DLV – A System for Declarative Problem Solving*

Thomas Eiter and **Wolfgang Faber** and **Christoph Koch**

Nicola Leone and **Gerald Pfeifer**

Institut für Informationssysteme, TU Wien
Favoritenstrasse 9-11
A-1040 Wien, Austria
{leone,pfeifer}@dbai.tuwien.ac.at, {eiter,faber}@kr.tuwien.ac.at

European Organization for Nuclear Research
CERN EP Division
CH-1211 Geneva, Switzerland
christoph.koch@cern.ch

Abstract

DLV is an efficient logic programming and non-monotonic reasoning (LPNMR) system with advanced knowledge representation mechanisms and interfaces to classic relational database systems.

Its core language is disjunctive datalog (function-free disjunctive logic programming) under the Answer Set Semantics with integrity constraints, both default and strong (or explicit) negation, and queries. Integer arithmetics and various built-in predicates are also supported.

In addition **DLV** has several frontends, namely brave and cautious reasoning, abductive diagnosis, consistency-based diagnosis, a subset of SQL3, planning with action languages, and logic programming with inheritance.

General Information

Currently **DLV** is available in binary form for various platforms (sparc-sun-solaris2.6, alpha-dec-osf4.0, i386-linux-elf-gnulibc2, i386-pc-solaris2.7, and i386-unknown-freebsdelf3.3 as of this writing) and it is easy to build **DLV** on further platforms.

Including all frontends, **DLV** consists of around 25000 lines ISO C++ code plus several scanners and parsers written in lex/flex and yacc/bison, respectively. **DLV** is being developed using GNU tools (GCC, flex, and bison) and is therefore portable to most Unix-like platforms. Additionally, the system has been successfully built with proprietary compilers such as those of Compaq and SCO.

For up-to-date information on the system and a full manual please refer to the project homepage (Faber & Pfeifer since 1996), where you can also download **DLV**.

*This work was supported by FWF (Austrian Science Funds) under the projects P11580-MAT and Z29-INF.

Description of the System

Kernel Language

The kernel language of **DLV** is disjunctive datalog extended with strong negation under the answer set semantics (Eiter, Gottlob, & Mannila 1997; Gelfond & Lifschitz 1991).

Syntax Strings starting with uppercase letters denote variables, while those starting with lower case letters denote constants. A *term* is either a variable or a constant. An *atom* is an expression $p(t_1, \ldots, t_n)$, where p is a *predicate* of arity n and $t_1, \ldots, t_n$ are terms. A *literal* l is either an atom a (in this case, it is *positive*), or a negated atom $-a$ (in this case, it is *negative*).

Given a literal l, its *complementary* literal is defined as $-a$ if $l = a$ and a if $l = -a$. A set L of literals is said to be *consistent* if for every literal $l \in L$, its complementary literal is not contained in L.

In addition to literals as defined above, **DLV** also supports built-ins, like `#int`, `#succ`, <, `+`, and `*`. For details, we refer to our full manual (Faber & Pfeifer since 1996).

A *disjunctive rule* (*rule*, for short) r is a formula

$$a_1 \ \mathtt{v} \ \cdots \ \mathtt{v} \ a_n \ \mathtt{:-} \ b_1, \cdots, b_k, \ \mathtt{not} \ b_{k+1}, \cdots, \ \mathtt{not} \ b_m.$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are literals, $n \geq 0$, $m \geq k \geq 0$, and `not` represents *negation-as-failure* (or *default negation*). The disjunction $a_1 \mathtt{v} \cdots \mathtt{v} a_n$ is the *head* of r, while the conjunction $b_1, ..., b_k$, $\mathtt{not}\ b_{k+1}, ...,$ $\mathtt{not}\ b_m$ is the *body* of r. A rule without head literals (i.e. $n = 0$) is usually referred to as *integrity constraint.* If the body is empty (i.e. $k = m = 0$), we usually omit the "`:-`" sign.

We denote by $H(r)$ the set of literals in the head, and by $B(r) = B^+(r) \cup B^-(r)$ the set of the body literals, where $B^+(r) = \{b_1, \ldots, b_k\}$ and $B^-(r) = \{b_{k+1}, \ldots,$

$b_m\}$ are the sets of positive and negative body literals, respectively.

