LOGIC PROGRAMMING AND THEOREM PROVING

LOGIC PROGRAMMING AND THEOREM PROVING

LOGIC PROGRAMMING

$\mathcal{A}$-System: Problem Solving through Abduction

Antonis C. Kakas
Dep. of Computer Science
University of Cyprus
B.O.Box 20537
CY-1678 Nicosia, Cyprus
antonis@ucy.ac.cy

Bert Van Nuffelen and Marc Denecker
Dep. of Computer Science
K.U.Leuven
Celestijnenlaan 200A
B-3000 Leuven, Belgium
{bertv, marcd}@cs.kuleuven.ac.be

Abstract

This paper presents a new system, called the $\mathcal{A}$-System, performing abductive reasoning within the framework of Abductive Logic Programming. It is based on a hybrid computational model that implements the abductive search in terms of two tightly coupled processes: a reduction process of the high-level logical representation to a lower-level constraint store and a lower-level constraint solving process. A set of initial "proof of principle" experiments demonstrate the versatility of the approach stemming from its declarative representation of problems and the good underlying computational behaviour of the system. The approach offers a general methodology of declarative problem solving in AI where an incremental and modular refinement of the high-level representation with extra domain knowledge can improve and scale the computational performance of the framework.

1 Introduction

Over the last two decades it has become clear that abduction can play a central role in addressing a variety of problems in Artificial Intelligence. These problems include diagnosis [Poole *et al.*, 1987; Console *et al.*, 1996], planning [Missiaen *et al.*, 1995; Kakas *et al.*, 2000; Shanahan, 2000] knowledge assimilation and belief revision, [Inoue and Sakama, 1995; Pagnucco, 1996] multi-agent coordination,[Ciampolini *et al.*, 2000; Kowalski and Sadri, 1999] and knowledge intensive learning [Muggleton, 2000; Mooney, 2000].

The essential feature of this abductive approach to problem solving is the fact that it allows the application problems to be formalized directly in their high-level declarative representation. A close link therefore emerges between declarative problem solving in AI and the logical reasoning of abduction. However, despite this variety of applications for abduction and its potential benefits it has not been easy to develop general systems for abduction that are computationally effective for problems of practical size and complexity.

This paper presents a new system, called the $\mathcal{A}$-System, that supports abductive reasoning within the framework of Abductive Logic Programming (ALP) [Kakas *et al.*, 1998; Denecker and Kakas, 2000]. In this framework problems are represented in a high-level declarative way with logic programming rules and classical first-order sentences (integrity constraints). Our work aims to examine the possibility of developing ALP into a declarative problem solving framework, suitable for a variety of AI problems, that is computationally viable for problems of practical scale.

A principled implementation of the $\mathcal{A}$-System is developed based on a hybrid computational model that formalizes the abductive search for a solution in terms of two interleaving processes: the logical reduction of the high-level representation of the problem and lower-level constraint solving. The abductive search is linked tightly to the construction of an associated constraint store.

To validate our approach we have carried out a set of "proof of principle" experiments with an initial implementation of the system that aim to test (a) the general underlying computational behaviour of the $\mathcal{A}$-System on different domains and (b) the flexibility of the framework to incorporate additional problem specific knowledge in a modular and computationally enhancing way. In particular, some of these experiments are designed to test the extend to which the high-level representation of problems can be computationally improved via extra domain knowledge added to it. The experiments considered include constraint satisfaction problems and standard AI Planning Systems Competition problems.

The $\mathcal{A}$-System has been developed as a follow up of two earlier ALP systems, the ACLP system [Kakas *et al.*, 2000] and SLDNFAC [Denecker and Schreye, 1998], bringing together features from these two systems. The main development in the design of the $\mathcal{A}$-System over these previous systems is the fact that the non-determinism in the abductive computation is now made explicit allowing the possibility of implementing this as a form of parameterized heuristic search coupled with a process of deterministic propagation of the state of computation. The $\mathcal{A}$-System is available from www.cs.kuleuven.ac.be/~dtai/kt.

2 Declarative Programming with Abduction

Declarative Problem Solving with a high-level representation of the problem at hand is closely related to *abduction*. The reason for this stems from the fact that in a declarative representation where one describes the expert knowledge on the problem domain rather than some method for its solution often, the task of solving the problems consists in filling the

missing information from this representation pertaining to the goal at hand. Typically, in a logical framework of representation the solution consists in finding the extension of some predicate(s) which are incompletely specified in the representation. The high-level theory describing the problem is then extended to a new one such that the problem goal is satisfied.

Computing such extensions of the theory representing our problem is an *abductive task*. Indeed, abduction as a problem solving method assumes that the general data structure for the solution to a problem (or solution carrier) is at the predicate level and hence that a solution is described in the same terms and level as the problem itself.

A framework for problem solving with abduction therefore needs to be expressing enough to allow high-level declarative representations of complex problems with missing or incomplete information. At the same time for such a framework to have a practical value it should provide ways of improving the computational effectiveness of this high-level representation for any particular problem. One such framework that combines representational expressiveness and computational flexibility is that of Abductive Constraint Logic Programming (ACLP) a framework that integrates together Abductive Logic Programming (ALP) with methods from Constraint Logic Programming (CLP). The A-system is developed within this framework. Let us briefly review the underlying framework of ALP.

A recent way to formalize ALP is to view this as a special case of ID-logic [Denecker, 2000], a logic extending classical logic with inductive definitions. Let a language of predicates, $\mathcal{L}$, be given consisting of three disjoint types of predicates: (i) P_D the *defined* predicates, (ii) $\mathcal{A}$ the *open or abducible* predicates and (iii) $\mathcal{C}$ the *constraint* predicates. Then a theory in ALP is defined as follows.

Definition 2.1 *An abductive theory is a triple* (P, A, IC) *where:*

- P *is a constraint logic program where only defined predicates appearing in the head of these rules.*
- A *is a set of ground abducible atoms over open predicates.*
- IC *is a set of first order formulae over* $\mathcal{L}$*, called* integrity constraints.

The representation of an application problem in an abductive theory (P, A, IC) splits, in this view of ALP in terms of ID-logic, into two parts. The set of rules in P represents the expert's strong *definitional knowledge* of the problem, i.e. knowledge which fully determines one or a group of predicates, the *defined* predicates, in terms of other *open* predicates. Each rule represents a *case* in which the defined predicate in the head will be true; the program P is an exhaustive enumeration of the cases. The open or *abducible* predicates have no definition. But some general information about them may be available indirectly through the expert's weaker *assertional knowledge* of the problem. This is represented by the theory IC of *integrity constraints*. The set A enumerates all the possible missing "values" for the open predicates. Abductive solutions are build from A giving a ground (partial) definition of these open predicates.

The constraint predicates are defined, as in CLP, by an underlying constraint theory which is independent of any particular abductive theory (P, A, IC). We will assume that the constraint theory is a finite domain theory that also includes equality over logical terms. The formal details of this are beyond the scope of this paper.

In ALP the abductive problem task is defined as follows.

Definition 2.2 *Given an abductive theory* (P, A, T) *and some query* Q*, consisting of a conjunction of literals over* $\mathcal{L}$*, an* abductive solution (or explanation) *for* Q *is a set* $\Delta \subseteq A$ *of ground abducible atoms together with an answer substitution* θ *such that* $P \cup \Delta$ *is consistent and:*

- $P \cup \Delta \models \forall(\theta(Q))$
- $P \cup \Delta \models IC$.

This definition is generic in that it defines the notion of an abductive solution in terms of any given semantics of standard (constraint) Logic Programming (LP). Each particular choice of semantics defines its own entailment relation $\models$ and hence its own notion of what is an abductive solution. In the context of ID-logic, its inductive definition semantics essentially coincides with the well-founded model semantics [Gelder *et al.*, 1991] for LP when this is a two valued model[1]. In the rest of this paper we will be adopting this well-founded model semantics for LP for the development of our $\mathcal{A}$-System.

A computed abductive solution Δ therefore gives a ground definition of the open predicates which in turn through the logic program, $P \cup \Delta$, extends this to the defined predicates.

As an example of an abductive theory we give below a part of the theory representing a planning domain. In this, $drive_vehicle$, is the only open predicate, $<, \leq, \neq$ are constraint predicates and all others are defined predicates. The predicates $vehicle$, $location$ and $time$ are simple predicates defining the finite domain of variables, e.g. using CLP notation $vehicle(Truck) :- Truck\ in\ 1..10$ for a problem where we have ten trucks.

$$
\begin{array}{l}
vehicle_at(Truck, Loc, T) :- \\
\quad init_vehicle_at(Truck, Loc), \\
\quad \neg clipped_vehicle_at(Truck, Loc, 0, T)^2. \\
vehicle_at(Truck, Loc, T) :- \\
\quad drive_vehicle(Truck, Loc, E), \\
\quad E < T, \\
\quad \neg clipped_vehicle_at(Truck, Loc, E, T).
\end{array}
$$

$$
\begin{array}{l}
clipped_vehicle_at(Truck, Loc1, E, T) :- \\
\quad drive_vehicle(Truck, Loc2, C), \\
\quad Loc2 \neq Loc1, \\
\quad E \leq C, C < T.
\end{array}
$$

$$
\begin{array}{l}
prec_vehicle(Truck, Loc, T) :- \\
\quad vehicle(Truck), \\
\quad location(Loc), \\
\quad time(T), \\
\quad petrol_in(Truck, T).
\end{array}
$$

$$
\begin{array}{l}
\forall Trc, Loc, T. \\
drive_vehicle(Trc, Loc, T) \rightarrow prec_vehicle(Trc, Loc, T).
\end{array}
$$

[1] The formal details of this are beyond the scope of this paper.

The initial state of the problem is given by a set of facts in the logic program P including statements of the form $init_vehicle_at(Truck, Loc)$, e.g. $init_vehicle_at(truck1, city1_1)$.

3 $\mathcal{A}$-System and Abductive Search

The computation of an abductive solution in the $\mathcal{A}$-System can be seen as a process of reduction of the query, Q, to a set of abducible hypotheses on the open predicates together with an associated constraint store $\mathcal{C}_\mathcal{S}$ of constraints over the general finite domain constraint theory supported by the particular ALP framework. This is similar to the computation of CLP with two important differences.

First the reduction involves through the hypotheses also a reduction of the integrity constraints IC. Effectively, when a new abductive hypothesis is made the integrity constraints need to be re-evaluated to remain true. This satisfaction of the IC can result in new constraints for the constraint store $\mathcal{C}_\mathcal{S}$ and possibly new goals at the same level as the original query. The second difference with CLP is the way that the constraint store $\mathcal{C}_\mathcal{S}$ is used in this reduction. As we will see in the next subsection, this is not merely a passive store of constraints to be evaluated at the end of the reduction but it is used actively during the reduction to affect the search. For example, it enables reductions to be pruned early by setting new constraints in $\mathcal{C}_\mathcal{S}$ provided that this remains satisfiable.

3.1 Abductive Inference in the $\mathcal{A}$-System

The logical reduction of a query Q in the $\mathcal{A}$-System can be defined as a derivation for Q through a rewriting process between states. We give here in brief the formal details of this.

A computational restriction on the form of the integrity constraints, IC, of an abductive theory (P, A, IC) is imposed in the $\mathcal{A}$-System. These sentences must be (or should be transformed to classically equivalent sentences) of the form: $\forall\neg(l_1, \ldots, l_n)$ where each l_i is a literal. Such an integrity constraint will also be called a denial or a negative goal and will be denoted by $\leftarrow l_1, \ldots, l_n$. Also to simplify the presentation, we assume that each rule in P is of the form $p(\overline{X}) \leftarrow B[\overline{X}, \overline{X}_i]$ where all variables are (implicitly) universally quantified, referred to as *free* variables, and $B[\overline{X}, \overline{Y}]$ is a conjunction of literals with free variables $\overline{X}$ occurring in the head and $\overline{Y}$ occurring only in the body. Rules are therefore in homogeneous form where no non-variable term can appear in their head e.g. a rule $p(c) \leftarrow q(Y)$ will be written equivalently as $p(X) \leftarrow X = c, q(Y)$.

A *state* $\mathcal{S}$ of the derivation consists of two types of elements: a set of literals (possibly with free variables), called positive goals, and a set of denials, called negative goals, of the form $\forall\overline{X}. \leftarrow F[\overline{X}, \overline{Y}]$ where $F[\overline{X}, \overline{Y}]$ is a conjunction of literals and $\overline{Y}$ are free variables. $\Delta(\mathcal{S})$ denotes the set of abducible atoms in $\mathcal{S}$, i.e. positive goal atoms whose predicate is an open predicate and $C(\mathcal{S})$ denotes the set of positive constraint atoms in $\mathcal{S}$. A derivation for Q starts with an initial state consisting of the literals of the query Q as positive goals and all the denials in IC as negative goals. The rewriting derivation proceeds by selecting a literal in a goal of $\mathcal{S}$ and applying a suitable inference rule yielding a new state. The main inference rules are given by the following rewrite rules (different elements of $\mathcal{S}$ are separated by ";"):

- $p(\overline{t}) \Rightarrow B_i[\overline{t}]$ if p is a defined predicate and $p(\overline{X}) \leftarrow B_i[\overline{X}]$ a rule.
- $p(\overline{t}) \Rightarrow \overline{t} = \overline{s}$ if p is an open predicate and $p(\overline{s}) \in \mathcal{S}$.
- $\neg A \Rightarrow\ \leftarrow A$.
- $\forall\overline{X}. \leftarrow p(\overline{t}), Q \Rightarrow F_1; ...; F_n$ (p a defined predicate) where $F_i = \forall\overline{X}', \overline{X}_i. \leftarrow B_i[t, \overline{X}_i], Q$ for each rule $p(\overline{Y}) \leftarrow B_i[\overline{Y}, \overline{X}_i]$, ($\overline{X}' \subseteq \overline{X}$).
- $\forall\overline{X}. \leftarrow p(\overline{t}), Q \Rightarrow \forall\overline{X}. \leftarrow p(\overline{t}), Q; F_1; ...; F_n$ (p an open predicate) where $F_i = \forall\overline{X}'. \leftarrow \overline{t} = \overline{s}, Q$ for each $p(\overline{s}_i)$ an abduced atom in $\mathcal{S}$, ($\overline{X}' \subseteq \overline{X}$).
- $\forall\overline{X}. \leftarrow C, Q \Rightarrow \neg C$, if C is a constraint atom without universally quantified variables.
- $\forall\overline{X}. \leftarrow C, Q \Rightarrow C; \forall\overline{X}. \leftarrow Q$, if C is a constraint atom without universally quantified variables.
- $\forall\overline{X}. \leftarrow \neg A, Q \Rightarrow A$ if A does not contain universally quantified variables.

A successful derivation terminates with a state $\mathcal{S}$ such that:

1. $\mathcal{S}$ contains positive goals only of the form of abducible atoms or constraint atoms,
2. negative goals in $\mathcal{S}$ are denials containing some open atom $p(\overline{t})$ which has already been selected and resolved with each abduced atom $p(\overline{s}) \in \mathcal{S}$, and
3. the constraint store $C(\mathcal{S})$ of $\mathcal{S}$ is satisfiable.

Otherwise, the derivation flounders when universally quantified variables appear in the selected literal in a denial. If the derivation does not succeed or flounder then it fails. Note that negation as failure in P is simply re-written (via the 3rd rule) to a denial of the positive atom.

Let $\mathcal{S}$ be the final state of a successful derivation. Then any substitution θ that assigns a ground term to each free variable of $\mathcal{S}$ and which satisfies the constraint store $C(\mathcal{S})$ is called a *solution substitution* of $\mathcal{S}$. Such a substitution always exists since $C(\mathcal{S})$ is satisfiable for a successful derivation.

Theorem 3.1 *Let (P, A, IC) be an abductive theory s.t. $P \models IC$, Q a query, $\mathcal{S}$ the final state of a successful derivation for Q and θ a solution substitution of $\mathcal{S}$. Then the pair $\theta(\Delta(\mathcal{S}))$ and θ is an abductive solution of Q.*

3.2 Abductive Search

The search for an abductive solution in the $\mathcal{A}$-System based on the above proof theory, depends very closely on the computed constraint store. Typically, the abducible hypotheses made during the computation are non-ground atoms whose variables are restricted by the constraints in the associated constraint store of the computation. This constraint store plays a central role not only in carrying (together with the abducibles) the solution but also in controlling the abductive search.

In many respects the computation can be viewed as a process of constructing a constraint store from the high-level specification and query of the abductive theory as any decision that we take, e.g. which abducible to introduce or how to satisfy an integrity constraint, extends differently the current constraint store. Hence we can turn things around and use the constraint store to guide the decisions that we make in this abductive search.

The overall pattern of the abductive computation in the $\mathcal{A}$-System is as follows:

1. *Deterministic Propagation of +ve and -ve Goals*
2. *Suspend and Evaluate Choices*
   (a) *Quasi-consistency of -ve Goals*
   (b) *Evaluate Choices on +ve goals*
3. *Global set of Choices: new +ve Goals, G_{new}*
4. *If G_{new} non-empty then Return to (1)*
5. *Otherwise, Exit and Ground Solution.*

The computation starts with a phase of *deterministic propagation* where all of the current goals that have only one possible rewriting are reduced until no such goals are left in the resulting state of the computation. The purpose of this phase is two-fold: (a) to expose and collect all the choice points in the current state of the computation and (b) to propagate the construction of the constraint store with all new constraints that are necessarily imposed under the choices made so far in the previous iteration steps. The updated constraint store is checked for satisfiability during this phase as it grows. This can have a significant effect on the computation as it enables us to detect early the ensuing failure of the choices made prior to this before committing to other choices.

All the choice points exposed by this deterministic phase are suspended and a new process of their evaluation begins. This explicit handling of the non-determinism in the computation allows the use of a parameterized form of heuristic search where a variety of different types of heuristics can be used. There are essentially two types of choices in the computation. These are (1) a choice of which rule from the program to use in rewriting a positive goal (cf. first and second rewrite rule) and (2) a choice of which way to satisfy a denial (cf. last three rewrite rules).

The choice points of the second type are considered in a phase called *quasi-consistency*[3] where we try to satisfy all the denials in the current state together if possible without the introduction of new positive goals. Choices result in a new possible constraint store, that we can compute by a subsidiary phase of deterministic propagation. The "quality" of this ensuing constraint store under various criteria, e.g. tightness of its finite domain variables or minimal change in the current constraint store, forms a possible heuristic criterion. A second criterion concerns the number and complexity of the new positive goals, that a choice would produce. For example, preferring new goals which are deterministic or which introduce new abducibles whose associated denials would have a simple form are different possibilities that we are currently exploring here in the implementation of the $\mathcal{A}$-System.

[3]This name reflects the fact that at this stage of the computation the satisfaction of the integrity constraint denials is in general contingent on a set of new goals.

After the phase of quasi-consistency the computation examines the other choice points left suspended, namely positive goals which are non-deterministic, and again it can apply a form of heuristic evaluation to choose amongst the different possibilities. Once a global decision is taken on all these choice points the computation either returns to the top to repeat the process, when the choice leads to new positive goals, or otherwise, it terminates successfully by labeling the constraint store and thus grounding the abductive solution.

This *global* approach, of the $\mathcal{A}$-System to choice making where it tries to cover together as many choice points as possible results in more informed decisions leading to less backtracking and thus to a better computational performance. It also facilitates the use of heuristics by helping to parameterize them and make them available at the top level.

An important parameter of the computation is the degree to which the constraint store is examined for its satisfiability (and other quality measures) during its construction. One possibility is to do a full check of the whole store at each point where this grows. But this can be costly as a large part of this maybe unnecessary. On the other hand an incomplete check may result in the late (after several choices) recognition of the inconsistency of the constraint store. An intermediate possibility is to localizing the check to the latest addition to the store, i.e to the variables of the current constraints and other variables connected to these by constraints already in the store. This induces a form of dynamic partition of the constraint store into connected components that reflects, to a certain extend, the high-level structure of the problem and query at hand and could help minimize the computation required to check the satisfiability of the store.

4 Experiments with the A-System

A number of different experiments have been carried out to test the underlying general computational behaviour of (an initial implementation of) the $\mathcal{A}$-System and how this can be affected by extending the high-level representation of problems with additional information.

The current version of the system is implemented as a meta-program on top of Sicstus Prolog 3.8.5 and uses its finite domain constraint solver. It is developed as an experimentation tool to test different datastructures and strategies. Currently a form of indexing on the abducibles and constraints is used in the datastructures. With respect to the parameters of the abductive search presented in the previous section the current version implements only a very basic form of heuristics. In particular, the methods used for checking the satisfiability of the constraint store during its construction are simple.

A first set of experiments consists of problems whose specification can be reduced to a finite domain constraint store in one phase of reduction. Such problems include standard constraint satisfaction problems, e.g. N-Queens, Graph Coloring and Scheduling problems. The aim here was to confirm that the execution will be deterministic and to compare the execution time with that of solving the problem directly in CLP to see the overhead cost of the reduction of the high-level rep-

resentation of the problem. The tables below show a sample of these experiments for the N-Queens problem and a graph coloring problem. For the graph coloring experiment a set of 4-colorable planar graphs were constructed. The run times are split into the time needed to find an abductive solution and the subsequent time needed to ground this by the CLP labeling. All experiments reported in this paper are done on a Linux machine (800 Mhz) with 512 MB of memory.

N-Queens

size	abductive solution (s)	clp grounding
30	0.7	10ms
40	1.9	20ms
50	4.1	160ms
60	8.9	160ms

Graph Coloring

size(nodes)	abductive solution (s)	clp grounding
80	1.9	10ms
100	3.7	20ms
200	21.3	20ms
400	233.3	50ms

Although the reduction of the whole specification is done in one step (no backtracking occurs over the abductive solution) in both experiments the construction time (abductive solution) increases clearly with the size. So the metainterpreter is some orders slower than the Prolog below, which is used in a CLP-program to set up the constraints.

Another set of experiments was carried out on planning problems. These are non-deterministic problems where the abductive computation and search of the $\mathcal{A}$-System can be tested. We selected from the latest AIPS2000 planning competition two domains: blocks world and logistics. The next table shows the performance of the system on the blocks world domain. For this experiment we used a specification with one action *move(X,Y,T)*, which denotes moving a block X onto Y at time T.

Problem	Planlength	Time(s)
20-0	37	1.7
30-1	57	3.5
45-0	88	10.2
55-1	105	17.2
60-0	114	30.8

The second considered domain is that of logistics. For this domain a specification with functors and the general event calculus is used. Futhermore the specification is two layered: the top layer is a compact high level description of the logistics domain. The lower level actions (from the original AIPS strips description) are derived from the high level ones. The first table below shows the performance of the $\mathcal{A}$-System on problems where the system has been used to compute only high-level abstract plans. The length of the plans are given by the final time when the goal is achieved (first number in the column) and the number of abduced actions describing the actual plan. A blank entry indicates that the system was unable to find a solution within a given time limit.

The results for the low level plans are presented in a second table. These plans are computed in a separated phase using as input the previous computed high level plans. This step expands the high level plans in number of time points and actions.

High level solutions

Problem	Length	Time(s)	Max time points
7-0	8/15	6.3	100
8-0	9/19	7.8	100
9-0	8/16	34.1 (*)	100
10-0	7/18	8.5	100
11-0	-	-	-
12-0	9/18	298.4 (*)	100
12-0	9/18	19.9	20

Low level solutions

Problem	Length	Time
7-0	21/40	40ms
8-0	21/47	50ms
9-0	15/39	40ms
10-0	18/46	50ms
12-0	24/45	50ms

Some of the problems took a while to return a solution. Because the finite domain solver is incomplete, the $\mathcal{A}$-System may go on with a branch although the constraint store is already unsatisfiable (see the marked entries in the above tables). We tried different satisfiability checks but none of them was overall successful. A parameter in such a check is the domain size of the variables (see the last column of the table of the high level plans): by decreasing the maximal number of time points the satisfiability check fails faster when the constraint store is insatisfiable. As consequence some problems could be solved in a reasonable time. Currently, we are investigating how we can improve this satisfiablity check.

Compared to previous experiments with SLDNFAC and ACLP, the $\mathcal{A}$-System is more reliable and more robust to changes in the order of execution of the query. However because the constraint solver is incomplete it still possible to have one derivation of $\mathcal{A}$-System failing to find a solution and another one finding it immediately. Another main difference of the $\mathcal{A}$-System with these previous systems is its separation of search and inference. This allows easy experimentation with several strategies and different degrees of deterministic propagation.

5 Related Work and Conclusions

There are several other abductive systems for ALP that have recently been developed. Different methods have been used e.g. bottom up computation [Iwayama and Satoh, 2000], tabling [Alferes *et al.*, 1999] and rewriting rules with the completion [Fung and Kowalski, 1997; Kowalski *et al.*, 1998]. The $\mathcal{A}$-System with its two earlier ALP systems, the ACLP system and SLDNFAC, is to our knowledge the first abductive system that has paid particular attention to the computational aspects of abduction using constraint solving extensively to control the abductive search and enhance its computational behaviour.

On a more abstract level our work is related to Answer Set Programming(ASP) [Gelfond and Lifschitz, 1991]. Strong connections have been established [Satoh and Iwayama, 1991] between ALP and ASP. At the level of semantics these two frameworks are for a large class of theories (ASP admits only a special type of integrity constraints) equivalent.

At the level of computation our construction of an abductive solution in the $\mathcal{A}$-System corresponds to the generation of a model in ASP but there are some significant differences in the respective computational models. It is therefore instructive to develop systematic experiments to compare systems from these two approaches to declarative problem solving.

The abductive search in the $\mathcal{A}$-System can be parametrized in a variety of ways to reflect the type of cooperation with constraint solver and the heuristics that are used. An important future development of the system is to structure further this parametric space and to study how more techniques from constraint programming and advanced heuristic search can be incorporated in order to improve the general computational behaviour of the framework on a variety of problems. In particular, we can study how recent heuristic methods for planning [Bonet and Geffner, 1999] can be generalized to the abductive computation of the $\mathcal{A}$-System.

This general improvement of the underlying computational efficiency of the ALP framework, or indeed of any other declarative problem solving framework, is clearly limited as we are aiming to use the framework on a general variety of problems. A complementary line of development of the $\mathcal{A}$-System and its associated ALP framework concerns the study of how to provide a programming environment where the user has the facility to incrementally refine her/his high-level representation of the problem in a modular way. A general methodology for declarative problem solving with abduction is emerging where the ALP modeling environment provides the possibility for extra problem specific (declarative or control) knowledge to be included that would enhance its computational performance.

References

[Alferes *et al.*, 1999] J. J. Alferes, L. M. Pereira, and T. Swift. Well-founded abduction via tabled dual programs. In *Proceedings of ICLP-99*, 1999.

[Bonet and Geffner, 1999] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proc. European Conference on Planning (ECP-99)*, 1999.

[Ciampolini *et al.*, 2000] A. Ciampolini, E. Lamma, P. Mello, and P. Torroni. An implementation for abductive logic agents. In *AI*IA'99, LNAI Vol. 1792*, 2000.

[Console *et al.*, 1996] L. Console, L. Portinale, and D. Theseider Dupré. Using compiled knowledge to guide and focus abductive diagnosis. *IEEE Transactions on Knowledge and Data Engineering*, 8 (5):690–706, 1996.

[Denecker and Kakas, 2000] M. Denecker and A.C. Kakas. *Abductive Logic Programming*. Special issue of JLP, Vol 44 (1-3), 2000.

[Denecker and Schreye, 1998] M. Denecker and D. De Schreye. Sldnfa: an abductive procedure for abductive logic programs. *Logic Programming*, 34(2):111–167, 1998.

[Denecker, 2000] M. Denecker. Extending classical logic with inductive definitions. In *Proceedings of CL-2000, 703-717*, 2000.

[Fung and Kowalski, 1997] T.H. Fung and R.A. Kowalski. The iff procedure for abductive logic programming. *Logic Programming*, 33(2):151–165, 1997.

[Gelder *et al.*, 1991] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[Gelfond and Lifschitz, 1991] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing, pp. 365-387*, 1991.

[Inoue and Sakama, 1995] K. Inoue and C. Sakama. Abductive framework for nonmonotonic theory change. In *Proceedings of IJCAI-95*, pages 204–210, 1995.

[Iwayama and Satoh, 2000] N. Iwayama and K. Satoh. Computing abduction by using tms with top-down expectation. *Logic Programming*, 44(1-3):179–206, 2000.

[Kakas *et al.*, 1998] A.C. Kakas, R.A. Kowalski, and F. Toni. The role of abduction in logic programming. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in AI and Logic Programming*, volume 5, pages 235–324. Oxford University Press, 1998.

[Kakas *et al.*, 2000] A.C. Kakas, A. Michael, and C. Mourlas. Aclp: Abductive constraint logic programming. *Journal of Logic Programming: Special Issue on Abductive Logic Programming*, 44(1-3):129–177, 2000.

[Kowalski and Sadri, 1999] R.A. Kowalski and F. Sadri. From logic programming to multi-agent systems. *Annals of Mathemathics and Artificial Intelligence*, 1999.

[Kowalski *et al.*, 1998] R.A. Kowalski, F. Toni, and G. Wetzel. Executing suspended logic programs. *Fundamenta Informaticae*, 34(3):203–224, 1998.

[Missiaen *et al.*, 1995] L.R. Missiaen, M. Denecker, and M. Bruynooghe. CHICA, an abductive planning system based on event calculus. *Journal of Logic and Computation*, 5(5):579–602, 1995.

[Mooney, 2000] R.J. Mooney. Integrating abduction and induction in machine learning. In *Abduction and Induction: essays on their relation and integration*, pages 181–191. Kluwer Academic Press, 2000.

[Muggleton, 2000] S. Muggleton. Theory completion in learning. In *Inductive Logic Programming, ILP-00*, 2000.

[Pagnucco, 1996] M. Pagnucco. *The role of abductive reasoning within the process of belief revision*. PhD Thesis, Dept. of Computer Science, University of Sydney, 1996.

[Poole *et al.*, 1987] D. Poole, R. Goebel, and R. Aleliunas. Theorist: A logical reasoning system for defaults and diagnosis. In *The Knowledge Frontier: Essays in the Representation of Knowledge*, pages 331–352. Springer-Verlag, 1987.

[Satoh and Iwayama, 1991] K. Satoh and N. Iwayama. Computing abduction by using the tms. In *Proc. of ICLP'91*, pages 505–518, 1991.

[Shanahan, 2000] M. Shanahan. An abductive event calculus planner. *Logic Programming*, 44(1-3):207–239, 2000.

A Comparative Study of Logic Programs with Preference

Torsten Schaub* **and Kewen Wang**
Institut für Informatik, Universität Potsdam
Postfach 60 15 53, D–14415 Potsdam, Germany
{torsten,kewen}@cs.uni-potsdam.de

Abstract

We are interested in semantical underpinnings for existing approaches to preference handling in extended logic programming (within the framework of answer set programming). As a starting point, we explore three different approaches that have been recently proposed in the literature. Because these approaches use rather different formal means, we furnish a series of uniform characterizations that allow us to gain insights into the relationships among these approaches. To be more precise, we provide different characterizations in terms of (i) fixpoints, (ii) order preservation, and (iii) translations into standard logic programs. While the two former provide semantics for logic programming with preference information, the latter furnishes implementation techniques for these approaches.

1 Introduction

Numerous approaches to logic programming with preference information have been proposed in the literature. So far, however, there is no systematic account on their structural differences, finally leading to solid semantical underpinnings. We address this shortcoming by a comparative study of a distinguished class of approaches to preference handling. This class consists of *selective* approaches remaining within the complexity class of extended logic programming (under answer sets semantics). These approaches are selective insofar as they use preferences to distinguish certain "models" of the original program.

We explore three different approaches that have been recently proposed in the literature, namely the ones in [Brewka and Eiter, 1999; Delgrande *et al.*, 2000; Wang *et al.*, 2000]. Our investigation adopts characterization techniques found in the same literature in order to shed light on the relationships among these approaches. This provides us with different characterizations in terms of (i) fixpoints, (ii) order preservation, and (iii) translations into standard logic programs. While the two former provide semantics for logic programming with preference information, the latter furnishes implementation techniques for these approaches. From another perspective, one can view (iii) as an axiomatization of the underlying strategy within the object language, while (i) may be regarded as a meta-level description of the corresponding construction process. One may view (ii) as the most semantical characterization because it tells us which "models" of the original program are selected by the respective preference handling strategy.

We limit (also in view of (iii)) our investigation to approaches to preference handling that remain within NP. This excludes approach like the ones in [Rintanen, 1995; Zhang and Foo, 1997] that step outside the complexity class of the underlying reasoning method. This applies also to the approach in [Sakama and Inoue, 1996], where preferences on literals are investigated. While the approach of [Gelfond and Son, 1997] remains within NP, it advocates strategies that are non-selective. Approaches that can be addressed within this framework include [Baader and Hollunder, 1993; Brewka, 1994] that were originally proposed for default logic.

2 Definitions and notation

We assume a basic familiarity with logic programming under *answer set semantics* [Gelfond and Lifschitz, 1991]. An *extended logic program* is a finite set of rules of the form

$$L_0 \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n, \qquad (1)$$

where $n \geq m \geq 0$, and each L_i $(0 \leq i \leq n)$ is a *literal*, ie. either an atom A or its negation $\neg A$. The set of all literals is denoted by Lit. Given a rule r as in (1), we let $head(r)$ denote the *head*, L_0, of r and $body(r)$ the *body*, $\{L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n\}$, of r. Further, let $body^+(r) = \{L_1, \ldots, L_m\}$ and $body^-(r) = \{L_{m+1}, \ldots, L_n\}$. A program is called *basic* if $body^-(r) = \emptyset$ for all its rules.

We define the reduct of a rule r as $r^+ = head(r) \leftarrow body^+(r)$. The *reduct*, Π^X, of a program Π *relative to* a set X of literals is defined by

$$\Pi^X = \{r^+ \mid r \in \Pi \text{ and } body^-(r) \cap X = \emptyset\}.$$

A set of literals X is *closed under* a basic program Π iff for any $r \in \Pi$, $head(r) \in X$ whenever $body^+(r) \subseteq X$. We say that X is *logically closed* iff it is either consistent (ie. it does not contain both a literal A and its negation $\neg A$) or equals Lit. The smallest set of literals which is both logically closed and closed under a basic program Π is denoted by $Cn(\Pi)$.

* Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada.

Finally, a set X of literals is an *answer set* of a program Π iff $Cn(\Pi^X) = X$. In what follows, we deal with *consistent* answer sets only.

The set Γ_Π^X of all *generating rules* of an answer set X from Π is given by

$$\Gamma_\Pi^X = \{r \in \Pi \mid body^+(r) \subseteq X \text{ and } body^-(r) \cap X = \emptyset\}.$$

As van Gelder in [1993], we define $C_\Pi(X) = Cn(\Pi^X)$. Note that the operator C_Π is anti-monotonic, which implies that the operator $A_\Pi(X) = C_\Pi(C_\Pi(X))$ is monotonic. A fixpoint of A_Π is called an *alternating fixpoint* for Π. Different semantics are captured by distinguishing different groups of fixpoints of A_Π.

A (*statically*) *ordered logic program*[1] is a pair $(\Pi, <)$, where Π is an extended logic program and $< \subseteq \Pi \times \Pi$ is an irreflexive and transitive relation. Given, $r_1, r_2 \in \Pi$, the relation $r_1 < r_2$ expresses that r_2 has higher priority than r_1.[2]

3 Preferred alternating fixpoints

The notion of answer sets (without preference) is based on a reduction of extended logic programs to basic programs (without default negation). Such a reduction is inapplicable when addressing conflicts by means of preference information since all conflicts between rules are simultaneously resolved when turning Π into Π^X. Rather conflict resolution must be characterized among the original rules in order to account for blockage between rules. That is, once the negative body $body^-(r)$ is eliminated there is no way to detect whether $head(r') \in body^-(r)$ holds in case of $r < r'$.

Such an approach is pursued in [Wang *et al.*, 2000] for characterizing "preferred" answer sets. Following earlier approaches based on default logic [Baader and Hollunder, 1993; Brewka, 1994], this approach is based on the concept of *activeness*: Let $X, Y \subseteq Lit$ be two sets of literals in an ordered logic program $(\Pi, <)$. A rule r in Π is *active* wrt the pair (X, Y), if $body^+(r) \subseteq X$ and $body^-(r) \cap Y = \emptyset$.

Definition 1 (Wang *et al.*,2000) *Let $(\Pi, <)$ be an ordered logic program and let X be a set of literals. We define*

$$\begin{array}{rcl} X_0 & = & \emptyset \quad \text{and for } i \geq 0 \\ X_{i+1} & = & X_i \cup \{head(r) \mid \\ & & \left.\begin{array}{ll} I. & r \in \Pi \text{ is active wrt } (X_i, X) \text{ and} \\ II. & \text{there is no rule } r' \in \Pi \text{ with } r < r' \\ & \text{such that} \\ & (a)\ r' \text{ is active wrt } (X, X_i) \text{ and} \\ & (b)\ head(r') \notin X_i \end{array}\right\} \end{array}$$

Then, $\mathcal{C}_{(\Pi,<)}(X) = \bigcup_{i\geq 0} X_i$ if $\bigcup_{i\geq 0} X_i$ is consistent.
Otherwise, $\mathcal{C}_{(\Pi,<)}(X) = Lit$.

The idea is to apply a rule r only if the question of application has been settled for all higher-ranked rules r'. That is, if either its prerequisites will never be derivable, viz. $body^+(r') \not\subseteq X$, or r' is defeated by what has been derived so far, viz. $body^-(r) \cap X_i \neq \emptyset$, or r' or another rule with the same head have already applied, viz. $head(r') \in X_i$.

As its original C_Π, the operator $\mathcal{C}_{(\Pi,<)}$ is anti-monotonic. Accordingly, we may define for any set $X \subseteq Lit$, the *alternating transformation* of $(\Pi, <)$ as $\mathcal{A}_{(\Pi,<)}(X) = \mathcal{C}_{(\Pi,<)}(\mathcal{C}_{(\Pi,<)}(X))$. A fixpoint of $\mathcal{A}_{(\Pi,<)}$ is called an *alternating fixpoint* of $(\Pi, <)$. Note that $\mathcal{A}_{(\Pi,<)}$ is monotonic.

Now, in analogy to Van Gelder [1993], a semantical framework for ordered logic programs in terms of sets of alternating fixpoints can be defined. Three different types of semantics are investigated in [Wang *et al.*, 2000]: (i) Preferred[3] answer sets, viz. alternating fixpoints being also fixpoints of $\mathcal{C}_{(\Pi,<)}$. (ii) Preferred regular extensions, viz. maximal normal[4] alternating fixpoints of $(\Pi, <)$. (iii) Preferred well-founded model, viz. the least alternating fixpoint of $(\Pi, <)$.

We put the prefix 'W-' whenever a distinction to other approaches is necessary.

For illustration, consider the following ordered logic program $(\Pi_2, <)$ due to [Baader and Hollunder, 1993]:

$$\begin{array}{rrcl} r_1: & \neg f & \leftarrow & p, not\ f \\ r_2: & w & \leftarrow & b, not\ \neg w \\ r_3: & f & \leftarrow & w, not\ \neg f \\ r_4: & b & \leftarrow & p \\ r_5: & p & \leftarrow & \end{array} \qquad r_2 < r_1 \qquad (2)$$

Observe that Π_2 admits two answer sets: $X = \{p, b, \neg f, w\}$ and $X' = \{p, b, f, w\}$. As argued in [Baader and Hollunder, 1993], X is the unique W-preferred answer set. To see this, observe that

$$\begin{array}{rclcrcl} X_0 & = & \emptyset & & X'_0 & = & \emptyset \\ X_1 & = & \{p\} & & X'_1 & = & \{p\} \\ X_2 & = & \{p, b, \neg f\} & & X'_2 & = & \{p, b\} \\ X_3 & = & \{p, b, \neg f, w\} & & X'_3 & = & X'_2 \neq X' \\ X_4 & = & X_3 = X & & & & \end{array}$$

Note that w cannot be included into X'_3 because r_1 is active wrt (X', X'_2) and r_1 is preferred to r_2.

4 Compiling order preservation

A translation of ordered logic programs $(\Pi, <)$ to standard ones Π' is developed in [Delgrande *et al.*, 2000]. The specific strategy used there ensures that the resulting program Π' admits only those answer sets of the original program Π that are *order preserving*:

Definition 2 *Let $(\Pi, <)$ be a statically ordered program and let X be an answer set of Π.*

Then, X is called $<$-preserving, if there exists an enumeration $\langle r_i \rangle_{i \in I}$ of Γ_Π^X such that for every $i, j \in I$ we have that:

0. *$body^+(r_i) \subseteq \{head(r_j) \mid j < i\}$; and*
1. *if $r_i < r_j$, then $j < i$; and*
2. *if $r_i < r'$ and $r' \in \Pi \setminus \Gamma_\Pi^X$, then*
   - *(a) $body^+(r') \not\subseteq X$ or*
   - *(b) $body^-(r') \cap \{head(r_j) \mid j < i\} \neq \emptyset$.*

[1] Also called *prioritized* logic program by some authors, as eg. in [Brewka and Eiter, 1999].

