A Logic Programming Agent for Controlling Concurrent Distributed Web Dialogues

Paulo Quaresma and Irene Pimenta Rodrigues
Departamento de Informática, Universidade de Évora, Portugal
{pq|ipr}@di.uevora.pt
tel: +351 21 2948536, fax: +351 21 2948541

September 17, 2000

Abstract

We present a Prolog based dialogue manager for a web information retrieval system that is able to cooperatively interact with the users helping them in their searches. The proposed framework is based on three specialised Prolog agents: an agent manager that receives all the user requests and it acts as an interface with the specific user process agent; the user process agent which is specific to each user and it has information about the user profile and the previous interrogation context; and an an agent monitor that informs the agent manager of the latest changes in the document databases allowing these changes to be transmitted to all users that have one of their previous queries results changed. This Prolog framework was implemented in a Linux environment using XSB Prolog. In the paper a detailed example in the law field is presented.

1 Introduction

We have designed a Prolog based dialogue manager for a Web Information Retrieval system. In order to perform a dialog with the user, we want:

- To infer what are the user intentions with the queries. When a user asks for documents with a particular keyword, usually he is interested in documents that may not have that keyword and he is not interested in all documents with that keyword.
- To supply pertinent answers or questions as a reply to a user question. The system must supply some information on the set of documents selected by the user query in order to help the user refining his query.

And we need:

- To record the previous user interactions with the system (user questions and the system answers).
- To obtain new partitions (labelled clusters) of the set of documents that the user selected with his query(ies).
- To use domain knowledge whenever the system has it.

In order to build the dialogue manager for a web information retrieval system we must take into account that:

- The number of users registered in the system is large (thousands).
- Typically there are more then one user using the system at the same instant.
- The users may interrupt their session for large periods of time.
- The users would like to be informed when a previous query result changes due to updates in the database.

These facts make impossible to use a process to deal with each user because the number of processes in an operative system is limited, and most of the processes will be blocked waiting for a user request.

The architecture that we designed to solve this problem has:

- An agent manager that receives all the user requests and keeps a database with all the users interrogation (dialogue) context.

  The agent manager is implemented in Prolog: it interfaces the web user requests; it verifies if the user is registered and if it has no pending requests; then it launches other Prolog agent, the agent process.
- Several process agents that given an user request and its interrogation context are able to answer the respective user and to actualise his interrogation context.

  The agent process when it is launched: consults the user database; answers the user; updates the user database (interrogation context); and informs the agent manager that it has finished.

- An agent monitor that informs the agent manager of the latest changes in the documents databases. These changes are transmitted to all users that have one of their previous queries results changed.

  This Prolog agent consults all the users database to check for differences in the user query results. When there are changes in a user, the agent adds a new request in the user database that will be handled by the agent process.

To model the knowledge the agent process represents four levels of knowledge using dynamic logic programming and the LUPS language [ALP+98, APP+99]. The knowledge levels are: Interaction, Domain, Information Retrieval and Text.

The Interaction Level is responsible for the dialogue management. This includes the ability of the system to infer user intentions and attitudes and the ability to represent the dialogue sentences in a dialogue structure. The dialogue structure will be kept in the user database. Note that dialogues in information retrieval systems [LJ94] are different from dialogues in computational linguistics [LA87, Pol90, CCC98, CL99, MP93, Loc98] because our user normally does not have a plan to execute, he actually does not have a precise goal such as: 'go to Boston' or 'phone to Mary'. Our users may want to see some documents, but they do not know which particular documents.

The Domain Level includes knowledge about the text domain and it has rules encoding that knowledge.

The Information Retrieval Level includes knowledge about documents and their relationship. The agent process has logic programming rules for computing labelled clusters that will be used as knowledge about the documents selected by the user.

The Text Level has knowledge about the words and sequence of words in each text of the knowledge base.

2 Dynamic LP and LUPS

Before describing the systems' architecture it is necessary to briefly present the dynamic logic programming paradigm and the language used to represent actions.

