An integrated graphic tool for developing and testing DLV programs

S. Perri, F. Ricca, G. Terracina, D. Cianni, and P. Veltri

Dipartimento di Matematica, Università della Calabria, 87036 Rende (CS), Italy
{perri,ricca,terracina}@mat.unical.it,
cianni_daniela@yahoo.it, veltri_p@libero.it

Abstract. In the last few years, significant improvements characterized state-of-the-art Answer Set Programming (ASP) systems. It is now well accepted that their applicability is becoming more and more suited for real world applications requiring complex reasoning tasks. Among the available ASP systems, DLV recently came up with a large variety of language extensions, front-ends and variants that significantly widened its range of applicability. This paper presents an integrated development environment, customized for DLV and some of its extensions, which aims to simplify both the development-and-test process and the coupling of this ASP system with DBMSs.

1 Introduction

In the last few years, the development of ASP systems like DLV [1], Smodels [2], GnT [3], and Cmodels [4] has renewed the interest in the area of non-monotonic reasoning and declarative logic programming for solving real world problems.

Moreover, the recent application of ASP systems in the areas of Knowledge Management, Security, and Information Integration [5, 6], has confirmed, on the one hand, the viability of the exploitation of disjunctive logic programming in real application settings. On the other hand, it has evidenced the lack of tools, like easy-to-use graphical environments, capable of supporting the programmers in managing large and complex projects (where the interaction with database management systems storing large amounts of data is also a crucial point).

On the contrary, imperative and object oriented programming languages are nowadays endowed with a rich set of tools allowing the user to create complex project infrastructures and to work on data residing in external databases in a quite simple way. This may discourage the usage of the declarative programming paradigm, even if it could provide the needed reasoning capabilities and, in principle, could significantly simplify the programming and maintenance tasks.

This paper provides a contribution in this setting. In fact, it presents a graphical programming environment, called VISUALDLV, which integrates several tools for developing, testing and executing logic programs (having possible interactions with external databases) in a quite simple way.

The development environment is tailored on DLV [1], an ASP system which has been recently enriched with several enhancements enabling the treatment of industrially-relevant applications [5].

The main features of VISUALDLV are:

- an easy-to-use integrated graphical environment which drives the programmer during all the phases of the implementation of projects based on DLV;
- the ability to perform both a static check (i.e., of the syntax) and a dynamic check (i.e., debugging) of the developed programs;
- the ability to help the programmer to avoid syntactic errors *during* the editing phase, with, e.g., automatic completion features;
- a specific interface which allows the programmer to graphically configure the interaction of DLV with external DBMSs (the system automatically generates the configuration options enabling this kind of interaction).

It is worth pointing out that, the presented system is a first step towards the implementation of an integrated development environment and, presently, it provides just the core functionalities outlined above. However, it has been designed in a modular way, so that further improvements can be easily integrated and existing functionalities can be extended. In the following, we first provide some background on the DLV execution modalities and debugging approach; then we present the developed system.

2 DLV execution modalities

In this section we describe DLV, a state-of-the-art ASP system [1]. In particular, we focus on three different modalities to invoke DLV: Standard Version [7], DLV^{IO} and DLV^{DB} [8]. The first one is the more common way to call DLV. Basically, the input program is supplied by means of text files and the output is provided on standard output. The second one adds to the standard version the possibility to configure basic interactions with one or more databases through ODBC. In this case, a part of the input can be imported from a DBMS, and part of the output can be exported into a DBMS. In the last one DLV tightly works with external DBMSs evaluating the programs directly in mass-memory, where the data resides (with some limitations on the supported language).

2.1 Standard version

The DLV system is an efficient engine for computing the answer sets (one, some, or all) of its input. The core language of DLV [1] is disjunctive datalog under the answer sets semantics (also known as stable model semantics [9]), which has been enriched with a number of extensions such as: true negation [10], (strong and weak) constraints [11], aggregate functions [12], and external function calls [13].

A detailed description of the DLV language is out of the scope of this paper. The interested reader is referred to [1, 12, 13]. In order to sketch its syntax, we next present a very symple example which will be also used throughout the paper to clarify the presented concepts.

Example 1. Assume that a travel agency needs to derive all the destinations reachable by an airline company, either by using its aircrafts or by exploiting code-share agreements. Moreover, the direct flights of each company are stored in facts of the form `flight(ID, FromX, ToY, Company)`, whereas the code-share agreements between

companies are stored in facts of the form `codeshare(Company1, Company2, ID);` if a code-share agreement holds between $Company1$ and the $Company2$ for the flight ID, it means that the flight ID is actually provided by an aircraft of $Company1$, but it can be considered also carried out by $Company2$. The DLV program that can derive all the connections is:

$$\begin{array}{lll} destinations(FromX, ToY, Company) & :- & flight(ID, FromX, ToY, Company). \\ destinations(FromX, ToY, Company) & :- & flight(ID, FromX, ToY, Company2), \\ & & codeshare(Company2, Company, ID). \\ destinations(FromX, ToY, Company) & :- & destinations(FromX, T2, Company), \\ & & destinations(T2, ToY, Company). \end{array}$$

□

In the standard execution modality, the input of DLV is stored in one or more text files. Those files are first parsed to create the internal data structures, which are then stored in main memory where the entire computation is performed.

