Explicit Dynamic User Profiles for a Collaborative Filtering Recommender System

M. Ilic, J. Leite, and M. Slota*

CENTRIA, Universidade Nova de Lisboa, Portugal

Abstract. User modelling and personalisation are the key aspects of recommender systems in terms of recommendation quality. While being very efficient and designed to work with huge amounts of data, present recommender systems often lack the facility of user integration when it comes to feedback and direct user modelling. In this paper we describe ERASP, an add-on to existing recommender systems which uses dynamic logic programming – an extension of answer set programming – as a means for users to specify and update their models, with the purpose of enhancing recommendations. We present promising experimental results.

1 Introduction

Nowadays, almost every e-commerce application provides a recommender system to suggest products or information that the user might want or need [19].

Common techniques for selecting the right item for recommendation are: collaborative filtering (e.g. [15]) where user ratings for objects are used to perform an inter-user comparison and then propose the best rated items; content-based recommendation (e.g.[8]) where descriptions of the content of items are matched against user profiles, employing techniques from the information retrieval field; knowledge-based recommendation (e.g. [9]) where knowledge about the user, the objects, and some distance measures between them are used to infer the right selections; and hybrid versions of these where two or more techniques (collaborative filtering being usually one of them) are used to overcome their individual limitations. For further details on this subject the reader is referred to [10].

The extent to which users find the recommendations satisfactory is the key feature of a recommendation system, and the accuracy of the user models that are employed is of significant importance to this goal. Such user models represent the user's taste and can be implicit (e.g. constructed from information about the user behavior), or explicit (e.g. constructed from direct feedback or input by the user, like ratings). The accuracy of a user model greatly depends on how well short-term and long-term interests are represented [7], making it a challenging task to include both sensibility to changes of taste and maintenance of permanent preferences. While implicit user modelling disburdens the user of providing direct feedback, explicit user modelling may be more confidence inspiring to the user since recommendations are based on a conscious assignment of preferences.

* Partially supported by FCT Scholarship SFRH/BD/38214/2007

Though most recommender systems are very efficient from a large-scale perspective, the effort in user involvement and interaction is calling for more attention. Moreover, problems concerning trust and security could be approached with a better integration of the user and more control over the user model [16].

This calls for more expressive ways for users to express their wishes. The natural way to approach this is through the use of symbolic knowledge representation languages. They provide the necessary tools for representing and reasoning about users, while providing formal semantics that make it possible to reason about the system, thus facilitating trust and security management.

However, we want to keep the advantages of the more automated recommendation techniques such as collaborative filtering and statistical analysis, and benefit from using large amounts of data collected over the years by existing systems that use these techniques. Unfortunately, the use of these methods makes it impossible to have the explicit user models we seek. A tight combination between expressive (logic based) knowledge representation languages and subsymbolic/statistical approaches is still the Holy Grail of Artificial Intelligence.

One solution to tackle this problem is through the use of a layered architecture, as suggested in [17], where expressive knowledge based user models, specified in *Dynamic Logic Programming* (DLP) [2, 18, 3], are used to enhance the recommendations provided by existing recommender systems.

In a nutshell, DLP is an extension of Answer-set Programming (ASP) [14] introduced to deal with knowledge updates. ASP is a form of declarative programming that is similar in syntax to traditional logic programming and close in semantics to non-monotonic logic, that is particularly suited for knowledge representation. Enormous progress concerning the theoretical foundations of ASP (c.f. [6] for more) have been made in recent years, and the existence of very efficient ASP solvers (e.g. DLV and SMODELS) make it possible to investigate some serious applications. Whereas in ASP knowledge is specified in a single theory, in DLP knowledge is given by a sequence of theories, each representing an update to the previous ones. The declarative semantics of DLP ensures that any contradictions that arise due to the updates are properly handled. Intuitively, one can add newer rules to the end of the sequence and DLP automatically ensures that these rules are in force and that the older rules are kept for as long as they are not in conflict with the newly added ones (c.f. [18] for more).