[2] Some authors, among them [Brewka and Eiter, 1999], attribute relation $<$ the inverse meaning.

[3] Originally called *prioritized*.

[4] An alternating fixpoint X is normal if $X \subseteq \mathcal{C}_{(\Pi,<)}(X)$.

Condition 0 makes the property of *groundedness*[5] explicit. Although any standard answer set is generated by a grounded sequence of rules, we will see in the sequel that this property is weakened when preferences are at issue. Condition 1 stipulates that $\langle r_i \rangle_{i \in I}$ is *compatible* with $<$, a property invariant to all of the considered approaches. Lastly, Condition 2 is comparable with Condition II in Definition 1; it guarantees that rules can never be blocked by lower-ranked rules.

As above, $X = \{p, b, \neg f, w\}$ is the only $<$-preserving answer set of Π_2; it can be generated by the grounded sequences $\langle r_5, r_4, r_1, r_2 \rangle$ and $\langle r_5, r_1, r_4, r_2 \rangle$ both of which satisfy conditions 1 and 2. The only grounded sequence generating $X' = \{p, b, f, w\}$, namely $\langle r_5, r_4, r_2, r_3 \rangle$, violates 2b.

The corresponding translation integrates ordering information into the logic program via a special-purpose predicate symbol $\prec$. This allows also for treating ordering information in a dynamic fashion. A logic program over a propositional language $\mathcal{L}$ is said to be *dynamically* ordered iff $\mathcal{L}$ contains the following pairwise disjoint categories: (i) a set N of terms serving as *names* for rules; (ii) a set At of (propositional) atoms of a program; and (iii) a set $At_{\prec}$ of *preference atoms* $s \prec t$, where $s, t \in N$ are names. For each such program Π, we assume furthermore a bijective function $n(\cdot)$ assigning to each rule $r \in \Pi$ a name $n(r) \in N$. To simplify notation, we usually write n_r instead of $n(r)$ (and we sometimes abbreviate n_{r_i} by n_i).

An atom $n_r \prec n_{r'} \in At_{\prec}$ amounts to asserting that $r < r'$ holds. A statically ordered program $(\Pi, <)$ can thus be captured by programs containing preference atoms only among their facts; it is then expressed by the program $\Pi \cup \{(n_r \prec n_{r'}) \leftarrow \ \mid r < r'\}$.

Given $r < r'$, one wants to ensure that r' is considered before r, in the sense that, for a given answer set X, rule r' is known to be applied or defeated *ahead of* r (cf. Condition II or 2 above, respectively). This is done by translating rules so that the order of rule application can be explicitly controlled. For this purpose, one needs to be able to detect when a rule has been applied or when a rule is defeated. For a rule r, there are two cases for it not to be applied: it may be that some literal in $body^+(r)$ does not appear in the answer set, or it may be that a literal in $body^-(r)$ is in the answer set. For detecting non-applicability (i.e., blockage), for each rule r in the given program Π, a new, special-purpose atom $\mathsf{bl}(n_r)$ is introduced. Similarly, a special-purpose atom $\mathsf{ap}(n_r)$ is introduced to detect the case where a rule has been applied. For controlling application of rule r the atom $\mathsf{ok}(n_r)$ is introduced. Informally, one concludes that it is ok to apply a rule just if it is ok with respect to every $<$-greater rule; for such a $<$-greater rule r', this will be the case just when r' is known to be blocked or applied.

More formally, given a dynamically ordered program Π over $\mathcal{L}$, let $\mathcal{L}^+$ be the language obtained from $\mathcal{L}$ by adding, for each $r, r' \in \Pi$, new pairwise distinct propositional atoms $\mathsf{ap}(n_r)$, $\mathsf{bl}(n_r)$, $\mathsf{ok}(n_r)$, and $\mathsf{ok}'(n_r, n_{r'})$. Then, the translation $\mathcal{T}$ maps an ordered program Π over $\mathcal{L}$ into a standard program $\mathcal{T}(\Pi)$ over $\mathcal{L}^+$ in the following way.

[5] This term is borrowed from the literature on default logic.

Definition 3 (Delgrande *et al.*,2000) *Let $\Pi = \{r_1, \dots, r_k\}$ be a dynamically ordered logic program over $\mathcal{L}$.*

Then, the logic program $\mathcal{T}(\Pi)$ over $\mathcal{L}^+$ is defined as $\mathcal{T}(\Pi) = \bigcup_{r \in \Pi} \tau(r)$, where $\tau(r)$ consists of the following rules, for $L^+ \in body^+(r)$, $L^- \in body^-(r)$, and $r', r'' \in \Pi$:

$$
\begin{array}{rrl}
a_1(r): & head(r) & \leftarrow \mathsf{ap}(n_r) \\
a_2(r): & \mathsf{ap}(n_r) & \leftarrow \mathsf{ok}(n_r), body(r) \\
b_1(r, L^+): & \mathsf{bl}(n_r) & \leftarrow \mathsf{ok}(n_r), not\ L^+ \\
b_2(r, L^-): & \mathsf{bl}(n_r) & \leftarrow \mathsf{ok}(n_r), L^- \\
 & & \\
c_1(r): & \mathsf{ok}(n_r) & \leftarrow \mathsf{ok}'(n_r, n_{r_1}), \dots, \mathsf{ok}'(n_r, n_{r_k}) \\
c_2(r, r'): & \mathsf{ok}'(n_r, n_{r'}) & \leftarrow not\ (n_r \prec n_{r'}) \\
c_3(r, r'): & \mathsf{ok}'(n_r, n_{r'}) & \leftarrow (n_r \prec n_{r'}), \mathsf{ap}(n_{r'}) \\
c_4(r, r'): & \mathsf{ok}'(n_r, n_{r'}) & \leftarrow (n_r \prec n_{r'}), \mathsf{bl}(n_{r'}) \\
 & & \\
t(r, r', r''): & n_r \prec n_{r''} & \leftarrow n_r \prec n_{r'}, n_{r'} \prec n_{r''} \\
as(r, r'): & \neg(n_{r'} \prec n_r) & \leftarrow n_r \prec n_{r'}
\end{array}
$$

We write $\mathcal{T}(\Pi, <)$ rather than $\mathcal{T}(\Pi')$, whenever Π' is the dynamically ordered program capturing $(\Pi, <)$.

The first four rules of $\tau(r)$ express applicability and blocking conditions of the original rules. The second group of rules encodes the strategy for handling preferences. The first of these rules, $c_1(r)$, "quantifies" over the rules in Π. This is necessary when dealing with dynamic preferences since preferences may vary depending on the corresponding answer set. The three rules $c_2(r, r')$, $c_3(r, r')$, and $c_4(r, r')$ specify the pairwise dependency of rules in view of the given preference ordering: For any pair of rules r, r' with $n_r \prec n_{r'}$, we derive $\mathsf{ok}'(n_r, n_{r'})$ whenever $n_r \prec n_{r'}$ fails to hold, or whenever either $\mathsf{ap}(n_{r'})$ or $\mathsf{bl}(n_{r'})$ is true. This allows us to derive $\mathsf{ok}(n_r)$, indicating that r may potentially be applied whenever we have for all r' with $n_r \prec n_{r'}$ that r' has been applied or cannot be applied. It is important to note that this is only one of many strategies for dealing with preferences: different strategies are obtainable by changing the specification of $\mathsf{ok}(\cdot)$ and $\mathsf{ok}'(\cdot, \cdot)$, as we will see below.

As shown in [Delgrande *et al.*, 2000], a set of literals X is a $<$-preserving answer set of Π iff $X = Y \cap \mathcal{L}$ for some answer set Y of $\mathcal{T}(\Pi, <)$. In the sequel, we refer to such answer sets as being D-*preferred*.

5 Synthesis

The last two sections have exposed three rather different ways of characterizing preferred answer sets. Despite their different characterizations, however, it turns out that the two approaches prefer similar answer sets.

5.1 Characterizing D-preference

We start by providing a fixpoint definition for D-*preference*. For this purpose, we assume a bijective mapping $rule(\cdot)$ from rule heads to rules, that is, $rule(head(r)) = r$; accordingly, $rule(\{head(r) \mid r \in R\}) = R$. Such mappings can be defined in a bijective way by distinguishing different occurrences of literals.

Definition 4 *Let $(\Pi, <)$ be a statically ordered logic pro-*

gram and let X be a set of literals. We define

$$
\begin{array}{rcl}
X_0 & = & \emptyset \quad \textit{and for } i \geq 0 \\
X_{i+1} & = & X_i \cup \{head(r) \mid \\
& & \left.\begin{array}{ll} I. & r \in \Pi \textit{ is active wrt } (X_i, X) \textit{ and} \\ II. & \textit{there is no rule } r' \in \Pi \textit{ with } r < r' \\ & \textit{such that} \\ & (a)\ r' \textit{ is active wrt } (X, X_i) \textit{ and} \\ & (b)\ r' \notin rule(X_i) \end{array}\right\}
\end{array}
$$

Then, $\mathcal{C}^{\mathrm{D}}_{(\Pi,<)}(X) = \bigcup_{i\geq 0} X_i$ if $\bigcup_{i\geq 0} X_i$ is consistent.
Otherwise, $\mathcal{C}^{\mathrm{D}}_{(\Pi,<)}(X) = Lit$.

The difference between this definition and Definition 1 manifests itself in IIb. While D-preference requires that a higher-ranked rule has effectively applied, W-preference contents itself with the presence of the head of the rule, no matter whether this was supplied by the rule itself.

This difference is nicely illustrated by program $(\Pi_3, <)$:

$$
\begin{array}{llll}
r_1: & a & \leftarrow & not\ b \qquad\qquad r_2 < r_1 \\
r_2: & b & \leftarrow & \\
r_3: & a & \leftarrow &
\end{array} \tag{3}
$$

While the only answer set $\{a, b\}$ is W-preferred set, there is no D-preferred answer set. This is the same with program $(\Pi'_3, <)$ obtained by replacing r_1 with $r'_1 : a \leftarrow b$.

We have the following result providing three alternative characterizations of D-preferred answer sets.

Theorem 1 *Let $(\Pi, <)$ be a statically ordered logic program over $\mathcal{L}$ and let X be a consistent set of literals.*
Then, the following propositions are equivalent.

1. $\mathcal{C}^{\mathrm{D}}_{(\Pi,<)}(X) = X$;
2. $X = Y \cap \mathcal{L}$ *for some answer set Y of $\mathcal{T}(\Pi, <)$;*
3. X *is a $<$-preserving answer set of* Π.

While the last result dealt with effective answer sets, the next one shows that applying operator $\mathcal{C}^{\mathrm{D}}_{(\Pi,<)}$ is equivalent to the application of van Gelder's operator $C_{\Pi'}$ to the translated program $\mathcal{T}(\Pi, <)$.

Theorem 2 *Let $(\Pi, <)$ be a statically ordered logic program over $\mathcal{L}$ and let X be a consistent set of literals over $\mathcal{L}$.*
Then, we have that $\mathcal{C}^{\mathrm{D}}_{(\Pi,<)}(X) = C_{\mathcal{T}(\Pi,<)}(Y) \cap \mathcal{L}$ for some set of literals Y over $\mathcal{L}^+$ such that $X = Y \cap \mathcal{L}$.

This result is important because it allows us to use the translation $\mathcal{T}(\Pi, <)$ for implementing further semantics by appeal to the alternating fixpoint idea.

5.2 Characterizing W-preference

We start by showing how W-preference can be characterized in terms of order preservation.

Definition 5 *Let $(\Pi, <)$ be a statically ordered program and let X be an answer set of Π.*
Then, X is called $<^{\mathrm{W}}$-preserving, if there exists an enumeration $\langle r_i \rangle_{i \in I}$ of Γ^X_Π such that for every $i, j \in I$ we have that:

0. (a) $body^+(r_i) \subseteq \{head(r_j) \mid j < i\}$ *or*
   (b) $head(r_i) \in \{head(r_j) \mid j < i\}$; *and*
1. *if $r_i < r_j$, then $j < i$; and*
2. *if $r_i < r'$ and $r' \in \Pi \setminus \Gamma^X_\Pi$, then*
   (a) $body^+(r') \not\subseteq X$ *or*
   (b) $body^-(r') \cap \{head(r_j) \mid j < i\} \neq \emptyset$ *or*
   (c) $head(r') \in \{head(r_j) \mid j < i\}$.

The primary difference of this concept of order preservation to the original one is clearly the weaker notion of groundedness. This involves the rules in Γ^X_Π (via Condition 0b) as well as those in $\Pi \setminus \Gamma^X_\Pi$ (via Condition 2c). The rest of the definition is the same as in Definition 2. For instance, answer set $\{a, b\}$ of Π_3 is generated by the $<^{\mathrm{W}}$-preserving rule sequence $\langle r_3, r_2 \rangle$. Note that r_1 satisfies 2c but neither 2a nor 2b. For a complement, in $(\Pi'_3, <)$, r'_1 is dealt with via Condition 0b.

Interestingly, this weaker notion of groundedness can be easily integrated into the translation given in the last section.

Definition 6 *Given the same prerequisites as in Definition 3.*
Then, the logic program $\mathcal{T}^{\mathrm{W}}(\Pi)$ over $\mathcal{L}^+$ is defined as $\mathcal{T}^{\mathrm{W}}(\Pi) = \bigcup_{r\in\Pi} \tau(r) \cup \{c_5(r, r') \mid r, r' \in \Pi\}$, where

$$c_5(r, r') :\ \mathsf{ok}'(n_r, n_{r'}) \leftarrow (n_r \prec n_{r'}), head(r')$$

The purpose of $c_5(r, r')$ is to eliminate rules from the preference handling process once their head has been derived.

We have the following result, showing in particular, how W-preference is implementable via off-the-shelf logic programming systems.

Theorem 3 *Let $(\Pi, <)$ be a statically ordered logic program over $\mathcal{L}$ and let X be a consistent set of literals. Then, the following propositions are equivalent.*

1. $\mathcal{C}_{(\Pi,<)}(X) = X$;
2. $X = Y \cap \mathcal{L}$ *for some answer set Y of $\mathcal{T}^{\mathrm{W}}(\Pi, <)$;*
3. X *is a $<^{\mathrm{W}}$-preserving answer set of* Π.

In analogy to what we have shown above, we have the following stronger result, opening the avenue for implementing more semantics based on W-preference:.

Theorem 4 *Let $(\Pi, <)$ be a statically ordered logic program over $\mathcal{L}$ and let X be a consistent set of literals over $\mathcal{L}$.*
Then, we have that $\mathcal{C}_{(\Pi,<)}(X) = C_{\mathcal{T}^{\mathrm{W}}(\Pi,<)}(Y) \cap \mathcal{L}$ for some set of literals Y over $\mathcal{L}^+$ such that $X = Y \cap \mathcal{L}$.

6 Brewka and Eiter's concept of preference

Another approach to preference was proposed by Brewka and Eiter in [1999]. For brevity, we omit technical details and simply say that an answer set is B-*preferred*; the reader is referred to [Brewka and Eiter, 1999; 2000] for details.

This approach differs in two significant ways from the two approaches given above. First, the construction of answer sets is separated from verifying whether they respect the given preferences. Interestingly, this verification is done on the basis of the prerequisite-free program obtained from the original one by "evaluating" $body^+(r)$ for each rule r wrt the separately constructed (standard) answer set. Second, rules that putatively lead to counter-intuitive results are removed from

the inference process. This is made explicit in [Brewka and Eiter, 2000], where the following filtering transformation is defined:[6]

$$Z_X(\Pi) = \Pi \setminus \{r \in \Pi \mid head(r) \in X, body^-(r) \cap X \neq \emptyset\} \quad (4)$$

Then, by definition, an answer set of Π is B-preferred iff it is a B-preferred answer set of $Z_X(\Pi)$.

The distinguishing example of this approach is given by program $(\Pi_5, <)$:

$$\begin{array}{llll} r_1: & b & \leftarrow & a, not\ \neg b \\ r_2: & \neg b & \leftarrow & not\ b \\ r_3: & a & \leftarrow & not\ \neg a \end{array} \qquad r_3 < r_2 < r_1 \quad (5)$$

Program Π_5 has two standard answer sets, $\{a, b\}$ and $\{a, \neg b\}$. While the former is B-preferred, neither of them is W- or D-preferred (see below). Also, we note that both answer sets of program $(\Pi_2, <)$ are B-preferred, while only $\{p, b, \neg f, w\}$ is W- and D-preferred.

In order to shed some light on these differences, we start by providing a fixpoint characterization of B-preference:

Definition 7 *Let $(\Pi, <)$ be an ordered logic program and let X be a set of literals. We define*

$$\begin{array}{rcl} X_0 & = & \emptyset \quad \textit{and for } i \geq 0 \\ X_{i+1} & = & X_i \cup \left\{ head(r) \;\middle|\; \begin{array}{ll} I. & r \in \Pi \textit{ is active wrt } (X, X) \textit{ and} \\ II. & \textit{there is no rule } r' \in \Pi \textit{ with } r < r' \\ & \textit{such that} \\ & (a)\ r' \textit{ is active wrt } (X, X_i) \textit{ and} \\ & (b)\ head(r') \notin X_i \end{array} \right\} \end{array}$$

Then, $\mathcal{C}^{\text{B}}_{(\Pi,<)}(X) = \bigcup_{i \geq 0} X_i$ if $\bigcup_{i \geq 0} X_i$ is consistent.

Otherwise, $\mathcal{C}_{(\Pi,<)}(X) = Lit$.

The difference between this definition[7] and its predecessors manifests itself in Condition I, where activeness is tested wrt (X, X) instead of (X_i, X) as in Definition 1 and 4. In fact, in Example (5) it is the (unprovability of the) prerequisite a of the highest-ranked rule r_1 that makes the construction of W- or D-preferred answer sets break down (cf. Definition 1 and 4). This is avoided with B-preference because once answer set $\{a, b\}$ is provided, its preference-compatibility is tested wrt the program obtained by replacing r_1 with $b \leftarrow not\ \neg b$.

B-preference can be captured by means of the following notion of order preservation:

Definition 8 *Let $(\Pi, <)$ be a statically ordered program and let X be an answer set of Π.*

Then, X is called $<^{\text{B}}$-preserving, if there exists an enumeration $\langle r_i \rangle_{i \in I}$ of Γ^X_Π such that, for every $i, j \in I$, we have that:

1. *if $r_i < r_j$, then $j < i$; and*
2. *if $r_i < r'$ and $r' \in \Pi \setminus \Gamma^X_\Pi$, then*
   (a) $body^+(r') \not\subseteq X$ or
   (b) $body^-(r') \cap \{head(r_j) \mid j < i\} \neq \emptyset$ or
   (c) $head(r') \in X$.

This definition differs in two ways from its predecessors. First, it drops any requirement on groundedness, expressed by Condition 0 above. This corresponds to using (X, X) instead of (X_i, X) in Definition 7. Hence, groundedness is fully disconnected from order preservation. In fact, observe that the B-preferred answer set $\{a, b\}$ of $(\Pi_5, <)$ is associated with the $<^{\text{B}}$-preserving sequence $\langle r_1, r_2 \rangle$, while the standard answer set itself is generated by the grounded sequence $\langle r_2, r_1 \rangle$.

Second, Condition 2c is more relaxed than in Definition 5. That is, any rule r' whose head is in X (as opposed to X_i) is taken as "applied". Apart from this, Condition 2c also integrates the filter-conditions from (4).[8] For illustration, consider Example (3) extended by $r_3 < r_2$:

$$\begin{array}{llll} r_1: & a & \leftarrow & not\ b \\ r_2: & b & \leftarrow & \\ r_3: & a & \leftarrow & \end{array} \qquad r_3 < r_2 < r_1 \quad (6)$$

While this program has no D- or W-preferred answer set, it has a B-preferred one: $\{a, b\}$ generated by $\langle r_2, r_3 \rangle$. The critical rule r_1 is handled by 2c. As a net result, Condition 2 is weaker than its counterpart in Definition 5.

We have the following results.

Theorem 5 *Let $(\Pi, <)$ be a statically ordered logic program over $\mathcal{L}$ and let X be a consistent answer set of Π.*

Then, the following propositions are equivalent.

1. *X is B-preferred;*
2. *$\mathcal{C}^{\text{B}}_{(Z_X(\Pi),<)}(X) = X$;*
3. *$X = Y \cap \mathcal{L}$ for some answer set Y of $\mathcal{T}^{\text{B}}(\Pi, <)$ (where $\mathcal{T}^{\text{B}}$ is defined in [Delgrande* et al.*, 2000]);*
4. *X is a $<^{\text{B}}$-preserving answer set of Π.*

Unlike theorems 1 and 3, the last result stipulates that X must be an answer set of Π. This requirement can only be dropped in case 3, while all other cases rely on this property.

7 Relationships

Up to now, we have tried to clarify the structural differences between the respective approaches. This has led to homogeneous characterizations that allow us to compare the examined approaches in a uniform way. As a result, we obtain insights into the relationships among these approaches.

First of all, we observe that all three approaches treat the blockage of (higher-ranked) rules in the same way. That is, a rule r' is found to be blocked if either its prerequisites in $body^+(r')$ are *never* derivable or if some member of $body^-(r')$ has been derived by higher-ranked or unrelated rules. This is reflected by the identity of conditions IIa and 2a/b in all three approaches, respectively. Although this is

[6] While this is integrated into [Brewka and Eiter, 1999, Def. 4.4], it is made explicit in [Brewka and Eiter, 2000, Def. 6].

[7] We have refrained from integrating (4) in order to keep the fixpoint operator comparable to those given in the previous sections. This is taken care of in the second proposition of Theorem 5.

[8] Condition $body^-(r') \cap X \neq \emptyset$ in (4) is obsolete since $r' \notin \Gamma^X_\Pi$.

arguably a sensible strategy, it leads to the loss of preferred answer sets on programs like

$$\begin{array}{llll} r_1: & a & \leftarrow & \textit{not}\ b \\ r_2: & b & \leftarrow & . \end{array} \qquad r_2 < r_1$$

Let us now discuss the differences among the approaches. The difference between D- and W-preference can be directly read off Definition 1 and 4; it manifests itself in Condition IIb and leads to the following relationship.

Theorem 6 *Every* D*-preferred answer set is* W*-preferred.*

Example (3) shows that the converse does not hold.

Interestingly, a similar relationship is obtained between W- and B-preference. In fact, Definition 8 can be interpreted as a weakening of Definition 5 by dropping Condition 0 and weakening Condition 2 (via 2c). We thus obtain the following result.

Theorem 7 *Every* W*-preferred answer set is* B*-preferred.*

Example (5) shows that the converse does not hold.

We obtain the following summarizing result by letting $\mathcal{AS}(\Pi) = \{X \mid C_\Pi(X) = X\}$ and $\mathcal{AS}_P(\Pi, <) = \{X \in \mathcal{AS}(\Pi) \mid X \text{ is } P\text{-preferred}\}$ for $P = \text{W}, \text{D}, \text{B}$.

Theorem 8 *Let* $(\Pi, <)$ *be a statically ordered logic program. Then, we have:*

$$\mathcal{AS}_{\text{D}}(\Pi, <) \subseteq \mathcal{AS}_{\text{W}}(\Pi, <) \subseteq \mathcal{AS}_{\text{B}}(\Pi, <) \subseteq \mathcal{AS}(\Pi)$$

In principle, this hierarchy is induced by a decreasing interaction between groundedness and preference. While D-preference requires the full compatibility of both concepts, this interaction is already weakened in W-preference, before it is fully abandoned in B-preference. This is nicely reflected by the evolution of Condition 0 in definitions 2, 5, and 8.

Notably, groundedness as such is not the ultimate distinguishing factor, as demonstrated by the fact that prerequisite-free programs do not necessarily lead to the same preferred answer sets, as witnessed in (3) and (6). Rather it is the degree of interaction between groundedness and preferences that makes the difference.

8 Conclusion

The notion of preference seems to be pervasive in logic programming when it comes to knowledge representation. This is reflected by numerous approaches that aim at enhancing logic programming with preferences in order to improve knowledge representation capacities. Despite the large variety of approaches, however, only very little attention has been paid to their structural differences and sameness, finally leading to solid semantical underpinnings.

This work is a first step towards a systematic account to logic programming with preferences. We elaborated upon three different approaches that were originally defined in rather heterogenous ways. We obtained three alternative yet uniform ways of characterizing preferred answer sets (in terms of fixpoints, order preservation, and an axiomatic account). The underlying uniformity provided us with a deeper understanding of how and which answer sets are preferred in each approach. This has led to a clarification of their relationships and subtle differences. In particular, we revealed that the investigated approaches yield an increasing number of answer sets depending on how tight they connect preference to groundedness.

An interesting technical result of this paper is given by the equivalences between the fixpoint operators and the standard logic programming operators applied to the correspondingly transformed programs (cf. Theorem 2 and 4). This opens the avenue for further concepts of preference handling on the basis of the alternating fixpoint theory and its issuing semantics. Further research includes dynamic preferences and more efficient algorithms for different semantics in a unifying way.

Acknowledgements. This work was supported by DFG under grant FOR 375/1-1, TP C.

References

[Baader and Hollunder, 1993] F. Baader and B. Hollunder. How to prefer more specific defaults in terminological default logic. In *Proc. IJCAI'93*, p 669–674, 1993.

[Brewka and Eiter, 1999] G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. *Artificial Intelligence*, 109(1-2):297–356, 1999.

[Brewka and Eiter, 2000] G. Brewka and T. Eiter. Prioritizing default logic. In St. Hölldobler, ed, *Intellectics and Computational Logic*. Kluwer, 2000. To appear.

[Brewka, 1994] G. Brewka. Adding priorities and specificity to default logic. In L. Pereira and D. Pearce, eds, *Proc. JELIA'94*, p 247–260. Springer, 1994.

[Delgrande *et al.*, 2000] J. Delgrande, T. Schaub, and H. Tompits. Logic programs with compiled preferences. In *Proc. ECAI 2000*, p 392–398. IOS Press, 2000.

[Gelfond and Lifschitz, 1991] M. Gelfond and V. Lifschitz. Classical negation in logic programs and deductive databases. *New Generation Computing*, 9:365–385, 1991.

[Gelfond and Son, 1997] M. Gelfond and T. Son. Reasoning with prioritized defaults. In J. Dix, L. Pereira, and T. Przymusinski, eds, *Workshop on Logic Programming and Knowledge Representation*, p 164–223. Springer, 1997.

[Rintanen, 1995] J. Rintanen. On specificity in default logic. In *Proc. IJCAI'95*, p 1474–1479. Morgan Kaufmann, 1995.

[Sakama and Inoue, 1996] C. Sakama and K. Inoue. Representing priorities in logic programs. In M. Maher, ed, *Proc. JCSLP'96*, p 82–96. MIT Press, 1996.

[van Gelder, 1993] A. van Gelder. The alternating fixpoint of logic programs with negation. *J. Computer and System Science*, 47:185–120, 1993.

[Wang *et al.*, 2000] K. Wang, L. Zhou, and F. Lin. Alternating fixpoint theory for logic programs with priority. In *Proc. Int'l Conf. Computational Logic*, p 164-178. Springer, 2000.

[Zhang and Foo, 1997] Y. Zhang and N. Foo. Answer sets for prioritized logic programs. In J. Maluszynski, ed, *Proc. ISLP'97*, p 69–84. MIT Press, 1997.

Reasoning with infinite stable models

Piero A. Bonatti
Dip. di Tecnologie dell'Informazione - Università di Milano
I-26013 Crema, Italy
bonatti@dti.unimi.it

Abstract

The existing proof-theoretic and software tools for nonmonotonic reasoning can only handle finite domains. In this paper we introduce a class of normal logic programs, called *finitary programs*, whose domain may be infinite, and such that credulous and skeptical entailment under the stable model semantics are computable. Finitary programs—that are characterized by two conditions on their dependency graph—are computationally complete (they can simulate arbitrary Turing machines). Further results include a compactness theorem and the proof that the two conditions defining finitary programs are, in some sense, "minimal". The existing methods for automated nonmonotonic reasoning are either complete for finitary programs, or can be easily extended to cover them.

1 Introduction

There is a complexity gap between propositional and first-order non-monotonic theories. Finite propositional theories are decidable (as in first-order logic), while the consequences of first-order theories are not recursively enumerable, in general. A computationally complete (Turing-equivalent) layer is missing between the two classes of theories. For this reason, most of the research on automated reasoning and proof-theory for nonmonotonic logics has been focussed on finite propositional theories or equivalent formalisms, such as function-free logic programs. In this way, the ability of reasoning about recursive data structures and infinite domains (such as lists, trees, XML/HTML documents, time, and so on) is completely lost. This is a strong limitation, both for standard tasks—such as reasoning about action and change when time is explicitly represented—and for emerging applications—such as using XML document bases as knowledge bases.

Of course, there may exist interesting, semi-decidable fragments of first-order nonmonotonic logics. For example, in some cases, the second-order circumscription formula can be expressed as a finite first-order formula. Similar examples are missing so far for Default Logic and Autoepistemic Logic.

Normal logic programs under the stable model semantics can be regarded as a fragment of both logics. In this paper we introduce a Turing-equivalent class of programs, called *finitary* programs, that opens the way to effective default and autoepistemic reasoning about infinite domains. The main theoretical features of finitary programs are the following:

- Their domain may be infinite.
- Nontheless both credulous and skeptical reasoning are semi-decidable. Ground queries are decidable.
- A form of compactness holds.
- Finitary programs are computationally complete, i.e. each Turing machine can be simulated by some finitary program.
- Finitary programs are defined by two conditions on their dependency graph, that are minimal, in the sense that if any condition were dropped, then semi-decidability and compactness would not be guaranteed anymore.

The paper is organized as follows: after a few technical preliminaries (Section 2), we characterize the subprogram needed to reason about a given ground formula F (Section 3). Then we introduce finitary programs (Section 4) and study their theoretical properties and expressiveness. In Section 5, the completeness proof for the skeptical resolution calculus introduced in [Bonatti, 1997] (originally formulated for function-free programs) is extended to all finitary programs and a generalization thereof (*almost finitary programs*). After a brief sketch of how existing credulous reasoners can be extended to finitary programs (Section 6), we conclude with a discussion of the results and related work. Some of the proofs are omitted due to space limitations.

2 Preliminaries

We assume the reader to be familiar with the classical theory of logic programming [Lloyd, 1984]. *Normal logic programs* (hereafter called simply "programs") are sets of rules of the form $A \leftarrow L_1, \ldots, L_n$ ($n \geq 0$) such that A is a logical atom and each L_i ($i = 1, \ldots n$) is a literal. As usual by "head" and "body" of such a rule we mean A and $L_1, \ldots, L_n$, respectively. A program is positive if it contains no occurrences of $\neg$. The ground instantiation of a program P is denoted by $Ground(P)$. The *Gelfond-Lifschitz transformation* P^I of program P w.r.t. an Herbrand interpretation I (represented, as usual as a set of ground atoms) is obtained by removing from $Ground(P)$ all the rules containing a negative literal

$\neg B$ such that $B \in I$, and by removing from the remaining rules all negative literals. An interpretation M is a *stable model* of P if M is the least Herbrand model of P^M. A formula F is *credulously* (resp. *skeptically*) *entailed* by P iff F is satisfied by some (resp. each) stable model of P.

The *dependency graph* of a program P is a labelled directed graph whose vertices are the ground atoms of P's language. Moreover, *i)* there exists an edge from B to A iff there is a rule $r \in \mathit{Ground}(P)$ with A in the head and an occurrence of B in the body; *ii)* such edge is labelled "negative" if B occurs in the scope of $\neg$, and "positive" otherwise. An atom A depends positively (resp. negatively) on B if there is a directed path from B to A in the dependency graph with an even (resp. odd) number of negative edges. A program is *call-consistent* if no atom depends negatively on itself. By *odd-cycle* we mean a cycle in the dependency graph with an odd number of negative edges. Call-consistency coincides with the absence of odd-cycles. Every call-consistent program has at least one stable model [Dung, 1992].

In the context of normal logic programs, a *splitting set* for a program P [Lifschitz and Turner, 1994] is a set of atoms U containing all the atoms occurring in the body of any rule $r \in \mathit{Ground}(P)$ whose head is in U. The set of rules $r \in \mathit{Ground}(P)$ whose head is in U—called the "bottom" of P w.r.t. U—will be denoted by $b_U(P)$. By $e_U(P, I)$ we denote the following partial evaluation of P w.r.t. $I \cap U$: remove from $\mathit{Ground}(P)$ each rule $A \leftarrow L_1, \ldots, L_n$ such that some L_i containing an atom of U is false in I, and remove from the remaining rules all the L_i containing a member of U. The following is a specialization to normal programs of a result in [Lifschitz and Turner, 1994].

Theorem 2.1 (Splitting theorem) *Let U be a splitting set for a normal logic program P. An interpretation M is a stable model of P iff $M = J \cup I$, where*

1. *I is a stable model of $b_U(P)$, and*
2. *J is a stable model of $e_U(P \setminus b_U(P), I)$.*

3 Relevant subprograms

This section contains the basic technical results needed to investigate decidability and undecidability issues. These results prove that a certain strict subset of $\mathit{Ground}(P)$ suffices to decide credulous and skeptical entailment. Intuitively, all is needed for inference are the rules affecting the behavior of odd-cycles (which may generate inconsistencies in the form of instability) and the definitions of the predicates on which the given goal F depends.

Definition 3.1 [Relevant universe and subprogram] The *relevant universe* for a ground formula F (w.r.t. program P), denoted by $U(P, F)$, is the set of all ground atoms A such that the dependency graph of P contains a path from A to an atom occurring either in F or in some odd-cycle of the graph.

The *relevant subprogram* for a ground formula F (w.r.t program P), denoted by $R(P, F)$, is the set of all rules in $\mathit{Ground}(P)$ whose head belongs to $U(P, F)$. ∎

Example 3.2 Consider the program P consisting of the rules:

$$
\begin{aligned}
& p(f(X)) \leftarrow p(X), q(X) \\
& q(X) \leftarrow s(X) \\
& u(a) \leftarrow \neg u(a) \\
& z(X) \leftarrow p(X)
\end{aligned}
$$

and let $F = p(f(a))$. Here

$$
\begin{aligned}
U(P, F) &= \{\, p(f(a)), p(a), q(a), s(a), u(a) \,\}\ , \\
R(P, F) &= \{\ (p(f(a)) \leftarrow p(a), q(a)),\ (q(a) \leftarrow s(a)), \\
&\qquad (u(a) \leftarrow \neg u(a))\}\ .
\end{aligned}
$$

∎

Note that for all rules $r \in \mathit{Ground}(P)$, if the head of r belongs to $U(P, F)$ then also all the other atoms in r belong to $U(P, F)$. The next proposition follows easily.

Proposition 3.3 *$U(P, F)$ is a splitting set for $\mathit{Ground}(P)$, and $R(P, F)$ coincides with $b_{U(P,F)}(P)$.*

Now the basic technical results can be proved.

Lemma 3.4 *For all ground formulae F, $R(P, F)$ has a stable model M_F iff P has a stable model M such that $M \cap U(P, F) = M_F$.*

Proof. ("If" part) Suppose M is a stable model of P. By Proposition 3.3, $U(P, F)$ is a splitting set for P and $R(P, F) = b_{U(P,F)}(P)$. Then, by the splitting theorem [Lifschitz and Turner, 1994], there exist a stable model I of $R(P, F)$, and a stable model J of $e_{U(P,F)}(P \setminus R(P, F), I)$, such that $M = I \cup J$. By definition, no atom in $U(P, F)$ occurs in $e_{U(P,F)}(P \setminus R(P, F), I)$, therefore $J \cap U(P, F) = \emptyset$. It follows that $M \cap U(P, F) = I$, and hence $M \cap U(P, F)$ is a stable model of $R(P, F)$.

("Only if" part) Suppose $R(P, F)$ has a stable model M_F. By definition, all the atoms occurring in an odd-cycle belong to $U(P, F)$. Consequently, the dependency graph of $e_{U(P,F)}(P \setminus R(P, F), I)$ contains no odd-cycles, i.e. $e_{U(P,F)}(P \setminus R(P, F), I)$ is call-consistent. Then, by a well-known result in [Dung, 1992], $e_{U(P,F)}(P \setminus R(P, F), I)$ has a stable model J. Let $M = J \cup M_F$. By the splitting theorem, the interpretation M is a stable model of P. Moreover, since $J \cap U(P, F) = \emptyset$ (cf. point 1), $M \cap U(P, F) = M_F$. ∎

Theorem 3.5 *For all ground formulae F,*

1. *P credulously entails F iff $R(P, F)$ does.*
2. *P skeptically entails F iff $R(P, F)$ does.*

Proof. if P credulously entails F, then there exists a stable model M of P such that $M \models F$. By Lemma 3.4, $M \cap U(P, F)$ is a stable model of $R(P, F)$. Moreover, since by definition $U(P, F)$ contains all the atoms occurring in F, F must have the same truth value in M and $M \cap U(P, F)$, and hence $M \cap U(P, F) \models F$. As a consequence $R(P, F)$ credulously entails F.

Conversely, suppose that $R(P, F)$ credulously entails F. Then there exists a stable model M_F of $R(P, F)$ such that $M_F \models F$. By Lemma 3.4, P has a stable model M such that $M \cap U(P, F) = M_F$. Then the models M and M_F must agree on the valuation of F (cf. the "only if" part of the proof) and hence $M \models F$, which means that P credulously entails F. This completes the proof of 1).

To prove 2), we demonstrate the equivalent statement:

P does not skeptically entail F iff $R(P, F)$ does not skeptically entail F.

This statement is equivalent to: P credulously entails $\neg F$ iff $R(P, F)$ credulously entails $\neg F$, that follows immediately from 1). ∎

4 Finitary programs

Now we introduce a class of programs whose consequences are recursively enumerable, although their domain may be infinite.

Definition 4.1 [Finitary programs] We say a program P is *finitary* if the following conditions hold:

1. For each node A of the dependency graph of P, the set of all nodes B such that A depends (either positively or negatively) on B is finite.
2. Only a finite number of nodes of the dependency graph of P occurs in an odd-cycle. ∎

For example, most classical programs on recursive data structures such as lists and trees (e.g. predicates `member`, `append`, `reverse`) satisfy the first condition. In these programs, the terms occurring in the body of a rule occur also in the head, typically as strict subterms of the head's arguments. This property clearly entails Condition 1.

The second condition is satisfied by most of the programs used for embedding NP-hard problems into logic programs [Cholewiński *et al.*, 1995; Eiter and Gottlob, 1993; Gottlob, 1992]. Such programs can be (re)formulated by using a single odd cycle involving one atom p and defined by simple rules such as $p \leftarrow \neg p$ and $p \leftarrow f, \neg p$ (if p does not occur elsewhere, then f can be used as the logical constant *false* in the rest of the program).

An example of finitary program without odd-cycles is illustrated in Figure 1. It credulously entails a ground goal $\mathtt{s}(t)$ iff t encodes a satisfiable formula. By adding rule $\bot \leftarrow \neg \mathtt{s}(f), \neg\bot$ we obtain another finitary program with one odd-cycle, such that $\mathtt{s}(t)$ is skeptically entailed iff the formula encoded by t is a logical consequence of the one encoded by f.

The following proposition follows straightforwardly from the definitions of $U(P, G)$, $R(P, G)$ and finitary programs. It will be needed to prove computability results.

Proposition 4.2 *If P is finitary then, for all ground goals G, $U(P, G)$ and $R(P, G)$ are finite.*

4.1 Compactness

In classical first-order logic, an infinite set of formulae is inconsistent iff it contains an inconsistent finite subset. A similar property—that in general does not apply to nonmonotonic logics—is enjoyed by finitary programs, too.

Definition 4.3 An *unstable kernel* for a program P is a subset K of $\mathit{Ground}(P)$ with the following properties:

1. K is *downward closed*, that is, for each atom A occurring in K, K contains all the rules $r \in \mathit{Ground}(P)$ whose head is A.
2. K has no stable models. ∎

Theorem 4.4 (Compactness) *A finitary program P has no stable models iff it has a finite unstable kernel.*

Proof. Let G be any ground atom in the language of P. From Lemma 3.4, it follows that P has no stable models iff $R(P, G)$ has no stable models. Clearly, $R(P, G)$ is downward closed by definition. Moreover, by Proposition 4.2, $R(P, G)$ is finite. Therefore P has no stable models iff $R(P, G)$ is a finite unstable kernel of P. ∎

4.2 Decidability and semi-decidability of inference

Hereafter we focus on the complexity of inference within the class of finitary programs. This subsection deals with upper bounds. We start with the proof that, for all ground goals, both credulous and skeptical inference are decidable.

