ASP: from Application Development to Syntax Extensions

Magdalena Ortiz

Universidad de las Américas, CENTIA
Sta. Catarina Mártir, Cholula, Puebla
72820 México
is103378@mail.udlap.mx

Abstract In this paper some theoretical results in ASP are discussed in the context of application development. An application of ASP in collaborative learning is presented as a motivation. Some extensions to the syntax of ASP are discussed. For a particular extension, namely logic programs with embedded implications, new results are given.

1 Introduction

Answer Set Programming (ASP) has become without a doubt one of the most important logical formalisms for non monotonic reasoning. It has been deeply studied for more than 15 years, many extensions have been proposed and its relations with other approaches have been more clearly understood. However, it seems surprising that the development of applications using ASP in reasoning tasks related to the solution of real life problems is still scarce. Many authors have pointed out lately that there is a strong need of developing applications of ASP. In the first place, ASP is intended to be useful. As a logic programming tool, it should fall into the "thinking" part of logic intelligent agents, and allow them to perform real commonsense reasoning. On the other hand, ASP needs to be faced with real life needs, in order to test its power and suitability for real reasoning tasks. At the same time, this should help the ASP community discover new challenges and give their research new directions.

In [4,5], we used ASP for learner modeling and supporting collaboration in learning environments. ASP proved to be a very suitable formalism. Modeling the application was natural and intuitive, and the behavior of the system is as expected. However, while developing this application, we met some issues suggesting that extensions to the syntax of ASP would allow more natural and direct problem solving. The aim of this paper is to point out these challenges, as well as to make a survey of the directions they have led our research to and the results achieved, namely related to extending the answer set semantics to programs with embedded implications in the body of the rules.

The papers is structured as follows: In section 2 some basic concepts and notation are given. In section 3, we will recall the application presented in [4,5] and discuss the problems encountered during its development. Having this motivation, in section 4 we will focus on the relevance of embedded implications in the body of the rules when modeling application using ASP. We will analyze some alternatives to overcome this problems and discuss their limitations. These alternatives include both translations between classes of programs as well as syntax extensions. Section 5 is devoted to a particular one of these alternatives. For this purpose, we will recall some of the results presented in [7] and give some more detailed results as well. Finally, in section 6 we present some still open questions and give some directions for future work.

2 Background and Notation

The language of propositional logic has an alphabet consisting of propositional symbols: $p_0,\ p_1,\dots$; connectives: $\wedge, \vee, \leftarrow, \perp$ and auxiliary symbols: $(,)$ where $\wedge, \vee, \leftarrow$ are 2-place connectives and $\perp$ is a 0-place connective. Propositional symbols are also called *atoms* or *atomic propositions*. Formulas and theories are defined as usual in logic. The formula $\neg F$ is introduced as an abbreviation of $\perp \leftarrow F$. The formula $F \rightarrow G$ is just another way of writing the formula $G \leftarrow F$. A signature $\mathcal{L}$ is a finite set of propositional symbols. If F is a formula then the *signature* of F, denoted as $\mathcal{L}_F$, is the set of propositional symbols that occur in F. A *literal* is either an atom a (a positive literal) or the negation of an atom $\neg a$ (a negative literal). A negated literal is the negation sign $\neg$ followed by any literal, i.e. $\neg a$ or $\neg\neg a$. Given a set of formulas $\mathcal{F}$, we define $\neg\mathcal{F} = \{\neg F \mid F \in \mathcal{F}\}$.

A clause is a formula of the form $H \leftarrow B$ where H and B, arbitrary formulas in principle, are called the *head* and *body* of the clause respectively. A *general clause* is a clause of the form $h_1 \vee \cdots \vee h_n \leftarrow b_1 \wedge \cdots \wedge b_m$ where $h_1 \cdots h_n$ are atoms and $b_1 \cdots b_m$ are literals. A *logic program* is a finite set of clauses. A set of general clauses is a *general program*. A clause of the form $H \leftarrow b_1 \wedge \cdots \wedge b_m$ will be also written as $H \leftarrow b_1, \cdots, b_m$. Let P be a program and M a set of atoms such that $M \subseteq \mathcal{L}_P$ then we define $\widetilde{M} = \mathcal{L}_P \setminus M$.