In this paper we describe the architecture, implementation and preliminary performance results of ERASP (Enhancing Recommendations with Answer-Set Programming), the system that resulted from following this path. Specifically, ERASP takes the output of an existing recommender algorithm (in this paper we used collaborative filtering, but it could be another) and enhances it taking into account explicit models and preferences specified both by the user and the owner of the system, represented in DLP. The main features of ERASP include:

– Providing the owner and user with a simple but expressive and extensible language to specify their models and preferences, by means of rules and employing existing (e.g. product characteristics) as well as user defined (e.g. own qualitative classifications based on product characteristics) concepts.

– Facilitating the update of user models by automatically detecting and solving contradictions that arise due to the evolution of the user's tastes and needs, which otherwise would discourage system usage.

– Taking advantage of existing recommender systems which may encode large amounts of data that should not be disregarded, particularly useful in the absence of user specified knowledge, while giving precedence to user specifications which, if violated, would turn the user away from the recommendation system.

– Enjoying a well defined semantics which allows the formal study of properties and provides support for explanations, improving interaction with the user.

– Having a connection with relational databases (ASP can be seen as a query language, more expressive than SQL), easing integration with existing systems.

– Allowing the use of both strong and default negation to reason with closed and open world assumptions, thus allowing to reason with incomplete information, and to encode non-deterministic choice, thus generating more than one set of recommendations, facilitating diversity each time the system is invoked;

The remainder of this paper is organised as follows: In Sect. 2, for self containment, we recap the notion of Dynamic Logic Programming, establishing the language used throughout. In Sect. 3 we present ERASP architecture, semantics and implementation. In Sect. 4 we present a short illustrative example and some experimental results. In Sect. 5 we discuss the results and conclude.

2 Dynamic Logic Programming

Let $\mathcal{A}$ be a set of propositional atoms. An **objective literal** is either an atom A or a strongly negated atom $\neg A$. A **default literal** is an objective literal preceded by *not*. A **literal** is either an objective literal or a default literal. A **rule** r is an ordered pair $H(r) \leftarrow B(r)$ where $H(r)$ (dubbed the head of the rule) is a literal and $B(r)$ (dubbed the body of the rule) is a finite set of literals. A rule with $H(r) = L_0$ and $B(r) = \{L_1, \ldots, L_n\}$ will simply be written as $L_0 \leftarrow L_1, \ldots, L_n$. A **generalised logic program** (GLP) P, in $\mathcal{A}$, is a finite or infinite set of rules. If $H(r) = \neg A$ (resp. $H(r) = not\, A$), then $\neg H(r) = A$ (resp. $not\, H(r) = A$). By the **expanded generalised logic program** corresponding to the GLP P, denoted by $\mathbf{P}$, we mean the GLP obtained by augmenting P with a rule of the form $not\, \neg H(r) \leftarrow B(r)$ for every rule, in P, of the form $H(r) \leftarrow B(r)$, where $H(r)$ is an objective literal. An **interpretation** M of $\mathcal{A}$ is a set of objective literals that is consistent, i.e. M does not contain both A and $\neg A$. An objective literal L is true in M, denoted by $M \models L$, iff $L \in M$, and false otherwise. A default literal $not\, L$ is true in M, denoted by $M \models not\, L$, iff $L \notin M$, and false otherwise. A set of literals B is true in M, denoted by $M \models B$, iff each literal in B is true in M. An interpretation M of $\mathcal{A}$ is an **answer set** of a GLP P iff $M' = least(\mathbf{P} \cup \{not\, A \mid A \notin M\})$, where $M' = M \cup \{not\_A \mid A \notin M\}$, A is an objective literal, and $least(.)$ denotes the least model of the program obtained from the argument program by replacing every $not\, A$ by not_A.