The answer sets computation can be split in three steps. In the first step (performed by the Grounder) the variables present in the input program are eliminated, generating the so-called *ground instantiation* of the program, which is a (usually much smaller) subset of all syntactically constructible instances of the rules of the program having precisely the same stable models. Then, the nondeterministic part of the computation is performed on this simplified ground program by the Model Generator (MG) module. The MG searches for candidate answer sets by employing a Davis-Putnam procedure similar to the ones employed in SAT-solvers. Basically, MG builds the answer set by tentatively assuming the truth of the literals, and "propagating" the deterministic consequences of those assumptions by applying suitable inference rules. If an assumption (also called choice point) leads to an inconsistency the system goes back to modify exactly those choice points that caused the inconsistency. The process continues until a candidate answer set is found or all the possible choices have been tried. Finally, each candidate answer set (which has been found by the MG) is analyzed by the Model Checker (MC), which verifies its stability (w.r.t. the Gelfond-Lifschitz transformation [9]). If the stability check succeeds then the system outputs the answer set; otherwise the MG continues its search by modifying the assumptions which caused the stability check failure.

2.2 DLV^{IO}

In this execution modality, the system allows input facts to be (possibly complex) views on database tables, which are stored in different DBMSs; moreover, it allows (parts of) the results of the execution to be exported in database relations. The logic program is evaluated completely in main-memory with the same evaluation strategy employed in the standard version; this allows DLV^{IO} to support completely the DLV language and all its extensions (like strong and weak constraints, aggregate functions, external function calls, etc.), with only minor restrictions (see below).

Intuitively, DLV^{IO} can be exploited when the user has to perform complex reasoning tasks but the data is available in database relations, or the output must be permanently stored in a database for further elaborations.

In order to perform these tasks, two built-in commands are introduced in the DLV syntax, namely the #import and the #export commands:

#import(databasename,“username”,“password”,“query”,predname, typeConv).
#export(databasename,“username”,“password”,predname,tablename).

An #import command retrieves data from a table “row by row” through the *query* specified by the user in SQL and creates one atom for each selected tuple. The name of each imported atom is set to *predname*, and is considered as a fact of the program. *typeConv* specifies the data conversion rules to be applied for converting database types into DLV data types.

The #export command generates a new tuple into *tablename* for each new truth value derived for *predname* by the program evaluation. Both commands require that an ODBC connection with *databasename* has been previously set up.

Note that if a program contains at least one #export command, the system will be able to compute only the first answer set.

A description of DLV^{IO} and its functionalities can be found in [8]; moreover, the system, along with a manual and some examples, are available for download at the address `http://www.mat.unical.it/terracina/dlvdb`.

Example 2. Consider the scenario introduced in Example 1, and assume that the information about direct flights (facts `flight`) are stored in a relation `flight_rel(ID, FromX, ToY, Company)` of the database `dbAirports`; whereas the code-share agreements between companies (facts `codeshare`) are stored in a relation `codeshare_rel (Company1, Company2, ID)` of another database `dbCommercial`. Finally, assume that, for security reasons, travel agencies are not allowed to directly access the databases `dbAirports` and `dbCommercial`, and, consequently, it is necessary to store the output result in a relation `composedCompanyRoutes` belonging to another database `dbTravelAgency` (accessible by the travel agencies).

To this end we must add the following directives to the DLV program of Example 1:

```
#import(dbAirports,“airportUser”,“airportPasswd” , “SELECT * FROM flight_rel”, flight,
                                                    type : U_INT, Q_CONST, Q_CONST, Q_CONST).
#import(dbCommercial,“commUser”,“commPasswd” , “SELECT * FROM codeshare_rel”,
                                                    codeshare, type : Q_CONST, Q_CONST, U_INT).
#export(dbTravelAgency,“agencyName”,“agencyPasswd”, destinations, composedCompanyRoutes).
```

The first two commands maps the predicate $flight$ to the relation `flight_rel` of `dbAirports`, and the predicate $codeshare$ to the relation `codeshare_rel` of `dbCommercial`; the last one maps the predicate $destinations$ to the relation `composedCompanyRoutes` of `dbTravelAgency`. □

2.3 DLV^{DB}

The user needing this execution modality has its data stored in (possibly distributed) database tables and wants to carry out some reasoning on them; however the amount of such data, or the amount of facts the reasoning generates on them, is such that the evaluation can not be carried out in main-memory. Then, the only way out is to evaluate the program directly in mass-memory.