Theorem 4.5 *For all finitary programs P and ground goals G, both the problem of deciding whether G is a credulous consequence of P and the problem of deciding whether G is a skeptical consequence of P are decidable.*

Proof. By Theorem 3.5, G is a credulous (resp. skeptical) consequence of P iff G is a credulous (resp. skeptical) consequence of $R(P, G)$. Moreover, by Proposition 4.2, $R(P, G)$ is finite, so the set of its stable models can be computed in finite time. It follows that the inference problems for P and G are both decidable. ∎

It follows easily that existentially quantified goals are semi-decidable.

Theorem 4.6 *For all finitary programs P and all goals G, both the problem of deciding whether $\exists G$ is a credulous consequence of P and the problem of deciding whether $\exists G$ is a skeptical consequence of P are semi-decidable.*

Proof. The formula $\exists G$ is credulously (res. skeptically) entailed by P iff there exists a grounding substitution θ such that $G\theta$ is credulously (res. skeptically) entailed by P. The latter problem is decidable (by Theorem 4.5), and all grounding θ for G can be recursively enumerated, so existential entailment can be reduced to a potentially infinite recursive sequence of decidable tests, that terminates if and only if some $G\theta$ is entailed. ∎

We shall see in the following section that this is a strict bound, i.e. existential entailment can be undecidable (cf. Corollary 4.11).

4.3 Minimality and expressiveness

Next we focus on lower bounds to the complexity of inference for the class of finitary programs and relaxations thereof. The next two results show that both of the conditions in Definition 4.1 are necessary for semi-decidability, i.e. Definition 4.1 is in some sense minimal.

Proposition 4.7 *Credulous and skeptical inference are not semi-decidable for the class of all programs satisfying Condition 2 of Definition 4.1.*

Proof. Note that locally stratified programs trivially satisfy Condition 2 of Definition 4.1 while some of them may violate Condition 1. For locally stratified programs, credulous and skeptical inference coincide with the well-founded semantics [Gelfond and Lifschitz, 1988] and are not semi-decidable [Schlipf, 1990], so the proposition immediately follows. ∎

```
s(and(X,Y)) ← s(X), s(Y)        s(A) ← member(A, [p, q, r, s]), ¬ns(A)
s(or(X,Y)) ← s(X)               ns(A) ← member(A, [p, q, r, s]), ¬s(A)
s(or(X,Y)) ← s(Y)               member(A, [A|L])
s(not(X)) ← ¬s(X)               member(A, [B|L]) ← member(A, L)
```

Figure 1: A finitary program for SAT

The other lower-bound results are based on positive programs that decide whether a given Turing machine terminates using a fixed portion of its tape.

Let $\mathcal{M}$ be an arbitrary Turing machine; let S and V be $\mathcal{M}$'s set of states and vocabulary, respectively. Recall that the actions of $\mathcal{M}$ are defined by 5-tuples $\langle s, v, v', s', m \rangle$, where s and v are the current state and symbol, respectively, v' is the symbol to be overwritten on v, s' is the next state, and m specifies $\mathcal{M}$'s head movement.

Consider the positive program $P_{\mathcal{M}}$ in Figure 2. The reader may easily verify that $\mathtt{t}(s, \mathtt{L}, \mathtt{V}, \mathtt{R})$ is entailed by $P_{\mathcal{M}}$ iff there exists a finite computation of $\mathcal{M}$, starting from a configuration with state s and tape described by $\mathtt{L}, \mathtt{V}, \mathtt{R}$, and using only the finite portion of tape corresponding to $\mathtt{L}, \mathtt{V}, \mathtt{R}$. Here $\mathtt{L}$ is a list representing a finite portion of the tape on the left of $\mathcal{M}$'s head, in reverse order; $\mathtt{V}$ is the current symbol; $\mathtt{R}$ is a (non reversed) finite portion of the tape on the right of $\mathcal{M}$'s head.

Program $P_{\mathcal{M}}$ satisfies Condition 1 of Definition 4.1. To prove this, note that the recursive calls to predicate $\mathtt{t}$ preserve the length k of the portion of tape encoded in the head. Therefore, for each ground atom $\mathtt{t}(s, l, v, r)$, the number of atoms $\mathtt{t}(s', l', v', r')$ connected to $\mathtt{t}(s, l, v, r)$ by a directed path in the dependency graph is bounded by $|S| \cdot |V|^k$.

The next two results are based on $P_{\mathcal{M}}$ and prove that if Condition 2 in Definition 4.1 were dropped, then inference would not be semi-decidable anymore. More precisely, the theorems say that some inferences would be as complex as deciding the termination of an arbitrary Turing machine.

Theorem 4.8 *For each Turing machine $\mathcal{M}$ with initial state s and tape τ with non-blank portion $\langle v_0, v_1, \ldots, v_n \rangle$, a program $P^1_{\mathcal{M}}$ and a goal G can be recursively constructed, such that $P^1_{\mathcal{M}}$ satisfies only Condition 1 in Definition 4.1, and $P^1_{\mathcal{M}}$ skeptically entails G iff $\mathcal{M}$ terminates.*

Proof. (Sketch) Let $P^1_{\mathcal{M}}$ consist of the program defined in Figure 2 plus the rules

$$\mathtt{u}(\mathtt{L},\mathtt{R}) \leftarrow \mathtt{blank_list}(\mathtt{L}), \mathtt{blank_list}(\mathtt{R}),$$
$$\mathtt{t}(s, \mathtt{L}, v_0, [v_1, \ldots, v_n \mid \mathtt{R}]),\ \neg\mathtt{u}(\mathtt{L},\mathtt{R}).$$
$$\mathtt{blank_list}([\,]).$$
$$\mathtt{blank_list}([b \mid \mathtt{L}]) \leftarrow \mathtt{blank_list}(\mathtt{L}).$$

where b represents the blank symbol. $P^1_{\mathcal{M}}$ satisfies Condition 1 of Definition 4.1 (cf. the argument for $P_{\mathcal{M}}$). Condition 2 of Definition 4.1 is violated, as there exist infinitely many odd-cycles, one for each ground atom $\mathtt{u}(x, y)$. Clearly, for any grounding substitution θ, the goal

$$(\mathtt{blank_list}(\mathtt{L}),\ \mathtt{blank_list}(\mathtt{R}),\ \mathtt{t}(s, \mathtt{L}, v_0, [v_1, \ldots, v_n \mid \mathtt{R}]))\theta$$

can be derived from the subprogram $P_{\mathcal{M}}$ and the clauses for $\mathtt{blank_list}$ iff $\mathcal{M}$ terminates using the portion of τ represented by $\mathtt{L}\theta, v_0, [v_1, \ldots, v_n \mid \mathtt{R}\theta]$. At the same time, $P^1_{\mathcal{M}}$ has a stable model iff the above goal cannot be derived from $P_{\mathcal{M}}$, because of the odd-cycle containing $\mathtt{u}(\mathtt{R},\mathtt{L})\theta$. It follows that for all G consisting of a propositional symbol not occurring in $P^1_{\mathcal{M}}$, $P^1_{\mathcal{M}}$ skeptically entails G iff $\mathcal{M}$ terminates. ∎

Theorem 4.9 *For each Turing machine $\mathcal{M}$ with initial state s and tape τ with non-blank portion $\langle v_0, v_1, \ldots, v_n \rangle$, a program $P^2_{\mathcal{M}}$ and a goal G can be recursively constructed, such that $P^2_{\mathcal{M}}$ satisfies only Condition 1 in Definition 4.1, and $P^2_{\mathcal{M}}$ credulously entails G iff $\mathcal{M}$ terminates.*

Proof. Let $P^2_{\mathcal{M}} = P^1_{\mathcal{M}} \cup \{p\}$, where p is a new propositional symbol not occurring in $P^1_{\mathcal{M}}$, and let $G = p$. Clearly $P^2_{\mathcal{M}}$ credulously entails G iff $P^2_{\mathcal{M}}$ has a stable model. Since $P^2_{\mathcal{M}}$ has a stable model iff $\mathcal{M}$ terminates (cf. the previous proof), the theorem immediately follows. ∎

The next result is based on a modification of $P_{\mathcal{M}}$ that keeps track of the result of the computation. The modified program $P^3_{\mathcal{M}}$ has one extra argument to return the final state of the tape. The theorem proves that the class of finitary programs is computationally complete by showing how any Turing machine can be simulated by a suitable finitary program.

Theorem 4.10 *For each Turing machine $\mathcal{M}$ with initial state s and tape τ with non-blank portion $\langle v_0, v_1, \ldots, v_n \rangle$, a (positive) finitary program $P^3_{\mathcal{M}}$ and a goal $\mathtt{p}(\mathtt{L},\mathtt{R},\mathtt{X})$ can be recursively constructed, in such a way that for all grounding substitutions θ, $P^3_{\mathcal{M}} \models \mathtt{p}(\mathtt{L},\mathtt{R},\mathtt{X})\theta$ iff $\mathcal{M}$ terminates and $\mathtt{X}\theta$ encodes the final tape of the computation.*

Corollary 4.11 *The problems of deciding whether a finitary program P credulously/skeptically entails an existentially quantified goal $\exists G$ are not decidable.*

5 Resolution calculus

Here the results of the previous sections are applied to the resolution calculus for skeptical stable model semantics introduced in [Bonatti, 1997]. In the original paper, the resolution calculus was proved complete w.r.t. function-free programs (soundness holds for all programs). In this section, we extend the completeness result to all finitary programs.

Theorem 5.1 *Let P be a finitary program, and let $(\bigwedge H \rightarrow \bigwedge G)\gamma$ be a ground skeptical consequence of P, where H and G are sequences of literals. Then the skeptical goal $(G \mid H)$ has a successful skeptical derivation from P with answer substitution θ more general than γ.*

$\mathtt{t}(s,\mathtt{L},v,[\mathtt{V}\mid\mathtt{R}]) \leftarrow \mathtt{t}(s',[v'\mid\mathtt{L}],\mathtt{V},\mathtt{R})$	for all 5-tuples $\langle\, s,v,v',s',\mathtt{right}\,\rangle$
$\mathtt{t}(s,[\mathtt{V}\mid\mathtt{L}],v,\mathtt{R}) \leftarrow \mathtt{t}(s',\mathtt{L},\mathtt{V},[v'\mid\mathtt{R}])$	for all 5-tuples $\langle\, s,v,v',s',\mathtt{left}\,\rangle$
$\mathtt{t}(s,\mathtt{L},v,\mathtt{R})$	if there exists no tuple for s and v.

Figure 2: A program deciding termination of a Turing machine with bounded tape

Proof. Let $F = (\bigwedge H \rightarrow \bigwedge G)\gamma$. Since F is a ground skeptical consequence of P, then F is also a skeptical consequence of $R(P,F)$ (by Theorem 3.5). Moreover, $R(P,F)$ is finite (by Proposition 4.2). Then, by the original completeness result [Bonatti, 1997], $(G \mid H)\gamma$ has a ground skeptical derivation from $R(P,F) \subseteq \mathit{Ground}(P)$. By standard lifting techniques (cf. [Lloyd, 1984]), a corresponding skeptical derivation of $(G \mid H)$ from P with answer substitution θ more general than γ can easily be obtained. ∎

Example 5.2 Consider again the program P of Example 3.2. This finitary program has no stable model, because of its third rule. Figure 3 shows a successful skeptical derivation for the skeptical conclusion $p(X)$, with empty answer substitution (which means that $\forall X.p(X)$ is a skeptical consequence of P). The Restricted Split of type II introduces two subgoals, each with a new hypothesis ($u(a)$ and $\neg u(a)$, respectively), obtained from an atom occurring in some odd-cycle. The Contradiction rule replaces the left-hand side of a goal with the negation of some hypothesis. Intuitively, the goal is proved by showing that the hypotheses on the right hand side cannot be satisfied. The Failure rule rewrites a goal with one of its *counter-supports*. In this example the counter-support of $u(a)$ is $u(a)$ itself (obtained by negating the unique support $\{\neg u(a)\}$ of $u(a)$). Resolution can be performed either with a program rule or with one of the hypotheses (treated as facts). Finally, the Success rule removes goals where there is nothing left to prove. As usual, the empty goal sequence is denoted by $\square$. ∎

With the help of the resolution calculus it can be shown that a class of normal programs larger than finitary programs is Turing equivalent. The extended class—called *almost finitary*—admits the kind of rules typically used to generate inconsistencies, without the restrictions of Definition 4.1.

Definition 5.3 [Almost finitary programs] A normal program P is *almost finitary* iff it can be partitioned into two (disjoint) subsets, P_1 and P_2, such that P_1 is finitary, and P_2 consists of rules of the form $A \leftarrow B_1,\ldots,B_n,\neg A$. ∎

Note that there is no restriction on P_2's rule variables. Given a ground instance $(A \leftarrow B_1,\ldots,B_n,\neg A)\theta$ of a rule in P_2, $A\theta$ may depend on infinitely many ground atoms B (i.e., Condition 1 of Definition 4.1 can be violated).

Theorem 5.4 *The skeptical resolution calculus is complete (in the sense of Theorem 5.1) for all almost finitary programs.*

Corollary 5.5 *The set of skeptical consequences of the form $\exists(\bigwedge H \rightarrow \bigwedge G)$, where H and G are sequences of literals, is semi-decidable for almost finitary programs.*

Remark 5.6 For almost finitary programs, ground credulous inference is not semi-decidable. This means that *skeptical inference from almost finitary programs cannot be implemented through credulous inference*. Only direct approaches such as the resolution calculus are possible.

6 Credulous automated reasoners

For function-free programs, there exist powerful automated reasoners based upon stable model generation, such as SMODELS [Niemelä and Simons, 1997] and DLV [Eiter *et al.*, 1997]. Internally, these engines operate on ground programs; for this reason they embody smart program instantiation routines. Such routines are applied before any other form of reasoning. It seems not difficult to extend the instantiation routines to deal with all finitary programs, so that given a ground goal G [1], only the relevant (ground and finite) subprogram $R(P,G)$ (or an equivalent subset thereof)[2] is generated. The rest of the engine needs no modification. By Theorem 3.5, soundness and completeness are preserved. In this way, our results can be used to extend the applicability range of this class of automated reasoners, with no modification to the core of their reasoning mechanisms.[3]

7 Summary and related work

For the first time, semi-decidable fragments of the stable model semantics have been explored in depth, and related to the number of atoms involved in odd-cycles. The main contributions of this paper can be summarized by recalling that finitary programs (Def. 4.1) are computationally complete (Theorem 4.10), can deal with function symbols, and enjoy a compactness result (Theorem 4.4) that guarantees semi-decidability of inference (Theorem 4.6). Finitary programs are characterized by two conditions on their dependency graph that are minimal, in the sense that if any of them were dropped, then inference would not always be semi-decidable (Theorems 4.7, 4.8 and 4.9). We proved that the skeptical resolution calculus is complete for finitary and almost finitary programs (Theorems 5.1 and 5.4), and sketched how other engines for nonmonotonic reasoning can be extended to deal with finitary programs (Section 6). We are currently extending our results to larger classes of programs, and to partial stable models.[4]

The work on finitary programs contributes to providing nonmonotonic logics with classical proof- and model-

[1]These engines can only deal with ground queries

[2]The smart instantiation routines remove some non-applicable rule instances.

[3]As noted by an anonymous referee, the same idea can be applied to function-free programs, in order to reduce the cost of program instantiation.

[4]The latter idea and evidence to its feasibility have been suggested by an anonymous referee.

$(p(X) \mid\ )$		(Initial goal)
$(p(X) \mid u(a))$	$(p(X) \mid \neg u(a))$	Restricted Split of type II
$(\neg u(a) \mid u(a))$	$(p(X) \mid \neg u(a))$	Contradiction rule
$(u(a) \mid u(a))$	$(p(X) \mid \neg u(a))$	Failure rule
$(\Box \mid u(a))$	$(p(X) \mid \neg u(a))$	Resolution with hypothesis
	$(p(X) \mid \neg u(a))$	Success rule
	$(u(a) \mid \neg u(a))$	Contradiction rule
	$(\neg u(a) \mid \neg u(a))$	Resolution with $u(a) \leftarrow \neg u(a)$
	$(\Box \mid \neg u(a))$	Resolution with hypothesis
	$\Box$	Success rule

Figure 3: A skeptical derivation

theoretic tools and results. Work in a similar direction comprises some pretty standard axiomatizations based on Hilbert-style systems [Levesque, 1990][5] and sequent calculi [Olivetti, 1992; Bonatti and Olivetti, 1997]. These axiomatizations for Default and Autoepistemic logic should be extended to first-order theories, perhaps adapting the techniques introduced in this paper. An infinitary proof-theory can be found in [Milnikel, 1999]. In [Rosati, 1999], issues related to decidable nomonotonic reasoning in MKNF are investigated. In [Cenzer *et al.*, 1999], sufficient conditions for the existence of recursively enumerable stable models are identified. They do not ensure that all stable models are r.e., so skeptical and credulous reasoning are not semi-decidable.

Acknowledgments

The author is grateful to the anonymous referees for their deep and careful reviews, and for their precious suggestions.

References

[Bonatti and Olivetti, 1997] P. A. Bonatti and N. Olivetti. A sequent calculus for skeptical default logic. In *Proceedings of TABLEAUX'97*, number 1227 in LNAI, pages 107–121. Springer Verlag, 1997.

[Bonatti, 1997] P. A. Bonatti. Resolution for skeptical stable semantics. *Journal of Automated Reasoning*. To appear. Preliminary version in [Dix *et al.*, 1997].

[Cenzer *et al.*, 1999] D. Cenzer, J. B. Remmel, and A. Vanderbilt. Locally determined logic programs. In *Proc. of LPNMR'99*, number 1730 in LNAI, pages 34–48, Berlin, 1999. Springer Verlag.

[Cholewiński *et al.*, 1995] P. Cholewiński, V. Marek, A. Mikitiuk, and M. Truszczyński. Experimenting with nonmonotonic reasoning. In *Proc. of ICLP'95*. MIT Press, 1995.

[Dix *et al.*, 1997] J. Dix, U. Furbach, and A. Nerode, editors. *Proc. of LPNMR'97*, number 1265 in LNAI, Berlin, 1997. Springer Verlag.

[Dung, 1992] P. M. Dung. On the relation between stable and well-founded semantics of logic programs. *Theoretical Computer Science*, 105:7–25, 1992.

[5]This axiomatization of Autoepistemic logic, formulated for first-order theories, is complete only for the propositional fragment.

[Eiter and Gottlob, 1993] T. Eiter and G. Gottlob. Complexity results for disjunctive logic programming and applications to nonmonotonic logics. In *Proc. of ILPS'93*. MIT Press, 1993.

[Eiter *et al.*, 1997] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A deductive system for nonmonotonic reasoning. In [Dix *et al.*, 1997], 1997.

[Gelfond and Lifschitz, 1988] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of the 5th ICLP*, pages 1070–1080. MIT Press, 1988.

[Gottlob, 1992] G. Gottlob. Complexity results for nonmonotonic logics. *Journal of Logic and Computation*, 2:397–425, 1992.

[Levesque, 1990] H. J. Levesque. All I know: a study in autoepistemic logic. *Artificial Intelligence*, 42:263–309, 1990.

[Lifschitz and Turner, 1994] V. Lifschitz and H. Turner. Splitting a logic program. In *Proc. of ICLP'94*, pages 23–37. MIT Press, 1994.

[Lifschitz, 1985] V. Lifschitz. Computing circumscription. In *Proceedings of IJCAI'85*, pages 121–127, 1985.

[Lloyd, 1984] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[Milnikel, 1999] R. S. Milnikel. *Nonomonotonic logic: a monotonic approach*. PhD thesis, Cornell University, 1999.

[Niemelä and Simons, 1997] I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal LP. In [Dix *et al.*, 1997], 1997.

[Olivetti, 1992] N. Olivetti. Tableaux and sequent calculus for minimal entailment. *Journal of Automated Reasoning*, 9:99–139, 1992.

[Rosati, 1999] R. Rosati. Towards first-order nonmonotonic reasoning. In *Proc. of LPNMR'99*, number 1730 in LNAI, pages 332–346, Berlin, 1999. Springer Verlag.

[Schlipf, 1990] J. Schlipf. The expressive power of the logic programming semantics. In *Proc. of PODS'90*, 1990.

LOGIC PROGRAMMING AND THEOREM PROVING

THEOREM PROVING

Splitting Without Backtracking*

Alexandre Riazanov and Andrei Voronkov
The University of Manchester

Abstract

Integrating the splitting rule into a saturation-based theorem prover may be highly beneficial for solving certain classes of first-order problems. The use of splitting in the context of saturation-based theorem proving based on explicit case analysis (as implemented in SPASS) employs backtracking which is difficult to implement as it affects design of the whole system. Here we present a "cheap" and efficient technique for implementing splitting that does not use backtracking.

1 Introduction

Case analysis in the form of the β-rule is the core of tableau-based theorem proving methods. If our aim is to refute a set of clauses $S \cup \{\phi \vee \psi\}$, we can reduce this task to refuting two simpler sets: $S \cup \{\phi\}$ and $S \cup \{\psi\}$. This is justified by the following argument. To show that S is false in all the models of the formula $\phi \vee \psi$, one can separately consider two cases: all the models of ϕ and the ones not necessarily satisfying ϕ but satisfying ψ. The saturation based theorem proving has adopted the case analysis principle in the form of splitting that can be formulated in the propositional case as the following inference rule:

$$
\begin{array}{ccc}
 & S \cup \{C_1 \vee C_2\} & \\
\swarrow & & \searrow \\
S \cup \{C_1\} & & S \cup \{C_2\}
\end{array}
$$

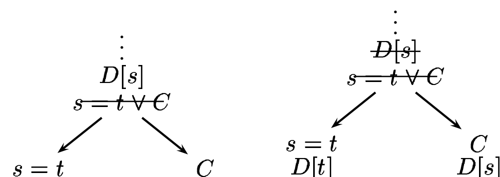
where S is a set of clauses, C_1 and C_2 are clauses. Unlike most of the other inference rules in saturation-based theorem proving that only modify a set of clauses by adding and removing some clauses, this rule makes two sets of clauses out of one. Now, to refute the original set $S \cup \{C_1 \vee C_2\}$, one must separately refute the two new sets: $S \cup \{C_1\}$ and $S \cup \{C_2\}$.

In the propositional case, this rule together with unit resolution gives a complete inference system. Finding a refutation in this system can be organised, for example, as depth-first exploring of branches. After finding a refutation on one of the splitting branches, the procedure backtracks and proceeds with the second branch. In what follows, we call this procedure *explicit case analysis*. A simple example of a propositional derivation which combines the splitting rule with resolution on branches is shown in Figure 1. The current search state is depicted as a tree. The nodes contain clauses common for all the branches below them. The clause or clauses to which an inference rule has been applied is put in a shaded box. For a moment, ignore the labels of the edges, for example p_1.

In the first-order case the splitting rule can be formulated in the same way as in the propositional case with an additional restriction: the clauses C_1 and C_2 do not have common variables. This rule can be combined with inference rules like resolution and paramodulation. Combined with some resolution strategies, it gives decision procedures for fragments of first-order logic (see e.g. [Fermüller *et al.*, 2001]).

The cost of naive backtracking is very high, since the clause set S can contain tens of thousands of clauses, so after several splits a lot of memory is needed to store backtrack points. Therefore, in practice (the SPASS prover [Weidenbach *et al.*, 1999]) explicit case analysis is implemented by adding to each clause its *split history*: the set of splits used to obtain this clause, represented by an array of bits. The split history also helps to implement *intelligent backtracking*: by analyzing the split history one can check which splits have actually been used to obtain a contradiction on the current branch.Clauses which do not belong to the currently exploited branch are put in a special waiting list, and popped back from the waiting list upon backtracking. There are complications caused by simplification by unit equalities belonging to a branch which can be illustrated by the following example.

Example 1.1 Consider the split on the left of this picture:



where C and $s = t$ have no common variables, $s \succ t$ in the simplification order used in the proof-search, and $D[s]$ is a clause with an occurrence of s. After the split the left branch

*Partially supported by grants from EPSRC and the Faculty of Science and Technology, the University of Manchester. The first author is also partially supported by an ORS award.

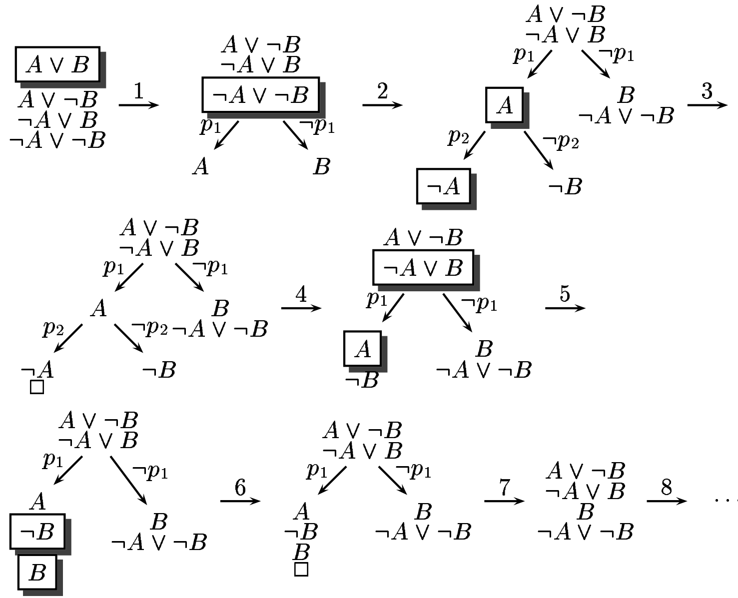


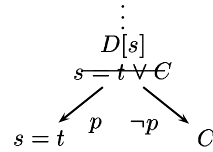
Figure 1: A derivation with splitting and unit resolution

contains two clauses $s = t$ and $D[s]$, so $D[s]$ can be simplified and replaced by $D[t]$, but only on the left branch, since there is no clause $s = t$ on the right branch. This means that $D[s]$ should still be kept on the right branch, so simplification by $s = t$ should be implemented according to the right-hand side of the picture.

To achieve efficiency, modern theorem provers maintain one or more indexes on terms, literals, or clauses. To implement splitting, maintainance and retrieval algorithms on indexes should be changed. Either one should have common indexes for all branches, or else indexes for the current branch only. In the former case, index retrieval algorithms should be changed to take into attention the splitting history. In the latter case, index maintainance algorithms should be changed. In either case, a significant amount of work on indexes is required upon backtracking.

In general, backtracking in resolution-based theorem provers requires a nontrivial implementation. In the modern saturation-based theorem provers for first-order logic, splitting with backtracking is implemented only in SPASS [Weidenbach *et al.*, 1999].

In this report we discuss an alternative to the explicit case analysis which can be implemented in resolution-based theorem provers without a significant overhead. We call it *splitting without backtracking*. The idea of splitting without backtracking can be formulated as follows. Consider again the split of Example 1.1, but let us mark the branches of the search tree by literals as follows. We take a new propositional symbol p and mark the left branch with p and the right branch with $\neg p$:



If there are several splits, we introduce a new propositional symbol for every split. Now, instead of dealing with branches, we simply add to each clause on a branch all labels used on that branch. Then the split can be described as the following inference rule which replaces a clause by two new clauses:

$$S \cup \{D[s], s = t \vee C\} \to S \cup \{D[s], s = t \vee p, C \vee \neg p\}.$$

It is not hard to argue that such a splitting preserves the unsatisfiability of the set of clauses (we will prove it formally below). The effect of splitting is very similar to the effect of the explicit case analysis, but branches are no more needed, since we are still dealing with just one set of clauses. The following simplification of $D[s]$ by $s = t$ on the left branch can now be described in at least two possible ways:

$$\begin{array}{l} S \cup \{D[s], s = t \vee p, C \vee \neg p\} \to \\ S \cup \{D[s], s = t \vee p, C \vee \neg p, D[t] \vee p\}; \end{array}$$

$$\begin{array}{l} S \cup \{D[s], s = t \vee p, C \vee \neg p\} \to \\ S \cup \{s = t \vee p, C \vee \neg p, D[t] \vee p, D[s] \vee \neg p\}. \end{array}$$

The second way corresponds to the simplification of Example 1.1, but we prefer the first way because it gives simpler clauses.

We will show that splitting without backtracking can simulate, in some sense, explicit case analysis. Moreover, by using various selection functions one can simulate a parallel version of case analysis, in which branches are exploited simultaneously. Apart from being easy to implement, splitting without backtracking has several other advantages: firstly,

we obtain the effect of intelligent backtracking for free, and secondly, some simplification rules can move clauses between the branches thus avoiding repeated work on different branches.

This article is organized as follows. In Section 2 we define some fundamental notions of this paper, for example those of *component* and *split*. We also recall some notions of the theory of resolution. In Section 3 we discuss several versions of the splitting rule, namely *binary* and *hyper* splitting, with or without *naming*. In Section 4 we show how one simulate different strategies of splitting by using appropriate literal selection functions. In particular, we show how one can simulate explicit case analysis by using so-called *blocking extension* of selection functions. An efficient algorithm for finding the maximal split of a clause into components is given in the full version of this paper. In Section 5 we discuss the *clause variance problem*: checking if a clause is a variant of another clause. We show that the clause variant problem is polynomial-time equivalent to the graph isomorphism problem. Finally, in Section 6 we discuss experiments carried out with VAMPIRE over a large collection of problems.

2 Preliminaries

We assume a fixed signature Σ. By $\mathbb{P}$ we denote a countable set of predicate symbols of arity 0 disjoint from Σ. Elements of $\mathbb{P}$ will be denoted by $p_1, p_2, \ldots$ and used as new names for clause components, or alternatively, as identifiers for branches.

We call a *clause* a set of literals. Sometimes clauses will be written as disjunction of their literals. We denote by $var(E)$ the set of all variables occurring in an expression E (e.g. clause). For a clause C, the *universal closure* of C, denoted by $\forall C$, is the formula $\forall x_1 \ldots \forall x_n C$, where $x_1, \ldots, x_n$ are all variables of C.

2.1 Components and splits

Definition 2.1 (component) *The clause C is called a* component *of a clause $C \vee D$ if C is nonempty, $var(C) \cap var(D) = \emptyset$ and C contains no predicate symbols in $\mathbb{P}$. The component C is called* minimal *if no proper subset of C is a component in $C \vee D$. The component C is called* trivial *if D is a clause of the signature $\mathbb{P}$ or empty.*

Note that every clause D has at least one component: the subset of D consisting of the literals of the signature Σ.

Definition 2.2 (split) *A* split *of a clause D is any representation of C as $C_1 \vee \ldots C_n \vee D'$, where $C_1, \ldots, C_n$ are different components of C. The split is* trivial *if $n = 1$ and C_1 is the trivial component of D. The split is* maximal *if $C_1, \ldots, C_n$ are all the minimal components of D.*

Note that in the maximal split the subclause D' only consists of the literals of the signature $\mathbb{P}$. Also, it is obvious that the maximal split of a clause is unique.

2.2 Resolution

Our technique for implementing splitting works in the context of a saturation procedure for the calculus of ordered resolution. For the purposes of this paper it is enough to recall only the nonground version of the ordered resolution rule itself, for other rules and calculi see e.g. [Bachmair and Ganzinger, 2001]. Let $\succ$ be a simplification order on terms, extended in the usual way on literals. We assume that a *selection function* select in every nonempty clause either a negative literal, or a subset of positive literals, and if any positive literal is selected, then all maximal w.r.t. $\succ$ literals are selected too. Then *ordered resolution* is the following rule:

$$\frac{A \vee C \quad \neg B \vee D}{(C \vee D)\theta},$$

where θ is a most general unifier of A and B, the atom A is selected in the clause $A \vee C$, the literal $\neg B$ is selected in the clause $\neg B \vee D$, and there is no literal in $D\theta$ greater than $A\theta$ in the used simplification ordering.

3 Splitting without backtracking

In this section we formulate several versions of splitting without backtracking and prove their soundness.

3.1 Binary splitting

Definition 3.1 (binary splitting rule) Let S be a set of clauses of the signature $\Sigma \cup \mathbb{P}$, $C \vee D$ be a clause with a nontrivial component C, and $p \in \mathbb{P}$ be an atom not occurring in $S \cup \{C \vee D\}$. Then the *binary splitting rule* is the following inference rule:

$$S \cup \{C \vee D\} \to S \cup \{C \vee p, D \vee \neg p\}. \qquad (1)$$

We also say that $S \cup \{C \vee p, D \vee \neg p\}$ is obtained from $S \cup \{C \vee D\}$ by *binary splitting*.

Theorem 3.2 *The binary splitting rule preserves satisfiability.*

PROOF. Any model for $S \cup \{C \vee p, D \vee \neg p\}$ is also a model for $S \cup \{C \vee D\}$ since $C \vee D$ is a logical consequence of $C \vee p$ and $D \vee \neg p$. In the reverse direction, a model M' for $S \cup \{C \vee p, D \vee \neg p\}$ can be obtained from a model M for $S \cup \{C \vee D\}$ by redefining p so that $M' \models p \leftrightarrow \neg \forall C$. □

It follows from the proof of Theorem 3.2 that the new predicate symbol p introduced by an application of the binary splitting rule (see (1)) can be regarded as a new name of the formula $\neg \forall C$. This implies an optimization of binary splitting which allows one to reuse the new names. This optimization can be formalized as follows.

Definition 3.3 (binary splitting with naming) A *naming function* is any function N from the set of nonempty clauses of the signature Σ to the set $\mathbb{P}$ with the following property: $N(C_1) = N(C_2)$ if and only if C_1 is a variant of C_2.

Let S be a set of clauses of the signature $\Sigma \cup \mathbb{P}$ and $C \vee D$ be a clause with a nontrivial component C. Let also N be a naming function. Then the *binary splitting rule with naming* is the following inference rule:

$$S \cup \{C \vee D\} \to S \cup \{C \vee N(C), D \vee \neg N(C)\}. \qquad (2)$$

One can also consider a modification of the binary splitting rule with naming where only the first application of the splitting introduces the clause $C \vee N(C)$, and all further applications of splitting with the same component C simply replace $C \vee D$ by *one* clause $D \vee \neg N(C)$.

Let us give an example that illustrates the difference between the binary splittings with and without naming. Suppose that we have a set of clauses $S \cup \{C \vee D_1, C \vee D_2\}$ such that C is a nontrivial component in both $C \vee D_1$ and $C \vee D_2$. Using binary splitting, we can perform the following derivation:

$$\begin{array}{l} S \cup \{C \vee D_1, C \vee D_2\} \rightarrow \\ S \cup \{C \vee p_1, D_1 \vee \neg p_1, C \vee D_2\} \rightarrow \\ S \cup \{C \vee p_1, D_1 \vee \neg p_1, C \vee p_2, D_2 \vee \neg p_2\}. \end{array}$$

Using binary splitting with naming, we can perform a different derivation (assuming $N(C) = p_1$).

$$\begin{array}{l} S \cup \{C \vee D_1, C \vee D_2\} \rightarrow \\ S \cup \{C \vee p_1, D_1 \vee \neg p_1, C \vee D_2\} \rightarrow \\ S \cup \{C \vee p_1, D_1 \vee \neg p_1, D_2 \vee \neg p_1\}. \end{array}$$

Preservation of satisfiability in the case of binary splitting with naming is more difficult to formulate. We can only guarantee preservation of satisfiability under some natural conditions.

Theorem 3.4 *Let I be an inference system on sets of clauses with the following properties: every inference preserves the set of models, i.e. for every inference $S \rightarrow S'$ in this system, S and S' have the same models. Let I' be the extension of I by binary splitting with naming. Then for every derivation $S_0 \rightarrow S_1 \rightarrow \ldots$ in I' such that S_0 is a set of clauses of the signature Σ and every $i \geq 0$, the set S_i is satisfiable if and only if so is S_{i+1}.*

PROOF. The proof essentially repeats the proof of Theorem 3.2. Take any model M of S_0 of the signature Σ and define its extension M' to the signature $\Sigma \cup \mathbb{P}$ as follows: define in M' each new predicate symbol $N(C) \in \mathbb{P}$ be equivalent to $\neg\forall C$. Then arguing as in the proof of Theorem 3.2 one can show that M' is a model of each S_i.

In the converse direction, note that every S_i is a logical consequence of S_{i+1}. Indeed, when S_{i+1} is obtained from S_i by applying an inference of the original system I, this holds by our assumption. When S_{i+1} is obtained from S_i by applying binary splitting with naming (2), this is also obvious, since S_{i+1} contains the clause $C \vee N(C)$. □ Note

that this proof can be easily adapted to the modified version of splitting. For the modified version in general it is not true any more that S_{i+1} contains the clause $C \vee N(C)$, but one can show that $C \vee N(C)$ is a logical consequence of S_{i+1}.

The conditions on the inference system I are natural and not restrictive. For example, the standard proofs of completeness in the theory of resolution [Bachmair and Ganzinger, 2001] use inference systems with two kinds of rule: addition of a clause implied by other clauses, and removal of a clause implied by other (smaller) clauses. It is easy to see that every such inference system preserves models.

3.2 Hyper splitting rule

Similar to the binary splitting, we can define hyper splitting, in which we split more than one component off a clause.

Definition 3.5 (hyper splitting rule) Let S be a set of clauses of the signature $\Sigma \cup \mathbb{P}$ and $C_1 \vee \ldots \vee C_n \vee D$ be a clause with nontrivial components $C_1, \ldots, C_n$. Let $p_1, \ldots, p_n$ be predicate symbols in $\mathbb{P}$ occurring in neither this clause nor S. Then the *hyper splitting rule* is the following inference rule:

$$\begin{array}{l} S \cup \{C_1 \vee \ldots \vee C_n \vee D\} \rightarrow \\ S \cup \{C_1 \vee p_1, \ldots, C_n \vee p_n, D \vee \neg p_1 \vee \ldots \vee \neg p_n\}. \end{array} \quad (3)$$

Let also N be a naming function. Then the *hyper splitting rule with naming* is the following inference rule:

$$\begin{array}{l} S \cup \{C_1 \vee \ldots \vee C_n \vee D\} \rightarrow \\ S \cup \{C_1 \vee N(C_1), \ldots, C_n \vee N(C_n), \\ \quad D \vee \neg N(C_1) \vee \ldots \vee \neg N(C_n)\}. \end{array} \quad (4)$$

Hyper splitting can be interpreted as repeatedly applied binary splitting. One can prove that hyper splitting (with or without naming) preserves satisfiability in exactly the same way as for binary splitting.

4 Literal selection and simulation of explicit case analysis

So far we only considered splitting as an inference rule. Here we will consider splitting as part of an inference system. As the inference system we consider binary resolution with superposition and negative selection. We assume a simplification ordering in which every literal of the signature Σ is greater than any atom in the signature $\mathbb{P}$. In order to define the inference system we have to define a selection function on clauses. We will define two kinds of selection functions: blocking and parallel. Both will be obtained by extending an arbitrary selection function on the clauses of the signature Σ to the clauses of the extended signature $\Sigma \cup \mathbb{P}$.

4.1 Blocking extension of a selection function

Definition 4.1 (blocking extension) Let s be any selection function on the clauses of the signature Σ. The *blocking extension* of s is the selection function on the clauses of the signature $\Sigma \cup \mathbb{P}$ defined as follows. (a) Let D be a clause containing a negative literal $\neg p$ of the signature $\mathbb{P}$, then the maximal one among such literals is selected in D. (b) Let D be a clause containing a positive literal p of the signature $\mathbb{P}$ and no negative literals in this signature. Then this literal is selected only when p is the maximal literal in D. Note that in this case D solely consists of positive literals of the signature $\mathbb{P}$.

It is not difficult to prove that we indeed obtain a selection function, since all literals of the signature $\mathbb{P}$ are less than all literals of the signature Σ. Since we will usually assume some selection function on the clauses of Σ, we will simply speak about the *blocking selection function* instead of the blocking extension of the selection function.

Now assume that we are using a blocking selection function. Let us also assume that each time we apply the splitting

rule, the newly introduced atom p is greater than any atom previously introduced by splitting. Let us show that the resulting inference system simulates, to some extent, explicit case analysis.

Consider a sequence of splits and applications of nonsimplifying inference rules. Let us call the *label* of a branch the set of all positive literals that mark edges of this branch. We transform clauses used in a derivation with splitting as follows. To each clause C that appears on a branch we add the label P of this branch, obtaining the clause $C \vee P$. We can show that each nonsimplifying inference that can be performed on a branch, can also be performed on the transformed clauses. For simplicity, we only consider the resolution rule.

