ProLogICA: a practical system for Abductive Logic Programming

Oliver Ray[*]
Imperial College London, UK
email: or@doc.ic.ac.uk

Antonis Kakas[†]
University of Cyprus, Cyprus
email: antonis@cs.ucy.ac.cy

Abstract

This paper presents a new system called *ProLogICA* for Abductive Logic Programming (ALP) with Negation as Failure (NAF) and Integrity Constraints (ICs). The system builds upon existing ALP techniques but includes several optimisations and extensions necessitated by recent applications in computational biology, temporal reasoning and machine learning. Unlike some other ALP systems that support non-ground abduction through the integrated use of constraint solving, we adopt a more lightweight approach which avoids this complexity at the expense of only computing ground hypotheses. We argue our approach is suited to a wide class of real-world problems and demonstrate the effectiveness of *ProLogICA* on three non-trivial applications.

Introduction

The utility of Abductive Logic Programming (ALP) for knowledge representation and problem solving with Negation as Failure (NAF) and Integrity Constraints (ICs) is widely accepted (Kakas, Kowalski, & Toni 1992; Kakas & Denecker 2002). Moreover, state-of-the-art ALP systems, such as the *A-system* (Kakas, Van Nuffelen, & Denecker 2001) and *CIFF* (Endriss *et al.* 2004) also incorporate powerful constraint solvers that enable the inference of non-ground abducibles and the solution of complex optimisation problems characteristic of some application domains. These systems are particularly effective in constraint-based tasks such as planning (Van Nuffelen & Denecker 2000) and scheduling (Kakas & Michael 2001).

But many applications do not require complex constraint handling capabilities and simpler ALP procedures with lower computational overheads are often appropriate. This is especially true of recent machine learning techniques, such as *Hybrid Abductive Inductive Learning* (HAIL) (Ray 2005), that integrate abduction and induction in a common reasoning framework. In this task, (i) the availability of an inductive reasoning module for generalising abductive explanations means that ground based abduction is sufficient, (ii) the use of standard Prolog as a representation language and execution model means that support for Prolog libraries and the efficient processing of logical integrity constraints is more useful than constraint solving, and (iii) since ALP is just one of part in a much larger system, low overheads and good performance are paramount.

This paper presents a practical lightweight ALP system called *ProLogICA*[1] that was first developed as the abductive component of HAIL, but has also been applied in three real-world applications involving temporal event calculus reasoning (Alberti *et al.* 2005), the inference of genetic regulatory networks (Papatheodorou, Kakas, & Sergot 2005), and a novel form of "in-silico genotyping" for predicting HIV drug resistance (Ray *et al.* 2006). In fact, because these applications were carried out in parallel with the development of *ProLogICA*, each of them prompted a number of developments that were incorporated into the computational model of the resulting procedure.

ProLogICA is closely based on the ALP procedure of Kakas and Mancarella (KM) (Kakas & Mancarella 1990a), but includes several optimisations and extensions that were necessitated by the three applications mentioned above. The optimisations are concerned with pruning redundant branches from abductive and consistency computations, whereas the extensions are concerned with overcoming the representational restrictions of the KM procedure and avoiding the computation of non-minimal solutions. The system also supports dynamic integrity constraints, which are generated on-the-fly as the computation unfolds, in order to avoid floundering in consistency computations.

Improved performance is realised, through a syntactic analysis of the ALP theory to identify sources of determinism that can be exploited and, also, through run time heuristics for literal selection and pruning. Enhanced functionality is provided by preprocessing the ALP theory to overcome the representational restrictions of KM and, additionally, by the provision of metalogical operators that enable a finer degree of control over the search space. *ProLogICA* supports most Prolog built-in predicates and offers a facility for depth bounded computation. These features may be customised by the user via system parameters described in this paper.

[*]Part of this work was completed while visiting the Department of Computer Science at the University of Cyprus.

[†]Part of this work was completed while visiting the Department of Computing at Imperial College London.

[1]"*ProLogICA*" stands for "Prolog with Integrity Constraints and Abduction", but is also obtained by rearranging the capitalised letters in "Abductive LOGIC PROgramming". The system can be downloaded from http://www.doc.ic.ac.uk/~or/proLogICA.

ALP

ALP is an extension of normal logic programs for reasoning with incomplete information. As in conventional logic programming, ALP seeks to establish the conditions under which a goal follows from a theory. But, in addition to computing a set of bindings for the variables in the goal, ALP returns a set of ground atoms that can be added to theory to ensure the goal succeeds. These atoms are usually drawn from a predefined set of ground atoms, called *abducibles*, which represent those facts for which there is only partial information in the form of integrity constraints.

Semantics

Formally, an abductive theory is a triple (P, IC, A) comprising a normal logic program P (domain knowledge), a set of formulae IC (integrity constraints), and a set of ground atoms A (abducibles). A goal G is a set of literals and an abductive explanation of G with respect to (P, IC, A) is a set of atoms $\Delta \subseteq A$ such that $P \cup \Delta \models^* \exists G \wedge \forall IC$, where $\models^*$ denotes the satisfaction of the formula on the right in a *stable model* of the program on left. In the terminology of (Kakas & Mancarella 1990b), G is said to be satisfied in a *Generalised Stable Model* (GSM) of (P, IC, A).

To discriminate between alternative abductive explanations, additional preference criteria are often utilised. Two popular desiderata are *minimality* and *basicality*. Formally, an explanation Δ of G with respect to (P, IC, A) is *minimal* iff there is no $\Delta' \subset \Delta$ such that Δ' is also an explanation of G, and is *basic* iff there is no $\Delta' \not\supseteq \Delta$ such that Δ' is also an explanation of Δ. Intuitively, an explanation Δ is minimal if none of its atoms are redundant; and it is basic if none of its atoms can be further explained.