In the logic programs presented in this paper we use to types of negation. The negation *not* is the usually called *default negation* and is the logic programming counterpart of the logical negation $\neg$. The other negation $\sim$ represents the *strong* or *explicit* negation. We use it for practical purposes, but any program with this negation can be easily translated into one without by simply renaming atoms and adding constraints. We will not go deeper into the subject, but the reader can just as well consider that only the default negation *not* is being used.

We will use I to refer to *intuitionistic* logic and $G_3$to refer to Gödel's three valued logic, also known as the logic of *here-and-there*. We say that a theory T is consistent with respect to logic X iff there is no formula A such that $T \vdash_X A$ and $T \vdash_X \neg A$. We say that a theory T is (literal) complete w.r.t. logic X iff, for all $a \in \mathcal{L}_T$, we have either $T \vdash_X a$ or $T \vdash_X \neg a$. Two programs P_1 and P_2 are *equivalent under logic* X, denoted as $P_1 \equiv_X P_2$, iff $P_1 \vdash_X A$ for every $A \in P_2$ and $P_2 \vdash_X A$ for every $A \in P_1$. For a given set of atoms M and a program P we will write $P \vdash_X M$ to abbreviate $P \vdash_X a$ for all $a \in M$ and $P \Vdash_X M$ to denote the fact that P *is consistent (w.r.t. logic* X*) and* $P \vdash_X M$. If one of the symbols $\vdash_X$ or $\Vdash_X$ lacks of the subscript X we assume that it refers to the intuitionistic logic I. As we will mention later, in this paper we are considering answer sets as defined in [6], i.e. in terms of intuitionistic extensions. Under this definition $P_1 \equiv_{\text{stable}} P_2$ means that P_1 has the same answer sets as P_2.

3 ASP for Collaborative Learning Environments

In the areas of computer assisted learning and tutoring systems, logic programming has been widely used, usually for knowledge representation. However, the development of logic-based applications for collaboration in learning communities is very scarce. In this paper, we address the problem from the perspective of ASP. In [23], we proposed ASP as a suitable basis for learner modeling in Computer Supported Collaborative Learning (CSCL) environments. We continued with this work in [5], where ASP was used to model how the agents in this kind of learning environments can effectively support collaboration in the community according to their learner models. Based on the work of [2,1], in our proposal the agents in a CSCL environment hold a set of beliefs about the learners, which they use to infer the best learning and collaboration opportunities for them within the community. The agents suggest the learners sets of tasks to work on, as well as collaboration groups according to the agents' beliefs about the interests and capabilities of the learners. For this purpose, the agents carry out non-monotonic inference about the interests and capabilities of the learners. The agent draws conclusions to propose the learners suitable tasks and work groups within the learning community.

In this context, some statements similar to the following one have to be expressed in a disjunctive program:

1. A learner is (normally) capable of applying a knowledge element, if he is capable of applying all the knowledge elements which are a specialization of it.
2. If a learner is capable of applying a knowledge element, he/she is also (normally) capable of applying the knowledge elements which are a specialization of it.

We would expect a natural and intuitive translation of these statements into logic, to look like this:

$$\begin{aligned} \mathit{capable}(Ke) \leftarrow\ & \forall Ke1[\mathit{specialization}(Ke, Ke1) \rightarrow \mathit{capable}(Ke1)], \\ & \mathit{not}\ {\sim}\mathit{capable}(Ke). \end{aligned} \tag{1}$$

This fist expression means that if a learner is capable of applying all the knowledge elements that are a specialization of Ke, he will also be capable of applying Ke. The last part of the rule: *not* $\sim$*capable*(Ke) represents the word *normally.* Intuitively, it weakens the rule allowing exceptions: we can believe a learner is capable of Ke as long as there is no evidence of the learner not being capable.

