Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software 79 (2006) 523–536
The Journal of Systems and Software
www.elsevier.com/locate/jss

Declarative programming of integrated peer-to-peer and Web based systems: the case of Prolog

Seng W. Loke *

School of Computer Science and Software Engineering, Monash University, 900 Dandenong Road, Caulfield East, Melbourne, Vic. 3145, Australia

Received 20 August 2003; received in revised form 6 April 2005; accepted 6 April 2005
Available online 10 May 2005

Abstract

Web and peer-to-peer systems have emerged as popular areas in distributed computing, and their integrated usage permits the benefits of both to be exploited. While much work in these areas have utilized the imperative programming paradigm, the need for declarative programming paradigms is increasingly being recognized, not only for the often cited advantages such as a higher level of abstraction and specialized language features, but also to tackle the querying and manipulation of knowledge and reasoning with semantics that will be the mainstay of the proposed next generation of the Web and peer-to-peer computing. This paper presents an approach towards integrative use of the Web and peer-to-peer systems within a declarative programming paradigm. We contend that logic programming can be useful in peer-to-peer computing, especially for querying and representing knowledge shared over peer networks, and for scripting applications that involve sophisticated search behaviour over peer networks. As an example of peer-to-peer querying expressed in a logic programming language, we propose a simple extension of Prolog, which we call *LogicPeer*, to enable goal evaluation over peers in a peer network. Using LogicPeer, we outline how a peer-to-peer version of a Yahoo-like system can be built and queried, and several other applications that involve decentralized knowledge sharing. We then show how LogicPeer can be used with *LogicWeb*, a Prolog extension to access Web pages, thereby integrating peer-to-peer querying and Web querying in a common declarative framework.
© 2005 Elsevier Inc. All rights reserved.

Keywords: Prolog; Integration; Logic programming; Web computing; Peer-to-peer computing; Declarative programming

1. Introduction

In recent years, *peer-to-peer computing* has been given tremendous attention by technologists, businesses, and trend watchers. Peers share or exchange computer resources and services by direct exchange, i.e. there is a decentralization away from heavy-weight servers to equal-weight peers. The CPU cycles, disk storage, and files on a computer can be utilized by other peers leading to distributed storage which integrates the available disk space of a number of computers, distributed parallel processing, and distributed content management. Such decentralization can provide greater fault-tolerance since there is no single point of failure, distributed resource control and maintenance (e.g., distributed content management where individuals maintain the content they publish and have a sense of control over the content), greater efficiency of information exchange since traffic is not routed through central bottlenecks, and ease of set-up and usage which partly comes from distributed resource administration and in situ information sharing (e.g., information need not be transferred to another host such as a central server to be published).

With content distributed among peers in a distributed network, mechanisms including protocols and query languages for performing searches over the distributed

* Tel.: +61 3 99032614; fax: +61 3 99032863.
E-mail address: swloke@csse.monash.edu.au

0164-1212/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2005.04.005

resources are an obvious need. The Gnutella protocol,[1] has been used as the basis of several peer-to-peer systems for searching distributed content over a network of peers using simple keyword queries. Sun's new project JXTA,[2] which provides a shell for peer-to-peer programming, comes with basic built-in commands and protocols for searching within a group of peers. Gnutella,[3] FreeNet,[4] and SETI@home,[5] are examples of systems which have shown how the peer-to-peer model can be used for sharing resources directly (e.g., MP3 file-sharing) with little or no intervention from a central server or authority.

Logic programming languages such as Prolog have been explored for Internet applications (Davison, 2002) resulting in high level models, and extensions for manipulating Web contents. Logic programming provides a higher level of abstraction than imperative languages for knowledge representation and processing. Also, logic programming languages such as Prolog has features such as backtracking search, pattern matching, declarative syntax, and meta-programming. For data representation, deductive databases generalizes relational databases.

In this paper, we apply the declarative programming paradigm for integrating Web and peer-to-peer computing by providing lightweight extensions of Prolog for this purpose. While we use Prolog, we see that the approach can be utilized in other logic programming languages and formalisms that are emerging such as Mercury (Somogyi et al., 1995) and other Semantic Web logic languages.[6] Prolog provides a useful basis to illustrate and develop our ideas due to its simple semantics, widespread familiarity, and more than twenty years of research to leverage on.

Our contributions in this paper are as follows. In Section 2, as an example of peer-to-peer queries expressed in a logic programming language, we describe a simple extension of Prolog with operators to access peer networks, which we call *LogicPeer*, to enable goal evaluation over peers in a peer network (which is not done by Nejdl et al., 2001). We argue that logic programming can be useful for querying and representing knowledge shared over peer networks, and for scripting applications that involve heuristics based sophisticated search behaviour over peer networks. We do not advocate logic programming as a replacement of current languages for all peer-to-peer applications but aim to highlight the benefits of logic programming for specific kinds of peer-to-peer applications. In Section 3, using this extended Prolog, we outline how a peer-to-peer version of a Yahoo-like system can be built and queried. We also briefly consider several other applications that involve decentralized knowledge sharing. Section 4 presents another extension to LogicPeer goals to enable cooperative goal evaluation. In Section 5, we show how our extended Prolog can be used with LogicWeb (Loke and Davison, 1998) thereby integrating peer-to-peer querying and Web querying in a common framework. We consider related work in Section 6 and conclude in Section 7.

We assume an acquaintance with Prolog in the discussions below.

2. Extending Prolog for peer-to-peer querying

2.1. Characteristics of peer-to-peer computing

We make the following assumptions about a peer network:

- Each peer has a unique identifier called the *peer identifier*. This can be an IP address and port number but need not be so. A peer-to-peer system normally has its own namespace. For example, ICQ[7] has ICQ numbers to identify peers which are independent of network addresses. We do not assume any particular format for peer identifiers but only that they exist. We also assume the existence of name mapping mechanisms (e.g., to resolve ICQ numbers to actual IP addresses). We show that LogicPeer is independent of the format of peer identifiers and the underlying mechanisms for resolution.
- Peers can send messages among each other using peer identifiers.
- A peer does not have the peer identifier of every other peer in the peer network but we assume that a peer has a set of peer identifiers for peers it knows about, which we call the *friends* of the peer. We assume that a peer has some (perhaps out-of-band) means of discovering other peers in the network in order to build or update this set of friend identifiers. Peer directories and IP multicast might be used for discovering friend identifiers as in Napster servers, Gnutella reflectors,[8] or JXTA peer directories.
- In our model, each peer has a collection of Prolog rules (and facts) against which goals it receives from other peers and goals initiated locally are evaluated. This collection of Prolog rules might include the knowledge base to be shared by this peer. This main-

[1] Gnutella protocol specification is at http://capnbry.dyndns.org/gnutella/protocol.php.
[2] http://www.jxta.org
[3] http://gnutella.wego.com/
[4] http://www.at-web.de/p2p/freenet.htm
[5] http://setiathome.ssl.berkeley.edu/
[6] See http://www.semanticweb.org/inference.html for a list.
[7] http://www.icq.com
[8] See http://www.clip2.com.

tains generality and does not exclude other forms of media that a peer might serve. For example, in a peer network sharing MP3 files, the Prolog fact database of a peer might store details (e.g., name, singer, etc) of these MP3 files.
- Each peer will have integrity and will not provide results that are false. For example, if it does not have a given MP3 file, it should not say it does. Details of how to ensure such integrity is outside the scope of this paper.