Procedure

The KM procedure is a standard ALP technique that computes abductive explanations by interleaving abductive derivations (in which abducibles are assumed) and consistency derivations (in which consistency is enforced). Given a theory (P, IC, A) and a goal G, the KM procedure starts an abductive derivation by unfolding the goal G against the program P in Prolog fashion until an abducible a is selected. At this point, a consistency derivation is invoked to see if the atom a can be added to the initially empty hypothesis Δ without violating the integrity constraints IC.

A consistency derivation comprises one separate branch for each resolvent of the integrity constraints and the abducible a under investigation. Each such resolvent is regarded as a query that must be shown to fail in order for the integrity check, as a whole, to succeed. This is established by repeatedly resolving on selected literals in all possible ways and, if necessary, assuming further abducibles (that are carried over to any remaining branches of the computation).

When all branches of the consistency derivation have been closed, the outer abductive computation continues with a and any other abducibles assumed in the consistency computation added to Δ – indicating that all subsequent calls to these abducibles should immediately succeed. If any branch of the consistency derivation cannot be closed, then the outer abductive computation is failed and the backtracking mechanism is invoked.

Negative literals, which are preceded by the connective not, are treated as abducibles subject to the (implicit) integrity constraint $\forall(not\, b \leftrightarrow \neg b)$. Thus, whenever a negative literal $not\, b$ is selected in an abductive (resp. consistency) derivation, it is assumed (resp. failed) by the KM procedure subject to there being a successful consistency (resp. abductive) derivation for the positive literal b – possibly resulting in deeply nested consistency and abductive computations.

Example

The KM procedure is best illustrated by an example. Fig. 1 shows an abductive theory describing the lactose metabolism of the bacterium E. Coli. The program P models the fact that E. coli can feed on the sugar lactose (`lact`) if it makes two enzymes permease (`perm`) and galactosidase (`gala`). Like all enzymes (`E`), these are made if they are coded by a gene (`G`) that is expressed. These enzymes are coded by two genes (`lac(y)` and `lac(z)`) in cluster of genes (`lac(X)`) – called an *operon* – that is only expressed when the amounts of glucose (`gluc`) are low (`lo`) and lactose are high (`hi`).[2] The abducibles A declare all ground instances of the predicates `amt` and `sugar` as assumable. The integrity constraints state that the amount of a substance (`S`) may not be both high and low; and can only be known if the substance is a sugar. (The atom `ic` denotes logical falsity, so the first constraint is equivalent to $\forall_S : \neg(amt(S, lo) \wedge amt(S, hi))$), while the second constraint is equivalent to $\forall_{S,V} : amt(S, V) \rightarrow sugar(S)$.)

```
%------ Domain Knowledge (P) ------%
feed(lact):-make(perm),make(gala).
make(E):-code(G,E),express(G).
express(lac(X)):-amt(gluc,lo),amt(lact,hi).
code(lac(y),perm).
code(lac(z),gala).

%--- Integrity Constraints (IC) ---%
ic :- amt(S,lo), amt(S,hi).
ic :- amt(S,V), not sugar(S).

%--------- Abducibles (A) ---------%
abducible_predicate(amt).
abducible_predicate(sugar).
```

Figure 1: ALP model of lactose metabolism regulation in E. coli. — Simplified from the case study in (Ray 2005).

The KM computation resulting from the goal `feed(lact)` is shown in Fig. 2. In the notation of (Kakas & Mancarella 1990a), abductive derivations are enclosed in a single-line boxes, while consistency computations are enclosed in double boxes. The initial goal is

[2] Biologically, this reflects the fact that glucose is a preferred food source that E. coli metabolises more efficiently than lactose.

shown at the very top of the computation. The next four goals are obtained by unfolding the top goal Prolog-style by resolving with the clauses in the program P.

Upon selecting `amt(gluc,lo)`, KM adds this abducible to the initially empty hypothesis Δ and begins the first consistency derivation. Since the abducible resolves with both integrity constraints, there are two branches, which, for convenience, are separated by a double horizontal line. The first must show the failure of `amt(gluc,hi)`, which it does by abducing its negation `not amt(gluc,hi)`.

Adding this to Δ, KM proceeds with the second branch, which must show the failure of `not sugar(gluc)`. This means showing `sugar(gluc)` by a subsidiary abductive computation. Since it is abducible, this is assumed subject to its own integrity check. But, as it does not trigger any integrity constraints, the innermost consistency derivation trivially succeeds; and, hence, so do the enclosing abductive and consistency derivations.

Note that a double box succeeds when all of its branches are disproved (denoted ■), whereas a single box succeeds when all of its goals are proved (denoted □). Note also how the hypothesis Δ grows monotonically throughout the computation. For compactness, we only show the hypothesis at the beginning and end of each (branch of a) consistency derivation. It is assumed the final hypothesis is exported to the enclosing abductive derivation.

Having abduced `amt(gluc,lo)`, `sugar(gluc)` and `not amt(gluc,hi)`, KM resumes the outer abductive computation, where the goals `amt(lact,hi)` and `make(gala)` are pending. The former results in a consistency derivation analogous to that described above, but the latter succeeds the already abduced atoms `amt(gluc,lo)` and `amt(lact,hi)` without needing to perform any additional consistency checks.

This example shows the top-down nature of KM and its handling of negation through subsidiary computations like NAF in conventional Prolog. It also illustrates the main strengths of this procedure, which are (i) the interleaving of abductive and consistency computations so that integrity violations are detected as soon as they arise, and (ii) the avoidance of excessive integrity checking by recording the *absence* of abducibles selected in consistency computations and by checking only those constraints that resolve with the abducible being assumed.

