LogiMOO: an Extensible Multi-User Virtual World with Natural Language Control *

PAUL TARAU, KOEN DE BOSSCHERE,† VERONICA DAHL, STEPHEN ROCHEFORT‡

▷ LogiMOO is a BinProlog-based Virtual World running under Netscape and Internet Explorer for distributed group-work over the INTERNET and user-crafted virtual places, virtual objects and agents. LogiMOO is implemented on top of a multi-threaded blackboard-based logic programming system (BinProlog) featuring Linda-style coordination. Remote and local blackboards support transparent distribution of data and processing over TCP/IP links, while threads ensure high-performance local client-server dynamics. Embedding in Netscape provides advanced VRML and HTML frame-based navigation and multi-media support, while LogiMOO handles virtual presence and acts as a very high-level multimedia object broker. User-friendliness is achieved through a controlled English interface written in terms of Assumption Grammars. Its language coverage is extensible in that the user can incorporate new nouns, verbs and adjectives as needed by changes in the world. Immediate evaluation of world knowledge by the parser yields representations which minimize the unknowns allowing us to deal with advanced Natural Language constructs like anaphora and relativization efficiently. We take advantage of the simplicity of our controlled language to provide as well an easy adaptation to other natural languages than English, with English-like representations as a universal interlingua.

Keywords: Virtual Worlds, high-level HTML,VRML,CGI programming distributed object brokerage, blackboard-based logic programming, Linda coordination, linear/intuitionistic assumptions, client-server applications in Prolog, embedded logic engines, natural language analysis, control through speech, multilingual interface ◁

*The first and the the third authors thank for support from NSERC (grants OGP0107411 and 611024), the second author is research associate with the Fund for Scientific Research – Flanders (Belgium).

Address correspondence to Paul Tarau, Département d'Informatique, Université de Moncton, CANADA E1A-3E9 E-mail: *tarau@info.umoncton.ca*.

†Vakgroep Elektronica en Informatiesystemen, Universiteit Gent, E-mail: *kdb@elis.rug.ac.be*.

‡Logic and Functional Programming Group, Department of Computing Sciences, Simon Fraser University, E-mail: {*veronica,srochefo*}*@cs.sfu.ca*.

655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$3.50

1. Introduction

MUDs and MOOs (Multi User Domains - Object Oriented) first introduced virtual presence and interaction in the context of networked games [23, 25]. Traditional MOOs use places called *rooms* and chat facilities to put in touch users represented by *avatars* for entertainment or information exchange purposes. The architecture is usually client/server, with users connecting to the server either through conventional telnet sessions or through more special purpose MOO shells.

Their direct descendents, Virtual Worlds, provide a strong unifying metaphor for various forms of net-walk, net-chat, and Internet-based virtual presence in general. They start where usual HTML shows its limitations: they do have state and require some form of virtual presence. *Being there* is the first step towards full virtualization of concrete ontologies, from entertainment and games to schools and businesses.

Some fairly large-scale projects (Intel's Moondo, Sony's Cyber Passage, Black Sun's CyberGate, Worlds Inc.'s WorldChat, Microsoft's VChat) converge towards a common interaction metaphor: an avatar represents each participant in a multi-user virtual world. Information exchange reuses our basic intuitions, with almost instant *learnability* for free.

The sophistication of their interaction metaphor, with VRML landscapes and realistic *avatars* moving in shared multi-user virtual spaces, will soon require high-level agent programming tools, once the initial fascination with looking human is not enough, and the automization of complex behavior becomes the next step. Towards this end, high-level coordination and deductive reasoning abilities are among the most important additions to various virtual world modeling languages.

Presently, despite their graphical sophistication, virtual worlds do not allow *controlling* behavior and object creation i.e., *programming with words*. Yet their characteristics favor the use of natural language: each virtual world represents a particular domain of interest, so that its associated relevant subset of language is naturally restricted; and the command language into which natural language sentences would have to be parsed is formal and straightforward enough while being already relatively close to natural language.

Our Virtual World implementation, LogiMOO is based on a set of embeddable logic programming components which inter-operate with standard Web tools.

Section 2 describes LogiMOO, a virtual world with natural language capabilities that makes use of linear and intuitionistic assumption techniques. Section 2.1 describes the coordination aspect; section 2.2, the LogiMOO kernel; section 2.3 describes our agent model; section 2.4 presents LogiMOO as a Netscape application; section 2.5 proposes a web of MOOs allowing LogiMOO access to Netscape based users with no access to BinProlog. Section 3 describes the main features of LogiMOO's natural language interface: coverage, dynamic knowledge handling, multisentential anaphoric reference, extensibility within the same language and to other languages, immediate evaluation, and our positional treatment of nouns. The last three sections respectively discuss related work, future work, and our conclusions.

2. The architecture of LogiMOO

LogiMOO [14, 38, 32] is a BinProlog-based Virtual World running under Web browsers for distributed group-work over the Internet and user-crafted virtual places, virtual objects and agents.

The main layers of the LogiMOO architecture are:

- *the underlying BinProlog system* which also provides client/server and CGI programming
- *LogiMOO's builtin operations* providing a set of MOO-like operations implemented as compiled Prolog predicates
- *the Natural Language compiler* which translates sentences to built-in LogiMOO predicates
- *the extensible user interface layer*, providing dynamic creation of server-side persistent objects, as result of Natural Language or built-in LogiMOO commands

A CGI-based BinProlog script *remote top-level* interacts with a remote LogiMOO server (Fig. 1). Objects in LogiMOO are represented as hyper-links (URLs) towards their owners' home pages where their native representation actually resides in various formats (HTML, VRML, GIF, JPEG, etc.). Embedding in Netscape allows advanced VRML or HTML frame-based navigation and multi-media support, while LogiMOO handles virtual presence and acts as a very high-level universal object broker.

The LogiMOO kernel behaves as any other MOO while offering a choice between interactive Prolog syntax and a Controlled English parser allowing people unfamiliar with Prolog to get along with the basic activities of the MOO: place and object creation, moving from one place to the other, giving/selling virtual objects, talking (*whisper* and *say*).