As an example of a search protocol operating over a peer network, we briefly outline Gnutella.[9] Gnutella is an example of a pure peer-to-peer system which has been used for file-sharing. A Gnutella program is configured with the addresses of a set of peer Gnutella programs that it can connect to. Each running Gnutella program (a peer) has access to a set of files that the user wants to share. The Gnutella protocol permits a recently started Gnutella program to send "ping" messages to discover other reachable running Gnutella programs, and to send search queries to the peers it discovers. On receiving a search query, a peer checks to see if it can satisfy it, and if it can, replies to the requestor. The reply goes back along the same path that the query took in reaching the peer. On receiving the reply, the requestor can download the information using the Web's HTTP protocol directly from the peer that replied. If the peer cannot satisfy the query, it passes it on to its friends and this process might continue with subsequent peers. A time-to-live value limits the lifetime of messages and each peer keeps track of the messages it forwarded so that messages are not forwarded again. Peers that have much content (and so satisfy more user queries) and are reachable via high bandwidth connections will tend to be more popular.

In the following subsections, we explore extensions of Prolog with new constructs that enable query (or goal) evaluations to take place over multiple peers in a peer network.

2.2. The opaque peer network model—LogicPeer I

In this model, we treat the peer network as opaque and assume an underlying protocol as the means for propagating Prolog queries among peers. The underlying protocol we employ here is the Gnutella protocol but with one main difference that a peer can convert a query into another query when passing it on. Also, replies are sent back along the same path as the query and no further downloading might be required depending on the query.

[9] An animation illustrating the Gnutella protocol can be found at http://www.limewire.com/article.htm.

We introduce a new kind of goal which we call the *global goal* written with a prefix "*" such as `*goal`. This goal is evaluated by being propagated across the peer network. We assume that each peer on the peer network have the means to receive, process and if necessary forward such goals to its peers. As an example, consider the following program which defines the parent relationship.

```
father_of(john,tim).
father_of(john,jack).
mother_of(angie,tim).
parent_of(X,Y) :-
  father_of(X,Y)
 ; mother_of(X,Y)
 ; *parent_of(X,Y)
```

";" is "or" in Prolog. This rule states that `X` is a parent of `Y` if `X` is the father or mother of `Y`, or `X` is the parent of `Y` as determined by other peers in the peer network. The first two goals in the body of the rule will be evaluated locally in the peer in which this program resides and in which this rule is invoked. The last goal `*parent_of(X,Y)` will be evaluated as follows: the goal is first passed to the peers of the local peer and propagated via the Gnutella protocol, and then solutions, if any, in the form of bindings to goal variables, or simply true or failure will be returned by peers in the peer network. Now, consider the following goal:

```
-?parent_of(john, tim).
```

Typically, such a goal can be typed into a Prolog command shell similar to a Unix command shell or the JXTA peer shell (Gong, 2001). This goal will be evaluated locally to be true but the following goal if evaluated only locally will fail (by Negation-As-Failure) due to the limited set of facts about `alice`:

```
-?parent_of(P, alice).
```

However, when this goal invokes the `parent_of/2` rule above, evaluation is not just local. The global goal at the end of the rule will be propagated to the friends of this peer to ask them for their answers. Within a prespecified waiting period, the local peer waits for and collects answers to the global goal. As in the Gnutella protocol, the time-to-live values on transmitted goals help to restrict propagation. It is also possible to restrict goal propagation to those peers within a peer group provided that the concept of a peer group is supported as in JXTA. Two questions now arise when a peer (other than the local peer) receives this goal (which we call the *original goal*):

1. What does the peer do with the goal and what are the possible replies?

2. Also, how does the originating peer decide on the answer to its goal if a number of answers are returned, some of which might be inconsistent?

2.2.1. For (1)

When a peer receives a goal (which comes with the sending peer's identifier), it first evaluates the goal against its own rule (and fact) database. The result of the goal evaluation is either success or failure. For success, the result is either true (if the goal has no unbound variables), or if the goal has unbound variables, the set of all possible bindings for each variable (e.g., possible values for P in the above goal). There will be no bindings in the case of goal failure. Failure of a goal can be due to inadequate information in the database and so, a goal to this peer is false by Negation-As-Failure.

Note that evaluation of the original goal might result in the evaluation of other global goals as specified in the bodies of invoked rules in this peer, and so the original goal might result in new goals being propagated through the network. For example, suppose this peer has the same rule as the sending peer:

```
parent_of(X,Y) :-
  father_of(X,Y)
 ; mother_of(X,Y)
 ; *parent_of(X,Y)
```

Then when evaluating the received goal, this rule is invoked which then initiates another global goal. In this case, the same global goal is passed on to the peers of this peer. It is also possible that a global goal is effectively "mapped" into another global goal. For example, suppose the rule on this peer was instead the following:

```
parent_of(X, Y) :- *child_of(Y,X)
```

Then, when evaluating the received goal, this rule is invoked which then initiates another global goal but this time, a different goal from the received goal is passed on to the peers of this peer. Hence, transformation of gueries is possible, unlike the Gnutella protocol. Now, after evaluation, the peer (if we assume somewhat autonomous) can now choose not to reply at all to the requesting peer, or it can return whatever results of its evaluation.

A question arises concerning queries which go unbounded in the network or which lead to infinite loops. One way to solve this problem is to tag each query (or goal) with a time-to-live (TTL) value which is modified (with time taken so far subtracted from it) across queries (and in our case when one goal is mapped into another, the TTL value is reduced even as they are passed between the different goals). Loops between peers (i.e., when one peer issues a query to another which in turn issues a query back to the other peer) can be prevented by such a TTL value. Other possible control tags are possible, such as peers-to-live (PTL) value which is a count of the maximum number of peers through which a goal and its counterparts (generated when mapping one goal to another via rule execution) can be propagated. For example, a goal initiated from a peer P1 with a PTL value of 3 means that it can at most be allowed to propagate through three (not necessarily distinct) other peers. In the case of a potential loop between P1 and P2, the goal can go from P1 to P2, from P2 to P1, and then from P1 to P2 and no more, or in the case of distinct peers, go from P1 to P2, P2 to P3 and then P3 to P4, and no further. Other finer grained control tags are (1) number of rules invoked, i.e. the goal can at most be used to (directly or indirectly) invoke up to N rules and no more, and (2) depth of query tree with peers as nodes in goal evaluation.

