello(world))
```

This has resulted from the system event, "*(write,Addr,Port,Term)*", which will occur whenever an event is posted to another agent. Meanwhile, over in the lower copy of ***WIN-PROLOG***, agent "bar" will have received two events, indicated by some further output in its console window, as shown below and in *Fig 3.9*:

```
| ?- bar - 0 = (read,`192.168.1.2`,2000,hello(world))
| ?- bar - 0 = hello(world)
```

The first of these pieces of output relates to the system event, "*(read,Addr,Port,Term)*", which occurs whenever an event posting is received from another agent: it is the exact mirror of the "*(write,Addr,Port,Term)*" event we have just seen at "foo".

The second event, "hello(world)", is a user event, and simply corresponds to what we posted at "foo": absolutely any Prolog tuple can be posted as an event, allowing programmers to mix and match data types and structures as they see fit. Let's now reply by hand; in the lower copy of ***WIN-PROLOG***, type the command:

```
| ?- agent_post( bar, 0, bonjour(le,monde) ).          <enter>
yes
```

This posts the message "Hello World" (in French), back to whatever agent is connected to link "0" (zero) of agent "bar". The lower copy of ***WIN-PROLOG*** will display another "*(write,Addr,Port,Term)*" system event:

```
| ?- bar - 0 = (write,`192.168.1.2`,2000,bonjour(le,monde))
```

The upper copy of ***WIN-PROLOG*** will display both a "*(read,Addr,Port,Term)*" system event and the user event, "bonjour(le,monde)", as shown here and in *Fig 3.10*:

```
| ?- foo - 0 = (read,`192.168.1.2`,1025,bonjour(le,monde))
| ?- foo - 0 = bonjour(le,monde)
```

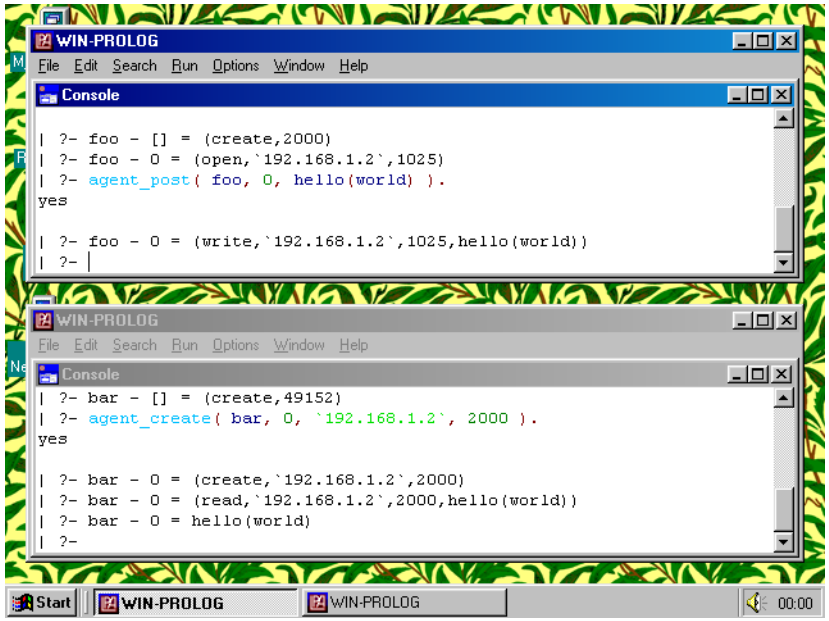


Fig 3.9 - A user event posted by "foo" to "bar"

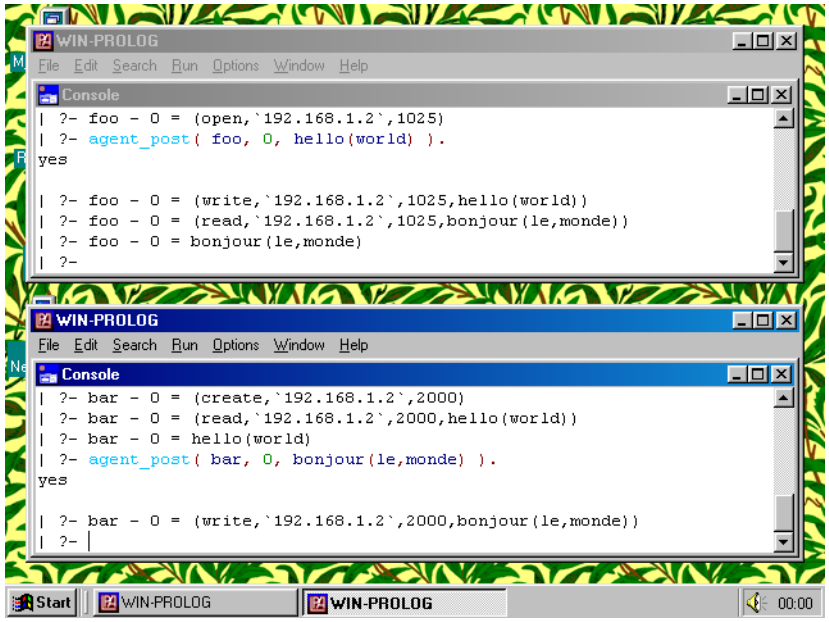


Fig 3.10 - A user event posted by "bar" to "foo"

Self-Scheduled Events

In addition to posting events to other agents, there will be times when it is desired to post an event to oneself, in order to schedule some activity when all other events have been fully processed. This is done with *agent_post/3*, just as before, except that the second argument, rather than specifying a link number, is set to the empty list, "[]". For example, in the upper copy of ***WIN-PROLOG***, type the following command:

```
| ?- agent_post( foo, [], calculate(22/7) ).          <enter>
yes
```

The user event, "calculate(22/7)", is posted to "foo" itself, rather than to some other agent connected to foo, resulting in a single piece of output from handler/3, as shown here and in *Fig 3.11*:

```
| ?- foo - [] = calculate(22 / 7)
```

Notice that no system event is associated with a self-posting: the only event that occurs is the specified user event; notice also how this example demonstrates the ease with which different data types can be included in events.

Closing Connections and Agents

Individual connections can be closed with *agent_close/2*, which shuts down just the one specified link; an entire agent, together with all its existing connections, can be closed using *agent_close/1*. In the upper copy of ***WIN-PROLOG***, close down "foo" and its one connection by typing the following command:

```
| ?- agent_close( foo ).                              <enter>
yes
```

This will initiate two system events, firstly "*(close,Addr,Port)*", which occurs whenever a single connection is closed, and secondly, "*(close,Port)*", which indicates that the agent itself is closing. Notice that the first of these events is posted as if from the link, specified as "0" (zero) in the second argument of handler/3, while the second is posted as if from the agent itself, specified by "[]" (empty list) as the link argument:

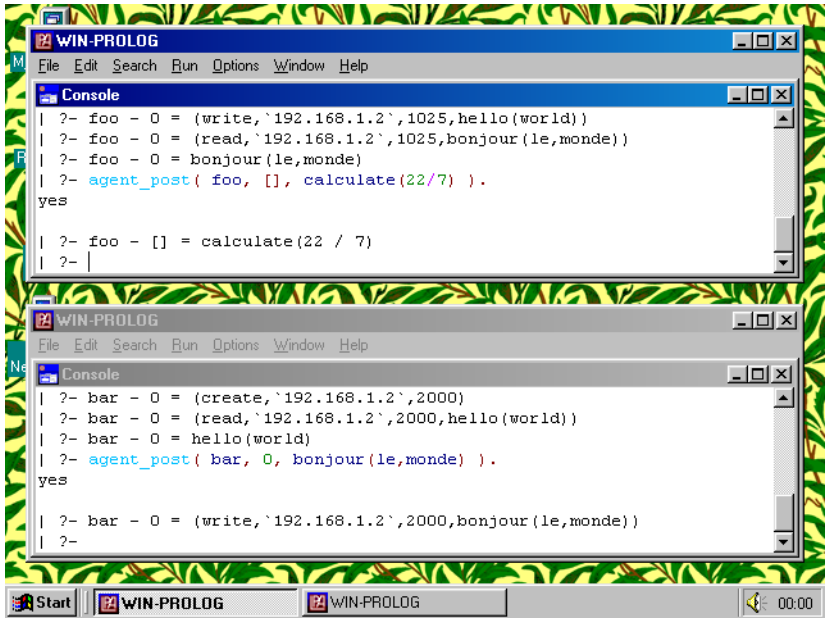


Fig 3.11 - A user event posted by "foo" to itself

```
| ?- foo - 0 = (close,`192.168.1.2`,1025)
| ?- foo - [] = (close,2000)
```

Meanwhile, over in the lower copy of ***WIN-PROLOG***, notice that "bar" has received a "*(close,Addr,Port)*" system event, indicating that its connection at link "0" (zero) has been closed. Now let's close "bar" itself, by typing this command into the lower copy of ***WIN-PROLOG***:

```
| ?- agent_close( bar ).                                    <enter>
yes
```

This will close the agent, resulting in just one outstanding "*(close,Port)*" system event, as shown below and in *Fig 3.12*:

```
| ?- bar - [] = (close,49152)
```

Interrogating Agents

As we have shown, agents written with Chimera are pretty much self-running: all that you need to do is to create them, and initiate one or more links, after which it is entirely up to you which events you generate and react to. Sometimes, however, you might want to find out more about what agents are running in any given instance of ***WIN-PROLOG***, and what connections have been made: a set of ancillary predicates is provided for this purpose.

Firstly, you can pick up a list of all existing agents using the predicate, *agent_dict/2*: this behaves just like the other "dictionary" predicates in ***WIN-PROLOG***, such as *wdict/2*, *timer_dict/2*, and so forth. The first argument specifies a flag, in the domain, [-1,0,1], which determines which types of agent names to return: all, unhidden or hidden respectively. You can find out more about unhidden and hidden atoms by looking at the definition of *hide/2* in the ***WIN-PROLOG*** Technical Reference: so far as we are concerned here, the normal value of this flag is "0" (zero).

Assuming you still have two copies of ***WIN-PROLOG*** running as before, and have closed down agents "foo" and "bar" as suggested above, let's start over. In the upper copy type the following command again:

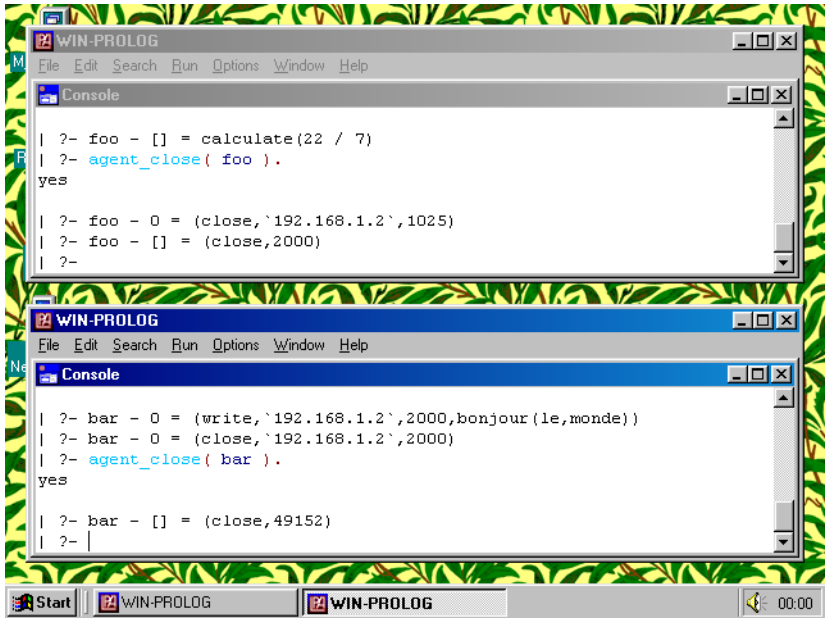


Fig 3.12 - System events after closing "foo" and "bar"

```
| ?- agent_create( foo, handler, 2000 ).                    <enter>
yes
```

You can save yourself some effort when re-entering commands, thanks to ***WIN-PROLOG***'s *Command History* mechanism: if you press <*ctrl-pgup*> repeatedly, you will cycle through previously entered commands; when you find the one you want, just press <*enter*> to resubmit it. You can, of course, edit the command first if you want to.

As before, our program, handler/3 will output a system event:

```
| ?- foo - [] = (create,2000)
```

Over in the lower copy of ***WIN-PROLOG***, either type in (or retrieve through the command history):

```
| ?- agent_create( bar, handler, Port ).                    <enter>
Port = 49152
```

Again, a spare dynamic port is assigned and returned for you to see, and the expected system event is reported:

```
| ?- bar - [] = (create,49152)
```

Now, just for variety, let's create two independent links to agent "foo": in the lower copy of ***WIN-PROLOG***, type the following command:

```
| ?- agent_create( bar, Link, `192.168.1.2`, 2000 ).        <enter>
Link = 0
```

Notice that this time, we've given a variable as the second argument to *agent_create/4*: more often than not, we don't really care what link number is used, since it's only a label; moreover, all events relating to this link pass the link number to our arity 3 event handler as its second argument. As expected, our handler/3 predicate reports a system event:

```
| ?- bar - 0 = (create,`192.168.1.2`,2000)
```

Now create the second link, using exactly the same command (pressing <*ctrl-pgup*> followed by <*enter*> will save you time):

```
| ?- agent_create( bar, Link, `192.168.1.2`, 2000 ).      <enter>
Link = 1
```

See how the next available link number is automatically assigned and returned, and also echoed by handler/3 during the next system event:

```
| ?- bar - 1 = (create,`192.168.1.2`,2000)
```

Let's return our attention to "foo", in the upper copy of ***WIN-PROLOG***: two system events will have occurred, as shown here and in *Fig 3.13*:

```
| ?- foo - 0 = (open,`192.168.1.2`,1026)
| ?- foo - 1 = (open,`192.168.1.2`,1027)
```

Notice that both the link numbers ("0" and "1") and port numbers ("1026" and "1027") have been generated automatically. Now we'll use *agent_dict/2* to list agents running; still in upper copy of ***WIN-PROLOG***, type the command:

```
| ?- agent_dict( 0, List ).                               <enter>
List = [foo]
```

As expected, we get back just one agent name, "foo". Next, let's see what we can find out about foo, using another predicate, *agent_data/4*; type the command:

```
| ?- agent_data( foo, Pred, Port, List ).                 <enter>
Pred = handler ,
Port = 2000 ,
List = [0,1]
```

The first argument is the name of the agent, "foo", which we can extract from the list returned by *agent_dict/2*; the remaining three arguments are variables that return, respectively, the *Pred* (event handler name), *Port* (TCP/IP port number) and a *List* of assigned link numbers. Using the latter argument, we can find out more information

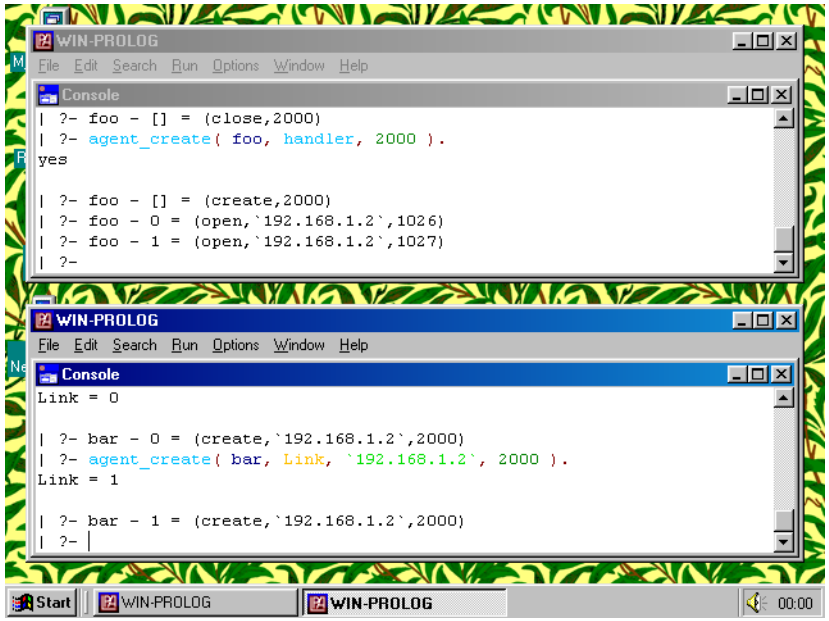


Fig 3.13 - System events following the creation of two links

about each individual link by calling agent_data/5; type the command:

```
| ?- agent_data( foo, 0, Sock, Host, Port ).          <enter>
Sock = bufnuaqc ,
Host = `192.168.1.2` ,
Port = 1026
```

This time we've specified the agent name ("foo") and one of its active link numbers ("0", zero); the remaining three arguments must be variables which return, respectively, the *Sock* (a random atom which names the Winsock "socket" that physically handles this connection), the *Host* (the IP address of the local machine, represented as a string) and the *Port* (an automatically assigned dynamic port number). You can type the following command to find out more about the other link, as shown below and in *Fig 3.14*:

```
| ?- agent_data( foo, 1, Sock, Host, Port ).          <enter>
Sock = hdlsdnlf ,
Host = `192.168.1.2` ,
Port = 1027
```

Automatic Shutdown

Normally, once agents are created and connected, it is "polite" to close them down upon completion. Having said that, if an application omits to close its agents before quitting, the ***WIN-PROLOG*** Windows Sockets (Winsock) support will perform a minimal shutdown sequence automatically. For example, enter the following command in the upper copy of ***WIN-PROLOG***:

```
| ?- halt.                                           <enter>
```

The upper copy of ***WIN-PROLOG*** will disappear, and the remaining copy will immediately echo a series of system events, as shown below and in *Fig 3.15*:

```
| ?- bar - 0 = (error,10054,sck_close)
| ?- bar - 0 = (close,`192.168.1.2`,2000)
| ?- bar - 1 = (error,10054,sck_close)
| ?- bar - 1 = (close,`192.168.1.2`,2000)
```

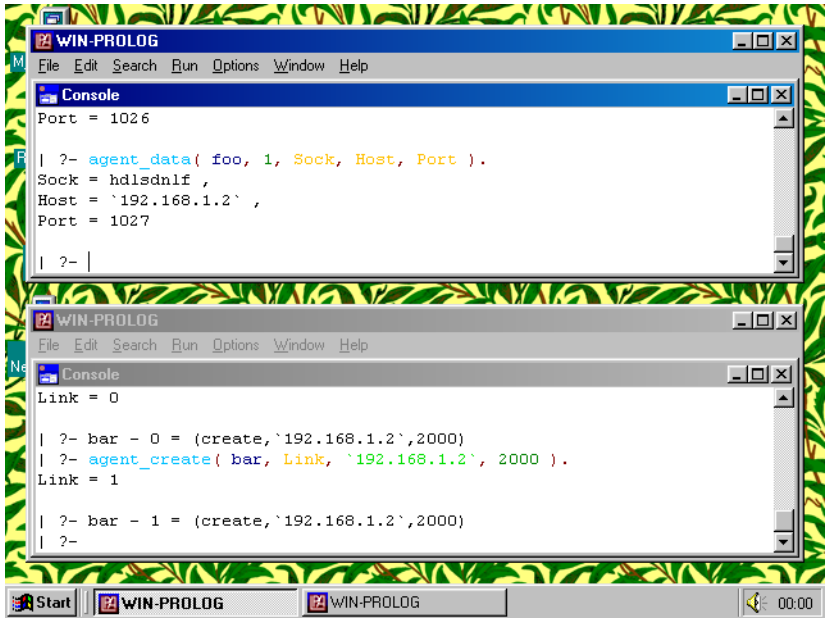


Fig 3.14 - Returning information about agents

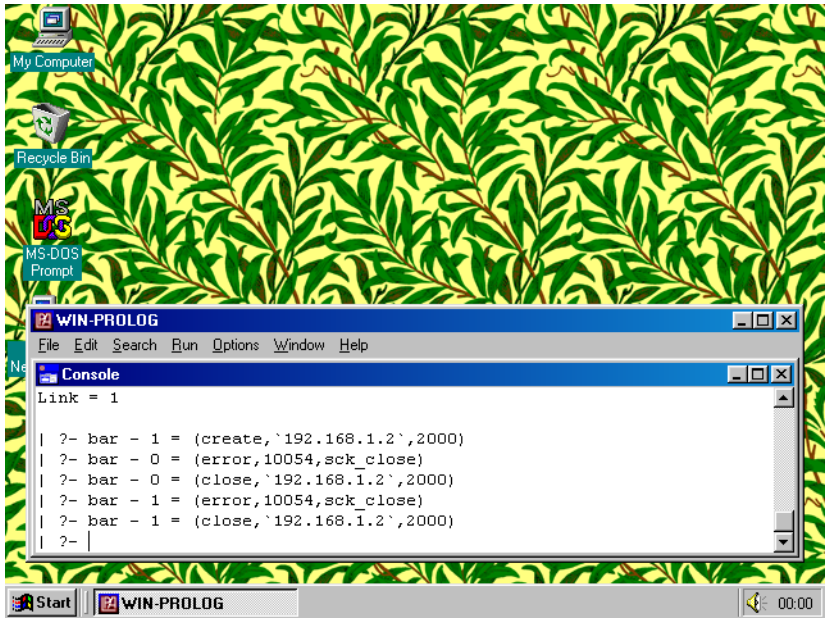


Fig 3.15 - System events following remote shutdown

This sequence shows a new system event, "*(error,Code,What)*", which is generated whenever a low-level error is encountered in Chimera communications protocols. Like all system events, it can generally be ignored or simply used for informational purposes, as here. The error number shown here, "10054", corresponds to the Win-sock error, "WSACONNRESET", which (as its name implies) states that a connection has been reset; the low-level event in which this error was detected was the socket-level "sck_close", which is used by Chimera as a trigger to close a link. Finally, let's shut down "bar" by typing the command in the remaining copy of ***WIN-PROLOG***:

```
| ?- agent_close( bar ).                                    <enter>
yes
```

As ever, a system event is reported, as shown below and in *Fig 3.16*:

```
| ?- bar - [] = (close,49152)
```

Version Control

Let's mention one final predicate, whose job is simply to display Chimera's version information; type this command into the remaining copy of ***WIN-PROLOG***:

```
| ?- agent_version( 1 ).                                    <enter>
-------------------------------------------------
Chimera 1.000 - Agents for WIN-PROLOG - 20 Jul 2005
Copyright (c) 2005 Logic Programming Associates Ltd
-------------------------------------------------
yes
```

A simple banner is displayed, as shown above and in *Fig 3.17*, which can be used to confirm which version of Chimera is installed on your system.

Conclusion

We have now completed our initial tour of and experimentation with the predicates of Chimera. In Chapter 4, we'll explore the event handler mechanism in greater detail, and see how it can be used to build interesting, distributed applications.

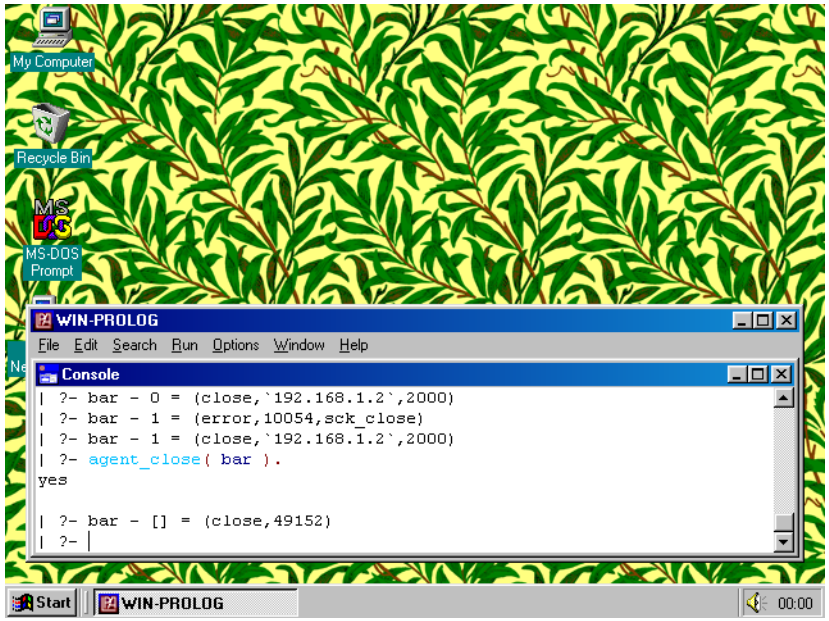


Fig 3.16 - Final system event after closing agent "bar"

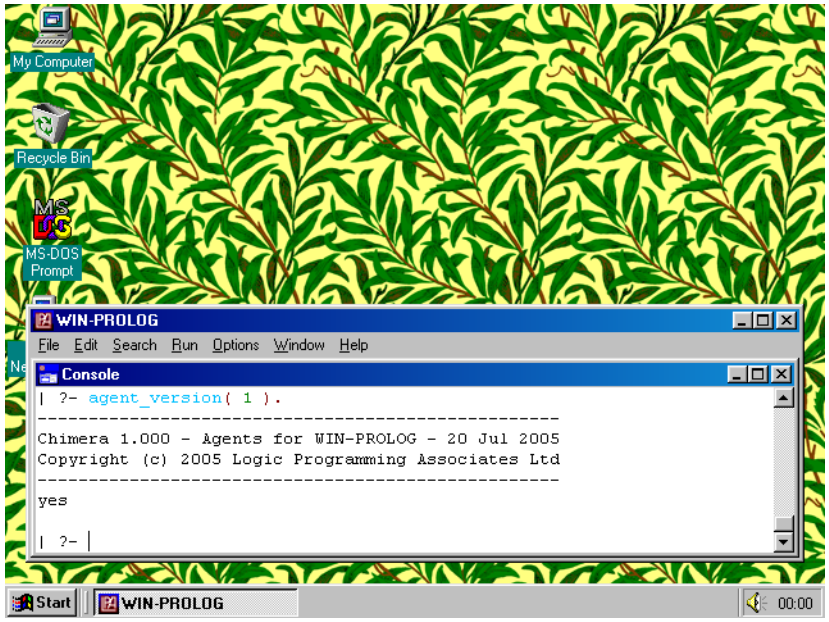


Fig 3.17 - Displaying Chimera's version banner

Chapter 4 - Events and the Agent Event Handler

In this chapter, we'll delve into the world of *Events* and *Event Handlers* as embodied in Chimera, showing how they interact with the predicates we discussed in *Chapter 3*, and developing some examples to demonstrate how to build interesting, distributed applications.

What is an Event Handler?

We've already had a look at a very simple event handler; in *Chapter 3*, we wrote and used a simple program, called handler/3, whose sole function was to display events, as they occurred, in ***WIN-PROLOG***'s console window. Let's look at this code again:

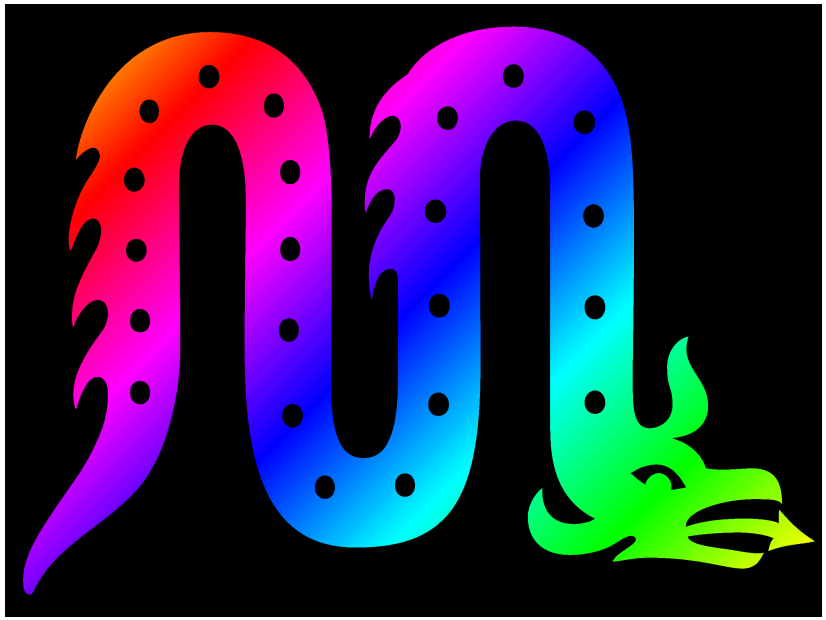
```
handler( Name, Link, Event ) :-
   writeq( Name - Link = Event ),
   nl.
```

This is a minimal example of an event handler, containing just one clause with all of its arguments specified as variables, and performing a simple output function. Because it does nothing else, this handler is effectively "transparent" to Chimera, functioning almost as a spypoint debugger; and yet, throughout Chapter 3, we saw a series of *System Events* that occurred whenever we communicated between agents or made or broke connections, all of which we ignored completely other than to echo them to the console window for illustrative purposes.

What this tells us is that Chimera can function fully without complex setup sequences or programming on our part: the baseline handler that we provide needs to do nothing special; in fact, nothing at all.

User Events as the Application

So here we are, with an event-driven programming environment in which we have no prescribed duties: then what is it that comprises an application? The answer, of course, is *User Events*.



C H I Modelling E R A

In *Chapter 3*, we created some ad-hoc user events when we posted messages such as, "hello(world)", "bonjour(le,monde)" and "calculate(22/7)" between and within our agents, "foo" and "bar". Because our event handler, handler/3, was just a vestigial piece of output code, these events had no particular meaning, and yet it is in the interpretation of such events, together with the consequential posting of additional events, that we will begin to construct our agents and distributed applications.

Consider the first event above, "hello(world)": this consists of a standard ***WIN-PROLOG*** *Tuple* (compound term), whose *Functor* is the atom, "hello", with the single argument, the atom, "world". Now imagine for a moment that you wrote a simple Prolog program, say:

```
reply_to( hello(X) ) :-
   write( welcome(X) ),
   nl.

reply_to( goodbye(X) ) :-
   write( farewell(X) ),
   nl.
```

You would be able to call this program using "hello(world)" as an argument:

```
| ?- reply_to( hello(world) ).                    <enter>
welcome(world)
yes
```

As shown above, this call "selects" the first clause in our simple program, resulting in the output of the text, "welcome(world)". The tuples that comprise user events have exactly the same purpose, allowing the appropriate case of the event handler to be selected.

Anatomy of an Event Handler

The above program is incredibly simple, day-one Prolog material; however, it makes a useful comparison to an event handler: let's start to experiment. Start up a copy of ***WIN-PROLOG***, and enter the following command to ensure that Chimera system file is loaded, as shown in *Fig 4.1*:

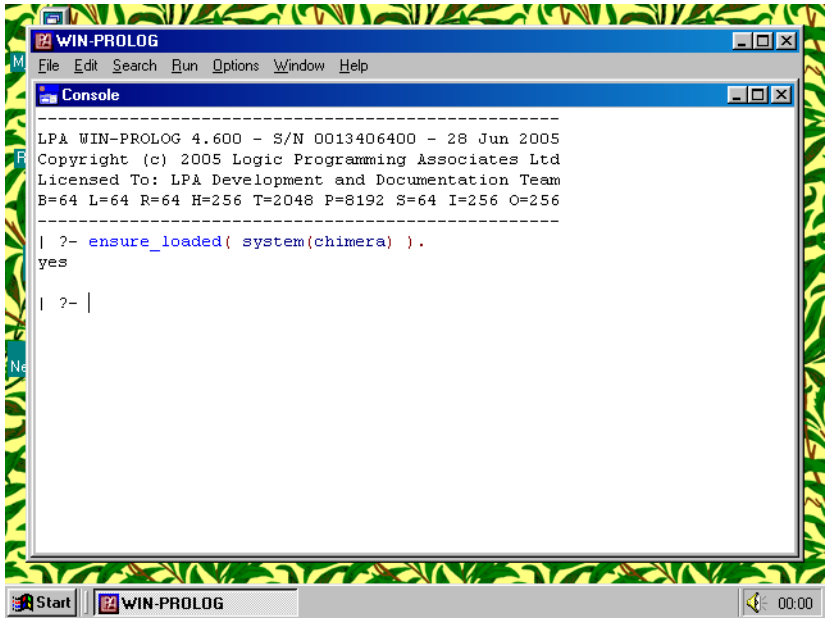


Fig 4.1 - Loading Chimera system file

```
| ?- ensure_loaded( system(chimera) ).                    <enter>
yes
```

Next, use "*File/New*" to create a new program window, and type in the code for foo_handler/3, as shown below and in *Fig 4.2*.