A *disjunctive datalog program* $\mathcal{P}$ is a finite set of rules.

Semantics **DLV** implements the consistent answer sets semantics which has originally been defined in (Gelfond & Lifschitz 1991).[1]

Before we are going to define this semantics, we need a few prerequisites. As usual, given a program $\mathcal{P}$, $U_{\mathcal{P}}$ (the *Herbrand Universe*) is the set of all constants appearing in $\mathcal{P}$ and $B_{\mathcal{P}}$ (the *Herbrand Base*) is the set of all possible combinations of predicate symbols appearing in $\mathcal{P}$ with constants of $U_{\mathcal{P}}$ possibly preceded by $-$, in other words, the set of ground literals constructible from the symbols in $\mathcal{P}$.

Given a rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of $U_{\mathcal{P}}$; $Ground(r)$ is also called the *Ground Instantiation* of r. In a similar way, given a program $\mathcal{P}$, $Ground(\mathcal{P})$ denotes the set $\bigcup_{r \in \mathcal{P}} Ground(r)$. For programs not containing variables $\mathcal{P} = Ground(\mathcal{P})$ holds.

For every program $\mathcal{P}$, we define its *answer sets* using its ground instantiation $Ground(\mathcal{P})$ in two steps, following (Lifschitz 1996): First we define the answer sets of positive programs, then we give a reduction of general programs to positive ones and use this reduction to define answer sets of general programs.

An interpretation I is a set of literals. A consistent interpretation $I \subseteq B_{\mathcal{P}}$ is called *closed under* a positive, i.e. `not`-free, program $\mathcal{P}$, if, for every $r \in Ground(\mathcal{P})$, $H(r) \cap I \neq \emptyset$ whenever $B(r) \subseteq I$. I is an *answer set* for a positive program $\mathcal{P}$ if it is minimal w.r.t. set inclusion and closed under $\mathcal{P}$.

The *reduct* or *Gelfond-Lifschitz transform* of a general ground program $\mathcal{P}$ w.r.t. a set $X \subseteq B_{\mathcal{P}}$ is the positive ground program $\mathcal{P}^X$, obtained from $\mathcal{P}$ by deleting all rules $r \in \mathcal{P}$ for which $B^-(r) \cap X \neq \emptyset$ holds, and deleting the negative body from the remaining rules.

An answer set of a general program $\mathcal{P}$ is a set $X \subseteq B_{\mathcal{P}}$ such that X is an answer set of $Ground(\mathcal{P})^X$.

Application Frontends

In addition to its kernel language, **DLV** provides a number of application frontends that show the suitability of our formalism for solving various problems from the areas of Artificial Intelligence, Knowledge Representation and (Deductive) Databases.

- The *Brave and Cautious Frontends* are simple extensions of the normal mode, where in addition to a disjunctive datalog program the user specifies a conjunction of literals (a query) and **DLV** checks whether this query holds in any respectively all answer sets of the program.
- The *Diagnoses Frontend* implements both abductive diagnosis (Poole 1989; Console, Theseider Dupré, & Torasso 1991), adapted to the semantics of logic programming (Kakas, Kowalski, & Toni 1993; Eiter, Gottlob, & Leone 1997), and consistency-based diagnosis (Reiter 1987; de Kleer, Mackworth, & Reiter 1992) and supports general diagnosis as well as single-failure and subset-minimal diagnosis.
- The *SQL3 Frontend* is a prototype implementation of the query interface of the SQL3 standard that has been approved by ISO last year.
- The *Inheritance Frontend* extends the kernel language of **DLV** with objects, and inheritance (Buccafurri, Faber, & Leone 1999). This extension allows us to naturally represent inheritance and default reasoning with (multi-level) exceptions, providing a natural solution also to the frame problem.
- Finally, the *Planning Frontend* implements a new logic-based planning language, called $\mathcal{K}$, which is well suited for planning under incomplete knowledge.

Architecture

An outline of the general architecture of our system is depicted in Fig.1.

The heart of the system is the **DLV** *core*. Wrapped around this basic block are frontend preprocessors and output filters (which also do some post-processing for frontends). The system takes input data from the user (mostly via the command line) and from the file system and/or database systems.