2.1 Dynamic Knowledge Representation

Given an *original*knowledge base KB, and a set of update rules represented by the *updating* knowledge base KB', it is possible to obtain a new *updated* knowledge base $KB^* = KB \oplus KB'$ that constitutes the *update of the knowledge base KB by the knowledge base KB'*. In order to make the meaning of the updated knowledge base $KB \oplus KB'$ declaratively clear and easily verifiable, in [ALP+98] there is a *complete semantic characterization* of the updated knowledge base $KB \oplus KB'$. It is defined by means of a simple, *linear-time* transformation of knowledge bases KB and KB' into a *normal logic program* written in a *meta-language.* As a result, not only the update transformation can be accomplished very efficiently, but also query answering in $KB \oplus KB'$ is reduced to query answering about *normal logic programs.*

2.2 Language for Dynamic Representation of Knowledge

Knowledge evolves from one *knowledge state* to another as a result of *knowledge updates.* Given the *current knowledge state KS*, its *successor knowledge state $KS' = KS[KB]$* is generated as a result of the occurrence of a non-empty set of simultaneous (parallel) *updates,* represented by the *updating* knowledge base KB. Consecutive knowledge states KS_n can be therefore represented as $KS_0[KB_1][KB_2]...[KB_n]$, where KS_0 is the default state and KB_i's represent consecutive *updating knowledge bases.*

Dynamic knowledge updates, as described above, did not provide any *language* for specifying (or programming) changes of knowledge states. Accordingly, in [APP+99] it was introduced a fully declarative, *high-level language for knowledge updates* called *LUPS* (*"Language of UPdateS"*) that describes transitions between consecutive knowledge states KS_n. It consists of *update commands,* which specify what updates should be applied to any given knowledge state KS_n in order to obtain the next knowledge state KS_{n+1}. In this way, update commands allow us to *implicitly* determine the *updating* knowledge base KB_{n+1}. The language LUPS can therefore be viewed as a *language for dynamic knowledge representation.* Below we provide a brief description of LUPS that does not include all of the available update commands and omits some details.

The simplest update command consists of adding a rule to the current knowledge state and has the form: $assert\ (L \leftarrow L_1, \ldots, L_k)$. For example, when a law stating that abortion is illegal is adopted, the knowledge state might be updated via the command: $assert\ (illegal \leftarrow abortion)$.

In general, the addition of a rule to a knowledge state may depend upon some preconditions being true in the current state. To allow for that, the assert command in LUPS has a more general form:

$$assert\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \quad (1)$$

The meaning of this assert command is that if the preconditions $L_{k+1}, \ldots, L_m$ are true in the current knowledge state, then the rule $L \leftarrow L_1, \ldots, L_k$ should hold true in the successor knowledge state. Normally, the so added rules are *inertial,* i.e., they remain in force from then on by inertia, until possibly defeated by some future update or until retracted.

However, in some cases the persistence of rules by inertia should not be assumed. Take, for instance, the simple fact *alarm_ring*. This is likely to be a *one-time* *event* that should not persist by inertia after the successor state. Accordingly, the assert command allows for the keyword *event*, indicating that the added *rule* is *non-inertial.*

$$assert\ event\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \tag{2}$$

Update commands themselves (rather than the rules they assert) may either be one-time, non-persistent update commands or they may remain in force until cancelled. In order to specify such *persistent update commands* (which we call *update laws*) there is the syntax:

$$always\ [event]\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \tag{3}$$

To cancel persistent update commands, we use:

$$cancel\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \tag{4}$$

To deal with rule deletion, we employ the *retraction* update command:

$$retract\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \tag{5}$$

meaning that, subject to precondition $L_{k+1}, \ldots, L_m$, the rule $L \leftarrow L_1, \ldots, L_k$ is retracted. Note that cancellation of a persistent update command is very different from retraction of a rule. Cancelling a persistent update means that the given update command will no longer continue to be applied, but it does not remove any inertial effects of the rules possibly asserted by its previous application(s).

3 The system architecture

The architecture is based on three specialised Prolog agents:

- An agent manager that receives all the user requests and it acts as an interface with the specific user process agent;
- The user process agent which is specific to each user and it has information about the user profile and the previous interrogation context;
- An agent monitor that informs the agent manager of the latest changes in the documents databases allowing these changes to be transmitted to all users that have one of their previous queries results changed.