2.2.2. For (2)

When the originating peer receives answers from peers, it needs to collate the results to provide a result for the original goal. We adopt the following method for combining results. All failure results are ignored, and if there is at least one true result, the original goal is true, and if there are bindings (i.e. open variables in the original goal), then the possible bindings for the original goal is the union of all possible bindings received (possibly from more than one peer). Hence, a goal such as `father_of(jack, alice)`, which queries if `jack` is the father of `alice`, is considered true if any one peer declares this as true (and returns bindings for open variables (if any)). The different possible bindings are retrieved upon backtracking or can be obtained collectively via `setof/3` or `findall/3`. We briefly discuss how we might handle inconsistent results from different peers in §7.

2.3. The transparent peer network model—LogicPeer II

In contrast to the LogicPeer I, in this model, the peer network is treated as transparent in that it is possible for a peer to obtain the identifiers of the friends of its friends, etc. The additional assumption here is that a peer can be queried for the identifiers of its friends by a logic program running in itself or a remote peer. By allowing a logic program to access the friends of friends recursively, customized search algorithms can be written which effectively (and perhaps partially) traverse the graph formed by the peer network. We first introduce the following new kinds of goals formulated with the following constructs.

1. **Providing explicit access to the friends of a peer**. We make the friends of a (local or remote) peer visible at the program-level via goals of the following form:

```
PeerID*friends(PL)
```

where PeerID is the identifier of a peer and PL is a list of peer identifiers for the friends of the peer. To obtain the friends of the local peer, we can use the goal:

```
-?self*friends(PL)
```

Note that writing self * Goal is the same as writing Goal in the local peer. We call "*" the *peer-switching* operator, as its meaning switches goal evaluation to (occur in) the specified peer.

2. **Send a goal to a specific peer**. To send a goal to be evaluated at a specific peer, we use goals of the form:

   ```
   PeerID*goal
   ```

   We prefix the goal by the peer identifier and "*".
3. **Concurrency**. We introduce two programming abstractions to simplify concurrent programming with peers:
   (a) A goal of the form
   PL[]goal
   evaluates in such a way that the goal must succeed with all the peers in the peer list PL and the results combined using the method given in the previous subsection. The goal PL[]goal fails if the result from at least one peer is failure. Evaluation of goal proceeds concurrently with all the peers in PL.
   (b) A goal of the form
   PL<>goal
   evaluates in such a way that it succeeds as long as the goal succeeds with at least one of the peers in the peer list PL. Failures are ignored and the results from successful evaluations are combined using the method given in the previous subsection. Also, if a peer is not contactable, it is ignored. The goal PL<>goal fails if the results from all the peers in PL are failures. Evaluation of goal proceeds concurrently with all the peers in PL.

The above constructs are inspired by modular logic programming where a goal (e.g., G) can be written which is evaluated against an explicitly stated logic program (e.g., M) and are written typically as M::G in rule bodies. These constructs can be viewed as a simple Prolog API to the peer-to-peer computing model.

As an example, the following program finds a peer which satisfies a given goal. The same goal is evaluated against the peers in a peer network in a depth first search manner starting from the local peer until the goal succeeds with some peer:

```
dfs_eval(Goal) :-
  eval(self, Goal).
    % try locally first
eval(P, Goal) :-
  P * Goal, !.
    % send goal to the peer
eval(P, Goal) :- % evaluate goal
  P * friends(PL),
    % get access to the
    % friends' identifiers
  member(Pl, PL),
    % choose a peer from
    % the friends list
  eval(Pl, Goal).
    % recursively, send
    % Goal to the chosen peer
```

In the first rule of the predicate eval/3, evaluation begins first with the given peer and once the goal succeeds, subsequent peers are not contacted. If evaluation fails, in the second rule, the peer P is first queried for its list of friends, and then one of the friends is selected via member/2 and the goal is recursively evaluated against the selected friend. The goal P * friends(PL) will fail if either the peer P cannot be contacted or its friends cannot be obtained, in which case, Prolog backtracking on the member/2 goal will result in another friend being chosen from the friend list. Evaluation completes when one peer P is found such that P * Goal succeeds, or there are no more peers to try.

The key difference between this program and LogicPeer I is that here, we explicitly send the same goal to the peers instead of relying on peers to propagate the goal to other peers. Hence, we allow the programmer to explicitly state who goals must be sent to and what these goals are. However, due to the lack of decentralized control as is the nature of peer-to-peer computing, there is no guarantee about what peers will do with the goals they receive. For example, a peer might further propagate the goal or it might not, and so, the above program might result in the same goal being propagated to the same peer multiple times. In the case where such a program is executed in a peer network where goals are not passed on by any particular underlying protocol, the ability of a peer to programmatically access the friends of another peer would permit control over how goals are forwarded to peers. Cooperation, however, is not a problem in a given peer network of knowledge bases, viewed as an integrated system (e.g. PeerDB Ng et al., 2003).

A concurrent version of the above search can be coded as follows, where a goal is evaluated against all the friends of a peer concurrently and the peers are visited in a depth first manner until the goal succeeds with some peer:

```
dfs_eval(Goal) :-
  Goal, !.
  % try locally first;
  % same as self * Goal
dfs_eval(Goal) :-
  concurrent_eval(self, Goal).
  % try peers concurrently
concurrent_eval(P, Goal) :-
  P * friends(PL),
    % retrieve friends
  try_goal(PL, Goal).
    % try the goal with friends
try_goal(PL, Goal) :-
  PL<>Goal, !.
  % send goal to all
  % friends concurrently
try_goal(PL, Goal) :-
  member(Pl, PL),
    % choose a friend
  concurrent_eval(Pl, Goal).
    % continue recursively
```

In the first rule of `dfs_eval/1`, the goal is first tried locally. If it fails, evaluation continues with `concurrent_eval/2` where the friends of the peer `P` are first retrieved and then sent to `try_goal/2`. In the first rule of `try_goal/2`, `Goal` is evaluated against all the friends concurrently. The goal `PL<>Goal` succeeds if `Goal` succeeds with one of the peers in `PL` and fails, otherwise, and on failure, the second rule of `try_goal/2` is invoked where one friend is selected for continued evaluation, recursively visiting the friends of this friend, etc. Evaluation completes when one peer is found where `Goal` succeeds, or there are no more peers to try. Due to parallelism, this version is more efficient on average than the previous version but more peers are contacted (i.e., more network traffic is generated).