Three main peculiarities characterize the system in this execution modality: *(i)* its ability to evaluate logic programs directly and completely on databases with a very

```
Auxiliary-Directives ::= Init-section [Table-definition]+ [Query-Section]?
                         [Final-section]*
Init-Section ::=USEDB DatabaseName:UserName:Password [System-Like]?.
Table-definition ::=
   [USE TableName [( AttrName [, AttrName]* )]? [AS ( SQL-Statement )]?
   [FROM DatabaseName:UserName:Password]?
   [MAPTO PredName [( SqlType [, SqlType]* )]? ]?.
   |
   CREATE TableName [( AttrName [, AttrName]* )]?
   [MAPTO PredName [( SqlType [, SqlType]* )]? ]?
   [KEEP_AFTER_EXECUTION]?.]
Query-Section ::= QUERY TableName.
Final-section ::=
   [DBOUTPUT DatabaseName:UserName:Password.
   |
   OUTPUT [APPEND | OVERWRITE]? PredName [AS AliasName]?
   [IN DatabaseName:UserName:Password.]
System-Like ::= LIKE [POSTGRES | ORACLE | DB2 | SQLSERVER | MYSQL]
```

Fig. 1. Grammar of the auxiliary directives.

limited usage of main-memory resources, *(ii)* its capability to map program predicates to (possibly complex and distributed) database views, and *(iii)* the possibility to easily specify which data is to be considered as input or as output for the program. As for DLV^{IO}, also in DLV^{DB} access to DBMSs is carried out through ODBC.

Currently, DLV^{DB} does not fully support the DLV language. In particular, only disjunction free stratified programs (possibly with built-ins and aggregate functions) are supported. However, it allows handling significantly greater amounts of data w.r.t. DLV and DLV^{IO} with also important improvements in query answering times.

In order to properly carry out the evaluation, this execution modality requires some explicit specifications for the mappings between input and output data and program predicates, as well as proper indications for the temporary relations possibly needed for the mass-memory evaluation. The grammar in which these directives must be expressed is shown in Figure 1.

Intuitively, the user must specify a working database in which the system has to perform the evaluation (the `Init-Section` in the grammar). Moreover, he can specify a set of table definitions, each of which must be mapped into one of the program predicates. Facts can reside on separate databases or they can be obtained as views on different tables. Attribute type declaration is needed only if the program must carry out arithmetic operations on them. `USE` and `CREATE` directives can be exploited to specify input and output data. Finally, the user can choose to copy the entire output of the evaluation or parts thereof in a database different from the working one by some `OUTPUT` directives.

Example 3. Consider again the scenario introduced in Examples 1 and 2, and suppose that, due to a huge size of input data, it is not possible to perform the evaluation in main-memory. In order to evaluate the program in mass-memory (on a DBMS), the auxiliary directives shown in Figure 2 should be used. Here, the first line is the `Init-Section` and states that the evaluation must be carried out in a database named *dlvdb*. The two `USE` directives are equivalent to (but more precise than) the #import commands of Example 2. Finally, the `OUTPUT` directive is equivalent to the #export command of Example 2. □

```
USEDB dlvdb:myname:mypasswd.
USE flight_rel (ID, FromX, ToY, Company) FROM dbAirports:airportUser:airportPasswd
MAPTO flight (integer, varchar(255), varchar(255), varchar(255)).
USE codeshare_rel (Company1, Company2, ID) FROM dbCommercial:commUser:commPasswd
MAPTO codeshare (varchar(255), varchar(255), integer).
CREATE destinations_rel (From, To, Company)
MAPTO destinations (varchar(255), varchar(255), varchar(255)) KEEP_AFTER_EXECUTION.
OUTPUT  destinations AS composedCompanyRoutes IN
        dbTravelAgency:agencyName:agencyPasswd.
```

Fig. 2. Auxiliary directives for Example 1.

3 Debugging DLV Programs

Debugging is the process of locating and fixing known errors (which are commonly called "bugs") on both computer programs and hardware devices. Unfortunately, debugging is difficult to be carried out due to the extremely high number of causes for a bug. As a consequence, techniques and tools (debuggers) helping the programmer to deal with this problem must be associated with each programming language.

However, while debugging an imperative program can be carried out by monitoring its execution (usually with a step-by-step strategy), debugging a program with a declarative semantics must follow a completely different approach. As an example, the notion of "unexpected" behaviour is substantially different comparing DLV and C++ programs. The absence of an intuitive operational semantics makes it harder to understand *why* the results of a declarative program are not the expected ones.

Intuitively, a bug in a DLV program P is a difference between what is actually modelled by P and what the programmer was planning to model with P. Examples of bugs of a DLV program are an unexpected number of answer sets or the presence/absence of a literal in a specific answer set.