As it can be seen in figure 1, each user communicates with the agent manager, which redirects the event to the specific user process agent (launching the user process, if needed). In order to obtain a cooperative answer, each user process agent accesses the respective user interaction context (composed by logic programming facts) and the databases. Afterwards, it updates

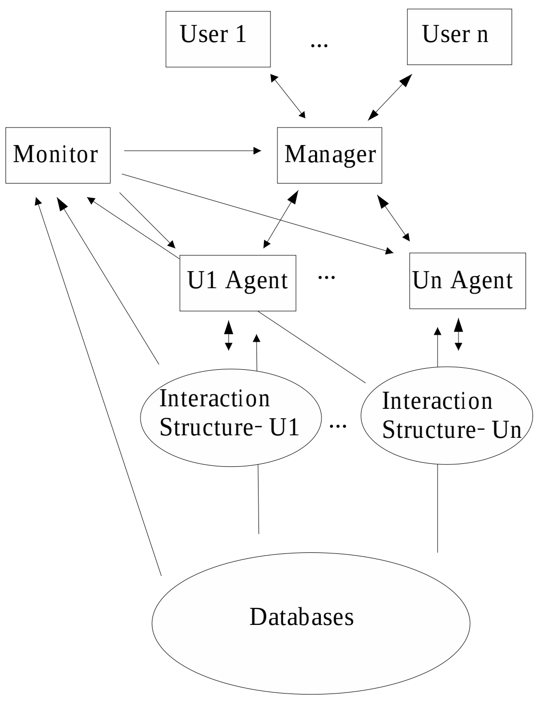

Figure 1: General architecture

the interaction structure and communicates the answer to the agent manager, which redirects it to the user.

On the other hand, the monitor agent is always accessing the databases and the interaction structures trying to detect changes in the given answers. Then, it informs the agent manager, which will inform the change to the user process agent.

This architecture was implemented in a Linux environment using XSB Prolog.

4 The agent manager

The agent manager receives the user requests, analyses them and redirects them to the respective user process agent.

In this process there is a technical problem related with the fact that it is not possible to have all the user process agents running at the same time (we expect thousands of users). Our solution was to have only alive the user agents correspondent to the active users. So, one of the agent manager tasks is to keep track of the active users and to launch the respective user process agent, if needed.

This behaviour is described by the following logic programming rules:

always inform(Manager,USER_AGENT_ID,A) *when* request(USER_ID,Manager,A), alive(USER_AGENT_ID)

always alive(USER_AGENT_ID) ← launch(USER_AGENT_ID)

As described, these rules state that after a request from the user ID, the user process agent is launched (if needed) and the user action is redirected to it. Note that *launch* is an abductive action, which will be abduced and executed only if needed, i.e. if its value is currently false or unknown.

The user event is analysed by the user process agent (see next section) and an answer is obtained. The answer is dynamically transformed in a HTML page which is sent to the correct user. This behaviour is represented by the following rule:

always inform(Manager,USER_ID,HTML) *when* inform(USER_AGENT_ID,Manager,A), tr(A,HTML)

Another task of the agent manager is to receive information from the monitor agent about database changes related to user queries (see section 6). The received information is redirected to the user process agent and it can be used afterwards to inform the user of the database changes. The correspondent dynamic logic programming rule is:

always inform(Manager,USER_AGENT_ID,P) *when* inform(MONITOR,Manager,P)

5 The user process agent

This agent is specific to each user and it processes the user requests. It is launched by the agent manager to treat a user request. This agent receives the user request and in order to process it, he must consult the user database to obtain the interpretation context for the user request.

When this agent terminates his service, the fulfilment of a user request, he updates the user database with the interpretation of the user request and the agent answer; he informs the manager agent that he has finished his work and he dies.

In order to fulfil a user request the agent must:

- Load the user interaction context. The interaction context is a structure where all the previous user-system interactions are kept.
- Interpret the speech act in the interaction context. The agent must infer the user intentions and beliefs from its speech act (multimodal) [QR99a].
- Perform a set actions that are inferred in order to fulfil the user intentions.
- Save the new interaction context for this user.
- Inform the manager agent that he has finished.

Interaction example Suppose that the user his a retired fireman that has saved a child from drowning in a river during his holidays putting his life at risk. Since he has caught a pneumonia as a consequence, he wants to know if he can get a pension for relevant services to his country.

[User - Q1:] Documents about pensions for relevant services?

[System - Q2:] There are 243 documents, listed in the right frame. For further refinement you may choose one (or none) option in each column:

Pensions that were:	*Pensions for:*
given	militaries
or refused	or civilians

Documents with keyword:
drowning
or medal
or in duty
or ...