Upon startup, input is possibly translated by a frontend. Together with relational database tables, provided by an Oracle database, an Objectivity database, or ASCII text files, the *Intelligent Grounding Module*, efficiently generates a subset of the grounded input program that has exactly the answer sets as the full program, but is much smaller in general.

After that, the Model Generator is started. It generates one answer set candidate at a time and verifies it using the Model Checker. Upon success, filtered output is generated for the answer set. This process is iterated until either no more answer sets exist or an explicitly specified number of answer sets has been computed.

Not shown in Fig.1 are various additional data structures, such as dependency graphs.

Applying the System

Methodology

The core language of **DLV** can be used to encode problems in a highly declarative fashion, following a "**Guess&Check**" paradigm. We will first describe this paradigm in an abstract way and then provide some concrete examples. We will see that several problems, also problems of high computational complexity, can be solved naturally in **DLV** by using this declarative programming technique. The power of disjunctive rules allows one to express problems, which are even more

[1] Note that we only consider *consistent answer sets*, while in (Lifschitz 1996) also the inconsistent set of all possible literals is a valid answer set.

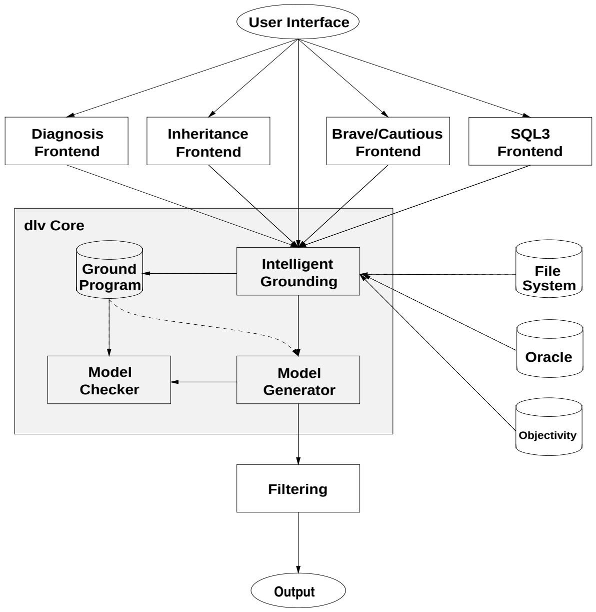

Figure 1: Overall architecture of **DLV**.

complex than NP uniformly over varying instances of the problem using a fixed program.

Given a set F_I of facts that specify an instance I of some problem P, a **Guess&Check** program $\mathcal{P}$ for P consists of the following two parts:

Guessing Part The guessing part $G \subseteq \mathcal{P}$ defines the search space, in a way such that answer sets of $G \cup F_I$ represent "solution candidates" of I.

Checking Part The checking part $C \subseteq \mathcal{P}$ tests whether a solution candidate is in fact a solution, such that the answer sets of $G \cup C \cup F_I$ represent the solutions for the problem instance I.

In general, we may allow both G and C to be arbitrary collections of rules in the program, and it may depend on the complexity of the problem which kind of rules are needed to realize these parts (in particular, the checking part); we defer this discussion to a later point in this section.

Without imposing restrictions on which rules G and C may contain, in the extremal case we might set G to the full program and let C be empty, i.e., all checking is moved to the guessing part such that solution candidates are always solutions. This is certainly not intended. However, in general the generation of the search space may be guarded by some rules, and such rules might be considered more appropriately placed in the guessing part than in the checking part. We do not pursue this issue any further here, and thus also refrain from giving a formal definition of how to separate a program into a guessing and a checking part.

For solving a number of problems, however, it is possible to design a natural **Guess&Check** program in which the two parts are clearly identifiable and have a simple structure:

- The guessing part G consists of a disjunctive rule which "guesses" a solution candidate S.
- The checking part C consists of integrity constraints which check the admissibility of S, possibly using auxiliary predicates which are defined by normal stratified rules.

In a sense, the disjunctive rule defines the search space in which rule applications are branching points, while the integrity constraints prune illegal branches.

As a first example, let us consider *Hamiltonian Path*, a classical NP-complete problem from graph theory.