The above two programs find a peer which satisfies a given goal but does not return the identifier of the peer itself. It is possible to return the value of this peer as a binding of a variable in the goal itself. So, if a peer succeeds with the goal, it returns the results which also includes its own identifier. Alternatively, the following version of the first program can be modified to return this peer (`NP` is instantiated with this peer's identifier):

```
dfs_eval(Goal, NP) :-
  eval(self, Goal, NP).
    % try locally first
eval(P, Goal, P) :-
  P * Goal, !.
    % evaluate goal
eval(P, Goal, NP) :-
   % evaluate goal
P * friends(PL),
   % get access to the
   % friends' identifiers
member(Pl, PL),
  % choose a peer from
  % the friends list
eval(Pl, Goal, NP).
  % send the goal to
  % the chosen peer
```

Using the operator "`[]`", one could write a program which finds a peer all of whose friends satisfy a given goal. For example, the program to find a peer who is supported by all its friends (i.e., fully supported) can be given as follows, where a predicate `supports/1` defines the meaning of a peer supporting a peer and whose argument is the supported peer's identifier:

```
fully_supported(P, SP) :-
  P * friends(PL),
    % retrieve friends
  all_supports(PL, P, SP).
    % check for PL support
all_supports(PL, P, P) :-
  PL[]supports(P), !.
    % evaluate the goal
    % supports/1
all_supports(_P, PL, SP) :-
  member(Pl, PL),
  fully_supported(Pl, SP).
    % check a chosen friend
```

The fully supported peer is returned in the variable `SP`. The goal `PL[]supports(P)` concurrently checks support with all the peers in `PL` succeeding only if all the peers supports `P`.

2.4. An operational semantics for peer-switching

We outline an operational semantics for the language of pure Prolog augmented with peer-switching. Such a language consists of clauses of the form

$\mathscr{A} : -\mathscr{G}$

where $\mathscr{G}$ is defined by

$\mathscr{G} ::= \mathscr{A} | \mathscr{P} * \mathscr{G} | (\mathscr{G}, \mathscr{G})$

$\mathscr{A}$ is an atomic goal and $\mathscr{P}$ is a peer identifier.

We consider executing the first program from §2.3 starting at peer P1, over the following acyclic peer network as shown in Fig. 1. The arrows from a peer P to a peer Q represents the fact that Q is a friend of P, i.e. Q is in P's friends list. Note that since all the peers are on the Internet, we assume that a peer can connect to any other peer if it knows its peer identifier (even if the peer is not currently in its friends list). For example, P1 has P2 and P3 in its friends list and does not know about P4. But if P1 obtains P4's identifier sometime later, it can connect directly to P4.

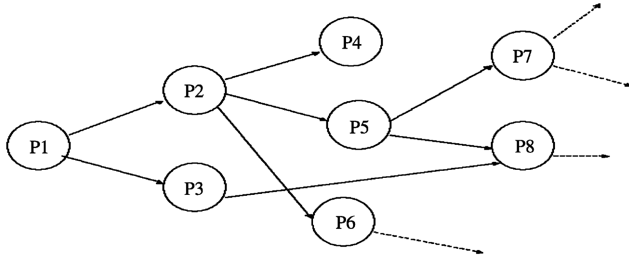

Fig. 1. An example peer network.

Consider the following execution trace of the goal evaluation, each line prefixed by the location where the goal is evaluated:

```
P1:eval(P1, goal)
P1:  P1*goal   // suppose goal failed
                 // when evaluated at P1
P1:  P1*friends(PL)  // the friends of
                       // P1 are retrieved
  ...
P1: eval(P2, goal)   // suppose the
             // member call retrieves P2
P1:P2 * goal       // send goal to node P2
                     // and evaluate it there
P2:goal          // evaluate goal at P2
           // and results returned to P1
P1:P2 * friends(PL1)  // suppose goal
    // at P2 failed, retrieve P2s friends
    // bringing the list of
    // P2s friends back to P1
P2:friends(PL1)  // evaluated at P2,
                 //results returned to P1
        ...
P1:eval(P4, goal)
        // suppose P4 was selected
P1:P4 * goal  // send goal to
      // node P4 and evaluate it there
P4:goal  // evaluate goal at P4
          // and results returned to P1
```

We first define the following relation $\mathscr{P}\vdash_\theta\mathscr{G}$ to hold when the goal $\mathscr{G}$ succeeded at peer $\mathscr{P}$ with result θ (the substitutions). The rules for such evaluation are as follows in the format [label] $\frac{\text{premises}}{\text{conclusion}}$, where *conclusion* holds whenever the *premises* hold:

$$[\text{true}]\frac{}{P\vdash_\epsilon \texttt{true}}$$

$$[\text{atom}]\frac{P\vdash_\theta H{:}-G \wedge \gamma = mgu(A, H\theta) \wedge P\vdash_\delta G\theta\gamma}{P\vdash_{\theta\gamma\delta}A}$$

$$[\text{peer-switching}]\frac{Q\vdash_\theta G}{P\vdash_\theta Q * G}$$

$$[\text{conjunction}]\frac{P\vdash_\theta G_1 \wedge P\vdash_\gamma G_2\theta}{P\vdash_{\theta\gamma}G_1, G_2}$$

Note that in peer-switching, the goal is sent from P to Q and results are brought back from Q to P. And as can be seen from the above trace, P1 maintains control of the evaluation in the sense that goals are sent from P1 to P2 and from P1 to P4, instead of goals being propagated directly from P2 to P4 as in the opaque model. Note that such evaluation does not require that P1 knows of every peer (e.g., even P2's) before evaluation as this defeats the purpose of the scalable peer network, but the friends of peers are maintained by peers individually and are retrieved by P1 only at goal evaluation time.

However, it is also possible to program the direct propagation of goals from one peer to another using a goal such as `P * (Q * (R * goal))`. Evaluating this goal will cause the goal `Q * (R * goal)` to be sent to P, and the goal `R * goal` to be sent to Q, and `goal` to be sent to R.

Now suppose we evaluate the goal

```
P * (goal, Q * (goal, R * goal))
```

Then, `(goal, Q * (goal, R * goal))` will be sent to P. At P, `goal` will be evaluated and `(goal, R * goal)` will be sent to Q. At Q, `goal` will be evaluated and `goal` will be sent to R.

Further control can also be retained assuming we have the Prolog conditional $G\text{–}>A;B$. We could have the following goal:

```
goal ->
  true
  ; P * (goal ->
    true
    ; Q * (goal ->
      true
      ; R * goal
    )
  ).
```

which evaluates as follows: first evaluate `goal` locally, and if it fails, send `(goal -> true; Q * (goal -> true; R * goal))` to P. At P, evaluate `goal`, and if it fails, send `(goal -> true; R * goal)` to Q. At Q, evaluate `goal` and if it fails, send `goal` to R. The claim is that with the cooperation of peers who will respect this semantics, such control of goal propagation can be programmed.

3. Applications

This section aims to illustrate the convenience of LogicPeer for applications involving the querying and manipulation of structured knowledge over peer

networks. Metadata matching with queries is usually based simply on attribute-value matching. We believe that the logic programming approach can go further, and would enable rule-based reasoning with resource descriptions and more expressive queries. Below, we first describe an example in greater detail illustrating the above mentioned advantages, and then point out several other applications.