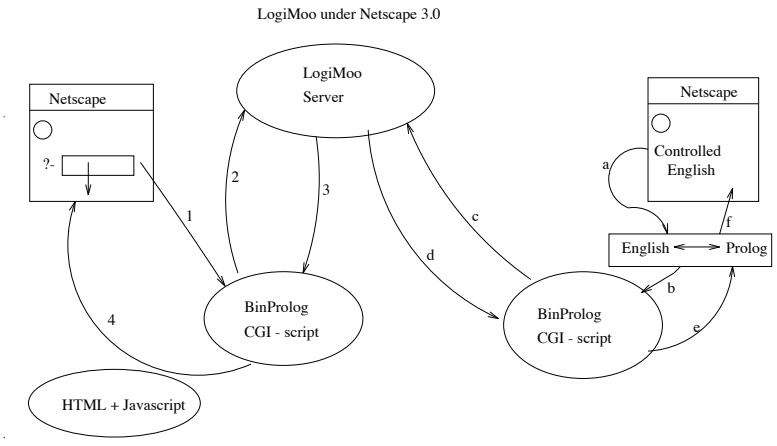

Figure 1. LogiMOO on the Web

In order to achieve this, we have incorporated natural language consultation capabilities into LogiMOO (see section 3). Our natural language front-end is extensible in the sense

that it is easy for the user to tailor it to a particular virtual world through defining the verbs specific to that world and establishing links between lexical objects and their WWW representations. These definitions are done in user-friendly terms, with the system being responsible for their integration into the rest of the grammar and for their correspondence with actual LogiMOO objects and actions.

2.1. Linda style coordination

LogiMOO is built upon a portable socket-level Re-implementation of Multi-BinProlog [11, 12, 13], i.e., BinProlog enriched with a Linda-like tuple space. Solaris 2.x threads ensure high-performance local client-server dynamics.

A blackboard's basic characteristics are (i) it is persistent, (ii) the data are manipulated associatively (i.e., based on their content, rather than on their address), (iii) all accesses are automatically synchronized. These are precisely the requirements for a MOO: the state of the MOO should be stored somewhere, and it should not be volatile; if we are looking for a particular object in a MOO, we will always refer to it with a name, never with its address location; a MOO is multi-user, so it should be synchronized at any time.

The basic operations on the blackboard are: blackboard creation and deletion, the creation of processes (independently running Prolog goals), putting Prolog terms on the blackboard, removing terms from the blackboard and checking the presence of terms. Both the get and the read primitives can be blocking or non-blocking. Furthermore, there are some more advanced primitives like blackboard operations working on lists of terms instead of one term, operations to collect the complete contents of a blackboard, and so on.

LogiMOO's primitive operations are implemented on top of Multi-BinProlog's Linda-style operations [33]. We refer to [14, 38] for a description of the wide variety of *blocking* and *non-blocking* as well as *non-deterministic* blackboard operations (backtracking through alternative answers). For reasons of embeddability in multi-paradigm environments and semantic simplicity we have decided to drop non-determinism and return to a subset close to the original Linda [7, 8, 9] operators (combined with unification), and a simple client-server architecture (although LogiMOO's design is now rapidly evolving towards a web of interconnected worlds). This turned out to be enough for simple (one-thread) agent programming.

`out(X)`	Puts X on the server
`in(X)`	Waits until it can take an object matching X from the server
`all(X,Xs)`	Reads the list Xs matching X currently on the server
`run(Goal)`	Starts a thread executing Goal
`local_out(X)`	Puts private information X on the local default blackboard.
`local_rd(X)`	Checks whether an object matching X is on the local blackboard.
`halt`	Stops current thread

The presence of the `all/2` collector compensates for the lack of non-deterministic operations. Note that the only blocking operation is `in/1`, and that that blocking `rd/1` is easily emulated in terms of `in/1` and `out/1`. Non-blocking `rd/1` is emulated with `all/2` (see also Figure 2).

A number of derived operations are built on top of the primitive LogiMOO operations.

`rd(X)`	Checks whether an object matching X is on the server's blackboard.
`cout(X)`	Conditional out: Puts X on the server unless an object matching X is found on the server.

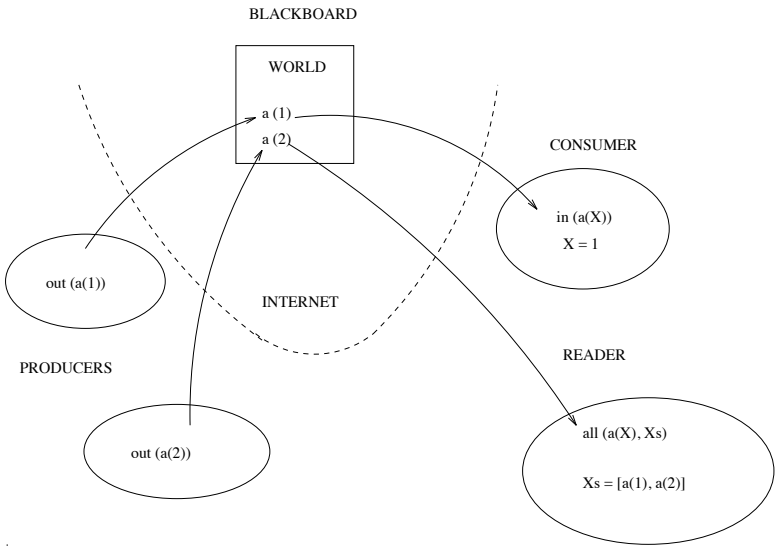

Figure 2. Basic Linda operations

`cin(X)`	Conditional in: Takes an object matching X from the server and fails if no such object is found.
`forall(X,G)`	Executes goal G for all objects on the server matching X.

2.2. The LogiMOO kernel

Verbs available in LogiMOO are defined through a set of Prolog predicates hiding the complexities of the distributed communication model through the usual metaphors: places (starting from a default lobby), ports, ability to *move* or *teleport* from one place to another, a *wizard* resident on the server, *ownership* of objects, the ability to *transfer* ownership and a built-in notifier agent watching for messages as a background thread.

Non-shared information is kept on the default local blackboard. The login procedure simply puts the name of the current user on the local blackboard, after enforcing unique identity on the server by sending a password to the server together with the name.

`Whereami(P)` unifies `P` with the location of the user's avatar.

Locally the name chosen by the user is accessible as:

```
whoami(X):-local_rd(i_am(X)).
```

The use of local blackboards (on which `local_rd` operates) allows high-speed access to private state information and allows for the implementation of security protocols.

To conditionally create a place (unless it exists) we use: `dig(Place)`. To create a port (only when the links are already existing places), we use: `port(P1,Dir,P2)`. To teleport `O` from `P1` to `P2`, we use: `move(O,P1,P2)`.

```
move(O,P1,P2):-cin(contains(P1,O)),out(contains(P2,O)).
```

On top of teleporting, we implement `go(Dir)`, `craft(O)` (which creates virtual objects with ownership), and `give(Who, What)`.

```
go(Dir):-
  whereami(Place),
  rd(port(Place,Dir,NewPlace)),
  whoami(Me),
  move(Me,Place,NewPlace),
  forall(has(Me,O),move(O,Place,NewPlace)).
```

`go/1` verifies accessibility of the target place through a port and updates the avatar's location.

Note that `forall/2` can be used to make someone's belongings follow him. As in the real world, this is usually done selectively on a subset of a user's belongings.