Options in column 1 and 2 are obtained using rules that model legal knowledge on legislation on pensions for relevant services[QR99b]. Options on column 3 are obtained by clustering the set of 243 documents [LQR99].

[User - Q3:] chooses: civilians and drowning.

[System - Q4:] There are 45 documents, listed in the right frame. For further refinement you may choose one (or none) option in each column:

Documents with keyword:
death
or accident
or ...

Options in this column are obtain by reclustering the 45 documents.

[User - Q5:] Documents where there was a drowning in a river.

[System - Q6:] Since there are 13 documents of the 45 above that match the concept, I've listed them in the usual frame. For further refinement you may choose one (or none) option in each column:

Documents with keyword:
car crash
or bridge
or ...

[User - Q7:] Where there was someone swimming?

[System - Q8:] Since there are no documents of the 13 above that match your new constraint. I assume that you want that your query refines Q3, i.e. you are looking for documents with the concept swimming, and drowning and pensions for relevant services for civilians. Do you want to see these documents?

[User - Q9:] Yes. .....

5.1 The user requests

This agent is expecting one of the following requests from the user:

- A request that results from a multimodal act, such as:
  - A request to return to a previous point of the interaction:
    $request(USER\_ID, new\_context(Val))$
    The system users have access to a representation of the interaction context, a tree with labels representing the requests. In order to generate this multimodal act the user clicks on the tree node representing the previous request. For instance, to return to point of request Q5, the user should click on the box labelled Q5-Q6 in figure 1; this click will generate the request:
    $request(user1, new\_context(\#Q5))$

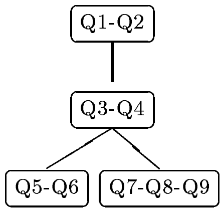

Figure 1: Dialogue Structure for Dialogue 1

  - A request to refine the previous selected set of documents with an expression (build with keywords connected by *and*s and *or*s):
    $request(USER\_ID, System, refine(Expr))$
    In the example, Q3 gives rise to this kind of request:
    $request(\text{user}, s, refine(civilians\ and\ drowning))$

- A natural language expression in the form of a speech act:

  1. $request(USER\_ID, System,$
     $inform(sys, USER\_ID, DRS))$
  2. $request(USER\_ID, system,$
     $inform\_ref(sys, USER\_ID, REF, DRS))$
     In the example, Q1 gives rise to this kind of request:
     $request(old\_fireman, s,$
     $inform\_ref(s, old\_fireman, x,$
     $[x, y{:}documents(x), concept(y), y{=}pension,$
     $about(x, y)]).$
  3. $request(inform(USER\_ID, system, no))$
  4. $request(inform(USER\_ID, system, ok))$
     In the example, Q9 gives rise to this kind of request.
     $request(inform(old\_fireman, system, ok))$

- A request from the agent monitor in order to inform the user that one of his previous requests has a different result due to changes in the documents database.

  $request(monitor, system,$
  $inform(system, User\_id, changes(Request))$

This set of requests will enable us to encode all the user interactions in our application.

5.2 The user interaction context

The user interaction context is kept in a dialogue structure. This structure records both user and system questions and answers. The structure is used to compute the meaning of a user query and to allow the user to return to a previous point of the dialogue and to build a new branch from there.

The Dialogue structure is made of segments that group sets of sentences (user and system sentences). The dialogue structure reflects the user intentions, it is built by taking into account the user and system intentions. The dialogue segments have precise inheritance rules defining how segments heritage their attributes from the attributes of their sentences[QR99a].

The dialogue structure also enables the system to recognise and to generate pertinent discourse phenomena such as anaphoric references.

In order to interpret a user request this agent must insert the new request into the user dialogue structure and as result of this process a new structure is built.

The following LUPS rule[APP$^+$99] controls the insertion of the user request in the dialogue structure:

always ds(DS_0) $\leftarrow$ update(U,DS_{-1},S,DS_0)
when request(U,A,S), ds(DS_{-1}).

This rule means that, when there is a request S and a discourse structure DS_{-1}, the new structure is obtained from the update of the old structure with the new request.

To fulfil the user request the agent will generate an action A, that is added to the old dialogue structure giving rise to a new one:

always ds(DS_1) $\leftarrow$ update(Ag,DS_0,A,DS_1)
when action(Ag,A), ds(DS_0).