HPATH: Given a directed graph $G = (V, E)$ and a vertex a of this graph, does there exist a path of G starting at a and passing through each vertex in V exactly once?

Suppose that the graph G is specified by means of predicates *node* (unary) and *arc* (binary), and the starting node is specified by the predicate *start* (unary). Then, the following **Guess&Check** program $\mathcal{P}_{hp}$ solves the Hamilton Path problem.

```
inPath(X,Y) v outPath(X,Y) :- arc(X,Y).

 :- inPath(X,Y), inPath(X,Y1), Y <> Y1.
 :- inPath(X,Y), inPath(X1,Y), X <> X1.
 :- node(X), not reached(X).
 reached(X) :- start(X).
 reached(X) :- reached(Y), inPath(Y,X).
```

The first rule guesses a subset of all given arcs, while the rest of the program checks whether it is a Hamiltonian Path. Here, the checking part C uses an auxiliary predicate `reached`, which his defined using positive recursion.

In particular, the first two constraints in C check whether the set of arcs S selected by `inPath` meets the following requirements, which any Hamiltonian Path must satisfy: There must not be two arcs starting at the same node, and there must not be two arcs ending in the same node.

The two rules after the constraints define reachability from the starting node with respect to the selected arc set S. This is used in the third constraint, which enforces that all nodes in the graph are reached from the starting node in the subgraph induced by S. This constraint also ensures that this subgraph is connected.

It is easy to see that a selected arc set S which satisfies all three constraints must contain the edges of a path $a = v_0, v_1, \ldots, v_k$ in G that starts at node a, and passes through distinct nodes until no further node is left, or it arrives at the starting node a again. In the latter case, this means that the path is a Hamiltonian Cycle, and by dropping the last edge, we have a Hamiltonian Path.

Thus, given a set of facts F for *node*, *arc*, and *start* which specify the problem input, the program $\mathcal{P}_{hp} \cup F$

has an answer set if and only if the input graph has a Hamiltonian Path.

If we want to compute a Hamiltonian Path rather than only answering that such a path exists, we can strip off the last edge from a Hamiltonian Cycle by adding a further constraint `:- start(Y), inPath(_,Y).` to the program. Then, the set S of selected edges in an answer sets of $\mathcal{P}_{hp} \cup F$ constitutes a Hamiltonian Path starting at a.

It is worth noting that **DLV** is able to solve problems which are located at the second level of the polynomial hierarchy, and indeed also such problems can be encoded by the **Guess&Check** technique, as in the following example called *Strategic Companies*.

STRATCOMP: Given the collection $C = \{c_1, \ldots c_m\}$ of companies c_i owned by a holding, and information about company control, compute the set of the strategic companies in the holding.

To briefly explain what "strategic" means in this context, imagine that each company produces some goods. Moreover, several companies jointly may have control over another company. Now, some companies should be sold, under the constraint that all goods can be still produced, and that no company is sold which would still be controlled by the holding after the transaction. A company is *strategic*, if it belongs to a *strategic set*, which is a minimal set of companies satisfying these constraints.

This problem is Σ_2^P-hard in general (Cadoli, Eiter, & Gottlob 1997); reformulated as a decision problem ("Given a further company c in the input, is c strategic?"), it is Σ_2^P-complete. To our knowledge, it is the only KR problem from the business domain of this complexity that has been considered so far.

In the following encoding, strat(X) means that X is strategic, company(X) that X is a company, produced_by(X, Y, Z) that product X is produced by companies Y and Z, and controlled_by(W, X, Y, Z) that W is jointly controlled by X, Y and Z. We have adopted the setting from (Cadoli, Eiter, & Gottlob 1997) where each product is produced by at most two companies and each company is jointly controlled by at most three other companies.

Given the facts F for *company*, *controlled_by* and *produced_by*, the answer sets of the following program $\mathcal{P}1$ (actually $\mathcal{P}1 \cup F$) correspond one-to-one to the strategic sets of the holding. Thus, the set of all strategic companies is given by the set of all companies c for which the fact strat(c) is true under brave reasoning.

```
r : strat(Y) v strat(Z) :- produced_by(X, Y, Z).
c : strat(W) :-  controlled_by(W, X, Y, Z),
                 strat(X), strat(Y), strat(Z).
```

Intuitively, the guessing part G of $\mathcal{P}1$ consists of the disjunctive rule r, and the checking part C consist of the normal rule c. This program exploits the minimization which is inherent to the semantics of answer sets for the check whether a candidate set S of companies that produces all goods and obeys company control is also minimal with respect to this property.