$$\begin{aligned} \mathit{capable}(Ke1) \leftarrow\ & \mathit{specialization}(Ke, Ke1), \\ & \mathit{capable}(Ke), \\ & \mathit{not}\ {\sim}\mathit{capable}(Ke1). \end{aligned} \tag{2}$$

This second expression goes in the opposite direction: if a learner is capable of applying a knowledge element Ke, we infer that he is also capable of all the knowledge elements that are a specialization of it. Note that the *normally* weakening is also present in the rule.

Rule 1 suggests the usefulness of having embedded implications in the body of rules when modeling applications using ASP. At the time we were developing this application, we came across other problems where similar situations appeared, and an extension to the syntax seemed to be an useful alternative. For example, in [9] the authors present some knowledge representation problems that don't seem to have an uniform and natural encoding as standard disjunctive programs. For this reason they introduce *parametric connectives*, which behave in a similar way to the extension we are proposing for some cases. We agree with the remark done by Michael Gelfond via e-mail communication, in the sense that an extension to the syntax of the language could be needed: *"The ability to use implication in the body seems to suggest the following translation: 'r is true if every element with property p has property q' The natural translation is:* $\forall(X)(p(X) \rightarrow q(X)) \rightarrow r$*. If no implication is allowed in the formal language the translation of this English statement loses its universal character. It now depends on the context and is prone to error."*

4 Embedded implications for Knowledge Representation in ASP

In order to solve knowledge representation problems similar to the one introduced in section 3, some alternatives have been proposed. In this section we discuss some of them in detail. The goal of the discussion is to see whether this alternatives really provide rule 1 and similar problems a natural and uniform encoding and a suitable semantics.

For our discussion we will take the example presented above. Additionally to rules 1 and 2, we will give some facts (EDB). We will call P the following program:

Example 1.

$$\begin{array}{ll} \mathit{knowElem}(k1). & \mathit{specialization}(k1, k2). \\ \mathit{knowElem}(k2). & \mathit{specialization}(k1, k3). \\ \mathit{knowElem}(k3). & \mathit{specialization}(k1, k4). \\ \mathit{knowElem}(k4). & \mathit{specialization}(k5, k6). \\ \mathit{knowElem}(k5). & \mathit{capable}(k2). \\ \mathit{knowElem}(k6). & \mathit{capable}(k3). \\ & \mathit{capable}(k4). \end{array}$$

$$\begin{aligned} capable(Ke) \leftarrow\ & hasSpecialization(Ke), \\ & \forall Ke1[specialization(Ke, Ke1) \rightarrow capable(Ke1)], \\ & not\ {\sim}capable(Ke). \end{aligned} \tag{1}$$

$$hasSpecialization(K) \leftarrow knowElem(K),\ specialization(K, K1).$$

$$\begin{aligned} capable(Ke1) \leftarrow\ & specialization(Ke, Ke1), \\ & capable(Ke), \\ & not\ {\sim}capable(Ke1). \end{aligned}$$

Roughly, we could classify the existing alternatives in two groups: those which try to find an encoding within the standard syntax that has the expected meaning and those that extend the syntax to be able to express this rule and give it a suitable semantics.

4.1 Coding Embedded implications in Disjunctive Logic Programs

Can rule 1 be encoded as a standard logic program? In this direction many approaches have been made. In [7] we discussed an alternative which we called *classical translation*. the sentence "every element that holds property p holds property q", is rewritten as "there is no element that holds property p and does not hold property q", i.e. rule 1 is replaced in P by:

$$\begin{aligned} capable(Ke) \leftarrow\ & hasSpecialization(Ke), \\ & not\ notHoldsForAll(Ke), not\ {\sim}capable(Ke). \\ notHoldsForAll(Ke) \leftarrow\ & specialization(Ke, Ke1), \\ & not\ capable(Ke1). \end{aligned} \tag{3}$$

