A Visual Interface for Drawing ASP Programs

Onofrio Febbraro, Kristian Reale, and Francesco Ricca

Dipartimento di Matematica, Università della Calabria, 87030 Rende, Italy
{lastname}@mat.unical.it

Abstract. Answer Set Programming (ASP) is a purely declarative logic programming paradigm proposed in the area of non-monotonic reasoning and logic programming. In the last few years, a rich set of tools for ASP-program development were proposed, including editors and debuggers. However, the task of designing a logic program consists of writing text files (more or less computer-assisted). In this paper we present a system that allows for *drawing* an ASP-program on the screen. The user does not have to edit text files or know the details of a specific ASP dialect, since our approach offers a fully graphic environment, inspired by QBE editors, for designing ASP programs.

1 Introduction

Answer Set Programming (ASP) is a declarative programming paradigm which has been proposed in the area of non-monotonic reasoning and logic programming. The idea of ASP is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and then use a solver to find such a solution [1]. The language of ASP is expressive (it is able to express all problems belonging to the complexity classes Σ_2^P and Π_2^P, under brave and cautious reasoning, respectively [2]); furthermore, the availability of some efficient ASP systems [3, 4] made ASP a powerful tool for developing advanced knowledge-based applications [5–7].

In order to facilitate the design of ASP applications, a rich set of tools for ASP-program development were proposed in the last few years, including editors [8, 9] and debuggers [10–12]. However, the task of designing a logic program consists of writing text files (more or less computer-assisted). Although the basic syntax of ASP is not particularly difficult, writing ASP programs might be uncomfortable for novices and error-prone; moreover, programmers often have to know the details of a specific ASP input dialect. To face with a similar problem in the field of databases, a number of tools and graphical user interfaces were proposed [13–16] starting from the 70s for facilitating the specification of queries. Today many commercial and free relational database query tools offer fully graphical Query By Example (QBE) interfaces for facilitating the end approach of users to systems and languages. The practical relevance of graphic tools is now well-recognized: a QBE interface is, indeed, the default in the user-oriented Microsoft Access.

However ASP still lacks this kind of tools, which might serve for reducing the difficulty of producing ASP programs for both novice and unexperienced programmers, and easing the encoding tasks for experts that prefer graphic tools.

In this paper we present *Visual ASP*, a system that allows for *drawing* an ASP-program on the screen. The user does not have to edit text files, or know the details of a specific ASP dialect, but he can exploit a fully graphic environment, inspired by QBE editors, for designing ASP programs. Currently, the system is able to load and store ASP programs in the syntax of the state-of-the-art ASP system DLV [17] , and supports all the main language extensions (i.e. disjunction, aggregates and constraints).