A **dynamic logic program** (DLP) is a sequence of generalised logic programs. Let $\mathcal{P} = (P_1, \ldots, P_n)$ be a DLP and P, P' be GLPs. We use $\rho(\mathcal{P})$ to denote

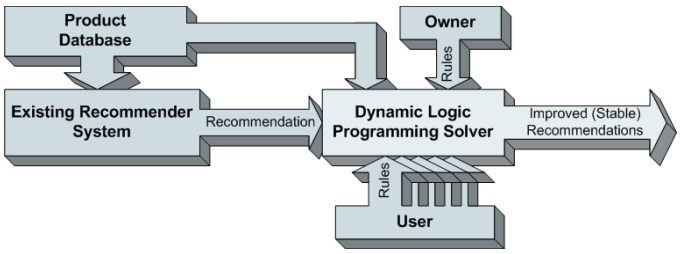

Fig. 1. ERASP System Architecture

the multiset of all rules appearing in the programs $\mathbf{P}_1, \ldots, \mathbf{P}_n$ and $(P, P', \mathcal{P})$ to denote the DLP $(P, P', P_1, \ldots, P_n)$. We can now set forth the definition of a semantics, *based on causal rejection of rules*, for DLPs. We start by defining the notion of conflicting rules as follows: two rules r and r' are **conflicting**, denoted by $r \bowtie r'$, iff $H(r) = not\, H(r')$. Let $\mathcal{P} = (P_1, \ldots, P_n)$ be a DLP and M and interpretation. M is a **(refined) dynamic stable model** of $\mathcal{P}$ iff

$$M' = least([\rho(\mathcal{P}) \setminus Rej(\mathcal{P}, M)] \cup Def(\mathcal{P}, M)) \text{ where:}$$
$$Rej(\mathcal{P}, M) = \{r \mid r \in \mathbf{P}_i, \exists r' \in \mathbf{P}_j, i \leq j, r \bowtie r', M \models B(r')\}$$
$$Def(\mathcal{P}, M) = \{not\, A \mid \nexists r \in \rho(\mathcal{P}), H(r) = A, M \models B(r)\}$$

3 Framework and its Implementation

In this Section, we introduce the architecture, its semantics and describe the implementation. ERASP's goal is to take the strengths of DLP as a framework for the representation of evolving knowledge, and put it at the service of both the user and owner of a recommender system, while at the same time ensuring some degree of integration with other recommendation modules, possibly based on distinct paradigms (e.g. statistical).

Fig. 1 depicts the system architecture, representing the information flow. To facilitate presentation, we assume a layered system where the output of an existing recommendation module is simply used as input to our system. We are aware that allowing for feedback from our system to the existing module could benefit its output, but such process would greatly depend on the particular module and we want to keep our proposal as general as possible, and concentrate on other aspects of the framework. The output of the initial module is assumed to be an interpretation, i.e. a consistent set of atoms representing the recommendations. We assume that our language contains a reserved predicate of the form $rec/1$ where the items are the terms of the predicate[1]. The owner policy, possibly used to encode desired marketing strategies (e.g. introduce some bias towards some

[1] It would be straightforward to also have some value associated with each recommendation, e.g. by using a predicate of the form $rec(item, value)$. However, to get our point across, we will keep to the simplified version.

products), is encoded as a generalised logic program. The user model (including its updates) is encoded as a dynamic logic program. The Product Database is a relational database that can easily be represented by a set of facts in a logic program. For simplicity, we assume such database to be part of the generalised logic program representing the owner's policy. A formalization of the system is given by the concept of Dynamic Recommender Frame:

Definition 1 (Dynamic Recommender Frame). *Let $\mathcal{A}$ be a set of propositional atoms. A Dynamic Recommender Frame (DRF), over $\mathcal{A}$, is a triple $\langle M, P_0, \mathcal{P} \rangle$ where M is an interpretation of $\mathcal{A}$, P_0 a generalised logic program over $\mathcal{A}$, and $\mathcal{P}$ a DLP over $\mathcal{A}$.*

The semantics of a Dynamic Recommender Frame is given by the set of dynamic stable models of its transformation into a DLP. This transformation is based on two natural principles: – the user's specification should prevail over both the initial recommendations and the owner's rules, since users would not accept a recommendation system that explicitly violates their rules; – the owner should be able to override the recommendations in the initial interpretation, e.g. to specify preference among products according to the profit. Intuitively, we construct a DLP with the initial program obtained from the initial recommendations, which is then updated with the owner's policy specification (P_0) and the user's specification ($\mathcal{P}$). A formal definition follows:

Definition 2 (Stable Recommendation Semantics). *Let $\mathcal{R} = \langle M, P_0, \mathcal{P} \rangle$ be a DRF and M_R an interpretation. M_R is a stable recommendation iff M_R is a dynamic stable model of $(P_M, P_0, \mathcal{P})$ where $P_M = \{A \leftarrow | \ A \in M\}$.*

According to this semantics, a Dynamic Recommender Frame can have more than one Stable Recommendation, each corresponding to one particular set of products that could be recommended to the user. This immediately represents a nice feature since it allows for the system to present the user with a different set of recommendations each time the user invokes the system adding diversity.