The reasoning above clearly points out that, in a declarative programming setting, even what must be meant for debugging is not obvious (as also pointed out by [14, 15]). In what follows, we consider that a debugger for DLV must allow the programmer to understand the "reasons" which "caused" the derivation of the various literals in an answer set or, in absence of it, to have a justification for the failure.

The DLV debugger we developed in this work uses information collected during the program evaluation, especially in the Model Generation phase (see Section 2.1).

In more detail, the MG module of DLV, introduced in Section 2.1, exploits a so-called backjumping (or non-chronological backtracking) technique (described in [16]), based on the ability to detect and to undo, during the backtracking phase, the choices directly causing an inconsistency. This technique constructs a data structure, called *Reason Table*, which stores for each literal the choices implying its presence/absence in the current (partial) answer set. The Reason Table is built (and updated) during the search, according to the reason calculus technique presented in [16]. The information stored in the Reason Table is directly used in the debugging modality to justify the presence/absence of a literal in an answer set (or the unsatisfiability of the program). Due to space limitations we cannot describe here the whole process of reasons computation; rather, we try to give an intuition with an example.

Example 4. Let P be the following program

```
a ∨ b.  c :- a.  d :- b.
```

At a certain point of the MG computation, a is chosen as true and its truth value is propagated trough the program rules, deriving truth values for other atoms. Obviously, in this case, c and $not\ b$ are derived as true. Thus, intuitively, we set in the Reason Table a as reason for c. But, what about the reason of a? We say that a is a choice and that its reason is itself. □

When DLV starts in debug mode, the main computation stops as soon as an answer set has been found, or when it is detected that no answer set can be found, and the system waits for some user command. The available commands are: *why*, *why_unstable*, *next_model*, *print_model*, *print_instantiation*, and *quit*. The first one can be used to know the choices implying a literal L (it can be read as "why is L in current model?"); the second command can be used to investigate why a program is unsatisfiable. In this case, the system reports the reason causing the last inconsistency found during the search. The remaining commands can be used to ask the system for looking for another answer set, printing the current answer set, printing the ground instantiation, and stopping the system. Currently, DLV^{DB} does not support debugging, because it exploits a completely different (mass-memory based) evaluation strategy. The next example shows the usage of commands *why*, and *why_unstable*.

Example 5. Consider again the program P of Example 4. In order to know why literal c appears in one of the answer sets of P we can use the command *why (c)*. This command will return a indicating that c is in the current model because of the choice of a.

Now, let add to P the following two strong constraints

```
:- c, not d.  :- d, not c.
```

Clearly, the program has no answer set. In fact, if we choose a as true the first constraint is violated (i.e. a caused the inconsistency, and this can be easily obtained by looking in the reason table); similarly, if we choose b the second constraint is violated (i.e. b caused the inconsistency). Assuming that the last choice actually made during the computation is b then the command *why_unstable* returns b. □

4 System Description

4.1 Functionalities

The functionalities implemented in VISUALDLV borrow several ideas from the wide variety of well known integrated tools available for developing programs with imperative languages (such as C++ and Java). The interesting innovation is the adaptation of such ideas to the declarative world, providing a wide set of features to assist the user in developing, configuring and testing DLV *projects*.

The main functionalities provided graphically by VISUALDLV are:

- *Project definition*. It allows to gather in a single logical unit several DLV program files, auxiliary directives and configuration options.
- *Automatic completion*. The editing of DLV programs and auxiliary directives is simplified by this functionality which suggests the user how to complete the portions of programs he is writing.

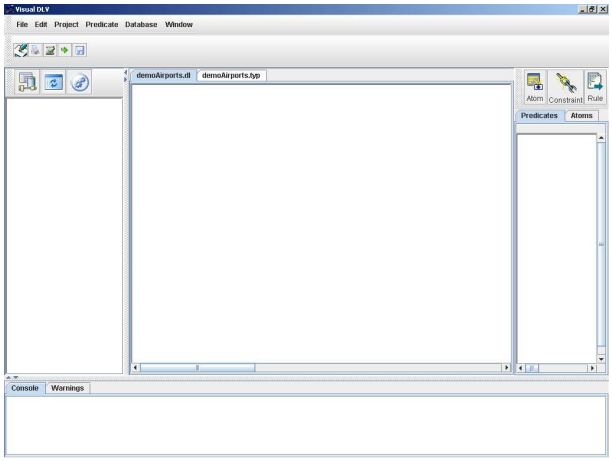


Fig. 3. The general structure of the system interface.