```
foo_handler( Name, Link, hello(X) ) :-
   write( Name/Link = welcome(X) ),
   nl.

foo_handler( Name, Link, goodbye(X) ) :-
   write( Name/Link = farewell(X) ),
   nl.
```

Select "*Run/Compile*" to compile the program, and then use the "*Window*" menu to get back to the console window.

This program is essentially identical to reply_to/1 (see above): all we have done is to rename the predicate, and add two new arguments (Name and Link) before the tuple and in the calls to *write/1*. You can call this code in very much the same way as before; type the command:

```
| ?- foo_handler( a, 0, hello(world) ).                  <enter>
a / 0 = welcome(world)
```

The extra values we passed in ("a" and "0" (zero)) are shown, together with our "welcome(X)" response, as shown in *Fig 4.3*.

So, if an event handler is really just a standard Prolog program, then what makes it special? The answer lies in part with those two additional arguments, and even more so with the event posting predicate, *agent_post/3*. To see this in action, let's create an agent, "foo", which uses this event handler. Type the command:

```
| ?- agent_create( foo, foo_handler, Port ).             <enter>
Port = 49152
```

Our new agent will be created at the first available dynamic TCP/IP port, in this case,

Fig 4.2 - Defining foo_handler/3 in a new program window

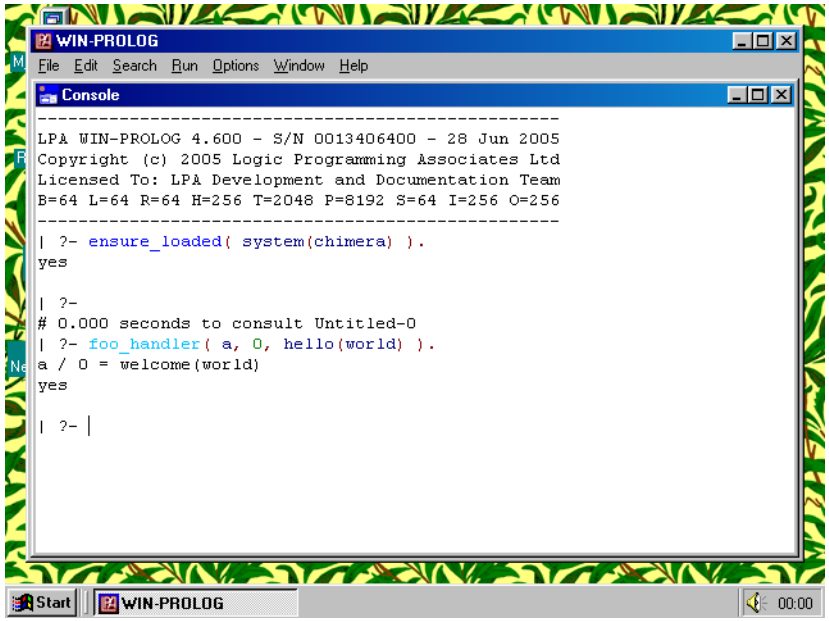


Fig 4.3 - The result of calling an event handler directly

49152. For now we can ignore this port number; instead, let's just post a message to "ourself", with the command:

```
| ?- agent_post( foo, [], hello(world) ).                    <enter>
yes
```

The first argument to *agent_post/3* simply specifies the name of the agent which is to accept the posting ("foo"), while the second specifies the link, or connection. If we specify this as an empty list ("[]"), the event is posted directly to the agent itself; if we specify an integer link number, then the posting is made to whichever agent is at the other end of the connection identified by that link: more on that later.

Notice that although we've posted the event to ourselves (well, to "foo"), it is not executed on the spot: ***WIN-PROLOG*** has returned with "yes" before any output took place, and at some time a fraction of a second later, the output arrives as shown below and in *Fig 4.4*:

```
| ?- foo / [] = welcome(world)
```

Notice also that when the event is finally handled, the Name and Link parameters have been bound to "foo" and "[]" (empty list) respectively, indicating the agent name/link to whom the original event was posted: again, more on this later.

Delayed Execution

The delayed execution that we have just witnessed is one of the key features of events handling, whether these are posted to oneself (as above) or other connected agents (see later). Effectively, events can be thought of as fragments of execution which do not need to be done right now, but which can be handled at some later stage, either locally or somewhere else across a network.

An important consideration of delayed execution is that there is no possibility of unifying variables in the event passed in to *agent_post/3*. In the "Untitled-0" program, enter a third clause to foo_handler/3:

```
foo_handler( Name, Link, return(Name=Link) ) :-
   write( returning(Name=Link) ),
```

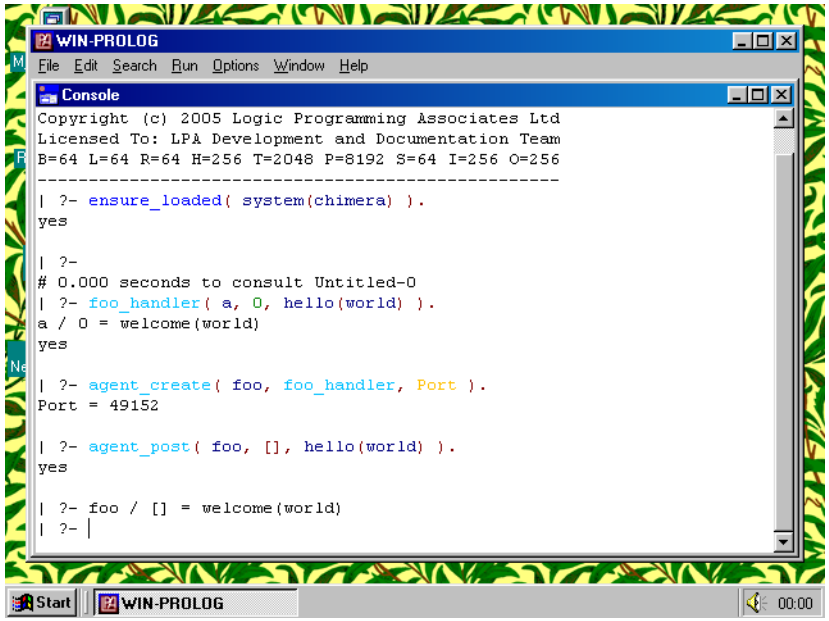


Fig 4.4 - The result of posting an event to ourselves

```
        nl.
```

Once again, compile the program ("*Run/Compile*") and return to the console window. Now try the direct call:

```
| ?- foo_handler( a, 0, return(X) ).                    <enter>
returning(a = 0)
X = a = 0
```

Notice how the output from our program occurs immediately, and how the variable, "X", is bound to the term "a = 0", as shown in *Fig 4.5*. Now try the following command:

```
| ?- agent_post( foo, [], return(X) ).                  <enter>
X = _
```

This time, no output occurs before our command returns, and the variable, "X", remains unbound. Again, the output appears subsequently, when the event has been dispatched, as shown below and in *Fig 4.6*:

```
| ?- returning(foo = [])
| ?-
```

Asynchronous Processing and Unidirectional Unification

The delayed execution mechanism used by *agent_post/3* leads to applications performing *Asynchronous Processing*: different portions of the application might be occurring in virtually any order. Additionally, variables in a handler can be bound to non-variables in posted events, but not vice versa: effectively, *Unidirectional Unification*.

Immediately this poses the question: how can values be returned by posted events? The answer is a short one: they can't! Instead, when an agent's event handler has processed an event, it "responds" by posting one or more events of its own.

In the simplest case, a new event can be posted directly back to the originator of whichever event is currently being handled: this is where the Name and Link parameters come in. Return to the "Untitled-0" window, and replace the old code with the

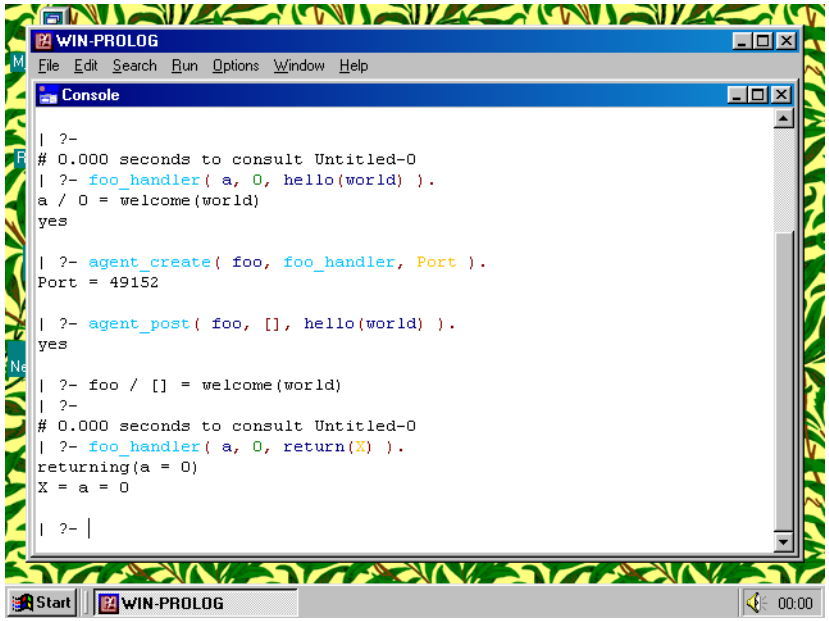


Fig 4.5 - Unification during a direct call to an event handler

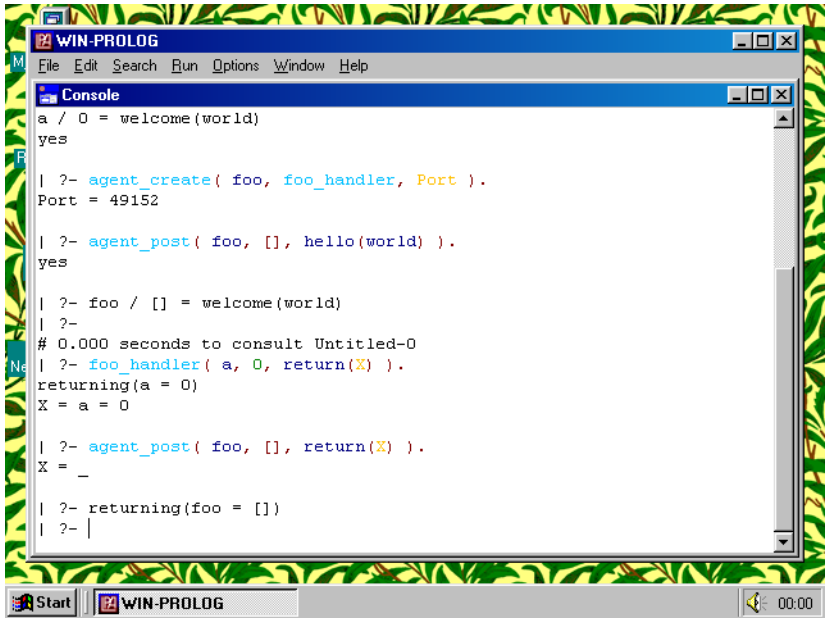


Fig 4.6 - Lack of unification in a posted event

following, revised definition for foo_handler/3:

```
foo_handler( Name, Link, hello(X) ) :-
   agent_post( Name, Link, write(welcome(X)) ),
   nl.

foo_handler( Name, Link, goodbye(X) ) :-
   agent_post( Name, Link, write(farewell(X)) ),
   nl.

foo_handler( Name, Link, return(Name=Link) ) :-
   agent_post( Name, Link, write(returning(Name=Link)) ),
   nl.

foo_handler( Name, Link, write(Term) ) :-
   write( Term ),
   nl.
```

Once again, compile the program and return to the console window. If we re-enter one of earlier commands (remember, you can use *<ctrl-pgup>* to find it and *<enter>* to re-submit to save typing), we will get similar behaviour to before, as shown below and in *Fig 4.7*:

```
| ?- agent_post( foo, [], hello(world) ).                <enter>
yes

| ?-
| ?- welcome(world)
```

Again, the output is delayed, and you might have noticed the additional prompt, "| ?-", before the output, and there's a simple explanation: the "hello(world)" case no longer performs output directly; instead its uses *agent_post/3* to post a "write(...)" event to whoever posted the original event. This new event is dispatched in turn, matching the final clause of our revised definition, and generating the output.

Now let's add a single clause to the "Untitled-0" window for a new event handler, bar_handler/3, which is based on our old handler/3 code (see *Chapter 3*):

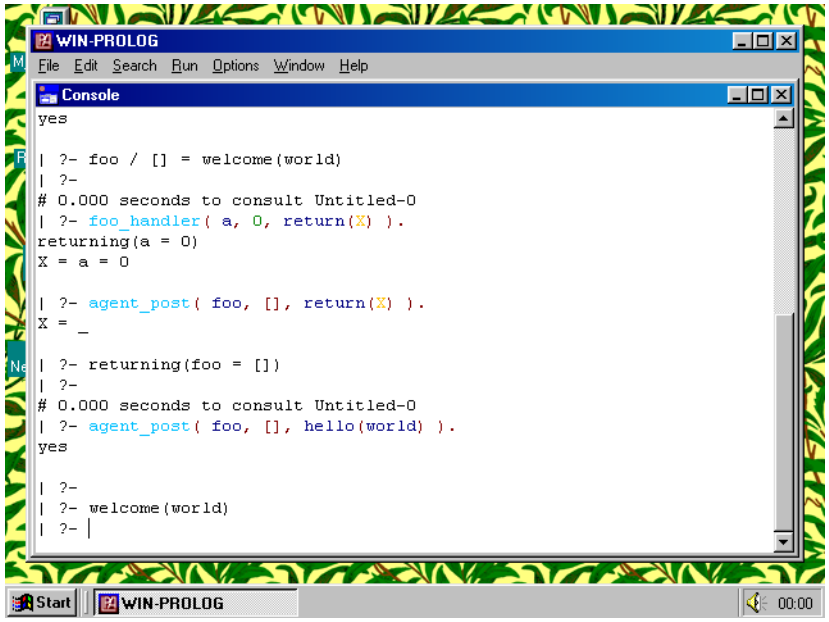


Fig 4.7 - Result of posting an event to the revised handler

```
bar_handler( Name, Link, User(Data) ) :-
   writeq( Name - Link = User : Data ),
   nl.
```

The main difference between this program and the old code is that we have replaced the third argument, which used to be a variable, with the tuple, "User(Data)": this means that this handle will only display user events, and moreover, only those events with a single data argument. Once again, compile the program, and return to the console window, before creating a new agent, "bar":

```
| ?- agent_create( bar, bar_handler, Port ).          <enter>
Port = 49153
```

Once again, notice that a dynamic port number is automatically assigned: we now have two agents "foo" and "bar", running simultaneously in a single copy of ***WIN-PROLOG***, on ports 49152 and 49153 respectively, and using the event handlers, foo_handler/3 and bar_handler/3 respectively. Let's create a link from "bar" to "foo": in order to do this, you need to know your machine's IP address (by running *IPCONFIG* or *WINIPCFG* as described in *Chapter 3*) or your machine's domain name. Assuming your machine is running at IP address, "192.168.1.2", and that "foo" is running on port 49152, enter the command:

```
| ?- agent_create( bar, Link, `192.168.1.2`, 49152 ).    <enter>
Link = 0
```

Agent "bar" should now have made a connection to "foo", identified as link number "0" (zero), as shown in *Fig 4.8*. Let's post an event from "bar" to "foo" using this connection:

```
| ?- agent_post( bar, 0, hello(world) ).                 <enter>
yes
```

This command posts the event, "hello(world)", to whatever agent is connected to "bar" at link "0" (zero). This message is received by agent "foo", which in turn posts the event, "write(hello(world))" back whichever Name/Link sent it the original event, which in this case is "bar". The end result is that bar_handler/3 receives the message, and writes it out to the console, as shown in here and in *Fig 4.9*:

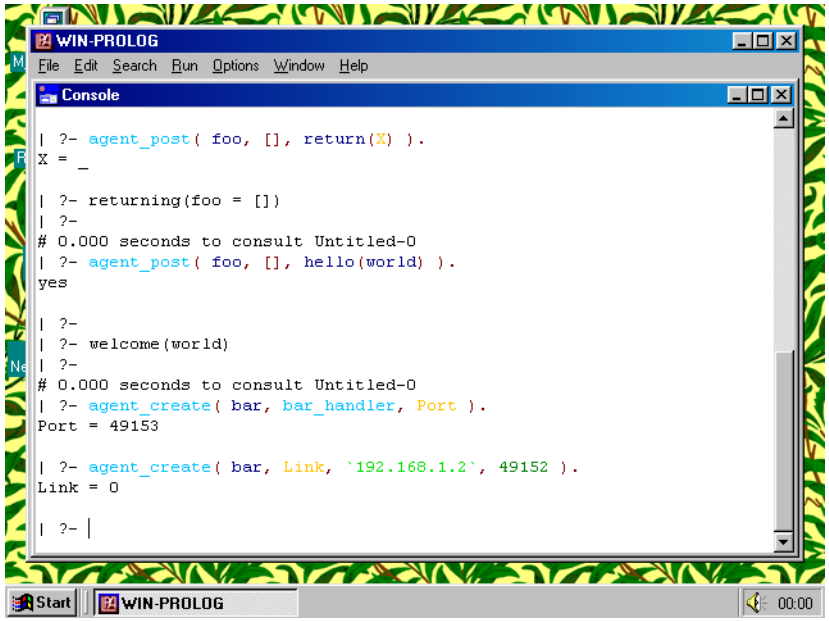


Fig 4.8 - Linking the agents, "foo" and "bar"

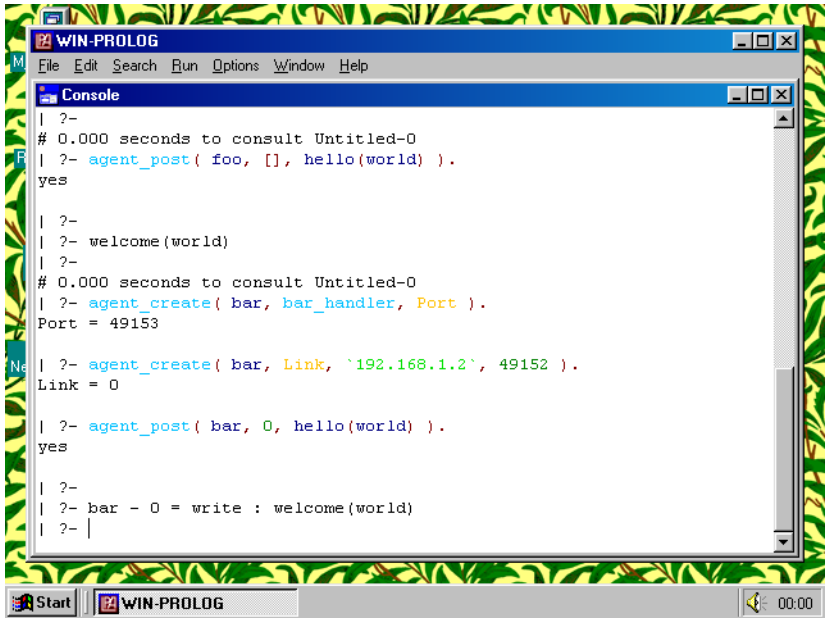


Fig 4.9 - Communications between agents "foo" and "bar"

```
| ?-
| ?- bar - 0 = write : welcome(world)
```

The Brains of the Matter

What we have just seen, for the first time, is how two agents communicate through their event handlers and *agent_post/3*: now we'll look at a slightly more interesting example. Select "*File/Open*", and navigate to the **WIN-PROLOG** "*EXAMPLES\CHIMERA*" directory to locate and load the file, *BRAINS.PL*. This contains a light-hearted agent which can perform two functions: mathematics evaluation and English/French translation, and its code is explained here.

```
/*
   Agent Brains - Brian D Steel - 07 Jul 05 / 20 Jul 05
   ====================================================

   Agent "brains" is a very simple demonstration of event
   handlers and message scheduling, which can be run locally
   on one machine or remotely across a network. To run
   "brains", type:

      ?- brains.

   This sets up the agent, and displays some information which
   you can use to connect to this agent from elsewhere.
*/
```

Before we get to work, let's make sure we've loaded up Chimera system file:

```
:- ensure_loaded( system(chimera) ).
```

The first clause sets up the agent, "brains", specifying the event handler, "thinker", and displays a short welcome message which includes the dynamic port that was allocated:

```
% create agent "brains" on a dynamic port and display details
```

```
brains :-
   agent_create( brains, thinker, Port ),
   write( `Welcome to Brains!~M~J` ),
   write( `==================~M~J~M~J` ),
   write( `Brains is running at Port: ` ),
   write( Port ),
   write( `~M~J~M~J` ).
```

The event handler, thinker/3, begins with a clause that processes the user event, "calculate(Maths)", which it processes by outputting some information to the console window, before calling *is/2* to perform the calculation, and posting a new event, "result(Maths=Result)", to whichever Name/Link posted the original event:

```
% "brains" has been asked to calculate some maths

thinker( Name, Link, calculate(Maths) ) :-
   write( `~M~JBrains is calculating ` ),
   write( Maths ),
   write( `~M~J` ),
   Result is Maths,
   agent_post( Name, Link, result(Maths=Result) ).
```

The next clause in our event handler reacts to the user event, "english(English)"; again, it outputs some information to the console window, before calling a predicate, translate/2, to convert the English text into French. If successful, a new event, "translation(English,French)", is posted back to the originator; otherwise, the event, "translation(English,???)" is posted instead, indicating that the translation could not be completed:

```
% "brains" has been asked to translate english to french

thinker( Name, Link, english(English) ) :-
   write( `~M~JBrains is translating the English, ` ),
   write( English ),
   write( `, into French~M~J` ),
   (  translate( English, French )
   -> agent_post( Name, Link, translation(English,French) )
```

```
   ;  agent_post( Name, Link, translation(English,???) )
   ).
```

Now we have a clause which processes the user event, "french(French)"; this is just like the previous clause, except that it calls translate/2 "back-to-front", to convert the French text into English. If successful, a new event, "translation(English,French)", is posted back to the originator; otherwise, the event, "translation(???,French)" is posted instead, again indicating that the translation could not be completed:

```
% "brains" has been asked to translate french to english

thinker( Name, Link, french(French) ) :-
   write( `~M~JBrains is translating the French, ` ),
   write( French ),
   write( `, into English~M~J` ),
   (  translate( English, French )
   -> agent_post( Name, Link, translation(English,French) )
   ;  agent_post( Name, Link, translation(???,French) )
   ).
```

The "result(Maths=Result)" event is posted by the handler clause for "calculate(Maths)", and this clause simply outputs data to the console:

```
% "brains" has been given the result of some maths

thinker( Name, Link, result(Maths=Result) ) :-
   write( `~M~JBrains says:` ),
   write( `~M~J~IThe result of ` ),
   write( Maths ),
   write( ` is ` ),
   write( Result ),
   write( `~M~J` ).
```

The "translation(English,French)" is also posted by this handler, in this case by one or other of the "english(English)" or "french(French)" cases, and again, this clause simply outputs data to the console:

```
% "brains" has been given the result of a translation

thinker( Name, Link, translation(English,French) ) :-
   write( `~M~JBrains says:` ),
   write( `~M~J~IThe English phrase: ` ),
   write( English ),
   write( `~M~J~Iis like the French: ` ),
   write( French ),
   write( `~M~J` ).
```

Translation programs don't come much dummer than this one. All we are doing here is working through two lists of words, attempting to match them one at a time to our vocabulary. Grammar? In your dreams! However, thanks to the magic of Prolog, this program can work forwards (English to French) or backwards (French to English):

```
% a really simple english/french translator (no grammar!)

translate( [], [] ).

translate( [Word|Words], [Mot|Mots] ) :-
   vocabulary( Word, Mot ),
   translate( Words, Mots ).
```

Moreover, vocabularies don't come much smaller than this, but it's just about enough to get a drink politely, and avoid being fed nasty sulphurous vegetables:

```
% an equally simple english/french vocabulary (twelve words!)

vocabulary( hello, bonjour ).
vocabulary( thankyou, merci ).
vocabulary( please, svp ).
vocabulary( goodbye, au_revoir ).
vocabulary( sir, monsieur ).
vocabulary( madam, madame ).
vocabulary( and, et ).
vocabulary( i, je ).
vocabulary( want, veux ).
```

```
vocabulary( hate, deteste ).
vocabulary( wine, le_vin ).
vocabulary( cabbage, le_chou ).
```

Compile this program, and once again switch to the console window, before typing the command:

```
| ?- brains.                                                    <enter>
Welcome to Brains!
==================

Brains is running at Port: 49154

yes
```

Assuming our previous agents, "foo" and "bar", are still running, and used dynamic ports 49152 and 49153, the chances are that "brains" will automatically be assigned port 49154, as shown above and in *Fig 4.10*. Now let's post an event to ourselves (brains):

```
| ?- agent_post( brains, [], calculate(22/7) ).          <enter>
yes
```

As expected, the command completes immediately, with no output. A fraction of a second later, the first of two piece of output appears:

```
| ?-
Brains is calculating 22 / 7
```

This has been displayed by the "calculate(Maths)" event, before it actually performs any arithmetic and then in turn posts the "result(Maths=Result)" event. The latter causes more output to appears a moment later still, as shown below and in *Fig 4.11*:

```
| ?-
Brains says:
        The result of 22 / 7 is 3.14285714285714
| ?-
```

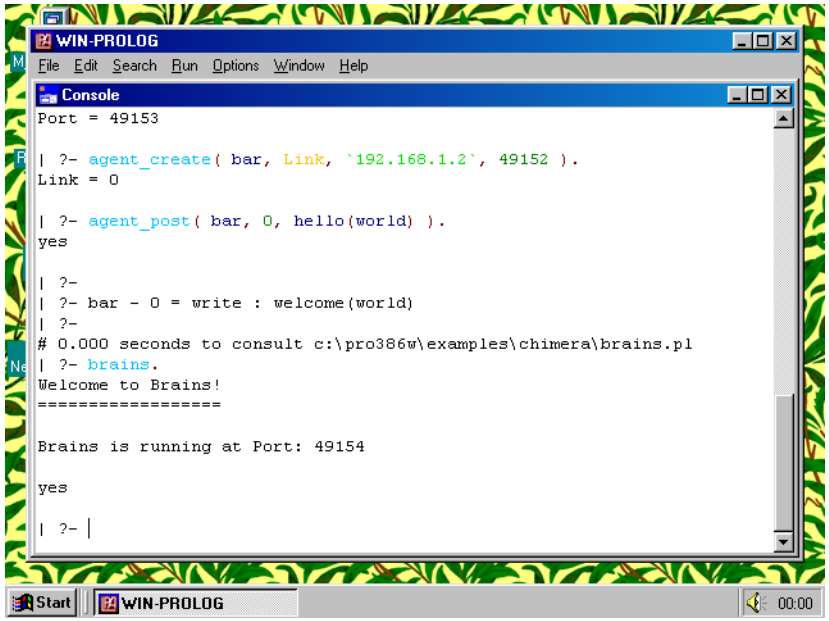


Fig 4.10 - Welcome to Brains!

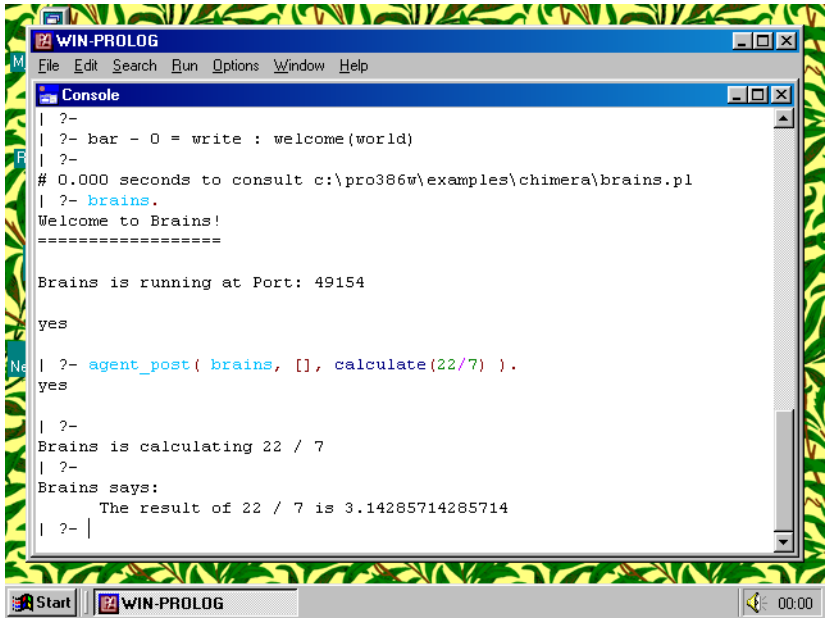


Fig 4.11 - Calculated output from Brains

Let's perform two more experiments with "brains". Firstly, we'll connect our simple agent, "bar", using our IP address (see above) and the port number reported by brains/0 above:

```
| ?- agent_create( bar, Link, `192.168.1.2`, 49154 ).     <enter>
Link = 1
```

This time, the returned link number is "1" (one): agent "bar" now has two links, with "0" (zero) being to agent "foo", and "1" (one) to agent "brains"; now type:

```
| ?- agent_post( bar, 1, calculate(22/7) ).               <enter>
yes
```

This posts a message to link "1" (one) of agent "bar", which should be connected to "brains". Sure enough, a moment later, the "calculate(22/7)" event is echoed by thinker/3:

```
| ?-
Brains is calculating 22 / 7
```

After outputting the above message, thinker/3 calls *is/2* to execute the given arithmetic expression, before posting the "result(Maths=Result)" event. This time, the event is posted back to "bar", rather than "brains" itself, and bar_handler/3 responds by displaying the result, as shown below and in *Fig 4.12*:

```
| ?- bar - 1 = result : (22 / 7 = 3.14285714285714)
| ?-
```

Distributed Brains

Now we've had a chance to look at an agent, "brains", which handles certain events as queries, and posts events containing their results; we've even connected to "brains" from a really simple agent, "bar", whose event handler simply outputs text to the console window. What we haven't done yet is show such connections working across multiple instances of *WIN-PROLOG*, or even across a network. Let's do that now: quit from *WIN-PROLOG* by typing the command:

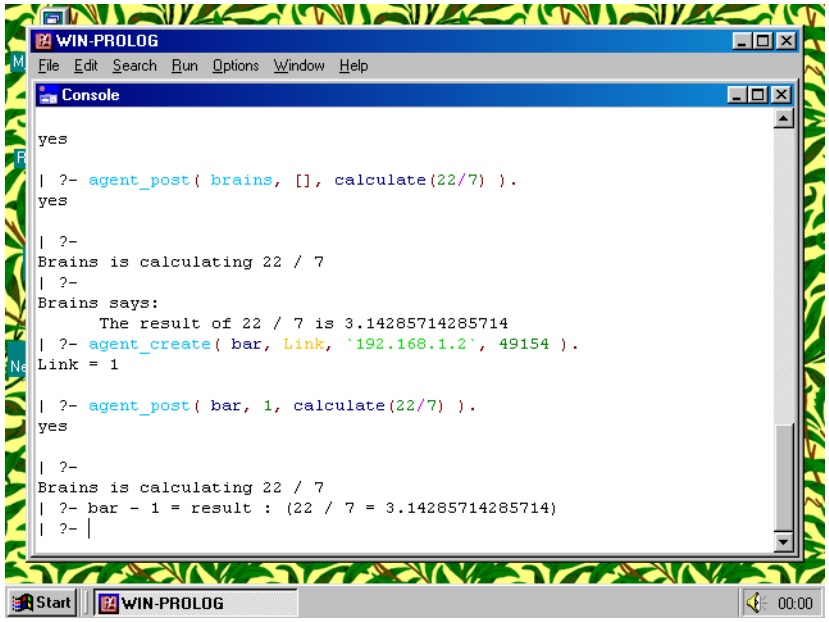


Fig 4.12 - Output from "bar" after communication with

```
| ?- halt.                                                    <enter>
```

Next, start up two copies of ***WIN-PROLOG***, above and below, as we did in *Chapter 3* and as shown in *Fig 4.13*; in each one, load a copy of *BRAINS.PL* and compile it, before returning to the respective console windows. In the upper copy of ***WIN-PROLOG***, enter the following command:

```
| ?- brains.                                                  <enter>
Welcome to Brains!
==================

Brains is running at Port: 49152

yes
```

Make a note of the port number that is reported (here, 49152), and then switch to the lower copy of ***WIN-PROLOG***, entering the same command there:

```
| ?- brains.                                                  <enter>
Welcome to Brains!
==================

Brains is running at Port: 49153

yes
```

Your second copy of "brains" will be allocated a different port number, probably 49153, as shown above and in *Fig 4.14*. Now, knowing your machine's IP address (see Chapter 3 for information about *IPCONFIG* or *WINIPCFG*) or its domain name, and using the port number reported by the upper copy of "brains", enter the following command into the lower copy of ***WIN-PROLOG***:

```
| ?- agent_create( brains, Link, `192.168.1.2`, 49152 ). <enter>
Link = 0
```

You will now have created a link, numbered "0" (zero), from the lower "brains" to the upper one: test this, by typing the command into the lower copy of ***WIN-PROLOG***:

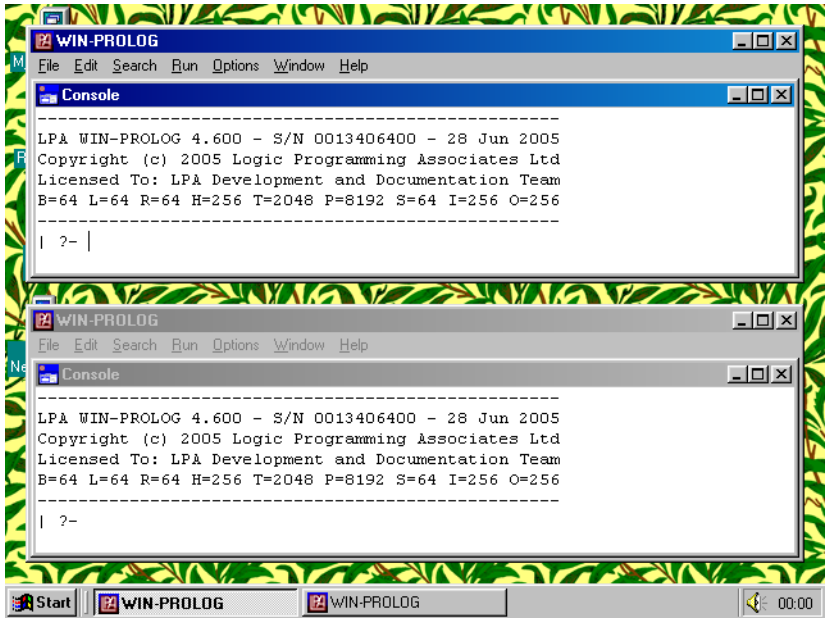


Fig 4.13 - Starting up two copies of WIN-PROLOG

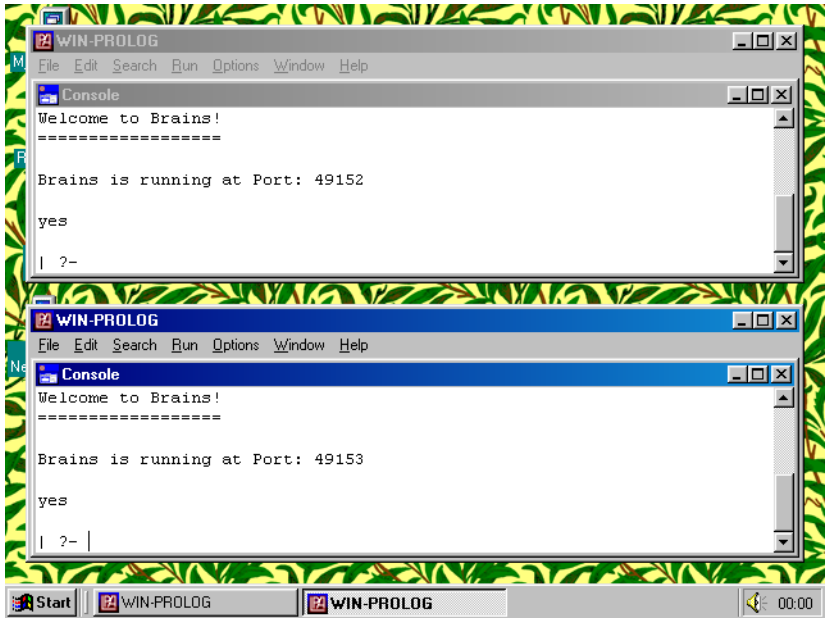


Fig 4.14 - Running two copies of Brains

```
| ?- agent_post( brains, 0, english([hello,sir]) ).      <enter>
yes
```

This posts an "english(English)" event to whichever agent is connected to link "0" (zero) of the lower copy of agent "brains", which of course is the upper copy of "brains". You will see the following output appear in the upper copy of ***WIN-PROLOG***:

```
| ?-
Brains is translating the English, [hello,sir], into French
```

which indicates that this (upper) agent has received the event, "english(English)", and is trying to translate it. Once successful, this agent posts a "translation(Englis h,French)" event back to the originator of the original message, and this results in output in the console window of the lower copy of ***WIN-PROLOG***, as shown below and in *Fig 4.15*:

```
| ?-
Brains says:
     The English phrase: [hello,sir]
     is like the French: [bonjour,monsieur]
```

The link between the two agents is a two-way channel: to show this, go to the upper copy of ***WIN-PROLOG*** and type:

```
| ?- agent_post( brains, 0, english([i,hate,cabbage]) ). <enter>
yes
```

This time, the lower copy of ***WIN-PROLOG*** will display the following:

```
| ?-
Brains is translating the English, [i,hate,cabbage], into
French
```

following which the upper copy of ***WIN-PROLOG*** will then display the result, as also shown below and in *Fig 4.16*:

```
| ?-
```

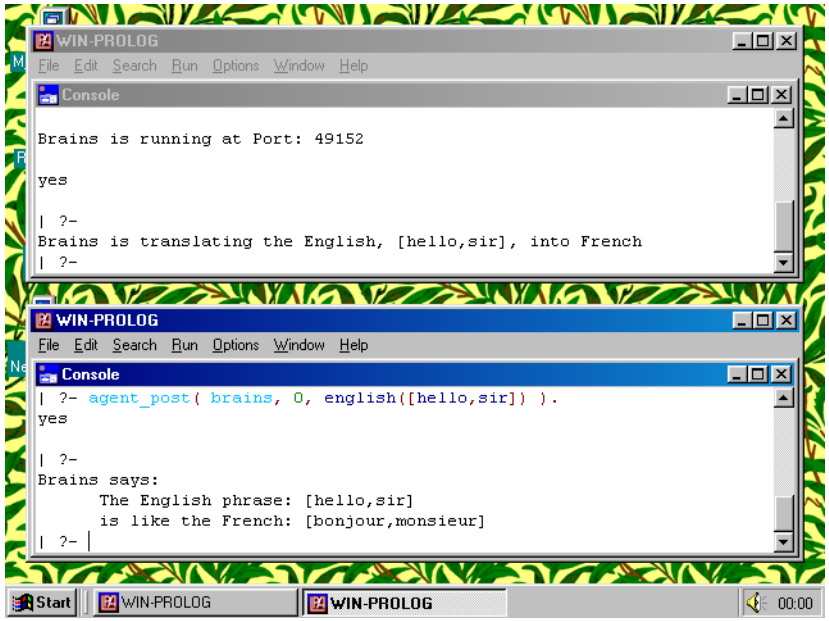


Fig 4.15 - Distributed execution of a Brains query

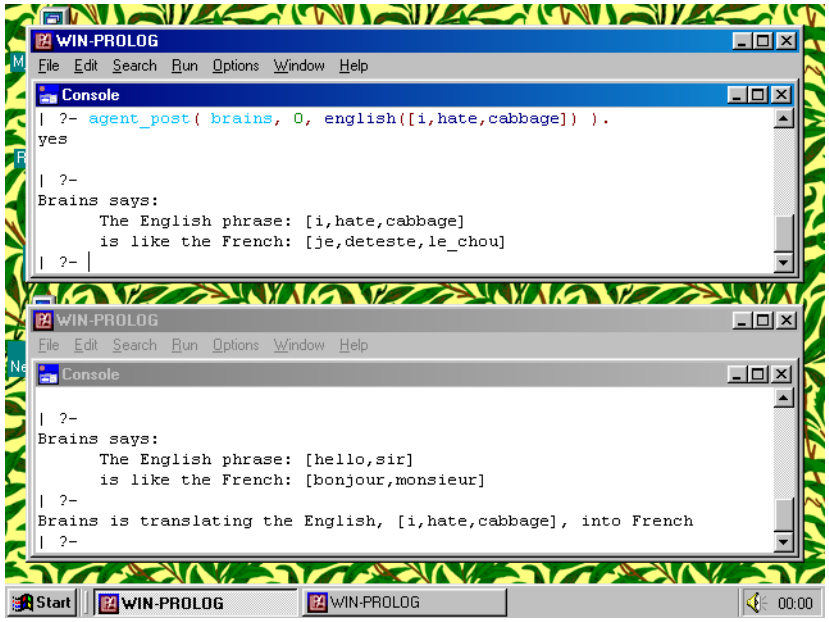


Fig 4.16 - Communication working in both directions

```
Brains says:
     The English phrase: [i,hate,cabbage]
     is like the French: [je,deteste,le_chou]
```

Success, Failure and Errors

When we were experimenting with our original handler/3 predicate in *Chapter 3*, a good many *System Events* were being echoed to the respective console windows, alongside our simple user events like "hello(world)". This is because this event handler matched all events, both those generated by the system, and those by the user. You might well be wondering what has happened to these events in our examples here in *Chapter 4*.

One of the features of Chimera event handler model is that it is entirely up to you just how many events you do - or don't - handle. In other words, when writing an event handler, you simply add the cases you want to process, and ignore the rest. If you don't have a case for a particular event, that event is simply discarded. Likewise, it doesn't actually matter whether your event handler succeeds or fails for any particular event.

For each event that is dispatched, Chimera makes one attempt to call your event handler: whether your handler succeeds or fails doesn't matter, any more than it matters whether or not you have cases for a particular event. The only case which will cause interest is an event handler that generates a runtime error: this is not "caught" (unless you choose to wrap your code fragments inside *catch/2*), and will be reported just like any other runtime error.

You also do not need to worry about leaving choice points: once your handler has returned (success or failure), it will not be called again under any circumstances. This is why the translate/2 predicate in our "brains" example could be written without cut (*!/0*).

Tracing Events

Because of the way event handlers are called by Chimera, it is extremely easy to "trace" event traffic. All that is need is a first clause in the event handler, effectively containing the code of handler/3, but finishing with a call to *fail/0*. Add the following

to the copy of "brains" in the upper copy of ***WIN-PROLOG***, immediately below the clause for brains/0, as shown in *Fig 4.17*:

```
% tracer case

thinker( Name, Link, Event ) :-
   nl,
   write( name = Name ),
   nl,
   write( link = Link ),
   nl,
   write( event = Event ),
   nl,
   fail.
```

Once again, compile this code and return to the console window, and then from the lower copy of ***WIN-PROLOG***, type the following command:

```
| ?- agent_post( brains, 0, french([bonjour,madame]) ).  <enter>
yes
```

Over in the upper copy of ***WIN-PROLOG***, our new first clause to thinker/3 will cause our modified "brains" to output a trace of all events dispatched, including system and user events, before failing, allowing the original handler cases to work as before:

```
| ?-
name = brains
link = 0
event = (read,192.168.1.2,1027,french([bonjour,madame]))
| ?-
name = brains
link = 0
event = french([bonjour,madame])

Brains is translating the French, [bonjour,madame], into Eng-
lish
| ?-
```

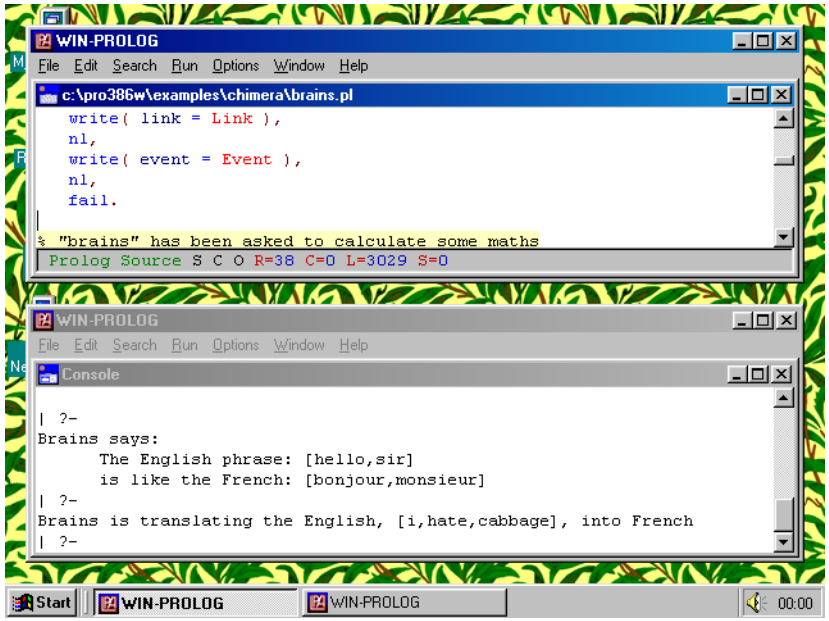


Fig 4.17 - Adding a tracer clause to the event handler

```
name = brains
link = 0
event = (write,192.168.1.2,1027,translation([hello,madam],[bonj
our,madame]))
```

When all is done, the lower copy of ***WIN-PROLOG*** will show the following result, also shown in *Fig 4.18*:

```
| ?-
Brains says:
     The English phrase: [hello,madam]
     is like the French: [bonjour,madame]
```

The CHITRACE Utility

The technique we have just used, namely adding a clause to the top of an event handler, which simply performs debugging output before failing back into the remaining clauses, is used in a more sophisticated form by a utility program, *CHITRACE.PL*, which you will find in the "*EXAMPLES\CHIMERA*" directory.

This file contains one main predicate, *trace_agent/3*, which you can simply call from the first clause of your own event handler. For each agent being traced, *trace_agent/3* creates a separate MDI window, and displays a pretty-printed version of every event, complete with a date/time stamp, to help debug agent applications.

As a final touch, *trace_agent/3* is designed to fail on completion, so that it automatically returns control to the remainder of your handler. This utility is described in further detail in *Chapter 7*.

Closing Down

Finally, let's close down the lower "brains", using *agent_close/1*, which closes the agent itself as well as all its links; type the following command into the lower copy of ***WIN-PROLOG***:

```
| ?- agent_close( brains ).                                    <enter>
yes
```

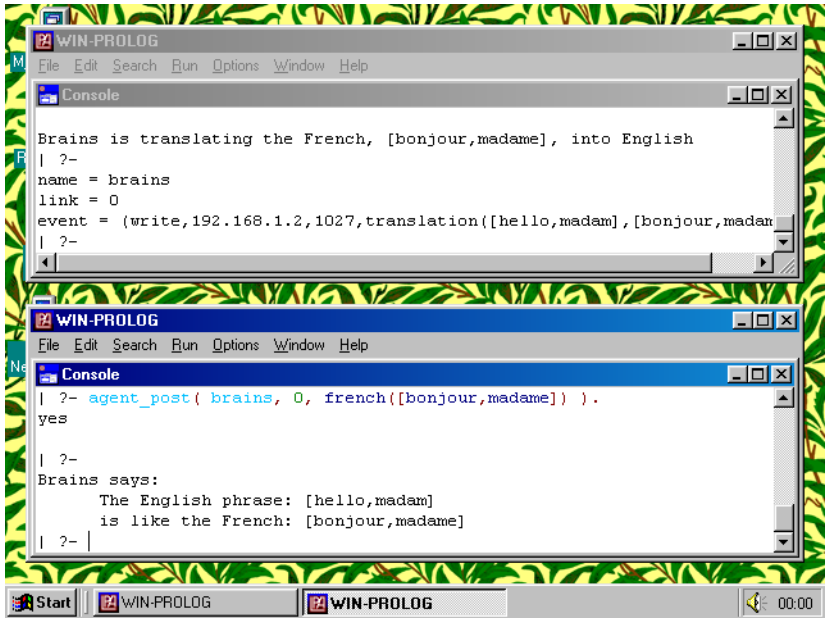


Fig 4.18 - Brains displaying events in the console window

This command will cause more output in the following command in the upper copy of ***WIN-PROLOG***, as shown below and in *Fig 4.19*:

```
| ?-
name = brains
link = 0
event = (close,192.168.1.2,1027)
```

As usual, there is no need to process this system event: Chimera performs all necessary link closing and tidying up on your behalf. Finally, let's release remaining resources by closing the modified upper copy of "brains"; type the following command into the upper copy of ***WIN-PROLOG***:

```
| ?- agent_close( brains ).                                    <enter>
yes
```

Our modified version of thinker/2 outputs one last system event, as shown below and in Fig *4.20*:

```
| ?-
name = brains
link = []
event = (close,49152)
```

Rounded Up

This concludes our investigations of Chimera Event Handlers; in Chapter 5, we will look at a more interesting, extended example application, in which multiple agent "gamblers" bet at a "roulette" table, and try to beat the house.

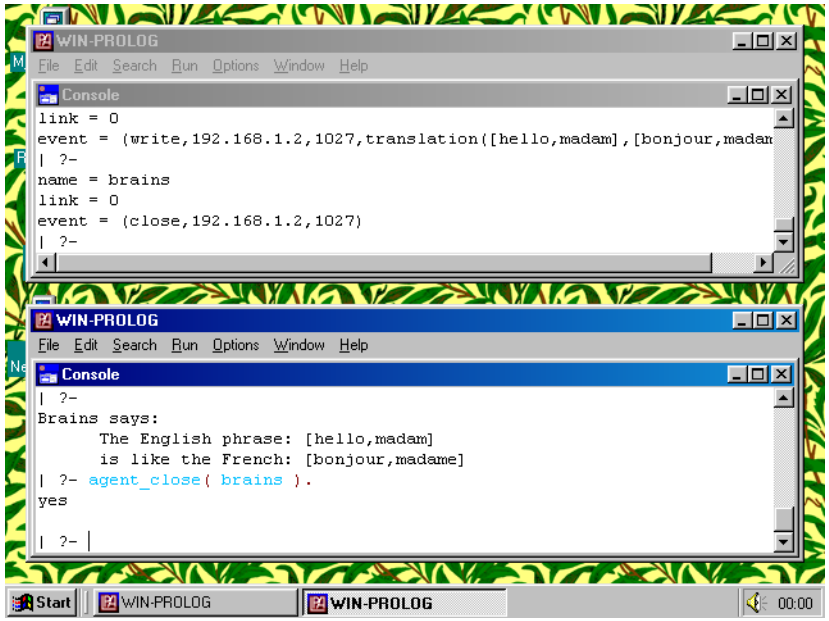


Fig 4.19 - Events after closing the right copy of Brains

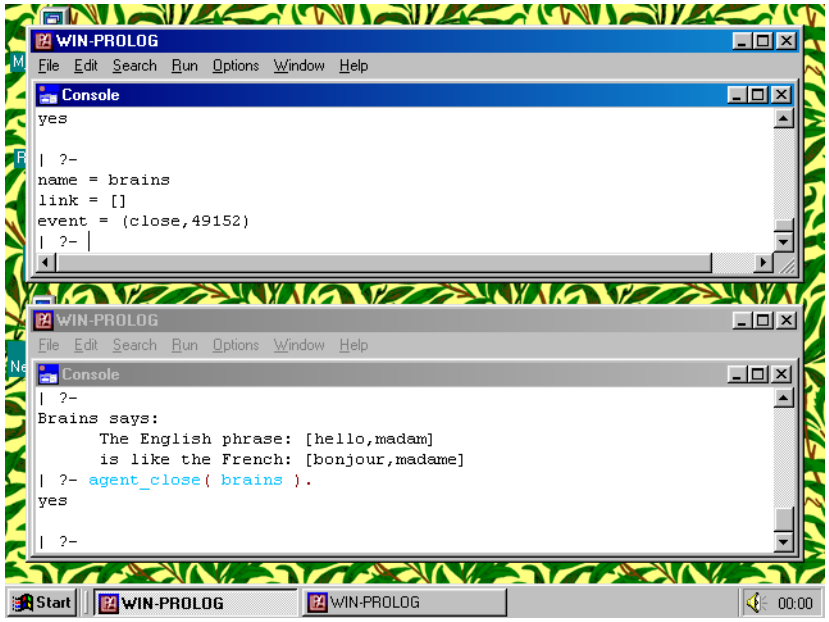


Fig 4.20 - Events after closing the left copy of Brains

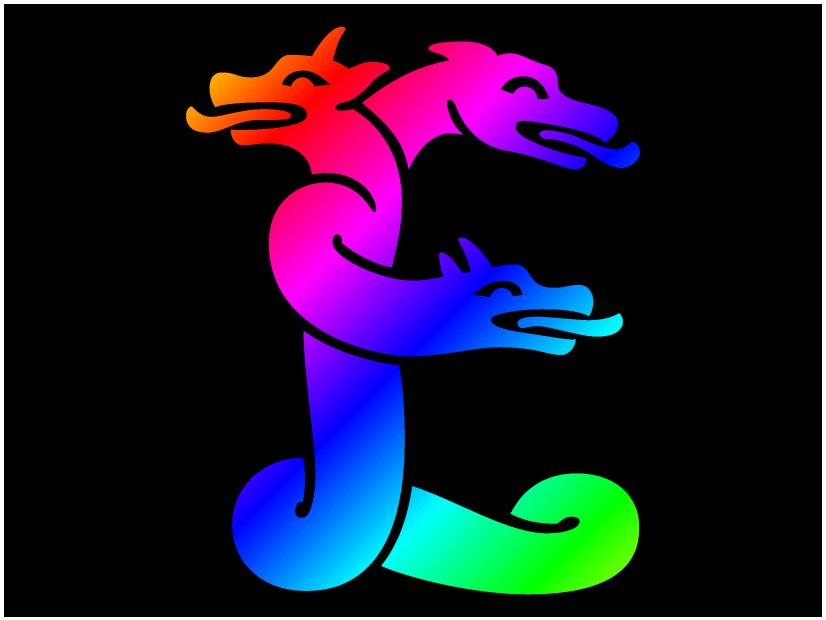
Chapter 5 - Multiple Agents: Gambler and Roulette

In this chapter, we will take a look at the world of multiple agents, using an interesting pair of example programs, "Gambler" and "Roulette", building on the understanding and knowledge we have attained in previous chapters.

The Pretext

While trying to think up an interesting, yet tractable example of a multi-agent system, which could serve to illustrate important techniques, and at the same time be attractive and amusing to run, an old paper-based "system" for winning at the casino game, *Roulette*, came to mind. Using this system, and betting only on the 1:1 bets of a roulette table, it is usually possible, with only a modest "bank", to win a predetermined amount of money over a relatively short number of consecutive bets. The system worked as follows:

```
1) Decide how much you want to win - say 10 euros

2) Write down a list of numbers which adds up to this amount;
   say: 1, 2, 3, 4

3) Add together the first and last numbers on the list, and
   stake this amount - in this case, initially 5 euros - on any
   one of the 1:1 bets: Red, Black, Low, High, Even or Odd

4) If you win, cross out the two numbers you added together,
   and return to step 3 with your shorter list

5) If you lose, append the amount bet to the list, and return
   to step 3 with your longer list

6) If at step 3, there is only one number, bet that amount

7) If at step 3, there no numbers, you've won your 10 euros!
   Stick the money somewhere safe, and return to step 1 ...
```



C H I M Enhanced R A

This system works surprisingly well: in fact, you can run two lists, side by side, betting on opposites (if list 1 bets "black", then list 2 bets "red", and so on), and both will have a tendency to win over any given sequence of spins of the roulette wheel. Furthermore, unlike the financially suicidal "doubling" technique, your exposure to massive bets and astronomical losses is relatively small.

How does it work? Well, put simply, imagine an alternating sequence of wins and losses: every time you win (50% of the time), you remove two numbers from your list; every time you lose (the other 50%), you add just one. Over time the list will shrink, until it finally vanishes.

Let's work an example, assuming alternate Red/Black results, with two gamblers, called "Red" and "Black" because they always bet on Red and Black respectively. Let's also assume that both gamblers want to win 10 euros, and have broken this amount down into the list, "1, 2, 3, 4"; here's how they both fare:

Gambler	List	Bet	Spin	Result	Profit
Red	1,2,3,4	1+4 = 5	1:RED	WIN	5
Black	1,2,3,4	1+4 = 5		LOSE	-5
Red	2,3	2+3 = 5	2:BLACK	LOSE	0
Black	1,2,3,4,5	1+5 = 6		WIN	1
Red	2,3,5	2+5 = 7	3:RED	WIN	7
Black	2,3,4	2+4 = 6		LOSE	-5
Red	3	3 = 3	4:BLACK	LOSE	4
Black	2,3,4,6	2+6 = 8		WIN	3
Red	3,3	3+3 = 6	5:RED	WIN	10
Black	3,4	3+4 = 7		LOSE	-4
Red	FINISHED		6:BLACK		10
Black	3,4,7	3+7 = 10		WIN	6
Red	FINISHED		7:RED		10

```
Black     4             4    = 4            LOSE            2

Red       FINISHED                 8:BLACK                  10
Black     4,4           4+4 = 8            WIN             10

Red       FINISHED                                         10
Black     FINISHED                                         10
```

As we can see, over a series of just 8 spins of the wheel, even though they were making simultaneous opposite bets, both "Red" and "Black" have managed to achieve their desired win of 10 euros. Neither was ever more than 5 euros out of pocket, and neither had to make an individual bet of greater than 10 euros.

Of course, real random sequences are seldom so kind: long runs of losses can occur, and even with this system, bet sizes can rise rapidly and bank reserves diminish dramatically, in a vain attempt to win a relatively modest amount. Nonetheless, it remains the case that, provided you don't lose twice as often as you win, and have sufficient credit to cover a bad run, you will always win eventually. And with a margin as wide as this, even the double-zero of American Roulette tables makes little difference: you are still winning, on average, 18 times out of 38, well within the 1:2 win:lose ratio beyond which this system would break..

Furthermore, it is possible to tailor this system to your level of nerves by choosing lists of different lengths before starting: longer lists of smaller numbers (such as a list of ten lots of "1" (one)) take more time to work through, but give you smaller individual bets and less exposure to the effects of losing streaks; shorter lists (such as, "5,5") let you win - or go bust - more quickly.

Adapting the System to Prolog

The old paper system uses "add first to last" for computing a bet because it's easy to remember and strike off, and "append to list" for losses because that's where you've got some nice clean paper to write down your next number. Mathematically, it makes no odds (oops!) from where in the list you take your two numbers, or where you insert your lost bets, so in Prolog, we might as well use pattern matching at the head of a list. Given a list, say:

```
X = [1,2,3,4]
```

we can compute our bet, "B" using an expression such as:

```
X = [H1,H2|T], B is H1 + H2
```

If we win, our new list is simply "T"; if we lose, it becomes:

```
[B|X]
```

When the list is empty ("[]"), we've won our intended amount, and the only other case to handle is when the list contains a single entry:

```
X = [H], B is H
```

Of course, we could have used *append/3* to stick our losses on the far end of the list, or even to pick up the last element to add to the first, but there is no point: all that matters is that we take two numbers (if available) to make each bet, removing them if we win, and only putting back one if we lose.

A Timely Warning

Just before we get into the details, perhaps it would be prudent to read and understand the following disclaimer:

```
No "system" for winning at a game of pure chance, includ-
ing Roulette, can ever be 100% reliable. The examples pre-
sented here are intended solely to illustrate the programming
of agents within Chimera and explicitly not for the purpose
of betting with real money at an Internet gambling site or any
other form of casino.

Neither Logic Programming Associates nor the author of these
programs accept any liability for losses incurred by attempting
to use the systems and algorithms described in the present
chapter and embodied within the "Gambler" and "Roulette" ex-
ample programs in a real gambling environment.
```

Please enjoy this program in the spirit in which it was written: as an interesting demonstration of multiple agents using an intelligent scheme to try to beat the odds of Roulette.

The Gambler/Roulette Architecture

Now that we've explored and understood how to implement an old, paper-based system for winning at roulette, it's time to see how to utilise this within a multi-agent environment. The idea was to have two programs, or agents, one emulating a roulette table and the other a gambler.

The gambler would arrive at the able, and buy some chips: the number of chips bought would reflect size of the gambler's "bank". A virtual croupier would announce, "place your bets", at which point the gambler would decide how much to stake, and on what 1:1 bet. After a few moments, the croupier would proclaim, "no more bets", and spin the roulette wheel, before declaring the result of the spin, and tell the gambler whether he had won or lost. After updating the accounts, the croupier would start over, with another "place your bets".

Using a Prolog-friendly form of the old paper system, it seemed plausible that a gambler agent could be written which would automatically win, at the expense of the roulette agent, over a period of time. And if one gambler could play at the roulette table, why not several? Because of the flexibility of Chimera's design, it would be relatively easy to write a system where multiple plays, making random or fixed bets could play simultaneously, both locally on any one computer and/or across a network, all winning over time, and hopefully taking the "house to the cleaners".

Communications Protocols

Any multi-agent system relies on a *Communications Protocol* to synchronise and react to events. All that such a protocol comprises, so far as we are concerned, is a set of events that we've designed, together with the responses we expect to get. In *Chapter 4*, we looked at a simple agent, "Brains", to which we could post an event such as "calculate(22/7)', whereupon the agent would respond by posting back the event, "result(22/7=3.14...)": this pair of events, and they way they were interrelated, was the first element of a rudimentary protocol.

In the Gambler/Roulette example, we are going to develop the protocol concept much further, such that a sustained *Conversation* develops between multiple agents. Let's look at an example of what we mean; consider this model, as if it were a script for a play:

Gambler:	Hello, I'd like to play, and I have 50 euros
Roulette:	[Checks that Gambler is not already playing] Welcome, you are gambler number 1! [Pauses] Place Your Bets!
Gambler:	[Calculates an amount to bet] Bet 5 on Red
Roulette:	[Checks that bet is valid] Bet Accepted

and so on. Each "actor" in the above script waits from some cue, and reacts appropriately. This is exactly how the Gambler/Roulette example works, with the added bonus that any number of gamblers can connect simultaneously: each gambler says "Hello", and is welcomed to the table; each can react to "Place Your Bets" by computing the next wager, and informing the roulette table, and so forth. Very little additional programming or complexity is required to handle three, five, or fifty simultaneous gamblers: the whole combined application is simply driven by events and responses - a communications protocol.

Broadcast Events and Registration

One area which becomes necessary with multiple agent systems is the ability to *Broadcast*, rather than simply to post, an event. Posting implies sending an event to a single, known agent, which might connected at the other end of a link, or might simply be oneself; broadcasting involves posting a message to multiple recipient agents.

It turns out that there is no need for explicit broadcasting predicates within Chimera: all the elements required to perform this function already exist. For any given agent, you can broadcast an event simply by posting it to each of that agent's links in turn, for example:

```
broadcast( Name, Event ) :-
   agent_data( Name, _, _, Links ),
   forall( member( Link, Links ),
           agent_post( Name, Link, Event )
         ).
```

While the above technique works perfectly well, it doesn't tell the entire story: in many applications, it is desirable to broadcast certain events only to selected connections, and this is where *Registration* comes in. By requiring agents to "register" with an application after establishing their incoming connections, is becomes possible to limit broadcasts to those agents wishing to receive them.

Timer Events

Any system comprising one or more agents needs some kind of stimulus to create an initial event; the response to this event may generate further events, each of which might post even more, creating a chain reaction that will end up with one of two results. Hopefully, the chain of events will eventually play itself out, at which point we can consider the handling of the original event complete. However, there is also a risk of the chain reaction running away with itself, with events being posted more quickly than can be processed.

All this leaves us with a conundrum: we want the Gambler/Roulette system to run indefinitely, and yet we don't want it to run away with itself. The solution to this is to ensure that the sequence of events following the initial event plays itself out, but then to reissue the initial event repeatedly in a controlled, periodic fashion.

The best way to achieve this kind of periodic behaviour is to generate *Timer Events*; fortunately, ***WIN-PROLOG*** already has a powerful system of *Programmable Timers*, driven by predicates such as *timer_create/2*, *timer_get/2* and *timer_set/2* (see the ***WIN-PROLOG*** Technical Reference), so there is no need for Chimera to duplicate these functions.

Using timers to fire periodic events also enables programs to "idle" productively, either allowing user input and other functions to continue unhindered, or to perform useful background operations such as the maintenance of databases.

Timer events are handled in *real time* by a special *Timer Handler*, very much in the same way as an agent's Event Handler handles system and user events. However, the "real time" aspect of timer handlers gives them absolute and immediate priority over everything else: indeed, one timer event will happily interrupt another midway through, whereas the system and user events in Chimera work through a polite queuing system, allowing events to be processed in strict rotation.

Timer events are really intended for firing short, monolithic code fragments for such purposes as updating a clock display, or profiling a program's execution: because they work in real time, and can interrupt anything, including other timer events, they are not ideal for invoking large chunks of an application. For this reason, the Roulette agent uses timer events for one purpose only: to post a user event to itself! This ensures that everything gets executed in the desired sequence.

Combining Dialogs and Storage

An important component of the Gambler/Roulette example was always going to be an interesting, animated user interface, and this meant designing large dialogs with lots of fields that could be set up by the user prior to or during execution, as well as for output of intermediate and final results. One problem always associated with programs that try to mirror activities in dialogs is there sheer amount of housekeeping required to reflect every database update to a control window, and vice versa. For example, code such as this is common in ***WIN-PROLOG*** applications:

```
increment_value( Value ) :-
   retract( my_value(Sofar) ),
   Extra is Sofar + Value,
   assert( my_value(Extra) ),
   write( Extra ) ~> String,
   wtext( (my_dialog,999), String ).
```

In this example, a lemma, my_value/1, is being replaced with one containing a newly computed value, after which the latter is written to a string and displayed in a dialog control window. Effectively, two copies of the value are being kept and maintained, one numeric (in the lemma) and one textual (in the dialog control). Each and every time the my_value/1 lemma is updated, code to update the dialog must also be called.

Suppose we did away with the lemma altogether, and instead used the dialog and its controls directly for storage? Rather than retracting the clause for a lemma, we could read the current value directly from a dialog control, perform our maths, and then rather than asserting the result *and* updating the dialog, just update the dialog. This would eliminate the classic double update problems associated with maintaining parallel copies of data, and could be abstracted into a simple property setting/getting predicate.

This is exactly the approach that was taken with Gambler/Roulette: each has a large dialog containing all relevant data, which is retrieved, updated and set directly, without an accompanying Prolog database of lemmas.

Gambler and Roulette: The Events

We've already discussed the concept of communications protocols, and looked at a hypothetical example, where a gambler arrives at a table and says, "Hello…", and is greeted with "Welcome…", and so on. In fact, this is pretty much exactly how the real Gambler/Roulette programs communicate. The periodic sequence in Roulette is initiated by a timer event, which results in a chain of events:

Event	Action
<timer event>	post *nomore(Time)* to self
nomore(Time)	broadcast *no_more_bets(Count)* to all registered players; post *spin(Time,Count)* to self
spin(Time,Count)	broadcast *the_result_was(Count,Spin)* to all; post *place(Time,Spin)* to self
place(Time,Spin)	post *you_have_won(Count,Value,Win,Spin)* to winners, *you_have_lost(Count,Value,Wager,Spin)* to losers, and *goodbye_player(Bets,Bank)* to all that have resigned; broadcast *place_your_bets(Count)* to all, and set a *<timer event>* to "Time"

The above sequence is perpetual and self-running: provided the events which are

posted can be processed within the initially specified time slice, Roulette will run in an orderly way.

The above events only represent part of Roulette's capabilities: it needs to be able to react to events posted by one or more instances of Gambler. Among these events, with their arguments abbreviated to ellipses, are:

Event	Action
hello(...)	check whether gambler is already playing, and if so post *already_playing(...)*; check proposed credit, and if no good, post *stake_too_small(...)*; otherwise, if everything appears to be alright, post *welcome_player(...)*
goodbye(...)	record that this gambler wants to stop: this will be acknowledged at the end of the current round during the *place(...)* event
bet(...)	gambler has made a bet: check credit, and if no good, post *not_enough_credit(...)*; check whether this round is still accepting bets, and if not, post *cannot_place_bet(...)*; check whether the house can afford to lose the bet, and if not, post *cannot_risk_bet(...)*; otherwise, if everything appears to be alright, post *bet_accepted(...)*

Other events have been included, such as "*bank(_)*", which posts an event containing a gambler's current bank account; "*bets(_)*", which posts an event enumerating all the valid bets, and "*odds(Bet)*", which posts back the odds paid on any given event: none of these events are used by the current implementation of Gambler.

The Roulette agent, with its timer, is the driving force behind the overall application: consequently, Gambler needs neither an "initial event" nor a timer to keep going. Instead, once registered, it simply reacts to broadcast and individually posted events from Roulette:

Event	Action
welcome_player(...)	initialise the dialog and start playing
goodbye_player(...)	complete tidy up the dialog and stop playing
place_your_bets(...)	compute bet, and post *bet(Count,Value,Wager)*
bet_accepted(...	increment bet count
not_enough_credit(...)	we've gone bust; post *stop(...)* to ourself
cannot_risk_bet(...)	the casino has gone bust; post *stop(...)* to ourself
cannot_place_bet(...)	we've missed this round: get ready to play the next
you_have_won(...)	adjust bank and bet list, and if done, post *stop(...)*
you_have_lost(...)	adjust bank and bet list

For the full details of how these events interact with each other, have a look at the source code of these two programs, *GAMBLER.PL* and *ROULETTE.PL*, which can be found in ***WIN-PROLOG***'s "*EXAMPLES\CHIMERA*" directory.

Running Roulette

Because agents in Chimera are individually named, we can run the Gambler and Roulette examples from within a single instance of ***WIN-PROLOG*** just as easily as we can amongst multiple instances across a network. Start a copy of ***WIN-PROLOG***, and then use "File/Open" to locate the "*EXAMPLES\CHIMERA*" directory, and then load the two programs, *GAMBLER.PL* and *ROULETTE.PL*. Once each has been compiled, using "Run/Compile", go to the console window and type the command:

```
| ?- roulette.                                                    <enter>
yes
```

This will display Roulette's control dialog, as shown in *Fig 5.1*. At this point, the

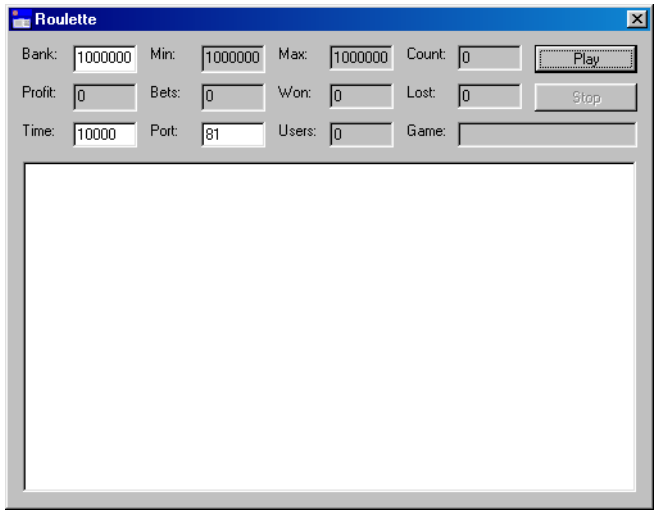


Fig 5.1 - The Roulette control dialog

Roulette agent is not running: instead, you have the opportunity to modify three settings, the "Bank", which states how much money the "house" has to give out in winnings; "Time", which is specified in milliseconds and is used to set the programmable timer that drives the application; finally, "Port" specifies the TCP/IP port at which Roulette will run.

The three editable fields are initially set to 1000000, 10000 and 81 respectively, meaning that the house has one million currency units to give out as winnings, that each round will last ten seconds, and that port 81 will be used to listen for incoming connection requests.

To start with, you should try leaving these settings unchanged. Click the "Play" button, and Roulette will start agent Roulette running. The three adjustable fields will be locked, and a trace of each main game event will start to appear, as shown in *Fig 5.2*.

Running Gambler

Now that we have set Roulette running, it's time to run Gambler; type the following command:

```
| ?- gambler( foo ).                                    <enter>
yes
```

The Gambler control dialog will appear, as shown in *Fig 5.3*, but again, the agent itself will not yet have started. Because we want to be able to run more than one instance of Gambler at a time, this example is run with an argument which names the particular instance of Gambler, in this case "foo".

This time, four editable fields are presented: "Win" allows you to specify your target win, "Bank" contains the amount you are prepared to lose before giving up, "Host" is the name or IP address of the computer that's running Roulette, and Port is the port at which it is running. The default values of 100, 10000, 192.168.1.2 and 81 state, respectively, that you want to win one hundred currency units, and have ten thousand units in your bank, and finally that Roulette is running on the computer at address 192.168.1.2 at port 81.

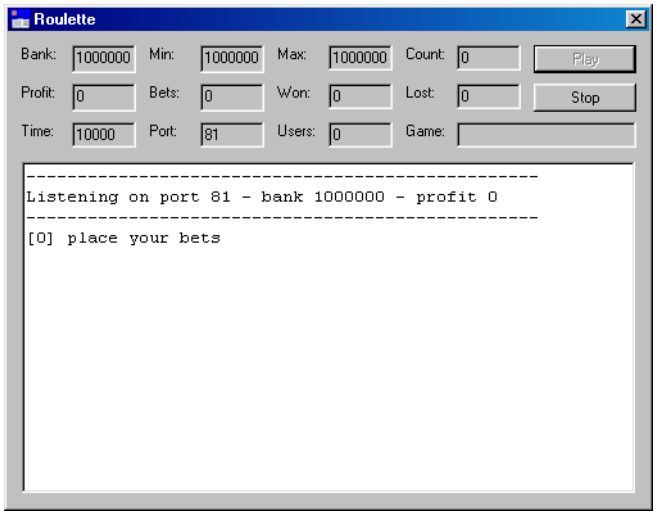


Fig 5.2 - The Roulette agent running

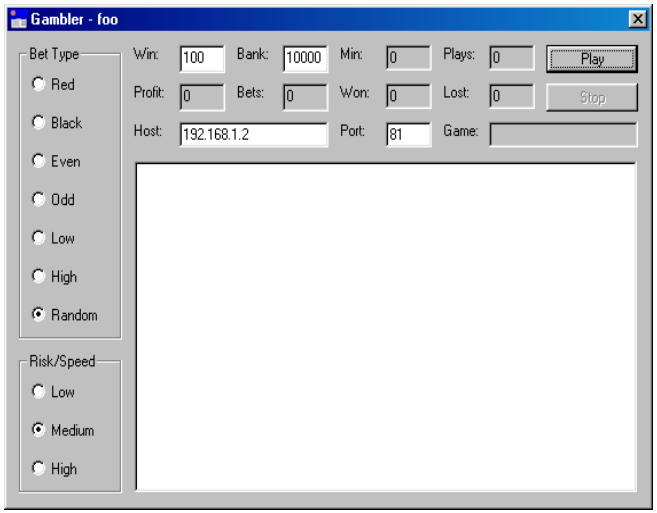


Fig 5.3 - The Gambler control dialog

The port number (81) needs to be the same as that specified when you ran Roulette; the host (192.168.1.2) should be changed to the name or address of the computer on which you're running Roulette. If this is simply the local machine, as will be the case if you've followed the instructions presented here, then you might need to change this value: see *Chapter 3* about how to use the *IPCONFIG* and *WINIPCFG* commands, which enable you to determine this value.

Two more groups of settings are present in the Gambler control dialog: "Bet Type" lets you specify whether the given instance of Gambler will make random bets, or one of six fixed bets; "Risk/Speed" lets you select the length of list used to compute bets: the lower the setting, the longer the list: remember that a longer list means the game will take longer to play out, but that you will experience less risk of going "bust" during a prolonged losing sequence.

Let's start the agent; click on Play: the adjustable fields are locked, and a trace of events begins to appear, as shown in *Fig 5.4*. Notice, however, that one group of settings, the "Bet Type", is left enabled: you can change the bet that you want Gambler to use as often as you like, even when it is running.

More Gamblers

As mentioned above, part of the design goal of the Gambler/Roulette example was to allow multiple gamblers to play simultaneously, allowing us to experiment with opposite bets or different risk strategies, to prove how well (or otherwise) our "system" manages to win. Set up two more Gambler instances now, but typing the following commands:

```
| ?- gambler( bar ).                                    <enter>
yes

| ?- gambler( sux ).                                    <enter>
yes
```

Two more Gambler control dialogs will appear. Minimize the ***WIN-PROLOG*** main window to reduce screen clutter, and move the three Gambler and one Roulette dialog about the screen so that all four are visible, as shown in *Fig 5.5*. Adjust the "Host" and "Port" values in Gamblers "bar" and "sux" to the same setting you used in

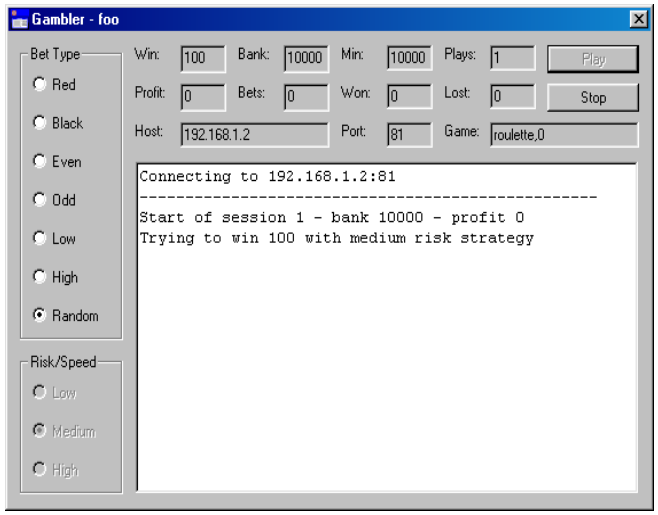


Fig 5.4 - The Gambler agent running

Fig 5.5 - Three Gambler agents and one Roulette agent

"foo", and then choose bet types of "Red" and "Black" in "bar" and "sux" respectively, forcing them to make fixed, opposite bets. Start both of these instances of Gambler by pressing their respective play buttons, and then sit back and watch while they both set about trying to winning. Even though they are making opposite bets on a single sequence of Roulette spins, there is a good chance they will both succeed, as shown in *Fig 5.6*.

Multiple Gamblers Across a Network

To get a real feel for the potential of multi-agent systems implemented with Chimera, it's worth trying to run Roulette and multiple instances of Gambler across a *Local Area Network* (*LAN*). Simply run Roulette on one machine, and as many copies of Gambler as you want on other machines on the same network. Make sure the "Host" and "Port" fields of each of the Gamblers match those of the machine and port where Roulette is running, and you're ready to go.

To make everything even more dynamic, reduce the "Time" field of Roulette to, say, 1000, meaning that each betting round will take just one second, and click "Play" to set this agent running. And for fun, choose one each of the fixed bets (Red, Black, etc) in each of six instances of Gambler. Click "Play" in each of these, and watch while the entire network buzzes with Gambler agents, hopefully all winning at the expense of Roulette. Once in a while, a Gambler may go bust after an exceptional sequence of losses: you can always set a new bank and start again, perhaps changing to a lower risk strategy.

Back Down to Earth

In this chapter, we have explored the creation of a multi-agent application, together with the development of a communications protocol, and introduced techniques such as event broadcasting and timers. So far, however, all of our examples have assumed that each end of a connection is implemented in Chimera; there will be times when it is desired to communicate with agents written in other languages or agent development toolkits: in Chapter 6 we explore exactly how to do just this.

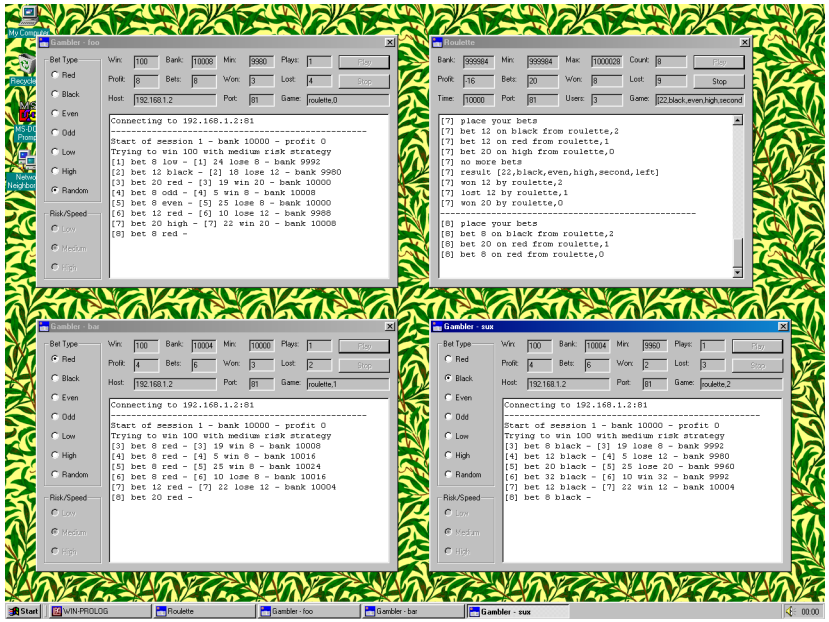


Fig 5.6 - Three Gambler agents running and winning

Chapter 6 - Term Streaming and Transport Syntax

In this chapter, we will discuss how to interface Chimera with agents written in other languages, discussing the concepts of *Term Streaming* and *Transport Syntax*, with particular reference to *KQML* (*Knowledge Query Manipulation Language*).

Lingua Franca

When one agent uses the *agent_post/3* predicate to post an event to another agent, perhaps across a network, the event needs to be converted into some format which is compatible with the network, but which both ends of the connection understand. This *Lingua Franca*, or mutually understood communications language, is critical to the functionality of agents.

In Chimera, an event comprises a ***WIN-PROLOG*** term, which is queued to invoke an event handler when all other events have been processed: the question is, how is this term transmitted over the network?

The answer is really very simple: when posting a term onto the network, a form of Prolog "write" is used to convert it into text; when this text is received at the remote end, a form of Prolog "read" is used to reassemble the term. By using a low-level, binary form of write and read, Chimera allows all terms, including those with atoms and strings containing *Unicode* characters, as well as integers, floating point numbers and compound structures, to be transmitted with absolute accuracy, even across the *ASCII*-only (7-bit character) Internet.

The binary representation used by Chimera is closely related to ***WIN-PROLOG***' *Object File* (.PC) format, which has the advantage of making it quick and efficient both to write and read. Unfortunately, there is one disadvantage: terms transmitted in this form are unlikely to be understood by agents written in other languages, or even by those written in LPA's old Agent Toolkit.

C H I M E Relational A

Term Streaming

By default, Chimera channels all of its input and output through a single predicate, *agent_stream/1*: it is this predicate that encodes terms in the ***WIN-PROLOG*** binary format, and reads them back. Let's experiment with this predicate to see how it works: start up a copy of ***WIN-PROLOG***, and enter the following command to ensure that Chimera system file is loaded, as shown in *Fig 6.1*:

```
| ?- ensure_loaded( system(chimera) ).                    <enter>
yes
```

The *agent_stream/1* predicate works in one of two modes, depending upon the type of its argument: if called with an unbound variable, it attempts to read the term from the *current input stream*, binding it to the variable; if called with any term other than an unbound variable, it writes that term to the *current output stream*. Type the command:

```
| ?- agent_stream( hello(world) ) ~> String.              <enter>
String = `ÿ□~\~L~E~@hello~X~L~E~@world~T`
```

By using the output redirection predicate, *~>/2*, we have diverted the output from *agent_stream/1* into a string, as shown above and in *Fig 6.2*: because this output contains binary data, including control characters, writing it directly to the console window would have had unpredictable effects, such as clearing the console's contents when the form feed character, "~L", is displayed.

Another reason for using string output is that it gives us a convenient way to demonstrate input to agent_stream/1; type the command:

```
| ?- agent_stream( hello(world) ) ~> String,
     agent_stream( Term ) <~ String.                      <enter>
String = `ÿ□~\~L~E~@hello~X~L~E~@world~T` ,
Term = hello(world)
```

Here, we've appended a second call to *agent_stream/1* to our original command, this time with a variable as its argument; we've also used the input redirection predicate, *<~/2* to feed the input from the string we created in the first call: this demonstrates

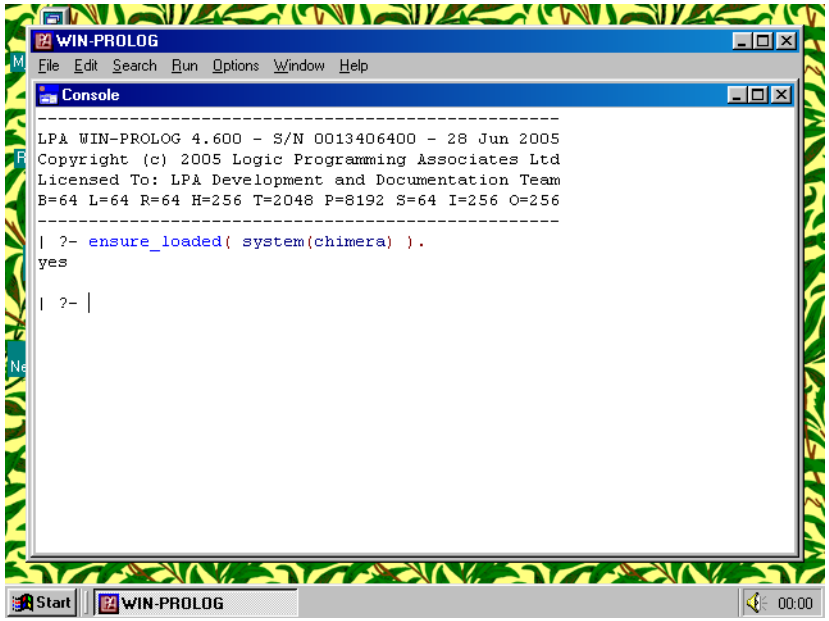


Fig 6.1 - Loading Chimera system file

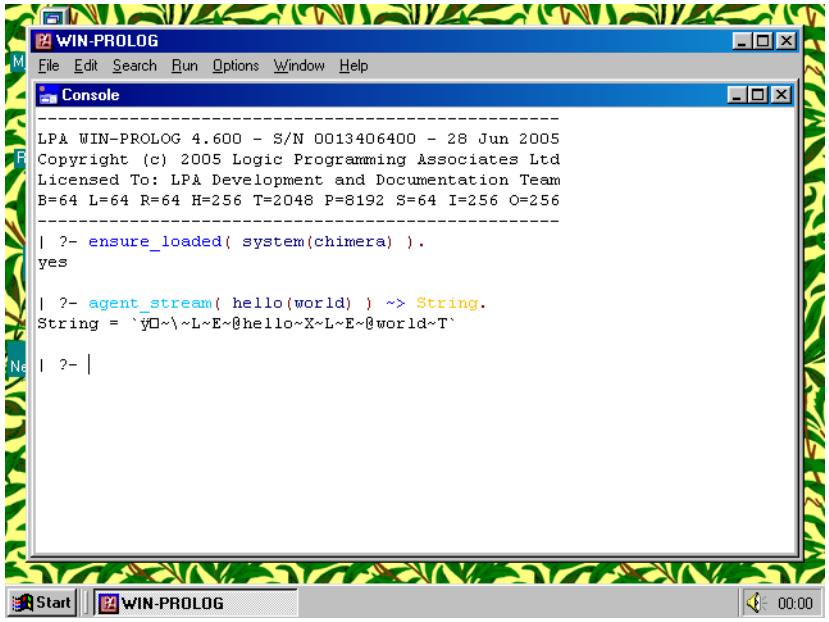


Fig 6.2 - Streaming of a term into a string

how *agent_stream/1* can reconstruct a copy of the original term, as shown above and in *Fig 6.3*.

Redefining Streaming

The reason we've taken such a detailed look at *agent_stream/1*, a predicate which, in all probability, you will never want or need to call directly, is because of the existence of its sister predicate, *agent_stream/2*.

Imagine you wanted to write your own term streaming predicate, rather than use *agent_stream/1*, so that you could communicate with some other agent that didn't understand ***WIN-PROLOG***'s binary object file format: provided that your predicate could write terms into some form of text representation, and read them back in from such text, then you can use *agent_stream/2* to assign your writer/reader to any agent you chose. Let's create a simple example; create a new program window by selecting "File/New", and type in the following definition:

```
streamer( Term ) :-
   (  var( Term )
   -> read( Term )
   ;  writeq( Term ),
      write( ` . ` )
   ).
```

We also need a simple event handler, and we'll make do with our original definition of handler/3 from *Chapter 3*:

```
handler( Name, Link, Event ) :-
   writeq( Name - Link = Event ),
   nl.
```

Once you've entered both of the above pieces of code, as shown in *Fig 6.4*, compile it all ("Run/Compile") and return to the console window. Now we'll create two agents; enter the commands:

```
| ?- agent_create( foo, handler, Port ).                    <enter>
Port = 49152
```

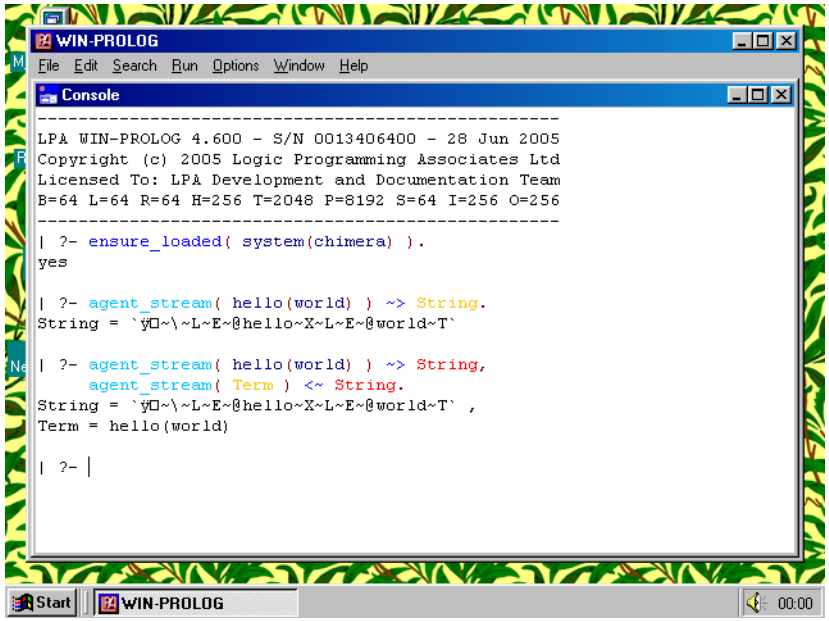


Fig 6.3 - Reconstruction of a term by streaming from a

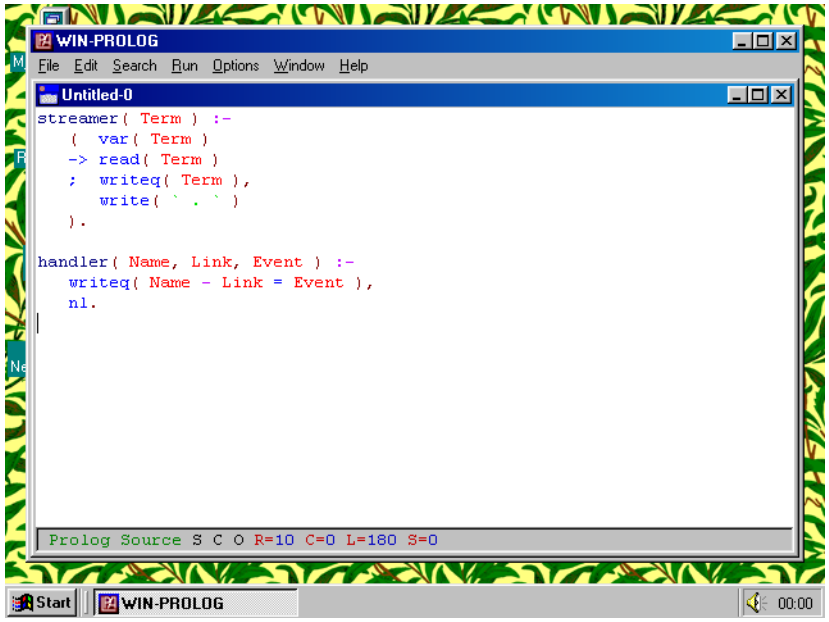


Fig 6.4 - A simple term streamer and event handler

```
| ?- foo - [] = (create,49152)
| ?- agent_create( bar, handler, Port ).                    <enter>
Port = 49153

| ?- bar - [] = (create,49153)
```

Notice how *handler/3* echoes the "*(create,Port)*" event for both agents, as shown above an in *Fig 6.5*. Now, noting the port number returned when we created "foo" ("49152" in the above example), we'll create a link from "bar" to "foo", using this port number; type:

```
| ?- agent_create( bar, Link, `192.168.1.2`, 49152 ).       <enter>
Link = 0

| ?- bar - 0 = (create,`192.168.1.2`,49152)
| ?- foo - 0 = (open,`192.168.1.2`,1025)
```

Again, *handler/3* echoes some more system events, as shown above. Let's check this link is working by posting an event from "bar" to "foo":

```
| ?- agent_post( bar, 0, hello(world) ).                    <enter>
yes

| ?- bar - 0 = (write,`192.168.1.2`,49152,hello(world))
| ?- foo - 0 = (read,`192.168.1.2`,1025,hello(world))
| ?- foo - 0 = hello(world)
```

Two system events are echoed, followed by our user event, "hello(world)", as shown above and in *Fig 6.6*. Now let's have a look at how to use *agent_stream/2*; first, type the following command:

```
| ?- agent_stream( foo, Stream ).                           <enter>
Stream = agent_stream
```

As we can see, when passed a variable as its second argument, this predicate simply returns the name of the arity 1 predicate that is currently assigned to the named

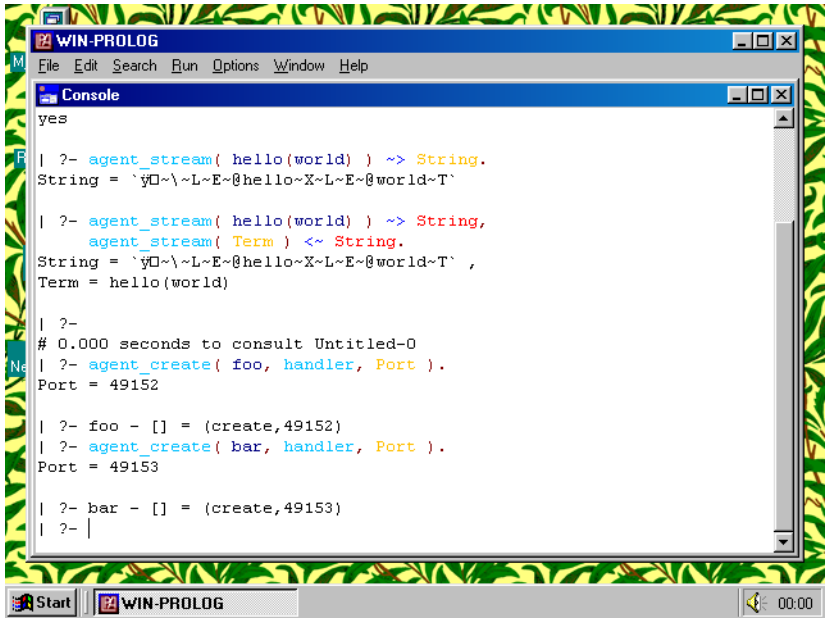


Fig 6.5 - System events echoed after agent creation

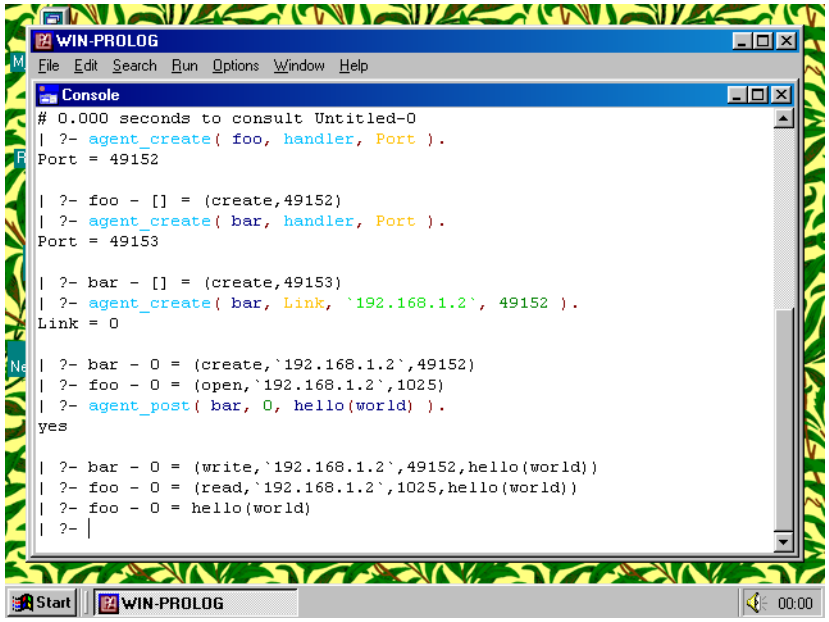


Fig 6.6 - A user event posted between two agents

agent (here, "foo") as its term writer/reader, or *Streamer*: as we'd expect, this command has returned the name of the default streamer, *agent_stream/1*. Now type the commands:

```
| ?- agent_stream( foo, streamer ).                    <enter>
yes

| ?- agent_stream( bar, streamer ).                    <enter>
yes
```

These two calls have assigned our predicate, streamer/1, to each of our agents, "foo" and "bar". Let's try posting an event by re-entering one of our previous commands, which you can find by pressing *<ctrl-pgup>* several times, and then submit by pressing *<enter>*:

```
| ?- agent_post( bar, 0, hello(world) ).               <enter>
yes

| ?- bar - 0 = (write,`192.168.1.2`,49152,hello(world))
| ?- foo - 0 = (read,`192.168.1.2`,1025,hello(world))
| ?- foo - 0 = hello(world)
```

As shown above and in *Fig 6.7*, our simple streamer has worked perfectly, allowing "bar" to post the user event, "hello(world)", to "foo", where it is echoed to the console window by handler/3, along with the usual system events.

KQML: Escape from Prolog

Our simple example, streamer/1, shows how easy it is to write a term streamer for Chimera, but otherwise doesn't achieve very much; all we've done is replace binary input and output with standard Prolog read and write: we're still stuck posting and scheduling Prolog terms. The real power of streamers come into play when handling some other, widely used transport syntax, such as that of the *Knowledge Query Manipulation Language*, better known simply by its acronym, *KQML*.

In some respects, KQML is rather Prolog-like: its syntax is based on that of *Lisp*, allowing simple *ground terms* (those not containing any unbound variables) to be

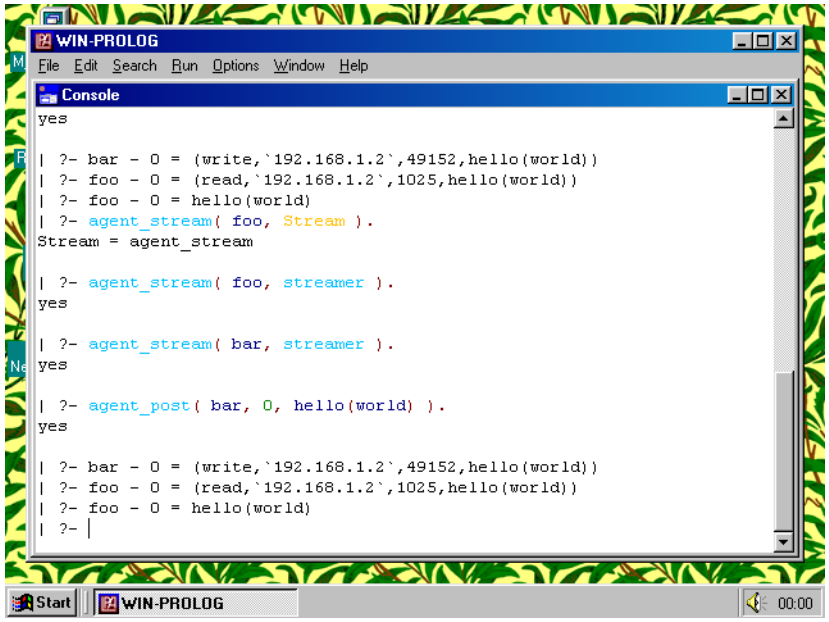


Fig 6.7 - A user event posted using a custom term stream-

represented cleanly and unambiguously. Here is an example of a simple KQML term, also called a *Performative*:

```
(say :hello world)
```

This performative has a name, "say", and a name:value paring denoted by the colon (":") that precedes the name component. You could imagine this structure represented as any number of different Prolog structures, including, for example:

```
[say,':hello',world]

say(':hello'(world))

say(':hello',world)
```

and so on. Once a suitable mapping between KQML performatives and Prolog terms can be agreed upon, all that is needed to make Chimera KQML-friendly is to design a predicate that can output a Prolog term in KQML syntax, and read that term back in from KQML text: a streamer predicate.

Backus Naur Form

The formal syntax of most computer languages is defined in something called *Backus Naur Form*, or *BNF*, and KQML is no exception; here is what this definition looks like:

```
<ascii>          )
<alphabetic>     ) are assumed
<numeric>        ) to be defined
<whitespace>     )
*                  means zero or more of the preceding term
-                  means set exclusion

<performative> ::= (<word> {<whitespace>:<word>
                            <whitespace><expression>}*)

<expression> ::= <word> | <quotation> | <string> |
```

```
                    (<word> {<whitespace><expression>}*)

<word> ::= <character><character>*

<character> ::= <alphabetic> | <numeric> | <special>

<special> ::= < | > | = | + | - | * | / | & | ^ |
              ~ | _ | @ | $ | % | : | . | ! | ?

<quotation> ::= '<expression> | `<comma-expression>

<comma-expression> ::= <word> | <quotation> | <string> |
                       ,<comma-expression> |
                       (<word> {<whitespace>
                                <comma-expression>}*)

<string> ::= "<stringchar>*" | #<digit><digit>*"<ascii>*

<stringchar> ::= \<ascii> | <ascii>-\-<double-quote>
```

The above BNF definition has been slightly corrected with respect to the original draft paper to correct an evident error, and is the version which seems to be most widely adopted.

Given this definition, it would be fairly straightforward to write a parser for KQML, thanks to *WIN-PROLOG*'s support for *Direct Clause Grammars*, or *DCGs*: however, DCGs work by processing a list of characters that has already been physically assembled, and not by reading from the current input stream, and as such, are not appropriate as an input mechanism for Chimera. Furthermore, DCGs cannot be used for output, so another solution needs to be found.

Direct Term Input and Output

While any given piece of BNF can be mapped fairly easily into a DCG, thankfully it is just as easy to interpret the BNF directly into a term input predicate. Take, for example, one entry in the KQML BNF definition:

```
<word> ::= <character><character>*
```

From this, we can see that a "word" comprises at least one character, followed by zero or more characters; we could write some code such as this:

```
read_word( Word ) :-
   read_character( Head ),
   read_characters( Tail ),
   atom_chars( Word, [Head|Tail] ).
```

Each fragment of the BNF definition can be matched by a similar piece of Prolog code, resulting in a program which, while resembling the BNF that it implements, works directly from the current input stream.

Similarly, we can define an output predicate with the nearly identical structure:

```
write_word( Word ) :-
   atom_chars( Word, [Head|Tail] ),
   write_character( Head ),
   write_characters( Tail ).
```

This is exactly the technique that we have used to create the file, *KQML.PL*, which is included in ***WIN-PROLOG***'s "*EXAMPLES\CHIMERA*" directory: two parallel programs, qr_performative/1 and qw_performative/1, perform KQML term reading and writing respectively; these in turn are called by the predicate, kqml_stream/1, which has all the properties of a Chimera term streamer.

Assuming that our two agents, "foo" and "bar" are still running, together with streamer/1, use the "*File/Open*" menu to locate the "*EXAMPLES\CHIMERA*" directory, and then load KQML.PL, compile it with "*Run/Compile*", and return to the console window and type the commands:

```
| ?- agent_stream( foo, kqml_stream ).                    <enter>
yes

| ?- agent_stream( bar, kqml_stream ).                    <enter>
yes
```

These commands will have assigned *kqml_stream/1* to be the term streaming predicates for agents "foo" and "bar". Now let's see how well it works, but we will need to post a "KQML-friendly" event, which means sending a performative; we gave a simple example of one of these earlier:

```
(say :hello world)
```

The kqml_stream/1 predicate maps this structure into the Prolog term:

```
say(':hello',world)
```

so let's try posting this as an event from "bar" to "foo":

```
| ?- agent_post( bar, 0, say(':hello',world) ).          <enter>
yes

| ?- bar - 0 = (write,`192.168.1.2`,49152,say(':hello',world))
| ?- foo - 0 = (read,`192.168.1.2`,1025,say(':hello',world))
| ?- foo - 0 = say(':hello',world)
```

As before, a series of events are displayed, indicating the successful posting of our event, as shown above and in *Fig 6.8*.

Since everything seems to work just as before, how can we show that we are really using KQML to transport our events? If we had another KQML agent to hand, we could try linking to that, but the simplest trick is to output the raw data to a window.

The KQML Raw Example

Sitting alongside *KQML.PL* in ***WIN-PROLOG***'s "*EXAMPLES\CHIMERA*" directory is another example program, *KQML_RAW.PL*. This is a curious hybrid, containing two agents: one, "kqml_agt", is written using the Chimera predicates; the other, "kqml_raw", is implemented directly in terms of the ***WIN-PROLOG*** Winsock predicates.

The first of these agents, kqml_agt, acts as a server, reacting to just three events:

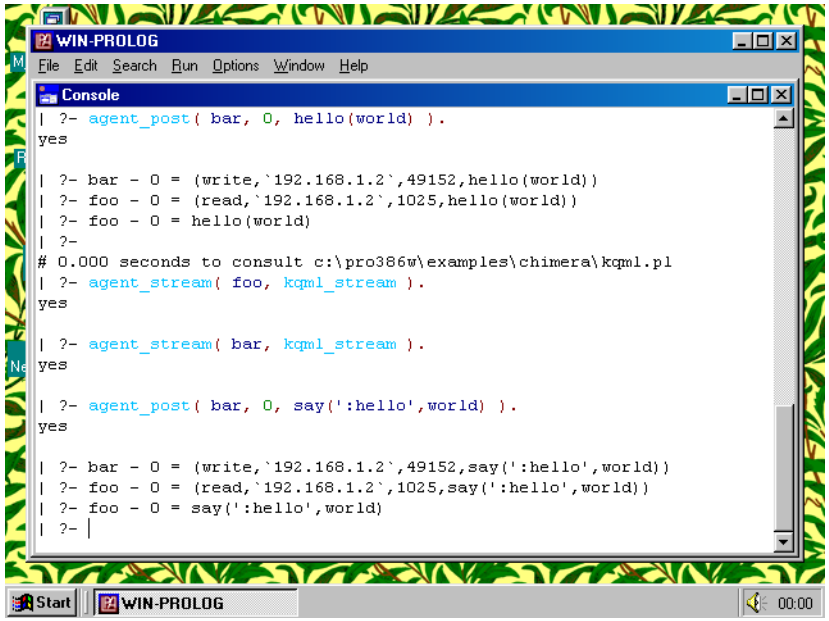


Fig 6.8 - Result of posting a KQML event

Event	Action
hello(|Args)	output debugging information and post the event, *'hi-there'(|Args)*
goodbye(|Args)	output debugging information and post the event, *'ta-ta-for-now'(|Args)*
Any(|Other)	output debugging information and post the event, *unknown(':ontology',agent,':content',String)*

Meanwhile, the second agent, kqml_raw, acts as a client, providing a predicate, kqml_raw/1, which encodes the first two events above directly as *WIN-PROLOG* strings, and allows you to try sending any other KQML of your own:

```
kqml_raw( hello ) :-
   kqml_raw_send( `(hello :ontology agent)` ).

kqml_raw( goodbye ) :-
   kqml_raw_send( `(goodbye :ontology agent)` ).

kqml_raw( String ) :-
   string( String ),
   kqml_raw_send( String ).
```

Use the "*File/Open*" to load *KQML_RAW.PL*, "*Run/Compile*" to compile it, and then return to the console window; now type:

```
| ?- kqml_agt( 123 ).                                    <enter>
yes
```

This will start kqml_agt running on TCP/IP port "123", as shown above: if this call generates an error, just try again, using another port number. Next, we'll start kqml_raw: in order to do this, you need to know your machine's IP address (by running *IPCONFIG* or *WINIPCFG* as described in *Chapter 3*) or your machine's domain name. Assuming your machine is running at IP address, "192.168.1.2", and that "kqml_agt" is running on port 123, enter the command:

```
| ?- kqml_raw( `192.168.1.2`, 123 ).                    <enter>
yes
```

Now that your two agents are up and running, you can post a raw KQML by calling kqml_raw/1, such as with the following command:

```
| ?- kqml_raw( hello ).                                 <enter>
kqml_raw - message `(hello :ontology agent)` sent
yes

| ?- kqml_agt - event "hello" with arguments
                                    [':ontology',agent] received
| ?- kqml_raw - message `(hi-there :ontology agent)` received
```

The kqml_raw/1 predicate displays some information directly to the console before succeeding; two subsequent events are later displayed, one by kqml_agt, and the other by kqml_raw, as shown above an in Fig 6.9. Let's try posting our own raw KQML performative, say:

```
(foo :bar #3"sux)
```

This contains the performative name, "foo", a value name, ":bar", and a three-character string, "sux", in this case written using the "#<*digit*><*digit*>*"<*ascii*>* format specified in the KQML BNF. Type the command:

```
| ?- kqml_raw( `(foo :bar #3"sux)` ).                   <enter>
kqml_raw - message `(foo :bar #3"sux)` sent
yes

| ?- kqml_agt - event "unknown" foo(':bar',`sux`) received
| ?- kqml_raw - message `(unknown :ontology agent :content
                          "foo(':bar',``sux``)")` received
```

Again, kqml_raw/1 directly displays its action to the console, before succeeding; two events follow as before, as above and by *Fig 6.10*.

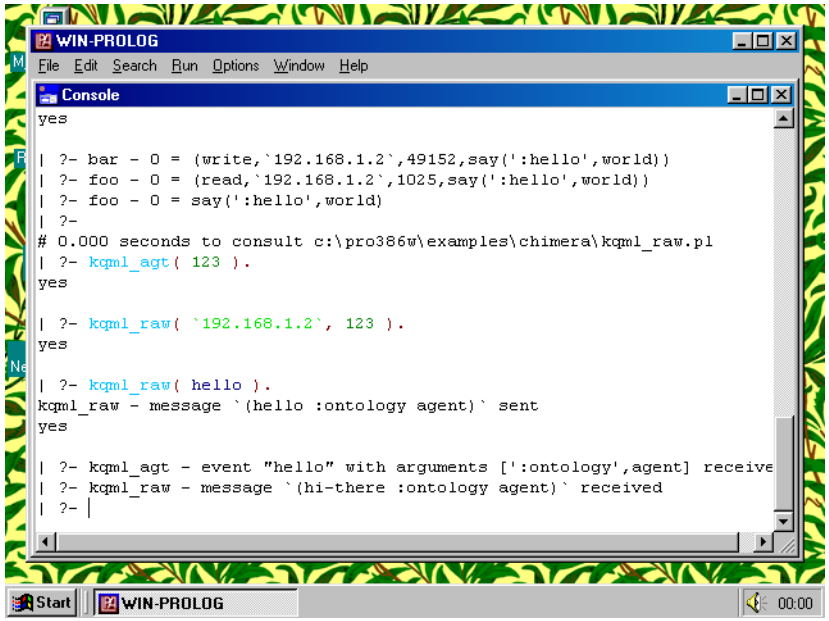


Fig 6.9 - Raw KQML being sent and returned as events

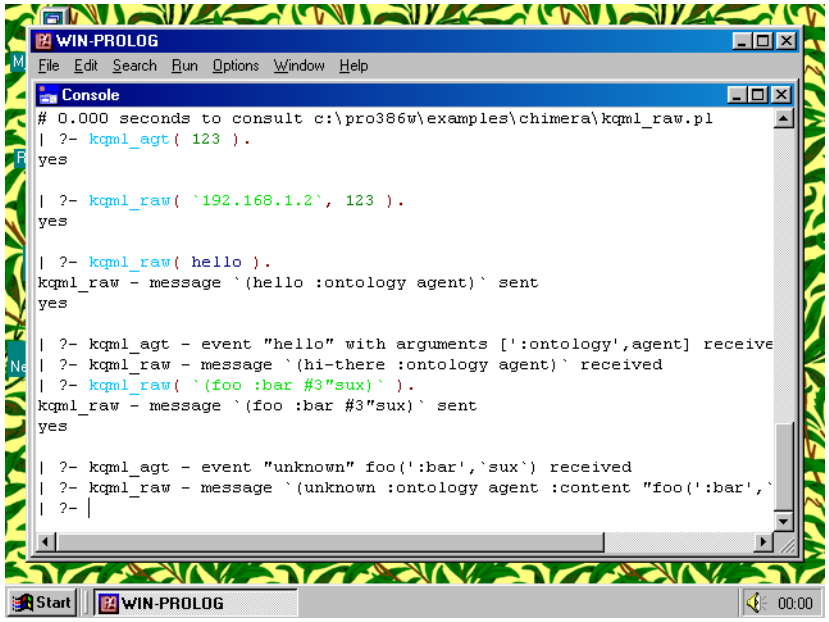


Fig 6.10 - Sending a custom KQML performative

Transport Syntax versus Agent Semantics

Throughout this chapter, we have been concerned with *Transport Syntax*, and not *Agent Semantics*: how events are represented, and not what one is supposed to do them. KQML, of course, is much more than just a transport syntax, although it can be used as one, as we have demonstrated. The semantics of KQML are beyond the scope and intention of this manual, but if you are interested in finding out more, a quick *Google* on the term, "KQML", will locate plenty of material.

Meanwhile, nestling within a subdirectory of ***WIN-PROLOG***'s "*EXAMPLES\CHIMERA*" directory is a detailed KQML example which simulates multiple traders buying and selling on a stock exchange, using rules developed in *flex*, LPA's *Expert System* toolkit. This multiple agent example includes many of the features of a typical KQML distributed application.

Almost There

We've learned how to change the low-level mechanism used by Chimera to communicate events between agents, and have seen how simple it should be to adapt this to handle any arbitrary syntax, using KQML as an example. The one remaining subject to discuss is the debugging of agent applications, and that's what we'll explore in *Chapter 7*.

Chapter 7 - Debugging Agents: CHITRACE

In this chapter, we will be looking at issues relating to the debugging of agent applications, and how to intercept events for this purpose, without affecting the running application, with particular reference to an example file, *CHITRACE.PL*.

Classical Prolog

Let's start by looking at how we might set about debugging a classical Prolog program, say *member/2*: this traditional example is written as two clauses:

```
member_of( Head, [Head|_] ).

member_of( Head,[_|Tail] ) :-
   member_of( Head, Tail ).
```

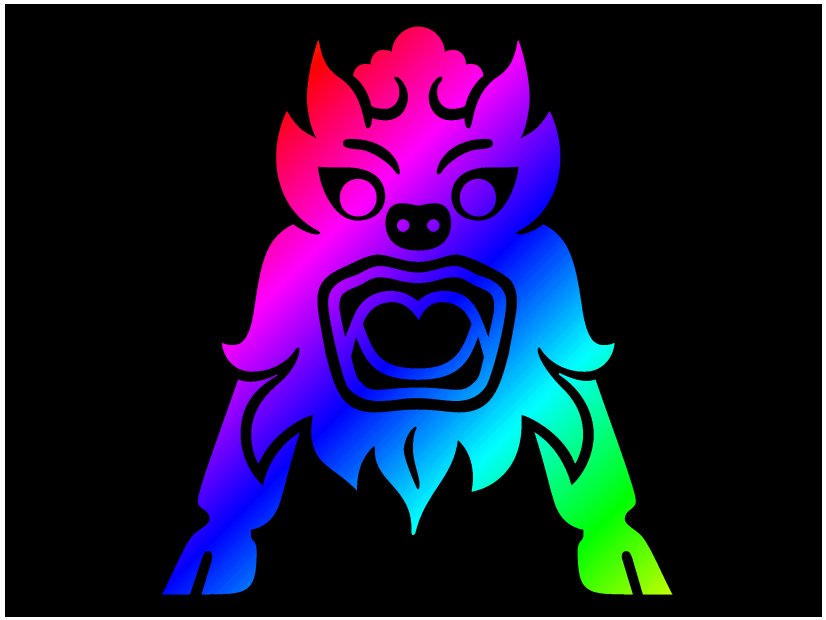
Note that in the above code, we've renamed the predicate to "member_of/2": this is because *member/2* is a built-in predicate in ***WIN-PROLOG***, and we want to avoid a clash of names.

Debugging programs like this is a well-rehearsed process, usually revolving around some form of execution trace, perhaps using ***WIN-PROLOG***'s excellent *Source Level Debugger*, or maybe some other tracer, as we shall use here. Time to get dirty again: start up a copy of ***WIN-PROLOG***, and enter the following command to ensure that Chimera system file is loaded, as shown in *Fig 7.1*:

```
| ?- ensure_loaded( system(chimera) ).                    <enter>
yes
```

Create a program window ("*File/New*"), and type in the definition of "member_of/2" as shown above, and compile it ("*Run/Compile*") before returning to the console. Now enter the following command:

```
| ?- member_of( 3, [1,2,3,4,5] ).                         <enter>
yes
```



C H I M E R Agents

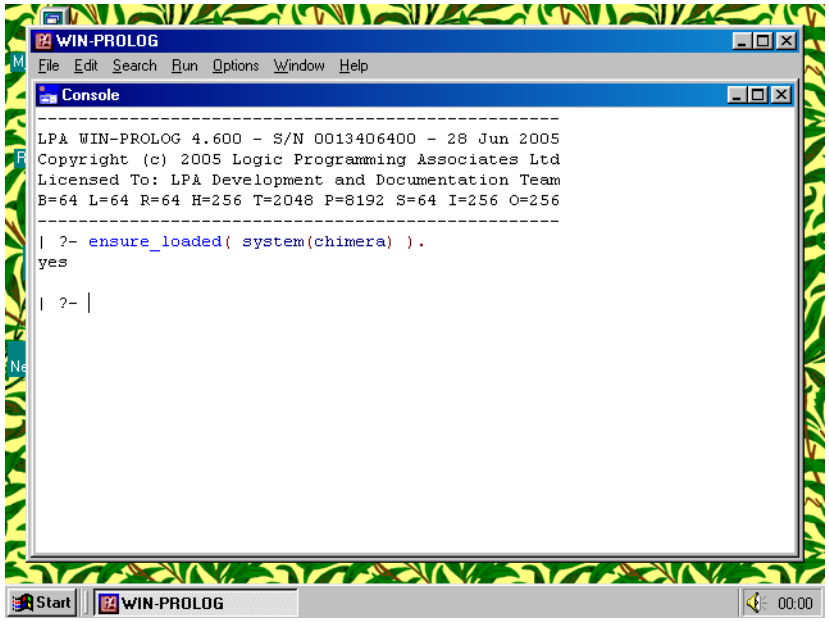


Fig 7.1 - Loading Chimera system file

Our program succeeds, confirming that "3" is a member of the list, "[1,2,3,4,5]"; now try the command:

```
| ?- member_of( 8, [1,2,3,4,5] ).                    <enter>
no
```

This time our program fails, as we might expect, as shown above and in *Fig 7.2*.

The TRACE.PL Library Utility

Imagine that we wanted to see how this simple program worked, perhaps just to give ourselves some confidence in its correctness, or perhaps to spot a bug: we could set *WIN-PROLOG* to *trace mode*, and then rerun the above commands, but this time we're going to look at a different utility, contained in the source-code library utility, *TRACE.PL*: load this file now, with the command:

```
| ?- ensure_loaded( library(trace) ).                <enter>

# 0.000 seconds to consult c:\pro386w\library\trace.pl
yes
```

This file contains a predicate, *?/1*, which can be used to perform a simple execution trace; a sister predicate, *?/2*, does the same job, but allows you to specify an output file to capture the trace: we'll be using the former in a moment. Both trace predicates are controlled very simply, using just four keys:

Key	Action
<enter>	skip this call (execute without tracing)
<space>	trace this call (recursively enters the program)
<esc>	abort the trace (stops the program)
<scroll lock>	if ON, run trace without waiting for keyboard input; if OFF, wait for a keystroke at each stage

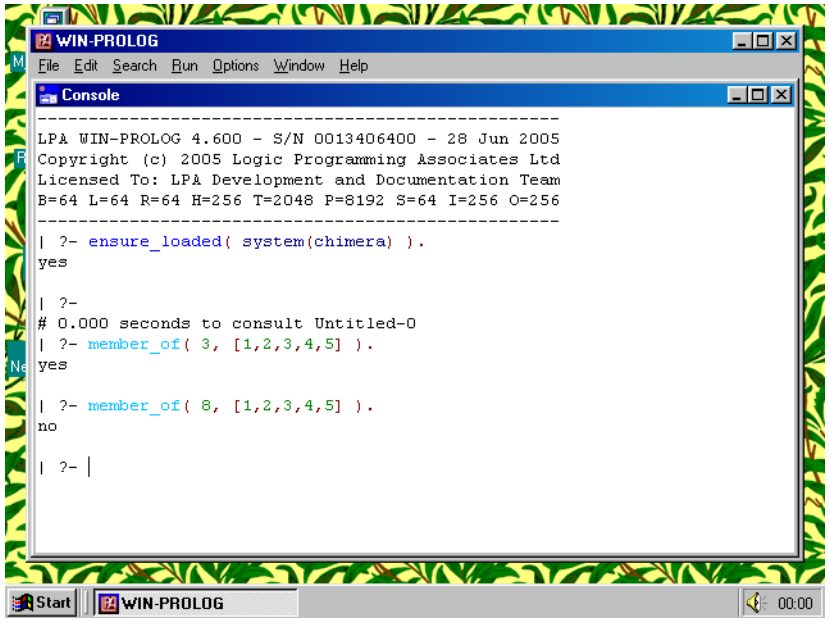


Fig 7.2 - Success and failure of member_of/2

For now, let's use the tracer in its "auto-running" mode, by pressing *<scroll lock>* as needed to set its mode to "on": this will save pressing *<space>* at the end of every line in the trace. Now type in following command:

```
| ?- ?( member_of( 3, [1,2,3,4,5] ) ).                    <enter>
C member_of(3,[1,2,3,4,5])
2 member_of(3,[1,2,3,4,5])
| C member_of(3,[2,3,4,5])
| 2 member_of(3,[2,3,4,5])
| | C member_of(3,[3,4,5])
| | 1 member_of(3,[3,4,5])
| | S member_of(3,[3,4,5])
| S member_of(3,[2,3,4,5])
S member_of(3,[1,2,3,4,5])
yes
```

If *<scroll lock>* is enabled ("on"), a simple trace of your query will have appeared, as shown above and in *Fig 7.3*; if you seem to be stuck at the first line, try pressing *<scroll lock>* again, or simply press *<space>* repeatedly to step through the execution, line by line.

In this trace, "C" means "call", "S" means "succeed", and the numbers, "1" and "2", tell you which clauses of member_of/2 has been matched. Each successive level of call is indented to the right, using the separator, "| ", to give you a feel for how deep into the program you are: these separators also help you match up any given call ("C") with its eventual success ("S"). Now try the command:

```
| ?- ?( member_of( 8, [1,2,3,4,5] ) ).                    <enter>
C member_of(8,[1,2,3,4,5])
2 member_of(8,[1,2,3,4,5])
| C member_of(8,[2,3,4,5])
| 2 member_of(8,[2,3,4,5])
| | C member_of(8,[3,4,5])
| | 2 member_of(8,[3,4,5])
| | | C member_of(8,[4,5])
| | | 2 member_of(8,[4,5])
| | | | C member_of(8,[5])
```

Fig 7.3 - A successful trace of member_of/2

```
| | | | 2 member_of(8,[5])
| | | | | C member_of(8,[])
| | | | | F member_of(8,[])
| | | | F member_of(8,[5])
| | | F member_of(8,[4,5])
| | F member_of(8,[3,4,5])
| F member_of(8,[2,3,4,5])
F member_of(8,[1,2,3,4,5])
no
```

Again, assuming <*scroll lock*> is enabled, or that you have pressed <*space*> a number of times, a trace will appear, as shown above and in *Fig 7.4*.

In this trace, "F" means "fail", and the absence of any number "1" shows why: at no point did we ever match on the first clause of member_of/2.

Sequential and Asynchronous Execution

The reason for the above foray into classical Prolog programming and execution tracing is to provide some sort of framework within which to discuss issues relating to the debugging of agents.

Using *TRACE.PL*, we can easily trace a given call, watching as the program matches clauses, calling itself recursively until it finally succeeds or fails. This is possible for one simple reason: classical Prolog programs are executed in a predictable, top-down, left-recursive mode, guaranteeing *sequential execution*: our tracer simply has to implement a Prolog interpreter and execute the program, outputting details of individual calls along the way.

Unfortunately, the nature of event-driven applications, such as agents written in Chimera, is that they do not follow a predictable, interpreter-friendly path: indeed, the sequence of events that takes place may be, to all intents and purposes, random, leading to unpredictable, *asynchronous execution*. Let's write another program; create a fresh program window ("*File/New*"), and start to enter the following code:

```
member_handler( Name, Link, member(Head,[Head|_]) ) :-
   agent_post( Name, Link, true(Head) ).
```

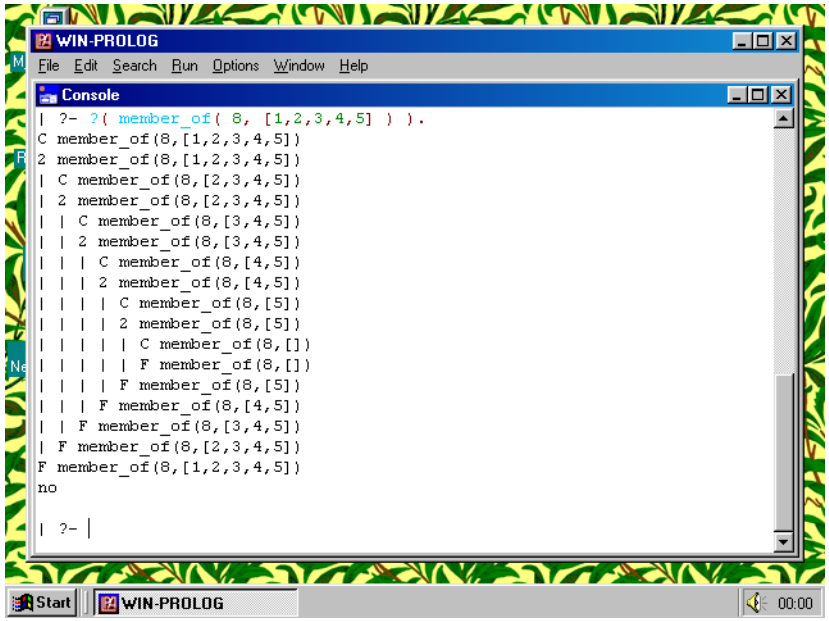


Fig 7.4 - A failed trace of member_of/2

```
member_handler( Name, Link, member(Head,[_|Tail]) ) :-
   agent_post( Name, Link, member(Head,Tail) ).
```

As you will recognise, this program is a Chimera *event handler*, and yet it resembles our earlier program, member_of/2, in that it matches two cases: the first clause handles the user event, "member(Head,[Head|_])", posting a new event, "true(Head)", while the second clauses handles "member(Head,[_|Tail])", posting the event, "member(Head,Tail)".

As it stands, this program is incomplete: firstly, we have no case for "true(Head)", and we also have no way of signalling failure. This is not an issue in classical Prolog programming: the mere absence of a matching case signals failure, and we don't need to handle success and failure explicitly. In an event-driven environment, there is no linking between calls and their results: everything that we want to return must be posted as an event, and/or processed in situ.

In this example, we need to generate a "fail(Head)" event when neither of the existing "member(...)" events matches, and we have to add cases for both this and the "true(Head)" event that we generate when the first clause matches. Here are the remaining cases:

```
member_handler( Name, Link, member(Head,_) ) :-
   agent_post( Name, Link, fail(Head) ).

member_handler( Name, Link, true(Head) ) :-
   write( true = Head ),
   nl.

member_handler( Name, Link, fail(Head) ) :-
   write( fail = Head ),
   nl.
```

When you have entered all five clauses, as shown in *Fig 7.5*, compile the program ("*Run/Compile*"), and return to the console window, before entering the following command:

Fig 7.5 - An event handler mimicking member_of/2

```
| ?- agent_create( foo, member_handler, Port ).          <enter>
Port = 49152
```

This will command will have created an agent, "foo", using our rather strange event handler, member_handler/3, at a dynamically assigned TCP/IP port, probably "49152", as shown here. In this example, the actual port number is unimportant, because we're not going to be creating any connection links. Let's post an event to ourselves to see whether this application works; type the command:

```
| ?- agent_post( foo, [], member(3,[1,2,3,4,5]) ).       <enter>
yes
```

As ever with posted events, the command returns immediately: we don't know whether or not it has "succeeded"; a moment later, we should see a result, as shown below and in *Fig 7.6*:

```
| ?- true = 3
```

This output has been displayed as the result of an *event sequence*, in which each event that was matched correctly posted a further event, ultimately finishing with the "true(Head)" case which simply performed some output, but no further postings. Now try the following command:

```
| ?- agent_post( foo, [], member(8,[1,2,3,4,5]) ).       <enter>
yes
```

Again, the command succeeds immediately, even though we are expecting our event sequence to fail. In due course, we get our desired result:

```
| ?- fail = 8
```

Debugging the Event Sequence

Once again, imagine that we wanted to see how our program works: with our previous example, member_of/2, we were able to trace its execution with the *?/1* predicate in the library utility, *TRACE.PL*. Let's see how that works here; type the command:

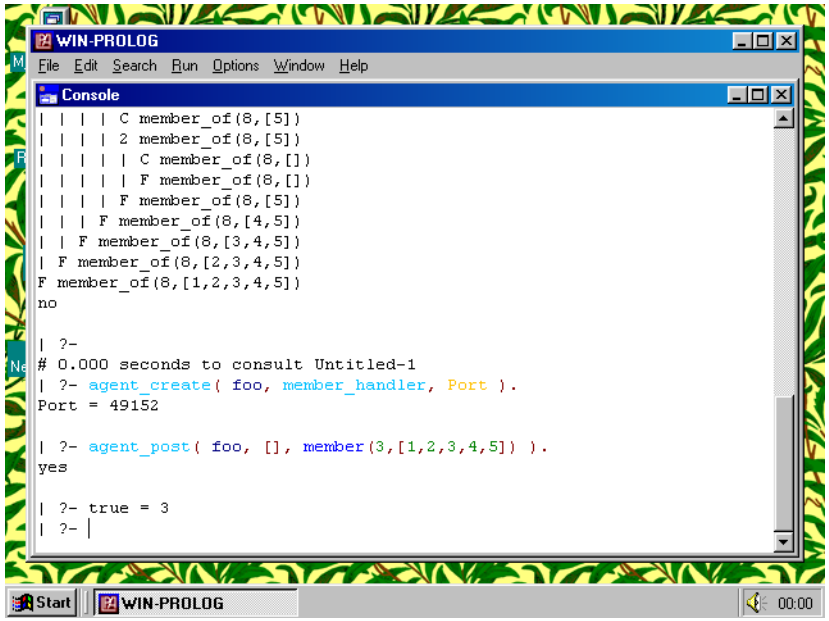


Fig 7.6 - Delayed result from a successful event sequence

```
| ?- ?( agent_post( foo, [], member(3,[1,2,3,4,5]) ) ).  <enter>
C agent_post(foo,[],member(3,[1,2,3,4,5]))
S agent_post(foo,[],member(3,[1,2,3,4,5]))
yes
```

Assuming *<scroll lock>* is still enabled, you will see the above trace of your event posting: this is not very instructive, since all it has done is confirm that the *agent_post/3* predicate has succeeded. A moment later, you will see result of your posting, as shown here and in *Fig 7.7*:

```
| ?- true = 3
```

The *?/1* predicate in *TRACE.PL* is fine for tracing sequential execution, but cannot follow the asynchronous execution of an event sequence. What we need is something which intercepts the individual events, displaying information about them so that we can get a feel for how our program is working.

Let's close down "foo" for now, because we'll be modifying its event handler in a few moments; enter the command:

```
| ?- agent_close( foo ).                                  <enter>
yes
```

Our agent, "foo", will have closed, and its resources and data structures returned to the system pool for recyling.

The CHITRACE.PL Example Utility

Just as ***WIN-PROLOG*** includes the utility, *TRACE.PL*, Chimera has its own source code trace utility, called *CHITRACE.PL*. You can find this file in the "EXAMPLES\CHIMERA" directory: use "*File/Open*" to locate and load this file now, so that it opens into a program window as shown in *Fig 7.8*.

Like so many of the programs that accompany Chimera, *CHITRACE.PL* is written in the form of an event handler, called *trace_agent/3*. It contains just one clause, which is written as follows:

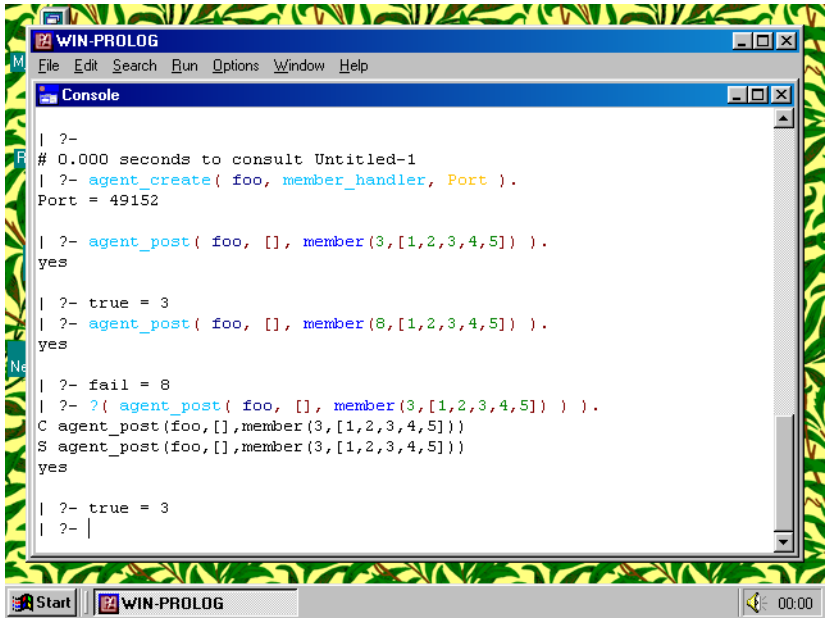


Fig 7.7 - Attempting to trace an event posting

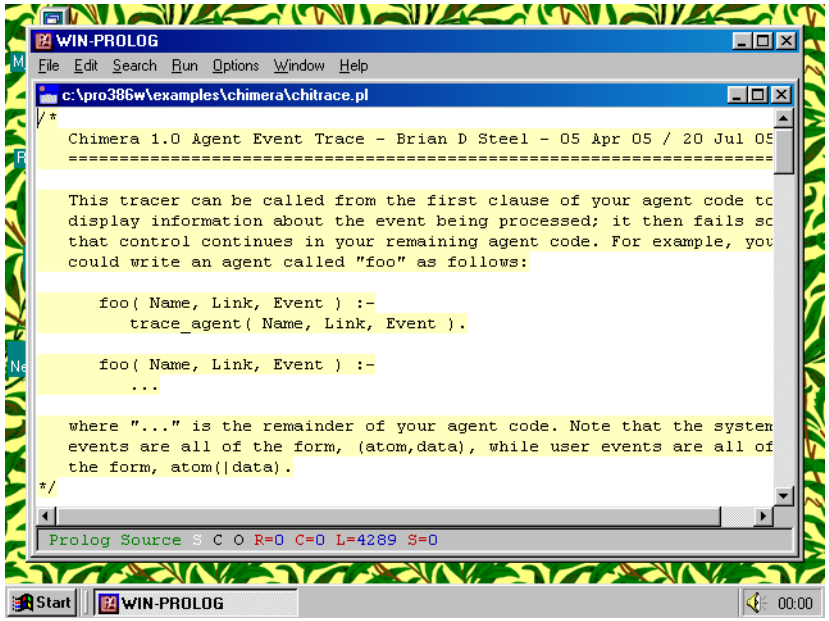


Fig 7.8 - The CHITRACE.PL source code event trace utility

```
trace_agent( Name, Link, Event ) :-
   (  type( Name, 3 ),
      (  type( Link, 1 )
      ;  type( Link, 5 )
      )
   -> (  trace_event( Name, Link, Event )
      -> fail
      )
   ).
```

The calls to *type/2* are simply filters to check that the "Name" is an atom (type "3"), and that the link is an integer or an empty list (type "1" or "5" respectively): assuming the types match, a deterministic call is made to trace_event/3, followed by a call to *fail/0* to force failure. The reason for this failure will become apparent in a few moments.

We don't need to delve much deeper into the code, but let's look at one case from trace_event/3:

```
trace_event( Name, [], (close,Port) ) :-
   trace_write( Name,
                [`Closed agent `,Name,` on port `,Port]
              ).
```

This particular case picks up the "*(close,Port)*" event, which occurs after an agent has been closed with *agent_close/1*. All this case does is call another predicate, trace_write/2, to display a debugging message into a window. The other cases for trace_event/3 simply do similar jobs for each of the remaining system events, as well as one final case for user events.

We can actually use *trace_agent/3* as an event handler, although any agent defined in this way won't do anything useful; for example, compile the *CHITRACE.PL* window ("*Run/Compile*"), return to the console, and type the command:

```
| ?- agent_create( foo, trace_agent, Port ).          <enter>
Port = 49152
```

This will create an agent, "foo", using the next available dynamic port, probably "49152", and you will notice that the ***WIN-PROLOG*** main window has been reorganised to display the console window in the top part, and a new window, "foo trace", in the bottom part; within "foo trace", there will be a single line of text, as shown here and in *Fig 7.9*:

```
Wed 20 Jul 2005 00:00:00 Created agent foo on port 49152
```

The current date and time are displayed, together with some simple text explaining what event has occurred. Let's type another command into the console:

```
| ?- agent_post( foo, [], hello(world) ).                    <enter>
yes
```

As ever, our command returns immediately, but this time we can see what event occurred by checking in "foo trace", which will contain a second line of text, as shown here an in *Fig 7.10*:

```
Wed 20 Jul 2005 00:00:00 Event hello with data [world]
                         scheduled on link []
```

Finally, let's close agent "foo":

```
| ?- agent_close( foo ).                                     <enter>
yes
```

A third line appears in "foo trace", as follows:

```
Wed 20 Jul 2005 00:00:00 Closed agent foo on port 49152
```

When we have seen enough, we can simply click on the "[X]" icon in "foo trace", and the window will close automatically.

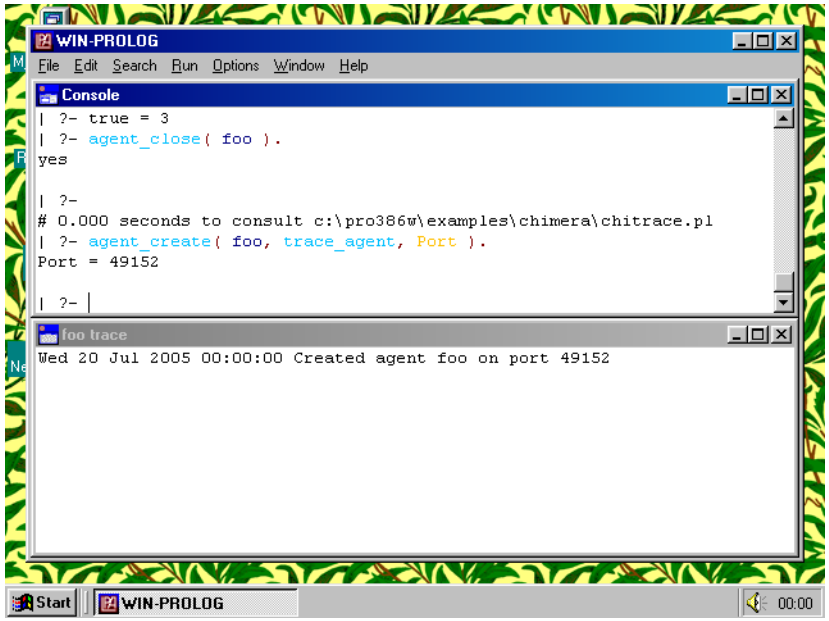


Fig 7.9 - A trace window created by trace_agent/3

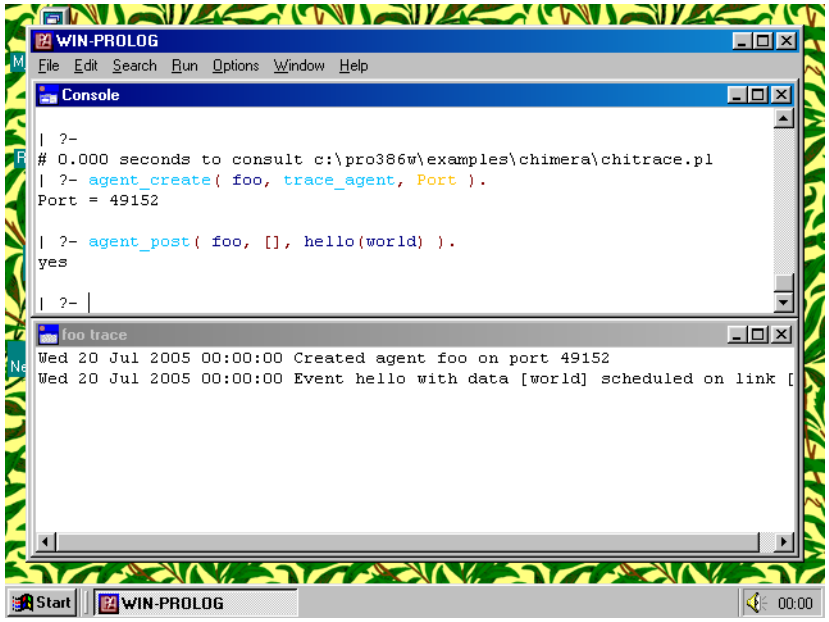


Fig 7.10 - A user event displayed in a trace window

Failure with a Purpose

A little earlier, while looking at code for *trace_agent/3*, we noted that this program is designed to fail on completion; despite this, it worked perfectly well as an event handler: remember, it doesn't matter whether your event handler succeeds or fails, because its execution is detached from any sequential calling process.

But why was *trace_agent/3* written to fail, if it doesn't actually matter? Well, the reason is really quite simple: although this program can be used as an event handler, as we've just seen, this is not its primary purpose; instead, it is designed to be called from an extra, first clause in an existing event handler. By failing, it ensures that any such initial clause backtracks into the remainder of the event handler, so that the original code still gets to process every event. Let's return to the "Untitled-1" window, where we have our code for member_handler/3, and add the following clause at the beginning of this program, as shown in *Fig 7.11*:

```
member_handler( Name, Link, Event ) :-
   trace_agent( Name, Link, Event ).
```

Now compile the program ("*Run/Compile*"), and return the console window. Let's get our agent, "foo", up and running again; use <*ctrl-pgup*> and <*enter*> to locate and resubmit the following command:

```
| ?- agent_create( foo, member_handler, Port ).          <enter>
Port = 49152
```

Once again agent "foo" starts running, but this time, another trace window, "foo trace", appears in the bottom half of the ***WIN-PROLOG*** main window, displaying the following event, as also shown in *Fig 7.12*:

```
Wed 20 Jul 2005 00:00:00 Created agent foo on port 49152
```

As before, we are not worried about the actual port number, because we are not planning to link this agent to any others. Once again use <*ctrl-pgup*> and <*enter*>, this time to locate and reenter the command:

```
| ?- agent_post( foo, [], member(3,[1,2,3,4,5]) ).       <enter>
```

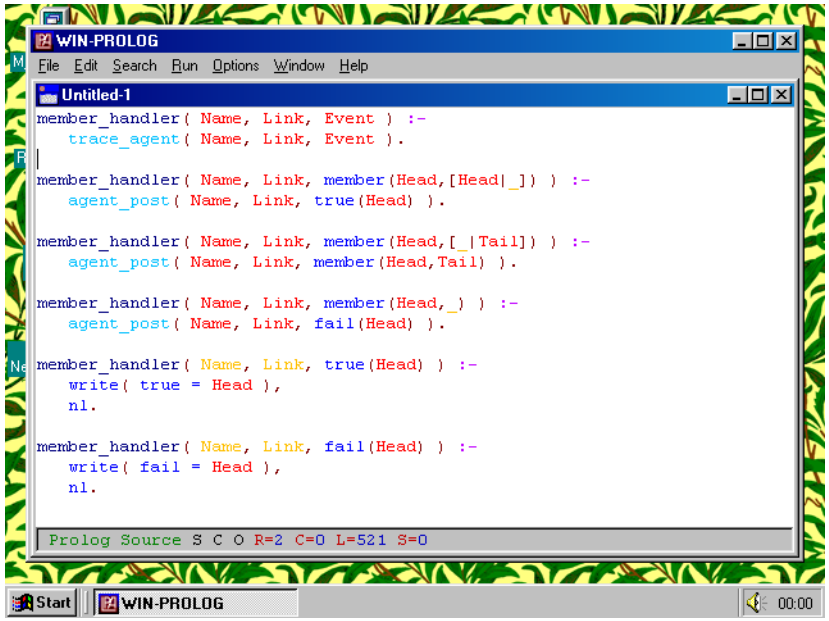


Fig 7.11 - Calling trace_handler/2 from an event handler

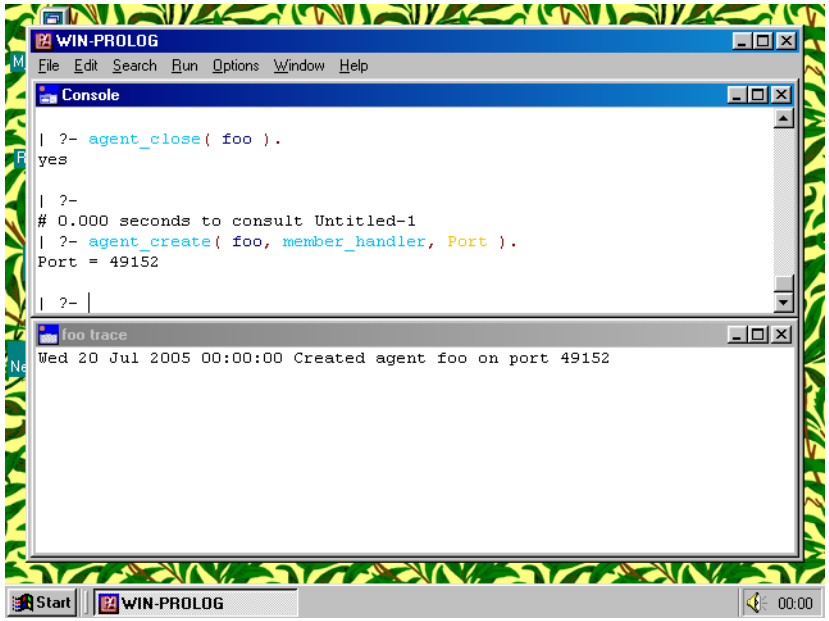


Fig 7.12 - Running an agent with a trace window

```
yes
```

As ever, the command will return immediately, to be followed a moment later by the output from the "true(Head)" case of member_handler/3:

```
| ?- true = 3
```

This time, however, we will also notice some activity in the "foo trace" window, as shown here and in *Fig 7.13*:

```
Wed 20 Jul 2005 00:00:00 Event member with data [3,[1,2,3,4,5]]
                         scheduled on link []
Wed 20 Jul 2005 00:00:00 Event member with data [3,[2,3,4,5]]
                         scheduled on link []
Wed 20 Jul 2005 00:00:00 Event member with data [3,[3,4,5]]
                         scheduled on link []
Wed 20 Jul 2005 00:00:00 Event true with data [3]
                         scheduled on link []
```

Each event that our programs has generated is first sent to *trace_agent/3*, which displays output in a window before failing back into the original event handler. All it has taken to view the entire event sequence for our agent is to add a single clause at the top of our event handler, giving us the general program structure:

```
<handler>( Name, Link, Event ) :-
   trace_agent( Name, Link, Event ).

<handler>( ...
```

where "*<handler>*" is the name of an event handler, and "..." denotes the remainder of the code. When debugging is complete, the first clause can simply be commented out or deleted.

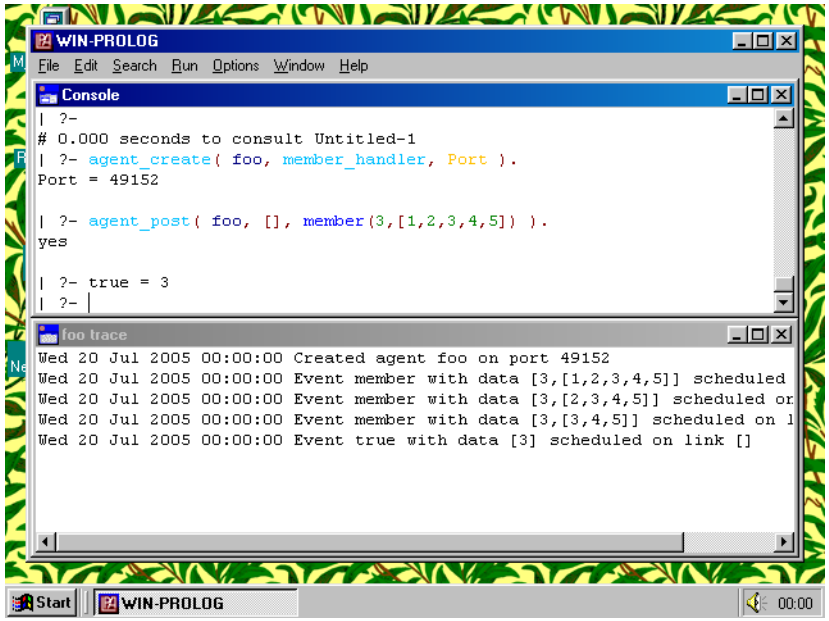


Fig 7.13 - Tracing a successful event sequence

Mixing CHITRACE.PL with TRACE.PL

Sometimes, simply displaying an event sequence might not be enough: you may want to see how an individual event case is processed. There is no reason why you cannot get the best of both worlds: use a call to the *CHITRACE.PL* predicate, *trace_agent/3*, to provide event sequence traces for one or more of your agent event handlers, and embed calls to the one or other of the TRACE.PL predicates, *?/1* and *?/2* into your code. For example, we could write the following event handler:

```
foo_handler( Name, Link, Event ) :-
   trace_agent( Name, Link, Event ).

foo_handler( Name, Link, Event ) :-
   nl,
   write( `Debugging Event` - Event ),
   nl,
   ?( member_handler( Name, Link, Event ) ),
   nl.
```

The first clause calls *trace_agent/2*, which, as we know, fails on completion; the second clause outputs some information to the console window, and then calls *?/1*, causing the execution of our original event handler, member_handler/3, to be traced. Return to the "Untitled-1" window, and delete the clause we added earlier, namely:

```
member_handler( Name, Link, Event ) :-
   trace_agent( Name, Link, Event ).
```

and replace it with the code we've just seen for foo_handler/3, as shown in *Fig 7.14*. Compile the resulting code ("Run/Compile"), and use the "Window" menu to bring the "foo trace" window back into view, before closing it (click the "[X]" icon); next, return to the console window, and make sure that *<scroll lock>* is enabled (on), before typing the command:

```
| ?- agent_create(foo, foo_handler, Port ).          <enter>
Port = 49152
```

This will close the existing instance of agent "foo", before starting a new copy, this

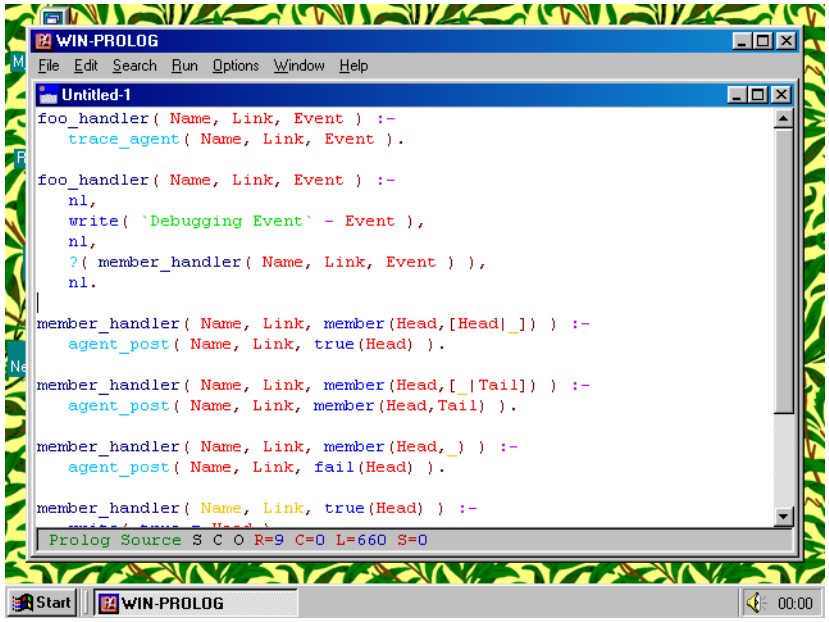


Fig 7.14 - Mixing sequential and asynchronous tracing

time using foo_handler/3 rather than member_handler/3 as the event handler. A new "foo trace" window will appear, and provided you have remembered to turn *<scroll lock>* on as requested, you will spot some additional output in the console window, as shown below and in *Fig 7.15*:

```
Debugging Event - (create,49152)
C member_handler(foo,[],(create,49152))
F member_handler(foo,[],(create,49152))
```

The console output confirms that a "*(create,Port)*" system event has occurred, and that your handler has failed to process it ("F" in the output from *?/1*). As you know, it doesn't matter whether your handler succeeds or fails; moreover, you might also remember that you don't need to process any of the system events: Chimera functions perfectly without any special handling of them. Once again making sure *<scroll lock>* is enabled (on), use *<ctrl-pgup>* and *<enter>*, to locate and reenter the command:

```
| ?- agent_post( foo, [], member(3,[1,2,3,4,5]) ).        <enter>
yes
```

As always, the command returns instantly; this time, however, it will be followed by a series of short traces from ?/1, as shown here:

```
| ?-
Debugging Event - member(3,[1,2,3,4,5])
C member_handler(foo,[],member(3,[1,2,3,4,5]))
2 member_handler(foo,[],member(3,[1,2,3,4,5]))
| C agent_post(foo,[],member(3,[2,3,4,5]))
| S agent_post(foo,[],member(3,[2,3,4,5]))
S member_handler(foo,[],member(3,[1,2,3,4,5]))

| ?-
Debugging Event - member(3,[2,3,4,5])
C member_handler(foo,[],member(3,[2,3,4,5]))
2 member_handler(foo,[],member(3,[2,3,4,5]))
| C agent_post(foo,[],member(3,[3,4,5]))
| S agent_post(foo,[],member(3,[3,4,5]))
```

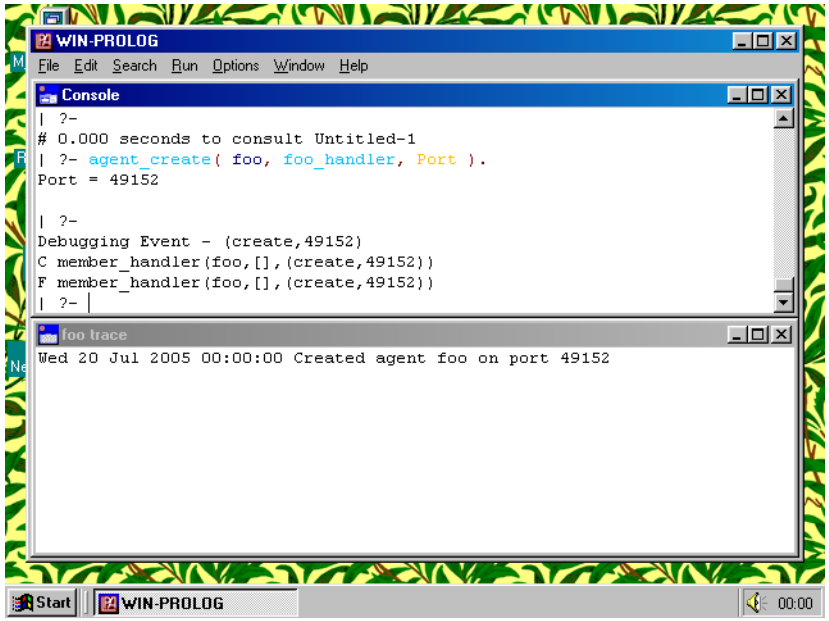


Fig 7.15 - Tracing an unhandled system event

```
S member_handler(foo,[],member(3,[2,3,4,5]))

| ?-
Debugging Event - member(3,[3,4,5])
C member_handler(foo,[],member(3,[3,4,5]))
1 member_handler(foo,[],member(3,[3,4,5]))
| C agent_post(foo,[],true(3))
| S agent_post(foo,[],true(3))
S member_handler(foo,[],member(3,[3,4,5]))

| ?-
Debugging Event - true(3)
C member_handler(foo,[],true(3))
4 member_handler(foo,[],true(3))
| C write(true = 3)
true = 3| S write(true = 3)
| C nl

| S nl
S member_handler(foo,[],true(3))
```

As we can see, our eventual output, "true = 3", is buried within the final part of the trace; meanwhile, the "foo trace" window shows the event sequence just like last time, as shown in *Fig 7.16*.

Postlude

And with that, we draw the narrative section of the Chimera Reference to its conclusion. Over the past seven chapters, we've described what we mean by an agent, and looked at what constitutes one; we've explored Chimera's predicates and events, before putting them into practice in a detailed, multi-agent example. Later, we've looked at transport syntaxes and seen how to adapt Chimera to KQML, and finally, we've delved into debugging.

All that remains now is to tag on a couple of Appendices containing the formal definitions for Chimera's predicates and events, and to sign off by saying that the author hopes you enjoy using Chimera as much as he enjoyed developing it.

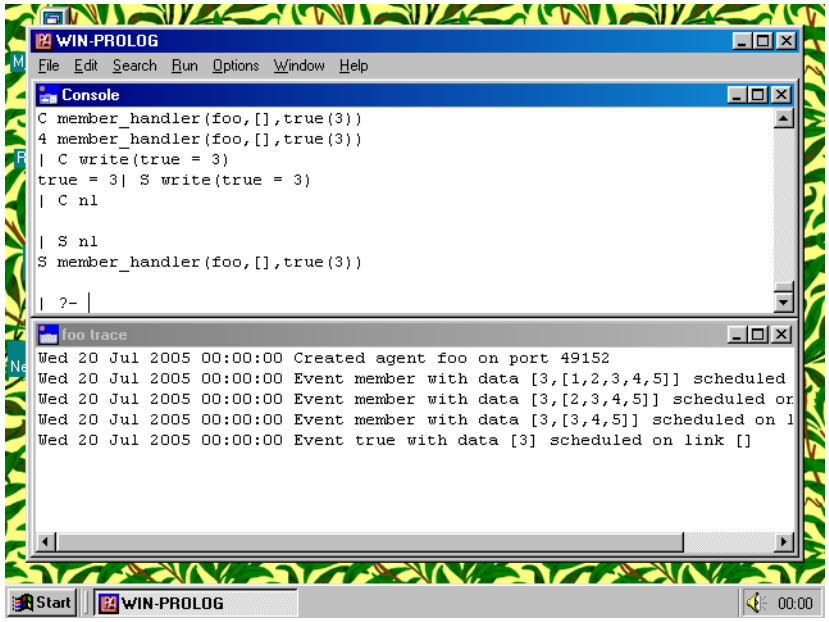


Fig 7.16 - Tracing the event handler in an event sequence

Appendix A - Chimera Predicate Reference

This appendix is arranged alphabetically, and contains formal definitions for the public predicates within Chimera, complete with comments, command-line examples of their use and notes about points of interest; it is designed as a reference section: please see *Chapter 3* for a tutorial overview of these predicates.

agent_close(+Name)

Name <atom>

Comments

Close an agent of the given *Name*, together with any connection links associated with it, and purge all related system and user events.

Examples

The following command creates an agent, "foo", using the predicate, "bar/3", as its event handler, at TCP/IP port "123":

```
| ?- agent_create( foo, bar, 123 ).                    <enter>
yes
```

After the agent has served its purpose, it can be closed with the command:

```
| ?- agent_close( foo ).                               <enter>
yes
```

Notes

Agents created with *agent_create/3* should be closed with *agent_close/1* when their function is complete. Closing an agent is not compulsory, but is a good idea: incoming connection requests will no longer be automatically accepted, data structures and resources will be recycled, and any active links are shut down gracefully.

Note that when you close a listening agent, all links are automatically closed first; moreover, all pending system and user events (those that have been posted to this agent, but not yet processed) are purged (deleted) and cannot later be recovered. As such, it is best to close a listening agent only once all connected links have completed their own processing and been individually closed by *agent_close/2* or by receipt and processing of an "*(error,Code,What)*" or "*(close,Host,Port)*" system event.

agent_close(+Name, +Link)

Name	<atom>
Link	<integer>

Comments

Close an agent link of the given *Name* and *Link* number, and purge all related system and user events.

Examples

The following command creates an agent, "foo", using the predicate, "bar/3", as its event handler, at TCP/IP port "123":

```
| ?- agent_create( foo, bar, 123 ).                    <enter>
yes
```

Assuming this command was successful, the following command creates a connection between this and an agent running on a machine at IP address "192.168.1.2" and at port "456":

```
| ?- agent_create( foo, Link, `192.168.1.2`, 456 ).    <enter>
Link = 0
```

After the agent link has served its purpose, it can be closed with the command:

```
| ?- agent_close( foo, 0 ).                            <enter>
yes
```

Notes

Agent links created with *agent_create/4* should be closed with *agent_close/2* when their function is complete. Closing an agent link is not compulsory, but is a good idea: further incoming events will no longer be received, and data structures and resources will be recycled.

Note that when you close an agent link, all pending system and user events (those that have been posted to this link, but not yet processed) are purged (deleted) and cannot later be recovered. Normally, a link should be closed when it has completed its own processing; in addition, it will be closed automatically in response to an "*(error,Code,What)*" or "*(close,Host,Port)*" system event.

agent_create(+Name, +Pred, ?Port)

Name	<atom>
Pred	<atom>
Port	<integer> in range [0..65535] or <variable>

Comments

Create an agent of the given *Name*, using the arity 3 event handler specified by *Pred*, listening for connections at a TCP/IP *Port*. If *Port* is an integer, an attempt is made to use the port specified; if *Port* is a variable, the first available dynamic port (one in the range [49152..65535]) is used and its integer value bound to the variable. Any previously existing agent of the given name, together with any connection links associated with it, is automatically closed.

Examples

The following command creates an agent, "foo", using the predicate, "bar/3", as its event handler, at TCP/IP port "123":

```
| ?- agent_create( foo, bar, 123 ).                    <enter>
yes
```

If the TCP/IP port assignment is unimportant, the third argument can be left as variable; the following command creates an agent, "sux", again using the event handler, "bar/3", at a dynamically assigned port:

```
| ?- agent_create( sux, bar, Port ).                   <enter>
Port = 49152
```

Notes

Agents created with *agent_create/3* are *listening agents*, or *servers*: this means that, in themselves, they do nothing other than idle away in the background, waiting for connections, or *links*, to be made to other agents, either by local calls to *agent_create/4*, or by *accepting* incoming connections from remote agents.

While it is perfectly possible to use *agent_post/3* to self-post messages to a listening agent, by specifying an empty list ("[]") as its second argument, most activity will occur through links to other agents.

The *event handler* specified in the second argument to *agent_create/3* can be any user-defined predicate with an arity of 3. Its arguments are unified with the agent name (as specified as the first argument of *agent_create/3*), link number (either an integer or, in the case of a self-posted event, an empty list ("[]")) and the event itself. Events take one of two forms: *system events* are generated within Chimera, and are conjunctions of the form:

```
(name,...)
```

while *user events* are the result of calls to *agent_post/3*, and are tuples of the form:

```
name(...)
```

The difference in data type between system and user events allows simple, pattern-based distinction between these two types of event, and also prevents incursions by the system into user name space.

agent_create(+Name, ?Link, +Host, +Port)

Name	<atom>
Link	<integer> or <variable>
Host	<string>
Port	<integer> in range [0..65535]

Comments

Create a connection for the agent of the given *Name*, with the given *Link* number, connecting to the given *Host* and TCP/IP *Port*. The *Host* can be specified as a any of a machine's network name, domain name or IP address. If *Link* is an integer, any existing connection with the given link number is automatically closed; if *Link* is a variable, the first available unused link number is used and its integer value bound to the variable.

Examples

The following command creates an agent, "foo", using the predicate, "bar/3", as its event handler, at TCP/IP port "123":

```
| ?- agent_create( foo, bar, 123 ).                    <enter>
yes
```

Assuming this command was successful, the following command creates a connection between this and an agent running on a machine at IP address "192.168.1.2" and at port "456":

```
| ?- agent_create( foo, Link, `192.168.1.2`, 456 ).    <enter>
Link = 0
```

The following command will directly replace the above link, attempting instead to establish a connection to an agent running on "www.lpa.co.uk" at port "789":

```
| ?- agent_create( foo, 0, `www.lpa.co.uk`, 789 ).     <enter>
yes
```

Although the call has succeeded, after a few moments the attempted connection will close, because (at present) there are no Chimera agents running on this port at LPA's web server.

Notes

Agents created with *agent_create/4* are *active agents*, or *clients*: this means that they have proactively made contact with a listening agent that was created with *agent_create/3*, and established a *link* or connection through which the two agents can communicate. The second argument of agent_create/4 specifies or returns the number of this link, and can be used together with the agent name to form a unique identifier for a particular link.

Links are not created directly with an original, listening agent: instead, when the listening agent receives notification of an incoming link, it automatically *accepts* the connection by creating a local active agent with an unused link number. Apart from when events are self-posted through *agent_post/3*, by specifying an empty list ("[]") as its second argument, most activity will occur through links between active agents.

Note that *agent_create/4* returns immediately with success, even if there is no listening agent running at the specified host and port: this is because Chimera is built entirely around *non-blocking Windows Sockets* (*Winsock*) calls, in order to support its event-driven architecture. If the connection fails, it will normally do so after a period ranging from a few seconds to several minutes, and will be signalled by two system events, first "*(error,Code,What)*" and then "*(close,Host,Port)*": in this case, Chimera will automatically close the link.

agent_data(+Name, -Pred, -Port, -List)

Name	<atom>
Pred	<variable>
Port	<variable>
List	<variable>

Comments

Return data associated with an agent of the given *Name*, binding *Pred* to an atom that names its arity 3 event handler, *Port* to an integer corresponding to its TCP/IP port, and *List* to the list of currently active link numbers.

Examples

The following command creates an agent, "foo", using the predicate, "bar/3", as its event handler, at TCP/IP port "123":