3.1. The DistributedYahoo peer network

We consider building a decentralized version of the popular Yahoo meta-index[10] using a peer network where each peer maintains its own concept hierarchy similar to the Yahoo categories and sub-categories. On a peer, one way to represent a concept hierarchy is via IS-A relationships as noted by Loke and Davison (1997). For example, a portion of a concept hierarchy about AI, logic programming, and agents might take the form of the predicate `isa/2` representing an IS-A relation between concepts in the form of `isa(Subconcept, Concept)` links:

```
isa(logic_programming, ai).
isa(agents, ai).
  :
isa(agent0, agent_languages).
```

To build an IS-A hierarchy, we define the transitive closure on `isa/2`:

```
trans_isa(Concept1, Concept2) :-
  isa(Concept1, Concept2).
trans_isa(Concept1, Concept2) :-
  isa(Concept1, Mid),
  trans_isa(Mid, Concept2).
```

We associate URLs with concepts using `page_about/2`:

```
page_about(agents,
  http://machtig.kub.nl/agents.html).
page_about(agent0,
  http://www.scs.ca/agent0.html).
  :% more page_about/2 facts
```

`trans_about/2` defines a transitive version of `page_about/2` which associates a concept with a URL if one of its sub-concepts is associated with that URL:

```
trans_about(Concept, URL) :-
  page_about(Concept, URL).
trans_about(Concept, URL) :-
  trans_isa(SubConcept, Concept),
  page_about(SubConcept, URL).
```

Each URL above might also be associated with a textual description or metadata fields, i.e. with facts of the form: `page_description(URL, Description)`.

Each peer participating in the DistributedYahoo network might maintain its own (similar) hierarchy and perhaps not exactly the same, and might have a different collection of URLs or some URLs in common with other peers. A query such as the following

```
-?trans_about(agents, URLs)
```

will query the local concept hierarchy for URLs about "agents" or about its subconcepts. A global goal such as the following

```
-?*trans_about(agents, URLs)
```

will query the peers in the Distributed-Yahoo peer network for such URLs.

Apart from querying about URLs, it is possible to find subconcepts of a given concept using a query such as:

```
-?*trans_isa(SubConcept, agents)
```

If different peers have different hierarchies, each peer would return different answers to this query. It is also possible to provide a classification of a given URL by a query such as:

```
-?page_about(Concept,
  http://www.w3.org).
```

Again, the concept hierarchy might vary with different peers, and so, each peer might return a different classification of the URL. Other kinds of centralized taxonomies could be distributed and queried in a similar way.

3.2. Other applications

Decentralized peer-to-peer versions of applications involving knowledge representation, querying, and matching can be implemented, such as service discovery where queries are matched to find particularly services in a network, expertise location where queries are matched against profiles of experts to find a required expert—e.g., a peer-to-peer version of the centralized expertise location system by Vivacqua (1999), and question-answering where questions are answered from a database of information such as the Web or Frequently Asked Questions (FAQ) collection. In all these exam-

[10] http://www.yahoo.com

ples, logic programming provides a rich representation language (e.g., for service matching, profile matching, and question-answer matching). An advantage of the peer-to-peer model in such applications is the decentralized knowledge maintenance, distributed handling of queries, and in situ knowledge-sharing.

4. Cooperative goal evaluation among peers

This section introduces an operator to enable peers to cooperate in solving a goal. For example, friend peers in different subtrees (or peers) and in different levels of the query tree which do not know each other can effectively cooperate in solving a goal.

A motivating example is as follows. The original node P0 invokes a query A that can be true if and only if B and C are true. P0 has three friend peers P1, P2 and P3, and directs subquery B to P1 and subquery C to P2. If P1 knows that B is true, it will return true B to P0, and P2 knows that C is true if and only if D and E are true. However, P2 only knows that D is true and P3 only knows that E is true. If P2 does not know P3, then P2 will return a false C to P0 because P2 does not know that E is true. Thus, P0 will be unable to prove A to be true, even though the information is available from its friend peers.

The problem is not with LogicPeer itself but with the following rule for A on P0 which directs subgoal evaluations to specific peers:

```
A :- P1 * B, P2 * C.
```

and the rule which P2 has for C:

```
C :- D, E.
```

P2 has the fact for D and so can prove D but is unable to prove E as it does not know about P3 (who knows E).

It would be convenient to enable P0 to allow its friend peers to effectively cooperate in answering its goals. For this purpose, we introduce an operator which we call *cooperate-with* denoted by "+" analogous to Brogi's union operator on logic programs in Brogi (1993). We rewrite the rule on P0 as follows:

```
A :- (P1+P2+P3) * (B,C).
```

which means that to evaluate A, evaluate the conjunction of B and C using the cooperation of peers P1, P2 and P3. The effect is analogous to the semantics of Brogi's union, which supposing that "+" is Brogi's union and P1, P2 and P3 are logic programs (i.e. sets of clauses), evaluates the conjunction against the logic program formed by the set theoretic union of the clauses from P1, P2 and P3. Clearly then, the conjunction of B and C will evaluate to true in the combined logic program of four clauses:

```
B.            % from P1
C :- D, E.    % from P2
D.            % from P2
E.            % from P3
```

However, different from Brogi's union, the peers P1, P2 and P3 might be on separate machines and so the evaluation needs to contact multiple hosts. The extended syntax for goals $\mathscr{G}$ is then

$$\mathscr{G} ::= \mathscr{A} | \mathscr{E} * \mathscr{G} | (\mathscr{G}, \mathscr{G})$$

where $\mathscr{E}$ is an expression given by

$$\mathscr{E} ::= \mathscr{P} | \mathscr{E} + \mathscr{E}$$

where $\mathscr{P}$ is a peer identifier.

The operational semantics with "+" is given in the following rules which augments the rules given earlier:

$$[\text{true}]\frac{}{E \vdash_{\epsilon} \texttt{true}}$$

$$[\text{atom}]\frac{E \vdash_{\theta} H :- G \wedge \gamma = mgu(A, H\theta) \wedge E \vdash_{\delta} G\theta\gamma}{E \vdash_{\theta\gamma\delta} A}$$

$$[\text{peer-switching}]\frac{F \vdash_{\theta} G}{E \vdash_{\theta} F * G}$$

$$[\text{conjunction}]\frac{E \vdash_{\theta} G_1 \wedge E \vdash_{\gamma} G_2\theta}{E \vdash_{\theta\gamma} G_1, G_2}$$

$$[\text{cooperate-with}: 1]\frac{E \vdash_{\theta} H :- G}{E + F \vdash_{\theta} H :- G}$$

$$[\text{cooperate-with}: 2]\frac{F \vdash_{\theta} H :- G}{E + F \vdash_{\theta} H :- G}$$

$$[\text{clause-from-a-peer}]\frac{(H :- G) found\ at\ P}{P \vdash_{\epsilon} H :- G}$$

The two rules for cooperate-with specifies a non-determinism which allows clauses to be retrieved from any of the peers mentioned in the expression $E + F$. In practice, this behaviour might be implemented by issuing a query for the same clause to each of the peers concurrently and stopping either when all peers fail (the clause is not found in any of the mentioned peers) or upon the first peer returning a success (the clause is found in at least one of the mentioned peers).