Let $A_1 \vee D_1$ and $\neg A_2 \vee D_2$ be two clauses in which A_1 and $\neg A_2$ are selected, θ be a most general unifier of A_1 and A_2, and no literal in $D_1\theta$ is greater than $A_1\theta$. Then we can apply a resolution inference as follows.

$$\frac{A_1 \vee D_1 \quad \neg A_2 \vee D_2}{D_1\theta \vee D_2\theta} \ .$$

Let P_1 and P_2 be the labels of the branches of $A_1 \vee D_1$ and $\neg A_2 \vee D_2$ respectively. Then the corresponding transformed clauses will be $A_1 \vee D_1 \vee P_1$ and $\neg A_2 \vee D_2 \vee P_2$. By our definition of a blocking selection function and the order $\succ$ it is not hard to argue that A_1 and $\neg A_2$ will be selected in the transformed clauses and that no literal in $D_1\theta \vee P_1\theta$ is greater than $A_1\theta$. Therefore, we can apply the rule

$$\frac{A_1 \vee D_1 \vee P_1 \quad \neg A_2 \vee D_2 \vee P_2}{D_1\theta \vee D_2\theta \vee P_1 \vee P_2} \ .$$

It is not hard to argue that $P_1 \vee P_2$ is the label of the common branch for $A_1 \vee D_1$ and $\neg A_2 \vee D_2$.

It is interesting to observe what happens to the literals which do *not* belong to the currently investigated branch. Suppose, for, simplicity, that we are working with the leftmost branch labelled by $p_1 \vee \ldots \vee p_n$. Then all clauses which are not on the current branch contain a negative literal of the signature $\mathbb{P}$, which by our choice of the selection function, will be selected. Take any such clause, for example, $\neg p_n \vee D$. The only inference rule we can apply to this clause is resolution with a clause $p_n \vee P$, where p_n is greater than any literal of P. But we can obtain such a clause only when the empty clause is obtained on the leftmost branch labelled by $p_n \vee P$, so $\neg p_n \vee D$ will be blocked for inferences until a contradiction on the leftmost branch is obtained. As soon as such a contradiction is obtained, we can "unblock" the clause $\neg p_n \vee D$ by resolving it against $p_n \vee P$ and obtaining $D \vee P$.

Thus, we obtain a nearly complete correspondence between search states of the explicit case analysis and the system with the binary splitting without backtracking. It is not an exact correspondence for several reasons. Firstly, simplification rules behave in a slightly different way. Secondly, splits done by splitting without backtracking often do not belong to particular branches, but are "shared" across different branches. To explain this effect, we show in Figure 2 a derivation using binary splitting and a blocking selection function which corresponds to the derivation of Figure 1. We preserve the same enumeration of inferences as in that figure. If a literal in the new signature $\mathbb{P}$ is selected, we put it in front of the clause, for example $\neg p_1 \vee B$ means that $\neg p_1$ is selected. For simplicity, we remove subsumed clauses and pure literals.

After inference 7 of the derivation using explicit case analysis, we have a copy of the clause $\neg A \vee \neg B$ which was put on the right branch, after we performed a split on this clause on the left branch. In the derivation using binary splitting, the split of $\neg A \vee \neg B$ into $\neg A \vee p_2$ and $\neg p_2 \vee \neg B$ remains even when we finish working on the left branch.

4.2 Parallel extension of a selection function

Definition 4.2 (parallel extension) Let s be any selection function on the clauses of the signature Σ. The *parallel extension* of s is a selection function s' on the clauses of the signature $\Sigma \cup \mathbb{P}$ defined as follows. Let D be a clause of the form $D' \vee P$, where D' is a clause of the signature Σ and P is a clause of the signature $\mathbb{P}$. If D' is not empty, then s' selects in $D' \vee P$ exactly the same literals as s selects in D'. When D' is empty, the maximal literal from P is selected.

In other words, the parallel extension always makes the selection in a clause D ignoring the literals of D' of the signature $\mathbb{P}$.

Again, it is not difficult to prove that we indeed obtain a selection function, since all literals of the signature $\mathbb{P}$ are less than all literals of the signature Σ. We will simply speak about a *parallel selection function* instead of the parallel extension of a selection function.

A parallel selection function ignores the splitting history of clause completely, and thus gives the effect of parallel search on all branches.

4.3 Subsumption resolution and splitting

One of the serious advantages of splitting without backtracking is that it can be productively combined with the simplification rule called *subsumption resolution* [Bachmair and Ganzinger, 2001]. Essentially, subsumption resolution is a special case of binary resolution that guarantees that the resolvent subsumes one of the parents. We say that a clause D_1 *subsumes* a clause D_2 if there exists a substitution σ such that $D_1\sigma \subseteq D_2$. It is known that clauses subsumed by other clauses can be removed from the search space.

Subsumption resolution is the following inference rule on sets of clauses:

$$\begin{array}{l} S \cup \{A_1 \vee D_1, \neg A_2 \vee D_2\} \rightarrow S \cup \{A_1 \vee D_1, D_2\} \\ S \cup \{\neg A_1 \vee D_1, A_2 \vee D_2\} \rightarrow S \cup \{\neg A_1 \vee D_1, D_2\} \end{array}$$

where there exists a substitution σ such that $A_1\sigma = A_2$ and $D_1\sigma \subseteq D_2$. The rule is called subsumption resolution because of its resemblance to subsumption: this rule is applicable if and only if $A_1 \vee D_1$ subsumes $A_2 \vee D_2$. Thus subsumption algorithms can be modified for finding pairs of clauses to which subsumption resolution is applicable.

Since subsumption resolution is a simplification rule (the clause $\neg A_2 \vee D_2$ or $A_2 \vee D_2$ is replaced by a smaller clause D_2), it can be applied independently of the selection function or order on literals.

Subsumption resolution combined with splitting can give the following effect: literals can be moved across different branches. For example, consider two clauses $C \vee p$ and $D \vee$

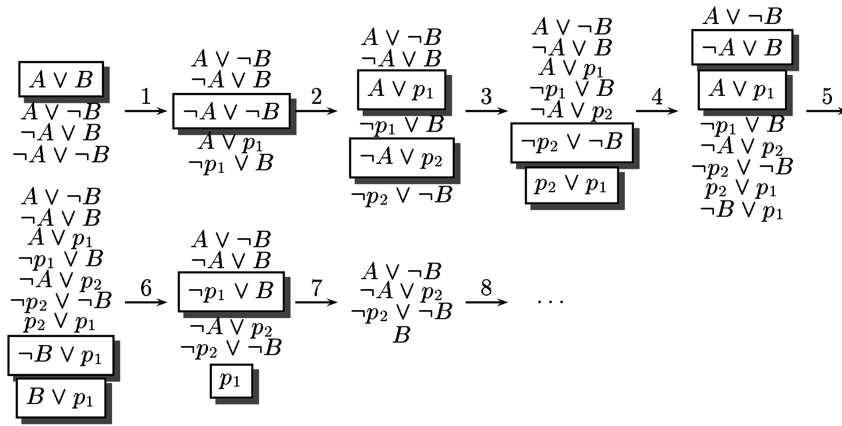


Figure 2: A derivation using binary splitting and blocking selection

$\neg p$ such that C subsumes D. In explicit case analysis, these clauses would belong to two different branches, because $C \vee p$ should be on the left branch of splitting on p, while $D \vee \neg p$ on the right branch. If we use subsumption resolution, we can replace $D \vee \neg p$ by the clause D, which corresponds to moving D from the right branch to the top.

The effect of subsumption resolution is even stronger when C is a variant of D or coincides with D. Then we obtain two clauses $C \vee p$ and $C \vee \neg p$, which by subsumption resolution followed by subsumption will be replaced by C. This has the effect of noting that C belongs to two different branches, and putting it on top and can save a lot of resources compared to explicit case analysis. Splitting with backtracking would repeat inferences with C on different branches.

Let us show that subsumption resolution and splitting can, in some cases, give the effect of condensing a clause. Condensing is defined in [Joyner, Jr., 1976]. A clause D' is said to be obtained from a clause D by *condensing* if D' is a proper subset of D and D subsumes D'.

Example 4.3 Let $C[\bar{x}]$ be a clause with variables in $\bar{x}$ and $\bar{y}$ be a sequence of variables disjoint from $\bar{x}$. Then $C[\bar{y}]$ is a variant of $C[\bar{x}]$. Consider the clause $C[\bar{x}] \vee C[\bar{y}]$. This clause can be condensed into $C[\bar{x}]$. Binary splitting will replace this clause by two clauses $C[\bar{x}] \vee p$ and $C[\bar{y}] \vee \neg p$. Application of subsumption resolution to these clauses yields the clause $C[\bar{x}]$ which subsumes each of these clauses. Thus, binary splitting and subsumption resolution in this example give the effect of condensing.

5 Complexity of splitting with naming

To implement splitting with naming, it is required to check whether a component is a variant of another component. In this section we prove that the complexity of checking whether one clause is a variant of another one is polynomial-time equivalent to the graph isomorphism problem.

Definition 5.1 (clause variance problem) *Clause variance* is the following decision problem. An instance of this problem is a pair of clauses (C_1, C_2). The answer is "yes" if C_1 is a variant of C_2.

Theorem 5.2 *The clause variance problem is polynomial-time equivalent to the graph isomorphism problem.*

The proof is given in the full version of this paper.

More generally, in practice we have to check whether a given component C is a variant of a component in a set of components S. In VAMPIRE we implement this *many-to-one clause variance* check using the code tree indexing described in [Voronkov, 1995; Riazanov and Voronkov, 2000b]. Code trees are used in the implementation of many-to-one forward subsumption, it is not hard to modify them to implement the many-to-one clause variance.

6 Experimental results

We implemented several versions splitting with backtracking in the new version of VAMPIRE [Riazanov and Voronkov, 2001]. In this section we present experimental results over 4658 nonunit clause form problems. Of these problems, 2820 problems come from the TPTP library [Sutcliffe and Suttner, 1998] and 1838 problems from the experiments in list software reuse described in [Schumann and Fischer, 1997]. Of the 4658 problems 3300 contain equality. For problems with equality, in addition to splitting without backtracking we implemented the *branch rewriting* rule which replaces a clause $C[s\sigma] \vee P$ by a clause $C[t\sigma] \vee P$, if a clause $s = t \vee P'$ with $P' \subseteq P$ is present. The effect of this simplification rule is similar to the effect of simplification by unit equalities on a splitting branch in an explicit case analysis procedure. To support our viewpoint that we describe a "cheap" implementation of rewriting, we note that branch rewriting was implemented in less than 3 hours.

We present the results for problems with and without equality separately. For all problems we used the same algorithm based on the limited resource strategy [Riazanov and Voronkov, 2000a] and the same selection function. This algorithm and selection function were chosen because they experimentally proved to be superior to other algorithms and selection functions when splitting was not used. When running VAMPIRE with splitting, we used the following parameters:

1. Hyper splitting with a Parallel selection function (P) vs. binary splitting with no selection.

no	166:11	220:7	227:12	256:6	280:4	300:1	359:3	388:3
PNR		94:36	75:15	125:30	130:9	153:9	215:14	241:11
NR			71:69	43:6	121:58	127:41	147:4	175:3
PN				95:60	111:50	104:20	180:39	198:28
N					98:72	101:52	115:9	143:8
PR						68:45	118:38	144:35
P							110:53	130:44
R								37:8
	PNR	NR	PN	N	PR	P	R	

Figure 3: Results on problems with equality

PN	65:3	74:11	76:0	80:1
N		14:13	16:2	18:1
no			19:6	21:5
P				10:7
	N	no	P	

Figure 4: Results on problems without equality

2. Naming (N) vs. no naming.
3. Branch Rewriting (R) vs. no branch rewriting.

We will use combinations of the three letters P, N, R, to denote different strategies used in the experiments. For example, PR means that the Parallel version with branch Rewriting but no naming was used. We use "no" to denote the strategy that uses no splitting at all, and $-$ the strategy that uses splitting without P,R, and N, i.e. simple binary splitting with a blocking selection function.

We used a Linux-running PC with 256M RAM and a 400MHz Intel III processor. For all problems we used the time limit of 1 minute. The chosen strategy without splitting solves 3091 problems. The total number of problems solved with or without splitting is 3189. So with splitting, VAMPIRE could solve 98 problems that could not be solved without splitting. Of these 98 problems, 77 are without equality and 21 with equality. Figures 3 and 4 show the comparative behavior of various strategies depending on the settings of the parameters P, N, R on problems with and without equality, respectively. The pair $n_1 : n_2$ in a row labelled by a strategy S_1 and the column labelled by a strategy S_2 means that there were n_1 problems solved by S_1 but not solved by S_2 and n_2 problems solved by S_2 but not solved by S_1. For example NR (Naming + branch Rewriting) could solve 71 problems with equality not solved by PN (Parallel selection + Naming), while PN could solve 69 problems with equality not solved by NR.

The following conclusion can be derived from these results:

1. On problems with equality, VAMPIRE with any version of splitting is on the average weaker than VAMPIRE without splitting, while on problems without equality it is on the average stronger.
2. Each of the following settings: naming, parallel splitting, and branch rewriting improve the results, by far the strongest one is naming.

We believe that the performance of splitting without backtracking can be considerably improved, but optimized implementation, more experiments, and insights in the behavior of splitting are required.

References

[Bachmair and Ganzinger, 2001] L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. Elsevier Science, 2001.

[Fermüller *et al.*, 2001] C. Fermüller, A. Leitsch, U. Hustadt, and T. Tammet. Resolution decision procedures. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 25, pages 1791–1849. Elsevier Science, 2001.

[Joyner, Jr., 1976] William H. Joyner, Jr. Resolution strategies as decision procedures. *Journal of the ACM*, 23(3):398–417, July 1976.

[Riazanov and Voronkov, 2000a] A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. Preprint CSPP-7, Department of Computer Science, University of Manchester, October 2000.

[Riazanov and Voronkov, 2000b] A. Riazanov and A. Voronkov. Partially adaptive code trees. In M. Ojeda-Aciego, I.P. de Guzmán, G. Brewka, and L.M. Pereira, editors, *Logics in Artificial Intelligence. European Workshop, JELIA 2000*, volume 1919 of *Lecture Notes in Artificial Intelligence*, pages 209–223, Málaga, Spain, 2000. Springer Verlag.

[Riazanov and Voronkov, 2001] A. Riazanov and A. Voronkov. Vampire 1.1 (system description). Siena, Italy, June 2001. Accepted to IJCAR 2001.

[Schumann and Fischer, 1997] J. Schumann and B. Fischer. NORA/HAMMR: Making deduction-based software component retrieval practical. In *Proc. Automated Software Engineering (ASE-97)*, pages 246–254, Lake Tahoe, November 1997. IEEE Computer Society Press.

[Sutcliffe and Suttner, 1998] G. Sutcliffe and C. Suttner. The TPTP problem library — CNF release v. 1.2.1. *Journal of Automated Reasoning*, 21(2), 1998.

[Voronkov, 1995] A. Voronkov. The anatomy of Vampire: Implementing bottom-up procedures with code trees. *Journal of Automated Reasoning*, 15(2):237–265, 1995.

[Weidenbach *et al.*, 1999] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Topic. System description: SPASS version 1.0.0. In H. Ganzinger, editor, *Automated Deduction—CADE-16. 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 378–382, Trento, Italy, July 1999.

UNSEARCHMO: Eliminating Redundant Search Space on Backtracking for Forward Chaining Theorem Proving*

Lifeng He
Faculty of Information Science and Technology
Aichi Prefectural University, Aichi, 480-1198 Japan

Abstract

This paper introduces how to eliminate redundant search space for forward chaining theorem proving as much as possible. We consider how to keep on minimal useful consequent atom sets for necessary branches in a proof tree. In the most cases, an unnecessary non-Horn clause used for forward chaining will be split only once. The increase of the search space by invoking unnecessary forward chaining clauses will be nearly linear, not exponential anymore. In a certain sense, we "unsearch" more than necessary. We explain the principle of our method, and provide an example to show that our approach is powerful for forward chaining theorem proving.

1 Introduction

Automated reasoning is one of the most important topics for artificial intelligence and computer science. Among others, theorem proving technologies for first-order predicate calculus have been attracting much interesting [Robinson, 1965; Loveland, 1968; Manthey and Bry, 1988].

It is well-known that if $\mathcal{S}'$ is an unsatisfiable subset of a clause set $\mathcal{S}$, it is generally easier to show $\mathcal{S}'$ unsatisfiable than to show $\mathcal{S}$ unsatisfiable. Obviously, the smaller such a $\mathcal{S}'$ is, the shorter a proof is. If forward chaining is used for reasoning about non-Horn clauses, it means that we need not consider every violated clause, but only those that can help us to find a refutation. It is obvious that invoking unnecessary non-Horn clauses will explode the search space exponentially.

SATCHMO (SATisfiability CHecking by MOdel generation) [Manthey and Bry, 1988] is potentially inefficient, since it uses all violated clauses to forward chaining. Addressing to this problem, two strategies have been developed. One is utilizing an intelligent strategy to only select those relevant-like non-Horn clauses for forward chaining, which is proposed in [Ramsay, 1991], refined in [Loveland *et al.*, 1995] (SATCHMORE) and further improved in [He *et al.*, 1998] ($\mathcal{A}$-SATCHMORE). The other method, proposed in [He, 2001] (called I-SATCHMO) is eliminating redundant searching space after unnecessary non-Horn clauses have been used for forward chaining by intelligent backtracking.

As indicated in [He, 2001], there are mainly two problems about the approach to select a suitable non-Horn clause for forward chaining. One is that it takes much overhead cost to decide whether a non-Horn clause is suitable or not. In general, the stricter the checking is, the more expensive the overhead cost is. The other is that any such strategy is only effective in a limited range and certainly fails in some more complicated cases. In fact, a non-Horn clause recognized as "relevant" by some advanced selecting strategy might be practically unnecessary to the refutation being made. In other words, there are always such irrelevant non-Horn clauses that can pass any relevancy test. These problems have been illustrated on some examples in [He, 2001].

The principle of the method of eliminating redundant searching space by intelligent backtracking is very simple: If one of the consequent atoms of a forward chaining clause used for forward chaining is found to be useless to the reasoning on backtracking, the use of this clause is known as unnecessary, and the remaining processing over the clause's consequence is immediately abandoned. It only takes a little overhead cost to find irrelevant non-Horn clauses and is compatible with the former strategy introduced above.

However, marking those consequent atoms contributed to derive antecendents of irrelevant forward chaining clauses as useful and mixing up the useful consequent atoms for different branches in a proof tree, I-SATCHMO might make redudant search.

This paper introduces a solution for these problems. As we will see, in our improved method, for each node in the proof tree, only those indispensable consequent atoms to derive the refutation at the node are marked as useful. Thus, our method can show a refutation for a given unsatisfiable clause set without UNSEARCHing MOre than necessary (hence the acronym UNSEARCHMO). In other words, our method is one of the most efficient strategies for forward chaining theorem proving.

2 SATCHMO and I-SATCHMO

We assume familiarity with the details relating to SATCHMO [Manthey and Bry, 1988], SATCHMORE [Loveland *et al.*,

*This work is partially supported by the Japanese Ministry of Education, Science, Sports and Culture and the Artificial Intelligence Research Promotion Foundation of Japan.

1995] as well as $\mathcal{A}$-SATCHMORE and limit ourselves to briefly reviewing the basic material needed for our presentation.

In this paper, a problem for checking unsatisfiability is represented by means of positive implicational clauses, each of which has of the form $A_1, \ldots, A_n \rightarrow C_1; \ldots; C_m$ $(n, m \geq 0)$. We refer to the implicit conjunction $A_1, \ldots, A_n$ as the antecedent of the clause, with each A_i being an antecedent atom. The implicit disjunction $C_1; \ldots; C_m$ is referred to as the consequent of the clause, and each C_j is a consequent atom. A clause with an empty consequent, i.e., $m = 0$, is called a *negative clause* and is written as $A_1, \ldots, A_n \rightarrow \bot$, where $\bot$ means *false*. A clause with an empty antecedent, i.e., $n = 0$, is written with consequent part only if it is a fact (Horn clause), and otherwise the antecedent atom *true* is added. Moreover, a given clause set is divided into two subsets. One is $\mathcal{BC}$, the Backward Chaining component that is a decidable Horn clause set consisting of all negative clauses, facts and any Horn clauses selected by user. The other is $\mathcal{FC}$, the Forward Chaining component that contains all the remaining clauses.

If goal $\mathcal{G}$ (a conjunction of atoms) can be proven to logically follow the clause set $\mathcal{S}$, we denote that with $\mathcal{S} \vdash \mathcal{G}$. On the other hand, if $\mathcal{I}$ is a ground atom set and A a ground atom, $A \in \mathcal{I}$ indicates that atom A is a member of $\mathcal{I}$. Because we utilizes forward chaining on non-Horn clauses same as SATCHMO, all of clauses must hold the so-called *range-restricted property*[1] to guarantee the soundness for refutation.

Suppose that $\mathcal{BC}$ be a decidable Horn clause set and $\mathcal{I}$ a ground atom set, then a conjunction (disjunction) of ground atoms is *satisfiable* in $\mathcal{BC} \cup \mathcal{I}$ if all (some) of its members can be derived from $\mathcal{BC} \cup \mathcal{I}$, and else *unsatisfiable*. A ground clause $\mathcal{A} \rightarrow \mathcal{C}$ is said *satisfiable* in $\mathcal{BC} \cup \mathcal{I}$ if $\mathcal{C}$ is satisfiable or $\mathcal{A}$ is unsatisfiable in $\mathcal{BC} \cup \mathcal{I}$, and else *violated*.

It is well-known that a clause set $\mathcal{S}$ is satisfiable if and only if we can find a model $\mathcal{M}$ for $\mathcal{S}$. If $\mathcal{M}$ is a model of $\mathcal{S}$, then each clause of $\mathcal{S}$ is satisfiable in $\mathcal{M}$. Based on such consideration, to check whether a given clause set $\mathcal{S} = \mathcal{BC} \cup \mathcal{FC}$ is satisfiable, SATCHMO goes to construct a model for this clause set by trying to satisfy all clauses in $\mathcal{S}$.

The reasoning made by SATCHMO to search a model for a given clause set $\mathcal{BC} \cup \mathcal{FC}$ can be graphically illustrated in a *Proof Tree* (abbreviated to *PT* hereafter) described as follows.

For the current node D (initially the root node $\{\mathcal{BC}\}$), and the ground consequent atom set $\mathcal{I}_D$ (initially empty):

1. If $\mathcal{BC} \cup \mathcal{I}_D \vdash \bot$, create a leaf node $\bot$ below node D. The process for the branch terminates.
2. If $\mathcal{BC} \cup \mathcal{I}_D \not\vdash \bot$, select an instantiated ground violated clause $A_1, \ldots, A_n \rightarrow C_1; \ldots; C_m$ from $\mathcal{FC}$. If no such clause exists, the process terminates to report that $\mathcal{BC} \cup \mathcal{FC}$ is satisfiable.
3. For each a consequent atom C_i of the selected violated $\mathcal{FC}$ clause, create a child node C_i below the current node. Taking node C_i as new current node, call this procedure recursively in depth-first strategy with the augmented ground consequent atom set $\mathcal{I}_{C_i}$, where $\mathcal{I}_{C_i} = \mathcal{I}_D \cup \{C_i\}$.

[1] A clause is range-restricted iff every variable in its consequent also occurs in its antecedent. An important property of such clauses is that if its antecedent is satisfied by a set of ground atoms, then every consequent atom is certainly ground. As indicated by Manthey and Bry (1988) and Bry and Yahya (2000), any first-order clause set can be transformed into the same "meaning" range-restricted one.

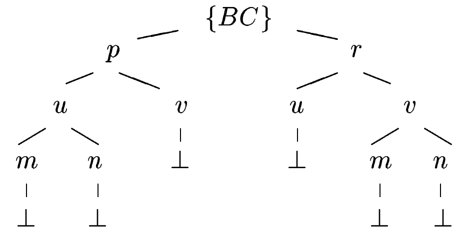


Figure 1: The proof tree PT constructed by SATCHMO

For node D, where the ground consequent atom set is $\mathcal{I}_D$, if all branches terminate in a leaf node $\bot$, it means that satisfying atom D there will lead to a refutation anyway. In other words, $\mathcal{BC} \cup \mathcal{I}_D$ cannot be extended as a model of $\mathcal{BC} \cup \mathcal{FC}$. Such node is said to be *unsatisfiable*. When the node is the root node $\{\mathcal{BC}\}$, then $\mathcal{BC} \cup \mathcal{FC}$ is unsatisfiable. Corresponding to creating node D in PV, an atom D is asserted into the reasoning database. On the other hand, when node D is proven to be unsatisfiable, the atom D will be retracted from the database. Moreover, whenever a node is proven unsatisfiable, the reasoning process will backtrack to its parent node if any.

Example 2.1 Consider the following propositional clause set $\mathcal{S} = \mathcal{BC} \cup \mathcal{FC}$.

$$\begin{array}{llll} \mathcal{BC}: & r, u \rightarrow \bot. & p, v \rightarrow \bot. & m \rightarrow \bot. \\ & n \rightarrow \bot. & v, s \rightarrow \bot. & u, t \rightarrow \bot. \end{array}$$

$$\begin{array}{llll} \mathcal{FC}: & true \rightarrow p;\ r. & true \rightarrow u;\ v. & w \rightarrow s;\ t. \\ & true \rightarrow m;\ n. & & \end{array}$$

The proof tree generated by SATCHMO is shown in Figure 1. It is obvious that SATCHMO made some redundant search. In fact, a refutation can be derived from the subset consisting of $\mathcal{BC}$ and the only $\mathcal{FC}$ clause $true \rightarrow m;\ n$.

By the way, for the given clause set in this example, every violated $\mathcal{FC}$ clause used by SATCHMO is recognized as "relevant" by either SATCHMORE or $\mathcal{A}$-SATCHMORE, the proof tree generated by SATCHMORE or $\mathcal{A}$-SATCHMORE is the same as that shown in Figure 1.

The redundant search space can be cut down by intelligent backtracking presented in I-SATCHMO. As we have seen, a consequent atom D is retracted from the reasoning database only if the node D in the proof tree has been proven as unsatisfiable. Suppose that $\mathcal{A} \rightarrow C_1; \ldots; C_m$ be the violated $\mathcal{FC}$ clause used for forward chaining at node D in the proof tree, then, node D holds m child nodes $C_1, \ldots, C_m$. If some consequent atom C_i is found to be useless to the reasoning at the time being retracted from the database (hence node C_i has been proven as unsatisfiable), i.e., it is useless to construct the partial proof tree below node C_i, the same partial proof tree can be directly constructed below node D, and therefore,

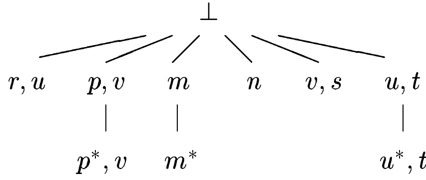


Figure 2: The deriving tree for goal $\perp$ at node m

at this point, we know that node D can be proven as unsatisfiable just in the same way made at node C_i. Since D has been known as *unsatisfiable*, the remaining process for showing node D unsatisfiable, i.e., the process on the consequent atoms $C_{i+1}, \ldots, C_m$ (if any), can be removed at once. On the other hand, the use of an $\mathcal{FC}$ clause $\mathcal{A} \rightarrow C_1; \ldots; C_m$ is *necessary* only if each C_i ($1 \leq C_i \leq m$) is useful to constructing the partial tree below node C_i.

Now we introduce I-SATCHMO how to decide what consequent atom is *useful* for the reasoning. Obviously, a consequent atom asserted can only be used to derive goal $\perp$ or antecedents of violated $\mathcal{FC}$ clauses. Therefore,I-SATCHMO marks an asserted consequent atom as useful whenever it contributed to derive goal $\perp$ or an antecedent of a violated $\mathcal{FC}$ clause.

The useful consequent atoms to derive goal $\mathcal{G}$ can be found from the deriving tree of goal $\mathcal{G}$ [He, 2001].

Definition 2.1 Suppose that a give clause set be $\mathcal{S} = \mathcal{BC} \cup \mathcal{FC}, \mathcal{I}$ the current consequent atom set established by forward chaining, $\mathcal{G}$ a goal such that $\mathcal{BC} \cup \mathcal{I} \vdash \mathcal{G}$. The deriving tree of goal $\mathcal{G}$ with respect to $\mathcal{BC} \cup \mathcal{I}$, denoted by $DT_{\mathcal{G}}^{\mathcal{BC} \cup \mathcal{I}}$, is the result tree constructed as follows.

1. The root node is $\mathcal{G}$.
2. Suppose that $A_1, A_2, \ldots, A_u, C_1^*, \ldots, C_j^*$ is a node in the tree being constructed.
   - (a) For each $\mathcal{BC}$ clause $D_1, \ldots, D_t \rightarrow H$ such that $H\sigma = A_1\sigma$, where σ is a most general unifier of H and A_1, create a child node $D_1\sigma, \ldots, D_t\sigma, A_2\sigma, \ldots, A_u\sigma, C_1^*, \ldots, C_j^*$.
   - (b) For each atom $C \in \mathcal{I}$ such that $C = A_1\theta$, where θ is a ground substitution (since C is a ground atom), create a child node $A_2\theta, \ldots, A_u\theta,\ C_1^*, \ldots, C_j^*, C_{j+1}^*$, where $C_{j+1} = C$.
3. Repeat this procedure recursively until an empty leaf or a leaf such that all of its atoms hold "$*$" is derived. Such a leaf is called *successful deriving leaf.*

Since $\mathcal{BC}$ is decidable and $\mathcal{BC} \cup \mathcal{I} \vdash \mathcal{G}$, $DT_{\mathcal{G}}^{\mathcal{BC} \cup \mathcal{I}}$ can be finally constructed with finding a successful deriving leaf. The consequent atoms that contributed to derive $\mathcal{G}$ from $\mathcal{BC} \cup \mathcal{I}$ are those atoms listed in the successful deriving leaf if any.

Example 2.2 Suppose that $\mathcal{S} = \mathcal{BC} \cup \mathcal{FC}$ be the clause set given in Example 2.1. Consider the leftmost node m in Figure 1, where $\mathcal{I}_m = \{p, u, m\}$. The deriving tree for goal $\perp$, $DT_{\perp}^{\mathcal{BC} \cup \mathcal{I}_m}$, is shown in Figure 2. To derive goal $\perp$, only m is useful.

The algorithm of I-SATCHMO to construct *PT* for checking whether a given clause set $\mathcal{S} = \mathcal{BC} \cup \mathcal{FC}$ is unsatisfiable can be described as follows.

For the current node D (initially the root node $\{\mathcal{BC}\}$) and the corresponding consequent atom set $\mathcal{I}_D$ (initially empty):

1. If $\mathcal{BC} \cup \mathcal{I}_D \vdash \perp$, create a leaf node $\perp$ under the current node, which means that node D is proven unsatisfiable. Mark all consequent atoms listed in the successful deriving node in $DT_{\perp}^{\mathcal{BC} \cup \mathcal{I}_D}$ as useful.
2. If $\mathcal{BC} \cup \mathcal{I}_D \not\vdash \perp$, select an instantiated ground violated clause from $\mathcal{FC}$. If no such clause exists, the process terminates to report that $\mathcal{BC} \cup \mathcal{FC}$ is not unsatisfiable.
3. For the selected violated non-Horn clause $\mathcal{A} \rightarrow C_1; \ldots;$ C_m, mark all consequent atoms listed in the successful deriving node in $DT_{\mathcal{A}}^{\mathcal{BC} \cup \mathcal{I}_D}$ as useful. Then, create a node C_i below the current node for each i such that $1 \leq i \leq m$. Taking C_i as the new current node, call this procedure recursively in depth-first strategy with the augmented consequent atom set $\mathcal{I}_{C_i} = \mathcal{I}_D \cup \{C_i\}$. However, if C_i has been found to be useless on backtracking, instead of each node C_j such that $i < j \leq m$, a special leaf node $\times$ is created, where $\times$ means that the process from there is pruned away. On the other hand, if all C_i are marked as useful, the use of $\mathcal{A} \rightarrow C_1;\ \ldots;\ C_m$ is known as necessary. In either case, node D is proven as unsatisfiable.

Example 2.3 Suppose that $\mathcal{S} = \mathcal{BC} \cup \mathcal{FC}$ be the clause set given in in Example 2.1.

In the branch $\{\mathcal{BC}\} \rightarrow p \rightarrow u \rightarrow m \rightarrow \perp$ in Figure 1, only atom m is used in forward chaining and marked as useful. In the same way, in the branch $\{\mathcal{BC}\} \rightarrow p \rightarrow u \rightarrow n \rightarrow \perp$, only atom n is used in forward chaining and marked as useful.

Since both m and n are proven to be unsatisfiable, its parent node, i.e., u is proven to be unsatisfiable. We backtrack to node u to retract u. At that time, we know u is not marked as useful in the reasoning. Therefore, it is useless for the reasoning and the use of the violated $\mathcal{FC}$ clause $true \rightarrow u; v$ selected at node p is unnecessary. I-SATCHMO prunes away the process to be made on the remaining consequent atom v and backtracks to node p at once.

Now, we know that p is also not marked as useful in the reasoning, therefore the violated $\mathcal{FC}$ clause $true \rightarrow p; r$ selected at the root node $\{\mathcal{BC}\}$ is unnecessary. Accordingly, we stop to process the remaining consequent atom r.

In this way, the proof tree constructed by I-SATCHMO is shown in Figure 3. As we see, unnecessary $\mathcal{FC}$ clauses are split only once.

3 UNSEARCHMO

Since I-SATCHMO marks an asserted consequent atom as useful whenever it contributes to derive thea antecendent of a violated $\mathcal{FC}$ clause, those consequent atoms contributed to unnecessary violated clauses are also marked as useful.

This problem can be easily solved by improving the marking work as follows: an asserted consequent atom is marked as useful only when it contributes to derive goal $\perp$ or the

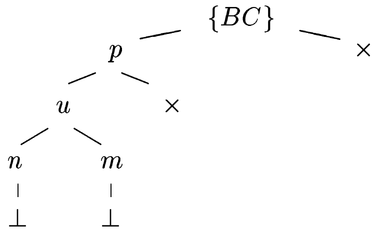


Figure 3: The proof tree PT constructed by I-SATCHMO in Example 2.3

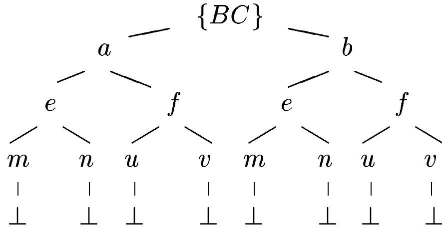


Figure 4: The proof tree PT constructed by I-SATCHMO for the clause set given in Example 3.1

antecedent of a violated $\mathcal{FC}$ clause that has been verified as necessary.

Moreover, without distinguishing the useful consequent atoms for different branches in proof tree, I-SATCHMO might mismark those consequent atoms that are useful for the refutation of unnecessary branches in a proof tree but useless for the necessary branches as "useful". In such cases, I-SATCHMO also possibly makes some redundant search. Let us see the following example.

Example 3.1 Consider the following propositional clause set $\mathcal{S} = \mathcal{BC} \cup \mathcal{FC}$.

$BC: \quad m \to \bot. \quad n \to \bot. \quad u \to \bot. \quad v \to \bot.$

$FC: \quad true \to a;\ b. \quad true \to e;\ f. \quad a, e \to m;\ n.$
$\qquad b, e \to m;\ n. \quad true \to u;\ v.$

The proof tree generated by I-SATCHMO is shown in Figure 4.

Consider the leftmost node labeled with e, where $\mathcal{I}_e = \{a, e\}$, the clause $a, e \to m;\ n$ becomes violated and is used to forward chaining. Since both m and n are useful for deriving $\bot$, the use of the clause is found to be necessary to the refutation, a and e are marked as useful. In this way, the process for the consequent atom f (the sibling atom of e) and b (the sibling atom of a) are made, respectively.

Obviously, the process for node b in the above example is unnecessary. Although the consequent atom a is useful for each branch from node e, it is useless for any branch from node f. It means that the refutation at node f can be made without the consequent atom a, that is, that can be made on the root node $\{\mathcal{BC}\}$ in the same way. Therefore, the redundant process for the consequent atom b can be eliminated.

Although the process made at the leftmost node e is also unnecessary in a strict meaning, it is unavoidable unless we select atom f first to process at node a or we first select the last non-Horn clause for forward chaining. It is the well-known nondetermination problem for theorem proving.

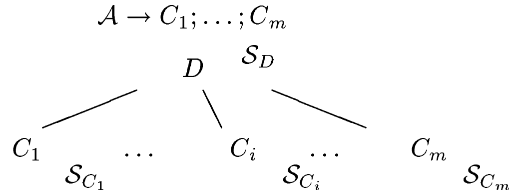


Figure 5: Finding the useful consequent atom set for parent node from those of its children nodes

Now, we consider how to distinguish and manage the useful consequent atoms for different branches in a proof tree. Without the loss of generality, shown as in Figure 5, suppose that $\mathcal{A} \to C_1; \ldots; C_m$ be the violated clause selected at node D and $\mathcal{S}_{C_i}$ be the useful consequent atom set relative to C_i for all i such that $1 \leq i \leq m$. We consider how to derive the useful consequent atom set $\mathcal{S}_D$ relative to node D.

Since our reasoning procedure is depth-first one, C_{i+1} is not taken into processing until node C_i has been proven to be unsatisfiable. Moreover, only when some of C_i is found to be useless or all node C_i $(1 \leq i \leq m)$ are proven to be unsatisfiable, their parent node D is proven to be unsatisfiable.

As we will know from the algorithm of UNSEARCHMO, when node C_i is proven to be unsatisfiable, the useful consequent atom set $\mathcal{S}_{C_i}$ for deriving the refutation at node C_i has already been established. The useful consequent atom set $\mathcal{S}_D$ for node D is constructed according to the way how node D is proven as unsatisfiable.

If node D is proven as unsatisfiable in the way that all of C_i $(1 \leq i \leq m)$ are proven as unsatisfiable, then the use of the clause $\mathcal{A} \to C_1; \ldots; C_m$ is found to be necessary. In order to complete the refutation at node D, the clause should be ably found to be violated there. Therefore, the useful consequent atoms to derive the antecedent $\mathcal{A}$ are useful for our refutation. Moreover, in order to complete the refutation made at each node C_i, each useful consequent atom for the refutation made at node C_i, except C_i itself (since it can be obtained by splitting $\mathcal{A} \to C_1; \ldots; C_m$ at node D), is also needed. Concludingly, in this case, the useful consequent atom set $\mathcal{S}_D$ can be derived as follows.

$$\mathcal{S}_D = \{\mathcal{S}_{C_1} - \{C_1\}\} \cup \cdots \cup \{\mathcal{S}_{C_m} - \{C_m\}\} \cup \mathcal{S}_{\mathcal{A}} \quad (1)$$

where $\mathcal{S}_{\mathcal{A}}$ is the useful consequent atom set for deriving the antecedent $\mathcal{A}$.

On the other hand, if node D is proven as unsatisfiable in the way that some of C_i $(1 \leq i \leq m)$ is found to be useless, then the use of the clause $\mathcal{A} \to C_1; \ldots; C_m$ is found to be unnecessary for the refutation made on node C_i, i.e., $C_i \notin \mathcal{S}_{C_i}$. In other words, the refutation found at node C_i can also be made at node D without using the clause $\mathcal{A} \to C_1;\ldots;C_m$. Thus, the useful consequent atom set $\mathcal{S}_D$ for deriving a refutation at node D is exactly the same as $\mathcal{S}_{C_i}$, i.e.,

$$\mathcal{S}_D = \mathcal{S}_{C_i} \quad (2)$$

In any case, when $\mathcal{S}_D$ is derived, all of $\mathcal{S}_{C_i}$ $(1 \le i \le m)$ are removed from database.

Let us make a comparison of I-SATCHMO's method and our new strategy. When node D is proven as unsatisfiable in the way that all of C_i are proven as unsatisfiable, the two methods are exactly similar, the useful consequent atom set $\mathcal{S}_D$ is obtained according to the formula (1). However, when node D is proven as unsatisfiable in the way that node C_i is found to be useless, the useful consequent atom set $\mathcal{S}_D$ derived by I-SATCHMO is as follows.

$$\mathcal{S}_D = \{\mathcal{S}_{C_1} - \{C_1\}\} \cup \cdots \cup \{\mathcal{S}_{C_{i-1}} - \{C_{i-1}\}\} \cup \mathcal{S}_{C_i} \quad (3)$$

Obviously, although the consequent atoms contained in $\{\mathcal{S}_{C_1} - \{C_1\}\}$, ..., $\{\mathcal{S}_{C_{i-1}} - \{C_{i-1}\}\}$ are useful for the branches beginning from C_1, $\cdots$, C_{i-1}, but useless for the refutation made at node C_i, and therefore is also useless to prove D unsatisfiable.

As we have seen, if a consequent atom, except the last atom of the consequence, of a violated $\mathcal{FC}$ clause is found to be useful, then its next sibling atom will be taken into processing. Therefore, the fewer the useful consequent atoms are, the more efficient a refutation is. It is quite clear that the useful consequent atom set for any node derived by our new strategy cannot be reduced any further, i.e., it is a minimal one. Since we can keep a minimal useful consequent atom set for a refutation, our new strategy is one of the most efficient approaches for forward chaining theorem proving.