```
| ?- agent_create( foo, bar, 123 ).                    <enter>
yes
```

Assuming this command was successful, the following command creates a connection between this and an agent running on a machine at IP address "192.168.1.2" and at port "456":

```
| ?- agent_create( foo, Link, `192.168.1.2`, 456 ).    <enter>
Link = 0
```

Provided that this is the only link currently defined for this agent, the following command will return information about the agent:

```
| ?- agent_data( foo, Pred, Port, List ).              <enter>
Pred = trace_agent ,
Port = 123 ,
List = [0]
```

Notes

Agents created with *agent_create/3* are maintained in a local database, and *agent_data/4* provides access to their event handler name, port number, and list of current connection links, which can be used for various informational and practical purposes. For example, the following call will close down all current links for a given named agent:

```
| ?- agent_data( foo, _, _, List ),
     forall( member( Link, List ),
             agent_close( foo, Link )
           ).                                                    <enter>
List = [0] ,
Link = _
```

Three predicates allow the combined Chimera local databases to be queried: *agent_dict/2* returns a list of names of current agents, while *agent_data/4* returns the event handler, port number and list of link numbers for any one of these; finally, *agent_data/5* returns the Winsock socket name, remote host address and port number for any given link.

agent_data(+Name, +Link, -Sock, -Host, -Port)

Name	<atom>
Link	<integer>
Sock	<variable>
Host	<variable>
Port	<variable>

Comments

Return data associated with an agent link of the given *Name* and *Link* number, binding *Sock* to an atom that names its Winsock socket, *Host* to a string containing the remote machine name, domain name or IP address, *Port* to an integer corresponding to its TCP/IP port.

Examples

The following command creates an agent, "foo", using the predicate, "bar/3", as its event handler, at TCP/IP port "123":

```
| ?- agent_create( foo, bar, 123 ).                    <enter>
yes
```

Assuming this command was successful, the following command creates a connection between this and an agent running on a machine at IP address "192.168.1.2" and at port "456":

```
| ?- agent_create( foo, Link, `192.168.1.2`, 456 ).    <enter>
Link = 0
```

Using the link number returned from this call, the following command will return information about the agent link:

```
| ?- agent_data( foo, 0, Sock, Host, Port ).           <enter>
Sock = abcdefgh ,
Host = `192.168.1.2` ,
Port = 456
```

Notes

Agent links created with *agent_create/4* are maintained in a local database, and *agent_data/5* provides access to their Winsock socket name, and remote host address and port number, which can be used for various informational and practical purposes.

Three predicates allow the combined Chimera local databases to be queried: *agent_dict/2* returns a list of names of current agents, while *agent_data/4* returns the event handler, port number and list of link numbers for any one of these; finally, *agent_data/5* returns the Winsock socket name, remote host address and port number for any given link.

agent_dict(+Flag, -List)

Flag	<integer> in the domain {-1,0,1}
List	<variable>

Comments

Return a list currently defined agents, as specified by the given visibility *Flag*, and bind the result to *List*.

Examples

The following command creates an agent, "foo", using the predicate, "bar/3", as its event handler, at TCP/IP port "123":

```
| ?- agent_create( foo, bar, 123 ).                    <enter>
yes
```

Provided that this is the only agent currently defined, the following command will return its name in a single element list:

```
| ?- agent_dict( 0, List ).                            <enter>
List = [foo]
```

Notes

Agents created with *agent_create/3* are maintained in a local database, and *agent_dict/2* provides access to the list of their names, which can be used for various informational and practical purposes. For example, the following call will close down all current agents and their links:

```
| ?- agent_dict( 0, List ),
     forall( member( Name, List ),
             agent_close( Name )
           ).                                           <enter>
List = [foo] ,
Name = _
```

Three predicates allow the combined Chimera local databases to be queried: *agent_dict/2* returns a list of names of current agents, while *agent_data/4* returns the event handler, port number and list of link numbers for any one of these; finally, *agent_data/5* returns the Winsock socket name, remote host address and port number for any given link.

The first argument of *agent_dict/2* is a flag which specifies which types of agent name to return, as shown in this table:

Flag	Meaning
0	return agents named by visible atoms
1	return agents named by hidden atoms
-1	return agents named by visible or hidden atoms

The terms, *hidden* and *visible*, refer to an atom's visibility state, which is controlled by the predicate, *hide/2*, which forms part of ***WIN-PROLOG***'s module system. In general, you can ignore the existence of hidden atoms, and call *agent_dict/2* with an argument of "0" (zero); see the ***WIN-PROLOG*** Technical Reference for more information about *hide/2*.

agent_post(+Name, +Link, +Term)

Name	<atom>
Link	<integer> or <empty list>
Term	<tuple>

Comments

Post a user event to the agent link specified by the given *Name* and *Link*: the event is a *Term* whose interpretation is left to the application. If *Link* is an integer, the event is posted to whichever agent is connected at the given link; if *Link* is an empty list ("[]"), the event is self-posted to the agent specified in *Name*.

Examples

The following command creates an agent, "foo", using the predicate, "bar/3", as its event handler, at TCP/IP port "123":

```
| ?- agent_create( foo, bar, 123 ).                    <enter>
yes
```

Assuming this command was successful, the following command creates a connection between this and an agent running on a machine at IP address "192.168.1.2" and at port "456":

```
| ?- agent_create( foo, Link, `192.168.1.2`, 456 ).    <enter>
Link = 0
```

Using the link number returned from this call, the following command will post the user event, "hello(world)", to whichever agent is running on the remote machine:

```
| ?- agent_post( foo, 0, hello(world) ).               <enter>
yes
```

By specifying an empty list ("[]"), rather than a link number, the following command will post the user event, "hello(world)", directly to the named agent:

```
| ?- agent_post( foo, [], hello(world) ).                    <enter>
yes
```

Notes

User events can be posted with *agent_post/3* both to remote agents via links that have been created with agent_create/4, as well as self-posted directly to agents created with *agent_create/3*. These two types of posting are distinguished by the value of the second argument: if it is an integer, the event is posted over the connection associated with the given link number; if it is an empty list ("[]"), the event is posted directly to the named agent.

Posted events are queued when they reach their target, and processed in strict rotation. An agent's event handler, whose name is specified as the second argument in the original call to *agent_create/3* that created the agent, receives the event when it has finished processing all previously posted events.

A user event is simply a ***WIN-PROLOG*** *tuple*, or *compound term*, which has the following most general form:

```
functor(|arguments)
```

where "functor" may be any Prolog term, although usually an atom, and "arguments" is any Prolog term, although usually a list of zero or more Prolog terms; for example:

```
hello(`world`,12345,[a,b,c])
```

consists of the functor, "hello", and a list of three arguments, the string, "`world`", the integer, "12345", and a list containing the three atoms, "a", "b" and "c".

The meaning of user events is left entirely to the receiving agent's event handler. For example, the following piece of code will accept and process the above event, writing some diagnostics to the console before reversing the argument sequence and posting a new event back to the sender:

```
bar( Name, Link, hello(|Args) ) :-
   write( `Event HELLO received with arguments, ` ),
```

```
        writeq( Args ),
        nl,
        reverse( Args, Back ),
        agent_post( Name, Link, olleh(|Back) ).
```

When invoked with the above user event, this event handler would write the following information into the current output stream:

```
Event HELLO received with arguments, [`world`,12345,[a,b,c]]
```

before reversing the sequence of arguments, and posting the following user event back to whoever posted the initial event:

```
olleh([a,b,c],12345,`world`)
```

While this particular example is trivial, it serves to illustrate that user events are embodied within tuples, or compound terms, whose interpretation is entirely unrestricted and up to the individual event handler. The sum total of events handled and generated comprises an application.

agent_stream(?Term)

Term	<non-variable> or <variable>

Comments

If *Term* is an unbound variable, read a non-variable term from the current input stream and bind it to the variable; if *Term* is not an unbound variable, write it to the current output stream.

Examples

The following command demonstrates the writing and reading of a term, "hello(world)", using string input and output as the transport medium:

```
| ?- agent_stream( hello(world) ) ~> String,
     agent_stream( Term ) <~ String.                    <enter>
String = `ÿ□~\~L~E~@hello~X~L~E~@world~T` ,
Term = hello(world)
```

Notes

Term streaming is the process by which terms (events) are rendered into text in a form suitable for transmitting across a network, and are reconstructed from this text at the receiving end of a connection. By default, Chimera uses *agent_stream/1* to perform this function; this predicate is not intended to be called directly, although it could conceivably have some uses in encoding a ***WIN-PROLOG*** term as text, or vice versa.

In order to allow communications with other agent systems, Chimera allows the user to define their own term streamer. Provided such a user-defined streamer can take any non-variable Prolog term, and represent it in some form of written text, or read such text and recreate a copy of the original term, it can be used in place of *agent_stream/1* by calling *agent_stream/2* to associate its name with a given agent. Such a user-defined streamer must be of arity 1, and must read when called with an unbound variable, and write when called with anything else. Additionally, it must fail, and not generate an error, whenever it encounters end of file on input.

agent_stream(+Name, ?Pred)

Name	<atom>
Pred	<atom>, <empty list> or <variable>

Comments

Get or set the name of the arity 1 term streamer predicate, *Pred*, for the given agent, *Name*. If *Pred* is an unbound variable, the name of the current term streamer is returned as an atom and bound it to the variable; if *Pred* is an atom, then an arity 1 predicate of that name is used to perform streaming on the given agent; if *Pred* is an empty list, this resets to the default, agent_stream, as the name of the term streamer for the given agent.

Examples

The following command creates an agent, "foo", using the predicate, "bar/3", as its event handler, at TCP/IP port "123":

```
| ?- agent_create( foo, bar, 123 ).                    <enter>
yes
```

Assuming this command was successful, the following command returns the name of the term streamer used by this agent:

```
| ?- agent_stream( foo, Pred ).                        <enter>
Pred = agent_stream
```

Supposing that a user predicate, bar/1, has been written which can perform term streaming, it can be assigned to the agent with the command:

```
| ?- agent_stream( foo, bar ).                         <enter>
yes
```

Notes

Term streaming is the process by which terms (events) are rendered into text in a

form suitable for transmitting across a network, and are reconstructed from this text at the receiving end of a connection. By default, Chimera uses *agent_stream/1* to perform this function; this predicate is not intended to be called directly, although it could conceivably have some uses in encoding a ***WIN-PROLOG*** as text, or vice versa.

In order to allow communications with other agent systems, Chimera allows the user to define their own term streamer. Provided such a user-defined streamer can take any non-variable Prolog term, and represent it in some form of written text, or read such text and recreate a copy of the original term, it can be used in place of *agent_stream/1* by calling *agent_stream/2* to associate its name with a given agent. Such a user-defined streamer must be of arity 1, and must read when called with an unbound variable, and write when called with anything else. Additionally, it must fail, and not generate an error, whenever it encounters end of file on input.

agent_version(+Mode)

Mode	<integer> in the domain {0,1}

Comments

Write the Chimera welcome banner to the current output stream, showing the version and date, using the given *Mode* to determine whether or not to print "tramlines" around the banner, using values of "1" (one) and "0" (zero) respectively.

Examples

The following command creates an agent, "foo", using the predicate, "bar/3", as its event handler, at TCP/IP port "123":

```
| ?- agent_version( 1 ).                                    <enter>
-------------------------------------------------
Chimera 1.000 - Agents for WIN-PROLOG - 20 Jul 2005
Copyright (c) 2005 Logic Programming Associates Ltd
-------------------------------------------------
yes
```

Notes

The primary purpose of this predicate is to confirm which version of Chimera is running on a given system. The banner, and *Mode* values, is designed to be compatible in layout and style with the ***WIN-PROLOG*** welcome banner:

```
-------------------------------------------------
LPA WIN-PROLOG 4.600 - S/N 0013406400 - 28 Jun 2005
Copyright (c) 2005 Logic Programming Associates Ltd
Licensed To: LPA Development and Documentation Team
B=64 L=64 R=64 H=256 T=2048 P=8192 S=64 I=256 O=256
-------------------------------------------------
```

If desired, the above banner can be generated using the *ver/1* predicate, which is called with the same *Mode* values as *agent_version/1* to control tramlines.

Appendix B - Chimera Event Reference

This appendix is arranged alphabetically, and contains formal definitions for the system events created by Chimera and general information about user events, complete with comments, command line and handler examples of their use and notes about points of interest; it is designed as a reference section: please see *Chapter 4* for a tutorial overview of events and event handling.

A Simple Output Utility

To avoid repetition in the descriptions that follow, it should be assumed that the following, simple output program has been defined:

```
bar_show( Name, List ) :-
   (  nl,
      write( `Agent ` ),
      write( Name ),
      write( `: ` ),
      forall( member( Item, List ),
              write( Item )
            ),
      nl
   ) ~> user.
```

This program can be used to output an agent's name and list of data parameters to the console window, for example with the command:

```
| ?- bar_show( foo, [one,` `,two,` `,3] ).                <enter>

Agent foo one two 3
yes
```

Each of the event descriptions will include one or more examples of a handler that calls bar_show/1 to output information to the console window.

(close,Port)

Port	<integer>

Comments

The agent receiving this event has been closed by a call to agent_close/1; *Port* contains the TCP/IP port that was being listened to by the agent prior to closing.

Examples

Consider the following event handler:

```
bar( Name, [], (close,Port) ) :-
   bar_show( Name,
               [`Closed agent `,Name,` on port `,Port]
             ).
```

The following command creates an agent, "foo", using the predicate, "bar/3", as its event handler, at TCP/IP port "123":

```
| ?- agent_create( foo, bar, 123 ).                    <enter>
yes
```

After the agent has served its purpose, it can be closed with the command:

```
| ?- agent_close( foo ).                               <enter>
yes
```

This will generate the *(close,Port)* event, causing the following output:

```
Agent foo: Closed agent foo on port 123
```

Notes

The *(Close,Port)* system event is posted after closing an agent with agent_close/1; the related system event, *(Close,Host,Port)*, is posted after closing an agent link with

agent_close/2. Note that both these events are posted after closure, and after all other events to the agent or agent link have been purged: receipt of these events confirms that there are no pending events scheduled for the agent or agent link respectively.

Note that *(Close,Port)* is only ever posted directly to an agent that has just been closed; in other words, it is self-posted: this means that the event handler's second argument, which contains the link identifier, will always be the empty list, ("[]"). Meanwhile, also note that *(Close,Host,Port)* is never posted directly to an agent, but always to the agent link that has just been closed, so the event handler's second argument will always be an integer.

(close,Host,Port)

Host	<string>
Port	<integer>

Comments

The agent link receiving this event has been closed by a call to *agent_close/2*; *Host* contains the machine name, domain name or IP address and *Port* contains the TCP/IP port that was connected by the link prior to closing.

Examples

Consider the following event handler:

```
bar( Name, Link, (close,Host,Port) ) :-
   bar_show( Name,
                [`Closed link `,Link,
                 ` to host `,Host,` on port `,Port]
              ).
```

The following command creates an agent, "foo", using the predicate, "bar/3", as its event handler, at TCP/IP port "123":

```
| ?- agent_create( foo, bar, 123 ).                    <enter>
yes
```

Assuming this command was successful, the following command creates a connection between this and an agent running on a machine at IP address "192.168.1.2" and at port "456":

```
| ?- agent_create( foo, Link, `192.168.1.2`, 456 ).    <enter>
Link = 0
```

After the agent link has served its purpose, it can be closed with the command:

```
| ?- agent_close( foo, 0 ).                            <enter>
```

```
yes
```

This will generate the *(close,Host,Port)* event, causing the following output:

```
Agent foo: Closed link 0 to host 192.168.1.2 on port 456
```

Notes

The *(Close,Port)* system event is posted after closing an agent with agent_close/1; the related system event, *(Close,Host,Port)*, is posted after closing an agent link with *agent_close/2*. Note that both these events are posted after closure, and after all other events to the agent or agent link have been purged: receipt of these events confirms that there are no pending events scheduled for the agent or agent link respectively.

Note that *(Close,Port)* is only ever posted directly to an agent that has just been closed; in other words, it is self-posted: this means that the event handler's second argument, which contains the link identifier, will always be the empty list, ("[]"). Meanwhile, also note that *(Close,Host,Port)* is never posted directly to an agent, but always to the agent link that has just been closed, so the event handler's second argument will always be an integer.

(create,Port)

Port <integer>

Comments

An agent has been created by a call to agent_create/1; *Port* contains the TCP/IP port that has been assigned to the agent.

Examples

Consider the following event handler:

```
bar( Name, [], (create,Port) ) :-
   bar_show( Name,
               [`Created agent `,Name,` on port `,Port]
             ).
```

The following command creates an agent, "foo", using the predicate, "bar/3", as its event handler, at TCP/IP port "123":

```
| ?- agent_create( foo, bar, 123 ).                    <enter>
yes
```

This will generate the *(create,Port)* event, causing the following output:

```
Agent foo: Created agent foo on port 123
```

Notes

The *(create,Port)* system event is posted after an agent has been created with *agent_create/1*; the related system event, *(create,Host,Port)*, is posted after creating an agent link with *agent_create/2*.

Note that *(create,Port)* is only ever posted directly to an agent that has just been

created; in other words, it is self-posted: this means that the event handler's second argument, which contains the link identifier, will always be the empty list, ("[]"). Meanwhile, also note that *(create,Host,Port)* is never posted directly to an agent, but always to the agent link that has just been created, so the event handler's second argument will always be an integer.

(create,Host,Port)

Host	<string>
Port	<integer>

Comments

An agent link has been created by a call to *agent_create/2*; *Host* contains the machine name, domain name or IP address and *Port* contains the TCP/IP port that are connected to by the link.

Examples

Consider the following event handler:

```
bar( Name, Link, (create,Host,Port) ) :-
   bar_show( Name,
               [`Created link `,Link,
                ` to host `,Host,` on port `,Port]
             ).
```

The following command creates an agent, "foo", using the predicate, "bar/3", as its event handler, at TCP/IP port "123":

```
| ?- agent_create( foo, bar, 123 ).                    <enter>
yes
```

Assuming this command was successful, the following command creates a connection between this and an agent running on a machine at IP address "192.168.1.2" and at port "456":

```
| ?- agent_create( foo, Link, `192.168.1.2`, 456 ).    <enter>
Link = 0
```

This will generate the *(create,Host,Port)* event, causing the following output:

```
Agent foo Created link 0 to host 192.168.1.2 on port 456
```

Notes

The *(create,Port)* system event is posted after an agent has been created with *agent_create/1*; the related system event, *(create,Host,Port)*, is posted after creating an agent link with *agent_create/2*.

Note that *(create,Port)* is only ever posted directly to an agent that has just been created; in other words, it is self-posted: this means that the event handler's second argument, which contains the link identifier, will always be the empty list, ("[]"). Meanwhile, also note that *(create,Host,Port)* is never posted directly to an agent, but always to the agent link that has just been created, so the event handler's second argument will always be an integer.

The *(create,Host,Port)* system event is related to the system event, *(open,Host,Port)*: both occur after an agent link has been established. The difference between them is that the latter is posted after an incoming connection has been automatically accepted, while the former is posted after proactively creating an agent link with *agent_create/2*.

Both the *(create,Host,Port)* and *(open,Host,Port)* system events can be used, if desired, to initiate a user-defined protocol between connecting agents, perhaps to exchange agent names or other registration data.

(error,Code,What)

Code	<integer>
What	<atom>

Comments

An unexpected error has occurred on the given agent link; this link is about to be closed automatically. The *Code* will either contain a Winsock error number, generally in the range [10000..10093], or a *WIN-PROLOG* error number. The second argument, *What*, will contain an atom indicating the error type:

What	Meaning
sck_accept	it was not possible to accept an incoming connection link
sck_close	an agent link has been closed abruptly
sck_connect	it was not possible to create an outgoing connection link
sck_read	an error occurred while receving data across the network or while parsing it in the streamer
sck_write	an error occurred while sending data across the network

Examples

Consider the following event handler:

```
bar( Name, Link, (error,Code,What) ) :-
   bar_show( Name,
                [`Error `,Code,` in event `,What,
                 ` on link `,Link]
             ).
```

The following command creates an agent, "foo", using the predicate, "bar/3", as its event handler, at TCP/IP port "123":

```
| ?- agent_create( foo, bar, 123 ).                    <enter>
yes
```

Assuming this command was successful, the following command attempts to create a connection between this and an agent running on a machine at IP address "192.168.1.2" and at port "456":

```
| ?- agent_create( foo, Link, `192.168.1.2`, 456 ).    <enter>
Link = 0
```

Although this command has returned successfully, if the remote host/port pair does not exist or otherwise refuses a connection, it will generate the *(error,Code,What)* event, causing the following output:

```
Agent foo: Error 10061 in event sck_connect on link 0
```

In this case, the error numbe corresponds to the Winsock error, WSACONNREFUSED, which simply means the connection was refused.

Notes

The *(error,Code,What)* system event is something of a catch-all: it reports any Winsock TCP/IP communications errors as they happen, as well as syntax and other errors in the term streamer, whether this be *agent_stream/1* or a user-defined streamer that has been assigned to an agent with the help of *agent_stream/2*.

Whenever a Winsock error is flagged by this event, the offending agent link is automatically closed, resulting in the purging of remaining queued events, a subsequent posting of the *(close,Host,Port)* system event; when streaming errors occur, the link is left open.

(open,Host,Port)

Host	<string>
Port	<integer>

Comments

An agent link has been accepted from an incoming connection request; *Host* contains the machine name, domain name or IP address and *Port* contains the TCP/IP port that are connected to by the link.

Examples

Consider the following event handler:

```
bar( Name, Link, (open,Host,Port) ) :-
   bar_show( Name,
                [`Opened link `,Link,
                 ` from host `,Host,` on port `,Port]
             ).
```

The following command creates an agent, "foo", using the predicate, "bar/3", as its event handler, at TCP/IP port "123":

```
| ?- agent_create( foo, bar, 123 ).                    <enter>
yes
```

Assuming this command was successful, suppose an agent running on a machine at IP address "192.168.1.2" requests to make an incoming connection; this will generate the *(open,Host,Port)* event, causing the following output:

```
Agent foo: Opened link 0 from host 192.168.1.2 on port 1025
```

Notes

The *(open,Host,Port)* system event is related to the system event, *(create,Host,Port)*: both occur after an agent link has been established. The difference between them

is that the former is posted after an incoming connection has been automatically accepted, while the latter is posted after proactively creating an agent link with *agent_create/2*.

Both the *(create,Host,Port)* and *(open,Host,Port)* system events can be used, if desired, to initiate a user-defined protocol between connecting agents, perhaps to exchange agent names or other registration data.

(read,Host,Port,Term)

Host	<string>
Port	<integer>
Term	<tuple>

Comments

An incoming user event, posted by the agent at the remote end of the link, has been read successfully; *Host* contains the machine name, domain name or IP address and *Port* contains the TCP/IP port that are connected to by the link, while *Term* contains the tuple that represents the user event.

Examples

Consider the following event handler:

```
bar( Name, Link, (read,Host,Port,Term) ) :-
   bar_show( Name,
                [`Read on link `,Link,
                 ` from host `,Host,` on port `,Port,
                 `~M~J                          `,term = Term]
             ).

yes
```

The following command creates an agent, "foo", using the predicate, "bar/3", as its event handler, at TCP/IP port "123":

```
| ?- agent_create( foo, bar, 123 ).                    <enter>
yes
```

Assuming this command was successful, the following command creates a connection between this and an agent running on a machine at IP address "192.168.1.2" and at port "456":

```
| ?- agent_create( foo, Link, `192.168.1.2`, 456 ).    <enter>
```

```
Link = 0
```

Suppose the agent at the other end of the link posts the user event, "hello(world)"; this will generate the *(read,Host,Port,Term)* event, causing the following output:

```
Agent foo: Read on link 0 from host 192.168.1.2 on port 456
                          term = hello(world)
```

Notes

The *(write,Host,Port,Term)* and *(read,Host,Port,Term)* system events are associated with calls to *agent_post/3*, when this predicate is used to post user events to an agent link. The source agent will be issued the former event, and the target agent will receive the latter, immediately prior to receiving the user event itself, as embodied within *Term*. Events that are self-posted, by calling *agent_post/3* with an empty list ("[]") as its second argument, do not generate *(write,Host,Port,Term)* and *(read,Host,Port,Term)* system events.

(write,Host,Port,Term)

Host	<string>
Port	<integer>
Term	<tuple>

Comments

An outgoing user event, posted to the agent at the remote end of the link, has been written successfully; *Host* contains the machine name, domain name or IP address and *Port* contains the TCP/IP port that are connected to by the link, while *Term* contains the tuple that represents the user event.

Examples

Consider the following event handler:

```
bar( Name, Link, (write,Host,Port,Term) ) :-
   bar_show( Name,
               [`Written on link `,Link,
                ` to host `,Host,` on port `,Port,
                `~M~J                          `,term = Term]
             ).
```

The following command creates an agent, "foo", using the predicate, "bar/3", as its event handler, at TCP/IP port "123":

```
| ?- agent_create( foo, bar, 123 ).                    <enter>
yes
```

Assuming this command was successful, the following command creates a connection between this and an agent running on a machine at IP address "192.168.1.2" and at port "456":

```
| ?- agent_create( foo, Link, `192.168.1.2`, 456 ).    <enter>
Link = 0
```

Using the link number returned from this call, the following command will post the user event, "hello(world)", to whichever agent is running on the remote machine:

```
| ?- agent_post( foo, 0, hello(world) ).                    <enter>
yes
```

This will generate the *(write,Host,Port,Term)* event, causing the following output:

```
Agent foo: Written on link 0 to host 192.168.1.2 on port 456
                          term = hello(world)
```

Notes

The *(write,Host,Port,Term)* and *(read,Host,Port,Term)* system events are associated with calls to *agent_post/3*, when this predicate is used to post user events to an agent link. The source agent will be issued the former event, and the target agent will receive the latter, immediately prior to receiving the user event itself, as embodied within *Term*. Events that are self-posted, by calling *agent_post/3* with an empty list ("[]") as its second argument, do not generate *(write,Host,Port,Term)* and *(read,Host,Port,Term)* system events.

User(|Data)

User	<term>
Data	<term>

Comments

A placeholder for any arbitrary user event, where *User* is the event name, normally expressed as an atom, and *Data* is its associated information, normally represented as a list of arguments.

Examples

Consider the following event handler:

```
bar( Name, Link, Head(|Tail) ) :-
   bar_show( Name,
                [`Event `,Head,` with data `,Tail,
                 ` scheduled on link `,Link]
              ).
```

The following command creates an agent, "foo", using the predicate, "bar/3", as its event handler, at TCP/IP port "123":

```
| ?- agent_create( foo, bar, 123 ).                    <enter>
yes
```

Assuming this command was successful, the following command creates a connection between this and an agent running on a machine at IP address "192.168.1.2" and at port "456":

```
| ?- agent_create( foo, Link, `192.168.1.2`, 456 ).    <enter>
Link = 0
```

Suppose the agent at the other end of the link posts the user event, "hello(world)"; this will generate the user event, "*hello(world)*", causing the following output:

```
Agent foo: Event hello with data [world] scheduled on link 0
```

Notes

All system events in Chimera are in the form of *conjunctions*, represented by the most general form:

```
(name,data)
```

where "name" is one of a small set of pre-defined atoms, and "data" is a simple Prolog term or conjunctions of such terms. In general, all system events can safely be ignored by applications unless, as in the examples presented in this appendix, they are intercepted for informational purposes. Chimera can function fully without any special processing of its system events.

Unlike system events, user events are critical to the operation of an application. A user event is simply a ***WIN-PROLOG*** *tuple*, or *compound term*, which has the following most general form:

```
functor(|arguments)
```

where "functor" may be any Prolog term, although usually an atom, and "arguments" is any Prolog term, although usually a list of zero or more Prolog terms; for example:

```
hello(`world`,12345,[a,b,c])
```

consists of the functor, "hello", and a list of three arguments, the string, "`world`", the integer, "12345", and a list containing the three atoms, "a", "b" and "c".

The meaning of user events is left entirely to the receiving agent's event handler: it is effectively the sum total of events handled and generated that comprises an application.

Index

Symbols

A

F

G

H

I

K

L

T

U

V

W