ERASP is implemented as an online application[2] using a PHP-based initial collaborative filtering recommender system. The product database consists of the complete MovieLens (http://www.grouplens.org/) dataset (3883 movies with title, genre and year). After rating some movies and receiving some initial recommendations (using the collaborative filtering algorithm), the user can edit his preferences encoded as a dynamic logic program using an interface which provides some help in rule creation. This program $\mathcal{P}$, the initial recommendations M, the product database and the owner specifications P_0 are given to a *SMODELS*-based solver. The solver computes the corresponding DLP $\mathcal{Q} = (P_M, P_0, \mathcal{P})$ and, using *Lparse*, produces an equivalent DLP $\mathcal{Q}_G$ without variables.

After the grounded dynamic logic program $\mathcal{Q}_G$ is parsed, it gets transformed into an equivalent normal logic program $\mathcal{Q}_G^R$. This transformation is described in detail in [5]. The stable models of $\mathcal{Q}_G^R$ directly correspond to the dynamic stable models of $\mathcal{Q}$. The transformed program is then passed to *Lparse* and

[2] Available at http://centria.di.fct.unl.pt/erasp/

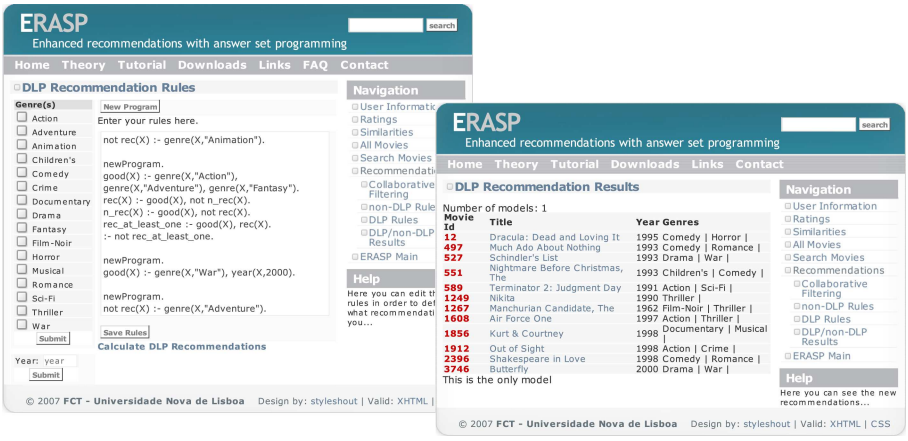

Fig. 2. ERASP Screenshots: a) User Model Interface b) Recommendations

Smodels in order to compute its stable models. Our program then writes the recommendations into an SQL database and presents them to the user. Fig. 2 presents two screen-shots of ERASP.

4 Illustrative Example and Experimental Results

In this Section, we show an example that illustrates some features of our proposal, and present the results of benchmark tests based on the example.

Let's consider a typical on-line movie recommender. Its product database contains information about a number of movies and its recommendations are based on some kind of statistical analysis performed over the years. The owner of the recommender system may want to explicitly influence the recommendations of the system in a certain way. She may also want to give the users the ability to specify some explicit information about their tastes in order to correct or refine the recommendations of the existing system. Below we will illustrate how our framework can help achieve both these goals in a simple way. A list of the movies involved in the example together with their relevant properties can be found in Table 1. We will also consider the initial interpretation M obtained from the statistical system to be constant throughout the example:

$$M = \{rec(497), rec(527), rec(551), rec(589), rec(1249), \\ rec(1267), rec(1580), rec(1608), rec(1912), rec(2396)\}$$

