Using the dlv System for Planning and Diagnostic Reasoning*

Thomas Eiter   Wolfgang Faber†   Nicola Leone   Gerald Pfeifer
Axel Polleres

Institut für Informationssysteme, TU Wien
A-1040 Wien, Austria

{eiter,faber}@kr.tuwien.ac.at,{leone,pfeifer,polleres}@dbai.tuwien.ac.at

Abstract

`dlv` is a state-of-the-art system for disjunctive logic programming, which may be used as an implementation bed for solving declarative knowledge representation problems. In this paper, we review the use of the `dlv` system for solving diagnostic problems, and contrast this with solving planning problems in `dlv`. This is possible using a declarative "guess and check" approach: The power of disjunction allows for nondeterministic generation of plans, which are verified in a step by step manner by applying state transition rules. Efficient processing of these phases, in an interleaved mode, will be provided through the algorithms incorporated in the `dlv` engine. Based on these ideas, we are currently developing a front-end for deductive planning in `dlv`, which will be included in future releases of `dlv`.

1 Introduction

Planning and diagnostic reasonings are very important AI tasks. Both fields have been studied quite extensively before, for an overview we refer to [4] (for diagnosis) and to [17, 22] (for planning).

In this paper, we compare the two domains and review the use of the `dlv` system for solving these problems. We show that both problems exhibit a goal-oriented "guess and check" structure, which allows for a declarative representation in disjunctive datalog, `dlv`'s language.

In particular, disjunctive rules are used for guessing a solution candidate, which are then checked for validity. We would like to point out that `dlv` does not really perform a naïve approach like this; rather it uses efficient algorithms to interleave the guessing and checking phases as good as possible, such that the generation of candidates can be pruned effectively, see e.g. [8, 12].

Let us first focus on the planning and diagnostic reasoning domains.

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

†Please address correspondence to this author.

1.1 Planning

In planning, given a description of a world, an initial situation, and a desired situation, the goal is to find a sequence of actions (which can change the situations), such that the desired situation is reached from the initial one. In addition, not all actions are applicable in every situations. We formulate our model of planning accordingly:

Definition 1 *A* planning problem *(PP) $\mathcal{P}$ is a quadruple $\langle F, A, I, G\rangle$, where:*

- *F is a set of* fluents*, which characterize the situations*
- *A is a set of* actions*, with a definition of their respective preconditions and effects*
- *I is a set of fluents describing the initial situation*
- *G is a set of fluents describing the goal situation*

□

We define fixed-length solutions to a PP as follows:

Definition 2 *Given a PP $\mathcal{P} = \langle F, A, I, G\rangle$ and an integer n, a* plan *for PP is a sequence $A = a_1 \cdots a_n$ such that there are $n+1$ situations $S_0, \ldots, S_n$, such that for each a_i in A, S_{i-1} is consistent with a_i's preconditions, and S_i is modified from S_{i-1} by exactly the effects of a_i. To make the plan valid for the given situations, S_0 must be entailed by I, and G must be entailed by S_n.* □

Consider the planning problem $\mathcal{P}_{switch} = \langle \{light\_on\}, \{toggle\}, \{\neg light\_on\}, \{light\_on\}\rangle$, where the action *toggle* has no precondition and the effect is that if *light_on* holds, $\neg light\_on$ holds after the action and vice versa. Then there exists a plan of length 1, *toggle*. In fact every sequence of *toggle* actions, whose length is an odd number, is a valid plan.

In this definition (and the example) we did not go into any details concerning the language involved in specifying the problem. We refer to [15, 3, 16, 18] for definitions of action languages, which are suited for this purpose.

1.2 Diagnosis

In diagnosis, given a theory and some observations, the goal is to find a set of hypotheses which can explain the observations by means of the theory. The notion of "explanation" is not a priori clear, therefore we define two notions of diagnosis, *abductive* and *consistency-based* diagnosis, following [4].

In the *abductive* model of diagnosis [1, 5, 20], the theory is a function-free disjunctive logic program, whose semantics is given by the set of its stable models [14]. This form of diagnostic reasoning has been recently proved to be highly expressive, as it is able to represent even problems located at the third level of the polynomial hierarchy [6].

Definition 3 *We define an* abductive diagnostic problem (ADP) *$\mathcal{P}$ as a triple $\langle H, T, O\rangle$, where:*

- *H is a set of ground atoms and is referred to as* hypotheses.