- *Dynamic syntax checking*. This functionality checks the syntactical correctness of the program during its development, warning the user in case of errors.
- *Configuration of the interactions with the databases*. It allows the user to easily, and graphically, specify which input data resides in external databases, and which parts of the program output must be permanently stored in a database.
- *Configuration of the execution*. It allows to select the execution options for DLV.
- *Presentation of results*. The output of the program (either its answer sets, or the database table contents) can be visualized within the same environment.
- *Debugging*. This functionality allows the user to interact with DLV in order to understand why a program does not produce the expected output.

In the following, we describe in more detail system's functionalities, using some screen-shots of the system to show how it works.

Interface overview

The general structure of the system interface is illustrated in Figure 3. The central area is the main editing area, where DLV programs and auxiliary directives can be typed. The left part of the interface is dedicated to the database management; in particular, as it will be more clear in the following, the list of the databases included in the project, as well as some database management features are located in this portion of the interface. The right part is dedicated to providing the summary of the concepts (atoms and predicates) defined in the currently open DLV programs and can be used as a support for editing. The bottom part contains two panels allowing the system to provide messages to the user, namely a *warning* panel, collecting all warning messages, and a *console* panel showing the output of the programs. Finally, in the upper part of the interface, classical menus and toolbars allow the user to access all the features of the system.

Project definition

Declarative programming allows specifying in a natural way complex problems; it is true. However, when the application scenario is composed by several sub-problems or

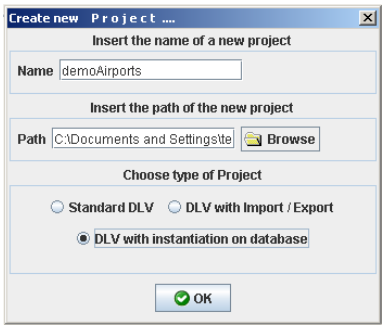


Fig. 4. The creation of a new project.

it requires the application of different reasoning modules the user can be easily involved with several program components, which should be developed and tested separately, but which logically belong to the same project.

Moreover, the various kinds of DLV execution modality described in Section 2 may require different kinds of interaction of the user with the GUI (e.g., the standard DLV version does not require information about external databases, which, on the contrary, is necessary for DLV^{DB} and DLV^{IO}) and different kinds of invocation parameter.

In order to face these issues, our system introduces the notion of *project*, i.e. a collection of DLV programs, auxiliary directives, database connections and configuration options defining, as a whole, a complete project.

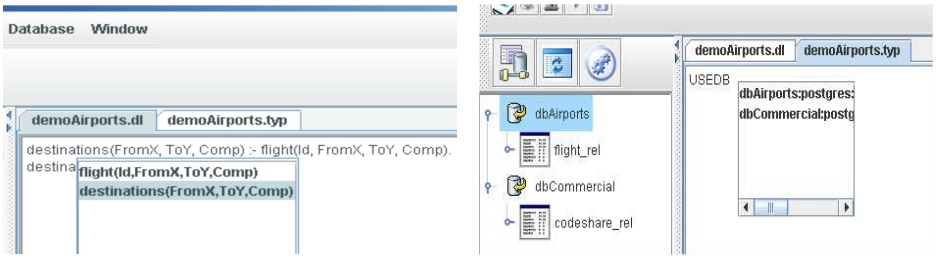
Figure 4 shows the interface allowing the definition of a new project. A project is characterized by a *name*; all its data is put in a folder having this name. Finally, the user has to specify the project type, which determines the DLV execution modality to exploit, and the kinds of interaction expected between the user and the system. In Figure 4 the user is choosing to create a DLV^{DB} project with name `demoAirports`.

Automatic completion

Following the success of other systems for imperative programming (like Visual C++, Eclipse, etc.) our system provides a functionality that suggests the user how to complete the portions of programs he is writing, just during the typing.

It is worth pointing out that imperative languages have both explicit data typing and fixed language constructs; this allows a quite straightforward definition of lists of legal keywords or of user-defined variables to be used in the automatic completion facilities.

On the contrary, declarative languages in general, and DLV in particular, do not comprise such features and, consequently, it is less evident what the automatic completion functionality must suggest to the user. In our system, the automatic completion works on what has been “declared” by the programmer up to that time; in other words, it works on the list of atoms previously specified in the program. Figure 5a illustrates this functionality; each time a rule is typed, it is parsed and the atoms it contains are added to the list of atoms defined by the user. Then, when the user is writing a new rule, the system shows a pop-up window where an atom is highlighted if its prefix corresponds to what the user is typing. Note that this functionality significantly simplifies the development of complex programs constituted by several rules and atoms. As an example, consider the program of Example 1 and assume that the user (without the support of



(a) Program completion (b) Auxiliary directives completion

Fig. 5. The automatic completion feature.

the automatic completion) types `destination` instead of `destinations`; there is no way for an automatic checker to understand whether the user intention was to define a new concept `destination` or if he just mistyped the predicate name `destinations`. Helping to prevent these kinds of errors *during* the programming phase, may allow the user to save a lot of time in the testing phase!