The direct encoding has some unexpected models, as presented in [7]. The codification can be *forced* to work as expected. There are many ways in which this forcing could be done. For example, if we add *not not capable*(Ke) to rule 3 the problem seems to be solved. Other *ad hoc* stratifications or reductions can be done. Some have been tried and are available on line[1]. However, none of this solutions, which are all more or less *ad hoc* to the problem really provide an uniform and well-behaved encoding for embedded implications. A translation from programs with embedded implications into standard disjunctive ones that preserves the expected behavior is given in [3], but it is questionable whether this approach is computationally convenient, and it does not solve the problem since it does not provide an easy and natural encoding of problems and an intuitive semantics.

[1] http://mail.udlap.mx/∼is103378/research/rigid/examples

4.2 Syntax Extension Alternatives to Embedded Implications

In other cases, the problem has been addressed by proposing an extension to the syntax. We will focus on one particular extension: *parametric connectives.* When proposing this extension the authors do not address the problem of embedded implications, but they are motivated by other problems encountered when using ASP to solve knowledge representation problems. However some problems that can be addressed by embedded implications can also be solved using parametric connectives, and this is the case of our example.

Parametric Connectives Parametric connectives were introduced by Leone and Perri in [9]. They are a good alternative for a natural and uniform encoding of several problems. This approach does not represent an explicit implication, but as we will see, the semantics and behavior are suitable for this problem. Using this approach, rule 1 will be represented as a parametric literal, namely a parametric conjunction. The intuition is translating the statement "being capable of all knowledge elements that are a specialization of K" into a term whose truth value is given by the conjunction of all instances of the predicate $capable(K')$ for all K' such that K' is an specialization of K. In our program P, rule 1 would be replaced by the following rule [2]:

$$\begin{aligned} capable(Ke) \leftarrow\ & hasSpecialization(Ke),\\ & \bigwedge\{capable(Ke1) : specialization(Ke, Ke1)\},\\ & not\ {\sim}capable(Ke). \end{aligned}$$

Doing the universal and then a local grounding on the program P, we have an instance of the rule in $ground(P)$ for every knowledge element Ke. Taking the set of facts in the EDB as a basis for the interpretation I, we obtain a valuation on the rules as follows:

$$\begin{aligned} capable(k1) \leftarrow\ & hasSpecialization(k1),\\ & capable(k2),\\ & capable(k3),\\ & capable(k4),\\ & not\ {\sim}capable(k1). \end{aligned}$$

$$\begin{aligned} capable(k2) \leftarrow\ & hasSpecialization(k2),\\ & not\ {\sim}capable(k2). \end{aligned}$$

Applying the extended *Gelfond-Lifschitz transform* on this program we obtain exactly the expected answer set, which contains $capable(k2)$, $capable(k3)$, $capable(k4)$, $capable(k1)$, and no other instances of $capable$. This coding of the program provides a natural and intuitive syntax, as well as the expected semantics.

[2] Note that *strong* negation does not strictly belong to the syntax of $DLP^{\bigwedge,\bigvee}$. We are assuming, as we roughly commented in section 2, that strong negation can be simulated and that its use only provides an easier and shorter encoding of programs.

5 KR with Positive Embedded Programs.

In [7] we presented a family of logic programs that allows a restricted use of implications in the body of the rules called positive embedded programs. Now we will address the same example modeling it as a positive embedded program.