- *T is a disjunctive datalog program and is referred to as* theory*.*
- *O is a set of ground literals and is referred to as* observations*.*

$\square$

Definition 4 *Given an ADP $\mathcal{P} = \langle H, T, O \rangle$, a*

- generic diagnosis *is a set $\Delta \subseteq H$, such that $T \cup \Delta \models O$ holds.*
- single error *diagnosis Δ is a generic diagnosis, for which $|\Delta| = 1$ holds additionally.*
- subset minimal *diagnosis Δ is a generic diagnosis and all subsets $\Delta^* \subset \Delta$ are not generic diagnoses.*

$\square$

On the other hand, the *consistency-based* diagnosis model refers to theories of first-order logic under classical semantics, similar to the original definition by Reiter [21].

Definition 5 *A* consistency-based diagnostic problem (CDP) *$\mathcal{P}$ is a triple $\langle H, T, O \rangle$, where:*

- *H is a set of ground atoms with predicate name ab (standing for* abnormal*) and is referred to as* hypotheses*.*
- *T is a set of first-order sentences and is referred to as* theory*.*
- *O is a set of ground literals and is referred to as* observations*.*

$\square$

We assume that these sentences are written in clausal form so that they can be represented by datalog programs. H is comprised of atoms of the form `ab(c)`, meaning that the component `c` behaves abnormally.

Definition 6 *Given a CDP $\mathcal{P} = \langle H, T, O \rangle$, a*

- generic diagnosis *is a set $\Delta \subseteq H$, such that $T \cup O \cup \Delta \cup \{\neg h | h \in H - \Delta\}$ is consistent in the classical sense.*
- single error *diagnosis Δ is a generic diagnosis, for which $|\Delta| = 1$ holds additionally.*
- subset minimal *diagnosis Δ is a generic diagnosis and all subsets $\Delta^* \subset \Delta$ are not generic diagnoses.*

$\square$

Examples for diagnostic reasoning can be found in Section 2.2 and in [4].

1.3 Planning versus Diagnostic Reasoning

Let us now consider the relationship between the two domains: First of all, it can be observed that both tasks are goal-oriented. In the case of diagnosis, the goal is to find explanations for the observations. For planning, the goal is to find a plan which leads to the desired situation. In a less procedural view, the observations and the desired situation actually are the goals.

In both diagnosis and planning, a choice must be made out of a pool of atomic items (hypotheses and actions), which forms a solution candidate. The difference is that the solutions are ordered in planning, while they are an unordered set for diagnosis.

For checking the validity of solution candidates, the theory is used in diagnosis, while in planning there are more implicit conditions, such as checking the admissibility (by means of the preconditions) and the correct execution (by means of effects and inertial laws) of actions. Of course, these checks also have to guarantee the satisfaction of the goals.

The analysis above shows that these problems follow a guess and check pattern involving a goal. As motivated in [7], a disjunctive logic programming approach is particularly suitable for representing and solving problems following such a pattern.

2 Representing Diagnosis and Planning Problems Using `dlv`

2.1 The `dlv` System

`dlv` is a powerful logic programming and non-monotonic reasoning (LPNMR) system with advanced knowledge representation mechanisms and interfaces to classic relational database systems [8, 9].

Its core language is disjunctive datalog (function-free logic programming) under the Answer Set semantics [14] with integrity constraints, strong (or explicit) negation, and queries. Integer arithmetics and various built-in predicates are also included in the core language.

In addition to this base language, `dlv` has several frontends, namely brave and cautious reasoning, abductive diagnosis, consistency-based diagnosis and a subset of SQL3. For up-to-date information on the system and a full manual please refer to the project homepage [13]. `dlv` is also available for download from that page.

Syntactically, `dlv` programs consist of rules and constraints, possibly also queries. A rule has the form

$$a_1 \texttt{v} \ \ldots \texttt{v} \ a_m \texttt{:-} \ b_1\texttt{,} \ \ldots\texttt{,} \ b_k\texttt{,} \ \texttt{not} \ b_{k+1}\texttt{,} \ \ldots\texttt{,} \ \texttt{not} \ b_n\texttt{.}$$

where $m \geq 1$ and $n \geq 0$.