The rule $[clause-from-a-peer]$, the cooperate-with rules and the first two conjuncts in the premise of the $[atom]$ rule represent the search for a matching clause for a goal. Messaging is involved from the querying peer to the queried peer as follows (unless the queried peer is the same as the querying peer): given a query issued from a peer P0 to find a clause $H :- G$ of a peer P, whose head matches with a goal A, if found, the subgoal G (or true) and answer substitutions θ and γ are returned from P to P0 so that computation can continue at P0 (represented by the last conjunct in the premise of the $[atom]$

rule); otherwise, a message indicating the failure to find a matching clause is returned.

For example, evaluating a goal such as `(P1 + P2 + P3) * C` will cause a search for a matching clause for `C` to be done concurrently with the peers P1, P2 and P3. When one is found, say P2 in the example above, goal evaluation will continue with evaluating the conjunction of D and E in the cooperation of P1, P2 and P3. The result is that D is proved from P2 and E from P3, and so the conjunction of D and E holds in the cooperation of P1, P2 and P3. Revisiting our motivating example above, with the rule

```
A :- (P1 + P2 + P3) * (B,C).
```

query A, which generates the subgoal `(P1 + P2 + P3) * (B,C)` will be evaluated to true due to the cooperation of P1, P2 and P3. Note that the cooperation of P1, P2 and P3 was initiated by P0 without P1, P2 and P3 exchanging messages among each other. In fact, P2 still might not know about P3 since all answers go back to P0—the idea is that different peers solve different subgoals in establishing a given goal (thus in effect working together towards the common goal).

The cooperate-with operator captures succinctly the sophisticated computation and messaging to establish a goal using multiple peers, with well-defined semantics (analogous to Brogi's union). While cooperate-with increases the chance of goals succeeding, it, in general, also involves much network traffic if a large number of peers are involved, and so should be used when a more directed search is not possible.

5. Integration with the Web

In Gnutella file-sharing, the Gnutella protocol is only used to locate the required file and not to retrieve the file itself. Rather than reinventing another protocol to download files, Gnutella uses the HTTP Web protocol. In this section, we provide examples of how our Prolog peer API can be used with a Prolog Web API for applications. There are many APIs for accessing the Web from Prolog programs as surveyed by Davison (2002). As an example, we consider LogicWeb (Loke and Davison, 1998).

5.1. LogicWeb

In brief, LogicWeb views the Web as a collection of logic programs which can be composed together using operators such as union, intersection, and encapsulation (Loke and Davison, 1998). This programming style makes it much easier to implement structured data representations on top of the Web, including light-weight databases and concept nets.

One of the key components of LogicWeb is the *context operator*:

```
lw(RequestMethod, URL)#>Goal
```

This executes `Goal` against the logic program called a *LogicWeb program* specified by the Web request method and the URL. Low-level issues such as page retrieval, parsing, and conversion into logic program format are hidden. In addition, if the program required by `Goal` is already on the client-side (because it was previously downloaded) then it is not retrieved again. LogicWeb programs are constructed from the data (e.g., a Web page) returned by HEAD, GET or POST HTTP requests. A LogicWeb program consists of facts storing the contents of a Web page such as the title, Web links, the text, and Prolog rules stored within the page (with a non-standard "`<LW_CODE>`" tag). Thus, the "`#>`" operator permits the programmer to think of Web computation as goals applied to programs, with no need for explicit Web page retrieval, parsing, or storage. This abstraction away from network issues, such as latency, bandwidth, and connection failure, is very useful for many kinds of programs.

5.2. LogicWeb and LogicPeer

The combination of LogicWeb and LogicPeer provides a means to interface with the Web and peer-to-peer networks within the same language. Both LogicWeb and LogicPeer can be used in the same Prolog rule. For example, the following predicate `get_info/2` first sends out a query to the peer network to find the URL of a server containing the required information, and then downloads the information:

```
get_info(query(Details, ServerURL),
      Info) :-
   *query(Details, ServerURL),
   lw(ServerURL,
   get)#>h_text(PageContents),
   extract_information(PageContents,
   Info).
```

The global goal `*query(Details, ServerURL)` has `Details` instantiated with a query and `ServerURL` open. The result of this global goal, on succeeding, is to instantiate `ServerURL` with the URL of a server satisfying the given query. Next, the LogicWeb goal retrieves the page from the given URL and `extract_information/2` extracts the desired information from the page. More generally, peer networks can be used as a means to finding URLs to resources. Conversely, Web pages can be used as a means to finding peer identifiers.

In previous work with LogicWeb, we have also embedded logic programming rules to Web pages, and these rules are invoked when the user clicks on links. The result is that Web link behaviour on such pages can be defined using rules. We present an example first reported in (Loke and Davison, 1997) of a `link_action/1` rule invoked by clicking on a Web link. This example maps names to URLs.

Currently, when a Web page is moved (e.g., from one directory to another), the links to it from other pages will cease to work. One solution is to define links in terms of page names rather than addresses, and use a central database (or databases) to map these names to their actual URLs. When an author moves a page he must update its URL in the relevant database, but the page name remains the same. This means that users of a page (who use the page name as a link reference) are unaffected by the movement of the page.

We imagine a database of page names and URLs of the form:

```
page_details(SW_Page,
  http://www.cs.mu.oz.au/~swloke).
page_details(Andrew's_Page,
  http://fivedots.ac.th/~ad).
    :
```

This database will be stored in a globally known Web page at http://www.db.com/details.html. We extend the URL syntax to include a page name reference such as http://www.db.com/details.html-$-SW_Page.

When such a URL is dereferenced on a page, we must define its meaning with `link_action/1`:

```
link_action(URLstring):-
  % first breaks the URLstring
  % into components
  break([DbURL, -$-, PgName],
    URLstring),
  lw(get,   DbURL)#>page_details(PgName,
    PgURL),
  show_page(PgURL). % displays the page
```

Such a rule is embedded within a Web page. So, when a link (within this page) with a URL string such as http://www.db.com/details.html-$-SW_Page is selected, the `link_action/1` rule will be invoked with this string as argument, retrieving the actual URL from the page details database.

With LogicPeer, it would be possible to not only retrieve the actual URLs of pages from a central Web database, but also to query peers for the actual URL, with code similar to `get_info/2` above. This means that clicking a Web link will result in a query being sent to a pre-programmed peer network (e.g., to friends) to find a desired URL, and then that URL is then used. We can therefore move from a central database of page names and URLs to a decentralized model—a decentralized database of page names and URLs distributed over a number of peers. It might even be possible, in this example, to contact the peer program of Seng to find the URL to Seng's page. Hence, what we achieve is an integration of peer-to-peer computing with Web linking. Of course, limits must be set on wait times for such peer-to-peer queries.