To avoid the complex integrity checking procedures needed to handle existentially quantified variables in abducibles, KM is committed to selecting ground abducibles only. This means KM must explore all ways of failing a constraint which may lead to variable bindings that result in the grounding of abducibles. However, experience shows that existing implementations of KM generate excessive numbers of choice points, which makes them too inefficient for the applications described in this paper. In addition, as discussed in the next section, KM imposes two restrictions upon ALP theories that are not always appropriate, and it provides no mechanisms for handling the large number of explanations that can arise in practical applications.

We have developed *ProLogICA* to address these issues.

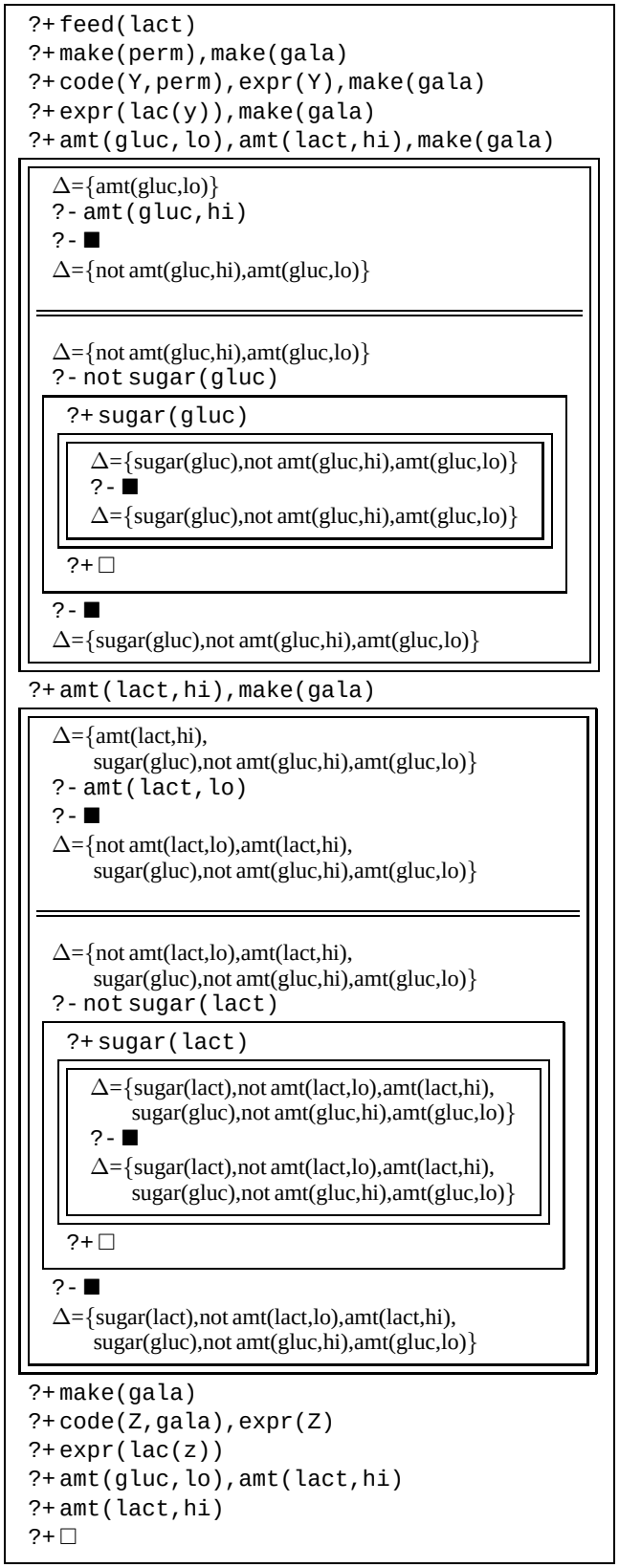

Figure 2: Successful KM computation for the query `feed(lact)` using the ALP theory in Fig. 1.

ProLogICA

Initialisation

ProLogICA is a meta-interpreter for SICStus Prolog 3.11.2. The code is contained in a single Prolog file `alp.pl` and is invoked with the command `sicstus -l alp`. A list of available commands is obtained by typing `help`. An ALP theory to be interpreted is loaded with the command `file('fname')`. If necessary, the working directory is set with the command `path('dirpth')`. The object file syntax is shown in Fig. 1. Negative literals may appear in the bodies of clauses using the operator `not/1` (which may not be nested). Integrity constraints are clauses with the head atom `ic`. Abducible predicates are declared by `abducible_predicate/1`. Built-in Prolog predicates can be used, but impure ones should be treated with care. Control primitives like cut and disjunction are not supported. All non-built-in predicates should be declared dynamic: e.g. with the following declaration for the code in Fig. 1:

```
:-dynamic feed/1,make/1,code/2,express/1,
amt/2,sugar/1,abducible_predicate/1,ic/0.
```

Invocation

Abductive queries are invoked through the predicate `demo/2`. There first argument should be instantiated to a list of goals. The second argument will become bound to a list of abducibles. Thus, running the query `?-demo([feed(lact)],D)` on the code in Fig. 1 results in the answer `D=[sugar(lact), not amt(lact,lo), amt(lact,hi), sugar(gluc), not amt(gluc,hi), amt(gluc,lo)]`. The Prolog backtracking mechanism will search for other explanations if the user types ";". Should `eval/2` be used in place of `demo/2`, the system will remove all non-minimal solutions that it computes.

Preprocessing

The KM procedure relies heavily on two representational restrictions: (i) that all integrity constraints contain an abducible predicate and (ii) that no abducible predicates are defined in the theory. These assumptions allow KM to avoid repeatedly re-checking all of the integrity constraints every time an atom is abduced. From a knowledge representation perspective, these restrictions can be motivated by arguing that abducibles are predicates whose extents are completely unknown except for the integrity constraints.