$a_1, \ldots, a_m, b_1, \ldots, b_n$ are *strong literals*, i.e., atoms $a\texttt{(}t_1\texttt{,} \ldots \texttt{,}t_f\texttt{)}$ or strongly negated atoms $\texttt{-}a\texttt{(}t_1\texttt{,} \ldots \texttt{,}t_f\texttt{)}$, where a is the predicate name of the atom and each t_i $(1 \leq i \leq f)$ is either a constant, a variable, or an anonymous variable. Note that in the actual implementation the order of the body literals is immaterial.

The part to the left of `:-` is called the *head* of the rule, the part to the right is the *body*. A constraint is a rule with an empty head ($m = 0$). A rule with $m = 1$ and an empty body is called a *fact*, the `:-` can be omitted in this case.

Constants are strings starting with a lower-case letter, while variables are strings starting with an upper-case letter. Anonymous variables are denoted by "_". Each occurrence of an anonymous variable stands for a variable that is different from all others in the entire rule.

Semantically, each program can have no, one or several consistent Answer Sets, each of which is also a minimal model of that program.

A nice example for the knowledge modeling power of `dlv` is *Graph 3-Colorability* – given an undirected graph, represented by facts of the form `edge(_,_)`, assign each node one of three colors such that no two adjacent nodes have the same color. The simple program in Figure 1 computes the legal 3-colorings of the graph.

```
node(X) :- edge(X,Y).
node(Y) :- edge(X,Y).

% Each node is either red, green, or blue.
colored(X,red) v colored(X,green) v colored(X,blue) :- node(X).

% Two nodes on the same edge must not have the same color.
:- edge(X,Y), colored(X,C), colored(Y,C).
```

Figure 1: Graph 3-Coloring

As another example, take the program in Figure 2, which computes the *Hamilton paths* of a directed graph. (Recall that a Hamilton path of a directed graph is a path through that graph that reaches each node exactly once.) In our example, the graph is given by a relation `arc(X,Y)` denoting its arcs and a starting node `start(X)`.

```
% start(X), the starting node, is reached by definition.
% Any other node is reached, if it is the destination
% of an arc that is part of the current path.
reached(X) :- start(X).
reached(X) :- inPath(_,X).

% Each arc starting in an already reached node is either
% in the path, or it is not.
inPath(X,Y) v -inPath(X,Y) :- reached(X), arc(X,Y).

% No two arcs with the same source resp. destination
% nodes may be included in the path.
% != is a built-in predicate denoting inequality.
:- inPath(X,Y), inPath(X,Y1), Y != Y1.
:- inPath(X,Y), inPath(X1,Y), X != X1.

% Each node has to be reached.
:- arc(X,_), not reached(X).
```

Figure 2: Finding Hamilton Paths

2.2 Diagnostic Reasoning in `dlv`

`dlv` provides two kinds of diagnosis frontends: one for *abductive diagnosis*, and one for *consistency based diagnosis*, as shown in Section 1.

Both frontends support three different modes: general diagnosis, where all diagnoses are computed, subset minimal diagnosis[1], and single failure diagnosis.

The diagnostic theory obeys the basic syntax described in Section 2.1. Hypotheses (resp. observations) are lists of atoms (resp. literals) followed by a dot (`.`) and are stored in files whose names carry the extension `.hyp` (resp. `.obs`). Note that hypotheses and positive observations are syntactically equal to facts, but their semantics is different.

In the case of abductive diagnosis, these modes are invoked by command line options `-FD`, `-FDmin` and `-FDsingle`, respectively, while for *consistency-based diagnosis* the corresponding command line options are `-FR`, `-FRmin` and `-FRsingle`.

For detailed information on the theoretical foundations of the implementation of the diagnosis frontends please refer to [4].

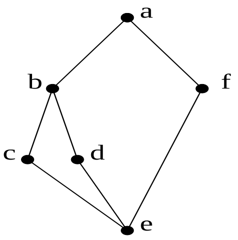

Figure 3: A computer network

As an example, consider the computer network depicted in Figure 3. Suppose that machine a is online in this network but we observe it cannot reach machine e. Which machines are offline? This can be easily modeled as a diagnosis problem, where the theory is:

```
reaches(X,X) :- node(X), not offline(X).
reaches(X,Z) :- reaches(X,Y), connected(Y,Z), not offline(Z).
```

the observations are:

```
not offline(a). not reaches(a,e).
```

and the hypotheses are:

```
offline(a). offline(b). offline(c).
offline(d). offline(e). offline(f).
```

The diagnoses in this case are D_1 ={`offline(b)`, `offline(f)`}, D_2 ={`offline(c)`, `offline(d)`, `offline(f)`}, D_3 ={`offline(e)`}, and all supersets of D_1, D_2, or D_3.

