High Level Logic Programming Tools for Remote Execution, Mobile Code and Agents

Paul Tarau[1], Veronica Dahl[2] and Koen De Bosschere[3]

[1] Université de Moncton
Département d'Informatique
Moncton, N.B. Canada E1A 3E9,
tarau@info.umoncton.ca
[2] Logic and Functional Programming Group
Department of Computing Sciences
Simon Fraser University
Burnaby, B.C. Canada V5A 1S6
veronica@cs.sfu.ca
[3] Vakgroep Elektronica en Informatiesystemen,
Universiteit Gent
St.-Pietersnieuwstraat 41
B-9000 Gent, Belgium
kdb@elis.rug.ac.be

Abstract. We describe a set of programming patterns used for implementing, remote execution mechanisms, mobile code and agents in a distributed logic programming framework. The particular focus of this paper is on the use of BinProlog's strong metaprogramming abilities. Some advanced logic programming constructs as intuitionistic implication, high-order call/N cooperate with encapsulated socket-level constructs for maximum configurability and efficiency. We show that strong metaprogramming is not a security threat if used through a set of *filtering interactors* which allow source level implementation of arbitrary security policies. Mobile code is implemented in a scalable way through a set of distributed client+server pairs interconnected through a master server acting only as an address exchange broker for peer-to-peer interactors. We have thoroughly tested our programming patterns and design principles through a realistic implementation in a widely used, freely available Prolog system (`http://clement.info.umoncton.ca/BinProlog`) as well as with its Java peers built on top of our unification enhanced Java based Linda implementation (`http://clement.info.umoncton.ca/LindaInteractor`).
Keywords: mobile code, remote execution, metaprogramming, agents, Linda coordination, blackboard-based logic programming, Prolog, distributed programming, virtual worlds, scalable and secure Internet programming, Prolog/Java embedding

1 Introduction

Although the Internet has been designed to survive nuclear war and its underlying packet switching technology is intrinsically peer-to-peer, fault-tolerant and scalable, successive higher level networking and programming layers have given away (often too easily) these abilties. Among the most annoying *and* at the same time the most pragmatically well thought decisions dominating the world of the Internet, at various programming and application development layers:

- classical client/server architectures have given up scalability for simplicity of programming and synchronization;
- programming languages/operating systems have given up powerful remote execution mechanisms for reasons ranging from security concerns to legacy single user/single process assumptions.

In this paper, we will show how some standard and some non-standard Logic Programming language tools will be used to elegantly get back the full potential of the Internet for building scalable, peer-to-peer, programmable multi-user communities. The virtual layer we will build is based on a small set of very-high level programming constructs making essential use of meta-programming, which is seen here as the ablity to view information either as code or as data, and efficiently switch between these views, on demand. In particular, we will show how powerful remote execution mechanisms, agents and mobile code can be expressed in this framework.

With important resources at their disposal, operating system/computer/network companies have found their, often complex and costly ways to get back some of the original power, hidden under various layers of software/hardware. Despite the existence of practical solutions to some of these problems, we hope that the unifying approach we propose in this paper will help clarify the logical structure of future peer-to-peer, scalable virtual communities, where the distinction between human and non-human agents becomes less and less obvious. We are also convinced that the power of the knowledge/processing component of logic programming will help the efforts towards intelligent network/mobile agent programming more effective.

We will describe our constructs as built on the top of the popular Linda coordination framework, enhanced with unification improved pattern matching, although our actual implementation is based on a more general *coordination logic*, allowing negotiation of removal of tuples as well as of suspension between user intents at *definition time*, when an object is stored on the (shared and synchronized) *blackboard* as well as at *reference time* when it is retrieved associatively[20].