Definition 1 (Positive embedded program). *[7] A* positive embedded conjunct *is either a literal or a formula of the form* $(a \rightarrow b)$*, where* a *and* b *are atoms. A* positive embedded clause *is a clause of the form* $H \leftarrow B$*, where* H *is an atom and* B *is a conjunction of positive embedded conjuncts.* P *is a* positive embedded program *if for every clause* $\alpha \in P$*,* α *is either a general clause or a positive embedded clause.*

In rule 1, the universal quantifier is interpreted as an abbreviation of a conjunction of elements. We could rewrite the quantifier as the explicit conjunction (remember that the Herbrand universe is finite). This subject could be subject of deeper discussion, but it is out if the scope of this paper. Thus we assume that on the ground program, the expression $\forall x[p(x)]$ becomes a simple conjunction $p(a_1), \ldots, p(a_n)$ of all the ground instances of predicate p. As an example, we give two ground instances of rule 1.

$$\begin{aligned}
\mathit{capable}(k1) \leftarrow\ & \mathit{hasSpecialization}(k1),\\
& \mathit{specialization}(k1,k1) \rightarrow \mathit{capable}(k1),\\
& \mathit{specialization}(k1,k2) \rightarrow \mathit{capable}(k2),\\
& \mathit{specialization}(k1,k3) \rightarrow \mathit{capable}(k3),\\
& \mathit{specialization}(k1,k4) \rightarrow \mathit{capable}(k4),\\
& \mathit{specialization}(k1,k5) \rightarrow \mathit{capable}(k5),\\
& \mathit{specialization}(k1,k6) \rightarrow \mathit{capable}(k6),\\
& \mathit{not} \sim\!\mathit{capable}(k1).\\[1ex]
\mathit{capable}(k2) \leftarrow\ & \mathit{hasSpecialization}(k2),\\
& \mathit{specialization}(k2,k1) \rightarrow \mathit{capable}(k1),\\
& \mathit{specialization}(k2,k2) \rightarrow \mathit{capable}(k2),\\
& \mathit{specialization}(k2,k3) \rightarrow \mathit{capable}(k3),\\
& \mathit{specialization}(k2,k4) \rightarrow \mathit{capable}(k4),\\
& \mathit{specialization}(k2,k5) \rightarrow \mathit{capable}(k5),\\
& \mathit{specialization}(k2,k6) \rightarrow \mathit{capable}(k6),\\
& \mathit{not} \sim\!\mathit{capable}(k2).
\end{aligned} \tag{4}$$

Here we see that rule 1 is expressed as a *positive embedded rule*. The answer sets semantics of this kind of programs is given in [7]. Here the semantics is not given by some extended *Gelfond-Lifschitz reduct*, but in terms of intuitionistic provability instead. Many authors have previously defined the answer set semantics of programs in terms of some non-classical logics. We follow the line started by Pearce[8] and studied in detail by Osorio et al. [6]. In [7] the answer sets of arbitrary theories are defined as intuitionistically complete and consistent extensions obtained by adding only negated and double negated literals.

From a practical point of view, embedded implications are not supported by any software that would allow us to compute answer sets. To approach this issue, we provide a translation that can be applied to positive embedded programs and may reduce them to a simpler family in order to implement them in some existing answer set solver. It is also our purpose to give the reader a clearer intuition of the semantics of programs with embedded implications as defined in [7].

5.1 Translation into General Programs

Before giving the details of the translation we will define some important concepts. In this section we also present the proofs of some results that were briefly introduced in in [7].

Remark 1. For one of the proofs, we will need a reduction presented in [7]. However, due to space limitations, we can not present the reduction here. A given program P can be reduced w.r.t. a set of negated and double negated literals $\neg\widetilde{M}$, $\neg\neg M$ into a positive program $red(P, M)$. For more details, please see [6].

Lemma 1. *Let P be a positive program and let H_P be the set of atoms that occur in the heads of the rules of P. If $P \Vdash_{\mathrm{G}_3} a$ then $a \in H_P$.*

Proof. The proof can be done by contradiction as follows: suppose $a \notin H_P$, take I to be a definite interpretation [3] in G_3 that is a model of P and that $I(a) = 2$. Construct a new interpretation I' such that $I'(a) = 1$ and $I'(x) = I(x)$ otherwise. It can be proved that for every rule $H \leftarrow B$ of P, we have that $I'(H) = I(H)$ (since a does not occur in H) and $I'(B) \leq I(B)$ (an exhaustive proof can be done, but it is rather straight forward: the body in a conjunction of elements, hence it behaves monotonically. The conjuncts with embedded implications behave also monotonically, prove all cases). Thus we have an interpretation I' such that $I'(P) = 2$ and $I'(a) = 1$, hence $P \not\Vdash_{\mathrm{G}_3} a$. (Note that if a were in the head of any rule, it could be the case that I' was not a model of P).