Creating things with `craft/1` marks them with ownership:

```
craft(O):-whoami(Me),
  rd(contains(Place,Me)),
  out(contains(Place,O)),
  out(has(Me,O)).
```

`Craft/1` gets the place where the user's avatar is located, then puts the object there and asserts ownership of the object by the user's avatar.

Property transfer (useful for online sales and banking) is prototyped as follows:

```
gives(From,To,O):-
  cin(has(From,O)),
  out(has(To,O)).

give(Who,What):-
   whoami(Me),
   cin(has(Me,What)),
   out(has(Who,What)).
```

`Give/2` simply changes ownership of the object by updating the `has/2` fact referring to it on the blackboard.

Although expressing a `sale` predicate in terms of `give/2` is easy, realistic transactions need a sound security system, not yet implemented in LogiMOO.

As usual, PGP-based public-key cryptography can ensure secure transactions on top of an insecure carrier. Although it is possible to implement electronic money and authentication within LogiMOO on top of standard Solaris tools like `des_crypt`, the embedding of LogiMOO in Netscape (described in section 2.4), allows use of high quality third-party encryption and E-cash technology. As technology matures, security becomes largely an orthogonal issue - especially when standard tools can be used to implement mechanisms that ensure interoperability with other secure software components.

`Look/0` recognizes specific objects and shows them in the most useful form. For instance, under the Netscape interface, users are shown as hyper-links to their home pages and objects created by a given user are shown as links relative to the user's home page.

Note that the unusual expressiveness of the blackboard for an important number of roles (messages, synchronization, etc.) shows that some of the traditional programming patterns are just implementation related intellectual artifacts. The existence of a unique construct covering them all (Linda+unification) helps towards building more programmer friendly, higher level abstractions.

2.3. The agent model

We define an *agent* as being a set of *behaviors*, as well as the usual code for logical inferences and arithmetic. Each behavior is usually attached to its own thread, although thread sharing mechanisms can be used on some machines for actual implementations. An agent *located* in a container is called an *avatar*.

Agents performing actions on objects can either *fail*, *succeed*, *wait* (to succeed) or *warn* about an *error* or *exception*.

The notifier is one of the simplest possible *agents*. It is automatically started from the login predicate as a background thread with `run(notifier(Name))`. The notifier's thread blocks until `in(mes_to(Name,Mes,From))` succeeds and the notifier simply outputs the message.

```
notifier(Name) :-
  in(mes_to(Name,Mes,From)),
  notify(Mes,From),
  notifier(Name).
```

The evaluation of `whisper/2`, defined as

```
whisper(To,Mes) :- whoami(Me),out(mes_to(To,Mes,Me)).
```

unblocks the matching `in(mes_to(Name,Mes,From))`, and consequently the notifier outputs the message with `notify(Mes,From)`. More generally, distributed event processing is implemented by creating an agent watching for a given pattern.

Remote processing [30] as well as security mechanisms are expressed modularly, by creating a *command server* thread:

```
% remote processing request
please(Who,What) :-
  whoami(Requester),
  out(please(Who,Requester,What)),
  whisper(Who,'Please'(What)).

% a remote command processor with intruder detection
command_server :-
  whoami(Me),
  repeat,
    in(please(Me,Requester,What)),
    ( friend_of(Me,Requester)->call(What),fail
    ; errmes(intruder(remote_action_attempted),Requester)
    ),
  fail.
```

Such security modules can be added, tailored, or left out, as needed, without requiring additional concepts.

Clearly, `command_server` and `notifier` threads can be seen as *behavior components* of a unique agent. Moreover, they actually might cooperate in a synchronized way as each `please/2` command triggers a `whisper/2` action to be served by a notifier later.

2.4. LogiMOO as a client/server Web application

Objects in LogiMOO[1] are represented as hyper-links (URLs) towards their owners' home pages where their native representation actually resides in various formats (HTML, VRML, GIF, JPEG, etc.).

LogiMOO redefines the Prolog toplevel for interacting with Multi-BinProlog either through telnet/rlogin[2] BinProlog's pipe-based lightweight Tcl/Tk interface, or through the Web with Netscape . The Web interface extensively uses advanced browser features:

- Netscape/Internet Explorer frames and forms
- Cosmo Player or WorldView plug-in for VRML navigation
- JavaScript to help BinProlog control Netscape frames from within
- BinProlog-based lightweight CGI-scripts

Netscape-based users are recognized as special as they do not keep local state (except for their name, password, and home page kept in the form itself). The stateless CGI Multi-BinProlog client spawn by a *submit* Netscape action connects to a local persistent LogiMOO server. Compilation to C and shared dynamically linked libraries make BinProlog competitive with Perl scripting.

The server periodically saves its state to a file, through a separate background thread while staying responsive to users. A telnet/rlogin-based console helps to monitor LogiMOO from a remote computer.

Figure 3 shows the embedding of LogiMOO as a frame-based Netscape application.

Queries are submitted through the CGI POST method. BinProlog reads the standard input using the `CONTENT_LENGTH` environment variable and after a small filter cleans up hexadecimal escapes, it extracts the actual query and its variables through a list-of-characters-to-term conversion. Finally, using a simple trick, we map HTML query syntax to Prolog without any application specific parsing:

```
:-op(1199,xfy,(&)).

(login=L & passwd=P & home=H & query=Query) :-
  login(L,P,H),
  metacall(Query).
```

Objects crafted by users are shown as URL's, relative to their homes. This allows users to *put* into LogiMOO objects of various formats (VRML, JPEG, WAV, AU) and gives multi-media capabilities for free. LogiMOO keeps the link while the actual object resides on the user's computer accessible by clicking on a link so that the user is free to update the actual object without having to notify LogiMOO. Note also that we do not need to provide navigation in VRML worlds or user defined HTML links - this is better to be left to the browser itself. What we provide is the ability to create those persistent links dynamically,

[1] Electronically available and remotely executable from URL `http://clement.info.umoncton.ca/~tarau/logimoo`. with a Netscape or compatible browser. We use Netscape throughout this section for exemplifying purposes, but our discussion extends to all browsers providing similar functionality.

[2] Users without their own browser can access LogiMOO through a Prolog shell hosted on our computers with a Unix-level guest account. Setting this up is quite easy by replacing /bin/sh in /etc/passwd with a Multi-BinProlog C-ified executable /opt/bin/mbp, customized to support LogiMOO.

Figure 3. LogiMOO as a frame-based Netscape application

as the result of a controlled natural language interaction with the user or her decision to trigger the action of a building agent.