Linda [5] based frameworks like Multi-BinProlog [8] offer a 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 operators (combined with unification), and simple client-server component melted together into a scalable peer-to-peer layer forming a 'web of interconnected worlds'.

```
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
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`. Notice that blocking `rd/1` is easily emulated in terms of `in/1` and `out/1`, while non-blocking `rd/1` is emulated with `all/2`.

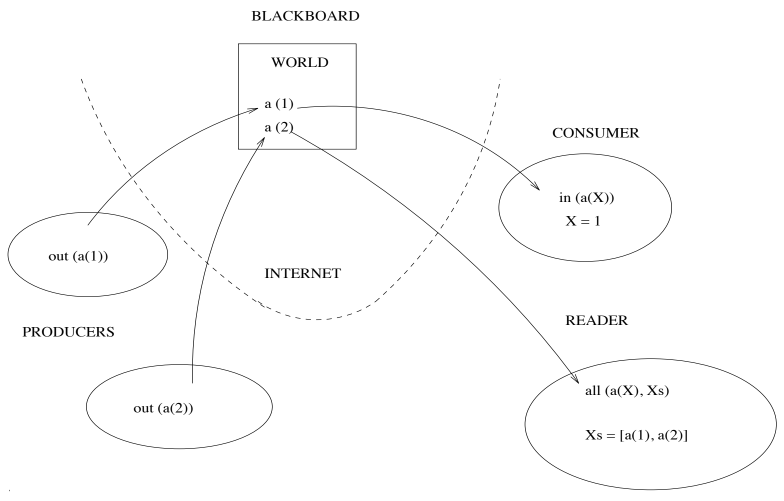

Fig. 1. Basic Linda operations

1.1 Simple Agents

A *servant* is one of the simplest possible *agents*:

```
servant(Who):-
  in(todo(Task)),
  secure(Task),
  run(Task),
  servant(Who).
```

The *servant* is started as a background thread with `run(servant(Name))`. No 'busy wait' is involved, as the servant's thread blocks until `in(todo(Task))` succeeds.

In an 'Internet chat' application, for instance, `whisper/2`, defined as

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

unblocks the matching `in(todo(mes_to(To,Mes,From)))` of a *notifier servant* which then outputs a message by `executing mes_to(To,Mes,From)`.

More generally, distributed event processing is implemented by creating a 'watching' agent for a given pattern.

Note the fundamental link between 'event-processing' and the more general Linda protocol: basically an `out/1` operation can be seen as generating an event and adding it to the event queue while an `in/1` operation can be seen as servicing an event. While usually event-loops switch on numeric or pseudo-numeric event constants in a rather rigid and un-compositional way, Linda-based event dispatching is extensible by adding new patterns. When (naturally) restricted to ground patterns, efficient indexing can be used with performance hits unnoticeable in a network lag context. Linda's `out/blocking in` combination can be seen as automating the complex if-then-else logic of (distributed) hierarchical message dispatching loops through unification.

Remote processing as well as simple security mechanisms are expressed easily, by creating 'command server' thread close to the following code:

```
% remote processing request
please(Who,What):-
  whoami(ForWho),
  out(please(Who,ForWho,What)).

% a remote command processor
command_server:-
  whoami(Me), % gets unique user identification information
  repeat,
    in(please(Me,ForWho,What)),
    ( friend_of(Me,ForWho)->call(What),fail % trusted friends
    ; errmes(intruder(remote_action_attempted),ForWho)
    ),
  fail.
```

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 if each `please/2` command would trigger a `whisper/2` action to be serviced by a notifier later. In this example security is based on allowing arbitrary execution for trusted (i.e. known to `friend_of/2`) clients of existing server-side code. Notice that by restricting `command_server` to execute only sequences of acceptable known commands, on top of code-as-data sent through `out/1` transactions, arbitrary security mechanisms can be put in place at source level. We will describe this later in more detail.

2 Towards a Web of Worlds

Our implementation is integrated in the freely available BinProlog compiler, starting with version 5.60. To ensure interoperability between Windows 95/NT

PCs and Unix machines we have conservatively implemented a generic socket package with operations specialized towards support for Linda operations and remote execution. 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 (Fig. 2)

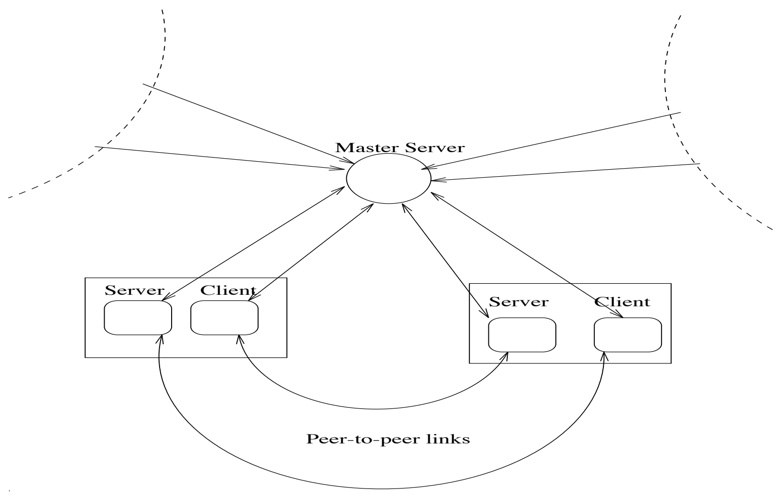

Fig. 2. A Web of Worlds

2.1 Server side code

The code for our 'generic' server, with various components which are overridden through use of intuitionistic implication to obtain customized special purpose servers at user level, follows. Higher order call/N [16], combined to intuitionistic assumptions are used to pass arbitrary *interactors* to this generic server code.