Let's consider the following owner specification:

$$P_0 : rec(12) \leftarrow not\ rec(15). \quad (1)$$
$$rec(15) \leftarrow not\ rec(12). \quad (2)$$
$$rec(X) \leftarrow rec(Y), year(Y, 1998), genre(Y, \text{“}Romance\text{”}), \\ genre(X, \text{“}Musical\text{”}), year(X, 1998). \quad (3)$$

ID	Title	Year	Genres
12	Dracula: Dead and Loving It	1995	Comedy, Horror
15	Cutthroat Island	1995	Action, Adventure
497	Much Ado About Nothing	1993	Comedy, Romance
527	Schindler's List	1993	Drama, War
551	Nightmare Before Christmas, The	1993	Children's, Comedy
558	Pagemaster, The	1994	Action, Adventure, Fantasy
589	Terminator 2: Judgment Day	1991	Action, Sci-Fi
1249	Nikita	1990	Thriller
1267	Manchurian Candidate, The	1962	Film-Noir, Thriller
1580	Men in Black	1997	Action, Adventure
1608	Air Force One	1997	Action, Thriller
1856	Kurt & Courtney	1998	Documentary, Musical
1912	Out of Sight	1998	Action, Crime
2394	Prince of Egypt, The	1998	Animation, Musical
2396	Shakespeare in Love	1998	Comedy, Romance
3746	Butterfly	2000	Drama, War

Table 1. Movies used in the example with some of their properties

Rules (1) and (2) specify that the system should non-deterministically recommend either movie 12 or 15[3]. Rule (3) encodes that the system should recommend all movies with the genre *Musical* from 1998 if any movie with the genre *Romance* from the same year is recommended. Adding an empty set of user specifications $\mathcal{P}_0 = ()$, the recommender frame $\langle M, P_0, \mathcal{P}_0 \rangle$ has two stable recommendations: $M_{R1} = M \cup \{rec(12), rec(1856), rec(2394)\}$ and $M_{R2} = M \cup \{rec(15), rec(1856), rec(2394)\}$. The reader can easily check that each of these two stable recommendations extend the results from the initial recommendation to reflect the wishes of the owner. We now turn to the user specifications:

$$P_1 : not\, rec(X) \leftarrow genre(X, \text{“}Animation\text{”}). \tag{4}$$
$$P_2 : good(X) \leftarrow genre(X, \text{“}Action\text{”}), genre(X, \text{“}Adventure\text{”}), genre(X, \text{“}Fantasy\text{”}). \tag{5}$$
$$rec(X) \leftarrow good(X), not\, n\_rec(X). \tag{6}$$
$$n\_rec(X) \leftarrow good(X), not\, rec(X). \tag{7}$$
$$rec\_at\_least\_one \leftarrow good(X), rec(X). \tag{8}$$
$$\leftarrow not\, rec\_at\_least\_one. \tag{9}$$
$$P_3 : good(X) \leftarrow genre(X, \text{“}War\text{”}), year(X, 2000). \tag{10}$$
$$P_4 : not\, rec(X) \leftarrow genre(X, \text{“}Adventure\text{”}). \tag{11}$$

In the single rule (4) of program P_1, the user simply overrides any previous rule that would recommend an *Animation* movie. In program P_2, she introduces the notion of a good movie in rule (5) and rules (6) – (9) make sure there is always at least one good movie recommended[4]. Program P_3 extends the definition of a good movie and program P_4 avoids recommendations of *Adventure* movies.

With $\mathcal{P}_1 = (P_1)$, $\mathcal{P}_2 = (P_1, P_2)$, $\mathcal{P}_3 = (P_1, P_2, P_3)$, $\mathcal{P}_4 = (P_1, P_2, P_3, P_4)$, the recommender frames $\langle M, P_0, \mathcal{P}_1 \rangle$, $\langle M, P_0, \mathcal{P}_2 \rangle$, $\langle M, P_0, \mathcal{P}_3 \rangle$ and $\langle M, P_0, \mathcal{P}_4 \rangle$ have

[3] The even loop through default negation is used in ASP to generate two models.