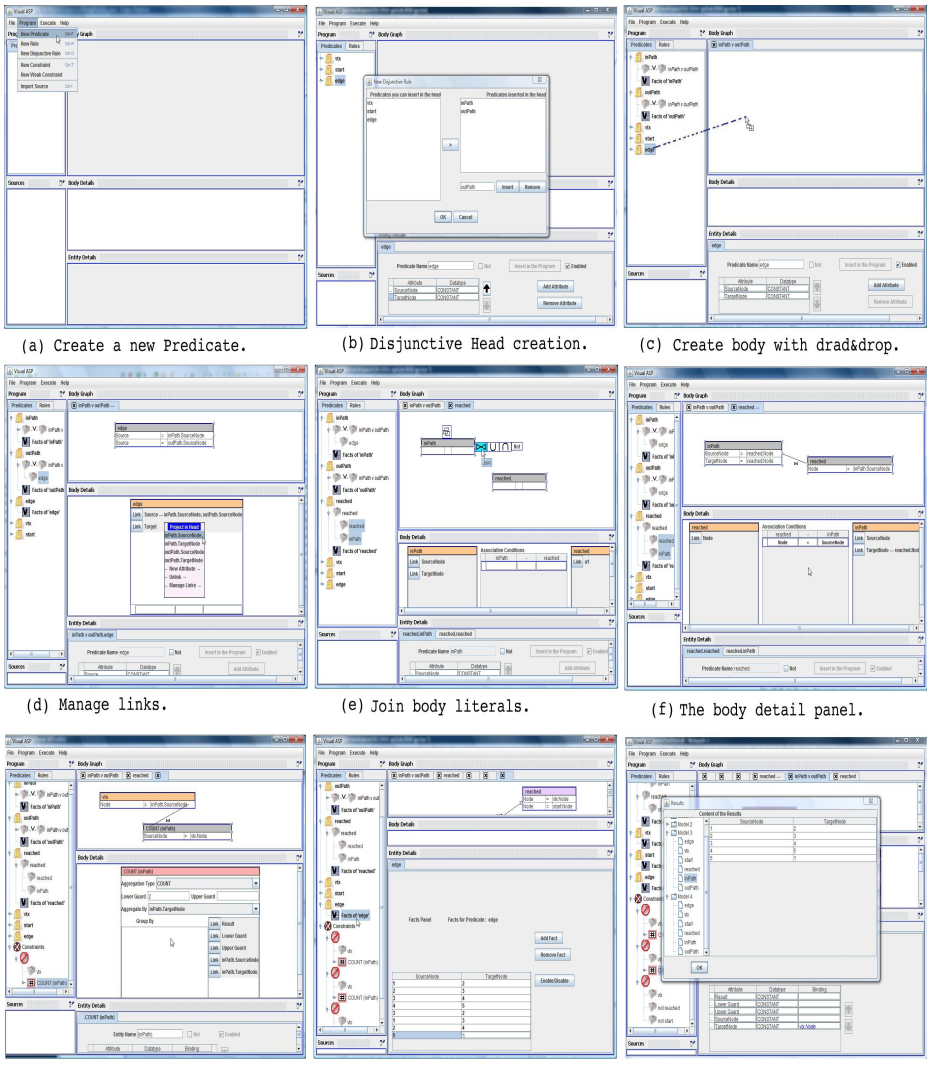
2 *Visual ASP*

In the following paragraphs we show how to use *Visual ASP* by exploiting an example.
Example Program. We consider the well-known *Hamiltonian Path* problem: *Given a finite directed graph $G = (V, A)$ and a node $a \in V$ of this graph, does there exist a path in G starting at a and passing through each node in V exactly once?* This is a classical NP-complete problem in graph theory. Suppose that the graph G is specified by using facts over predicates vtx (unary) and $edge$ (binary), and the starting node a is specified by the predicate $start$ (unary). The following program solves the problem:

```
% Guess arcs of the path
r1: inPath(X,Y) v outPath(X,Y) :- edge(X,Y).

% Auxiliary rules
r2: reached(X) :- reached(Y), inPath(Y,X).
r3: reached(X) :- start(X).

% Each vertex in the path must have at most
% one incoming and one outgoing edge.
r4: :- vtx(X), 2 <=  #count{ Y : inPath(X,Y) }.
r5: :- vtx(X), 2 <= #count{ Y : inPath(Y,X) }.

% All vertexes must be in the path.
r6: :- vtx(X), not reached(X), not start(X).
```

The disjunctive rule (r_1) guesses a subset S of the arcs to be in the path, while the rest of the program checks whether S constitutes a Hamiltonian Path. Here, an auxiliary predicate *reached* is defined, which specifies the set of nodes which are reached from the starting node. In the checking part, the first two constraints (namely, r_4 and r_5) ensure that the set of arcs S selected by *inPath* meets the following requirements, which any Hamiltonian Path must satisfy: (i) a vertex must have at most one incoming edge, and (ii) a vertex must have at most one outgoing edge. The third constraint enforces that all nodes in the graph are reached from the starting node in the subgraph induced by S.
System Usage. We now show how to employ *Visual ASP* for *drawing* that solution on the computer's screen. Note that, the system supports many different ways of creating and modifying rules and constraints. For respecting the space constraints, we report only one of the possible combination of commands and shortcuts that can be exploited for designing a program solving the considered problem;[1] however, the application can be tried by downloading it from *http://www.mat.unical.it/ricca/programgui.html*. We start by adding the input predicates: vtx and $edge$. More in detail, to add a predicate to the program (see Figure 1a) click on the *Program* menu, select *New Predicate*. A dialog will ask for the name of the predicate to be inserted, and after specifying the required information and confirming the command, a new predicate icon appears on the left panel (labelled *Program*). In the panel placed in the bottom-right (labelled *Entity Details*) one can specify the arity of the predicate (currently selected in the program

[1] Note also that the example program reported here only contains non-ground rules and facts; but, if needed, the interface also allows for writing ground rules.



(a) Create a new Predicate. (b) Disjunctive Head creation. (c) Create body with drad&drop.

(d) Manage links. (e) Join body literals. (f) The body detail panel.

(g) Aggregate literal details. (h) Manage facts. (i) User-friendly output.