```
run_server(Port,Fuel):-
  new_server(Port,NewServer),
  register_server(Port),
    server(NewServer)=>>server_loop(NewServer,Fuel),
  close_socket(NewServer).

server_loop(Server,Fuel):-
  for(I,1,Fuel),
    interact(Server),
    assumed(server_done(Mes)),
  !.
```

```
new_service(Server,Service):-
  default_timeout(T),
  new_service(Server,T,Service).

interact(Server):-
  default_server_action(Interactor),
  ( call(Interactor,Server)->true % higher-order call to interactor
  ; \+ errmes(failed_interaction(Server),Interactor)
  ),
  !.
```

Note the use of a specialized *server-side* interpreter `server_loop`, configurable through the use of higher-order 'question/answer' closures we have called *interactors*.

We have found out that use of intuitionistic implications (pioneered by Miller's work [14,15,9]) helps to overcome (to some extent) Prolog's lack of object oriented programming facilities, by allowing us to 'inject' the right interactor into the generic (and therefore reusable) interpreter. BinProlog's `=>>` temporarily assumes a clause in 'asserta' order, i.e. at the beginning of the predicate. The assumption is scoped to be only usable to prove its right side goal and vanishes on backtracking. We refer to [19,21,7] for more information on assumptions and their applications.

2.2 Client side code

The following example shows how the same client-side code is used for both for point-to-point and 'broadcast' communication, depending on a `to_all(Pattern)` assumption allowing to select matching targets.

```
new_client(Connection):-
  default_host(Host),
  default_port(Port),
  new_client(Host,Port,Connection).

ask_server(Question,Answer):-
  assumed(to_all(ServerPattern)),!,
  all_servers(ServerPattern,Xs),
  ask_all_servers(Xs,Question),
  Answer=yes.
ask_server(Question,Answer):-
  ask_a_server(Question,Answer).
```

Slightly more complex code (see file extra.pl in the BinProlog distribution) also handles proxy forwarding services transparently.

3 Running remote code

On an Intranet of trusted users and computers, or in different windows of an unconnected PC or workstation a user might want to experiment with a server allowing arbitrary Prolog command execution. One can start a local remote predicate call server with:

```
?-run_unrestricted_server.
```

BinProlog's convention is that if the name returned by `default_host/1` is different from `localhost` it assumes that the user wants to get connected and interoperate with other BinProlog users.

To override this default setting one can use either

```
   set_host('my.full.internet.address')
```

or temporarily override the default host by assuming it as a `host/1` fact with BinProlog's intuitionistic assumption. Such assumptions are scoped and forgotten on backtracking. With

```
?-server_host('my.full.internet.address')=>>run_server.
```

a given server is registered on a master server given by (also overridable):

```
default_master_server(Host,Port)
```

An unrestricted server can run benign commands like