The guessing rule r intuitively selects one of the companies $\mathtt{c}_1$ and $\mathtt{c}_2$ that produce some item g, which is described by produced_by$(\mathtt{g}, \mathtt{c}_1, \mathtt{c}_2)$. If there were no company control information, minimality of answer sets would then naturally ensure that the answer sets of $F \cup \{r\}$ correspond to the strategic sets; no further checking is needed. However, in case such control information, given by facts controlled_by$(\mathtt{c}, \mathtt{c}_1, \mathtt{c}_2, \mathtt{c}_3)$, is available, the rule c in the program checks that no company is sold that would be controlled by other companies in the strategic set, by simply requesting that this company must be strategic as well. The minimality of the strategic sets is automatically ensured by the minimality of answer sets. The answer sets of $\mathcal{P}2$ correspond one-to-one to the strategic sets of the given instance.

It is interesting to note that the checking constraint c interferes with the guessing rule r: applying c may spoil the minimal answer set generated by rule r. Such feedback from the checking part C to the guessing part G is in fact needed to solve Σ_2^P-hard problems.

In general, if a program encodes a problem that is Σ_2^P-complete, then the checking part C must contain disjunctive rules unless C has feedback to the guessing part G.

Finally, note that STRATCOMP can not be expressed by a fixed normal logic program uniformly on all collections of facts produced_by$(p, c1, c2)$ and controlled_by$(c, c1, c2, c3)$ (unless NP $= \Sigma_2^P$, an unlikely event).

Specifics

DLV is the result of putting theoretical results into practice. It is the first system supporting answer set semantics for full disjunctive logic programs with negation, integrity constraints, queries, and arithmetic built-ins.

The semantics of the language is both precise and intuitive, which provides a clean, declarative, and easy-to use framework for knowledge representation and reasoning.

The availability of a system supporting such an expressive language in an efficient way is stimulating AI and database people to use logic-based systems for the development of their applications.

Furthermore, it is possible to formulate translations from many other formalisms to **DLV**'s core language, such that the answer sets of the translated programs correspond to the solutions in the other formalism. **DLV** incorporates some of these translations as frontends. Currently frontends for diagnostic reasoning, SQL3, planning with action languages, and logic programming with inheritance exist.

We believe that **DLV** can be used in this way – as a core engine – for many problem domains. The advantage of this approach is that people with different background do not have to be aware of **DLV**'s syntax and semantics.

Users and Usability

Prospective users of the **DLV** core system should have a basic knowledge of logics for knowledge representation. As explained in the previous section, if a frontend for a particular language exists, a user need not even know about logics, but of course knowledge about the frontend language is still required.

Currently, the **DLV** system is used for educational purposes in courses on Databases and on AI, both in European and American universities. It is also used by several researchers for knowledge representation, for verifying theoretical work, and for performance comparisons.

Furthermore, **DLV** is currently under evaluation at CERN, the European Laboratory for Particle Physics located near Geneva in Switzerland and France, for an advanced deductive database application that involves complex knowledge manipulations on large-sized databases.

Evaluating the System

Benchmarks

It is a well-known in the area of benchmarking that the only really useful benchmark is the one where a (prospective) user of a system tests that system with exactly the kind of application he is going to use.

Nevertheless, artificial benchmarks do have some merits in developing and improving the performance of systems. Moreover, they are also very useful in evaluating the progress of various implementations, so there has been some work in that area, too, and it seems that **DLV** compares favorably to similar systems (Eiter *et al.* 1998; Janhunen *et al.* 2000).

Also for the development of some deductive database applications **DLV** can compete with database systems. Indeed, **DLV** is being considered by CERN for such an application which could not be handled by other systems.

Problem Size

As far as data structures are concerned, **DLV** does not have any real limit on the problem size it can handle. For example, we have verified current versions on programs with 1 million literals in 1 million rules.