However, these assumptions are not always practicable as the first is often violated when the application provides partial knowledge of abducible predicates, and the second is often violated when the application suggests general domain integrity constraints that do not explicitly mention any abducible predicates. Moreover, in situations that involve refining incomplete theories (Ray 2005), these assumptions are particularly inconvenient.

Thus, *ProLogICA* provides a preprocessor that transforms any abductive theory into a theory satisfying (i) and (ii). In fact, it is well known that (ii) can always be ensured by replacing each abducible predicate `p/n` by a new abducible predicate `p'/n` and adding a clause to the program of the form `p(X1,...,Xn):- p'(X1,...,Xn)`. Since, this usually leads to the violation of (i), *ProLogICA* repeatedly unfolds each constraint with no abducibles by resolving on a selected atom with the program clauses.

Since, in general, it may be impossible to ensure that all constraints derived in this way contain abducibles, after a preset number of steps, *ProLogICA* collects those with abducibles, as these can be efficiently processed during the computation, and leaves the others to be checked at the end. The transformed clauses can be seen by typing `list` and can be recomputed with the command `reload`.

Features

ProLogICA offers many features not found in existing KM implementations. To ensure termination, derivations are depth bounded. To avoid floundering in consistency computations, dynamic integrity constraints are issued (Kakas, Michael, & Mourlas 2000). A selection policy is used that preferentially selects ground atoms, evaluates built-ins as soon as they are executable, and processes assumed or inconsistent abducibles when they become ground. A meta-predicate `atleast/2` is offered for ensuring the success of N goals from a given list whilst assuming the fewest number of abducibles. Efficient pruning mechanisms are also provided, as described below.

Settings

Customisation of *ProLogICA* is achieved via user-definable system parameters whose values are set with the command `set(param,value)`. Boolean settings take the values `true` or `false`, while numeric settings take integers. The parameters are summarised below (with default values).

max_ab_depth (70): bounds the combined depth of all abductive derivations in a computation.

max_con_depth (50): bounds the individual depth of each consistency derivations in a computation.

max_ic_unfold (5): bounds the number of times integrity constraints are unfolded during preprocessing. Setting this to 0 disables integrity unfolding, but still performs a consistency check at the end of each computation. Setting this to a negative value disables all pre and postprocessing of integrity constraints and abducibles.

enable_pruning (true): activates pruning mechanisms that effect a substantial reduction in the number of redundant branches in the search space at the expense of a negligible loss of performance on non-redundant branches. This is done mainly by eliminating unnecessary backtracking points in consistency derivations.

exploit_determinism (true): uses so-called *closed predicates*, which do not depend on any abducibles, to enable further pruning of consistency computations. Although it reduces execution times, this option can lead to a loss of completeness if the program contains positive cycles.

attempt_minimal (false): To avoid computing non-minimal solutions, *ProLogICA* offers a heuristic search strategy that attempts to minimise the number of literals

in each explanation. This almost always reduces the number of non-minimal hypotheses, but can reduce or increase the execution time according to context.

show_negatives (false): When true, this option prevents negative literals being removed from computed explanations during postprocessing.

show_constraints (false): When true, this option results in dynamic integrity constraints being appended on to computed explanations during postprocessing.

show_time (false): When set, this option results in execution times being appended on to computed explanations during postprocessing.

debug_info (false): When true, this option results in minimal debugging information being written onto the user output during execution.

Limitations

ProLogICA avoids the need for more complex constraint solving techniques by precluding the selection of non-ground abducibles in abductive computations. This brings with it the possibility of floundering - where a branch of the search space cannot be explored as none of the goal literals can be selected. In consistency computations floundering is avoided through the use of dynamic integrity constraints. In abductive computations floundering is partially avoided by postponing the selection on non-ground abducibles. But, the programmer must ensure that variables are sufficiently grounded so floundering does not prune useful solutions. (This is analogous to the use of standard Prolog without constructive negation.) *ProLogICA*'s preprocessing facility does not use clever heuristics in the unfolding of integrity constraints. No subsumption checking is performed when issuing dynamic integrity constraints. More user-friendly debugging and visualisation tools are needed.

Applications[3]

Temporal Reasoning in Multi-Agent Systems

ProLogICA has been used within a multi-agent systems project called *Societies of Computees* (SOCS) (Alberti *et al.* 2005; Stathis *et al.* 2004) to implement the Temporal Reasoning (TR) component of its computees (autonomous agents). This aspect of the computee architecture allows an agent to maintain a coherent model of its evolving environment and infer temporal information missing from its necessarily incomplete view of the external world.

The TR representation is based on a variant of the Abductive Event Calculus (Eshghi 1988; Denecker, Missiaen, & Bruynooghe 1992; Shanahan 2000), based on the Language $\mathcal{E}$ (Kakas & Miller 1997), which models how the truth or falsity of certain fluents (properties) vary over time as a result of certain events (actions) happening, or not. Atoms of the form $holds\_at(F,T)$ denote that the fluent F is true at time T, while atoms of the form $happens(A,T)$ denote that the action A occurs at time T.