Fig. 1. Creating a program with *Visual ASP*.

panel) by adding a number of attributes.The system allows for specifying a name for each attribute, and a type. This additional information is very useful during the editing phase, since it allows for rapidly identifying and joining attributes. The result of the insertion of predicates vtx, $start$ and $edge$ having attributes with labels is depicted in Figure 1b. Now we insert the disjunctive rule r_1 by selecting "*New Disjunctive Rule*" on the *Program* menu. The system opens a dialog window used to specify the head of the rule (see Figure 1b), and adds this new rule in the panel on the left. Then, one can specify its body by dragging predicates from the *Program* panel to a specific area in the *Body Graph* panel (which is situated on the top-right part of the interface). In our example, (see Figure 1c) we drag the $edge$ predicate in the body graph corresponding to our disjunctive rule; a box representing the just added body predicate appears and by clicking on the *Link* button a pop-up menu allows for rapidly joining body and head (see in Figure 1d). The creation of rule r_2 is a bit more involved, since its body contains two literals sharing variables. We insert a new rule by selecting on the *Program* menu the *New Rule*; and, we name its head predicate $reached$; after that we drag in the body panel $inPath$ and $reached$ (note that this definition is recursive). Then, we join the two body literals by selecting both of them (this is done by clicking on the corresponding boxes in the body panel while pressing the shift key) and clicking on the *join* icon that appears on the top left of the selected boxes (see Figure 1e). After that, the system will show a dialog window where one can setup the join. The details of the join are reported in the *Body Details* panel (see Figure 1f), where the joined attributes can further modified. As before, we properly link head and body attributes, in this case by acting on the link button corresponding to the second attribute of $inPath$. We continue our example by specifying the constraints. To create the constraint defined by the rule r_4 we select *New Constraint* from the *Program* menu. The constraint, having a "forbidden" icon, appears on the *Program* panel. The body of constraints can be specified in the same way as the body of rules i.e. by dragging predicates from the *Program* panel. In our case we drag vtx and $inPath$. To aggregate the information contained in the predicate $inPath$ we select that predicate and, with a right-click, we click on *Aggregate* and select *Count* from a drop-down menu. The result, shown in Figure 1g, is the specification of that new constraint. Note that, in the $BodyDetails$ panel, an aggregate-specific box allows for setting guards, local variables etc. At this point we repeat a similar procedure to insert the remaining rule and constraint. The entire graphical representation of our program solving the Hamiltonian Path is reported in Figure 1h. Note that, the program panel offers a sort of outline, listing on a tree-shaped view predicates and rules defining them. An alternative view of the program, merely reporting the list of its rules, can be displayed by clicking on the tab "*Rules*" placed on the top of the panel. In order to test our program we specify an instance of the problem by adding some fact. More in detail, to add facts to a predicate (e.g. $edge$), we select *Facts of edge* on the $Program$ panel, and add the facts in the table shown in the $EntityDetails$ panel (see Figure 1h). Finally, to execute the program, we set up a *Run Configuration* where we can specify the path of the solver that we want to use. Currently we support the DLV system only. To open the Run Configuration we select *Run* from the menu *Execute*. Answer sets are reported on a user-friendly output form exploiting a tree-shaped list of models, and the output is shown in tabular form on the right. Basically, it is sufficient to select a

predicate leaf in the answer-sets tree to display the corresponding table on the right part of the form (predicate $inPath$ of the first answer set is selected, see Figure 1i).

Some Implementation Detail. Logic programs are internally represented by suitable *Data Structures* that are exploited by the GUI. Input programs are loaded in the system by the *Parser* module, which creates a representation of logic rules according to the internal data structures. The output of the system is produced by the *Output Builder* module, which can either write the designed program in a file or feed it as input of an ASP solver. The output of the ASP solver is then represented on the screen in a user-friendly interface. The system has been implemented in Java. In particular, the GUI is based on the JGraph (`http://www.jgraph.com/`) library, and the system is currently able to handle programs in DLV format. The system features a flexible design for all its modules, based on the *composite*, *strategy* and *builder* design patters [18], conceived for being extended for supporting other language features, dialects and systems.