We can however give the illusion that BinProlog commands from within LogiMOO actually allow arbitrary Web navigation through use of a one line JavaScript, dynamically generated as an answer to a query:

```
auto_show(URL,File) :-
  make_cmd(['<body
     onLoad="window.open(''',URL,'/',File,''','_self')">'],Cmd),
  writeln([Cmd]).
```

For instance, typing:

```
auto_show('http://eve.info.umoncton.ca:8080/~logimoo','lobby.wrl')
```

in the LogiMOO Prolog query text area, will instantly show LogiMOO's VRML lobby in the Netscape output frame, from where the user can further explore links independently. Return to LogiMOO is achieved simply by clicking on the VRML floor of the room. By using the `_parent` Netscape pseudo-target instead of `_self` the full window is replaced. With `_blank` or a named new target, an additional Netscape browser is spawned, allowing independent navigation, while keeping LogiMOO on-screen.

Clearly, achieving exclusively in Prolog or any other existing LP or CLP the equivalent of what we have developed in about 1-man/month total programming time would require a significant effort. We are more and more convinced that embedding logic programming tools in a multi-paradigm environment can compensate for their lack of advanced visual and Internet programming abilities and, ultimately, make them competitive for commercial development despite their small market share.

As an embedded application, LogiMOO acts as a broker between various multi-paradigm, multi-media Netscape components. It therefore keeps (a minimum amount of) state and user information. Its full Prolog command language gives arbitrary extensibility through objects and agents. Although file transfers and various protocols are implementable with the underlying Multi-BinProlog system, we have chosen to represent non-symbolic objects as hyper-links towards their owner's WWW home. Our design philosophy was to duplicate as little existing components as possible while achieving as much functionality as possible. At some point, we expect that LogiMOO will grow by itself through user extensions, much more than our own development effort, as a truly open virtual world, together with its present and future VRML and Java-centric cousins.

2.5. Beyond Linda: a Web of MOOs

Clearly, representing places with separate blackboards and avatars/objects with processes/terms on local and remote blackboards is the most natural representation, especially when thinking in terms of a distributed *Web of MOOs*. We will describe the basic ideas behind LogiMOO in terms of a simpler mapping reminiscent from client/server architectures, which also allows users not having BinProlog on their own computers to access LogiMOO. The *world* is represented by a blackboard on a server. Connectionless Netscape *hits* from the user's computer create short-lived CGI clients on the computer hosting the server[3].

[3]This has been found very useful for teaching applications with low-end PCs.

Multi-BinProlog blackboards come in two flavors [12]: there are local blackboards, primarily used for efficient communication and synchronization between processes running on the same (multi-)processor, and there are virtual blackboards that are primarily used for communication with processes running on other machines. A virtual blackboard is a kind of alias, or link to another (remote) blackboard. Logically, it cannot be distinguished from the remote blackboard itself. At the implementation level, a virtual blackboard does not contain data, but is just a local representation of the remote blackboard. All operations issued on a virtual blackboard are automatically forwarded to the remote blackboard. Furthermore, the remote blackboard can in turn be a virtual blackboard pointing to yet another remote blackboard. The last blackboard in a chain must always be a physical blackboard containing real data (see Figure 4).

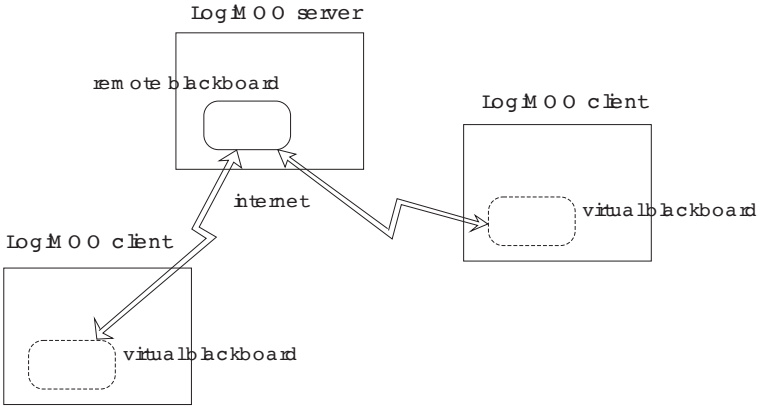

Figure 4. Virtual and remote blackboards

A pure Linda-based language is not immediately suitable to support a MOO because a Linda-application consists of a number of processes that are created by one application in order to solve a given problem. Here, client processes must be able to connect and disconnect at any time. This requirement can be completely fulfilled by the concept of virtual blackboard. A client wanting to connect with a MOO, creates a local virtual blackboard, hooks it up to the MOO, and from then on, it can interact with the MOO in an unconstrained way by communicating with the local virtual blackboard.

The first version of LogiMOO has been built upon Multi-BinProlog [11, 13], i.e., BinProlog enriched with a Linda-like tuple space and virtual blackboards. The following version (which also interconnects with a new Java based multi-threaded Linda server) uses BinProlog's new built-in Linda subsystem.

To ensure interoperability between Windows 95 PCs and Unix machines we have implemented a generic socket package with operations specialized towards support for Linda operations, remote execution and mobile code [35]. A master server on a *well-known* host/port is used to exchange identification information among peers composed of a client and a multiplexing server. We refer to [36] for more details on the underlying BinProlog based Internet infrastructure used in this re-implementation of LogiMOO.

3. The natural language based user interface

There is a very strong move towards the use of natural language as a command language today, with General Magic, Microsoft, IBM and telecommunication companies pioneering its use for major industrial applications which routinely use spoken language to communicate with the user both for input and output.

We expect that as the domain of intelligent software agents matures, the current emphasis on the interaction will be balanced towards more sophisticated reasoning abilities, with LP in a very good position to provide them.

One of the key design ideas behind LogiMOO was that natural language has a serious potential as an effective programming language, at least as far as end user interaction (scripting) is concerned. Our views are confirmed by programs like the recently released Microsoft Agent [26] or General Magic's upcoming Portico [17], a voice-only intelligent assistant able to learn and remember the state its interaction with the user.

LogiMOO is one of the very few existing virtual worlds that can be controlled with natural language. The reasons why we wanted to provide LogiMOO with a NL interface are:

1. natural language is the most convenient way for us to communicate;
2. a natural language interface is the first step towards voice-controlled interaction with the virtual world[4];
3. a virtual world is a perfect environment to experiment with natural language because the domain of discourse is limited;
4. natural language is needed for the upcoming speech recognition/generation based human/computer interaction tools

The peculiar features of the world to be consulted- a virtual world- induced novel parsing features which are interesting in themselves: flexible handling of dynamic knowledge, immediate evaluation of noun phrase representations, allowing us to be economic with representation itself, inference of some basic syntactic categories from the context, a treatment of nouns as proper nouns, easy extensibility within the same language as well as into other natural languages. We shall examine each of these features in turn, after describing our natural language coverage.

It is interesting that the use of these features results in a completely deterministic parser (no backtrack).