Proposition 1. *Let P be a logic program and let H_P be the set of atoms that occur in the heads of the rules of P. If $a \notin H_P$, then $P \equiv_{\mathrm{stable}} P \cup \{\neg a\}$*

Proof. Suppose $P \cup \{\neg a\} \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_{\mathrm{I}} M$. Since $P \cup \{\neg a\} \cup \neg\widetilde{M} \cup \neg\neg M$ is consistent and complete, then $a \notin M$, and $\neg a \in \neg\widetilde{M}$. Thus $P \cup \{\neg a\} \cup \neg\widetilde{M} \cup \neg\neg M \equiv_{\mathrm{I}} P \cup \neg\widetilde{M} \cup \neg\neg M$, so $P \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_{\mathrm{I}} M$. Now suppose $P \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_{\mathrm{I}} M$. If $a \notin M$, then $\neg a \in \neg\widetilde{M}$ and we have trivially that $P \cup \{\neg a\} \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_{\mathrm{I}} M$. If $a \in M$, then $P \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_{\mathrm{I}} a$. Let P' be $red(P, M)$. Then by Lemma 2 in [7] we have that $P' \Vdash_{\mathrm{I}} a$ and thus, by Lemma 1, $a \in H_{P'}$, which implies $a \in H_P$ and we have a contradiction, so it can not be the case that $a \in M$.

[3] A *definite* interpretation is one in which all literals take the values 0 or 2 (equiv. $\perp$ or $\top$)

Definition 2. *Given a set of literals L and a positive embedded rule r, r^L is obtained from r by replacing every conjunct of the form $(a \to b)$ by b if $a \in L$, by $\top$ if $\neg a \in L$ and doing no replacements otherwise. For any positive embedded program P, P^L is the program obtained by replacing every conjunctive rule $r \in P$ by r^L.*

Remark 2. For any positive embedded program P and a set of literals L such that $P \Vdash_{\mathrm{I}} L$, $P^L \equiv P$.

Proposition 2. *Let P be a positive embedded program. Let F be the set of facts in P and $Lit(H)$ the literals occurring in the heads of the rules in P. Let $F' \subseteq F$ and $\neg\widetilde{H}' \subseteq \neg(\mathcal{L}_P \setminus Lit(H))$. Then $P \equiv_{\text{stable}} P^{F \cup \neg\widetilde{H}'}$*

Proof. By Proposition 1 we know that $P \equiv_{\text{stable}} P \cup \neg(\mathcal{L}_P \setminus Lit(H))$ and as we have trivially that $P \cup \neg(\mathcal{L}_P \setminus Lit(H)) \Vdash_{\mathrm{I}} F \cup \neg(\mathcal{L}_P \setminus Lit(H))$, then $P \equiv_{\text{stable}} P^{F \cup \neg(\mathcal{L}_P \setminus Lit(H))}$.

Going back to our example, let P be the program in 1 after grounding. Let's define F to be the set of facts in P and $L_H := Lit(H)$ as the set of literals that appear in the heads of the rules of P. Moreover, we can define $F_{spec} \subset F$ to be the subset of F that contains all instances of *specialization*, and $\widetilde{L_{H\,spec}} \subset \widetilde{L_H}$ contains all instances of *specialization* that are not in L_H. Now we have the following sets:

$$\begin{aligned} F_{spec} &= \{\ specialization(k1, k2), \ldots, specialization(k5, k6)\} \\ \neg\widetilde{L_{H\,spec}} &= \{\ \neg specialization(k1, k1), \ldots \neg specialization(k6, k6)\} \end{aligned}$$