3 Related Work and Conclusion

In this paper, we presented a graphical interface for designing ASP programs, that is able to support all the powerful language constructs of ASP, like disjunction, recursion, unstratified negation, constraints, and aggregates. In the literature different formalisms were proposed that use a visual approach to logic programming. The articles [20, 21] describe a visual logic programming language based on a topological diagrammatic notation which combines Venn/Euler-like diagrams and DAGs (directed acyclic graphs). This formalism allows to represent, by basic syntactic elements (square boxes, rounded boxes, circles, arrows, lines) the constructs used in logic programming, including function symbols. Comparing the approach of [20, 21] with the one proposed in this paper we note that the latter does not directly support aggregates that are widely used in real applications. Moreover our visual interface recalls the very well known and widely adopted QBE formalism for relational databases, and thus it has a clear advantage in familiarity with a large community that already uses this kind of visual languages. Concerning other systems that feature a full graphic tool for creating logic programs, we mention OntoDLV [19] and OntoStudio (*http://www.ontoprise.de*) that allow for specifying conjunctive queries and rules respectively and are strongly dependent on the features of the underlying logic-based ontology language; contrarily, *Visual ASP* supports all the major language features of ASP, and can be easily extended to support other languages (including OntoDLP). A practical advantage of *Visual ASP* is its similarity with the well-known QBE editors, that makes it more familiar for developers accustomed to database tools. As far as future work is concerned, we plan to extend our system by adding a reverse-rengineering tool (allowing for seamless editing of programs in both text and graphical form); and by including advanced error checking, debugging tools and rewring procedures in order to optimize the programs self. We plan also to enrich the interface with other advanced tools for improving search and visualization of logic entities. An interesting task to develop can be also the possibility of executing the programs in other solvers; the Run Configuration, for example, already allows for setting a different solver, and we planned to develop some other solver-specific output builders. An assessment of *Visual ASP* in a logic programming course of our University is also planned for verifying its applicability for teaching ASP.

References

1. Lifschitz, V.: Answer Set Planning. Proc. of the ICLP'99, Las Cruces, New Mexico, USA, The MIT Press (November 1999) 23–37
2. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM Transactions on Database Systems **22**(3) (September 1997) 364–418
3. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The First Answer Set Programming System Competition. Proc. of the LPNMR 2007.,
4. Denecher, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The Second Answer Set Programming Competition. Proc. of the LPNMR 2009.,
5. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
6. Lobo, J., Minker, J., Rajasekar, A.: Foundations of Disjunctive Logic Programming. The MIT Press, Cambridge, Massachusetts (1992)
7. Grasso, G., Iiritano, S., Leone, N., Ricca, F.: Some DLV Applications for Knowledge Management. : Proc. of the LPNMR 2009. Vol. 5753 of LNCS, Springer (2009) 591–597
8. Perri, S., Ricca, F., Terracina, G., Cianni, D., Veltri, P.: An integrated graphic tool for developing and testing DLV programs. Proc. of the SEA'07. (2007) 86–100
9. Sureshkumar, A., Vos, M.D., Brain, M., Fitch, J.: APE: An AnsProlog* Environment. Proc. of the SEA'07. (2007) 101–115
10. Brain, M., Gebser, M., Puhrer, J., Schaub, T., Tompits, H., Woltran, S.: That is Illogical Captain! The Debugging Support Tool spock for Answer-Set Programs: System Description. Proc. of the SEA'07. (2007) 71–85
11. Brain, M., De Vos, M.: Debugging Logic Programs under the Answer Set Semantics. : Proc. ASP05, Bath, UK (July 2005)
12. El-Khatib, O., Pontelli, E., Son, T.C.: Justification and debugging of answer set programs in ASP. : Proc. of the IWAD, California, USA, ACM (September 2005)
13. Young, D., Shneiderman, B.: A Graphical Filter/Flow Representation of Boolean Queries: A Prototype Implementation and Evaluation. HCI Lab. & Dep. of Computer Science (1993)
14. Proper, H.A.: Interactive Query Formulation using Query by Navigation. Asymetrix Research Report 94-4, Asymetrix Research Laboratory (1994)
15. Polyviou, S., P. Evripidou, G.S.: Query by Browsing: A Visual Query Language Based on the Relational Model and the Desktop User Interface Paradigm., University of Cyprus (2004)
16. Santucci, G., Sottile, P.A.: Query by Diagram: a Visual Environment for Querying Databases., Dip. di Informatica e Sistemistica, Universitá di Roma 'La Sapienza' (1993)
17. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TCL **7**(3) (July 2006) 499–562
18. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison Wesley (2002)
19. Ricca, F., Leone, N.: Disjunctive Logic Programming with types and objects: The DLV$^+$ System. Journal of Applied Logics **5**(3) (2007) 545–573
20. Puigsegur, J., Agustí, J.: Visual Logic Programming by means of Diagram Transformations. Proc. of the APPIA-GULP-PRODE, Conference on Declarative Programming (July 1998)
21. Puigsegur, J., Agustí, J., Robertson, D., Shorlemmer, W.M.: Visual Logic Programming through Set Inclusion and Chaining. II IA R.R. Visual Reasoning Workshop (1996)