The same functionality is provided by the system also for the definition of the auxiliary directives, necessary for DLV^{DB} projects. In this case, the automatic completion is more context sensitive, because the auxiliary directives are characterized by a precise grammar (see Figures 1 and 5b).

Dynamic syntax checking

When the user types a rule, it is parsed by the parsing module and its syntactical correctness is verified. If an error is identified, a message is displayed in the warning panel. Note that these warning messages do not block the user interaction; this is important in order to let the system accommodate also to further extensions of the DLV language currently not expected by the parser.

Presently, only the correctness of the syntax is checked; however, we plan to extend this feature to carry out more refined checking tasks. As an example, one of the most frequent errors in developing datalog rules is the mistyping of a variable name involved in a join; in this case, the rule is syntactically correct, but it contains a semantic error. If the system would warn the user about the presence of variables in some atom not joined with any other atom of the rule, the user could easily check whether this situation is wanted or it is the result of a mistyped variable name.

Interaction with external databases

As pointed out in the previous section, DLV^{IO} and DLV^{DB} extend the capabilities of DLV allowing various kinds of interactions with external databases via ODBC. Our system provides various functionalities aiming to simplify the correct configuration of DLV^{IO} and DLV^{DB}. In more detail, it provides both functionalities for accessing, querying and manipulating data residing in external databases, and functionalities for graphically compiling the auxiliary directives.

Figure 6 illustrates some of the capabilities for accessing and querying data residing in external databases. Each database is accessed via ODBC and, consequently, in order to access it, the database name, the user and password for it must be supplied. For each opened database, the list of tables and their structure are shown. Moreover, the

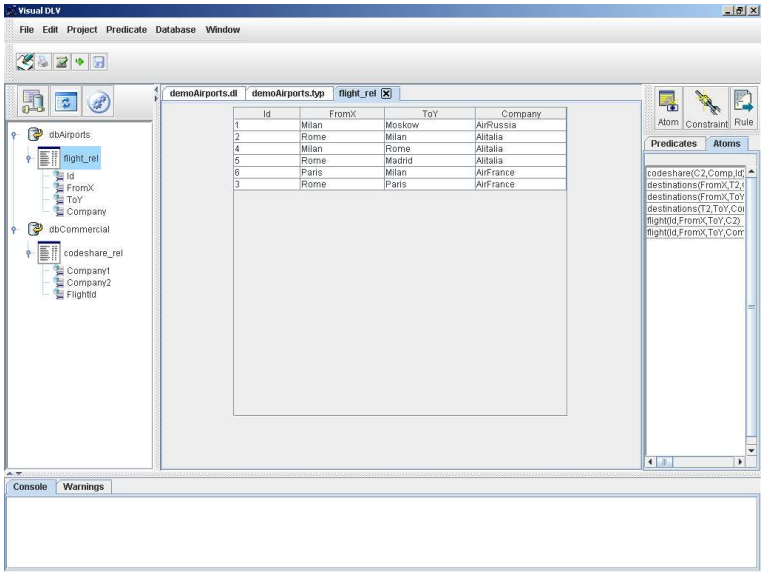


Fig. 6. Interaction with external databases.

user can visualize the content of the various tables (in the Figure, the content of table `flight_rel` is shown). Finally, other editing operations can be carried out, such as the execution of SQL statements (including CREATE or ALTER statements) and table deletion. In other words, the system provides a restricted (but common) set of database management features.

Concerning the support in compiling auxiliary directives, even if DLV^{DB} provides several simplifications in their specification (see the manual on the system's web site), writing them by hand could by quite hard for a non specialist. For this reason, our system provides both the automatic completion facility and an automatic generation feature for such directives. Figures 5b and 7 graphically show both of them.

In particular, Figure 5b illustrates an example of automatic completion for the `USEDB` directive; here, the grammar specifies that after the `USEDB` keyword the database connection parameters must be specified. Then, the system suggests such information, based on the databases currently open in the project.

Figure 7 illustrates the form to automatically create a `USE` directive. It can be activated with a right-click on the table that must be "used" as input in the program; the system automatically retrieves from the database all the information necessary to generate the directive. Moreover, it provides the user with a preview of it, in order to let him check the correctness. `CREATE` directives can be generated analogously; in this case, the user must select one of the predicates listed in the right part of the main interface.