The abductive theory is formed of two parts. The "domain independent" axioms, shown below, formalize how fluents are caused by events and how they persist forwards in time. For an agent to believe a fluent F holds at time T, either (i) an event A previously occurred that initiated F, or (ii) F was observed to hold in the past, or (iii) F is assumed (abduced) to hold initially – providing that no intervening event terminates F. Negative fluents are represented $neg(F)$ and are treated symmetrically.

$$holds\_at(F,T) \leftarrow happens(A,T_1), \\ T_1 < T, initiates(A,T_1,F), not\ clipped(T_1,F,T).$$

$$holds\_at(F,T) \leftarrow observed(F,T_1), \\ T_1 \leq T, not\ clipped(T_1,F,T).$$

$$holds\_at(F,T) \leftarrow assume\_holds(F,0), \\ not\ clipped(0,F,T).$$

$$clipped(T_1,F,T_2) \leftarrow happens(A,\ T), \\ T_1 \leq T < T_2, terminates(A,T,F).$$

The second part of the theory contains the domain specific axioms stating which actions initiate and terminate which fluents under which conditions. For example, if our domain involves the parking of cars in a car park, we may have the following axioms – stating that the action of parking a car will result in the car being in the car park if there is a free place or, even if their is no free space, will still result in the car being in the car park providing there is a parking attendant (with access to specially reserved places):

$$initiates(park(Car),T,in\_car\_park(Car)) \leftarrow \\ holds\_at(free\_place,T).$$

$$initiates(park(Car),T,in\_car\_park(Car)) \leftarrow \\ holds\_at(neg(free\_place),T), \\ holds\_at(park\_attendant,T).$$

There is one domain independent integrity constraint

$$\bot \leftarrow holds\_at(F,T), holds\_at(neg(F),T).$$

stating that a fluent and its negation cannot hold at the same time; and there may be other domain specific integrity constraints constraining evolution of one or more fluents over time. There is just one abducible predicate $assume\_holds$ expressing the fact that we have incomplete information only for fluents. As the computee operates, it also receives information from its environment in the form of a narrative containing observations about particular fluents holding and certain events happening at specific times. For example, an agent may get the following narrative.

$$observed(park\_attendant,1).$$

$$happens(park(car1),3).$$

After adding these observations to its knowledge base, the agent uses an abductive engine to infer missing knowledge about the evolution of fluents. In the example above the agent should infer that at time 3 the attendant and car are in the car park, but there may or may not

[3] In contrast to the previous two sections, this section uses the notation of classical logic to express clauses and theories: with the connective $\leftarrow$ in place of `:-` and with $\bot$ in place of `ic`.

Query	0	1	2	3	4	5	6	7	8	9	Avg.
CIFF (v2)	54.2	52.1	95.5	44.3	94.3	45.0	44.7	49.8	101	101	68.3
ProLogICA (v0)	1.17	1.38	3.31	2.12	2.51	2.19	2.22	2.37	2.27	2.29	2.18
ProLogICA (v1)	0.16	0.16	0.28	0.16	0.17	0.16	0.16	0.17	0.17	0.17	0.18

Table 1: Execution time (in seconds) of CIFF vs. *ProLogICA* for 10 queries in the SOCS temporal reasoning domain. — Experiments using the ALP model developed in (Alberti *et al.* 2005).

be a free place. Thus SOCS agents distinguish sceptical conclusions like $holds\_at(in\_car\_park(car1), 3)$ and $holds\_at(park\_attendant, 3)$ from credulous conclusions like $holds\_at(free\_place, 3)$. Formally a ground atom $holds\_at(f, t)$ is *credulously* entailed iff it has an abductive explanation; and it is *sceptically* entailed iff it is credulously entailed but its complement $holds\_at(neg(f), t)$ is not.

The SOCS consortium has implemented a computational agent platform, called PROSOCS (Stathis *et al.* 2004), that is based on the SOCS architecture. Since this platform is parametric on the underlying abductive procedure, it has enabled the SOCS consortium to carry out experiments with different abductive systems in order to achieve the most effective setup. Such experiments have compared the performance of *ProLogICA* with that of an alternative ALP system, CIFF (Endriss *et al.* 2004), which is used elsewhere in the SOCS project for agent planning tasks.

Table 1 compares the execution times of *ProLogICA* and CIFF for 10 different queries in a test-bed domain. The table shows execution times, in seconds, on a Pentium III desktop PC. For completeness, Table 1 includes data for two versions of *ProLogICA*: the latest version (v1) and an earlier version (v0) that was originally used in (Alberti *et al.* 2005) under the name ASLD(N,IC). On average, *ProLogICA* (v0) was 30 times faster than CIFF (v2), and the *ProLogICA* (v1) was 300 times faster than CIFF (v2).

According to (Alberti *et al.* 2005), *ProLogICA* is marginally slower than an unsound optimisation of CIFF, called *triggering*, that involves suppressing the unfolding of any $holds\_at$ atoms that appear within an integrity constraint. But, while their performance is comparable, *ProLogICA* without triggering produced only correct answers, while CIFF with triggering did not. Overall, *ProLogICA* was found to exhibit a high level of robustness, so that, in spite of its lightweight nature, it was found to be "significantly more effective" in this application.

Inference of Genetic Regulatory Networks

ProLogICA has been applied to the task of inferring gene interactions using data from microarray experiments on M. tuberculosis using the method developed in (Papatheodorou, Kakas, & Sergot 2005). Published biological data measures changes in levels of gene expression in response to genetic modifications, such as knocking-out (turning off) or over-expressing (turning on) genes, and environmental stresses, such as changes in temperature or nutrient concentration. The aim of these experiments is to obtain insights into the complex feedback mechanisms by which the product of one gene affects the expression of another. As these interactions cannot be observed directly, they must be inferred indirectly from gene expression levels; and the complexity of interactions and volume of experimental data means that automatic methods are needed in this task.

As explained in (Papatheodorou, Kakas, & Sergot 2005), statistical analysis of the raw microarray data from each experiment reveals significant changes in the expression of particular genes. These differences are represented by atoms of the form $increases\_expression(E, G)$ and $reduces\_expression(E, G)$, where E is an experiment and G is a gene. Similarly, the experimental conditions are recorded by atoms of the form $knocks\_out(E, G)$ and $over\_expresses(E, G)$. Given a set of experimental observations and the general model of gene interactions described below, *ProLogICA* was used to abduce atoms of the form $induces(F, G)$ or $inhibits(F, G)$ stating that one gene F induces (increases) or inhibits (reduces) the expression of another G. These elementary hypotheses can then be built into paths and networks of gene interactions.