3.1. Language coverage

Since LogiMOO handles mostly commands, outermost sentences will be imperative sentences. In these, the subject (the avatar that the user is controlling) is usually left implicit (notice however that embedded sentences, such as relative clauses, are descriptive rather than imperative, and therefore do include a subject). The user first enters a set of imperative subjectless sentences, and this input is sent through the parser to be converted into

[4] With operating systems as OS/2 Warp integrating basic voice recognition and products like Dragon Systems' continuous unrestricted speech recognizer widely available, controlling software as well as programming in a speech-friendly environment becomes increasingly realistic.

LogiMOO kernel predicates which are then executed to complete the actions. Verbs in the LogiMOO environment represent actions that can take place in the virtual world.

Because they are in imperative form, with their subject left implicit, LogiMOO sentences reduce to verb phrases, which can be of the following forms:

An intransitive verb.
A transitive verb followed by a noun.
A transitive verb followed by a noun phrase.
A transitive verb followed by a prepositional phrase.
A bitransitive verb followed by two noun phrases.
A bitransitive verb followed by a noun phrase and a prepositional phrase.

A prepositional phrase is defined as

A preposition followed by a noun phrase.

The noun phrase forms allowed are

A proper name.
A pronoun (anaphora).
A determiner followed by a noun.

In addition, we identify communication inputs which occur when a user wants his/her avatar to say, whisper or yell some message, e.g.,

say hi how are you.

This form of input is introduced by either:

The word whisper *followed by a prepositional phrase followed by a message.*
The word say *followed by a message.*
The word yell *followed by a message.*

Table 1 shows some sample parses.

NL Input	**Translation**	**LogiMOO action**
look.	`look.`	Provides a description of the room the users avatar currently occupies.
craft a car.	`craft(car).`	Creates a virtual object, car, owned by the avatar.
craft a car. give it to john.	`craft(car),` `give(john, car).`	Creates a virtual object, car, and gives it to john.
take the car that john crafted.	`take(car).`	Puts a car object crafted by john into the avatar's possession.

Table 1. Sample Parses

Notice that the last command in the table produces the name (i.e., *car*) which designates the object referred to by the noun phrase. This name is obtained by consulting the world

knowledge to get the name of the car that John crafted. Ambiguity regarding which car is meant among several in the world is avoided because the same name invoked by a different user, or by the same user in a different room, has a different representation internally. If, however, the user wants to craft more than one car in the same room, each should be differently designated in the command list (e.g., craft car1, give it to wizard, craft car2, give it to Stephen).

3.2. Handling Dynamic Knowledge

Because virtual worlds are eminently constructive, their dynamic changes must be accommodated in a flexible yet discriminatory manner. For instance, we must distinguish between *static knowledge*, i.e. world knowledge that exists before a user's sequence of world-changing commands, and *dynamic knowledge*, i.e. the new knowledge that results from those commands, since these may in some cases be tentative and subject to revision.

Static knowledge is obtained previous to the parsing of a sequence of natural language commands, through a small Prolog program which stores the current state of the world in predicates such as `is_avatar(X), is_crafted(X)`, etc.

Dynamic knowledge is created by execution of a natural language command, and described with the aid of the same predicates as static knowledge, but these are put on a blackboard instead of simply extending the static world knowledge. Once the complete series of a user's commands has been executed, with later commands possibly having revised the results of previous commands in the same interaction, the resulting blackboard information is made available to be used in the next iteration, for gathering the state of the world before the next sequence of natural language commands.

3.3. Multisentential Anaphoric Reference

Our system maintains yet another type of knowledge- *hypothetical knowledge*, accessible only by the parser, which helps it decide what parts of speech should be related. In particular, the knowledge that noun phrases are potential referents of an anaphora is kept through linear assumptions which are consumed upon encountering, for instance, a pronoun which might refer to the noun phrase hypothesized as a referent. For example, while analyzing the sequence of commands: "Craft a flower, give it to John", the parser hypothesizes that "flower" might turn out to be the referent of some pronoun appearing somewhere in the rest of the discourse, and upon encountering "it", the right object (namely, "flower") is associated with this pronoun. Although not shown above, gender and number information is also useful to check compatibility between a potential referent and an anaphora. A more complete analysis of anaphoric resolution through Assumption Grammars can be found in [10].

3.4. Language Extensibility

Within the same language Our goal of language extensibility comes from the need to dynamically introduce new concepts into the world, and with them, new vocabulary in the analyzer. For instance, "craft a gnu" must be accepted even if no gnus exist in the virtual world yet, and no corresponding entry exists in the lexicon.

For this reason, our parser recognizes a noun from its context in the sentence rather than from any lexical definition. Adjectives can be treated similarly, by requiring them to be

used in controlled fashion, e.g. within relative clauses, as in "a car that is red", so that the parser can infer adjectival function unequivocally from the word's position as an attribute. Note that since the world is described in terms of physical metaphors, adjectives will refer to such properties as color, shape, position, etc., and statements about them will in general be conditionless clauses (facts).

Allowing verbs to be inferred from context is more difficult. The syntactic definition part of defining new verbs can be done by example, i.e. by gleaning from the user information re. number of arguments from similarity with other proposed sample verbs, on which the user would have to just click (e.g., "smile" as a sample intransitive verb, "look" as a sample transitive verb, "give" as a sample bitransitive verb, etc.). This allows user-friendliness by not requiring the user to handle syntactic notions such as "transitive" or "intransitive", but instead leaving it to our interface to invisibly replicate a similar lexical definition from the analogy with existing sample words. But the predicates obtained from verbs as a result of parsing cannot simply translate into a constant (as for most nouns) or a unary predicate (as for most adjectives), since in general, they must translate into n-ary predicates corresponding to actions, and the verb being a new one, these actions can in general require description through full Prolog clauses. Our present solution is to require the user to provide a Prolog definition of the new command that the new verb refers to. Future work will investigate higher level solutions to this problem.

Of course, we can think of extending a grammar with other types of words than nouns, verbs and adjectives. However, we have chosen to focus on just these categories because they are the most likely to be application-dependent, and because in the case of verbs, they are the ones that will induce corresponding new LogiMOO commands.

Extensibility to other natural languages Given that we accept only controlled language, and that some words, such as nouns and adjectives, do not need to be explicitly defined in a lexicon, but are inferred by the system from their first use in a command, we have a simple way of adapting our English analyzer into other languages.

In order to parameterize the language, we record which language we are using in the call, by means of an intuitionistic implication, e.g.:

```
?- language(spanish)=>parse([susurra,al,brujo]).
```

Lexical items will then be specialized according to the language, and will still induce an English-based semantic representation.

In order to explain how our rules do this, let us first observe that there are two types of rules in the English grammar which are language dependent:

- rules that create a predicate name which is reminiscent of the noun, verb or adjective from which they are derived,
- rules containing a lexical item (i.e., a symbol preceded by `'#'`).

For rules of the first type, we shall maintain the English predicate name regardless of the language of origin, as a kind of interlingua allowing us to go from one language to another.