DS_1 will be the structure to be used for the interpretation of the next request by this user.

5.3 The inference of user attitudes

In order to be collaborative our system needs to model user attitudes (intentions and beliefs). This task is also achieved through the use of logic programming framework rules and the dynamic LP semantics[QR99c].

The system mental state is represented by an extended logic program that can be decomposed in several modules (see [QL95] for a more complete description of these modules):

— Description of the effects and the pre-conditions of the speech acts in terms of beliefs and intentions;

— Definition of behaviour rules that define how the attitudes are related and how they are transferred between the users and the system (cooperatively).

For instance, the rules which describe the effect of an inform and a request speech act from the point of view of the receptor (assuming cooperative agents) are:

> *always* bel(A,bel(B,P)) *when* inform(B,A,P)
>
> *always* bel(A,int(B,Action)) *when* request(B,A,Action)

In order to represent collaborative behaviour it is necessary to model how information is transferred from the different agents:

> *always* bel(A,P) *when* bel(A,bel(B,P))
>
> *always* int(A,Action) *when* bel(A,int(B,Action))

These two rules allow beliefs and intentions to be transferred between agents if they are not inconsistent with their previous mental state.

After each event (for instance a user question) the agents' model (logic program) needs to be updated with the description of the event that occurred. The dialogue system recognises the speech act and it constructs the associated speech act (request or inform). The speech act will be used to update the logic program in order to obtain a new model. Using this new model it is possible to obtain the intentions of the system.

5.4 The agent actions

The agent actions are inferred in order to satisfy the user request after its interpretation.

The agent actions are commands for the interaction system, they are composed of the following items:

- A set of documents that match the user query, a list of document numbers.

  The set of documents is obtained by sending the information retrieval search engine, SINO, the command "sino > search Q", with Q being the semantic interpretation of the user request transformed into a sino query[QR99b].

  Our information retrieval system is based on SINO, a text search engine from the AustLII Institute [GMK97] SINO is a word based text search engine that allows boolean and free text queries.

- A set of suggestions for further refinement of the user current query, i.e.a list of lists with keywords to be displayed.

  This suggestions are obtained by:

  - building labelled clusters of documents.
  - using domain knowledge, the update of the semantic/pragmatic interpretation of the user request enables the computation of new models. The domain knowledge models will supply a set of concepts that can be used as suggestions for the user query refinement.

- The new dialogue structure.

The inference of the agent actions is launched by the following rules:

> *always* part1(Q,L) *when* ds(D), translate(D,Q), sino(Q,L).
>
> *always* part2(Q,Sets) *when* part1(Q,L) clusters(L,Sets).
>
> *always* part3(Sem,M) *when* ds(D), obtain(D,Sem), models(Sem,M).
>
> *always* int(A,inform(A,User,[L,Sets,M]) *when* part1(Q,L), part2(Q,Sets), part3(S,M).

These rules state that the new intended action is composed by the information obtained from the information retrieval module, the clustering module, and the knowledge representation module.

5.5 The agent top goal

The agent top goal is to receive a request and to act cooperatively. Then, it saves its states and launches the goal *terminates*.

> *always* action(A,Action) *when* request(U,A,R), ds(D), int(A,Action)
>
> *always* terminates *when* action(Ag,A), ds(D), save(D)

6 The agent monitor

The agent monitor informs the agent manager of the users that must be informed about the latest changes in the documents databases.

This agent runs after any change in the documents databases, and he searches the users that may have the results of their previous sessions changed by the update.

The agent monitor must consult all user database to check for differences in each user query result. When there are changes in a user, the agent adds a new request in the user database that will be handled by the agent process.

A user may define in his profile if wants to be informed when there are relevant changes in the documents database.

When the agent monitor is launched:

1. He builds a list with all the users that want to be warned whenever there are relevant changes in the documents database.
2. For each user in this list he opens the user database where the interaction context is kept; and consults the dialogue structure.
3. Then, for each user request he checks if the new interpretation, after the changes, give the same results.

   This is done by launching the agent process requests with the request to be checked, after replacing the user database by a new one with a dialogue structure that only has the previous user requests.