A different way of integrating peer-to-peer computing with Web servers, similar to the idea of federating Web servers, is to form a peer network where each peer is a Web server. A Web server on receiving a Web request tries first to satisfy that request locally and if it fails to do so, passes that request on to its friends. A reply from the Web server to the client might then not just be "File Not Found" but "File Not Found but Please wait, I will check with my friends". Such Web server behaviour can be coded in our language, and in fact, it is a translation of Web queries into peer-to-peer queries. Web queries might get translated into queries to a peer network and back to Web queries and so on, thereby integrating these two information spaces.

Queries might also have a more constructive flavour, that is, instead of leaving queried repositories unchanged, it might be possible for queries to update repositories. For example, in an integrated LogicPeer and LogicWeb language, one can retrieve knowledge from Web pages and insert them into peer networks and conversely, one can retrieve knowledge from peer networks and insert them into the Web.

6. Related work

Many peer-to-peer frameworks have been implemented in imperative languages (e.g., Java). More recent work have used SQL-based querying for peer-to-peer databases (e.g., Hoschek, 2002; Ng et al., 2003) and Datalog based querying for RDF-based metadata in peer-to-peer networks (Nejdl et al., 2001). However, they do not provide the peer-to-peer model of accessing friends as we have provided—our approach allows queries to be directed to particular friends as dictated by a LogicPeer program, enabling programmer control over how queries should be evaluated and propagated over a peer-to-peer network.

Integrating FIPA multiagent technology with peer-to-peer computing has been proposed[11] and several workshops have explored the integration between the agent paradigm and peer-to-peer computing towards

[11] See http://www.fipa.org/docs/output/f-out-00076/

more active peers.[12] It would not be too difficult to implement interaction protocols among logic programming agents to simulate Gnutella-like behaviour. We do not seek to solve all peer-to-peer issues in one paper but LogicPeer provide a means to program high-level peer network-wide queries.

Towards the Semantic Web, there are extensions to browsers with a Prolog engine to process RDF based metadata.[13] Such work can be applied to peer-to-peer knowledge-sharing for representing and sharing structured knowledge (and/or metadata) in domain-specialized peer groups. Ontologies can be used to match concepts expressed with different keywords in our DistributedYahoo application where we do not expect strong centralized control over the vocabulary used to describe the peers' concept hierarchies.

Apart from Prolog, there are other kinds of inference engines for the semantic Web such as the full first order logic based languages such as Larch (Garland et al., 1993), description logic based languages such as Loom (USC, 2004), TRIPLE (Sintek and Decker, 2002) a language and system for reasoning with RDF, and Inference Web (McGuinness and Pinheiroda Silva, 2003) for reasoning with OWL ontology descriptions and generating explanations for conclusions made. There has also been recent work on developing rule markup languages such as XRML (Lee and Sohn, 2003) to add rules to the Semantic Web in a standard format. Indeed, adding such rule-based behaviour to the Web is not a new idea (see, for example, Loke and Davison, 1998) though the use of rules for (effectively) publishing knowledge in a standard format has not yet been given adequate attention until more recently with XRML-like developments. When knowledge published in a standard format is translated into the appropriate form, an inference engine appropriate to the application needs selected from the repertoire mentioned above can be applied to reason over the knowledge. Prolog can be used to reason with such knowledge translated into Prolog form. This paper has extended Prolog with constructs to effectively extract knowledge from the Web and peer-to-peer systems, enabling applications that need to access different kinds of knowledge possibly stored using different paradigms. For example, one can use a referral peer network to find suitable Web sites or hyper-link to target URLs obtained by querying a peer-to-peer network. One could also query the semantic Web to extract rules embedded in Web pages which are then used to filter information obtained from a database, or utilize a referral network to judge the reliability of a Web service. Alternatively, a referral network can be used to obtain suggestions about how to handle a complex semantic Web query. Such mixed use of paradigms can be conveniently expressed in our language. For example, one can use a referral network to find suitable Web sites, and upon processing these Web sites, obtain URLs to Semantic Web ontologies, and upon processing these ontologies, discover new contacts, who are then queried via a peer-to-peer network. Results from the peer-to-peer network might be URLs to other Web sites. However, we do not claim Prolog as a panacea for all applications and Prolog might not be adequate for certain kinds of inferencing wherein some other inference formalism such as those mentioned above can then be applied. The adding of LogicPeer and LogicWeb operators to other such inference languages might then be considered.

Also interesting is work which utilizes the idea of semantic links between documents (or parts of documents) by Zhuge (2003). A similar idea was later applied to declare semantic links between peers in a peer-to-peer network in (Zhuge et al., 2005). Each link is declared explicitly with well-defined relationships such as sequence, equal to, similar to, subtype, cause and effect, and reference. Different from the semantic link approach which takes a meta-level God-view, our approach of establishing links is (i) Web-like in that one uses references (peer IDs or URLs) from within a document or peer to refer to other documents or peers, and (ii) programmatic in that such references are embedded within logic programs. While we see that the declarative semantic link approach can guide document searches and peer-to-peer search, and has clearly understood semantics (perhaps standardized across peers), we believe our approach also has its uses—in bringing the expressive power of a full programming language to reason about Web links and peer-to-peer relationships and complements the semantic links approach. Synergies can perhaps be considered between these two approaches, where given pre-defined semantic links between documents (or its parts) and between peers (in a peer-to-peer network), one can write applications as LogicPeer/LogicWeb programs which traverse and reason with these semantic link networks (similarly to how LogicWeb programs can be written which can follow and reason with existing links on Web pages).

7. Conclusions and future work

We have presented an approach towards integrative use of the Web and peer-to-peer systems within a declarative programming paradigm. We have also argued that logic programming can be useful in peer-to-peer computing for querying and representing knowledge shared over peer networks, and for scripting search behaviour over peer networks. We have illustrated our point using Prolog-based querying of knowledge-sharing peer networks.

[12] See http://p2p.ingce.unibo.it/cfp.html.

[13] See http://www.mozilla.org/rdf/doc/inference.html for examples.

Prolog has been used as a specification notation, as a rapid prototyping language, and for software process modelling (Ciancarini and Levi, 1995). The work here will extend the expressiveness of Prolog to specify and prototype programs that access peer-to-peer and Web based systems, and to model processes involving such distributed paradigms.

We can build LogicPeer over existing peer-to-peer protocols and toolkits. We have already explained how LogicPeer I can be built with a Gnutella-like protocol as the underlying protocol. It is also possible to interface the JXTA library with Prolog. We are currently investigating a multi-threaded Prolog language called Jinni[14] implemented in Java for this purpose, though Jinni does not allow backtracking over different nodes. Multi-threading will ease the implementation of the concurrency operators "[ ]" and "<>", and Jinni has mobile code capabilities where execution state (continuations) can be transferred from one node to another (as required for peer-switching).

We also note that controlling the evaluation of Prolog (and subsets thereof) programs has been investigated using meta-level techniques (e.g., Stolle et al., in press), and this is important for the Semantic Web where evaluation needs to be controlled due to constraints on user wait time and resources.