Another crucial factor for hard input are suitable heuristics. Here we already have developed an interesting approach (Faber, Leone, & Pfeifer 1999) and are actively working on various new approaches.

To give an idea of the sizes of the problems that **DLV** can currently handle, and of the problems solvable by **DLV** in the near future, below we provide the execution times of a number of hard benchmark instances reporting also the improvements over the last year.

Problem	*Jul. '98*	*Feb. '99*	*Jun. '99*	*Nov. '99*
3COL[a]	> 1000s	26.4s	2.1s	0.5s
HPATH[b]	> 1000s	> 1000s	10.8s	0.3s
PRIME[c]	—	21.2s	10.2s	0.8s
STRATCOMP[d]	54.6s	8.0s	6.9s	5.4s
BW P4[e]	> 1000s	> 1000s	32.4s	6.3s
BW Split P4[f]	> 1000s	> 1000s	10.5s	2.3s

[a] find one coloring of a random graph with 150 nodes and 350 edges
[b] find one Hamiltonian Path in a random graph with 25 nodes and 120 arcs
[c] find all prime implicants of a random 3CNF with 546 clauses and 127 variables
[d] find all strategic sets a randomly chosen company occurs in (71 companies and 213 products)
[e] find one plan of length 9 involving 11 blocks
[f] linear encoding for *e*

References

Apt, K. R.; Blair, H. A.; and Walker, A. 1988. Towards a theory of declarative knowledge. In Minker, J., ed., *Foundations of Deductive Databases and Logic Programming.* Los Altos, California: Morgan Kaufmann Publishers, Inc. 89–148.

Buccafurri, F.; Faber, W.; and Leone, N. 1999. Disjunctive Logic Programs with Inheritance. In *Proceedings of the 16th International Conference on Logic Programming (ICLP '99).*

Cadoli, M.; Eiter, T.; and Gottlob, G. 1997. Default Logic as a Query Language. *IEEE Transactions on Knowledge and Data Engineering* 9(3):448–463.

Console, L.; Theseider Dupré, D.; and Torasso, P. 1991. On the Relationship Between Abduction and Deduction. *Journal of Logic and Computation* 1(5):661–690.

de Kleer, J.; Mackworth, A. K.; and Reiter, R. 1992. Characterizing diagnoses and systems. *Artificial Intelligence* 56(2–3):197–222.

Eiter, T.; Leone, N.; Mateis, C.; Pfeifer, G.; and Scarcello, F. 1998. The KR System `dlv`: Progress Report, Comparisons and Benchmarks. In Cohn, A. G.; Schubert, L.; and Shapiro, S. C., eds., *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, 406–417. Morgan Kaufmann Publishers.

Eiter, T.; Gottlob, G.; and Leone, N. 1997. Abduction from Logic Programs: Semantics and Complexity. *Theoretical Computer Science* 189(1–2):129–177.

Eiter, T.; Gottlob, G.; and Mannila, H. 1997. Disjunctive Datalog. *ACM Transactions on Database Systems* 22(3):315–363.

Faber, W., and Pfeifer, G. since 1996. `dlv` homepage. `<URL:http://www.dbai.tuwien.ac.at/proj/dlv/>`.

Faber, W.; Leone, N.; and Pfeifer, G. 1999. Pushing Goal Derivation in DLP Computations. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, Lecture Notes in AI (LNAI), 177–191. El Paso, Texas, USA: Springer Verlag.

Gelfond, M., and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* 9:365–385.

Janhunen, T.; Niemela, I.; Simons, P.; and You, J.-H. 2000. Partiality and disjunctions in stable model semantics. In *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR2000)*.

Kakas, A.; Kowalski, R.; and Toni, F. 1993. Abductive Logic Programming. *Journal of Logic and Computation*.

Lifschitz, V. 1996. Foundations of logic programming. In Brewka, G., ed., *Principles of Knowledge Representation*. Stanford: CSLI Publications. 69–127.

Poole, D. 1989. Explanation and Prediction: An Architecture for Default and Abductive Reasoning. *Computational Intelligence* 5(1):97–110.

Reiter, R. 1987. A Theory of Diagnosis From First Principles. *Artificial Intelligence* 32:57–95.