   In order to check if the request results are the same, the agent monitor must compare the new agent action with the old one (just the list of documents selected by the query).
4. Whenever there are changes in the results of a user request interpretation this agent sends a message to the agent manager with the request:

   request(monitor,system, inform(system, User_id, changes(Request))

   The agent manager is responsible for sending this request into the agent process requests whenever the user has a new request.

This agent communicates with the other two agents: the agent manager receives its messages, and the agent process requests are launched by it in order to test for different results.

The performance of the monitor agent can be improved by parallelising the treatment of each user.

7 Conclusions

We have presented a logic programming based architecture for controlling cooperative multiuser web information retrieval systems. This architecture is based on three king of agents: the agent manager, which interacts with the user and with the specialised user process agents; the user process agents, which have the knowledge specific to each user and interacts cooperatively with them; and the monitor agent, which tries to detect changes in the databases and to inform the users about them.

The architecture was implemented using XSB Prolog over a legal information retrieval system.

The initial evaluation results show that our cooperative system is able to help the users decreasing the number of queries needed to obtain the desired documents (around 20%). Although these preliminary results seem to be positive we haven't finished a complete evaluation of the results (we expect to have a more formal evaluation by the end of the year 2000).

References

[ALP$^+$98] J. J. Alferes, J. Leite, L. M. Pereira, H. Przymusinska, and T. Przymuzinski. Dynamic logic programming. In *Proc. of KR'98*, 1998.

[APP$^+$99] J. J. Alferes, L. M. Pereira, H. Przymusinska, T. C. Przymusinski, and P. Quaresma. Preliminary exploration on actions as updates. In M. C. Meo and M. Vilares-Ferro, editors, *Procs. of the 1999 Joint Conference on Declarative Programming (AGP'99)*, pages 259–271, L'Aquila, Italy, September 1999.

[CCC98] J. Chu-Carroll and S. Carberry. Response generation in planning dialogues. *Computational Linguistics*, 24(3), 1998.

[CL99] Sandra Carberry and Lynn Lambert. A process model for recognizing communicative acts and modeling negotiation subdialogs. *Computational Linguistics*, 25(1), 1999.

[GMK97] G. Greenleaf, A. Mowbray, and G. King. Law on the net via austlii - 14 m hypertext links can't be right? In *In Information Online and On Disk'97 Conference, Sydney*, 1997.

[LA87] Diane Litman and James Allen. A plan recognition model for subdialogues in conversation. *Cognitive Science*, 11(1), 1987.

[LJ94] Brian Logan and Karen Sparck Jones. Belif revision and dialogue management in information retrieva. Technical report, University of Cambridge, Computer Laboratory, 1994.

[Loc98] Karen E. Lochbaum. A collaborative planning model of intentional structure. *Computational Linguistics*, 24(4), 1998.

[LQR99] José Gabriel Lopes, Paulo Quaresma, and Irene Pimenta Rodrigues. *A Dialog System for Controlling Question/Answer Dialogues*, volume 2, pages 75–86. R. Potapova, Moscow, Russia, 1999.

[MP93] Johanna Moore and Cecile Paris. Planning text for advisory dialogues:capturing intentional and rhetorical information. *Computational Linguistics*, 19(4), 1993.

[Pol90] Martha Pollack. Plans as complex mental attitudes. In Philip Cohen, Jerry Morgan, and Martha Pollack, editors, *Intentions in Communications*. MIT Press Cambridge, 1990.

[QL95] P. Quaresma and J. G. Lopes. Unified logic programming approach to the abduction of plans and intentions in information-seeking dialogues. *Journal of Logic Programming*, 54, 1995.

[QR99a] P. Quaresma and I. Rodrigues. An information retrieval system with cooperative behavior. In *NODALIDA'99 – Nordic Conference of Computational Linguistics*, Trondheim, Norway, 1999.

[QR99b] P. Quaresma and I. Rodrigues. Pgr: A cooperative legal ir system on the web. In Graham Greenleaf and Andrew Mowbray, editors, *2nd AustLII Conference on Law and Internet*, Sydney, Australia, 1999. Invited paper.

[QR99c] P. Quaresma and I. Rodrigues. Using dynamic logic programming to model cooperative dialogues. In *AAAI'99 Fall Symposium on Modal and Temporal Logics based Planning for Open Networked Multimedia Systems*, Cape Cod, USA, 1999. To be published by IOS Press.