[4] These rules are a classic construct of ASP. The actual recommendation system would, like most answer-set solvers, have special syntactical shortcuts for this kind of specifications, since we cannot expect the user to write them from scratch.

Input	All				No title				1000				1000 no title			
User DLP	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$
Parsing	195	196	199	197	151	150	149	150	95	96	96	96	81	84	83	85
Grounding	1596	1591	1598	1639	1156	1159	1154	1181	468	470	475	476	329	330	332	339
Transformation	496	503	502	504	305	310	320	407	131	133	133	135	134	137	136	138
Stable models	1289	1294	1312	1320	919	922	963	859	308	310	326	310	218	220	235	223
Total	3576	3584	3611	3660	2531	2541	2586	2597	1002	1009	1030	1017	762	771	786	785

Table 2. All test results ordered by input and program (time in milliseconds)

2, 2, 6 and 1 stable recommendations, respectively, which, for lack of space we cannot list here. Instead, we list only the final one and invite the reader to see how it complies with the user rules, how contradictory user rules are solved (e.g. adventure movies that are good, such as movie 558), as well as those between user and owner rules (e.g. movie 15 is no longer recommended).

$$M = \{rec(12), rec(497), rec(527), rec(551), rec(589), rec(1249), rec(1267), \\ rec(1608), rec(1856), rec(1912), rec(2396), rec(3746)\}$$

We now turn our attention to time performance. To investigate the importance of the size of the input, we tested using the programs specified above but with the following four databases with varying number of movies and concepts:

- **All:** 3883 movies, all concepts (17406 atoms)
- **No title:** 3883 movies, all concepts except $title(ID, Title)$ (9640 atoms)
- **1000:** 1000 movies, all concepts (4428 atoms)
- **1000 no title:** 1000 movies, all concepts except $title(ID, Title)$ (3440 atoms)

For each database and each recommender frame we computed all the stable recommendations 100 times using an Intel Pentium D 3.4 GHz processor with 2 MB cache and 1 GB of RAM. The average times for each step of the implementation are shown in Table 2. As can be seen from the table, the time it takes to compute the recommendations varies with the size of the database. As expected, the parsing time grows linearly. The size of the database directly determines the number of terms substituted for the variables in the grounding phase hence the number of generated rules. The transformation is then linear and the worst case upper bound is $2m + l$ rules after the transformation, where m is the number of rules in the grounded DLP and l is the number of atoms in the grounded DLP. The last phase, computation of stable models, is known to be NP-hard although, as usual, SMODELS performs quite well in keeping the computation time low, making ERASP look quite promising. Apart from that, it can be run on computers much faster than the one we had at our disposal. It is also worth noting that the addition of common user rules has little effect on performance.

5 Concluding Remarks

In this paper we proposed ERASP, a system that can work as an add-on for existing recommender systems. Owners of such systems should be able to plug in

the application as a recommendation enhancer, for offering users the possibility of explicit preference creation, for defining specific system rules, or both. A rule-based language like DLP can empower owners of recommender systems with the necessary tools to employ marketing strategies with precision, while keeping the diversity of recommendations and following user's preferences. Moreover, it can be used as a query tool to extract information from the database using a more sophisticated language than for example SQL, allowing the system to be used on its own without the need of a previously existing recommender systems. Users of ERASP can find a rich language to interact with the recommender system, gaining control, if they so desire, over the recommendations provided.

ERASP has a formal declarative semantics based on ASP and DLP, thus inheriting many of their theoretical properties and efficient implementations.

ERASP also enjoys other formally provable properties, e.g. the property of both positive and negative supportiveness which insures there always exists an explanation for each recommended item. The provided semantics makes multiple recommendation sets possible, facilitating diversity and non-determinism in recommendations.

ERASP has been implemented and preliminary tests are very encouraging. For databases of some reasonable size (few thousand products) often found in specialised e-commerce applications, the system is readily applicable. For larger databases, not only there is still room for implementation optimizations, but we can drastically improve its performance with an iteration mechanism which would first try to compute the stable models only with a part of the database, of fixed size, chosen according to some criteria of the initial recommender system, e.g. according to the rating of the nearest neighbors in a collaborative filtering recommender system. If recommendations are found, they are presented to the user. If there are no recommendations, then they are recomputed with a bigger (or different) part of the database. The number of items chosen in each step and control over whether and how the iterative approach is being used could depend on the demands of the domain, it could even be controlled directly by the owner and/or user or change dynamically according to some immediate external changes, like the load of the server or similar.