We define a new program $P' := P \cup \neg\widetilde{L_{H\,spec}}$. By Proposition 1 we know that the stable models of P are preserved in P'. Let $L := F_{spec} \cup \neg\widetilde{L_{H\,spec}}$. It is easy to see that every ground instance of *specialization* is in L, and that $P' \Vdash_{\mathrm{I}} L$. so we can apply the P'^L reduction to obtain a general program that has exactly the answer sets of P. We give as an example the first two rules after applying the reduction:

$$\begin{aligned} capable(k1) \leftarrow\ & hasSpecialization(k1), capable(k2), capable(k3), \\ & capable(k4), not\ {\sim}capable(k1). \\ capable(k2) \leftarrow\ & hasSpecialization(k2), not\ {\sim}capable(k2). \end{aligned}$$

The only answer set of the general program we obtain contains exactly the expected instances of *capable*: $capable(k2)$, $capable(k3)$, $capable(k4)$, $capable(k1)$.

6 Conclusions and Future Work

In this paper we have presented a review of some theoretical results in ASP. However, the results were motivated by real applications, and this research has been deeply related to the improvements done to the application we described. From our point of view this practical support gives the results additional relevance. With this work we intend to make a small collaboration towards making ASP a real useful formalism suitable for solving real life problems and developing useful applications. Concerning the extension of the ASP syntax to wider classes of programs, for example allowing the use of embedded implications, there is still much work to be done. More complete and precise results must be given. Some important issues were not given a proper treatment yet, like equivalence and strong equivalence, the use of two types of negation, etc. We hope to have some results in this direction soon.

This work should be a motivation to reconsider research in the theory of ASP from the context of real applications. The ultimate goal of this paper was to discuss how solving real commonsense reasoning problems should be a main source of research directions for the ASP community. We hope to have made clear that research in theory and applications are deeply related and it is not the case they should develop independently from each other.

References

1. Gerardo Ayala. Intelligent agents supporting the social construction of knowledge in a lifelong learning enviroment. In *Proceedings of the International Workshop on New Technologies for Collaborative Learning (NTCL 2000)*, pages 79–88, Hyogo, Japan, November 2000.
2. Gerardo Ayala and Yano Yoneo. Learner models for supporting awareness and collaboration in a cscl environment. *Lecture Notes in Computer Science 1086*, pages 158–167, 1996.
3. Juan Antonio Navarro. Properties of translations for logic programs. In Balder Ten Cate, editor, *ESSLLI03 Student Session. European summer School of Logic, Language and Information*, Vienna, Austria, August 2003.
4. Magdalena Ortiz, Gerardo Ayala, and Mauricio Osorio. Formalizing the learner model for cscl environments. In *Proceedings of the Fourth Mexican International Conference on Computer Science (ENC 03)*, pages 151–158. IEEE Computer Society and SMCC, Mexican Society for Computer Science, 2003.
5. Magdalena Ortiz de la Fuente. An application of answer sets programming for supporting collaboration in agent-based cscl enviroments. In Balder Ten Cate, editor, *ESSLLI03 Student Session. European summer School of Logic, Language and Information*, Vienna, Austria, August 2003.
6. Mauricio Osorio, Juan Antonio Navarro, and José Arrazola. Applications of intuitionistic logic in answer set programming. Accepted to appear at the TPLP journal, 2003.
7. Mauricio Osorio and Magdalena Ortiz. Embedded implications and minimality in asp. In *Accepted to apprear in 15th International Conference on Applications of Declarative Programming and Knowledge Management. INAP 2004*, Postdam, Germany, March 2004.
8. David Pearce. Stable inference as intuitionistic validity. *Logic Programming*, 38:79–91, 1999.
9. Simona Perri and Nicola Leone. Parametric connectives in disjunctive logic programming. In *ASP03 Answer Set Programming: Advances in Theory and Implementation*, Messina, Sicily, September 2003.