[1]For positive non-disjunctive theories only.

2.3 Planning in dlv

In the literature the relation between planning, reasoning about actions, and non-monotonic logic programming has been studied quite extensively [15, 23, 2]. However, most of this work does not deal with disjunctive logic programming. Exceptions are [10, 19], where several planning problems are encoded using disjunctive datalog.

Due to lack of space, we omit the step of describing an action language and the associated translation to disjunctive datalog (this will be the task performed by a planning front-end, which we are currently implementing), and directly show the encoding of an example planning domain in disjunctive datalog.

The following encoding is similar to the ones presented in [10, 18], but there is a main difference: In our approach, we view situations as states of knowledge, rather than complete states. Our approach can be thought of as modelling an autonomous robot, which usually is not omniscient, hence his knowledge may be incomplete.

We show by example of the well-known blocksworld domain, how such an encoding can be achieved in disjunctive datalog. The objects in the blocksworld are one table and an arbitrary number of labeled cubic blocks. They are referred to as `locations`:

```
location(table).   location(L) :- block(L).
```

Since the number of blocks varies in different problem instances, the blocks are defined together with these instances.

We have not described the time yet: We use integer built-in predicates and constants to model time. The first time is always `0`, and the last time `k` is specified when invoking `dlv` via the command-line option `-N=k`. `#int(T)` represents all numbers in [0..k], `#succ(T,T1)` defines the successor relation in the same interval, and the constant `#maxint` is substituted by `k`. There is also an inequality operator `<>`, which is defined over these numbers. We define `actiontime`, which represents all times at which actions may be initiated (all but the last).

```
actiontime(T) :- #int(T), T <> #maxint.
```

The state of the blocksworld at a particular time can be described by the relation `on(B,L,T)`, which specifies that block `B` resides on location `L` at time `T`.

There is one action, which is moving a block from one location to another location. A move is started at one point in time, and it is completed before the next time. At any time `T` which allows for initiating an action, the action of moving a block `B` to location `L` (where `B` must be different from `L`) may be taken (`move(B,L,T)`) or not (`-move(B,L,T)`). This single rule serves as the guessing part of our program.

```
move(B,L,T) v -move(B,L,T) :- block(B), location(L),
                              actiontime(T), B <> L.
```

In order to move a block, some preconditions must hold. The first is that no other block may be on the moved block.

```
:- move(B,L,T), on(B1,B,T).
```

Similar, if a block should be moved onto another block, no block may be on the latter. Note that an arbitrary number of blocks may be moved onto the table.

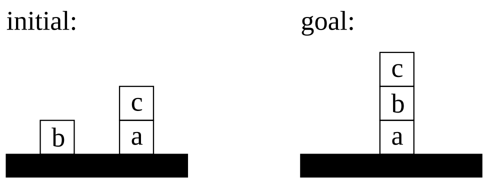

Figure 4: The Sussman anomaly.

```
% Define the involved blocks.
block(a).  block(b).  block(c).

% Specify the initial and goal situations.
on(a,table,0).  on(b,table,0).  on(c,a,0).
on(a,table,#maxint), on(b,a,#maxint), on(c,b,#maxint)?
```

Figure 5: The Sussman anomaly in disjunctive datalog.

```
:- block(B1), move(B,B1,T), on(B2,B1,T).
```

Next, the effects of an action are that the moved block is no longer on its previous location and that it is on the new location at the next point in time.

```
on(B,L,T1) :- move(B,L,T), #succ(T,T1).
-on(B,L,T1) :- move(B,_,T), on(B,L,T), #succ(T,T1).
```

It is required that at most one action is initiated at one time:

```
:- move(B,L,T), move(B1,L1,T), B != B1.
:- move(B,L,T), move(B1,L1,T), L != L1.
```

The `on` relation is inertial. This means that unless it is changed by an action, it remains stable over the time.

```
on(B,L,T1) :- on(B,L,T), #succ(T,T1), not -on(B,L,T1).
```

For a problem instance, the involved blocks, the initial state (describing where each block is located) must be specified by facts. The task is to find a sequence of moves (a *plan*) such that the goal state (which is given by a query) is reached by applying the action sequence to the initial state.