For rules of the second type, we replace `'#native_word'` by `'@english_word'`), as in:

```
verb(give(X,Y)):- @give.
```

and then we define an English and alternative lexicons as follows:

```
%English lexicon:          % Spanish lexicon:

@give:- #give, -english.    @give:-  #de, -spanish.
```

Notice that the language we're at is checked *after* the corresponding word is found. This is to ensure speed, since in this way, the word to be parsed will be recognized right away. In generation mode, we might want to switch the order around.

Of course, more realistically, we will need features such as gender and number in order to produce the right words in each language. For instance, whereas in English we have only one lexical form for the definite article, whether it is singular, plural, feminine or masculine, in Spanish we have four different lexical items covering all these forms.

3.5. Immediate Evaluation

Commands get processed, as we saw, after their translation into formulas from their natural language expression. These formulas are conjunctions of actions expressed as predicates, whose arguments are constants representing objects, people, places, etc. These constants are produced by evaluating noun phrases on the fly during the parse. In other words, instead of generating a formula to represent a noun phrase, its components will be evaluated right away to generate a constant satisfying the noun phrase's description. For instance, if the command sequence is that of the two last sentences shown in table 1, parsing of "the car that john crafted" will directly produce "car" (the constant representing the entity that john crafted) rather than a descriptive formula such as "`crafted(john,X), is_a(X,car)`".

3.6. Treatment of nouns

Because our system translates a noun into a constant with same name, our treatment of nouns can be viewed as similar to that of proper names. As mentioned earlier, internally a distinction will be made between cars crafted by different users or by the same user in different rooms, but when the same user wants to refer to two distinct noun referents in the same room, a different noun name must be used for each, e.g. "car1" and "car2". Thus ambiguity is dealt with automatically without needing to resort to the explicit construction of internal unique identifiers for each object as is the case in many other systems. This makes object referencing very direct and allows us to proceed to the immediate evaluation of noun phrases described in the previous section. The resulting formulas are therefore simpler.

Adjectives can also be recognized by context, as explained earlier, but will generate a predicate (e.g., red(X) from the adjective "red") rather than a constant. This predicate, as we have seen, will evaluate immediately rather than being inserted into the formula being constructed.

4. Related work

Recent logic-based web applications have been presented at [15, 37, 39].

A survey of logic programming approaches to web applications in terms of the usual classification into client-based systems, server-side systems, and peer-to-peer systems has been provided in [22]. Client-side systems [22, 6, 4] offer more sophisticated user interfaces than server-side ones, and avoid networking programs that can affect server-side applications. They include HTML extensions to incorporate Prolog code, support libraries for web applications, Java integration with Prolog, logic-based web querying languages (Weblog [19]) W-ACE [27]). Some server-side systems use libraries which enable Prolog programs to process information from CGI input and generate suitable replies.Others use sockets for communication between the invoked CGI interface scripts and the task process or a higher-level communication layer based on active modules (PiLLoW/CIAO [6]). The main problems include the possibility of network load failures and of server overload, how to support multiple queries on a shared resource, and how to deal with lengthy browser interactions. A logic-programming related difficulty is how to deal with backtracking to a previous stage in the user interaction. Still other server-side systems completely replace the traditional web server by software which combines the functionality of a server with the particular task (e.g. the ECLiPse HTTP server library [4]).

Peer-to-peer systems (e.g. [24] LogicWeb [22]) use other abstractions (message passing or blackboards) but retain the Internet as their underlying communication layer. This allows them to implement multi-agent systems, where all participants must communicate on equal terms, bypassing the intrinsic inequality of the client/server model. Our present work fits within this category.

We are not aware of other systems using logic programming for virtual world simulation, although a large number of sophisticated Web-based applications and tools have been implemented in LP/CLP languages, for instance [5, 21, 6, 4, 31, 20]. The closest application with a clear virtual world flavor is the Ubique Doors (tm) server [28] which shows (Flat Concurrent) Prolog lists in log files although we do not know exactly how closely it is based on LP technology. This server, combined with the Sesame (tm) client emulates co-presence and cooperative work at virtual places implemented on top of existing Web pages and ftp directories. On the other hand applications of MOO technology usually combined with VRML navigation are spreading quite fast. Among them, some of the most impressive are:

- Sony's Cyber Passage, `http://vs.sony.co.jp/VS-E/vstop.html`,
- Black Sun's CyberGate, `http://www2.blacksun.com/beta/c-gate`,
- Netscape's CoolTalk+Live3D/CosmoPlayer, `http://home.netscape.com`,
- Worlds Inc.'s WorldChat, `http://www.worlds.net`,
- Intel's Moondo, `http://www.intel.com/iaweb/moondo`,
- SenseMedia's The Sprawl, `http://sensemedia.net/sprawl`.

Moreover, other declarative languages are starting to be used for WWW applications. The Carnegie Mellon FoxNET project [16] implements a full featured Web server. Microsoft's Active VRML proposal promises a declarative (purely functional) description of 3D movement and behavior.

We have not yet found any MUD/MOO environments that handle NL processing. Other MUD/MOO environments fall into two general categories. Environments such as *Moondo* by Intel [18], *CyberGate* by BlackSun [3], and *Cyber Passage* by Sony [29], fall into the

category of point and click graphical environments. These completely avoid the need for NL processing as the only text involved seems to be that for chatting with other avatars. All movement and actions are completed with mouse point and click actions. The second category, in which environments such as MediaMOO [2] and the Avalon MUD [1] fall into, are text-based systems. These systems lack NL processing and focus on the use of pattern matching techniques to gather information.

Further, there is little work being done that is specific to the advantages gained by connecting MUDs/MOOs to the World Wide Web using logic programming such as the use of Prolog. As such, objects in LogiMOO are represented as hyper-links (URLs) towards their owners' home pages where their native representation actually resides in various formats (HTML, VRML, GIF, JPEG, etc.). At the same time, logic programming adds deductive database facilities in a uniform framework, hypothetical reasoning tools (through Assumption Grammars), and logic programming data and code use the same representation which makes meta-programming easy.

Virtual Worlds technologies pioneered by [3, 18, 29, 40] are becoming part of standard setting applications like Netscape Communicator or Internet Explorer. Most of them concentrate on the interaction metaphor and/or visualization without a principled approach to the underlying coordination logic.

Compared to other currently known MUD/MOO environments, this interface bridges the gap between those that are graphical based and those that are pattern matching based. By filling the gap, we are able to provide the users with a natural form of textual interaction on which graphical environments can still be built.

Although the current interaction is controlled completely through the natural language interface, this does restrict efficiency of maneuvering an avatar through the virtual worlds.