The algorithm of our prover UNSEARCHMO to construct *PT* for checking whether a given clause set $\mathcal{S} = \mathcal{BC} \cup \mathcal{FC}$ is unsatisfiable can be described as follows.

For the current node D (initially the root node $\{\mathcal{BC}\}$) and its corresponding consequent atom set $\mathcal{I}_D$ (initially empty):

1. If $\mathcal{BC} \cup \mathcal{I}_D \vdash \bot$, create a leaf node $\bot$ below node D, which means node D is proven unsatisfiable. The useful consequent atom set for node D, $\mathcal{S}_D$, that consists of those consequent atoms listed in the successful deriving leaf in $DT_{\bot}^{\mathcal{BC} \cup \mathcal{I}_D}$, is established.
2. If $\mathcal{BC} \cup \mathcal{I}_D \not\vdash \bot$, select an instantiated ground violated clause $\mathcal{A} \rightarrow C_1; \ldots; C_m$ from $\mathcal{FC}$. If no such clause exists, $\mathcal{BC} \cup \mathcal{FC}$ is satisfiable.
3. For the selected violated non-Horn clause $\mathcal{A} \rightarrow C_1; \ldots;$ C_m, create a node C_i for all i such that $1 \le i \le m$. Taking C_i as the new current node, call this procedure recursively in depth-first strategy with the augmented consequent atom set $\mathcal{I}_{C_i} = \mathcal{I}_D \cup \{C_i\}$. However, if C_i is found to be useless on backtracking, instead of each node C_j such that $i < j \le m$, create a special node $\times$. The useful consequent atom set for node D, $\mathcal{S}_D$, is established according to the formula (2). On the other hand, if every C_i for $1 \le i \le m$ is found to be useful, the useful consequent atom set $\mathcal{S}_D$ is established according to the formula (1). In either case, node D is proven unsatisfiable.

Example 3.2 Suppose that $\mathcal{S} = \mathcal{BC} \cup \mathcal{FC}$ be the clause set given in in Example 2.1. The proof tree constructed by UNSEARCHMO is shown in Figure 6. Redundant search space has been eliminated.

The proof of the correctness of our UNSEARCHMO is not difficult. Since UNSEARCHMO just cuts down some branches in the proof tree constructed by SATCHMO, if SATCHMO can show a given clause set unsatisfiable, then UNSEARCHMO can certainly show the same thing. This shows the completeness of UNSEARCHMO.

On the other hand, as we have indicated above, an asserted consequent atom D is found useless at the time being retracted from the reasoning database, its parent node is able to be proven as unsatisfiable. Therefore, all remaining process to show its parent node unsatisfiable can be removed without the loss of the soundness. The further details are omitted here since the lack of space.

It is easy to implement UNSEARCHMO by Prolog. The code is omitted here also because of the lack of space.

4 An Example

Example 4.1 Let $\mathcal{S}$ be the following clause set, an extension of the benchmark problem SYN009-1 given in TPTP problem library [Sutcliffe and Suttner, 2000].

$$\begin{array}{ll}\mathcal{BC}: & p(X,Y,Z), m(X,Y,Z) \rightarrow \bot. \\ & q(X,Y,Z), m(X,Y,Z) \rightarrow \bot. \\ & r(X,Y,Z), m(X,Y,Z) \rightarrow \bot. \\ & n(c,c,c) \rightarrow \bot. \\ & s(a). \quad s(b). \quad s(c). \end{array}$$

$$\begin{array}{ll}\mathcal{FC}: & s(X), s(Y), s(Z) \\ & \quad \rightarrow p(X,Y,Z); q(X,Y,Z); r(X,Y,Z). \\ & s(X), s(Y), s(Z) \rightarrow m(X,Y,Z); n(X,Y,Z). \end{array}$$

The first $\mathcal{FC}$ clause will be fully split by any of SATCHMO, SATCHMORE and $\mathcal{A}$-SATCHMORE. That means more than 3^{27} cases will be considered. As a result, we gave up the our tests on SATCHMO, SATCHMORE and $\mathcal{A}$-SATCHMORE with a run of three days without termination on an Intel PentiumIII/650MHZ workstation, respectively.

Now, we see how I-SATCHMO and UNSEARCHMO work. Initially, the first $\mathcal{FC}$ clause is used for forward chaining with each substitution such that its antecedent becomes satisfiable. In this way, a branch $\{\mathcal{BC}\} \rightarrow p(a,a,a) \rightarrow p(a,a,b) \rightarrow p(a,a,c) \rightarrow \cdots \rightarrow p(c,c,c)$ is generated.

At node $p(c,c,c)$, the second $\mathcal{FC}$ clause with the substitution $\{X = a, Y = a, Z = a\}$, i.e., $s(a), s(a), s(a) \rightarrow m(a,a,a); n(a,a,a)$, is first split. For node $m(a,a,a)$, $\bot$ can be derived and the useful consequent atom set is established as $\{p(a,a,a), m(a,a,a)\}$. At node $n(a,a,a)$, $\bot$ can

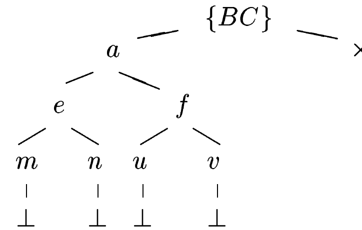


Figure 6: The proof tree PT constructed by UNSEARCHMO for the clause set given in Example 3.1

not be derived, and the second $\mathcal{FC}$ clause with the substitution $\{X = a, Y = a, Z = b\}$, i.e., $s(a), s(a), s(b) \rightarrow m(a, a, b); n(a, a, b)$, is split. At node $m(a, a, b)$, similar as node $m(a, a, a)$, $\perp$ is derived and the useful consequent atom set is $\{p(a, a, b), m(a, a, b)\}$. For node $n(a, a, b)$, similar as node $n(a, a, a)$, no refutation can be derived, then the second non-Horn clause will be split with the substitution $\{X = a, Y = a, Z = c\}$, and so on.

The second non-Horn clause will be further split until $s(c), s(c), s(c) \rightarrow m(c, c, c); n(c, c, c)$ is used for forward chaining at node $n(b, c, c)$. Then, at last a refutation can be derived from each of node $n(b, c, c)$'s child node simultaneously. The useful consequent atom set for the refutation at node $m(c, c, c)$ is $\{p(c, c, c), m(c, c, c)\}$ and that at node $n(c, c, c)$ is just $\{n(c, c, c)\}$. According to the formula (1), the useful consequent atom set for node $n(b, c, c)$ is constructed as $\{p(c, c, c)\}$.

If we do not distinguish the useful consequent atoms for different branches, as I-SATCHMO does, we will find that all p-facts have been marked as useful. Then, the same process made on each p-fact $p(x, y, z)$, where each of x, y and z is one of a, b and c, would be repeated for the corresponding q-fact $q(x, y, z)$ (also for the corresponding r-fact $r(x, y, z)$). Similar to other existing strategies, no answer can be obtained within a reasonable time.

However, at node $n(b, c, c)$, where the useful consequent atom set is $\{p(c, c, c)\}$, UNSEARCHMO finds $n(b, c, c)$ useless for the refutation, i.e., the clause $s(b), s(c), s(c) \rightarrow m(b, c, c); n(b, c, c)$ selected at node $n(b, c, c)$'s parent $n(a, c, c)$ is not necessary. According to the formula (1), the useful consequent atom set derived at its sibling node $m(b, c, c)$, i.e., $\{p(b, c, c), m(b, c, c)\}$, is abandoned, and the useful consequent atom set for node $n(a, c, c)$ is still $\{p(c, c, c)\}$.

Similar backtracking repeatedly continues to node $p(c, c, c)$, where the useful consequent atom set established by UNSEARCHMO is still $\{p(c, c, c)\}$. The same process made on node $p(c, c, c)$ will be also made on its sibling node $q(c, c, c)$ and node $r(c, c, c)$, and the corresponding useful consequent atom set are found to be $\{q(c, c, c)\}$ and $\{r(c, c, c)\}$, respectively. The violated clause $s(c), s(c), s(c) \rightarrow p(c, c, c); q(c, c, c);\ r(c, c, c)$ selected at node $p(b, c, c)$ is found to be necessary.

Now, let us see what happens when UNSEARCHMO backtracks to $p(c, c, c)$'s parent node $p(b, c, c)$, where, according to the formula (1), useful consequent atom set is surprisedly found to be empty! no other consequent atom is useful for our refutation! UNSEARCHMO simply backtracks to the root node $\{\mathcal{BC}\}$ to conclude that the given clause set is unsatisfiable.

With the help of distinguishing the useful consequent atoms for different branches, UNSEARCHMO only takes 0.002 seconds to find its solution on an Intel PentiumIII/650MHZ workstation.

5 Conclusion

In this paper, we have introduced how to eliminate redundant search space on backtracking for forward chaining theorem proving as much as possible. The redundant search branches are immediately pruned away when unnecessary ones are found on backtracking. At any reasoning point, only those necessary violated clauses are completely split for forward chaining. Repeated search by invoking unnecessary non-Horn clauses can be effectively eliminated. The experimental result has shown that our method is powerful for forward chaining theorem proving.

Since our strategy prunes away unnecessary proof branches after unnecessary non-Horn clauses are used for forward chaining, the violated non-Horn clauses for forward chaining can be used in any order decided by any advance checking strategy. That is, we can incorporate the relevancy checking proposed in SATCHMORE and the availability testing introduced in $\mathcal{A}$-SATCHMORE into our strategies to improve the performance further.

It is obvious that our method can be applied to disjunction logic programming (database). The principle of our method to eliminate redundant search space in forward chaining based prover can also be applied to reason about other logics whenever forward chaining strategy is used. This gives our method a wide application in automated reasoning field.

As future works, we will use our prover to test those benchmark problems provided in TPTP library [Sutcliffe and Suttner, 2000], to clear which class of problems will be helped by our approach.

References

[Bry and Yahya, 2000] Bry, F. and Yahya, A.: Positive Unit Hyperresolution Tableaux and Their Application to Minimal Model Generation. *Journal of Automated Reasoning*, 25:35-82, 2000.

[He *et al.*, 1998] He, L., Chao, Y., Simajiri, Y., Seki, H. and Itoh, H.: $\mathcal{A}$-SATCHMORE: SATCHMORE with Availability Checking. *New Generation Computing*, 16:55-74, 1998.

[He, 2001] He, L.: I-SATCHMO: an Improvement of SATCHMO. *To appear in J. of automated reasoning*.

[Loveland, 1968] Loveland, D.W.: Mechanical Theorem Proving by Model Elimination. *J. of the ACM*, 15:236-251, 1968.

[Loveland *et al.*, 1995] Loveland, D.W., Reed, D.W. and Wilson, D.S.: SATCHMORE: SATCHMO with RElevancy. *Journal of Automated Reasoning*, 14:325-351, 1995.

[Manthey and Bry, 1988] Manthey, R. and Bry, F.: SATCHMO: a theorem prover implemented in Prolog. In *Proceedings of 9th intl. Conf. on Automated Deduction*, 1988.

[Ramsay, 1991] Ramsay, A.: Generating Relevant Models. *Journal of Automated Reasoning*, 7:359-368, 1991.

[Robinson, 1965] Robinson, J.A.: A Machine-oriented logic based on the resolution principle. *J. of Ass. comput. Mach.*, 12, 23-41, 1965.

[Sutcliffe and Suttner, 2000] Sutcliffe, G. and Suttner, C.: http://www.cs.jcu.edu.au/~ tptp/

Theorem Proving with Structured Theories

Sheila McIlraith* and **Eyal Amir**
Department of Computer Science,
Gates Building, Wing 2A
Stanford University, Stanford, CA 94305-9020, USA
{sheila.mcilraith,eyal.amir}@cs.stanford.edu

Abstract

Motivated by the problem of query answering over multiple structured commonsense theories, we exploit graph-based techniques to improve the efficiency of theorem proving for structured theories. Theories are organized into subtheories that are minimally connected by the literals they share. We present message-passing algorithms that reason over these theories using consequence finding, specializing our algorithms for the case of first-order resolution, and for batch and concurrent theorem proving. We provide an algorithm that restricts the interaction between subtheories by exploiting the polarity of literals. We attempt to minimize the reasoning within each individual partition by exploiting existing algorithms for focused incremental and general consequence finding. Finally, we propose an algorithm that compiles each subtheory into one in a reduced sublanguage. We have proven the soundness and completeness of all of these algorithms.

1 Introduction

Theorem provers are becoming increasingly prevalent as query-answering machinery for reasoning over single or multiple large commonsense knowledge bases (KBs) [3]. Commonsense KBs, as exemplified by Cycorp's Cyc and the High Performance Knowledge Base (HPKB) systems developed by Stanford's Knowledge Systems Lab and by SRI often comprise tens/hundreds of thousands of logical axioms, embodying loosely coupled content in a variety of different subject domains. Unlike mathematical theories (the original domain of automated theorem provers), commonsense theories are often highly structured and with large signatures, lending themselves to graph-based techniques for improving the efficiency of reasoning.

Graph-based algorithms are commonly used as a means of exploiting structure to improve the efficiency of reasoning in Bayes Nets (e.g., [18]), Constraint Satisfaction Problems (CSPs) (e.g., [12]) and most recently in logical reasoning (e.g., [11; 3; 25]). In all cases, the basic approach is to convert a graphical representation of the problem into a tree-structured representation, where each node in the tree represents a tightly-connected subproblem, and the arcs represent the loose coupling between subproblems. Inference is done locally at each node and the necessary information is propagated between nodes to provide a global solution. Inference thus proves to be linear in the tree structure, and often worst-case exponential within the individual nodes.

We leverage these ideas to perform more efficient sound and complete theorem proving over theories in first-order logic (FOL) and propositional logic. In this paper we assume that we are given a first-order or propositional theory that is partitioned into subtheories that are minimally coupled, sharing minimal vocabulary. Sometimes this partitioning is provided by the user because the problem requires reasoning over multiple KBs. Other times, a partitioning is induced automatically to improve the efficiency of reasoning. (Some automated techniques for performing this partitioning are discussed in [3; 2].) This partitioning can be depicted as a graph in which each node represents a particular partition or subtheory and each arc represents shared vocabulary between subtheories. Theorem proving is performed locally in each subtheory, and relevant information propagated to ensure sound and complete entailment in the global theory. To maximize the effectiveness of structure-based theorem proving we must 1) minimize the coupling between nodes of the tree to reduce information being passed, and 2) minimize local inference within each node, while, in both cases, preserving global soundness and completeness.

In this paper we present message-passing algorithms that reason over partitioned theories, minimizing the number of messages sent between partitions and the local inference within partitions. We first extend the applicability of a message-passing algorithm presented in [3] to a larger class of local reasoning procedures. In Section 3 we modify this algorithm to use first-order *resolution* as the local reasoning procedure. In Section 4 we exploit Lyndon's Interpolation Theorem to provide an algorithm that reduces the size of the communication languages connecting partitions by considering the polarity of literals. Finally, in Section 5 we attempt to minimize the reasoning within each partition using algorithms for focused and incremental consequence finding. We also provide an algorithm for compiling partitioned propositional theories into theories in a reduced sublanguage. We

* Knowledge Systems Laboratory (KSL)

have proven the soundness and completeness of all of these algorithms with respect to reasoning procedures that are complete for consequence finding in a specified sublanguage. Proofs omitted from this paper can be found at [22].

2 Partition-Based Logical Reasoning

In this section we describe the basic framework adopted in this paper. We extend it with new soundness and completeness results that will enable us to minimize local inference.

Following [3], we say that $\{\mathcal{A}_i\}_{i \leq n}$ is a *partitioning* of a logical theory $\mathcal{A}$ if $\mathcal{A} = \bigcup_i \mathcal{A}_i$. Each individual $\mathcal{A}_i$ is a set of axioms called a *partition*, $L(\mathcal{A}_i)$ is its signature (the set of non-logical symbols), and $\mathcal{L}(\mathcal{A}_i)$ is its language (the set of formulae built with $L(\mathcal{A}_i)$). The partitions may share literals and axioms. A partitioning of a theory induces a graphical representation, $G = (V, E, l)$, which we call the theory's *intersection graph*. Each node of the intersection graph, i, represents an individual partition, $\mathcal{A}_i$, ($V = \{1, ..., n\}$), two nodes i, j are linked by an edge if $\mathcal{L}(\mathcal{A}_i)$ and $\mathcal{L}(\mathcal{A}_j)$ have a non-logical symbol in common ($E = \{(i,j) \mid L(\mathcal{A}_i) \cap L(\mathcal{A}_j) \neq \emptyset\}$), and the edges are labeled with the set of symbols that the associated partitions share ($l(i,j) = L(\mathcal{A}_i) \cap L(\mathcal{A}_j)$). We refer to $l(i,j)$ as the *communication language* between partitions $\mathcal{A}_i$ and $\mathcal{A}_j$. We ensure that the intersection graph is connected by adding a minimal number of edges to E with empty labels, $l(i,j) = \emptyset$. Figure 1 illustrates a propositional theory $\mathcal{A}$ in clausal form (left-hand side) and its partitioning displayed as an intersection graph (right-hand side). (Figures 1, 2 and 3 first appeared in [3].)

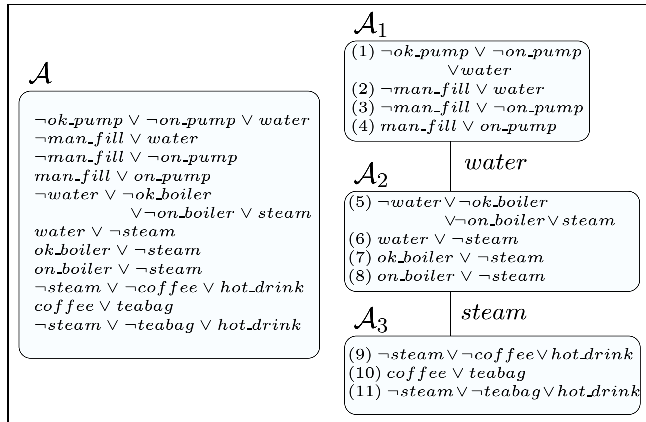


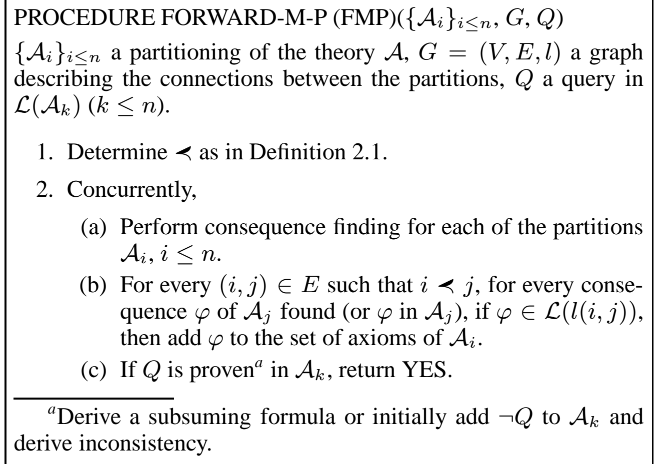
Figure 1: Partitioned theory $\mathcal{A}$ intersection graph G.

Figure 2 displays FORWARD-M-P (FMP), a message-passing algorithm for partition-based logical reasoning. It takes as input a partitioned theory, $\mathcal{A}$, an associated graph structure $G = (V, E, l)$, and a query formula Q in $\mathcal{L}(\mathcal{A}_k)$, and returns YES if the query was entailed by $\mathcal{A}$. The algorithm uses procedures that generate consequences (consequence finders) as the local reasoning mechanism within each partition or graphical node. It passes a concluded formula to an adjacent node if the formula's signature is in the communication language l of the adjacent node, and that node is on the path to the node containing the query.

Recall, consequence finding (as opposed to proof finding) was defined by Lee [19] to be the problem of finding all non-tautological logical consequences of a theory or sentences that subsume them. A prime implicate generator is a popular example of a consequence finder [1].

To determine the direction in which messages should be sent in the graph G, step 1 in FMP computes a strict partial order over nodes in the graph using the partitioning together with a query, Q.

Definition 2.1 ($\prec$) *Given partitioned theory $\mathcal{A} = \bigcup_{i \leq n} \mathcal{A}_i$, associated graph $G = (V, E, l)$ and query $Q \in \mathcal{L}(\mathcal{A}_k)$, let $dist(i,j)$ $(i, j \in V)$ be the length of the shortest path between nodes i, j in G. Then $i \prec j$ iff $dist(i,k) < dist(j,k)$.*


PROCEDURE FORWARD-M-P (FMP)($\{\mathcal{A}_i\}_{i \leq n}, G, Q$)
$\{\mathcal{A}_i\}_{i \leq n}$ a partitioning of the theory $\mathcal{A}$, $G = (V, E, l)$ a graph describing the connections between the partitions, Q a query in $\mathcal{L}(\mathcal{A}_k)$ $(k \leq n)$.

1. Determine $\prec$ as in Definition 2.1.
2. Concurrently,
   (a) Perform consequence finding for each of the partitions $\mathcal{A}_i, i \leq n$.
   (b) For every $(i,j) \in E$ such that $i \prec j$, for every consequence φ of $\mathcal{A}_j$ found (or φ in $\mathcal{A}_j$), if $\varphi \in \mathcal{L}(l(i,j))$, then add φ to the set of axioms of $\mathcal{A}_i$.
   (c) If Q is proven[a] in $\mathcal{A}_k$, return YES.

[a] Derive a subsuming formula or initially add $\neg Q$ to $\mathcal{A}_k$ and derive inconsistency.

Figure 2: A forward message-passing algorithm.

Figure 3 illustrates an execution of the FMP algorithm using resolution as the consequence finder within a partition. As can be seen from the example, the partitioning reduces the number of possible inference steps by precluding the direct resolution of axioms residing in different partitions. Indeed, [3] showed that partition-based reasoning reduces the search space significantly, as a function of the size of the communication language between partitions.

FMP is sound and complete if we guarantee some properties of the graph G and the consequence finders used for each partition. The graph G is required to be a tree that is *properly labeled* for $\mathcal{A}$.

Definition 2.2 (Proper Labeling) *A tree-structured representation, $G = (V, E, l)$, of a partitioned theory $\mathcal{A} = \{\mathcal{A}_i\}_{i \leq n}$ is said to have a* proper labeling, *if for all $(i,j) \in E$ and $\mathcal{B}_1, \mathcal{B}_2$, the two subtheories of $\mathcal{A}$ on the two sides of the edge (i,j) in G, it is true that $l(i,j) \supseteq L(\mathcal{B}_1) \cap L(\mathcal{B}_2)$.*

For example, every intersection graph that is a tree is properly labeled. A simple algorithm called BREAK-CYCLES

[1] Recall, an implicate is a clause entailed by a theory. It is prime if it is minimal in some way. Definitions of prime vary including the use of subsumption, syntactic minimality, or entailment.

Using FMP to prove hot_drink			
Part.	Resolve	Generating	
$\mathcal{A}_1$	(2) , (4)	$on_pump \vee water$	(m1)
$\mathcal{A}_1$	$(m1), (1)$	$ok_pump \vee water$	(m2)
$\mathcal{A}_1$	$(m2), (12)$	$water$	(m3)
		clause $water$ passed from $\mathcal{A}_1$ to $\mathcal{A}_2$	
$\mathcal{A}_2$	(m3) , (5)	$ok_boiler \wedge on_boiler \supset steam$	(m4)
$\mathcal{A}_2$	(m4) , (13)	$\neg on_boiler \vee steam$	(m5)
$\mathcal{A}_2$	(m5) , (14)	$steam$	(m6)
		clause $steam$ passed from $\mathcal{A}_2$ to $\mathcal{A}_3$	
$\mathcal{A}_3$	(9) , (10)	$\neg steam \vee teabag \vee hot_drink$	(m7)
$\mathcal{A}_3$	(m7) , (11)	$\neg steam \vee hot_drink$	(m8)
$\mathcal{A}_3$	(m8) , $(m6)$	hot_drink	(m9)

Figure 3: A proof of hot_drink from $\mathcal{A}$ in Figure 1 after asserting ok_pump (12) in $\mathcal{A}_1$ and ok_boiler (13), on_boiler (14) in $\mathcal{A}_2$.

that transforms an intersection graph that is not tree into a properly labeled tree was presented in [3]. Note that the notion of proper labeling is equivalent, in this context, to the running intersection property used in Bayes Nets.

The consequence finders applied to each partition i are required to be *complete for $\mathcal{L}_i$-generation* for a sublanguage $\mathcal{L}_i \subset \mathcal{L}(\mathcal{A}_i)$ that depends on the graph G and the query Q.

Definition 2.3 (Completeness for $\mathcal{L}$-Generation) *Let $\mathcal{A}$ be a set of axioms, $\mathcal{L} \subseteq \mathcal{L}(\mathcal{A})$ a language, and $\mathfrak{R}$ a consequence finder. Let $C_{\mathfrak{R},\mathcal{L}}(\mathcal{A})$ be the consequences of $\mathcal{A}$ generated by $\mathfrak{R}$ that are in $\mathcal{L}$. $\mathfrak{R}$ is* complete for $\mathcal{L}$-generation *if for all $\varphi \in \mathcal{L}$, if $\mathcal{A} \models \varphi$, then $C_{\mathfrak{R},\mathcal{L}}(\mathcal{A}) \models \varphi$.*

Theorem 2.4 (Soundness and Completeness) *Let $\mathcal{A}$ be a partitioned theory $\{\mathcal{A}_i\}_{i\leq n}$ of arbitrary propositional or first-order formulae, G a tree that is properly labeled with respect to $\mathcal{A}$, and $Q \in \mathcal{L}(\mathcal{A}_k)$, $k \leq n$, a query. For all $i \leq n$, let $\mathcal{L}_i = \mathcal{L}(l(i,j))$ for j such that $(i,j) \in E$ and $j \prec i$ (there is only one such j), and let $\{\mathfrak{R}_i\}_{i\leq n}$ be reasoning procedures associated with partitions $\{\mathcal{A}_i\}_{i\leq n}$. If every $\mathfrak{R}_i$ is complete for $\mathcal{L}_i$-generation then $\mathcal{A} \models Q$ iff FMP($\{\mathcal{A}_i\}_{i\leq n}$, G, Q) outputs YES.*

This soundness and completeness result improves upon a soundness and completeness result in [3] by allowing consequence finders that focus on consequences in the communication language between partitions. In certain cases, we can restrict consequence finding in FMP even further by using reasoners that are complete for *$\mathcal{L}$-consequence finding*.

Definition 2.5 (Completeness for $\mathcal{L}$-Consequence Finding) *Let $\mathcal{A}$ be a set of axioms, $\mathcal{L} \subseteq \mathcal{L}(\mathcal{A})$ a language, and $\mathfrak{R}$ a consequence finder. $\mathfrak{R}$ is* complete for $\mathcal{L}$-consequence finding *iff for every $\varphi \in \mathcal{L}$ that is not a tautology, $\mathcal{A} \models \varphi$ iff there exists $\psi \in \mathcal{L}$ such that $\mathcal{A} \vdash_{\mathfrak{R}} \psi$ and ψ subsumes[2] φ.*

[2]For clausal theories, we say that clause ψ subsumes φ if there is a substitution θ such that $\psi\theta \subset \varphi$.

Observe that every reasoner that is complete for $\mathcal{L}$-consequence finding is also complete for $\mathcal{L}$-generation, for any language $\mathcal{L}$ that is closed under subsumption [14]. The notion of a consequence finder restricting consequence generation to consequences in a designated sublanguage was discussed by Inoue [17], and further developed by del Val [14] and others. Most results on the completeness of consequence finding exploit resolution-based reasoners, where completeness results for $\mathcal{L}$-consequence finding are generally restricted to a clausal language $\mathcal{L}$. The FMP reasoners in Theorem 2.4 must be complete for $\mathcal{L}_i$-generation in arbitrary FOL languages, $\mathcal{L}_i$. Corollary 2.6 refines Theorem 2.4 by restricting $\mathcal{A}_i$ and $\mathcal{L}_i$ to propositional clausal languages and requiring reasoners to be complete for $\mathcal{L}_i$-consequence finding rather than $\mathcal{L}_i$-generation.

Corollary 2.6 (Soundness and Completeness) *Let $\mathcal{A}$ be a partitioned theory $\{\mathcal{A}_i\}_{i\leq n}$ of propositional clauses, G a tree that is properly labeled with respect to $\mathcal{A}$, and $Q \in \mathcal{L}(\mathcal{A}_k)$, $k \leq n$, a query. Let $\mathcal{L}_i = \mathcal{L}(l(i,j))$ for j such that $(i,j) \in E$ and $j \prec i$ (there is only one such j), and let $\{\mathfrak{R}_i\}_{i\leq n}$ be reasoning procedures associated with partitions $\{\mathcal{A}_i\}_{i\leq n}$. If every $\mathfrak{R}_i$ is complete for $\mathcal{L}_i$-consequence finding then $\mathcal{A} \models Q$ iff FMP($\{\mathcal{A}_i\}_{i\leq n}$, G, Q) outputs YES.*

In Section 5 we provide examples of reasoners that are complete for $\mathcal{L}$-consequence finding and show how to exploit them to focus reasoning within a partition.

3 Local Inference Using Resolution

In this section, we specialize FMP to *resolution-based* consequence finders. *Resolution* [26] is one of the most widely used reasoning methods for automated deduction, and more specifically for consequence finding. It requires the input formula to be in clausal form, i.e., a conjunction of disjunctions of unquantified literals. The *resolution rule* is complete for consequence finding (e.g., [19; 27]) and so is *linear resolution* and some of its variants (e.g., [23]).

We present algorithm RESOLUTION-M-P (RES-MP), that uses resolution (or resolution strategies), in Figure 4. The rest of this section is devoted to explaining four different implementations for subroutine RES-SEND(φ, j, i), used by this procedure to send appropriate messages between partitions: the first implementation is for clausal propositional theories; the second is for clausal FOL theories, with associated graph G, that is a properly labeled trees and whose labels include all the function and constant symbols of the language; the third is also for clausal FOL theories, however it uses *unskolemization* and subsequent Skolemization to generate the messages to be passed between partitions; the fourth is a refinement of the third for the same class of theories that avoids unskolemization.

In the propositional case, subroutine RES-SEND(φ, j, i) (Implementation 1) simply adds φ to $\mathcal{A}_i$, as done in FMP. If the resolution strategies being employed satisfy the conditions of Corollary 2.6, then RES-MP is sound and complete.

In the FOL case, implementing RES-SEND requires more care. To illustrate, consider the case where resolution generates the clause $P(B,x)$ (B a constant symbol and x a variable). It also implicitly proves that $\exists b\ P(b,x)$. RES-MP

PROCEDURE RESOLUTION-M-P(RES-MP)($\{\mathcal{A}_i\}_{i\leq n}$, G, Q)
$\{\mathcal{A}_i\}_{i\leq n}$ a partitioned theory, $G=(V,E,l)$ a graph, Q a query formula in the language of $\mathcal{L}(\mathcal{A}_k)$ ($k\leq n$).

1. Determine $\prec$ as in Definition 2.1.
2. Add the clausal form of $\neg Q$ to $\mathcal{A}_k$.
3. Concurrently,
   (a) Perform resolution for each of the partitions $\mathcal{A}_i$, $i\leq n$.
   (b) For every $(i,j)\in E$ such that $i\prec j$, if partition $\mathcal{A}_j$ includes the clause φ (as input or resolvent) and the predicates of φ are in $\mathcal{L}(l(i,j))$, then perform RES-SEND(φ, j, i).
   (c) If Q is proven in $\mathcal{A}_k$, return YES.

Figure 4: A resolution forward message-passing algorithm.

may need to send $\exists b\ P(b,x)$ from one partition to another, but it cannot send $P(B,x)$ if B is not in the communication language between partitions (for ground theories there is no such problem (see [27])). In the first-order case, completeness for consequence finding for a clausal first-order logic language (e.g., Lee's result for resolution) does not guarantee completeness for $\mathcal{L}$-generation for the corresponding full FOL language, $\mathcal{L}$. This problem is also reflected in a slightly different statement of Craig's interpolation theorem [10] that applies for resolution [27].

A simple way of addressing this problem is to add all constant and function symbols to the communication language between every connected set of partitions. This has the advantage of preserving soundness and completeness, and is simple to implement. In this case, subroutine RES-SEND(φ, j, i) (Implementation 2) simply adds φ to $\mathcal{A}_i$, as done in FMP.

In large systems that consist of many partitions, the addition of so many constant and function symbols to each of the other partitions has the potential to be computationally inefficient, leading to many unnecessary and irrelevant deduction steps. Arguably, a more compelling way of addressing the problems associated with resolution for first-order theories is to infer the existential formula $\exists b\ P(b,x)$ from $P(B,x)$, send this formula to the proper partition and Skolemize it there. For example, if $\varphi=P(f(g(B)),x)$ is the clause that RES-SEND gets, replacing it with $\exists b\ P(b,x)$ eliminates unnecessary work of the receiving partition.

The process of conservatively replacing function and constant symbols by existentially quantified variables is called *unskolemization* or *reverse Skolemization* and is discussed in [5; 9; 8]. [8] presents an algorithm *U* that is complete for our purposes and generalizes and simplifies an algorithm of [9]. (Space precludes inclusion of the algorithm.)

Theorem 3.1 ([8]) *Let V be a vocabulary and φ,ψ be formulae such that $\psi\in\mathcal{L}(V)$ and $\varphi\Rightarrow\psi$. There exists $F\in\mathcal{L}(V)$ that is generated by algorithm* U *such that $F\Rightarrow\psi$.*

Thus, for every resolution strategy that is complete for $\mathcal{L}$-consequence finding, unskolemizing φ using procedure *U* for $V=l(i,j)$ and then Skolemizing the result gives us a combined procedure for message generation that is complete for $\mathcal{L}_j$-generation. This procedure can then be used readily in RES-MP (or in FMP), upholding the soundness and completeness to that supplied by Theorem 2.4. The subroutine RES-SEND(φ, j, i) that implements this approach is presented in Figure 5. It replaces φ with a a set of formulae in $\mathcal{L}(l(i,j))$ that follows from φ. It then Skolemizes the resulting formulae for inclusion in $\mathcal{A}_i$.

PROCEDURE RES-SEND(φ, j, i) (Implementation 3)
φ a formula, $j,i\leq n$.

1. Unskolemize φ into a set of formulae, Φ in $\mathcal{L}(l(i,j))$, treating every symbol of $L(\varphi)\setminus l(i,j)$ as a Skolem symbol.
2. For every $\varphi_2\in\Phi$, if φ_2 is not subsumed by a clause that is in $\mathcal{A}_i$, then add the Skolemized version of φ_2 to the set of axioms of $\mathcal{A}_i$.

Figure 5: Subroutine RES-SEND using unskolemization.

Procedure *U* may generate more than one formula for any given clause φ. For example, if $\varphi=P(x,f(x),u,g(u))$, for $l(i,j)=\{P\}$, then we must generate both $\forall x\exists y\forall u\exists v P(x,y,u,v)$ and $\forall u\exists v\forall x\exists y P(x,y,u,v)$ (φ entails both quantified formulae, and there is no one quantified formula that entails both of them). In our case we can avoid some of these quantified formulae by replacing the *unskolemize and then Skolemize* process of RES-SEND (Implementation 3) with a procedure that produces a set of formulae directly (Implementation 4). It is presented in Figure 6.

PROCEDURE RES-SEND(φ, j, i) (Implementation 4)
φ a formula, $j,i\leq n$.

1. Set $S\leftarrow L(\varphi)\setminus l(i,j)$.
2. For every term instance, $t=f(t_1,...,t_k)$, in φ, if $f\in S$ and t is not a subexpression of another term $t'=f'(t'_1,...,t'_{k'})$ of φ with $f'\in S$, then replace t with "$x\leftarrow t$" for some new variable, x (if $k=0$, t is a constant symbol).
3. Nondeterministically[a], for every pair of marked arguments "$x\leftarrow\alpha$", "$y\leftarrow\beta$", in φ, if α,β are unifiable, then unify all occurrences of x,y (i.e., unify α_i,β_i for all markings $x\leftarrow\alpha_i$, $y\leftarrow\beta_i$).
4. For every marked argument "$x\leftarrow\alpha$" in φ, do
   (a) Collect all marked arguments with the same variable on the left-hand side of the "$\leftarrow$" sign. Suppose these are $x\leftarrow\alpha_1,...,x\leftarrow\alpha_l$.
   (b) Let $y_1,...,y_r$ be all the variables occurring in $\alpha_1,...,\alpha_l$. For every $i\leq l$, replace "$x\leftarrow\alpha_i$" with $f(y_1,...,y_r)$ in φ, for a fresh function symbol f (if $r=0$, f is a fresh constant symbol).
5. Add φ to $\mathcal{A}_i$.

[a] Nondeterministically select the set of pairs for which to unify all occurrences of x,y.

Figure 6: Subroutine RES-SEND without unskolemization.

Steps 2–3 of procedure RES-SEND(φ, j, i) (Implementation 4) correspond to similar steps in procedure *U* presented

in [8], simplifying where appropriate for our setup. Our procedure differs from unskolemizing procedures in step 4, where it stops short of replacing the Skolem functions and constants with new existentially quantified variables. Instead, it replaces them with new functions and constant symbols. The nondeterminism of step 3 is used to add *all* the possible combinations of unified terms, which is required to ensure completeness.

For example, if $\varphi = P(f(g(B)), x)$ and $l(i,j) = \{P\}$, then RES-SEND adds $P(A,x)$ to $\mathcal{A}_i$, for a new constant symbol, A. If $\varphi = P(x, f(x), u, g(u))$, for $l(i,j) = \{P\}$, then RES-SEND adds $P(x, h_1(x), u, h_2(u))$ to $\mathcal{A}_i$, for new function symbols h_1, h_2. Finally, if $\varphi = P(x, f(x), u, f(g(u)))$, then RES-SEND adds $P(x, f(x), u, h(u))$ and $P(h_1(u), h_2(u), u, h_2(u))$ to $\mathcal{A}_i$, for h, h_1, h_2 new function symbols.

Theorem 3.2 (Soundness & Completeness of RES-MP) *Let $\mathcal{A}$ be the partitioned theory $\bigcup_{i \leq n} \mathcal{A}_i$ of propositional or first-order clauses, G a tree that is properly labeled with respect to $\mathcal{A}$, and $Q \in \mathcal{L}(\mathcal{A}_k)$ $k \leq n$, a sentence that is the query. $\mathcal{A} \models Q$ iff applying RES-MP($\{\mathcal{A}_i\}_{i \leq n}$, G, Q) (with Implementation 4 of RES-SEND) outputs YES.*

PROOF SKETCH Soundness and completeness of the algorithm follow from that of FMP, if we show that RES-SEND (Implementation 4) adds enough sentences (implying completeness) to $\mathcal{A}_i$ that are implied by φ (thus sound) in the restricted language $\mathcal{L}(l(i,j))$.

If we add all sentences φ that are submitted to RES-SEND to $\mathcal{A}_i$ without any translation, then our soundness and completeness result for FMP applies (this is the case where we add all the constant and function symbols to all $l(i,j)$).

We use Theorem 3.1 to prove that we add enough sentences to $\mathcal{A}_i$. Let φ_2 be a quantified formula that is the result of applying algorithm *U* to φ. Then, φ_2 results from a clause C generated in step 4 of algorithm *U* (respectively, Step 3 in RES-SEND). In algorithm *U*, for each variable x, the markings "$x \leftarrow \alpha_i$" in C are converted to a new variable that is existentially quantified immediately to the right of the quantification of the variables $y_1, ..., y_r$. φ_2 is a result of ordering the quantifiers in a consistent manner to this rule (this process is done in steps 5–6 of algorithm *U*).

Step 4 of RES-SEND performs the same kind of replacement that algorithm *U* performs, but uses new function symbols instead of new quantified variables. Since each new quantified variable in φ_2 is to the right of the variables on which it depends, and our new function uses exactly those variables as arguments, then Step 4 generates a clause C' from C that entails φ_2. Thus, the clauses added to $\mathcal{A}_i$ by RES-SEND entail all the clauses generated by unskolemizing φ using *U*. From Theorem 3.1, these clauses entail all the sentences in $\mathcal{L}(l(i,j))$ that are implied by φ.

To see that the result is still sound, notice that the set of clauses that we add to $\mathcal{A}_i$ has the same consequences as φ in $\mathcal{L}(l(i,j))$ (i.e., if we add those clauses to $\mathcal{A}_j$ we get a conservative extension of $\mathcal{A}_j$). ■

Resolution strategies that can be readily used in RES-MP, while preserving completeness, include linear resolution [23], *directional resolution* [25] and *lock resolution* [7]. Strategies such as set-of-support and semantic resolution can be used with somewhat different treatments.