Configuration of project execution

The execution of a DLV program can be often a tricky task for a non specialist; in fact, the wide range of extensions developed for DLV in the last ten years produced a wide set of options that can be specified within the command line. Our system deals with this situation providing the comprehensive set of DLV options in a user-friendly

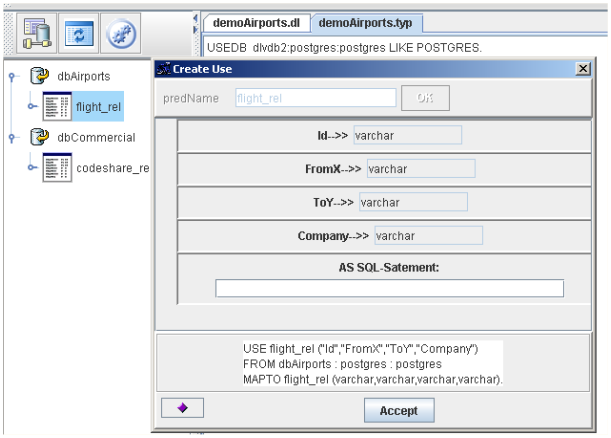


Fig. 7. Automatic generation of auxiliary directives.

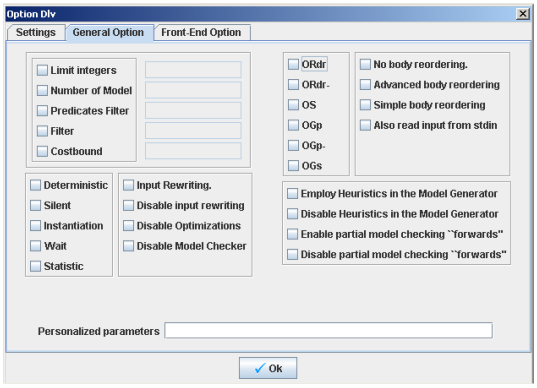


Fig. 8. Options for the execution of DLV programs.

fashion, as shown in Figure 8. The user can choose graphically the needed options and the system automatically generates the corresponding portion of command line. The system is also open to further extensions of DLV allowing the user to input personalized execution options. This configuration phase can be carried out once and for all the runs of the current project.

After this, when the user wants to run his project, the system proposes him the list of program files currently active, and the user can choose those ones that must be included in the current run. Moreover, an expert user can personalize the command line proposed by the system, if he think it is necessary.

Presentation of results

During the execution of DLV (resp., DLV^{IO}, DLV^{DB}) the output is redirected to the *console* panel, located in the lower part of the interface (see Figure 3) in such a way that the user can check the program output from the same environment. Moreover, the output redirected to database tables in DLV^{IO} or DLV^{DB} can be analyzed as illustrated in Figure 6.

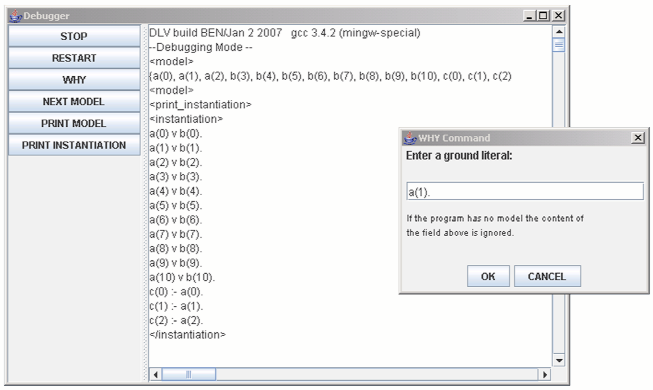


Fig. 9. The Debugger graphical interface.

Debugging of a Program

The debugging of a program is, in general, a crucial task in the development of an application. In VISUALDLV it can be carried out through a graphic interface (see Figure 9); this is promptly displayed when the user asks to run DLV in debugging mode. In this case, VISUALDLV transparently adds to the invocation parameters the "-debug" option. Figure 9 shows the first model found by DLV for the program:

```
a(X) ∨ b(X) :- #int(X).  c(X):-a(X), X<3.  #maxint(10).
```

and the answer of the debugger to the user request "print instantiation".

All the debugging commands available for the user can be activated with the menus on the left side of the interface, as shown in Figure 9; these are automatically translated and forwarded to DLV in the proper format (as XML tags).

Note that, the debugger interface is a non-modal window, so that the programmer can contemporarily look at the input program during a debugging session (without the need to stop the debugger). However, the debugger must be re-launched after any modification to the input program is applied.

4.2 Architecture

The architecture of the system is shown in Figure 10. The Graphical User Interface (GUI) allows the user to access all the system's functionalities. These are implemented by five main modules.

The *Parser*, is responsible of translating DLV programs and auxiliary directives, taken both from the user interface and by pre-existing files, in suitable internal data structures. These are currently used for the automatic completion and the dynamic syntax checking features, but can be the basis also for more refined functionalities (e.g., a graphical representation of the dependencies between program predicates, etc.).

The *Editor* module implements classical file editing operations and provides the automatic completion feature.