```
?-remote_run((write(hello),nl)).
```

as well as more malicious ones like:

```
?-remote_run(pcollect('cat /etc/passwd',Answer))
```

to collect password information ready for one's favorite cracking algorithm.

Note that registered servers are in principle accessible to other users and therefore *not* secure. We will give next two simple approaches to implement security.

3.1 A first approach to security: servers with restricted interactors

Here is the code of the chat server:

```
chat_server_interactor(mes(From,Cs),Answer):-show_mes(From,Cs,Answer).
chat_server_interactor(ping(T),R):-term_server_interactor(ping(T),R).

chat_server:-
  server_interactor(chat_server_interactor)=>>
  run_server.
```

By overriding the default `server_interactor` with our `chat_server_interactor` we tell to `chat_server` that only commands matching known, secure operations (i.e. `mes/2` and `ping/1`) have to be performed on behalf of remote users.

This might look very simple but some of BinProlog's key features (intuitionistic implication and higher-order call/N) are used inside the 'generic' server code (see file extra.pl in the BinProlog distribution) to achieve this form of configurability.

Security is achieved by having specialized meta-interpreters 'filtering' requests on the server side. Intuitionistic assumption is used to override the generic server's *interactor*, i.e. the inner server operation filtering allowed requests, while allowing reuse of most of the generic server's code in an object oriented programming style.

Here is some *interactor* code for two other 'secure' servers.

The first one (BinProlog's current default) is a Linda Term server (to be started with `run_server/0`) enhanced with moderately weak remote management commands. It offers a good combination of security and convenience: it only allows modifying its dynamic database remotely through Linda operations and displaying messages. It can be shut down remotely and checked if alive and of course it can be subject to resource attacks by malicious users. However it can do no harm to one's file system and cannot be used to collect information about one's computer.

```
term_server_interactor(cin(X),R):-serve_cin(X,R).
term_server_interactor(out(X),R):-serve_out(X,R).
term_server_interactor(cout(X),R):-serve_cout(X,R).
term_server_interactor(rd(X),R):-serve_rd(X,R).
term_server_interactor(all(X),Xs):-serve_facts(X,X,Xs).
term_server_interactor(all(X,G),Xs):-serve_collect(X,G,Xs).
term_server_interactor(id(R),R):-this_id(R).
term_server_interactor(ping(T),T):-ctime(T).
term_server_interactor(stop(Mes),yes):-assumel(server_done(Mes)).
term_server_interactor(halt(X),yes):-serve_halt(X).
term_server_interactor(mes(From,Cs),Answer):-
  show_mes(From,Cs,Answer).
term_server_interactor(proxy(H,P,Q),R):-
  ask_a_server(H,P,Q,A)->R=A;R=no.
term_server_interactor(in(X),R):-serve_cn(X,R).
```

Note that this interactor even supports proxy forwarding to a given host/port. Clearly, the secure execution of forwarded queries is the responsibility of the proxy target.

3.2 A second approach to security: starting an Intranet specific master server

Keeping one's host/port information secret from other users can be achieved by starting a master server local to a secure physical or virtual Intranet.

For users behind a firewall, this might actually be the only way to try out these operations as the default master server might be unreachable.

A local master server on, e.g. port 7788 is started with something like:

```
?- master_server('my.secure.local.computer',7788)=>run_master_server.
```

Client and server programs intended to be managed by a local master server will have simply to assume a fact:

```
   master_server('my.secure.local.computer',7788).
```

or set the master server with:

```
  set_master_server('my.secure.local.computer',7788)
```

To keep the workload of the master server minimal, only when an error is detected by a client, the master server is asked to refresh its information and possibly remove dead servers from its database.

3.3 Interaction with Java Applets.

BinProlog starting from version 5.40 communicates with our recently released Java based *Linda Interactors*[1] special purpose trimmed down pure Prolog engines written in Java which support the same unification based Linda protocol as BinProlog. The natural extension was to allow Java applets to participate to the rest of our 'peer-to-peer' network of BinProlog interactors. As creating a server component within a Java applet is impossible due to Java's (ultra)-conservative security policies we have simply written a receiving-end *servant* close to our example in subsection 1.1, which relies on a proxy server on the site where the applet originates from, for seamless integration in our world of peer-to-peer interactors.

Here is the code for a more realistic servant, multiplexed among multiple servers and usable inside a Java applet.