4 Minimizing Node Coupling Using Polarity

FMP and RES-MP use the communication language to determine relevant inference steps between formulae in connected partitions. This section improves the efficiency of FMP and RES-MP by exploiting the polarity of predicates in our partitions to further constrain the communication language between partitions. This leads to a reduction in the number of messages that are passed between adjacent partitions, and thus a reduction in the search space size of the global reasoning problem. Our results are predicated on Lyndon's Interpolation Theorem [20], an extension to Craig's Theorem [10].

Theorem 4.1 (Lyndon's Interpolation Theorem) *Let α, β be sentences such that $\alpha \vdash \beta$. Then there exists a sentence γ such that $\alpha \vdash \gamma$ and $\gamma \vdash \beta$, and that every relation symbol that appears positively [negatively] in γ appears positively [negatively] in both α and β. γ is referred to as the* interpolant *of α and β.*

This theorem guarantees that FMP need only send clauses with literals that may be used in subsequent inference steps. For example, let $\{\mathcal{A}_1, \mathcal{A}_2\}$ be a partitioned theory, $G = (V = \{1,2\},\ E = \{(1,2)\},\ l)$ be a graph, and $Q \in \mathcal{L}(\mathcal{A}_2)$, be a query. If FMP concluded P from $\mathcal{A}_1$, and P does not show positively in $\mathcal{A}_2 \Rightarrow Q$ (i.e., P does not show negatively in $\mathcal{A}_2$ and does not show positively in Q), then there is no need to send the message P from $\mathcal{A}_1$ to $\mathcal{A}_2$.

Procedure POLARIZE (Figure 7) takes as input a partitioned theory, associated tree $G = (V, E, l)$, and a query Q. It returns a new graph $G' = (V, E, l')$ that is minimal with respect to our interpretation of Lyndon's Interpolation Theorem. The labels of the graph now include predicate/propositional symbols with associated polarities (the same symbol may appear both positively and negatively on an edge label). All function and object symbols that appeared in l also appear in l' for the respective edges.

Theorem 4.2 (Soundness and Completeness) *Let $\mathcal{A}$ be a partitioned theory $\{\mathcal{A}_i\}_{i \leq n}$ of arbitrary propositional or first-order formulae, G a tree that is properly labeled with respect to $\mathcal{A}$, and $Q \in \mathcal{L}(\mathcal{A}_k)$, $k \leq n$, a query. Let G' be the result of running POLARIZE($\{\mathcal{A}_i\}_{i \leq n}$, G, Q). Let $\mathcal{L}_i = \mathcal{L}(l(i,j))$ for j such that $(i,j) \in E$ and $j \prec i$ (there is only one such j), and let $\{\mathfrak{R}_i\}_{i \leq n}$ be reasoning procedures associated with partitions $\{\mathcal{A}_i\}_{i \leq n}$. If every $\mathfrak{R}_i$ is complete for $\mathcal{L}_i$-generation then $\mathcal{A} \models \varphi$ iff FMP($\{\mathcal{A}_i\}_{i \leq n}$, G',Q) outputs YES.*

Darwiche [11] proposed a more restricted use of polarity in graph-based algorithms for propositional SAT-search. His proposal is equivalent to first finding those propositional symbols that appear with a unique polarity throughout the theory and then assigning them the appropriate truth value. In contrast, our proposed exploitation of polarity is useful for both propositional and first-order theories, it is more effective in constraining inference steps, and is applicable to a broader class of message-passing algorithms problems. In particular,

PROCEDURE POLARIZE($\{\mathcal{A}_i\}_{i\leq n}, G, Q$)
$\{\mathcal{A}_i\}_{i\leq n}$ a partitioning of the theory $\mathcal{A}$, $G = (V, E, l)$ a tree and Q a query formula in $\mathcal{L}(\mathcal{A}_k)$ $(k \leq n)$.

1. For every $i, j \in V$, set $l'(i, j)$ to be the set of object and function symbols that appear in $l(i, j)$, if there are any.
2. Rewrite $\{\mathcal{A}_i\}_{i\leq n}$ such that all negations appear in front of literals (i.e., in negation normal form).
3. Determine $\prec$ as in Definition 2.1.
4. For all $(i, j) \in E$ such that $i \prec j$, for every predicate symbol $P \in l(i, j)$,
   (a) Let V_1, V_2 be the two sets of vertices in V separated by i in G, with $j \in V_1$.
   (b) If $[\neg]P$ appears in V_1 then,
   if $[\neg]P$ appears in Q or $\neg[\neg]P$ appears in $\mathcal{A}_m$, for some $m \in V_2$, then add $[\neg]P$ to $l'(i, j)$.
5. Return $G' = (V, E, l')$.

Figure 7: Constraining the communication language of $\{\mathcal{A}_i\}_{i\leq n}$ by exploiting polarity.

our method is useful in cases where symbols appear with different polarities in different partitions.

5 Minimizing Local Inference

To maximize the effectiveness of structure-based theorem proving, we must minimize local inference within each node of our tree-structured problem representation, while preserving global soundness and completeness. First-order and propositional consequence finding algorithms have been developed that limit deduction steps to those leading to *interesting* consequences, *skipping* deduction steps that do not.

In the propositional case, the most popular algorithms are certain $\mathcal{L}$-(prime) implicate finders. (See [21] for an excellent survey.) SOL-resolution (skipping ordered linear resolution) [17] and SFK-resolution (skip-filtered, kernel resolution) [14] are two first-order resolution-based $\mathcal{L}$-consequence finders. SFK-resolution is complete for first-order $\mathcal{L}$-consequence finding, reducing to Directional Resolution in the propositional case [13]. In contrast, SOL-resolution is not complete for first-order $\mathcal{L}$-consequence finding, but is complete for first-order *incremental* $\mathcal{L}$-consequence finding. Given new input Φ, an *incremental $\mathcal{L}$-consequence finder* finds the consequences of $\mathcal{A} \cup \Phi$ that were not entailed by Φ alone. Defining *completeness for incremental $\mathcal{L}$-consequence finding* is analogous to Definition 2.5.

In the rest of this section, we propose strategies that exploit our graphical models and specialized consequence finding algorithms to improve the efficiency of reasoning. Following the results in previous sections, using SFK-resolution as a reasoner within partitions will preserve the soundness and completeness of the global problem while significantly reducing the number of inference steps. SFK-resolution can be used by all of the procedures below. Unless otherwise noted, the algorithms we describe are limited to propositional theories because first-order consequence finders may fail to terminate, even for decidable cases of FOL.

The first strategy is compilation. Figure 8 provides an algorithm, COMPILE($\{\mathcal{A}_i\}_{i\leq n}$, G), that takes as input a partitioned theory $\{\mathcal{A}_i\}_{i\leq n}$ and associated tree G, that is properly labeled, and outputs a compiled partitioned theory $\{\mathcal{A}_i'\}_{i\leq n}$. Each new partition is composed of the logical consequences of partition $\mathcal{A}_i$ that are in the language $\mathcal{L}_{comm_i}$, all the communication languages associated with $\mathcal{A}_i$. Prime implicate finders have commonly been used for knowledge compilation, particularly in propositional cases. SFK-resolution can be used as the sound and complete $\mathcal{L}$-consequence finder in Step 2 of COMPILE.

Knowledge compilation can often create a large theory. Each partition produced by COMPILE($\{\mathcal{A}_i\}_{i\leq n}$, G) will be of worst case size of $\mathcal{O}(2^{|L(\mathcal{L}_{comm_i})|})$ clauses. Since our assumption is that partitions are produced to minimize communication between partitions, $\mid L(\mathcal{L}_{comm_i}) \mid$ should be much smaller than $\mid L(\mathcal{A}_i) \mid$. As a consequence, we might expect the compiled theory to be smaller than the original theory, though this is not guaranteed. Under the further assumption that the theories in partitions are fairly static, the cost of compilation will be amortized over many queries. We discuss further options for compilation, including the use of partial compilation, in a longer paper.

PROCEDURE COMPILE($\{\mathcal{A}_i\}_{i\leq n}, G$)
$\{\mathcal{A}_i\}_{i\leq n}$ a partitioning of the theory $\mathcal{A}$, $G = (V, E, l)$ a tree with proper labeling for $\mathcal{A}$. For each partition $\mathcal{A}_i$, For $i = 1, \ldots, n$,

1. Let $\mathcal{L}_{comm_i} = \mathcal{L}(\bigcup_{(i,j)\in E} l(i, j))$
2. Using a sound and complete $\mathcal{L}$-consequence finder, perform $\mathcal{L}_{comm_i}$-consequence finding on each partition $\mathcal{A}_i$, placing the output in a new partition $\mathcal{A}_i'$.

Figure 8: A partition-based theory compilation algorithm.

Proposition 5.1 *Let $\mathcal{A} = \bigcup_{i\leq n} \mathcal{A}_i$ be a partitioned theory with associated tree G that is properly labeled for $\mathcal{A}$. Let $\mathcal{L}_{comm_i} = \mathcal{L}(\bigcup_{(i,j)\in E} l(i, j))$. For all $\varphi \in \mathcal{L}_i \subseteq \mathcal{L}_{comm_i} \subseteq \mathcal{L}(\mathcal{A}_i)$, $\mathcal{A}_i \models \varphi$ iff $\mathcal{A}_i' \models \varphi$, where $\{\mathcal{A}_i'\}_{i\leq n}$ are the compiled partitions output by COMPILE($\{\mathcal{A}_i\}_{i\leq n}$, G).*

We may use our compiled theories in several different strategies for batch-style and concurrent theorem proving, as well as in our previous message-passing algorithms. Figure 9 presents an algorithm for batch-style structure-based theorem proving. BATCH-MP takes as input a (possibly compiled) partitioned theory, associated tree G that is properly labeled, and query Q. For each partition in order, it exploits focused $\mathcal{L}$-consequence finding to compute all the relevant consequences of that theory. It passes the conclusions towards the partition with the query. This algorithm is very similar to the bucket elimination algorithm of [13]. BATCH-MP preserves soundness and completeness of the global problem, while exploiting focused search within each partition.

Theorem 5.2 (Soundness and Completeness) *Let $\mathcal{A}$ be a set of clauses in propositional logic. Let $\{\mathfrak{R}_i\}_{i\leq n}$ be the $\mathcal{L}_i$-consequence finders associated with partitions $\{\mathcal{A}_i\}_{i\leq n}$*

PROCEDURE BATCH-MP ($\{\mathcal{A}_i\}_{i\leq n}$, G, Q)
$\{\mathcal{A}_i\}_{i\leq n}$ a (compiled) partitioning of the theory $\mathcal{A}$, $G = (V, E, l)$ a properly labeled tree describing the connections between the partitions, Q a query in $\mathcal{L}(\mathcal{A}_k)$ ($k \leq n$).

1. If $\{\mathcal{A}_i\}_{i\leq n}$, is a compiled theory, replace partition A_k with the partition A_k from the uncompiled theory.
2. Determine $\prec$ as in Definition 2.1.
3. Let $\mathcal{L}_i = \mathcal{L}(l(i,j))$ for j such that $(i,j) \in E$ and $j \prec i$[a].
4. Following $\prec$ in a decreasing order, for every $(i,j) \in E$ such that $j \prec i$[a],
   Run the $\mathcal{L}_i$-consequence finder on $\mathcal{A}_i$ until it has exhausted its consequences, and add the consequences in $\mathcal{L}_i$ to $\mathcal{A}_j$.
5. If Q is proven[b] in $\mathcal{A}_k$, return YES.

[a] There is only one such j.
[b] Derive a subsuming formula or initially add $\neg Q$ to $\mathcal{A}_k$ and derive inconsistency.

Figure 9: A batch-style message-passing algorithm.

in step 4 of BATCH-MP ($\{\mathcal{A}_i\}_{i\leq n}$,G,Q). If every $\mathfrak{R}_i$ is complete for $\mathcal{L}_i$-consequence finding then $\mathcal{A} \models Q$ iff applying BATCH-MP($\{\mathcal{A}_i\}_{i\leq n}$,G,Q) outputs YES.

Our final algorithm, CONCURRENT-MP, (Figure 10), takes as input a (possibly compiled) partitioned theory, associated tree G that is properly labeled, and query Q. It exploits incremental $\mathcal{L}$-consequence finding in the output communication language of each partition to compute the relevant incremental consequences of that theory, and then passes them towards the partition with the query. Once again, SFK-resolution can be used as the sound and complete $\mathcal{L}$-consequence generator for the preprocessing (Step 4). In the case where the theory is compiled into propositional prime implicates, the consequences in $\mathcal{L}_i$ may simply be picked out of the existing consequences in $\mathcal{A}_i$. SOL-resolution can be used as the sound and complete incremental $\mathcal{L}$-consequence finder (Step 6a). CONCURRENT-MP preserves soundness and completeness of the global problem in the propositional case, while exploiting focused search within each partition.

Theorem 5.3 (Soundness and Completeness) *Let $\mathcal{A}$ be a set of clauses in propositional logic. Let $\{\mathfrak{R}_i\}_{i\leq n}$ be the $\mathcal{L}_i$-consequence finders associated with partitions $\{\mathcal{A}_i\}_{i\leq n}$ in step 4 of CONCURRENT-MP($\{\mathcal{A}_i\}_{i\leq n}$,G,Q) and let $\{\mathfrak{R}'_i\}_{i\leq n}$ be the incremental $\mathcal{L}_i$-consequence finders associated with partitions $\{\mathcal{A}_i\}_{i\leq n}$ in step 6 of CONCURRENT-MP($\{\mathcal{A}_i\}_{i\leq n}$,G,Q). If every $\mathfrak{R}_i$ is complete for $\mathcal{L}_i$-consequence finding, and every $\mathfrak{R}'_i$ is complete for incremental $\mathcal{L}_i$-consequence finding then $\mathcal{A} \models Q$ iff applying CONCURRENT-MP($\{\mathcal{A}_i\}_{i\leq n}$,G,Q) outputs YES.*

PROCEDURE CONCURRENT-MP ($\{\mathcal{A}_i\}_{i\leq n}$, G, Q)
$\{\mathcal{A}_i\}_{i\leq n}$ a (compiled) partitioning of the theory $\mathcal{A}$, $G = (V, E, l)$ a properly labeled tree describing the connections between the partitions, Q a query in $\mathcal{L}(\mathcal{A}_k)$ ($k \leq n$).

1. Determine $\prec$ as in Definition 2.1.
2. Let $\mathcal{L}_i = \mathcal{L}(l(i,j))$ for j such that $(i,j) \in E$ and $j \prec i$[a].
3. If $\{\mathcal{A}_i\}_{i\leq n}$, is a compiled theory, then replace partition A_k with the partition A_k from the uncompiled theory.
4. For every $i \leq n$, run the $\mathcal{L}_i$-consequence finder on partition $\mathcal{A}_i$ until it has exhausted its consequences.
5. For every $(i,j) \in E$ such that $j \prec i$[a], add the $\mathcal{L}_i$-prime implicates to partition $\mathcal{A}_j$.
6. Concurrently,
   (a) For every $(i,j) \in E$ such that $j \prec i$[a], perform incremental $\mathcal{L}_i$-consequence finding for each of the partition $\mathcal{A}_i$ and add the the consequences in $\mathcal{L}_i$ to $\mathcal{A}_j$.
   (b) If Q is proven[b] in $\mathcal{A}_k$, return YES.

[a] There is only one such j.
[b] Derive a subsuming formula or initially add $\neg Q$ to $\mathcal{A}_k$ and derive inconsistency.

Figure 10: A concurrent message-passing algorithm.

6 Related Work

A number of AI reasoning systems exploit some type of structure to improve the efficiency of reasoning. While our exploitation of graph-based techniques is similar to that used in Bayes Nets (e.g., [18]) our work is distinguished in that we reason with logical rather than probabilistic theories, where notions of structure and independence take on different roles in reasoning. Our work is most significantly distinguished from work on CSPs (e.g., [12]) and more recently, logical reasoning (e.g., [11; 25]) in that we reason with explicitly partitioned theories using message passing algorithms and our algorithms apply to FOL as well as propositional theories.

In the area of FOL theorem proving, our work is related to research on parallel theorem proving (see surveys in [6; 15]) and on combining logical systems (e.g., [24; 4]). Most parallel theorem prover implementations are guided by lookahead and subgoals to decompose the search space dynamically or allow messages to be sent between different provers working in parallel, using heuristics to decide on which messages are relevant to each prover. These approaches typically look at decompositions into very few sub-problems. In addition, the first approach typically requires complete independence of the sub-spaces or the search is repeated on much of the space by several reasoners. In the second approach there is no clear methodology for deciding what messages should be sent and from which partition to which.

The work on combining logical systems focuses on combinations of signature-disjoint theories (allowing the queries to include symbols from all signatures) and decision procedures suitable for those theories. All approaches either nondeterministically instantiate the (newly created) variables connecting the theories or restrict the theories to be convex (disjunctions are intuitionistic) and have information flowing back and forth between the theories. In contrast, we focus on the structure of interactions between theories with signatures that share symbols and the efficiency of reasoning with consequence finders and theorem provers. We do not have any

restrictions on the language besides finiteness.

Work on formalizing and reasoning with *context* (see [1] for a survey) can be related to theorem proving with structured theories by viewing the contextual theories as interacting sets of theories. Unfortunately, to introduce explicit contexts, a language that is more expressive than FOL is needed. Consequently, a number of researchers have focused on context for propositional logic, while much of the reasoning work has focused on proof checking (e.g., GETFOL [16]).

7 Summary

In this paper we exploited graph-based techniques to improve the efficiency of theorem proving for structured theories. Theories were organized into subtheories that were minimally connected by the literals they share. We presented message-passing algorithms that reason over these theories using consequence finding, specializing our algorithms for the case of first-order resolution, and for batch and concurrent theorem proving. We provided an algorithm that restricts the interaction between subtheories by exploiting the polarity of literals. We attempted to minimize the reasoning within each individual partition by exploiting existing algorithms for focused incremental and general consequence finding. Finally, we proposed an algorithm that compiles each subtheory into one in a reduced sublanguage. We have proven the soundness and completeness of all of these algorithms. The results presented in this paper contribute towards addressing the problem of reasoning efficiently with large or multiple structured commonsense theories.

Acknowledgements

We wish to thank the anonymous IJCAI reviewers for their thorough review of this paper, and Alvaro del Val and Pierre Marquis for helpful comments on the relationship between our work and previous work on consequence finding. This research was supported in part by DARPA grant N66001-97-C-8554-P00004, NAVY grant N66001-00-C-8027, and by DARPA grant N66001-00-C-8018 (RKF program).

References

[1] V. Akman and M. Surav. Steps toward formalizing context. *AI Magazine*, 17(3):55–72, 1996.

[2] E. Amir. Efficient approximation for triangulation of minimum treewidth. Manuscript submitted for publication. Available at http://www-formal.stanford.edu/eyal/papers/decomp-uai2001.ps, 2001.

[3] E. Amir and S. McIlraith. Paritition-based logical reasoning. In *Proc. KR '2000*, pages 389–400. Morgan Kaufmann, 2000.

[4] F. Baader and K. U. Schulz. Combination of constraint solvers for free and quasi-free structures. *Theoretical Computer Science*, 192(1):107–161, 1998.

[5] W. W. Bledsoe and A. M. Ballantyne. Unskolemizing. Technical Report Memo ATP-41, Mathematics Department, University of Texas, Austin, 1978.

[6] M. P. Bonacina and J. Hsiang. Parallelization of deduction strategies: an analytical study. *Journal of Automated Reasoning*, 13:1–33, 1994.

[7] R. S. Boyer. *Locking: a restriction of resolution*. PhD thesis, Mathematics Department, University of Texas, Austin, 1971.

[8] R. Chadha and D. A. Plaisted. Finding logical consequences using unskolemization. In *Proceedings of ISMIS'93*, volume 689 of *LNAI*, pages 255–264. Springer-Verlag, 1993.

[9] P. Cox and T. Pietrzykowski. A complete nonredundant algorithm for reversed skolemization. *Theoretical Computer Science*, 28:239–261, 1984.

[10] W. Craig. Linear reasoning. a new form of the Herbrand-Gentzen theorem. *J. of Symbolic Logic*, 22:250–268, 1957.

[11] A. Darwiche. Utilizing knowledge-based semantics in graph-based algorithms. In *Proc. AAAI '96*, pages 607–613, 1996.

[12] R. Dechter and J. Pearl. Tree Clustering Schemes for Constraint Processing. In *Proc. AAAI '88*, 1988.

[13] R. Dechter and I. Rish. Directional resolution: The Davis-Putnam procedure, revisited. In *Proc. KR '94*, pages 134–145. Morgan Kaufmann, 1994.

[14] A. del Val. A new method for consequence finding and compilation in restricted language. In *Proc. AAAI '99*, pages 259–264. AAAI Press/MIT Press, 1999.

[15] J. Denzinger and I. Dahn. Cooperating theorem provers. In W. Bibel and P. Schmitt, editors, *Automated Deduction. A basis for applications.*, volume 2, chapter 14, pages 383–416. Kluwer, 1998.

[16] F. Giunchiglia. GETFOL manual - GETFOL version 2.0. Technical Report DIST-TR-92-0010, DIST - University of Genoa, 1994. Available at http://ftp.mrg.dist.unige.it/pub/mrg-ftp/92-0010.ps.gz.

[17] K. Inoue. Linear resolution for consequence finding. *Artificial Intelligence*, 56(2-3):301–353, Aug. 1992.

[18] F. V. Jensen, S. L. Lauritzen, and K. G. Olesen. Bayesian updating in recursive graphical models by local computation. *Computational Statistics Quarterly*, 4:269–282, 1990.

[19] R. C.-T. Lee. *A Completeness Theorem and a Computer Program for Finding Theorems Derivable from Given Axioms*. PhD thesis, University of California, Berkeley, 1967.

[20] R. C. Lyndon. An interpolation theorem in the predicate calculus. *Pacific Journal of Mathematics*, 9(1):129–142, 1959.

[21] P. Marquis. Consequence finding algorithms. In *Algorithms for Defeasible and Uncertain Reasoning*, volume 5 of *Handbook on Deafeasible Reasoning and Uncertainty Management Systems*, pages 41–145. Kluwer, 2000.

[22] S. McIlraith and E. Amir. Theorem proving with structured theories (full report). Technical Report KSL-01-04, KSL, Computer Science Dept., Stanford U., Apr. 2001.

[23] E. Minicozzi and R. Reiter. A note on linear resolution strategies in consequence-finding. *Artificial Intelligence*, 3:175–180, 1972.

[24] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming Languages and Systems*, 1(2):245–257, Oct. 1979.

[25] I. Rish and R. Dechter. Resolution versus search: two strategies for SAT. *Journal of Automated Reasoning*, 24(1-2):225–275, 2000.

[26] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. of the ACM*, 12(1):23–41, 1965.

[27] J. R. Slagle. Interpolation theorems for resolution in lower predicate calculus. *J. of the ACM*, 17(3):535–542, July 1970.

LOGIC PROGRAMMING AND THEOREM PROVING

ANSWER SET PROGRAMMING

Experimenting with Heuristics for Answer Set Programming*

Wolfgang Faber
Inst. f. Informationssysteme 184/3, TU Wien
A-1040 Wien, Austria
faber@kr.tuwien.ac.at

Nicola Leone
Dep. of Mathematics, Univ. of Calabria
87030 Rende (CS), Italy
leone@unical.it

Gerald Pfeifer
Inst. f. Informationssysteme 184/2, TU Wien
A-1040 Wien, Austria
pfeifer@dbai.tuwien.ac.at

Abstract

Answer Set Programming (ASP) is a novel programming paradigm, which allows to solve problems in a simple and highly declarative way. The language of ASP (function-free disjunctive logic programming) is very expressive, and allows to represent even problems of high complexity (every problem in the complexity class $\Sigma_2^P = \mathrm{NP}^{\mathrm{NP}}$).

As for SAT solvers, the heuristic for the selection of the branching literal (i.e., the criterion determining the literal to be assumed true at a given stage of the computation) dramatically affects the performance of an ASP system. While heuristics for SAT have received a fair deal of research in AI, only little work in heuristics for ASP has been done so far.

In this paper, we extend to the ASP framework a number of heuristics which have been successfully employed in existing systems, and we compare them experimentally. To this end, we implement such heuristics in the ASP system DLV, and we evaluate their respective efficiency on a number of benchmark problems taken from various domains. The experiments show interesting results, and evidence a couple of promising heuristic criteria for ASP, which sensibly outperform the heuristic of DLV.

1 Introduction

Answer set programming (ASP) is a declarative approach to programming, which has been recently proposed in the area of nonmonotonic reasoning and logic programming. The knowledge representation language of ASP is very expressive: function-free logic programs with classical negation where disjunction is allowed in the heads of the rules and nonmonotonic negation may occur in the bodies of the rules. The intended models of an ASP program (i.e., the semantics of the program) are subset-minimal models which are "grounded" in a precise sense, they are called *answer sets* [Gelfond and Lifschitz, 1991]. The idea of answer set programming is to represent a given computational problem by an ASP program whose answer sets correspond to solutions, and then use an answer set solver to find such a solution [Lifschitz, 1999].

As an example, consider the well-known problem of 3-colorability, which is the assignment of three colors to the nodes of a graph in such a way that adjacent nodes have different colors. This problem is known to be NP-complete. Suppose that the nodes and the edges are represented by a set F of facts with predicates *node* (unary) and *edge* (binary), respectively. Then, the following ASP program allows us to determine the admissible ways of coloring the given graph.

$$r_1: \quad color(X,r) \vee color(X,y) \vee color(X,g) :- node(X)$$
$$r_2: \quad :- edge(X,Y), color(X,C), color(Y,C)$$

Rule r_1 above states that every node of the graph is colored **r**ed or **y**ellow or **g**reen, while r_2 forbids the assignment of the same color to any adjacent nodes. The minimality of answer sets guarantees that every node is assigned only one color. Thus, there is a one-to-one correspondence between the solutions of the 3-coloring problem and the answer sets of $F \cup \{r_1, r_2\}$. The graph is 3-colorable if and only if $F \cup \{r_1, r_2\}$ has some answer set.

An advantage of answer set programming over SAT-based programming is that problems can be encoded more easily in the ASP language than in propositional CNF formulas, thanks to the nonmonotonic character of disjunction and negation as failure [Lifschitz, 1999]. Importantly, due to the support for variables, *every* problem in the complexity class Σ_2^P (i.e., in $\mathrm{NP}^{\mathrm{NP}}$) can be directly encoded in an ASP program which can then be used to solve all problem instances in a uniform way [Eiter *et al.*, 1997].

The high expressiveness of answer set programming comes at the price of a high computational cost in the worst case, which makes the implementation of efficient ASP systems a difficult task. Nevertheless, some efforts have been done in this direction, and a number of ASP systems are now available. The two best known ASP systems are DLV [Eiter *et al.*, 2000], and Smodels [Niemelä, 1999; Simons, 2000], but also other systems support ASP to some extent, including CCALC

*This work was supported by FWF (Austrian Science Funds) under the project Z29-INF and P14781.

[McCain and Turner, 1998], DCS [East and Truszczyński, 2000], QUIP [Egly *et al.*, 2000], and XSB [Rao *et al.*, 1997].

The core of an ASP system is model generation, where a model of the program is produced, which is then subjected to an answer set check. For the generation of models, ASP systems employ procedures which are similar to Davis-Putnam procedures used in SAT solvers. As for SAT solvers, the heuristic (branching rule) for the selection of the branching literal (i.e., the criterion determining the literal to be assumed true at a given stage of the computation) is fundamentally important for the efficiency of a model generation procedure, and dramatically affects the overall performance of an ASP system. While a lot of work has been done in AI developing new heuristics and comparing alternative heuristics for SAT (see, e.g., [Hooker and Vinay, 1995; Li and Anbulagan, 1997; Freeman, 1995]), only little work has been done so far for ASP systems. In particular, we are not aware of any previous comparison between heuristics for ASP.

In this paper, we evaluate different heuristics for ASP systems. To this end, we first consider a couple of heuristics which have been very successful in SAT solvers or in non-disjunctive ASP systems, and we extend them to the framework of (disjunctive) ASP. We then perform an experimentation activity to compare the different heuristics. In particular, we implement the heuristics in the ASP system DLV, and we compare their respective efficiency on a number of benchmark problems taken from various domains. The experiments show interesting results, and evidence a couple of promising heuristic criteria for ASP, which sensibly outperform the heuristic of DLV. Nevertheless, this paper is not at all a conclusive work on heuristics for ASP; rather, it is a first step in this field that will hopefully stimulate further works on the design and evaluation of heuristics for ASP, which are strongly needed to build efficient ASP solvers.

2 Answer Set Programming Language

In this section, we provide a formal definition of the syntax and semantics of the answer set programming language supported by DLV: disjunctive datalog extended with strong negation. For further background, see [Gelfond and Lifschitz, 1991; Eiter *et al.*, 2000].

ASP Programs

A *(disjunctive) rule* r is a formula

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

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are classical literals (atoms possibly preceded by the symbol $\neg$) and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of r, while the conjunction $b_1, \cdots, b_k, \texttt{not } b_{k+1}, \cdots, \texttt{not } b_m$ is the *body*, $b_1, \cdots, b_k$ the *positive body*, and $\texttt{not } b_{k+1}, \cdots, \texttt{not } b_m$ the *negative body* of r. Comparison operators (like $=, <, >, <>$) are built-in predicates in ASP systems, and may appear in the bodies of rules.

A *disjunctive datalog program* (also called *ASP program* in this paper) $\mathcal{P}$ is a finite set of rules.

As usual, an object (atom, rule, etc.) is called *ground* or *propositional*, if it contains no variables.

Answer Sets

We describe the semantics of consistent answer sets, which has originally been defined in [Gelfond and Lifschitz, 1991].

Given a program $\mathcal{P}$, let the *Herbrand Universe* $U_\mathcal{P}$ be the set of all constants appearing in $\mathcal{P}$ and the *Herbrand Base* $B_\mathcal{P}$ be the set of all possible combinations of predicate symbols appearing in $\mathcal{P}$ with constants of $U_\mathcal{P}$, possibly preceded by $\neg$.

A set L of literals is said to be *consistent* if, for every literal $\ell \in L$, its complementary literal is not contained in L.

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 $U_\mathcal{P}$. Similarly, given a program $\mathcal{P}$, the *ground instantiation* $Ground(\mathcal{P})$ of $\mathcal{P}$ is the set $\bigcup_{r \in \mathcal{P}} Ground(r)$.

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 ground literals. A consistent interpretation $I \subseteq B_\mathcal{P}$ is *closed under* $\mathcal{P}$ (where $\mathcal{P}$ is a positive program), if, for every $r \in Ground(\mathcal{P})$, at least one literal in the head is true whenever all literals in the body are true. X is an *answer set* for $\mathcal{P}$ 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 (i) deleting all rules $r \in \mathcal{P}$ whose negative body is false w.r.t. X and (ii) 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 $Ground(\mathcal{P})^X$.

3 Answer Sets Computation

In this section, we describe the main steps of the computational process performed by ASP systems. We will refer particularly to the computational engine of the DLV system, which will be used for the experiments, but also other ASP systems, like Smodels, employ a very similar procedure.

An answer set program $\mathcal{P}$ in general contains variables. The first step of a computation of an ASP system eliminates these variables, generating a ground instantiation $ground(\mathcal{P})$ of $\mathcal{P}$.[1] The hard part of the computation is then performed on this ground ASP program generated by the instantiator.

The heart of the computation is performed by the Model Generator, which is sketched in Figure 1. Roughly, the Model Generator produces some "candidate" answer sets. The stability of each of them is subsequently verified by the function *IsAnswerSet(I)*, which verifies whether the given "candidate" I is a minimal model of the program $Ground(\mathcal{P})^I$ obtained by applying the GL-transformation w.r.t. I and outputs the model, if so. *IsAnswerSet(I)* returns True if the computation should be stopped and False otherwise.

[1] Note that $ground(\mathcal{P})$ is not the full set of all syntactically constructible instances of the rules of $\mathcal{P}$; rather, it is a subset (often much smaller) of it having precisely the same answer sets as $\mathcal{P}$ [Faber *et al.*, 1999a].

Function ModelGenerator(I: Interpretation): Boolean;
var inconsistency: Boolean;
begin
I := DetCons(I);
if I = $\mathcal{L}$ **then return** False; (* inconsistency *)
if no atom is undefined in I **then return** IsAnswerSet(I);
Select an undefined ground atom A according to a heuristic;
if ModelGenerator($I \cup \{A\}$) **then return** True;
else return ModelGenerator($I \cup \{\texttt{not}\ A\}$);
end;

Figure 1: Computation of Answer Sets

The ModelGenerator function is first called with parameter I set to the empty interpretation.[2] If the program $\mathcal{P}$ has an answer set, then the function returns True setting I to the computed answer set; otherwise it returns False. The Model Generator is similar to the Davis-Putnam procedure employed by SAT solvers. It first calls a function DetCons(), which returns the extension of I with the literals that can be deterministically inferred (or the set of all literals $\mathcal{L}$ upon inconsistency). This function is similar to a unit propagation procedure employed by SAT solvers, but exploits the peculiarities of ASP for making further inferences (e.g., it exploits the knowledge that every answer set is a minimal model). If DetCons does not detect any inconsistency, an atom A is selected according to a heuristic criterion and ModelGenerator is called on $I \cup \{A\}$ and on $I \cup \{\texttt{not}\ A\}$. The atom A plays the role of a branching variable of a SAT solver. And indeed, like for SAT solvers, the selection of a "good" atom A is crucial for the performance of an ASP system. In the next section, we describe a number of heuristic criteria for the selection of such branching atoms.

4 Heuristics

Throughout this section, we assume that a ground ASP program $\mathcal{P}$ and an interpretation I have been fixed. Here, we describe the heuristic criteria that will be compared in Section 7. We consider "dynamic heuristics" (the ASP equivalent of UP heuristics for SAT), that is, branching rules where the heuristic value of a literal Q depends on the result of taking Q true and computing its consequences. Given a literal Q, $ext(Q)$ will denote the intepretation resulting from the application of DetCons (see previous section) on $I \cup \{Q\}$; w.l.o.g., we assume that $ext(Q)$ is consistent, otherwise Q is automatically set to false and the heuristic is not evaluated on Q at all.

Heuristic $\mathbf{h_1}$. This is an extension of the branching rule adopted in the system SATZ [Li and Anbulagan, 1997] – one of the most efficient SAT solvers – to the framework of ASP.

The *length* of a rule r (w.r.t. an interpretation J), is the number of undefined literals occurring in r. Let $Unsat_k(Q)$ denote the number of unsatisfied rules[3] of length k w.r.t. $ext(Q)$, which have a greater length w.r.t. I. In other words, $Unsat_k(Q)$ is the number of unsatisfied rules whose length shrinks to k if the truth of Q is assumed and propagated in the interpretation I. The weight $w_1(Q)$ is

$$w_1(Q) = \Sigma_{k>1}\ Unsat_k(Q) * 5^{-k}$$

Thus, the weight function w_1 prefers literals introducing a higher number of short unsatisfied rules. Intuitively, the introduction of a high number of short unsatisfied rules is preferred because it creates more and stronger constraints on the intepretation so that a contradiction can be found earlier [Li and Anbulagan, 1997]. We combine the weight of an atom Q with the weight of its complement $\texttt{not}\ Q$ to favour Q such that $w_1(Q)$ and $w_1(\texttt{not}\ Q)$ are roughly equal, to avoid that a possible failure leads to a very bad state. To this end, as in SATZ, we define the combined weight *comb*-$w_1(Q)$ of an atom Q as follows:

$$comb\text{-}w_1(Q) = w_1(Q)*w_1(\texttt{not}\ Q)*1024+w_1(Q)+w_1(\texttt{not}\ Q).$$

Given two atoms A and B, heuristic h_1 prefers B over A ($A <_{h1} B$) iff *comb*-$w_1(A) <$ *comb*-$w_1(B)$. Once a $<_{h1}$-maximum atom Q is selected, heuristic h_1 takes Q if $w_1(Q) > w_1(\texttt{not}\ Q)$, $\texttt{not}\ Q$ else.

Heuristic $\mathbf{h_2}$. The second heuristic we consider is inspired to the branching rule of Smodels – a well known ASP system [Simons, 2000]. Let $|J|$ denote the number of atoms which are either true or false in a (three-valued) intepretation J. Then, define

$$w_2(Q) = |ext(Q)|.$$

Since w_2 maximizes the size of the resulting intepretation, it minimizes the atoms which are left undefined. Intuitively, this minimizes the size of the remaining search space (which is 2^u, where u is the number of undefined atoms in $ext(Q)$) [Simons, 2000]. Similar to Smodels, the heuristic h_2 cautiously maximizes the minimum of $w_2(Q)$ and $w_2(\texttt{not}\ Q)$. More precisely, the preference relationship $<_{h2}$ of h_2 is defined as follows. Given two atoms A and B, $A <_{h2} B$ if $min(h_2(A), h_2(\texttt{not}\ A)) < min(h_2(B), h_2(\texttt{not}\ B))$; otherwise, $A <_{h2} B$ if $min(h_2(A), h_2(\texttt{not}\ A)) = min(h_2(B), h_2(\texttt{not}\ B))$ and $max(h_2(A), h_2(\texttt{not}\ A)) < max(h_2(B), h_2(\texttt{not}\ B))$. Once a $<_{h2}$-maximum atom Q is selected, heuristic h_2 takes either Q or $\texttt{not}\ Q$, depending on the same selection principle.

Remark. It is worthwhile noting that the heuristic of Smodels, while following the above intuition, is more advanced and sophisticated than h_2. Unfortunately, it is defined for non-disjunctive programs, and centered around properties of unstratified negation. which is not so important in our framework. We do not see any immediate extension of Smodels' heuristic to the framework of disjunctive ASP programs. ∎

Heuristic $\mathbf{h_3}$. Let us consider now the heuristic used in the DLV system. Even if this is more "naive" than the previous heuristics, we will benchmark it in order to evaluate the impact of changing the branching rule on the test system.

A peculiar property of answer sets is *supportedness*: For each true atom A of an answer set I, there exists a rule r of the program such that the body of r is true w.r.t. I and A is the only true atom in the head of r. Since an ASP system must eventually converge to a supported interpretation, ASP

[2]Observe that the interpretations built during the computation are 3-valued, that is a literal can be True, False or Undefined (if its value has not been set yet) w.r.t. to an interpretation I.

[3]Recall that a rule r is satisfied w.r.t. J if the body of r is false w.r.t. J or the head of r is true w.r.t. J.

systems try to keep the interpretations “as much supported as possible” during the intermediate steps of the computation. To this end, the DLV system counts the number of *UnsupportedTrue (UT)* atoms, i.e., atoms which are true in the current interpretation but still miss a supporting rule (further details on UTs can be found in [Faber *et al.*, 1999b] where they are called MBTs). For instance, the rule $:- \, not \; x$ implies that x must be true in every answer set of the program; but it does not give a “support” for x. Thus, in the DLV system x is taken true to satisfy the rule, and it is added to the set of UnsupportedTrue; it will be removed from this set once a supporting rule for x will be found (e.g., $x \vee b :- c$ is a supporting rule for x in the interpretation $I = \{x, \texttt{not}\, b, c\}$). Given a literal Q, let $UT(Q)$ be the number of UT atoms in $ext(Q)$. Moreover, let $UT_2(Q)$ and $UT_3(Q)$ be the number of UT atoms occurring, respectively, in the heads of exactly 2 and 3 unsatisfied rules w.r.t. $ext(Q)$. The heuristic h_3 of DLV considers $UT(Q)$, $UT_2(Q)$ and $UT_3(Q)$ in a prioritized way, to favor atoms yielding interpretations with fewer $UT/UT_2/UT_3$ atoms (which should more likely lead to a supported model). If all UT counters are equal, then the heuristic considers the total number $Sat(Q)$ of rules which are satisfied w.r.t. $ext(Q)$. More precisely, given two atoms A and B:

1. $A <_{h3} B$ if $UT(A) > UT(B)$;
2. otherwise, $A <_{h3} B$ if $UT(A) = UT(B)$ and $UT_2(A) > UT_2(B)$;
3. otherwise, $A <_{h3} B$ if $UT_2(A) = UT_2(B)$ and $UT_3(A) > UT_3(B)$;
4. otherwise, $A <_3 B$ if $UT_3(A) = UT_3(B)$ and $Sat(A) < Sat(B)$.

A $<_{h3}$-maximum atom is selected by the heuristic h_3 of DLV. Unlike the previous heuristics, h_3 considers only atoms (instead of literals), and it does not take into account what happens when the selected atom Q leads to a failure (i.e., $ext(\texttt{not}\; Q)$ is not considered in the heuristic).