One classical instance is the so-called Sussman anomaly[2] [24], depicted in Figure 4, and its representation in disjunctive datalog is shown in Figure 5. When we look for a plan of length 3 (by specifying `-N=3` on the commandline) for this instance, exactly one is found:

```
move(c,table,0), move(b,a,1), move(c,b,2)
```

Indeed, this is the only valid plan.

[2]It is called anomaly, because at the time it was formulated (and also quite some time after that) planners could not handle it correctly.

3 Conclusion and Further Work

We have discussed some relationships between two important AI tasks, namely Diagnosis and Planning. The way in which they are related, namely by showing a guess and check pattern, shows that disjunctive datalog is particularly suitable for representing and solving these types of problems.

The `dlv` system is a state-of-the-art system for solving problems represented in disjunctive datalog, and can thus serve as the computational engine for solving both diagnostic and planning problems. To ease the use of the system, frontends to `dlv` are developed, which transform problems written in a particular domain language to disjunctive datalog automatically. They are an integral part of `dlv`.

For diagnostic reasoning we have already developed such a frontend by using techniques which are described in detail in [4].

For planning, we have shown by example how such problems can be represented. A frontend which can transform planning problems specified in an action language to disjunctive datalog is currently under development and will be included in one of the next releases of `dlv`.

To get an idea about performance, we refer to the webpage [11] at the University of Texas, where you can find the report [10], which contains benchmark data for `dlv` and other systems.

References

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

[2] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding Planning Problems in Nonmonotonic Logic Programs. In *Proceedings of the European Conference on Planning 1997 (ECP-97)*, pages 169–181. Springer Verlag, 1997.

[3] P. M. Dung. Representing Actions in Logic Programming and Its Applications in Database Updates. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 222–238, 1993.

[4] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The Diagnosis Frontend of the `dlv` System. *AI Communications – The European Journal on Artificial Intelligence*, 12(1–2):99–111, 1999.

[5] T. Eiter, G. Gottlob, and N. Leone. Abduction from Logic Programs: Semantics and Complexity. In *Theoretical Computer Science* [6], pages 129–177.

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

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

[8] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A Deductive System for Nonmonotonic Reasoning. In J. Dix, U. Furbach, and A. Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR '97)*, number 1265 in Lecture Notes in AI (LNAI), pages 363–374, Berlin, 1997. Springer.

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

[10] E. Erdem. Applications of Logic Programming to Planning: Computational Experiments. Unpublished draft, 1999.

[11] E. Erdem. Website for applications of logic programming to planning: Computational experiments, since 1998. `<URL:http://www.cs.utexas.edu/users/esra/experiments/experiments.html>`.

[12] W. Faber, N. Leone, and G. Pfeifer. 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), El Paso, Texas, USA, December 1999. Springer Verlag.

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

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

[15] M. Gelfond and V. Lifschitz. Representing Action and Change by Logic Programs. *Journal of Logic Programming*, 17:301–321, 1993.

[16] E. Giunchiglia and V. Lifschitz. An Action Language Based on Causal Explanation: Preliminary Report. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI '98)*, pages 623–630, 1998.

[17] H. Kautz and B. Selman. Planning as Satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359–363, 1992.

[18] V. Lifschitz. Action Languages, Answer Sets and Planning. In K. Apt, V. W. Marek, M. Truszczyński, and D. S. Warren, editors, *The Logic Programming Paradigm - A 25-Year Perspective*, pages 357–373. Springer Verlag, 1999.

[19] V. Lifschitz. Answer set planning. In *Proceedings of the 16th International Conference on Logic Programming – ICLP'99*, 1999.

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

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

[22] M. Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia.* MIT Press, 1997.

[23] V. Subrahmanian and C. Zaniolo. Relating Stable Models and AI Planning Domains. In *Proceedings of the 12^{th} International Conference on Logic Programming*, pages 233–247, 1995.

[24] G. J. Sussman. The Virtuous Nature of Bugs. In J. Allen, J. Hendler, and A. Tate, editors, *Readings in Planning*, chapter 3, pages 111–117. Morgan Kaufmann Publishers, Inc., 1990. originally written 1974.