```
run_servant:-
  default_server_interactor(Interactor),
  this_id(ID),
  out(servant_id(ID)), % registers this servant
  repeat,
    in(query(ServerId,Query)), % waits for a query
    (call(Interactor,Query,Reply)->Answer=Reply;Answer=no),
    % sends back an answer, if needed
    ( ServerId=[]->true % no reply sent to anonymous servers
    ; out(answer(ServerId,Answer))
    ),
  functor(Query,stop,_),  % stops when required
  !,
  in(servant_id(ID)).
```

[1] available at http://clement.info.umoncton/BinProlog/LindaInteractor.tar.gz

Note the presence of an overridable client-side interactor, allowing the generic servant code to be easily reused/specialized. Multiplexing is achieved by having each server's in/1 and out/1 data marked with unique `server_id/1` records. The server side code for a single query is simply:

```
ask_servant(Query,Answer):-
   this_id(ID),
   % identifies the server the query comes from
   out(query(ID,Query)), % asks servant
   in(answer(ID,R)),     % gets back answer
   Answer=R.
```

To integrate multiple Java applet based clients in our 'Web of Worlds', we use a more complex forwarding server, also available as equivalent Java code, to be run as a BinProlog and/or Java daemon[2] on the same machine as the HTTP server the applet comes from.

```
run_forwarding_server:-
  server_interactor(forwarding_server_interactor)=>>
  run_server.

forwarding_server_interactor(mes(From,Cs),R):-!,R=yes,
   forward_to_servants(mes(From,Cs)).
forwarding_server_interactor(Q,A):-
   term_server_interactor(Q,A).

forward_to_servants(Query):-
   clause(servant_id(_),true),
   assert(query([],Query)),
   fail.
forward_to_servants(_).
```

Note that the forwarding server has the ability to interact with multiple servants, in particular with multiple Java applets.

3.4 Remote execution and mobile agents

The MOO[3] inspired 'Web of Worlds' metaphor [22,24] combined with the idea of 'negotiation' between the *agent's intentions* and the *'structure of the world'*, represented as a set of Linda blackboards storing state information on servers

[2] Alternatively, BinProlog can be embedded as a server side include in an Apache server (written in C) while the equivalent Java code is easily embeddable in the Java based Jigsaw HTTP server.

[3] Multi User Domains (MUDs), Object Oriented - venerable but still well doing ancestors of more recent multi-user Virtual Worlds, which are usually 3D-animation (VRML) based

connected over the the Internet allows a simple and secure remote execution mechanism through specialized *server-side* interpreters.

Implementation of arbitrary remote execution is easy in a Linda + Prolog system due to Prolog's metaprogramming abilities. No complex serialization or remote procedure/method call packages are needed. In BinProlog (starting with version 5.60) code fetched lazily over the network is cached in a local database and then dynamically recompiled on the fly if usage statistics indicate that it is not volatile and it is heavily used locally.

As an example, high performance file-transfer (actually known to be faster than well-known `http` as `ftp` protocols) is implemented by orchestrating Prolog based `remote_run` control operations which are used to trigger direct full speed transfers entirely left to optimized C built-ins like `sock2file/2`:

```
fget(RemoteFile,LocalFile):-
   remote_run(RF,fopen(RemoteFile,'rb',RF)),
   fopen(LocalFile,'wb',LF),
   term_chars(to_sock(RF),Cmd),
   new_client(Socket),
     sock_write(Socket,Cmd),
     socket(Socket)=>>from_sock(LF,_),
     fclose(LF),
     sock_read(Socket,_),
   close_socket(Socket),
   remote_run(fclose(RF)).