Multi-user blackboard systems have been described in [8, 12]. Among the original features of LogiMOO's multi-user blackboards are multi-threading and interoperation with Web protocols (httpd). BinProlog is not the only Prolog featuring blackboard processing. Commercial systems like SICStus Prolog also contain Linda subsystems. This makes the LogiMOO architecture fairly portable.

5. Future work

A Java based implementation, using a minimal set of logic programming components (unification, associative search) is on its way. It will be integrated in the existing LogiMOO framework on the server side. It holds promise for smooth cooperation with existing Java class hierarchies as well as various BinProlog based LogiMOO components.

This reimplementation of LogiMOO uses Jinni [34], a new, lightweight, pure logic programming language, intended to be used as a flexible scripting tool for gluing together knowledge processing components and Java objects in networked client/server applications and thin client environments. By supporting multiple threads, control mobility and inference processing, Jinni is well suited for quick prototyping of intelligent mobile agent programs.

It allows bidirectional communication with the existing LogiMOO framework, allowing creation of combined Java/Prolog mobile-agent programs. In particular, Java applets can be used as front-end in browsers instead of the more resource consuming CGIs LogiMOO is currently based on. It holds promise for smooth cooperation with existing Java class hierarchies as well as various BinProlog based LogiMOO components.

Intelligence and flexible metaprogramming on the logic programming side combined with visualization and WWW programming abilities on the Java side will allow easy component integration in various concrete containers.

The LogiMOO system is currently being used in teaching to introduce basic concepts of distributed programming and as a testbed for LogiMOO based virtual communities. Future directions are to include LogiMOO as a tool for virtual tele-education where distance education students and instructors may use LogiMOO as a teaching/learning environment.

With respect to the natural language processing component, the next logical step is the use speech recognition in order to interact with LogiMOO and other interoperable components running under Netscape as, for instance, VRML plugins. As we proceed with Jinni's interfacing with Microsoft Agent, LogiMOO will benefit from spoken input/output via the underlying implementation layer.

There is presently a growing interest in enhancing the web's role as a universal repository of information by adding computational content to it. A common example of *active pages* have form based submission mechanisms (the user invokes programs on remote hosts by submitting information via a form document).

The web itself is evolving into a stateful new model consisting of a set of connected MOOs. Under this model, our present methodologies for Prolog-based natural language interaction within a LogiMOO world can be extended for controlling the web itself through natural language.

6. Conclusion

We have shown that Prolog with appropriate coordination language extensions is a practical tool for virtual world simulation. A synergy between MOOs, Linda-style coordination and Prolog's powerful associative search and dynamic object creation facilities could be expected. Multi-BinProlog's threads and virtual blackboards make this synergy possible. A logic programming approach to MOO programming has the advantage of having all the right tools within a unified environment. Embedding in Netscape ensures implicit platform independence of our server and seamless cooperation with present and future third party Netscape tools.

We have also presented a natural language interface to LogiMOO which takes a controlled form of English and translates it into LogiMOO kernel predicates which are executed as actions in the virtual environment. Pronominal references in multisentential input are allowed. Extensibility within the same language is achieved by inferring new nouns and adjectives from their context in the sentences, and by a dialogue with the user that allows new verbs and their corresponding LogiMOO actions to be described in a user-friendly way. Extensibility into different natural languages is obtained not through the usual machine translation approach, but by abstracting a core set of language independent rules from our English parser and then adding a language specific lexicon (currently available for English, French, Spanish) to complete the grammar definition. A simple change of one lexicon module into another effects the language change invisibly, so that users across the world can type in their interactions in their own language, these are recorded in a neutral but invisible form, from which any retrieval continues to respect the language of the caller.

Acknowledgments

We thank the anonymous referees for their suggestions and constructive criticism. Special thanks go to Daniel Perron for long discussions that helped to come up with the initial idea

of LogiMOO and useful comments during the development of the project.

REFERENCES

1. The Avalon MUD. http://www.avalon-rpg.com/.
2. MediaMOO. telnet://guest@purple-crayon.media.mit.edu:8888/.
3. BlackSun. CyberGate. http://www.blaxxsun.com/.
4. Ph. Bonnet, L. Bressnan S., Leth, and B. Thomsen. Towards ECLIPSE Agents on the Internet. In Tarau et al. [37]. http://clement.info.umoncton.ca/ lpnet.
5. D. Cabeza and M. Hermenegildo. html.pl: A HTML Package for (C)LP systems. Technical report, 1996. Available from http://www.clip.dia.fi.upm.es.
6. D. Cabeza and M. Hermenegildo. The Pillow/CIAO Library for Internet/WWW Programming using Computational Logic Systems. In Tarau et al. [37]. http://clement.info.umoncton.ca/ lpnet.
7. N. Carriero and D. Gelernter. Linda in context. *CACM*, 32(4):444–458, 1989.
8. P. Ciancarini. Coordinating Rule-Based Software Processes with ESP. *ACM Transactions on Software Engineering and Methodology*, 2(3):203–227, 1993.
9. P. Ciancarini. Distributed Programming with Logic Tuple Spaces. *New Generation Computing*, 12(3):251–284, May 1994.
10. Veronica Dahl, Paul Tarau, and Renwei Li. Assumption Grammars for Processing Natural Language. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 256–270, MIT press, 1997.
11. K. De Bosschere and P. Tarau. Blackboard Communication in Logic Programming. In *Proceedings of the PARCO'93 Conference*, pages 257–264, Grenoble, France, September 1993.
12. K. De Bosschere and P. Tarau. Blackboard-based Extensions in Prolog. *Software — Practice and Experience*, 26(1):49–69, January 1996.
13. Koen De Bosschere and J.-M. Jacquet. Multi-Prolog: Definition, Operational Semantics and Implementation. In *Proceedings of the ICLP'93 conference*, Budapest, Hungary, 1993.
14. Koen De Bosschere, Daniel Perron, and Paul Tarau. LogiMOO: Prolog Technology for Virtual Worlds. In *Proceedings of PAP'96*, pages 51–64, London, April 1996.
15. A. Eliens, editor. *Proceedings of the Workshop on logic programming and the Web*, Santa Clara, California, April 1997.
16. Robert Harper and Peter Lee. Advanced Languages for Systems Software. Technical report, 1994. CMU-CS-FOX-94-01.
17. General Magic Inc. Portico. 1998. http://www.genmagic.com/portico/portico.html.
18. Intel. Moondo. http://www.intel.com/iaweb/moondo/index.htm.
19. Lakshmanan, L. V. S. and Sadri, F. and Subramanian, I.N. A Declarative Language for Querying and Restructuring the WWW. In *Proc. of the Post-ICDE IEEEWorkshop on Research Issues in Data Engineering*, feb 1996.

20. S. W. Locke, A. Davison, and Sterling L. Lightweight Deductive Databases for the World-Wide Web. In Tarau et al. [37]. http://clement.info.umoncton.ca/ lpnet.