The ALP theory developed by (Papatheodorou, Kakas, & Sergot 2005) models the general principles by which changes in gene expression can be explained by possible gene interactions. Because these principles are (assumed to be) independent of any particular network topology, they are succinctly and modularly expressed commonsense laws such as one below, which states that an increase in the expression of a gene X in an experiment E can be explained by hypothesising that another gene G, which is knocked-out in E, is an inhibitor of X, providing that there are no other effects that could better account for the increase of X. Here, $incr\_affected\_by\_other\_gene(E, G, X)$ means that the increase of X in E is due to some gene other than G and $incr\_affected\_by\_EnvFact(E, X)$ means it is due to an environmental change.

$$
\begin{aligned}
&increases\_expression(E, X) \leftarrow \\
&\quad knocks\_out(E, G), inhibits(G, X), \\
&\quad not\ incr\_affected\_by\_other\_gene(E, G, X), \\
&\quad not\ incr\_affected\_by\_EnvFact(E, X).
\end{aligned}
$$

$$
\begin{aligned}
&incr\_affected\_by\_other\_gene(E, G, X) \\
&\quad reduces\_expression(E, Gx), \\
&\quad Gx \neq X, Gx \neq G, \\
&\quad related\_genes(Gx, G), inhibits(Gx, X).
\end{aligned}
$$

As formalised in the second rule above, an alternative explanation for the increase in the expression of X could be some other gene Gx whose expression is reduced in experiment E and which inhibits X. The $related\_genes$ predicate is one of several mechanisms in the model for declaratively constraining the search space by exploiting background

Query	0	1	2	3	4	Avg.
KM	>7000	>7000	>7000	>7000	>7000	>7000
ProLogICA (v0)	0.47	12.4	12.4	281	282	118
ProLogICA (v1)	0.03	0.08	0.08	0.16	0.13	0.09

Table 2: Execution time (in seconds) of KM vs. *ProLogICA* for 5 queries in the genetic regulatory network domain. — Experiments using the ALP model developed in (Papatheodorou, Kakas, & Sergot 2005).

biological information about the relationships between different genes. There are about a dozen background rules covering the different combinations of conditions that could arise. The model also contains several integrity constraints like the one below which states that a gene cannot not both induce and inhibit another. Other constraints, not shown, restrict the interactions of genes located on the same operon.

$$\bot \leftarrow induces(F, G), inhibits(F, G).$$

Given this simple model of gene interactions *ProLogICA* was used to analyse data from 5 microarray experiments involving genes thought to be implicated in the response of M. tuberculosis to heat shock. (Papatheodorou, Kakas, & Sergot 2005) report how the system was able to rediscover several gene interactions already known in the literature and to suggest further experiments for investigating other possible interactions. In addition, the highly recursive nature of the model provided a challenging application on which to evaluate the efficiency of *ProLogICA*. Table 2 shows the time taken for *ProLogICA* compared to a standard implementation of KM (Kakas & Mancarella 1998) – similarly depth-bounded for a fair comparison – to find all solutions to 5 different queries each containing up to 5 goals. Whereas KM did not terminate on any of the queries (or return any answers) after two hours, *ProLogICA* computed all minimal solutions in less than two tenths of a second.

Clinical Management of HIV/AIDS

ProLogICA has been used in a ALP approach for assisting medical practitioners in the selection of anti-retroviral drugs for patients infected with Human Immunodeficiency Virus (HIV) (Ray *et al.* 2006). The difficulty is the ability of HIV to accrue genetic mutations that confer resistance to known medications. For this reason, clinical guidelines advocate the use of laboratory *resistance tests* to help identify which mutations a patient is carrying and predict which drugs they are most likely resistant to. But, these tests have several drawbacks: first, they cost between one and two hundred dollars; second, they require medical access that most infected individuals do not have; and third, they cannot reliably detect *minority strains* of HIV (often comprising up to 20% of a patient's viral population) which may be harbouring drug-resistant mutations. Thus clinicians must carefully study a patient's medical history for any additional clues that may suggest the presence or absence of mutations.

To address this task, we developed a novel "in-Silico" Sequencing System built on top of *ProLogICA* to assist in the interpretation of resistance tests and, more importantly, to infer likely mutations and drug resistances in the absence of such tests. These inferences are made using a patient's clinical history, which details previous treatment failures and successes, and a set of rules expressing which mutations confer resistance to which drugs. We do not invent these associations; instead we download the same rules used by the resistance testing laboratories to predict drug resistances from mutations determined by genetic sampling of clinical isolates from infected patients. But, instead of using them deductively (i.e. *forwards*) to predict likely drug resistances implied by observed mutations, we use the rules abductively (i.e. *backwards*) so as to explain observed drug resistances in terms of likely mutations.

The domain knowledge contains 64 rules (obtained from the French national AIDS research agency) for 16 FDA-approved drugs. These rules are represented using head atoms of the form $resistant(P, T, D)$ to denote that patient a P is resistant to a drug D at time T, and body atoms of the form $mutation(P, T, M)$ to denote that patient P is carrying the mutation M at time T. A typical rule is shown below for the drug zidovudine and states that the patient is resistant to zidovudine if he is carrying at least one of the three mutations $151M$, $69i$ or $215YF$. The notation $151M$ refers to a mutation at codon 151 in the viral genome whereby the wild type amino acid (according to a standard reference strain of HIV) is replaced by the mutant methionine M. Similarly, $69i$ denotes the *insertion* of an amino acid at position 69, and $215YF$ is a compact way of writing $215Y$ or $215F$.