Heuristic $\mathbf{h_4}$. Finally, we have considered a simple “balanced version” h_4 of the heuristic h_3 of DLV, where also the complement of an atom is evaluated for the heuristic. Given an atom A, let $UT'(A) = UT(A) + UT(\texttt{not}\; A)$, $UT_2'(A) = UT_2(A) + UT_2(\texttt{not}\; A)$, $UT_3'(A) = UT_3(A) + UT_3(\texttt{not}\; A)$, and $Sat'(A) = Sat(A) + Sat(\texttt{not}\; A)$. The heuristic h_4 works precisely as h_3, but considers the primed counters. Once the best atom has been selected, it is taken positive or negative, depending on h_3.

5 Benchmark Programs

To evaluate the different heuristics presented in the previous section, we chose a couple of benchmark problems: 3SAT, Blocksworld Planning, Hamiltonian path, and Strategic Companies.

3SAT is one of the best researched problems in AI and generally used for solving many other problems by translating them to 3SAT, solving the 3SAT problem and transforming the solution back to the original domain:

> Let Φ be a propositional formula in conjunctive normal form (CNF) $\Phi = \bigwedge_{i=1}^{n}(d_{i,1} \vee \ldots \vee d_{i,3})$ where the $d_{i,j}$ are literals over the propositional variables $x_1, \ldots, x_m$.
>
> Φ is satisfiable, iff there exists a consistent conjunction I of literals such that $I \models \Phi$ (see e.g. [Papadimitriou, 1994] for a complete definition).

3SAT is a classical NP-complete problem and can be easily represented in our formalism as follows:

For every propositional variable x_i ($1 \leq i \leq m$), we add the following rule which ensures that we either assume that variable x_i or its complement nx_i true:

$$x_i \vee nx_i.$$

And for every clause $d_1 \vee \ldots \vee d_3$ in Φ we add the constraint

$$:- \, \texttt{not}\; \bar{d}_1, \ldots, \texttt{not}\; \bar{d}_3.$$

where $\bar{d}_i$ ($1 \leq i \leq 3$) is x_j if d_i is a positive literal x_j, and nx_j if d_i is a negative literal $\texttt{not}\; x_j$.

Hamiltonian Path (HAMPATH) is another classical NP-complete problem from the area of graph theory:

> Given an undirected graph $G = (V, E)$, where V is the set of vertices of G and E is the set of edges, and a node $a \in V$ of this graph, does there exist a path of G starting at a and passing through each node in V exactly once?

Suppose that the graph G is specified by using two predicates $node(X)$ and $arc(X, Y)$[4], and the starting node is specified by the predicate $start$ (unary) which contains only a single tuple. Then, the following program solves the problem HAMPATH.

% Each node has to be reached.
$:- \, node(X), \texttt{not}\; reached(X).$
$reached(X) :- start(X). \quad reached(X) :- inPath(Y, X).$
% Guess whether to take a path or not.
$inPath(X, Y) \vee outPath(X, Y) :- reached(X), arc(X, Y).$
% At most one incoming/outgoing arc!
$:- \, inPath(X, Y), inPath(X, Y1), Y <> Y1.$
$:- \, inPath(X, Y), inPath(X1, Y), X <> X1.$

Blocksworld (BW) is a classic problem from the planning domain, and one of the oldest problems in AI:

> Given a table and a number of blocks in a (known) initial state and a desired goal state, try to reach that goal state by moving one block at a time such that each block is either on top of another block or the table at any given time step.

Figure 2 shows a simple example that can be solved in three time steps: First we move block c to the table, then block b on top of a, and finally c on top of b.

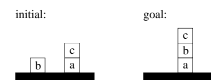


Figure 2: Simple BW Example

[4]Predicate arc is symmetric, since undirected arcs are bidirectional.

Due to space restrictions we refer to [Erdem, 1999; Faber *et al.*, 1999a] for a complete encoding.

Strategic Companies (STRATCOMP) finally, is a Σ_2^P-complete problem, which has been first described in [Cadoli *et al.*, 1997]:

> A holding owns c companies, each of which produces some goods. Some of these companies may jointly control others. This is modelled by means of predicates $prod(P, C1, C2)$ (P is produced by $C1$ and $C2$) and $cont(C, C1, C2, C3)$ (company C is jointly controlled by $C1$, $C2$ and $C3$).
>
> 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 afterwards. A company is strategic, if it belongs to a *strategic set*, which is a minimal set of companies satisfying these constraints.

The answer sets of the following natural program correspond one to one to the strategic sets. Checking whether any given company C is strategic is done by brave reasoning: "Is there *any* answer set containing C?"

$s_c(C1) \vee s_c(C2) :- prod(P, C1, C2).$
$s_c(C) :- cont(C, C1, C2, C3), s_c(C1), s_c(C2), s_c(C3).$

As in [Cadoli *et al.*, 1997] we assume that each product is produced by at most two companies and each company is jointly controlled by at most three companies to allow for an easier representation.

6 Benchmark Data

For 3SAT, we have randomly generated 3CNF formulas over n variables using a tool by Selman and Kautz [Selman and Kautz, 1997]. For each size we generated 8 such instances, where we kept the ratio between the number of clauses and the number of variables at 4.3, which is near the cross-over point for random 3SAT [Crawford and Auton, 1996].

The instances for HAMPATH were generated using a tool by Patrik Simons which has been used to compare Smodels against SAT solvers (cf. [Simons, 2000]), and is available at `http://tcs.hut.fi/Software/smodels/misc/hamilton.tar.gz`. For each problem size n we generated 8 instances, always assuming node 1 as the starting node.

The blocksworld problems P3 and P4 have been employed in [Erdem, 1999] to compare ASP systems, and can be solved in 8 and 9 steps, respectively. We augmented these by problems P5 and P6 which require 11 and 12 steps, respectively. For each of these problems, we generated 8 random permutations of the input.

For STRATCOMP, finally, we randomly generated 8 instances for each problem size n, with n companies and n products. Each company O is controlled by one to five companies, where the actual number of companies is uniform randomly chosen. On average there are 1.5 *cont* relations per company.

The benchmark data are available at `http://www.dbai.tuwien.ac.at/proj/dlv/`. All experiments were performed on an Athlon/750 FreeBSD 4.2 machine with 256MB of main memory. The binaries were produced with GCC 2.95.2.

7 Experimental Results and Conclusion

The results of our experiments are displayed in the graphs of Figures 3–6. In each graph, the horizontal axis reports a parameter representing the size of the instance. On the vertical axis, we report the average running time (expressed in seconds) over the 8 instances of the same size we have run (see previous section).

Remark. All heuristics have been implemented in a straightforward way, without optimizations, so the running times reported in the graph are meaningful only for comparing the relative efficiencies of the heuristics. ∎

We have allowed a running time of 600 seconds for each problem instance. In the graphs, the line of an heuristic stops whenever some problem instance was not solved in the maximum allowed time. The following table displays, for each heuristic, the maximum instance-size where the heuristic could solve all problem instances in the maximum allowed time.

	h_1	h_2	h_3	h_4
3SAT	270	270	250	260
HAMPATH	80	40	70	> 100
BW	$>$ P6	$>$ P6	P5	$>$ P6
STRATCOMP	> 1200	> 1200	< 700	> 1200

As expected, heuristic h_3, the "native" heuristic of the DLV system, which does not combine the heuristic values of complementary atoms, is the worst in most cases. It does not terminate on the instance P6 of BW, it could not solve any of the benchmark instances of STRATCOMP (it does not appear at all in Figure 5), and stopped earlier than the others on 3SAT.

Heuristic h_1, the extension of SATZ heuristic to ASP, behaves very well on average. On 3SAT, BW, and STRATCOMP, h_1 could solve all benchmark instances we have run. It is the fastest on BW and one of the two fastest on 3SAT. It shows a negative behaviour only on HAMPATH. In this problem, considering UnsupportedTrue literals seems to be crucial for the efficiency.

Heuristic h_4 is surprisingly good compared to h_3. It is a simple "balanced version" of heuristic h_3 (the heuristic values of the positive and of the negative literal are combined by sum). This simple extension to h_3 dramatically improves the performance. Indeed, heuristic h_4 solves nearly all instances we ran (only on 3SAT it stopped a bit earlier than other heuristics). It is the best heuristic on STRATCOMP and, importantly, on HAMPATH, where it beats all other heuristics of a relevant factor.

The behaviour of heuristic h_2, based on the minimization of the undefined atoms, is rather controversial. It behaves very well on 3SAT and BW, but it is extremely bad on HAMPATH, where it stops at 40 nodes already and is beaten even by the "naive" heuristic h_3. This confirms that further studies are needed to find a proper extension of the heuristic of Smodels to the framework of disjunctive ASP.

Concluding, we observe that both heuristic h_1 and heuristic h_4, significantly improve the efficiency of the native heuristic h_3 of the DLV system. The dramatic improvement obtained by the simple change from h_3 to h_4, confirms even more the importance of a careful study of branching rules in ASP systems. This paper is only a first step in this field, we hope that

further works will follow, for proposing new heuristics for ASP and for better understanding the existing ones, in order to improve the efficiency of ASP systems.

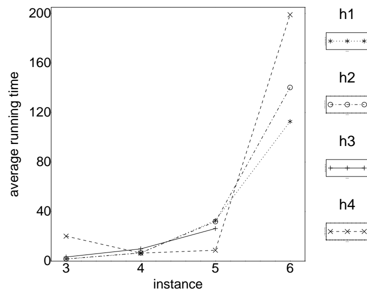


Figure 3: Blocksworld problems, average running times

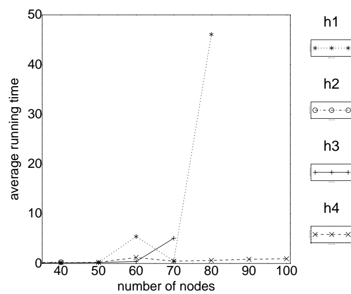


Figure 4: Hamiltonian Path problems, average running times

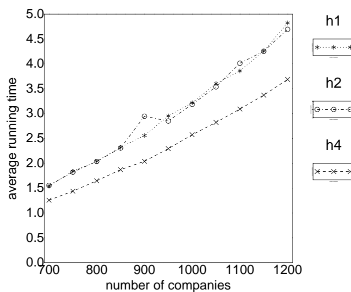


Figure 5: Strategic Companies, average running times

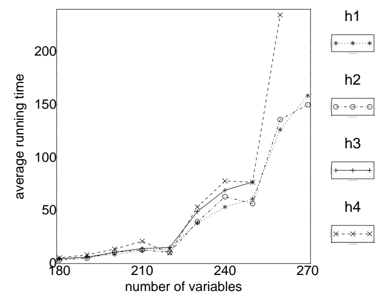


Figure 6: 3SAT problems, average running times

References

[Cadoli *et al.*, 1997] M. Cadoli, T. Eiter, and G. Gottlob. Default Logic as a Query Language. *IEEE TKDE*, 9(3):448–463, 1997.

[Crawford and Auton, 1996] James M. Crawford and Larry D. Auton. Experimental results on the crossover point in random 3sat. *Artificial Intelligence*, 81(1–2):31–57, March 1996.

[East and Truszczyński, 2000] D. East and M. Truszczyński. dcs: An implementation of DATALOG with Constraints. *NMR'2000.*

[Egly *et al.*, 2000] U. Egly, T. Eiter, H. Tompits, and S. Woltran. Solving Advanced Reasoning Tasks using Quantified Boolean Formulas. In *AAAI'00*, pp. 417–422. AAAI Press.

[Eiter *et al.*, 1997] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[Eiter *et al.*, 2000] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving Using the DLV System. *Logic-Based Artificial Intelligence*, pp. 79–103. Kluwer, 2000.

[Erdem, 1999] E. Erdem. Applications of Logic Programming to Planning: Computational Experiments. Unpublished draft. `http://www.cs.utexas.edu/users/esra/papers.html`, 1999.

[Faber *et al.*, 1999a] W. Faber, N. Leone, C. Mateis, and G. Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. *DDLP'99*, pp. 135–139.

[Faber *et al.*, 1999b] W. Faber, N. Leone, and G. Pfeifer. Pushing Goal Derivation in DLP Computations. *LPNMR'99*, pp. 177–191.

[Freeman, 1995] J.W. Freeman. *Improvements on Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.

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

[Hooker and Vinay, 1995] J.N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15:359–383, 1995.

[Li and Anbulagan, 1997] C.L. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI 1997*, pp. 366–371.

[Lifschitz, 1996] V. Lifschitz. Foundations of logic programming. *Principles of Knowledge Representation*, pp. 69–127. CSLI Publications, Stanford, 1996.

[Lifschitz, 1999] V. Lifschitz. Answer set planning. *ICLP'99*, pp. 23–37. The MIT Press.

[McCain and Turner, 1998] N. McCain and H. Turner. Satisfiability planning with causal theories. *KR'98*, pp. 212–223. Morgan Kaufmann Publishers, 1998.

[Niemelä, 1999] I. Niemelä. Logic programming with stable model semantics as constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999.

[Papadimitriou, 1994] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Rao *et al.*, 1997] P. Rao, K.F. Sagonas, T. Swift, D.S. Warren, and J. Freire. XSB: A System for Efficiently Computing Well-Founded Semantics. *LPNMR'97*, pp. 2–17.

[Selman and Kautz, 1997] B. Selman and H. Kautz, 1997. `ftp://ftp.research.att.com/dist/ai/`.

[Simons, 2000] P. Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Finland, 2000.

Graph Theoretical Characterization and Computation of Answer Sets

Thomas Linke
Institut für Informatik, Universität Potsdam
Postfach 60 15 53, D–14415 Potsdam, Germany, linke@cs.uni-potsdam.de

Abstract

We give a graph theoretical characterization of answer sets of normal logic programs. We show that there is a one-to-one correspondence between answer sets and a special, non-standard graph coloring of so-called *block graphs* of logic programs. This leads us to an alternative implementation paradigm to compute answer sets, by computing non-standard graph colorings. Our approach is rule-based and not atom-based like most of the currently known methods. We present an implementation for computing answer sets which works on polynomial space.

1 Introduction

Answer set semantics [Gelfond and Lifschitz, 1991] was established as an alternative declarative semantics for logic programs. Originally, it was defined for extended logic programs[1] [Gelfond and Lifschitz, 1991] as a generalization of the stable model semantics [Gelfond and Lifschitz, 1988]. Currently there are various applications of answer set programming, e.g. [Dimopoulos *et al.*, 1997; Liu *et al.*, 1998; Niemelä, 1999]. Furthermore, there are reasonably efficient implementations available for computing answer sets, e.g. `smodels` [Niemelä and Simons, 1997] and `dlv` [Eiter *et al.*, 1997]. Systems, like `DeReS` [Cholewiński *et al.*, 1996] and `quip` [Egly *et al.*, 2000], are also able to compute answer sets, although they were designed to deal with more general formalisms.

Both systems and most of the theoretical results as well deal with answer sets in terms of atoms (or literals). This paper aims at a different point of view, namely characterizing and computing answer sets in terms of rules. Intuitively, the head p of a rule $p \leftarrow q_1, \ldots, q_n, not\ s_1, \ldots, not\ s_k$ is in some answer set A if $q_1, \ldots, q_n$ are in A and none of $s_1, \ldots, s_k$ is in A. Let

$$P = \left\{ \begin{array}{ccc} a \leftarrow b, not\ e & b \leftarrow d & c \leftarrow b \\ d \leftarrow & e \leftarrow d, not\ f & f \leftarrow a \end{array} \right\} \quad (1)$$

be a logic program and let us call the rules r_a, r_b, r_c, r_d, r_e, and r_f, respectively. Then P has two different answer sets $A_1 = \{d, b, c, a, f\}$ and $A_2 = \{d, b, c, e\}$. It is easy to see that the application of r_f blocks the application of r_e wrt A_1, because if r_f contributes to A_1, then $f \in A_1$ and thus r_e cannot be applied. Analogously, r_e blocks r_a wrt answer set A_2. This observation leads us to a strictly blockage-based approach. More precisely, we represent the block relation between rules as a so-called *block graph*. Answer sets then are characterized as special non-standard graph colorings of block graphs. Each node of the block graph (corresponding to some rule) is colored with one of two colors, representing application or non-application of the corresponding rule. The block graph has quadratic size of the corresponding logic program. Since the block graph serves as basic data structure for our implementation, it needs polynomial space.

[1] Extended logic programs are logic programs with classical negation.

2 Background

We deal with normal logic programs which contain the symbol not used for *negation as failure*. A *rule*, r, is any expression of the form

$$p \leftarrow q_1, \ldots, q_n, not\ s_1, \ldots, not\ s_k \quad (2)$$

where p, q_i ($0 \leq i \leq n$) and s_j ($0 \leq j \leq k$) are atoms. A rule is a *fact* if n=k=0, it is called *basic* if k=0. For a rule r we define $head(r) = p$ and $body(r) = \{q_1, \ldots, q_n, not\ s_1, \ldots, not\ s_k\}$. Furthermore, let $body^+(r) = \{q_1, \ldots, q_n\}$ denote the positive part and $body^-(r) = \{s_1, \ldots, s_k\}$ the negative part of $body(r)$. Definitions of the head, the body, the positive and negative body of a rule are generalized to sets of rules in the usual way. The elements of $body^+(r)$ are referred to as the *prerequisites* or *positive body atoms* of r. The elements of $body^-(r)$ are referred to as the *negative body atoms* of r. If $body^+(r) = \emptyset$, then r is said to be *prerequisite-free*. A set of rules of the form (2) is called a *(normal) logic program*. A program is called *prerequisite-free* if each of its rules is prerequisite-free. We denote the set of all facts of program P by F_P.

Let r be a rule. r^+ then denotes the rule $head(r) \leftarrow body^+(r)$. For a logic program P let $P^+ = \{r^+ \mid r \in P\}$. A set of atoms X is *closed under* a basic program P iff for any $r \in P$, $head(r) \in X$ whenever $body(r) \subseteq X$. The smallest set of atoms which is closed under a basic program P is denoted by $\mathrm{Cn}(P)$.

The *reduct*, P^X, of a program P *relative to* a set X of atoms is defined by $P^X = \{r^+ \mid r \in P \text{ and } body^-(r) \cap X = \emptyset\}$. We say that a set X of atoms is an *answer set* of a program P iff $\mathrm{Cn}(P^X) = X$. The reduct P^X is often called the *Gelfond-Lifschitz reduction*. Observe, that there are programs, e.g. $\{p \leftarrow not\ p\}$, that do not possess an answer set. Throughout this paper, we use the term "answer set" instead of "stable

model" since it is the more general one.

A set of rules S of the form (2) is *grounded* iff there exists an enumeration $\langle r_i\rangle_{i\in I}$ of S such that for all $i\in I$ we have that $body^+(r_i) \subseteq head(\{r_1,\dots,r_{i-1}\})$. For a set of rules S and a set of atoms X we define the set of generating rules of S wrt X as follows

$$GR(S,X)=\{r\in S \mid body^+(r) \subseteq X, body^-(r)\cap X=\emptyset\}. \quad (3)$$

The following result relates groundedness and generating rules to answer sets.

Lemma 2.1 *Let P be a logic program and X be a set of atoms. Then X is an answer set of P iff (i) $GR(P,X)$ is grounded and (ii) $X=\mathrm{Cn}(GR(P,X)^+)$.*

This lemma characterizes answer sets in terms of generating rules. Observe, that in general $GR(P,X)^+ \neq P^X$ (take $P=\{a\leftarrow, b\leftarrow c\}$ and $X=\{a\}$).

Assume that for each program P we have

$$\text{for each rule } r\in P \text{ we have } |body^+(r)| \leq 1. \quad (4)$$

In Section 5, we show how to generalize our approach to both normal logic programs with multiple positive body atoms and also to extended logic programs. Therefore, the assumption above is not a real restriction.

We need some graph theoretical terminology. A directed graph G is a pair $G=(V,A)$ such that V is a finite, non-empty set (vertices) and $A\subseteq V\times V$ is a set (arcs). For a directed graph $G=(V,A)$ and a vertex $v\in V$, we define the set of all *predecessors* of v as $\gamma^-(v)=\{u \mid (u,v)\in A\}$. Analogously, the set of all *successors* of v is defined as $\gamma^+(v)=\{u \mid (v,u)\in A\}$. A *path from v to v'* in $G=(V,A)$ is a finite subset $P_{vv'}\subseteq V$ such that $P_{vv'}=\{v_1,\dots,v_n\}$, $v=v_1$, $v'=v_n$ and $(v_i,v_{i+1})\in A$ for each $1\leq i<n$. The arcs of a path $P_{vv'}$ are defined as $Arcs(P_{vv'})=\{(v_i,v_{i+1}) \mid 1\leq i<n\}$. A path from v to v for some $v\in V$ is called a *cycle* in G.

In order to represent more information in a directed graph, we need a special kind of labeled graphs. $(V,A^0\cup A^1)$ is a directed graph whose arcs $A^0\cup A^1$ are labeled with zero (*0-arcs*) and with one (*1-arcs*), respectively. For G we distinguish 0-predecessors (0-successors) from 1-predecessors (1-successors) denoted by $\gamma_0^-(v)$ ($\gamma_0^+(v)$) and $\gamma_1^-(v)$ ($\gamma_1^+(v)$) for $v\in V$, respectively. A path $P_{vv'}$ in G is called *0-path* if $Arcs(P_{vv'})\subseteq A^0$. The *length* of a cycle in a graph $(V,A^0\cup A^1)$ is the total number of 1-arcs occurring in the cycle. Additionally, we call a cycle *even* (*odd*) if its length is even (odd).

3 Block Graphs and Application Colorings

We now go on with a formal definition of the conditions under which a rule blocks another rule.

Definition 3.1 *Let P be a logic program s.t. condition (4) holds, and let $P'\subseteq P$ maximal grounded. The block graph $\Gamma_P=(V_P,A_P^0\cup A_P^1)$ of P is a directed graph with vertices $V_P=P$ and two different kinds of arcs defined as follows*

$$A_P^0 = \{(r',r) \mid r',r\in P' \text{ and } head(r')\in body^+(r)\}$$
$$A_P^1 = \{(r',r) \mid r',r\in P' \text{ and } head(r')\in body^-(r)\}.$$

Observe, that there exists a unique maximal grounded set $P'\subseteq P$ for each program P, that is, Γ_P is well-defined. This definition captures the conditions under which a rule r' blocks another rule r (e.g. $(r',r)\in A^1$). We also gather all groundedness information in Γ_P, due to the restriction to rules in the maximal grounded part of logic program P. This is important because a block relation between two rules r' and r becomes effective only if r' is groundable through other rules. E.g. for program $P=\{p\leftarrow q, q\leftarrow p\}$ the maximal grounded subset of rules is empty and therefore Γ_P contains no 0-arcs. Figure 1 shows the block graph of program (1).

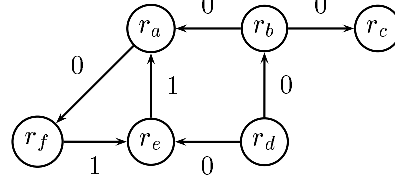


Figure 1: Block graph of program (1).

We now define so-called *application colorings* or *a-colorings* for block graphs. A subset of rules $G_r\subseteq P$ is a *grounded 0-path* for $r\in P$ if G_r is a 0-path from some fact to r in Γ_P.

Definition 3.2 *Let P be a logic program s.t. condition (4) holds, let $\Gamma_P=(P,A_P^0\cup A_P^1)$ be the corresponding block graph and let $c: P\mapsto\{\ominus,\oplus\}$ be a mapping. Then c is an a-coloring of Γ_P iff the following conditions hold for each $r\in P$*

A1 *$c(r)=\ominus$ iff one of the following conditions holds*
a. $\gamma_0^-(r)\neq\emptyset$ and for each $r'\in\gamma_0^-(r)$ we have $c(r')=\ominus$
b. there is some $r''\in\gamma_1^-(r)$ s.t. $c(r'')=\oplus$.

A2 *$c(r)=\oplus$ iff both of the following conditions hold*
a. $\gamma_0^-(r)=\emptyset$ or it exists grounded 0-path G_r s.t. $c(G_r)=\oplus$[2]
b. for each $r''\in\gamma_1^-(r)$ we have $c(r'')=\ominus$.

Let c be an a-coloring of some block graph Γ_P. Rules are then intuitively applied wrt some answer set of P if they are colored $\oplus$. Condition **A1** specifies that a rule r is colored $\ominus$ (not applied) if and only if r is not "grounded" (**A1***a*) **or** r is blocked by some other rule (**A1***b*). A rule is colored $\oplus$ (applied) if and only if it is grounded (**A2***a*) **and** it is not blocked by some other rule (**A2***b*) . This captures the intuition which rules apply wrt to some answer set and which do not (see Section 1).

Let $r_p = p\leftarrow not\ p$. Then program $P=\{r_p\}$ has block graph $(P,\emptyset,\{(r_p,r_p)\})$. By Definition 3.2 there is no a-coloring of Γ_P. For this reason, we need both conditions **A1** and **A2**.

Lemma 3.1 *Let P be a logic program s.t. condition (4) holds and let c be an a-coloring of Γ_P. Then condition* **A1** *holds iff condition* **A2** *does not hold.*

In general, we do not have the equivalence stated in this lemma, because there are examples (see above) where no a-coloring exists. Lemma 3.1 states that a-colorings are well-defined in the sense that they assign exactly one color to each node.

[2]For a set of rules $S\subseteq P$ we write $c(S)=\oplus$ or $c(S)=\ominus$ if for each $r\in S$ we have $c(r)=\oplus$ or $c(r)=\ominus$, respectively.

We obtain the main result:

Theorem 3.2 *Let P be a logic program s.t. condition (4) holds and let Γ_P be the block graph of P. Then P has an answer set A iff Γ_P has an a-coloring c. Furthermore, we have $GR(P,A) = \{r \in P \mid c(r) = \oplus\}$.*

Answer sets can therefore be computed by computing a-colorings, e.g. $c(\{r_e\}) = \ominus$ and $c(\{r_d, r_b, r_c, r_a, r_f\}) = \oplus$ correspond to answer set A_1 of program (1).

4 Computation of A-colorings

For the following description of our algorithm to compute a-colorings, let P be some logic program s.t. condition (4) holds. Let $c : P \mapsto \{\ominus, \oplus\}$ be a partial mapping. c is represented by a pair of (disjoint) sets $(c_\ominus, c_\oplus)$ s.t. $c_\ominus = \{r \in P \mid c(r) = \ominus\}$ and $c_\oplus = \{r \in P \mid c(r) = \oplus\}$[3]. We refer to mapping c with the tuple $(c_\ominus, c_\oplus)$ and vice versa. Assume that Γ_P is a global parameter of each presented procedure (indicated through index P). Let U and N be sets of nodes s.t. U contains the currently uncolored nodes ($U = P \setminus (c_\ominus \cup c_\oplus)$) and N contains colored nodes whose color has to be propagated. Figure 2 shows the implementation of the non-deterministic procedure **color**$_P$ in pseudo code.

```
procedure  colorP(U, N : list;  c : partial mapping)
 var  n : node;
   if  propagateP(N, c)  fails  then  fail;
   U := U \ (c⊖ ∪ c⊕);
   if  chooseP(U, c, n)  fails  then
      c := (c⊖ ∪ U, c⊕);
       if propagateP(U, c)  fails then fail  else  output  c;
   else
      U := U \ {n};
      c := (c⊖, c⊕ ∪ {n});
       if  colorP(U, {n}, c)  succeeds then exit
       else
          c := (c⊖ ∪ {n}, c⊕ \ {n});
           if colorP(U, {n}, c)  succeeds then exit  else  fail;
```

Figure 2: Definition of procedure **color**$_P$.

Notice that all presented procedures (except **choose**$_P$) return some partial mapping through parameter c or fail. **choose**$_P$ returns some node or fails.

When calling **color**$_P$ the first time, we start with $c = (\emptyset, F_P)$, $U = P \setminus F_P$ and $N = F_P$. That is, we start with all facts colored $\oplus$. Basically, **color**$_P$ takes both a partial mapping c and a set of uncolored nodes U and aims at coloring these nodes. This is done by choosing some uncolored node n ($n \in U$) with **choose**$_P$ and by trying to color it $\oplus$ first. In case of failure **color**$_P$ tries to color node n with $\ominus$. If this also fails **color**$_P$ fails. Therefore, we say that node n is used as a *choice point*. All different a-colorings are obtained by backtracking over choice points.

choose$_P(U, c, n)$ selects some uncolored node n ($n \in U$) s.t. $\gamma_0^-(n) = \emptyset$ and $\gamma_1^-(n) \neq \emptyset$ or the following condition holds:

CP there is some $n' \in \gamma_0^-(n)$ s.t. $c(n') = \oplus$.

[3]Since c is not total we do not necessarily have $P = c_\ominus \cup c_\oplus$.

If there is no such n then **choose**$_P$ fails. This strategy to select choice points ensures that nodes $c_\oplus$ are grounded. Observe that, if **choose**$_P$ fails and $U \neq \emptyset$ then we have to color all nodes in U with $\ominus$, since they cannot be grounded through rules $c_\oplus$.

During recursive calls N contains the choice point of the former recursion level. The color of nodes N has to be propagated with **propagate**$_P$ (see Figure 3) for two reasons. First, when coloring nodes in N with color x ($x \in \{\ominus, \oplus\}$) it is not checked whether this is allowed (wrt the actual c). This check is done by **propagate**$_P$. This means, **color**$_P$ fails only during propagation (see Theorem 4.1). Second, propagating already colored nodes prunes the search space and thus reduces the necessary number of choices. Since choice points make up the exponential part of our problem, propagation becomes the essential part of our approach.

Currently, we propagate only in arc direction as it is sufficient for correctness and completeness of the algorithm. Therefore, we have to deal with four propagation cases: if a node is colored x ($x \in \{\ominus, \oplus\}$) then this color has to be propagated over 1- and over 0-arcs. Let $r', r \in P$ be nodes s.t. $(r', r) \in A^0 \cup A^1$ and assume that r' is already colored. Then we have to propagate this color to node r. For example, propagating $c(r') = \oplus$ over 1-arcs gives $c(r) = \ominus$. For reasons of correctness, we cannot propagate colors without any further tests. We have got the following result.

Theorem 4.1 *Let P be a logic program s.t. condition (4) holds, let Γ_P be the corresponding block graph and let $c : P \mapsto \{\ominus, \oplus\}$ be a mapping. If c is an a-coloring of Γ_P then for each $r' \in P$ if $r' \in F_P$ then $c(r') = \oplus$ and the following conditions hold:*

(A) for each $r \in \gamma_1^+(r')$ we have $c(r) = \ominus$ if $c(r') = \oplus$

(B) for each $r \in \gamma_1^+(r')$ we have $c(r) = \oplus$ if $c(r') = \ominus$ and for each $r'' \in \gamma_1^-(r) : c(r'') = \ominus$ and $[\gamma_0^-(r) = \emptyset$ or there is some $r'' \in \gamma_0^-(r) : c(r'') = \oplus]$

(C) for each $r \in \gamma_0^+(r')$ we have $c(r) = \oplus$ if $c(r') = \oplus$ and for each $r'' \in \gamma_1^-(r) : c(r'') = \ominus$

(D) for each $r \in \gamma_0^+(r')$ we have $c(r) = \ominus$ if $c(r') = \ominus$ and for each $r'' \in \gamma_0^-(r) : c(r'') = \ominus$.

According to Theorem 3.2, a rule contributes to some answer set A if it is colored $\oplus$. In case of (A) there is no further condition, because a node r has to be colored $\ominus$ if there is some 1-predecessor r'' of r which is colored $\oplus$ (take $r'' = r'$ in $\mathbf{A1}b$ Definition 3.2). In other words, r cannot be applied if it is blocked by some other applied rule. Intuitively, condition (B) says that r has to be applied if all of its 1-predecessors are colored $\ominus$ (r is not blocked) and one of its 0-predecessors is colored $\oplus$ ($body^+(r)$ is a consequence of applied rules or r is a fact). Condition (C) states that rule r is applied if it is "grounded" through one of its 0-predecessors and if it is not blocked by some other rule. The last condition postulates that rule r cannot be applied if $body^+(r)$ cannot be derived from other applied rules. Theorem 4.1 implies that a mapping c is no a-coloring if **propagate**$_P$ fails. Figure 3 shows the implementation of **propagate**$_P$. The purpose of **propagate**$_P$ is to

```
procedure   propagate_P(N : list; c : partial mapping,)
  var   n' : node;
      while   N ≠ ∅   do
            select  n'  from  N;
            if   (n' ∈ c_⊕)   then
(A)               if   propA_P(n', c)  fails   then   fail;
(C)               if   propC_P(n', c)  fails   then   fail;
            else
(B)               if   propB_P(n', c)  fails   then   fail;
(D)               if   propD_P(n', c)  fails   then   fail.
```

Figure 3: Definition of procedure **propagate**$_P$.

apply the corresponding propagation cases, e.g. if $c(n') = \oplus$ then cases (A) and (C) have to be applied.

The four procedures used in **propagate**$_P$ can be easily implemented. For example, **propB**$_P$ is shown in Figure 4. First,

```
procedure   propB_P(n' : node;  c : partial mapping)
  var   n : node;   S : set of nodes;
  S := {n ∈ γ_1^+(n') | condition (B) holds for n};
   while   S ≠ ∅   do
     select  n  from  S;
     if   n ∈ c_⊖   then   fail;
     if   n ∉ c_⊕   then
        c := (c_⊖, c_⊕ ∪ {n});
        propagate_P(n, c).
```

Figure 4: Definition of procedure **propB**$_P$.

it determines the set S of all 1-successors of n' s.t. condition (B) holds. Finally, it tests whether all nodes in S can be colored $\oplus$. If node $n \in S$ is currently uncolored it is colored $\oplus$ and its color is propagated. If $c(n) = \ominus$ then **propB**$_P$ fails, otherwise n is already colored $\oplus$ and we go on with the next node from S. The procedures for the remaining propagation cases can be implemented analogously. Whenever some currently uncolored node is colored during propagation, this color is recursively propagated by calling **propagate**$_P$.

For partial mapping $c : P \mapsto \{\ominus, \oplus\}$ we define the set of *corresponding answer sets* A_c as

$$\begin{aligned} A_c = \quad & \{X \mid X \text{ is answer set of } P \text{ and} \\ & c_\oplus \subseteq GR(P,X) \text{ and } c_\ominus \cap GR(P,X) = \emptyset\}. \end{aligned}$$

If c is undefined for all nodes then A_c contains all answer sets of P. If c is a total mapping then A_c contains exactly one answer set of P (if c is an a-coloring). With this notation we formulate the following result:

Theorem 4.2 *Let P be a logic program s.t. condition (4) holds, let c and $c(r)$ be partial mappings. Then for each $r \in (c_\ominus \cup c_\oplus)$ we have if* **propagate**$_P(\{r\}, c)$ *succeeds and $c(r)$ is the actual partial mapping after executing* **propagate**$_P$ *then $A_c = A_{c(r)}$.*

This theorem states that **propagate**$_P$ neither discards nor introduces answer sets corresponding to some partial mapping c. Hence, it justifies that only nodes used as choice points lead to different answer sets. Therefore backtracking is necessary only over choice points (see Figure 2).

Define $C_P = \{c \mid c$ is some output of **color**$_P(P \setminus F_P, F_P, (\emptyset, F_P))\}$. With this notation, we obtain correctness and completeness of **color**$_P$.

Theorem 4.3 *Let P be a logic program s.t. condition (4) holds, let Γ_P be its block graph, let $c : P \mapsto \{\ominus, \oplus\}$ be a mapping and let C_P be defined as above. Then c is an a-coloring of Γ_P iff $c \in C_P$.*

Let us demonstrate how **color**$_P$ computes the a-colorings of the block graph of program (1) (see Figure 1). We invoke **color**$_P(U, N, c)$ with $U = P \setminus \{r_d\}$, $N = \{r_d\}$ and $c = (\emptyset, \{r_d\})$. First, **propagate**$_P(N, c)$ is executed. By propagating $c(r_d) = \oplus$ with case (C) we get $c(r_b) = \oplus$ and recursively $c(r_c) = \oplus$. This gives $c = (\emptyset, \{r_d, r_b, r_c\})$. After updating uncolored nodes we obtain $U = \{r_a, r_e, r_f\}$. Now **choose**$_P(U, c, n)$ (n variable) is executed. For **choose**$_P$ there are two possibilities to compute the next choice point s.t. **CP** hold, namely r_a and r_e. Assume $n = r_a$. After updating U we have $U = \{r_e, r_f\}$ and the first recursive call **color**$_P(U, \{r_a\}, c)$ is executed where $c = (\emptyset, \{r_d, r_b, r_c, r_a\})$. Again color $\oplus$ of node r_a has to be propagated by executing **propagate**$_P(\{r_a\}, (\emptyset, \{r_d, r_b, r_c, r_a\}))$. By using propagation case (C) for $c(r_a) = \oplus$ we obtain $c(r_f) = \oplus$. This color is recursively propagated using case (A), which gives $c(r_e) = \ominus$. This leads to $c = (\{r_e\}, \{r_d, r_b, r_c, r_a, r_f\})$. Since U becomes the empty set, **choose**$_P$ fails and c is the first output. Invoking backtracking means that the last recursive call to **color**$_P$ fails. Then $c = (\{r_a\}, \{r_d, r_b, r_c\})$ and **color**$_P(U, \{r_a\}, c)$ is executed. By using case (D) for $c(r_a) = \ominus$ we obtain $c(r_f) = \ominus$ and thus $c(r_e) = \oplus$ with case (B). Hence the second solution is $c = (\{r_a, r_f\}, \{r_d, r_b, r_c, r_e\})$. Since there is no other choice point, we have no further solutions.

5 Generalizations

In this section, we discuss generalizations of the presented approach. First, we show how to apply our method to normal logic programs with multiple positive body literals. Let P be a normal program without restrictions. For each rule $r = p \leftarrow q_1, \ldots, q_n, not\ s_1, \ldots, not\ s_k$ we define

$$P_r = \left\{ \begin{array}{l} p \leftarrow q', not\ s_1, \ldots, not\ s_k \\ q' \leftarrow q'_1, \ldots, q'_n \end{array} \right\} \cup \{q'_i \leftarrow q_i \mid 1 \leq i \leq n\} \quad (5)$$

where $q', q'_1, \ldots, q'_n$ are new atoms not appearing in P. For a program P we set $P_N = \bigcup_{r \in P} P_r$. Hence, we have defined a local program transformation which has linear size of the original program. Each normal program is transformed into some program in which $body^-(r) = \emptyset$ for each rule r with $|body^+(r)| > 1$. That is why we may interpret P_N as some kind of normal form of P. The following lemma obviously holds:

Lemma 5.1 *Let P be a normal logic program and let A be a set of atoms. Then A is an answer set of P iff there exists an answer set A_N of P_N s.t. A and A_N contain exactly the same atoms out of the set of all atoms occurring in P.*

It is straightforward to extend the algorithm presented in Section 4 to normal programs P_N. Let us call all rules $r \in P_N$ with $|body^+(r)| > 1$ *AND*-nodes and all other rules *OR*-nodes. Observe, that on the one hand, we do not have to modify Definition 3.1 of block graphs for programs P_N. It stays as

it is. We just distinguish two different kinds of nodes in Γ_{P_N}. On the other hand, Definition 3.2 and procedures presented in Section 4 deal only with OR-nodes and consequently we have to extend it to AND-nodes. For AND-node r we know by definition that $\gamma_1^-(r) = \emptyset$ (see (5)). Therefore, r cannot be blocked and we do not have to consider cases $\mathbf{A1}b$ and $\mathbf{A2}b$ of Definition 3.2. In order to extend Definition 3.2, we require that the following conditions hold for each AND-node r (corresponding to conditions $\mathbf{A1}a$ and $\mathbf{A2}a$ of Definition 3.2 for OR-nodes):

A3 $c(r) = \ominus$ iff there is some $r' \in \gamma_0^-(r)$ s.t. $c(r') = \ominus$

A4 $c(r) = \oplus$ iff for each $r' \in \gamma_0^-(r)$ we have $c(r') = \oplus$.

According to (5), for AND-node r we have $\gamma_1^-(r) = \gamma_1^+(r) = \emptyset$, $|\gamma_0^+(r)| = 1$ and $\gamma_0^+(r) \cup \gamma_0^-(r)$ will never contain any AND-node. For this reason, we obtain only two new propagation cases, in which we propagate the color of OR-nodes over 0-arcs to AND-nodes. Let $r \in \gamma_0^+(r')$ be some AND-node and let $x \in \{\ominus, \oplus\}$ be the actual color of r' (OR-node). Then we have the following new cases

(C') for each $r \in \gamma_0^+(r')$ we have $c(r) = \oplus$ if $c(r') = \oplus$ and for each $r'' \in \gamma_0^-(r) : c(r'') = \oplus$

(D') for each $r \in \gamma_0^+(r')$ we have $c(r) = \ominus$ if $c(r') = \ominus$,

which can be easily integrated into **propagate**$_P$.