from_sock(F,yes):-
   assumed(socket(S)),
   sock2file(S,F),!.
```

Once the basic Linda protocol is in place, and Prolog terms are sent through sockets, a command filtering server loop simply listens and executes the set of 'allowed' commands.

Despite Prolog's lack of object oriented features we have implemented code reuse with intuitionistic assumptions [21,7].

For instance, to iterate over the set of servers forming the receiving end of our 'Web of Worlds', after retrieving the list from a 'master server' which constantly monitors them making sure that the list reflects login/logout information, we simply override `host/1` and `port/1` with intuitionistic implication:

```
ask_all_servers(ListOfServers,Question):-
  member(server_id(_,H,P),ListOfServers),
  host(H)=>>port(P)=>>ask_a_server(Question,_),
  fail.
ask_all_servers(_,_).
```

To specialize a generic server into either a master server or a secure 'chat-only' server which merges messages from BinProlog users world-wide we simply override the filtering step in the generic server's main interpreter loop.

Implementing agents 'roaming over' a set of servers is a simple and efficient high-level operation. First, we get the list of servers from the master server. Then we iterate through the appropriate remote-call negotiation with each site. Our agent's behavior is constrained through security and resource limitations of participating interpreters, having their own command filtering policies.

In particular, on a chat-only server, our roaming agent can only display a message. If the local interpreter allows gathering user information then our agent can collect it. If the local interpreter allows question/answering our agent will actually interact with the human user through the local server window.

Note that 'mobile agents' do not have to be implemented as code physically moved from one site to another. In this sense we can talk about *virtual mobile agents* which are actually sets of synchronized remote predicate calls originating from a unique site, where most of the code is based/executed, while code actually moved can be kept to strict minimum, i.e. only a few remotely asserted clauses[4].

Our mobile agents are seen as *'connection brokers'* between participating independent server/client sites. For instance if two sites are willing to have a private conversation or code exchange they can do so by simply using the server/port/password information our agent can help them to exchange.

Note that full metaprogramming combined with source-level mobile-code have the potential of better security than byte-code level verification as it happens in Java, as meaningful analysis and verification of program properties is possible.

4 Related work

A very large number of research projects have recently started on mobile agent programming. Among the pioneers, Kahn and Cerf's Knowbots [11] Among the most promising recent developments Luca Cardelli's Oblique project at Digital and mobile agent applications [1] and IBM Japan's aglets [10]. We share their emphasis on going beyond *code mobility* as present in Java, for instance, towards *control mobility*. We think that distributed containers with ability to negotiate with agents the resulting local and global behavior can offer a secure and flexible approach to Internet aware distributed programming. A growing number of sophisticated Web-based applications and tools are on the way to be implemented in LP/CLP languages. Among them, work with a similar emphasis on remote execution, agents, virtual worlds can be found in [3,13,4,2,18,12,24].

5 Conclusion and future work

Our remote execution mechanisms are based on a set of filtering interpreters which can be customized to support arbitrary negotiations with remote agents

[4] Note however that suspending execution at one site, and then restarting it at another site can be done quite efficiently in BinProlog, where continuations are first order objects which can be put into a term to be sent over a socket, then read in and executed.

and are plugged in generic servers. The practical implementation is built on proven client/server technology, on top of a generic socket package, while giving the illusion of a 'Web of MOOs' with roaming mobile agents at the next level of abstraction.