These rules exploit the meta-predicate $atleast/2$ to ensure the satisfaction of a given number of goals from a given list using the fewest number of abducibles. The advantage of using this predicate is that, when given the goal $resistant(P, T, zidovudine)$ the system will not even consider showing the first two mutations if it has previously established $mutation(P, T, "215YF")$.

$$\begin{array}{l} resistant(P, T, zidovudine) \leftarrow \\ \quad atleast(1, [mutation(P, T, '151M'), \\ \quad\quad mutation(P, T, '69i'), mutation(P, T, '215YF')]). \end{array}$$

Because the treatment of HIV usually involves the prescription of powerful cocktails of at least three anti-retrovirals, we provide two clauses to link the possible ineffectiveness (resp. effectiveness) of a combination treatment with the resistance (resp. non-resistance) to one of the drugs D in the set of drugs S.

$$\begin{array}{l} effective(P, T, S) \leftarrow \\ \quad in(D, S), not\ resistant(P, T, D). \end{array}$$

$$\begin{array}{l} ineffective(P, T, S) \leftarrow \\ \quad not\ effective(P, T, S). \end{array}$$

Query	0	1	2	3	4	5	6	7	8	9	Avg.
KM	197,760	5,439	>85,211	56,916	435,600	9,828	28,231	263	263	15,022	>83,453
ProLogICA	5,492	1,208	56,165	4,062	3,570	1,979	2,395	74	74	1,436	7,646
+atleast	4,770	1,164	25,625	3,075	3,238	1,939	2,375	74	74	73	4,241
+minimal	1,283	51	333	138	1,628	191	1,938	73	73	72	578

Table 3: Number of hypotheses returned by KM vs. *ProLogICA* for 10 patients in the HIV resistance domain. — Experiments using the ALP model developed in (Ray *et al.* 2006).

To correctly model the dynamics of drug resistance, we need an integrity constraint that captures a key biological principle underlying this work: namely, the persistence of mutations. As formalised below, this constraint states that if a patient P is carrying a mutation M at time T, then he or she must also be carrying that mutation at all later times T'.

$$\bot \leftarrow mutation(P,T,M),\ T' \geq T,\ not\, mutation(P,T',M).$$

Declaring the predicate $mutation/3$ as abducible, this theory can explain a history of treatment successes and failures in terms of mutations the patient is (or is not) carrying. For example, one explanation of the observations below is that $p56$ was *not* carrying mutations $151M$, $69i$, $215YF$, $41L$, $67N$, $70R$ and $210W$ at time 1, but *was* carrying mutation $215YF$ at time 2.

$$effective(p56, 1, [zidovudine, lamivudine, indinavir]).$$

$$ineffective(p56, 2, [zidovudine, lamivudine, indinavir]).$$

An interesting feature of this approach is that, since we can never really be sure which mutations a patient is carrying, there is no point in looking for an optimal hypothesis. Thus, in contrast to previous applications of ALP, we accept that multiple explanations are inevitable and seek to develop ways of extracting useful information from them. For this purpose, we developed a domain specific system that uses *ProLogICA* to compute a set of explanations for a patient's clinical history and then analyses those explanations so as to predict which drugs the patient may be resistant to. This is done by ranking each drug by to the number of explanations that imply its resistance. As expected, a major computational obstacle is the sheer number of explanations.

A typical patient taking 3 different drugs at each of 10 time points results in more solutions (tens of thousands) than can be conveniently analysed. To efficiently reduce the number of redundant explanations, we applied *ProLogICA*'s minimisation routine incrementally after each time point. The reduction in the number of solutions for real clinical data from 10 HIV-infected patients is shown in Table 3. On average, *ProLogICA* provided an order of magnitude improvement of a standard implementation of KM (Kakas & Mancarella 1998). Moreover, the use of *ProLogICA*'s $atleast$ predicate resulted in another factor of 2 improvement, and the incremental application of minimality produced further order of magnitude improvement (and a significantly faster computation).

Conclusions, Related and Future Work

ProLogICA is a practical system for ALP whose utility has been shown by recent applications in the areas of multi-agent systems (Alberti *et al.* 2005) and bioinformatics (Ray *et al.* 2006; Papatheodorou, Kakas, & Sergot 2005). It is a lightweight system that is appropriate for many tasks which do not require the ability to solve complex optimisation problems, and is versatile enough to be used as a component of a larger reasoning system (Ray 2005).

In many respects, *ProLogICA* takes a step back from some recent ALP systems such as ACLP (Kakas, Michael, & Mourlas 2000), SLDNFAC (Denecker & Schreye 1998), A-system (Kakas, Van Nuffelen, & Denecker 2001), and CIFF (Endriss *et al.* 2004). In these systems, the emphasis is on integrating abduction with constraint solving to enable the solution of complex optimisation problems. These systems work by producing non-ground (existentially quantified) explanations with an associated set of constraints and then rely on the efficient satisfiability check of an external constraint solver. By contrast, *ProLogICA* exploits the simplicity of early procedures for ground abduction (Eshghi & Kowalski 1989; Kakas & Mancarella 1990a) by overcoming some of their well-known inefficiencies.

ProLogICA was motivated by recent attempts to apply ALP in areas of systems biology where it seems that highly specific explanations and models are required. Here, the task is to identify one or more hypotheses from a (large) set of ground hypotheses in order to explain some experimental observations. *ProLogICA* was originally developed as the abductive component of the HAIL learning system, which aims to support scientific theory development through the integration of abductive and inductive logic programming within a common reasoning framework. In this setting, ALP is only required to construct a specific account of the input observations to seed further generalisations. *ProLogICA* is well suited to both of these settings.