Further issues in our work are as follows: (1) Consistency among results obtained from different peers for the same goal, and consistency of results with real-world sense, are important issues in a logic-based approach. One way to resolve inconsistencies is to ask the user but this places a burden on the user and assumes that the user does know which results to use. Another possibility is to automate the combining of results with integrity constraints to detect inconsistencies. *Cooperative answers* (i.e., not just the answer set of bindings but also the reason for the answer) might be used. Global consistency is impossible to achieve in a distributed peer-to-peer system but local consistency between a peer's current knowledge base and incoming knowledge (and perhaps with several friends' knowledge bases) could be resolved. (2) Cached results, history of queries, and metadata descriptions at a peer can be used to speed-up queries. We can also send messages with routing instructions for peers. (3) We would like computations to proceed in a lazy evaluation style, that is, while information is still being fetched, processing can proceed. We hope to employ work from concurrent LogicWeb (Davison and Loke, 1999) and Webstream (Hong and Clark, 2003) for this purpose. Also, our operational semantics sketched here can be extended to the other LogicPeer operators. (4) While the transparent peer model enables a degree of control over how goals are to be evaluated over a peer network, dealing with the unpredictable behaviour of peers (e.g., programs not being executed at a peer) and the extra-functional properties of LogicPeer programs including time-to-live of programs are further avenues to be investigated. Some of these issues are already being addressed in existing peer-to-peer communities, though not in the context of a declarative programming language. We see our work as complementary to such work providing an integrative declarative querying front-end to the underlying peer network.

[14] http://www.binnetcorp.com/Jinni/index.html

Acknowledgement

The comments and contributions of the anonymous reviewers are gratefully acknowledged, one of whose coments motivated the cooperate-with operator.

References

Brogi, A., 1993. Program construction in computational logic. Ph.D. thesis, University of Pisa.

Ciancarini, P., Levi, G., 1995. Applications of Logic Programming in Software Engineering. In: PAP Workshop on LP and SE. Paris, France, available at <http://citeseer.nj.nec.com/ciancarini95-applications.html>.

Davison, A., 2002. Logic programming languages for the Internet. In: Kakas, A., Sadri, F. (Eds.), Computational Logic: from Logic Programming into the Future. Springer, pp. 66–104, Availabe from: <http://fivedots.coe.psu.ac.th/~ad/papers/summBob.ps.gz>.

Davison, A., Loke, S., April 1999. A concurrent logic programming model of the Web. In: PACLP'99: The 7th Int. Conf. and Exhibition on the Practical Application of Constraints and Logic Programming.

Garland, S., Guttag, J., Horning, J., July 1993. An overview of larch. In: Lauer, P. (Ed.), Functional Programming, Concurrency, Simulation and Automated Reasoning. pp. 329–348, available at <http://www.sds.lcs.mit.edu/~garland/publications/overview93.ps>.

Gong, L., 2001. Project JXTA: Technology Overview. Available from <http://www.jxta.org/project/www/docs/TechOverview.pdf>.

Hong, T., Clark, K., 2003. Agent Integrated Concurrent Web Programming. Available from <http://www.doc.ic.ac.uk/~twh1/academic/papers/software-pe.pdf>.

Hoschek, W., March 2002. A Unified Peer-to-Peer Database Framework for XQueries over Dynamic Distributed Content and its Application for Scalable Service Discovery. Ph.D. thesis.

Lee, J., Sohn, M., 2003. Extensible rule markup language—toward the intelligent Web platform. Communications of the ACM 46 (5), 59–64.

Loke, S., Davison, A., July 1997. A two-level world wide web model with logic programming links. In: De Bosschere, K., Hermenegildo, M., Tarau, P. (Eds.), Proceedings of the 2nd Workshop on Logic Programming Tools for Internet Applications (at the 14th ICLP). pp. 41–54, available at <http://clement.info.umoncton.ca/~lpnet/proceedings97/loke.ps>.

Loke, S., Davison, A., 1998. LogicWeb: enhancing the web with logic programming. Journal of Logic Programming 36 (3), 195–240, September.

McGuinness, D., Pinheiroda Silva, P., March 2003. Inference Web: portable and shareable explanations for question answering. In: Proceedings of the American Association for Artificial Intelligence Spring Symposium Workshop on New Directions for Question

Answering. pp. 67–71, available at <http://www.ksl.stanford.edu/people/pp/papers/McGuinness_SSS_2003.pdf>.

Nejdl, W., Wolf, B., Qu, C., Decker, S., Sintek, M., Naeve, A., Nilsson, M., Palmer, M., Risch, T., 2001. EDUTELLA: A P2P Networking Infrastructure Based on RDF (Whitepaper). Available at <http://edutella.jxta.org/reports/edutella-whitepaper.pdf>.

Ng, W., Ooi, B., Tan, K., Zhou, A., March 2003. PeerDB: A P2P-based System for Distributed Data Sharing. In: Proceedings of the 19th International Conference on Data Engineering 2003 (ICDE 2003). Available at <http://xena1.ddns.comp.nus.edu.sg/p2p/peerdb.pdf>.

Sintek, M., Decker, S., June 2002. triple—a query, inference, and transformation language for the semantic Web. In: Proceedings of the International Semantic Web Conference (ISWC). Available at <http://triple.semanticweb.org/iswc2002/TripleReport.pdf>.

Somogyi, Z., Henderson, F., Conway, T., February 1995. Mercury: an efficient purely declarative logic programming language. In: Proceedings of the Australian Computer Science Conference. pp. 499–512, available at <http://www.cs.mu.oz.au/research/mercury/information/papers/acsc95.ps.gz>.

Stolle, R., Hogan, A., Bradley, E., 2005. Agenda control for heterogeneous reasoners. The Journal of Logic and Algebraic Programming 62 (1), 41–69.

USC, 2004. Loom. Project Web site at <http://www.isi.edu/isd/LOOM/>.

Vivacqua, A., 1999. Agents for Expertise Location. In: Proceedings of the 1999 AAAI Spring Symposium on Intelligent Agents in Cyberspace. Available at <http://www.media.mit.edu/~adriana/projects/EF/EFssymp.html>.

Zhuge, H., 2003. Active e-document framework ADF: model and tool. Information and Management 41 (1), 87–97, Availabe from: <http://kg.ict.ac.cn/publication.htm>.

Zhuge, H., Liu, J., Feng, L., Sun, X., He, C., 2005. Query Routing in a Peer-to-Peer Semantic Link Network. Computational Intelligence 21 (2), 197–216, Availabe from: <http://www.blackwell-synergy.com/links/doi/10.1111/j.0824-7935.2005.00271.x/abs/>.

Dr. Seng Loke is currently a Faculty of Information Technology Research Fellow in Monash University, Melbourne, Australia. He received his PhD in 1998 for LogicWeb, a model integrating logic programming with the Web. He has published more than 120 articles in journals, books, conferences and workshops. His main interests are in models of computation and declarative programming in new distributed computing areas such as pervasive computing, peer-to-peer computing, ad hoc wireless environments, Semantic Web, service-oriented computing and context-awareness in pervasive computing.