A Java based Linda implementation, using a minimal set of logic programming components (unification, associative search) has been recently released (the Java TermServer, available at http://clement.info.umoncton.ca/BinProlog). It allows to communicate bidirectionally 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 co-operation with existing Java class hierarchies as well as various BinProlog based LogiMOO components with intelligence and flexible metaprogramming on the logic programming side combined with visualization and WWW programming abilities on the Java side.

Future work will focus on intelligent mobile agents integrating knowledge and controlled natural language processing abilities, following our previous work described in [6,17,22].

Acknowledgement

We thank for support from NSERC (grants OGP0107411 and 611024), and from the FESR of the Université de Moncton. Koen De Bosschere is research associate with the Fund for Scientific Research – Flanders.

References

1. K. A. Bharat and L. Cardelli. Migratory applications. In *Proceedings of the 8th Annual ACM Symposium on User Interface Software and Technology*, Nov. 1995. http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-138.html.
2. P. Bonnet, L. Bressnan S., Leth, and B. Thomsen. Towards ECLIPSE Agents on the Internet. In Tarau et al. [23]. http://clement.info.umoncton.ca/ lpnet.
3. 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.
4. D. Cabeza and M. Hermenegildo. The Pillow/CIAO Library for Internet/WWW Programming using Computational Logic Systems. In Tarau et al. [23]. http://clement.info.umoncton.ca/ lpnet.
5. N. Carriero and D. Gelernter. Linda in context. *CACM*, 32(4):444–458, 1989.
6. V. Dahl, A. Fall, S. Rochefort, and P. Tarau. A Hypothetical Reasoning Framework for NL Processing. In *Proc. 8th IEEE International Conference on Tools with Artificial Intelligence*, Toulouse, France, November 1996.
7. V. Dahl, P. Tarau, and R. Li. Assumption Grammars for Processing Natural Language. 1997. accepted to ICLP'97, Proceedings to appear at MIT press, 1997.
8. K. De Bosschere and P. Tarau. Blackboard-based Extensions in Prolog. *Software — Practice and Experience*, 26(1):49–69, Jan. 1996.

9. J. S. Hodas and D. Miller. Representing objects in a logic programming language with scoping constructs. In D. D. H. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 511 – 526. MIT Press, June 1990.
10. IBM. Aglets. http://www.trl.ibm.co.jp/aglets.
11. R. E. Kahn and V. G. Cerf. The digital library project, volume i: The world of knowbots. 1988. Unpublished manuscript, Corporation for National Research Initiatives, Reston, Va., Mar.
12. S. W. Locke, A. Davison, and S. L. Lightweight Deductive Databases for the World-Wide Web. In Tarau et al. [23]. http://clement.info.umoncton.ca/ lpnet.
13. 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.
14. D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6(1 and 2):79–108, January/March 1989.
15. D. A. Miller. Lexical scoping as universal quantification. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 268–283, Cambridge, Massachusetts London, England, 1989. MIT Press.
16. A. Mycroft and R. A. O'Keefe. A polimorphic type system for prolog. *Artificial Intelligence*, (23):295–307, 1984.
17. S. Rochefort, V. Dahl, and P. Tarau. Controlling Virtual Worlds through Extensible Natural Language. In *AAAI Symposium on NLP for the WWW*, Stanford University, CA, 1997.
18. P. Szeredi, K. Molnár, and R. Scott. Serving Multiple HTML Clients from a Prolog Application. In Tarau et al. [23]. http://clement.info.umoncton.ca/ lpnet.
19. P. Tarau. BinProlog 5.40 User Guide. Technical Report 97-1, Département d'Informatique, Université de Moncton, Apr. 1997. Available from *http://clement.info.umoncton.ca/BinProlog*.
20. P. Tarau and V. Dahl. A Coordination Logic for Agent Programming in Virtual Worlds. In W. Conen and G. Neumann, editors, *Proceedings of Asian'96 Post-Conference Workshop on Coordination Technology for Collaborative Applications*, Singapore, Dec. 1996.
21. P. Tarau, V. Dahl, and A. Fall. Backtrackable State with Linear Affine Implication and Assumption Grammars. In J. Jaffar and R. H. Yap, editors, *Concurrency and Parallelism, Programming, Networking, and Security*, Lecture Notes in Computer Science 1179, pages 53–64, Singapore, Dec. 1996. Springer.
22. P. Tarau, V. Dahl, S. Rochefort, and K. De Bosschere. LogiMOO: a Multi-User Virtual World with Agents and Natural Language Programming. In *Proceedings of CHI'97*, Mar. 1997. to appear.
23. P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo, editors. *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP'96, Bonn, Sept. 1996. http://clement.info.umoncton.ca/ lpnet.
24. P. Tarau and K. De Bosschere. Virtual World Brokerage with BinProlog and Netscape. In Tarau et al. [23]. http://clement.info.umoncton.ca/ lpnet.