An alternative framework for performing ground abduction in Logic Programming is the framework of Answer Set Programming (ASP) (Lifschitz 1999). ASP systems such as SMODELS (Simons, Niemelä, & Soininen 2002) and DLV (Dell'Armi *et al.* 2001) have been continuously improving and offer nowadays a very efficient computational paradigm for many problem domains. In contrast to ALP approaches, which are top-down query-driven, ASP approaches are bottom-up data-driven. They operate by translating extended logic programs into ground propositional theories and employing satisfiability solvers to compute the models of the original program.

Although *ProLogICA* has proven to be an effective ALP system for the applications considered in this paper, a systematic comparison of ALP and ASP remains to be carried out. Using the translation introduced in (Satoh & Iwayama 1991) – whereby ALP problems can be encoded into the ASP formalism – would enable a comparison of the two approaches on the problems described above. Such a comparison may also lead to new approaches for combining different aspects of ALP and ASP technologies. For example, ALP could utilise ASP methods in its integrity constraint checking phase after it has generated a possible explanation.

Acknowledgments

We acknowledge the help of Andrea Bracialli who kindly provided the experimental data in Table 1. We are grateful to Irene Papatheodorou for her valuable comments regarding the gene interaction model.

References

Alberti, M.; Bracciali, A.; Chesani, F.; Ciampolini, A.; Endriss, U.; Gavanelli, M.; Guerri, A.; Kakas, A.; Lamma, E.; Lu, W.; Mancarella, P.; Mello, P.; Milano, M.; Riguzzi, F.; Sadri, F.; Stathis, K.; Terreni, G.; Toni, F.; Torroni, P.; and A.Yip. 2005. Experiments with animated societies of computees. Technical Report D14, SOCS Consortium. http://lia.deis.unibo.it/Research/Projects/SOCS/.

Dell'Armi, T.; Faber, W.; Ielpa, G.; Koch, C.; Leone, N.; Perri, S.; and Pfeifer, G. 2001. System description: Dlv. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, 424–428.

Denecker, M., and Schreye, D. D. 1998. SLDNFA: An Abductive Procedure for Abductive Logic Programs. *Journal of Logic Programming* 34(2):111–167.

Denecker, M.; Missiaen, L.; and Bruynooghe, M. 1992. Temporal reasoning with abductive event calculus. In *Proceedings of the 10th European conference on Artificial intelligence*, 384–388.

Endriss, U.; Mancarella, P.; Sadri, F.; Terreni, G.; and Toni, F. 2004. The CIFF Proof Procedure for Abductive Logic Programming with Constraints. In *Proceedings of the 9th European Conference on Logics in Artificial Intelligence*, 31–44. Springer Verlag.

Eshghi, K., and Kowalski, R. 1989. Abduction compared with negation by failure. In Levi, G., and Martelli, M., eds., *Proceedings of the 6th International Conference on Logic Programming*, 234–254. MIT Press.

Eshghi, K. 1988. Abductive planning with event calculus. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*, 562–579.

Kakas, A., and Denecker, M. 2002. Abduction in Logic Programming. In Kakas, A., and Sadri, F., eds., *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, volume 2407 of *Lecture Notes in Computer Science*. Springer. 402–436.

Kakas, A., and Mancarella, P. 1990a. Database Updates through Abduction. In *Proceedings of the 16th International Conference on Very Large Databases*, 650–661.

Kakas, A., and Mancarella, P. 1990b. Generalized Stable Models: a Semantics for Abduction. In *Proceedings of the 9th European Conference on Artificial Intelligence*, 385–391. Pitman.

Kakas, A., and Mancarella, P. 1998. KM ALP procedure. At http://www.cs.ucy.ac.cy/aclp/alp_int.pl.

Kakas, A., and Michael, A. 2001. An abductive-based scheduler for air-crew assignment. *Applied Artificial Intelligence* 15(3):333–360.

Kakas, A., and Miller, R. 1997. A simple declarative language for describing narratives with actions. *Journal of Logic Programming* 31:157–200.

Kakas, A.; Kowalski, R.; and Toni, F. 1992. Abductive Logic Programming. *Journal of Logic and Computation* 2(6):719–770.

Kakas, A.; Michael, A.; and Mourlas, C. 2000. ACLP: Abductive constraint logic programming. *Journal of Logic Programming* 44(1-3):129–177.

Kakas, A.; Van Nuffelen, B.; and Denecker, M. 2001. A-system: Problem solving through abduction. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, 591–596.

Lifschitz, V. 1999. Action languages, answer sets and planning. In *The Logic Programming Paradigm: a 25 year perspective*, 357–373. Springer.

Papatheodorou, I.; Kakas, A.; and Sergot, M. 2005. Inference of gene relations from microarray data by abduction. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 3662 of *Lecture Notes in Computer Science*, 389–393.

Ray, O.; Antoniades, A.; Kakas, A.; and Demetriades, I. 2006. Abductive Logic Programming in the Clinical Management of HIV/AIDS. In *Proceedings of the 17th European Conference on Artificial Intelligence*. to appear.

Ray, O. 2005. *Hybrid Abductive-Inductive Learning*. Ph.D. Dissertation, Department of Computing, Imperial College London, UK.

Satoh, K., and Iwayama, N. 1991. Computing Abduction by Using the TMS. In *Proceedings of the 8th International Conference on Logic Programming*, 505–518. MIT Press.

Shanahan, M. 2000. An abductive event calculus planner. *Journal of Logic Programming* 44(1-3):207–240.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1-2):181–234.

Stathis, K.; Kakas, A.; Lu, W.; Demetriou, N.; Endriss, U.; and Bracciali, A. 2004. Prosocs: A platform for programming software agents in computational logic. In Müller, J., and Petta, P., eds., *Proceedings of the 4th International Symposium 'From Agent Theory to Agent Implementation'*.

Van Nuffelen, B., and Denecker, M. 2000. Problem solving in ID-logic with aggregates: some experiments. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*, Session on Abduction, 1–5.