The *DB Connection Handler*, manages all the interactions of the system with the external databases, such as ODBC connections, table contents viewing, database query-

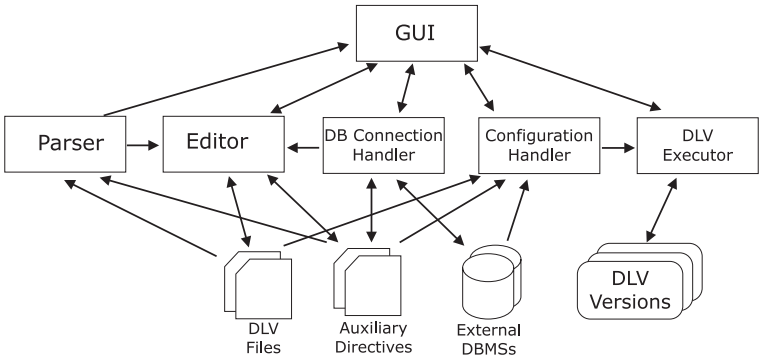


Fig. 10. The general architecture of the system.

ing and manipulation, etc. Moreover, it interacts with the GUI for the generation of the auxiliary directives.

The *Configuration Handler* is responsible of storing and managing all configuration information of the current project. In particular, it takes into account both the project typology and the options specified by the user through the interface, to compose the correct command line needed to invoke DLV (resp., DLV^{IO}, DLV^{DB}).

The *DLV Executor* invokes the proper versions of DLV (including the debugging version) and redirects the corresponding output (possibly reformatted) to the GUI.

Note that, the proposed tool might be extended in order to support other flavors of ASP, e.g, the Smodels language. This can be done by adding both specialized parser and executor modules[1].

5 Conclusions

In this paper we have presented a graphic integrated environment, called VISUALDLV, for the development of DLV applications. Our system represents a first step toward the implementation of an integrated and complete suite of tools for a DLV developer. It integrates many interesting features which help the programmers during the development phases: editing, configuration, interaction with external DBMS, debugging, and deployment. We are currently working on several improvements of the existing functionalities (e.g. enabling drag-and-drop facilities for the generation of DLV^{DB} directives, etc.), and we are planning the introduction of additional capabilities, such as a graphical representation of program dependencies and a tree view of answer sets.

References

1. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL **7**(3) (2006) 499–562
2. Niemelä, I., Simons, P., Syrjänen, T.: Smodels: A System for Answer Set Programming. In: NMR'2000 (2000)

[1] In the interface, we can deal with that by adding a new kind of project, let say Smodels project.

3. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In: Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7). LNCS 2923
4. Lierler, Y.: Cmodels for Tight Disjunctive Logic Programs. In: W(C)LP 19th Workshop on (Constraint) Logic Programming, Ulm, Germany. Ulmer Informatik-Berichte, Universität Ulm, Germany (2005) 163–166
5. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kałka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszkis, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In: Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005), Baltimore, Maryland, USA, ACM Press (2005) 915–917
6. Massacci, F.: Computer Aided Security Requirements Engineering with ASP Nonmonotonic Reasoning, ASP and Constraints, Seminar N 05171. Dagstuhl Seminar on Nonmonotonic Reasoning, Answer Set Programming and Constraints (2005)
7. Faber, W., Pfeifer, G.: `DLV` homepage (since 1996) `http://www.dlvsystem.com/`.
8. Terracina, G., Leone, N., Lio, V., Panetta, C.: Adding efficient data management to logic programming systems. In: Proc. of 16th International Symposium on Methodologies for Intelligent Systems (ISMIS 2006), Bari, Italy, Lecture Notes in Artificial Intelligence (4203), (2006) 524–533
9. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: Logic Programming: Proceedings Fifth Intl Conference and Symposium, Cambridge, Mass., MIT Press (1988) 1070–1080
10. Buccafurri, F., Leone, N., Rullo, P.: Stable Models and their Computation for Logic Programming with Inheritance and True Negation. JLP **27**(1) (1996) 5–43
11. Buccafurri, F., Leone, N., Rullo, P.: Enhancing Disjunctive Datalog by Constraints. IEEE TKDE **12**(5) (2000)
12. Calimeri, F., Faber, W., Leone, N., Perri, S.: Declarative and Computational Properties of Logic Programs with Aggregates. In: Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05). (2005) 406–411
13. Calimeri, F., Ianni, G.: External sources of computation for Answer Set Solvers. In: LPNMR'05. LNCS 3662
14. Brain, M., Vos, M.D.: Debugging Logic Programs under the Answer Set Semantics. In: Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation, Bath, UK (2005)
15. El-Khatib, O., Pontelli, E., Son, T.C.: Justification and debugging of answer set programs in ASP. In: Proceedings of the Sixth International Workshop on Automated Debugging, California, USA, ACM (2005)
16. Ricca, F., Faber, W., Leone, N.: A Backjumping Technique for Disjunctive Logic Programming. AI Communications **19**(2) (2006) 155–172