Another way to improve the system in terms of speed would be to restrict the database used to concepts explicitly appearing in the rules, as shown in our tests when we used an input database without the concept $title(ID, Title)$. Furthermore, the grounding and transformation phase of the implementation still have room for optimizations.

One issue that needs to be tackled is that of the user interface, namely the task of writing rules, burdening for the user [12]. Without addressing this issue, ERASP can still provide added value for experts of specific domains that are willing to learn how o write such rules to satisfy their demand of higher accuracy and, more important, reliability of recommendations. Even for less demanding users, there are still some easy to write rules that provide some basic interaction with the recommender system that is of great help. Dealing with this issue includes transforming natural language sentences into logic programs [13],

creating natural language interfaces for databases [4], creating rules by tagging and suggestion and learning rules by induction [1].

As for related work, in [11] Defeasible Logic Programming is employed in the ArgueNet website recommender system. Lack of space prevents us from comparing both architectures. To the best of our knowledge, ArgueNet has not been implemented yet.

References

1. J.S. Aitken. Learning information extraction rules: An inductive logic programming approach. In *Procs. of ECAI'02*. IOS Press, 2002.
2. J. Alferes, J. Leite, L. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *J. Logic Programming*, 45(1-3), 2000.
3. J. J. Alferes, F. Banti, A. Brogi, and J. A. Leite. The refined extension principle for semantics of dynamic logic programming. *Studia Logica*, 79(1), 2005.
4. I. Androutsopoulos, G.D. Ritchie, and P. Thanisch. Natural language interfaces to databases–an introduction. *Journal of Language Engineering*, 1(1):29–81, 1995.
5. F. Banti, J. J. Alferes, and A. Brogi. Operational semantics for DyLPs. In *Procs. of EPIA'05*, 2005.
6. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
7. D. Billsus and M. J. Pazzani. User modeling for adaptive news access. *User Model. User-Adapt. Interact*, 10(2-3):147–180, 2000.
8. D. Billsus and M. J. Pazzani. Content-based recommendation systems. In *The Adaptive Web*, volume 4321 of *LNCS*, pages 325–341. Springer, 2007.
9. R. Burke. Knowledge-based recommender systems. In *Encyclopedia of Library and Information Systems*, volume 69. M. Dekker, 2000.
10. R. D. Burke. Hybrid recommender systems: Survey and experiments. *User Model. User-Adapt. Interact*, 12(4):331–370, 2002.
11. C. Chesñevar and A. Maguitman. ArgueNet: An argument-based recommender system for solving web search queries. In *Procs. of the 2nd. International IEEE Conference on Intelligent Systems*, pages 282–287. IEEE Press, June 2004.
12. M. Claypool, P. Le, M. Wased, and D. Brown. Implicit interest indicators. In *Intelligent User Interfaces*, pages 33–40, 2001.
13. N. E. Fuchs and R. Schwitter. Specifying logic programs in controlled natural language. Technical Report ifi-95.17, 1, 1995.
14. M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Procs. of ICLP'90*. MIT Press, 1990.
15. D. Goldberg, D. Nichols, B. M Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, December 1992. Special Issue on Information Filtering.
16. S. K. Lam, D. Frankowski, and J. Riedl. Do you trust your recommendations? An exploration of security and privacy issues in recommender systems. In *Procs. of ETRICS'06*, 2006.
17. J. Leite and M. Ilic. Answer-set programming based dynamic user modeling for recommender systems. In *Procs. of EPIA'07*, volume 4874 of *LNAI*, pages 29–42. Springer, 2007.
18. J. A. Leite. *Evolving Knowledge Bases*. IOS press, 2003.
19. J. Ben Schafer, J. A. Konstan, and J. Riedl. E-commerce recommendation applications. *Data Min. Knowl. Discov*, 5(1/2):115–153, 2001.