21. S. W. Loke and A. Davison. Logic programming with the world-wide web. In *Proceedings of the 7th ACM Conference on Hypertext*, pages 235–245. ACM Press, 1996.

22. Seng Wai Loke. *Adding Logic Programming Behaviour to the World Wide Web*. Phd thesis, University of Melbourne, Australia, 1998.

23. J. Ludewig. Problems in Modeling the Software Development Process as an Adventure Game. In H. Rombach, V. Basili, and R. Selby, editors, *Int. Workshop on Experimental Sw Engineering Issues*, volume 706, pages 23–26, Dagsthul, Germany, Sept 1992. Springer.

24. McCabe, F.G. and Clark, K.L. April- Agent Process Interaction Language. In *Intelligent Agents, (LNAI 890)*. Springer-Verlag, 1995.

25. T. Meyer, D. Blair, and S. Hader. WAXweb: a MOO-based collaborative hypermedia system for WWW. *Computer Networks and ISDN Systems*, 28(1/2):77–84, 1995.

26. MicrosoftCorp. Microsoft Agent. 1998. http://www.microsoft.com/msagent/agentdl.asp/.

27. Pontelli, E. and Gupta, G. W-ACE: A Logic Language for Intelligent Internet Programming. In *Proc. of IEEE 9th ICTAI'97*, pages 2–10, 1997.

28. Ehud Shapiro. Enhancing the WWW with Co-Presence. In *Proceedings of the 2nd International Conference on the WWW*, 1994.

29. Sony. Cyber Passage. http://vs.sony.co.jp/VS-E/vstop.html.

30. J. W. Stamos and D. K. Gifford. Remote Evaluation. *ACM Transaction on Programming Languages and Systems*, 12(4):537–565, October 1990.

31. Peter Szeredi, Katalin Molnár, and Rob Scott. Serving Multiple HTML Clients from a Prolog Application. In Tarau et al. [37]. http://clement.info.umoncton.ca/ lpnet.

32. Paul Tarau. Logic Programming and Virtual Worlds. In *Proceedings of INAP96*, Tokyo, November 1996. keynote address.

33. Paul Tarau. BinProlog 5.75 User Guide. Technical Report 97-1, Département d'Informatique, Université de Moncton, April 1997. Available from *http://clement.info.umoncton.ca/BinProlog*.

34. Paul Tarau. Jinni: a Lightweight Java-based Logic Engine for Internet Programming. In Kostis Sagonas, editor, *Proceedings of JICSLP'98 Implementation of LP languages Workshop*, Manchester, U.K., June 1998. invited talk.

35. Paul Tarau and Veronica Dahl. Mobile Threads through First Order Continuations. In *Proceedings of APPAI-GULP-PRODE'98*, Coruna, Spain, July 1998.

36. Paul Tarau, Veronica Dahl, and Koen De Bosschere. A Logic Programming Infrastructure for Remote Execution, Mobile Code and Agents. In *Proceedings of WETICE'97*, pages 106–112, IEEE Computer Society Press, June 1997.

37. Paul Tarau, Andrew Davison, Koen De Bosschere, and Manuel Hermenegildo, editors. *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP'96, Bonn, September 1996. http://clement.info.umoncton.ca/ lpnet.

38. Paul Tarau and Koen De Bosschere. Virtual World Brokerage with BinProlog and Netscape. In Tarau et al. [37]. http://clement.info.umoncton.ca/ lpnet.

39. Paul Tarau, Koen De Bosschere, and Manuel Hermenegildo, editors. *Proceedings of the 2nd International Workshop on Logic Programming Tools for INTERNET Applications*, ICLP'97, Leuven, July 1997. http://clement.info.umoncton.ca/ lpnet.

40. Worlds. AlphaWorld. http://www.worlds.net/products/alphaworld.

Appendix

CGIs by example

CGI (common Gate Interface) allows HTTP servers to call an external program and display its output as a Web page. BinProlog has some built-in facilities which make CGI-programming easy. The following shows a simple Web page access counter.

```
main:-header,inc(X),show_counter(X).

header:-
  write('220 ok'),nl,
  write('content-type: text/html'),nl,nl.

show_counter(X):-write(counter(X)),write('.'),nl.

inc(X):-F='/tmp/counter_state.pro',
  ( see_or_fail(F)-> see(F),read(counter(X)),seen
  ; X=0
  ),X1 is X+1,
  tell(F),show_counter(X1),told.
```

To install a similar CGI script at your site, put the **bp** executable in directory **cgi-bin**, together with the program and call it from a HTML page as follows:

```
<A HREF=
  "/cgi-bin/bp?$/<MY ABSOLUTE PATH>/counter.pro">
  Click here!
</A>
```

You can try it out by clicking on it at:

```
http://clement.info.umoncton.ca/~tarau
```

Example of NL interaction in LogiMOO

Here is a trace based on the single-user version of LogiMOO we have used during the development of the NL interface. The single user version emulate multiple avatars through the command `I am <avatar>` allowing to impersonate multiple users. As the latest version of BinProlog blackboards scales transparently from single to multi-user mode, the same code is used for both single user and networked configurations. This is especially useful during debugging of non network related code like the NL parser.

```
TEST: I am Joe. Craft a cat. Where is the cat?
WORDS: [i,am,joe,.,craft,a,cat,.,where,is,the,cat,?]
SENTENCES: [i,am,joe] [craft,a,cat] [where,is,the,cat]
```

```
==BEGIN COMMAND RESULTS==
login as: joe with password: none
your home is at http://142.58.28.116/~guest

SUCCEEDING(iam(joe))
SUCCEEDING(craft(cat))
cat is in kitchen
SUCCEEDING(where(cat))

==END COMMAND RESULTS==

TEST: Craft a Gnu. Who has it? Where is it? Where am I?
WORDS: [craft,a,gnu,.,who,has,it,?,where,is,it,?,where,am,i,?]
SENTENCES: [craft,a,gnu] [who,has,it] [where,is,it] [where,am,i]

==BEGIN COMMAND RESULTS==
SUCCEEDING(craft(gnu))
joe has gnu
SUCCEEDING(who(has,gnu))
gnu is in kitchen
SUCCEEDING(where(gnu))
you are in the kitchen
SUCCEEDING(whereami)

==END COMMAND RESULTS==

TEST: Give to the Wizard the Gnu that I crafted. Who has it?
WORDS: [give,to,the,wizard,the,gnu,that,i,crafted,.,who,has,it,?]
SENTENCES:
  [give,to,the,wizard,the,gnu,that,i,crafted]
  [who,has,it]

==BEGIN COMMAND RESULTS==
logimoo:<joe># 'wizard:I give you gnu'
SUCCEEDING(give(wizard,gnu))
wizard has gnu
SUCCEEDING(who(has,gnu))
```