[Gelfond and Lifschitz, 1990] show that logic programs with classical negation are equivalent to normal logic programs when new atoms are introduced. With this technique our approach is also suitable for computing answer sets of general logic programs (with classical negation). Additionally, we may apply techniques presented in [Janhunen *et al.*, 2000] to handle disjunctive logic programs.

6 Related Work

Directed graphs are often associated with logic programs and used in theory and applications. Usually, the nodes of these graphs are the atoms of the programs, e.g. dependency graphs or TMS networks [Doyle, 1979][4]. These approaches use rules to define graphs on atoms whereas we have used atoms to define graphs on rules.

Other approaches [Dimopoulos and Torres, 1996; Brignoli *et al.*, 1999] are more or less rule-based but have some serious drawbacks: they deal only with prerequisite-free programs, because (wrt to answer set semantics) there is some equivalent prerequisite-free program for each program. Since in general equivalent prerequisite-free programs have exponential size of the original ones, approaches which rely on this equivalence need exponential space.

In fact, the block graph is a specialization of graphs defined on rules in [Papadimitriou and Sideri, 1994; Linke and Schaub, 2000] for default theories. Whereas in the case of default logic the aforementioned graphs are abstractions of the essential blocking information, here they contain all information necessary for computing answer sets. Although we focus on the practical usage of the block graph it may also be used as a tool for theoretical analysis of logic programs. For example, results presented in [Linke and Schaub, 2000] imply that a logic program without odd cycles always has some answer set.

Clearly, **color**$_P$ is, like `smodels`, a Davis-Putnam like procedure. Once again, the main difference is that **color**$_P$ determines answer sets in terms of generating rules whereas `smodels` and `dlv` construct answer sets in terms of literals. Using rules instead of atoms has the advantage that we have complete knowledge about which rule is responsible for some atom belonging to an answer set. Atom-based approaches additionally have to detect the responsible rule and ensure groundedness, because in general there may be several rules with the same head. We obtain groundedness of generating rules as a by-product of our strategy to select choice points with procedure **choose**$_P$.

7 Conclusion

The main contribution of this paper was the definition of the block graph Γ_P of a program P. As a theoretical tool, the block graph seems to be suitable for investigations of many concepts for logic programs, e.g. answer set semantics, well-founded semantics or query-answering. As a first result, we have described answer sets as a-colorings (a non-standard kind of graph colorings of Γ_P). This led us to an alternative algorithm to compute answer sets by computing a-colorings which needs polynomial space.

Finally, let us give first experimental results to demonstrate the practical usefulness of our algorithm. We have used two NP-complete problems proposed in [Cholewiński *et al.*, 1995]: the problem of finding a Hamiltonian path in a graph (**Ham**) and the independent set problem (**Ind**).

Concerning time, our first prolog implementation (development time 6 month) is not comparable with state of the art implementations. However, Theorem 4.2 suggests to compare the number of used choice points, because it reflects how an algorithm deals with the exponential part of a problem. Unfortunately, only `smodels` gives information about its choice points. For this reason, we have concentrated on comparing our approach with `smodels`. Results are given for finding all solutions of different instances of **Ham** and **Ind**. Table 1 shows results for some **Ham**-encodings of complete graphs K_n where n is the number of nodes[5]. Surprisingly, it turns out that our non-monotonic reasoning system (`noMoRe`) performs very well on this problem class. That is, with growing problem size we need less choice points than `smodels`. This can also be seen in Table 2 which shows the corresponding time measurements. For finding all Hamiltonian cycles of a K_{10} we need less time than the actual `smodels` version. To be fair, for **Ind**-problems of graphs Cir_n[6] we need twice

	K_7	K_8	K_9	K_{10}
`smodels`	4800	86364	1864470	45168575
`noMoRe`	15500	123406	1226934	12539358

Table 1: Number of choice points for **HAM**-problems.

[4] Truth maintenance systems can be translated into logic programs [Brewka, 1991].

[5] In a complete graph each node is connected to each other node.

[6] A so-called circle graph Cir_n has n nodes $\{v_1, \ldots, v_n\}$ and arcs $A = \{(v_i, v_{i+1}) \mid 1 \leq i \leq n\} \cup \{(v_n, v_1)\}$.

the choice points (and much more time) smodels needs, because we have not yet implemented backward-propagation. However, even with the same number of choice points smodels is faster than noMoRe, because noMoRe uses general backtracking of prolog, whereas smodels backtracking is highly specialized for computing answer sets. The same applies to dlv.

	Ham for K_n			**Ind** for Cir_n		
$n =$	8	9	10	40	50	60
smodels	54	1334	38550	8	219	4052
dlv	4	50	493	13	259	4594
noMoRe	198	2577	34775	38	640	11586

Table 2: Time measurements in seconds for **HAM**- and **IND**-problems on a SUN Ultra2 with two 300MHz Sparc processors.

For future work, we may also think of different improvements of our algorithm. First of all, we have to integrate backward propagation (propagating colors against arc direction), since this will definitely improve efficiency by further reducing the number of choice points. Second, we may try to pre-color not only facts but also some other nodes, e.g. each node n with $(n, n) \in A^1$ has to be colored $\ominus$. The block graph may also be used for other improvements. For example, it is possible to replace 0-paths without incoming and outgoing 1-arcs by only one 0-arc. Finally, we have to investigate different heuristics for procedure $\mathbf{choose}_P$ to select the next choice point.

The approach as presented in this paper has been implemented in ECLiPSe-Prolog [Aggoun *et al.*, 2000]. The current prototype is available at http://www.cs.uni-potsdam.de/~linke/nomore.

Acknowledgements

I would like to thank T. Schaub, Ph. Besnard, K. Wang, K. Konczak, Ch. Anger and S.M. Model for commenting on previous versions of this paper.

This work was partially supported by the German Science Foundation (DFG) within Project "Nichtmonotone Inferenzsysteme zur Verarbeitung konfligierender Regeln".

References

[Aggoun *et al.*, 2000] A. Aggoun, D. Chan, P. Dufresne and other. Eclipse User Manual Release 5.0. ECLiPSe is available at http://www.icparc.ic.ac.uk/eclipse, 2000.

[Brewka, 1991] G. Brewka. *Nonmonotonic Reasoning: Logical Foundations of Commonsense*. Cambridge University Press, Cambridge, 1991.

[Brignoli *et al.*, 1999] G. Brignoli, S. Costantini, O. D'Antona, and A. Provetti. Characterizing and computing stable models of logic programs: the non-stratified case. In *Proc. of Conf. on Information Technology*, pp. 197–201, 1999.

[Cholewiński *et al.*, 1995] P. Cholewiński, V. Marek, A. Mikitiuk, and M. Truszczyński. Experimenting with nonmonotonic reasoning. In *Proc. of the Int. Conf. on Logic Programming*, pp. 267–281. MIT Press, 1995.

[Cholewiński *et al.*, 1996] P. Cholewiński, V. Marek, and M. Truszczyński. Default reasoning system DeReS. In *Proc. KR-1996*, pp. 518–528. Morgan Kaufmann Publishers, 1996.

[Dimopoulos and Torres, 1996] Y. Dimopoulos and A. Torres. Graph theoretical structures in logic programs and default theories. *Theoretical Computer Science*, 170:209–244, 1996.

[Dimopoulos *et al.*, 1997] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in non-monotonic logic programs. Proc. of the 4th European Conf. on Planing, pp. 169–181, Toulouse, France, 1997. Springer Verlag.

[Doyle, 1979] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[Egly *et al.*, 2000] U. Egly, Th. Eiter, H. Tompits, and St. Woltran. Solving advanced reasoning tasks using quantified boolean formulae. Proc. AAAI-2000, pp. 417–422, 2000.

[Eiter *et al.*, 1997] 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, *LPNMR-1997*, pages 363–374. Springer Verlag, 1997.

[Gelfond and Lifschitz, 1988] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of the Int. Conf. on Logic Programming*, 1988.

[Gelfond and Lifschitz, 1990] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proc. of the Int. Conf. on Logic Programming*, pp. 579–597, 1990.

[Gelfond and Lifschitz, 1991] M. Gelfond and V. Lifschitz. Classical negation in logic programs and deductive databases. *New Generation Computing*, 9:365–385, 1991.

[Janhunen *et al.*, 2000] T. Janhunen, I. Niemelä, P. Simons, and J. You. Unfolding partiality and disjunctions in stable model semantics. In A.G. Cohn, F. Guinchiglia, and B.Selman, editors, *Proc. KR-2000*, pp. 411–419, 2000.

[Linke and Schaub, 2000] T. Linke and T. Schaub. Alternative foundations for Reiter's default logic. *Artificial Intelligence*, 124:31–86, 2000.

[Liu *et al.*, 1998] X. Liu, C. Ramakrishnan, and S.A. Smolka. Fully local and efficient evaluation of alternating fixed points. Proc. of the 4th Int. Conf. on Tools and Algorithms for the Construction Analysis of Systems, pp. 5–19, Lisbon, Portugal, 1998. Springer Verlag.

[Niemelä and Simons, 1997] I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal logic programs. In J. Dix, U. Furbach, and A. Nerode, editors, *Proc. LPNMR-1997*, pp. 420–429. Springer, 1997.

[Niemelä, 1999] I. Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.

[Papadimitriou and Sideri, 1994] C. Papadimitriou and M. Sideri. Default theories that always have extensions. *Artificial Intelligence*, 69:347–357, 1994.

LOGIC PROGRAMMING AND THEOREM PROVING

LOGIC PROGRAMMING

A Framework for Declarative Update Specifications in Logic Programs

Thomas Eiter and **Michael Fink** and **Giuliana Sabbatini** and **Hans Tompits**
Institut für Informationssysteme, Abteilung Wissensbasierte Systeme 184/3,
Technische Universität Wien, Favoritenstrasse 9-11, A-1040 Vienna, Austria
E-mail:{eiter,michael,giuliana,tompits}@kr.tuwien.ac.at

Abstract

Recently, several approaches for updating knowledge bases represented as logic programs have been proposed. In this paper, we present a generic framework for declarative specifications of update policies, which is built upon such approaches. It extends the LUPS language for update specifications and incorporates the notion of events into the framework. An update policy allows an agent to flexibly react upon new information, arriving as an event, and perform suitable changes of its knowledge base. The framework compiles update policies to logic programs by means of generic translations, and can be instantiated in terms of different concrete update approaches. It thus provides a flexible tool for designing adaptive reasoning agents.

1 Introduction

Updating knowledge bases is an important issue for the realization of intelligent agents, since, in general, an agent is situated in a changing environment and must adjust its knowledge base when new information is available. While for classical knowledge bases this issue has been well-studied, approaches to update nonmonotonic knowledge bases, like, e.g., updates of logic programs [Alferes *et al.*, 2000; Eiter *et al.*, 2000; Zhang and Foo, 1998; Inoue and Sakama, 1999] or of default theories [Williams and Antoniou, 1998], are more recent.

The problem of updating logic programs, on which we focus here, deals with the incorporation of an update P, given by a rule or a set of rules, into the current knowledge base KB. Accordingly, sequences $P_1, \ldots, P_n$ of updates lead to sequences $(KB, P_1, \ldots, P_n)$ of logic programs, which are given a declarative semantics. To broaden this approach, Alferes *et al.* [1999a] have proposed the LUPS update language, in which updates consist of sets of *update commands*. Such commands permit to specify changes to KB in terms of adding or removing rules from it. For instance, a typical command is `assert` $a \leftarrow b$ `when` c, stating that rule $a \leftarrow b$ should be added to KB if c is currently true in it. Similarly, `retract` b expresses that b must be eliminated from KB, without any further condition.

However, a certain limitation of LUPS and the above mentioned formalisms is that while they handle *ad hoc* changes of KB, they are not conceived for handling a *yet unknown* update, which will arrive as the environment evolves. In fact, these approaches lack the possibility to specify how an agent should react upon the arrival of such an update. For example, we would like to express that, on arrival of the fact $best_buy(shop_1)$, this should be added to KB, while best-buy information about other shops is removed from KB.

In this paper, we address this issue and present a declarative framework for specifying update behavior of an agent. The agent receives new information in terms of a set of rules (which is called an *event*), and adjusts its KB in accord to a given *update policy*, consisting of statements in a declarative language. Our main contributions are summarized as follows:

(1) We present a *generic* framework for specifying update behavior, which can be instantiated with different update approaches to logic programs. This is facilitated by a *layered approach*: At the top level, the update policy is evaluated, given an event and the agent's current belief set, to single out the update commands U which need to be performed on KB. At the next layer, U is compiled to a set P of rules to be incorporated to KB; at the bottom level, the updated knowledge base is represented as a sequence of logic programs, serving as input for the underlying update semantics for logic programs, which determines the new current belief set.

(2) We define a declarative language for update policies, generalizing LUPS by various features. Most importantly, access to incoming events is facilitated. For example, **retract**$(best_buy(shop_1))$ $[\![\mathbf{E} : best_buy(shop_2)]\!]$ states that if $best_buy(shop_2)$ is told, then $best_buy(shop_1)$ is removed from the knowledge base. Statements like this may involve further conditions on the current belief set, and other commands to be executed (which is not possible in LUPS). The language thus enables the flexible handling of events, such as simply recording changes in the environment, skipping uninteresting updates, or applying default actions.

(3) We analyze some properties of the framework, using the update answer set semantics of Eiter *et al.* [2000] as a representative of similar approaches. In particular, useful properties concerning KB maintenance are explored, and the complexity of the framework is determined. Moreover, we describe a possible realization of the framework in the agent system IMPACT [Subrahmanian *et al.*, 2000], providing evidence that our approach is a viable tool for developing adaptive reasoning agents.

2 Preliminaries

We assume the reader familiar with *extended logic programs* (ELPs) [Gelfond and Lifschitz, 1991]. For a rule r, we write $H(r)$ and $B(r)$ to denote the head and body of r, respectively. Furthermore, not stands for default negation and $\neg$ for strong negation. $Lit_{\mathcal{A}}$ is the set of all literals over a set of atoms $\mathcal{A}$, and $\mathcal{L}_{\mathcal{A}}$ is the set of all rules constructible from $Lit_{\mathcal{A}}$.

An *update program*, $\boldsymbol{P}$, is a sequence $(P_1, \ldots, P_n)$ of ELPs, where $n \geq 1$. We adopt an abstract view of the semantics of ELPs and update programs, given as a mapping $Bel(\cdot)$, which associates with every sequence $\boldsymbol{P}$ a set $Bel(\boldsymbol{P}) \subseteq \mathcal{L}_{\mathcal{A}}$ of rules; intuitively, $Bel(\boldsymbol{P})$ are the consequences of $\boldsymbol{P}$. Different instantiations of $Bel(\cdot)$ are possible, according to various proposals for update semantics. We only assume that $Bel(\cdot)$ satisfies some elementary properties which any "reasonable" semantics satisfies. In particular, we assume that $P_n \subseteq Bel(\boldsymbol{P})$ holds, and that the following property is satisfied: given $A \leftarrow\, \in Bel(\boldsymbol{P})$ and $A \in B(r)$, then $r \in Bel(\boldsymbol{P})$ iff $H(r) \leftarrow B(r) \setminus \{A\} \in Bel(\boldsymbol{P})$.

We use here the semantics of Eiter *et al.* [2000], which coincides with the semantics of inheritance programs due to Buccafurri *et al.* [1999]. The semantics of ELPs P and update sequences $\boldsymbol{P}$ with variables is defined as usual through their ground versions $\mathcal{G}(P)$ and $\mathcal{G}(\boldsymbol{P})$ over the Herbrand universe, respectively. In what follows, let $\mathcal{A}$, P, $\boldsymbol{P}$, etc. be ground.

An *interpretation* is a set $S \subseteq Lit_{\mathcal{A}}$ which contains no complementary pair of literals. S is a (consistent) *answer set* of an ELP P iff it is a minimal model of the *reduct* P^S, which results from P by deleting all rules whose body contains some default literal $not\, L$ with $L \in S$, and by removing all default literals in the bodies of the remaining rules [Gelfond and Lifschitz, 1991]. By $\mathcal{AS}(P)$ we denote the collection of all answer sets of P. The *rejection set*, $Rej(S, \boldsymbol{P})$, of $\boldsymbol{P}$ with respect to the interpretation S is given by $Rej(S, \boldsymbol{P}) = \bigcup_{i=1}^{n} Rej_i(S, \boldsymbol{P})$, where $Rej_n(S, \boldsymbol{P}) = \emptyset$, and, for $n > i \geq 1$, $Rej_i(S, \boldsymbol{P})$ contains every rule $r \in P_i$ such that $H(r') = \neg H(r)$ and $B(r) \cup B(r') \subseteq S$, for some $r' \in P_j \setminus Rej_j(S, \boldsymbol{P})$ with $j > i$. Then, S is an *answer set* of $\boldsymbol{P} = (P_1, \ldots, P_n)$ iff S is an answer set of $\bigcup_i P_i \setminus Rej(S, \boldsymbol{P})$. We denote the collection of all answer sets of $\boldsymbol{P}$ by $\mathcal{AS}(\boldsymbol{P})$. Since $n = 1$ implies $Rej(S, \boldsymbol{P}) = \emptyset$, the semantics extends the answer set semantics. [Eiter *et al.*, 2000] describes a characterization of the update semantics in terms of single ELPs.

Example 1 *Let $P_0 = \{b \leftarrow not\, a,\, a \leftarrow\ \}$, $P_1 = \{\neg a \leftarrow\ ,\, c \leftarrow\ \}$, and $P_2 = \{\neg c \leftarrow\ \}$. Then, P_0 has the single answer set $S_0 = \{a\}$ with $Rej(S_0, P_0) = \emptyset$; (P_0, P_1) has answer set $S_1 = \{\neg a, c, b\}$ with $Rej(S_1, (P_0, P_1)) = \{a \leftarrow\}$; and $(P_0,\, P_1,\, P_2)$ possesses $S_2 = \{\neg a, \neg c, b\}$ as unique answer set with $Rej(S_2, (P_0, P_1, P_2)) = \{c \leftarrow, a \leftarrow\}$.*

The *belief set* $Bel_{\mathcal{A}}(\boldsymbol{P})$ is the set of all rules $r \in \mathcal{L}_{\mathcal{A}}$ such that r is true in each $S \in \mathcal{AS}(\boldsymbol{P})$. We shall drop the subscript "$\mathcal{A}$" if no ambiguity can arise. With a slight abuse of notation, for a literal L, we write $L \in Bel_{\mathcal{A}}(\boldsymbol{P})$ if $L \leftarrow\, \in Bel_{\mathcal{A}}(\boldsymbol{P})$.

3 Update Policies

We first describe our generic framework for event-based updating, and afterwards the EPI language ("the language around") for specifying *update policies*.

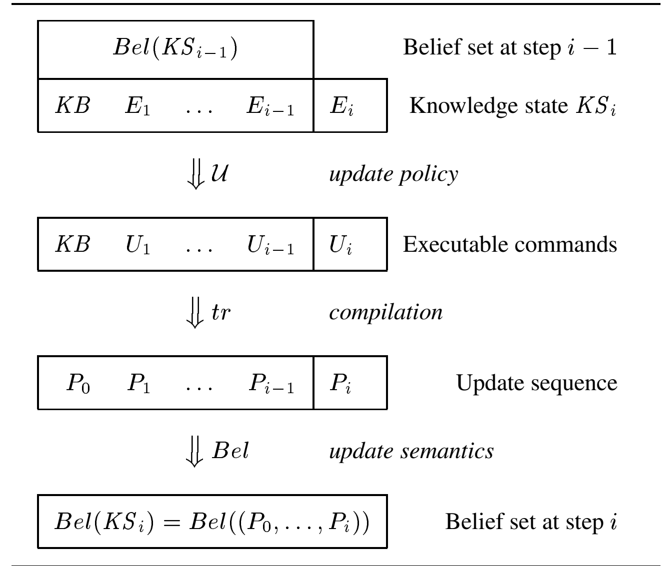


Figure 1: From knowledge state to belief set at step i.

3.1 Basic Framework

We start with the formal notions of an *event* and of the *knowledge state* of an agent.

Definition 1 *An* event class *is a collection $\mathcal{EC} \subseteq 2^{\mathcal{L}_{\mathcal{A}}}$ of finite sets of rules. The members $E \in \mathcal{EC}$ are called* events.

Informally, $\mathcal{EC}$ describes the possible events (i.e., sets of communicated rules) an agent may witness. For example, the collection $\mathcal{F}$ of all sets of facts from a subset $\mathcal{A}' \subseteq \mathcal{A}$ of atoms may be an event class. In what follows, we assume that an event class $\mathcal{EC}$ has been fixed.

Definition 2 *A* knowledge state *$KS = \langle KB; E_1, \ldots, E_n \rangle$ consists of an ELP KB (the* initial knowledge base*) and a sequence $E_1, \ldots, E_n$ of events $E_i \in \mathcal{EC}$, $i \in \{1, \ldots, n\}$. For $i \geq 0$, $KS_i = \langle KB; E_1, \ldots, E_i \rangle$ is the projection of KS to the first i events.*

Intuitively, KS describes the evolution of the agent's knowledge, starting from its initial knowledge base. When a new event E_i occurs, the current knowledge state KS_{i-1} changes to $KS_i = \langle KB; E_1, \ldots, E_{i-1}, E_i \rangle$, which requests the agent to incorporate the event E_i into its knowledge base and adapt its belief set.

The procedure for adapting the belief set $Bel(KS_{i-1})$ on arrival of E_i is illustrated in Figure 1. Informally, at step i of the knowledge evolution, we are given the belief set $Bel(KS_{i-1})$ and the knowledge state $KS_{i-1} = \langle KB; E_1, \ldots, E_{i-1} \rangle$, together with the new event E_i, and we want to compute $Bel(KS_i)$ in terms of the update policy $\mathcal{U}$. First, a set U_i of *executable commands* is determined from $\mathcal{U}$. Afterwards, given the previously computed sets $U_1, \ldots, U_{i-1}$, the sequence $(KB; U_1, \ldots, U_i)$ is compiled by the transformation tr into the update sequence $\boldsymbol{P} = (P_0, P_1, \ldots, P_i)$. Then, $Bel(KS_i)$ is given by $Bel(\boldsymbol{P})$.

$\langle stat\rangle$	::=	$\langle comm\rangle$ [**if** $\langle cond1\rangle$] [$[\![\langle cond2\rangle]\!]$];
$\langle c_name\rangle$	::=	**assert**[**_event**] | **retract**[**_event**] | **always**[**_event**] | **cancel** | **ignore** ;
$\langle r_id\rangle$	::=	$\langle rule\rangle$ | $\langle r_var\rangle$;
$\langle lit_id\rangle$	::=	$\langle literal\rangle$ | $\langle lit_var\rangle$;
$\langle comm\rangle$	::=	$\langle c_name\rangle(\langle r_id\rangle)$;
$\langle cond1\rangle$	::=	[**not**] $\langle comm\rangle$ | [**not**] $\langle comm\rangle, \langle cond1\rangle$;
$\langle cond2\rangle$	::=	$\langle kb_conds\rangle$ | $\mathbf{E} : \langle ev_conds\rangle$ | $\langle kb_conds\rangle$, $\mathbf{E} : \langle ev_conds\rangle$;
$\langle kb_conds\rangle$	::=	$\langle kb_cond\rangle$ | $\langle kb_cond\rangle$, $\langle kb_conds\rangle$;
$\langle kb_cond\rangle$	::=	$\langle r_id\rangle$ | $\langle lit_id\rangle$;
$\langle ev_conds\rangle$	::=	$\langle ev_cond\rangle$ | $\langle ev_cond\rangle$, $\langle ev_conds\rangle$;
$\langle ev_cond\rangle$	::=	$\langle lit_id\rangle$ | $\langle r_id\rangle$;

Table 1: Syntax of an update statement in EPI.

3.2 Language EPI: Syntax

The language EPI generalizes the update specification language LUPS [Alferes *et al.*, 1999a], by allowing update statements to depend on other update statements in the same EPI program, and more complex conditions on both the current belief set and the actual event (note that LUPS has no provision to support external events). These features make it suitable for implementing rational reactive agents, capable, e.g., of filtering incoming information.

The syntax of EPI is given in Table 1. In what follows, we use cmd to denote update commands and ρ to refer to rules or rule variables. In general, an EPI statement may have the form

$$cmd_1(\rho_1)\ \mathbf{if}\ [\mathbf{not}]cmd_2(\rho_2), \ldots, [\mathbf{not}]cmd_m(\rho_m)[\![c_1, \mathbf{E}: c_2]\!]$$

which states conditional assertion or retraction of a rule ρ_1, expressed by $cmd_1(\rho_1)$, depending on other commands $[\mathbf{not}]cmd_2(\rho_2)$, ..., $[\mathbf{not}]cmd_m(\rho_m)$, and conditioned with the proviso whether c_1 belongs to the current belief set and whether c_2 is in the actual event. The basic EPI commands are the same as those in LUPS (for their meaning, cf. [Alferes *et al.*, 1999a]), plus the additional command **ignore**, which allows to skip unintended updates from the environment, which otherwise would be incorporated into the knowledge base. Each condition in $[\![\cdot]\!]$, both of the form c_1 and $\mathbf{E}: c_2$, can be substituted by a list of such conditions. Note that in LUPS no conditions on rules and external events can be explicitly expressed, nor dependencies between update commands. We also extend the language by permitting variables for rules and literals in the update commands, ranging over the universe of the current belief set and of the current event (syntactic safety conditions can be easily checked). By convention, variable names start with capital letters.

Definition 3 *An* update policy $\mathcal{U}$ *is a finite set of* EPI *statements.*

For instance, the EPI statement

$$\mathbf{assert}(R)\ \mathbf{if\ not\ ignore}(R)[\![E : R]\!] \qquad (1)$$

means that all rules in the event have to be incorporated into the new knowledge base, except if it is explicitly specified that the rule is to be ignored. Similarly, the command **retract** forces a rule to be deactivated. The option **event** states that an assertion or retraction has only temporary value and is not supposed to persist by inertia in subsequent steps. The precise meaning of the different update commands will be made clear in the next section.

Example 2 *Consider a simple agent selecting Web shops in search for some specific merchandise. Suppose its knowledge base, KB, contains the rules*

$$r_1: \quad query(S) \leftarrow sale(S), up(S), not\ \neg query(S);$$
$$r_2: \quad try_query \leftarrow query(S);$$
$$r_3: \quad notify \leftarrow not\ try_query;$$

and a fact $r_0 : date(0)$ as an initial time stamp. Here, r_1 expresses that a shop S, which has a sale and whose Web site is up, is queried by default, and r_2, r_3 serve to detect that no site is queried, which causes 'notify' to be true. Assume that an event, E, might be any consistent set of facts or ground rules of the form $sale(s) \leftarrow date(t)$, stating that shop s has a sale on date t, such that E contains at most one time stamp $date(\cdot)$.

An update policy $\mathcal{U}$ may be defined as follows. Assume it contains the incorporate-by-default statement (1), as well as:

$$\mathbf{always}(sale(S) \leftarrow date(T))\ \mathbf{if\ assert}(sale(S) \leftarrow date(T));$$
$$\mathbf{cancel}(sale(S) \leftarrow date(T))[\![date(T), T \neq T', \mathbf{E}: date(T')]\!];$$
$$\mathbf{retract}(sale(S) \leftarrow date(T))[\![date(T), T \neq T', \mathbf{E}: date(T')]\!].$$

Informally, the first statement repeatedly confirms the information about a future sale, which guarantees that it is effective on the given date, while the second statement revokes this. The third one removes information about a previously ended sale (assuming the time stamps increase). Furthermore, $\mathcal{U}$ includes also the following statements:

$$\mathbf{retract}(date(T))[\![date(T), T \neq T', \mathbf{E}: date(T')]\!];$$
$$\mathbf{ignore}(sale(s_1))[\![\mathbf{E}: sale(s_1)]\!];$$
$$\mathbf{ignore}(sale(s_1) \leftarrow date(T))[\![\mathbf{E}: sale(s_1) \leftarrow date(T)]\!].$$

The first statement keeps the time stamp $date(t)$ in KB unique, and removes the old value. The other statements simply state that sales information about shop s_1 is ignored.

3.3 Language EPI: Semantics

According to the overall structure of the semantics of EPI, as depicted in Figure 1, at step i, we first determine the executable command U_i given the current knowledge state $KS_{i-1} = \langle KB; E_1, \ldots, E_{i-1}\rangle$ and its associated belief set $Bel(KS_{i-1}) = Bel(\boldsymbol{P}_{i-1})$, where $\boldsymbol{P}_{i-1} = (P_0, \ldots, P_{i-1})$. To this end, we evaluate the update policy $\mathcal{U}$ over the new event E_i and the belief set $Bel(\boldsymbol{P}_{i-1})$.

Let $\mathcal{G}(\mathcal{U})$ be the grounded version of $\mathcal{U}$ over the language $\mathcal{A}$ underlying the given update sequence and the received events. Then, the set $\mathcal{G}(\mathcal{U})^i$ of reduced update statements at step i is given by

$$\mathcal{G}(\mathcal{U})^i = \{\ cmd(\rho)\ \mathbf{if}\ C_1 \mid cmd(\rho)\ \mathbf{if}\ C_1[\![C_2]\!] \in \mathcal{G}(\mathcal{U}),\ \text{where}\ C_2 = c_1, \ldots, c_l, \mathbf{E}: r_1, \ldots, r_m,\ \text{and such that}\ c_1, \ldots, c_l \in Bel(\boldsymbol{P}_{i-1})\ \text{and}\ r_1, \ldots, r_m \in E_i\ \}.$$

The update statements in $\mathcal{G}(\mathcal{U})^i$ are thus of the form $cmd_1(\rho_1)$ **if** [**not**] $cmd_2(\rho_2), \ldots,$ [**not**] $cmd_m(\rho_m)$. Semantically, we interpret them as ordinary logic program rules

$cmd_1(\rho_1) \leftarrow [not\,]cmd_2(\rho_2), \ldots, [not\,]cmd_m(\rho_m)$. The program $\Pi_i^{\mathcal{U}}$ is the collection of all these rules, given $\mathcal{G}(\mathcal{U})^i$, together with the following constraints, which exclude contradictory commands:

$$\leftarrow \mathbf{assert[_event]}(R), \mathbf{retract[_event]}(R);$$
$$\leftarrow \mathbf{always[_event]}(R), \mathbf{cancel}(R).$$

Definition 4 *Let $KS = \langle KB; E_1, \ldots, E_n \rangle$ be a knowledge state and $\mathcal{U}$ an update policy. Then, U_i is a set of* executable update commands *at step i ($i \leq n$) iff U_i is an answer set of the grounding $\mathcal{G}(\Pi_i^{\mathcal{U}})$ of $\Pi_i^{\mathcal{U}}$.*

Since update statements do not contain strong negation, executable update commands are actually *stable models* of $\mathcal{G}(\Pi_i^{\mathcal{U}})$ [Gelfond and Lifschitz, 1988]. Furthermore, since programs may in general have more than one answer set, or no answer set at all, and the agent must commit itself to a single set of update commands, we assume a suitable *selection function*, $Sel(\cdot)$, returning a particular U_i if an answer set exists, or, otherwise, returning $U_i = \{\mathbf{assert}(\bot_i \leftarrow)\}$, where $\bot_i$ is a reserved atom. These atoms are used for signaling that the update policy encountered inconsistency. They can easily be filtered out from $Bel(\cdot)$, if needed, restricting the outcomes of the update to the original language.

Next we compile the executable commands $U_1, \ldots, U_i$ into an update sequence $(P_0, \ldots, P_i)$, serving as input for the belief function $Bel(\cdot)$. This is realized by means of a transformation $tr(\cdot)$, which is a generic and adapted version of a similar mapping introduced by Alferes *et al.* [1999a]. In what follows, we assume a suitable naming function for rules in the update sequence, enforcing that each rule r is associated with a unique name n_r.

Definition 5 *Let $KS = \langle KB; E_1, \ldots, E_n \rangle$ be a knowledge state and $\mathcal{U}$ an update policy. Then, for $i \geq 0$, $tr(KB; U_1, \ldots, U_i) = (P_0, P_1, \ldots, P_i)$ is inductively defined as follows, where $U_1, \ldots, U_i$ are the executable commands according to Definition 4:*

$i = 0$: *Set $P_0 = \{H(r) \leftarrow B(r), on(n_r) \mid r \in KB\} \cup \{on(n_r) \leftarrow \mid r \in KB\}$, where $on(\cdot)$ are new atoms. Furthermore, initialize the sets PC_0 of* persistent commands *and EC_0 of* effective commands *to $\emptyset$.*

$i \geq 1$: *EC_i, PC_i and P_i are as follows:*

$$\begin{aligned}
EC_i &= \{cmd(r) \mid cmd(r) \in U_i \wedge \mathbf{ignore}(r) \notin U_i\};\\
PC_i &= PC_{i-1} \cup \{\mathbf{always}(r) \mid \mathbf{always}(r) \in EC_i\}\\
&\quad \cup \{\mathbf{always_event}(r) \mid \mathbf{always_event}(r) \in EC_i\\
&\qquad \wedge \mathbf{always}(r) \notin EC_i \cup PC_{i-1}\}\\
&\quad \setminus (\{\mathbf{always_event}(r) \mid \mathbf{always}(r) \in EC_i\}\\
&\qquad \cup \{\mathbf{always[_event]}(r) \mid \mathbf{cancel}(r) \in EC_i\});\\
P_i &= \{on(n_r) \leftarrow\ , H(r) \leftarrow B(r), on(n_r) \mid\\
&\qquad \mathbf{assert[_event]}(r) \in EC_i\\
&\qquad \vee \mathbf{always[_event]}(r) \in PC_i\}\\
&\quad \cup \{on(n_r) \leftarrow \mid \mathbf{retract_event}(r) \in EC_{i-1}\\
&\qquad \wedge \mathbf{retract[_event]}(r) \notin EC_i\}\\
&\quad \cup \{\neg on(n_r) \leftarrow \mid (\mathbf{retract[_event]}(r) \in EC_i\\
&\qquad \wedge \mathbf{always[_event]}(r) \notin PC_i)\\
&\qquad \vee (\mathbf{always_event}(r) \in PC_{i-1}\\
&\qquad\quad \wedge\ \mathbf{cancel}(r) \in EC_i\\
&\qquad\quad \wedge\ \mathbf{assert[_event]}(r) \notin EC_i)\\
&\qquad \vee (\mathbf{assert_event}(r) \in EC_{i-1}\\
&\qquad\quad \wedge\ \mathbf{always[_event]}(r) \notin PC_i\\
&\qquad\qquad \wedge \mathbf{assert[_event]}(r) \notin EC_i)\}.
\end{aligned}$$

On the basis of this compilation, we can define the belief set for a knowledge state KS:

Definition 6 *Let KS and $\mathcal{U}$ be as in Definition 5, and let $U_1, \ldots, U_n$ be the corresponding executable commands obtained from Definition 4. Then, the* belief set *of KS is given by $Bel(KS) = Bel(tr(KB; U_1, \ldots, U_n))$.*

Example 3 *Reconsider Example 2 and suppose the event $E_1 = \{sale(s_0),\ date(1)\}$ occurs at $KS = \langle KB \rangle$. Then,*

$$\begin{aligned}
\mathcal{G}(\mathcal{U})^1 = \{&\mathbf{assert}(sale(s_0))\ \mathbf{if\ not\ ignore}(sale(s_0)),\\
&\mathbf{assert}(date(1))\ \mathbf{if\ not\ ignore}(date(1)),\\
&\mathbf{retract}(date(0))\}.
\end{aligned}$$

The corresponding program $\Pi_1^{\mathcal{U}}$ has the single answer set $\{\mathbf{assert}(sale(s_0)),\ \mathbf{assert}(date(1)),\ \mathbf{retract}(date(0))\}$, which is compiled, via function $tr(\cdot)$, to $PC_1 = PC_0 \setminus \{\mathbf{assert_[event]}(date(0))\} = \emptyset$ and $P_1 = \{sale(s_0) \leftarrow on(r_1');\ on(r_1') \leftarrow\ ;\ date(1) \leftarrow on(r_2');\ on(r_2') \leftarrow\ ;\ \neg on(r_0) \leftarrow\}$. As easily seen, the belief set $Bel(\langle KB; E_1 \rangle)$ $= Bel((P_0, P_1))$ contains $sale(s_0)$ and $query(s_0)$.

4 Properties

In this section, we discuss some properties of $Bel(KS)$ for particular update policies, using the definition of $Bel(\cdot)$ based on the update answer sets approach of Eiter *et al.* [2000], as explained in Section 2. We stress that the properties given below are also satisfied by similar instantiations of $Bel(\cdot)$, like, e.g., dynamic logic programming [Alferes *et al.*, 2000].

First, we note some basic properties:

- If $\mathcal{U} = \emptyset$ (called *empty policy*), then KB will never be updated; the belief set is independent of $E_1, \ldots, E_n$, and thus *static*. Hence, $Bel(KS_i) = Bel(KB)$, for each $i = 1, \ldots, n$.
- If $\mathcal{U} = \{\mathbf{assert}(R)[\![\mathbf{E} : R]\!]\}$ (called *unconditional assert policy*), then all rules contained in the received events are directly incorporated into the update sequence. Thus, $Bel(KS_i) = Bel((KB, E_1, \ldots, E_i))$, for each $i = 1, \ldots, n$.
- If U_i is empty, then the knowledge is not updated, i.e., $P_i = \emptyset$. We thus have $Bel(KS_i) = Bel(KS_{i-1})$.
- Similarly, if $U_i = \{\mathbf{assert}(\bot_i) \leftarrow\}$, then $Bel(KS_i) = Bel(KS_{i-1})$.

Physical removal of rules

An important issue is the growth of the agent's knowledge base, as the modular construction of the update sequence through transformation $tr(\cdot)$ causes some rules and facts to be repeatedly inserted. This is addressed next, where we discuss the physical removal of rules from the knowledge base.

Lemma 1 *Let $\boldsymbol{P} = (P_0, \ldots, P_n)$ be an update sequence. For every $r \in P_i, r' \in P_j$ with $i < j$, the following holds: if (i) $r = r'$, or (ii) $r = L \leftarrow$ and $r' = \neg L \leftarrow$, or (iii) $r' = L \leftarrow$ such that no rule $r'' \in P_k$ with $H(r'') = \neg L$ exists, where $k \in \{j+1, \ldots, n\}$, and $\neg L \in B(r)$, then $Bel_{\mathcal{A}}(\boldsymbol{P}) = Bel_{\mathcal{A}}(P_0, \ldots, P_{i-1}, P_i \setminus \{r\}, P_{i+1}, \ldots, P_n)$.*

The following property holds:

Theorem 1 *Let KS be a knowledge state and $Bel(KS) = Bel(\boldsymbol{P})$, where $\boldsymbol{P} = (P_0, \ldots, P_n)$. Furthermore, let $\boldsymbol{P}^*$ result from $\boldsymbol{P}$ after repeatedly removing rules as in Lemma 1, and let $\boldsymbol{P}^- = (P_0^-, \ldots, P_n^-)$, where*

$$P_i^- = \{H(r) \leftarrow B(r) \setminus \{on(n_r)\} \mid r \in \boldsymbol{P}_i^*,\ on(n_r) \leftarrow \in \boldsymbol{P}^*\} \setminus \{on(n_s) \leftarrow \mid on(n_s) \leftarrow \in \boldsymbol{P}\}.$$

Then, $Bel_{\mathcal{A}}(KS) = Bel_{\mathcal{A}}(\boldsymbol{P}^-)$.

Thus, we can purge the knowledge base and remove duplicates of rules, as well as all deactivated (retracted) rules.

History Contraction

Another relevant issue is the possibility, for some special case, to *contract the agent's update history*, and compute its belief set at step i merely based on information at step $i-1$. Let us call $\mathcal{U}$ a *factual assert policy* if all **assert**[_**event**] and **always**[_**event**] statements in $\mathcal{U}$ involve only facts. In this case, the compilation $tr(\cdot)$ for a knowledge state $KS = \langle KB; E_1, \ldots, E_n \rangle$ can be simplified thus: (1) $P_0 = KB$, and (2) the construction of each P_i involves facts $L \leftarrow$ and $\neg L \leftarrow$ instead of $on(n_r) \leftarrow$ and $\neg on(n_r) \leftarrow$, respectively.

For such sequences, the following holds:

Lemma 2 *Let $\boldsymbol{P} = (P_0, \ldots, P_n)$ be an update sequence such that P_i contains only facts, for $1 \leq i \leq n$. Then, $Bel_{\mathcal{A}}(\boldsymbol{P}) = Bel_{\mathcal{A}}(P_0, P_{u_n})$, where $P_{u_1} = P_1$, and $P_{u_{i+1}} = P_{i+1} \cup (P_{u_i} \setminus \{L \leftarrow \mid \neg L \leftarrow \in P_{i+1}\})$.*

We can thus assert the following proposition for history contraction:

Theorem 2 *Let $\mathcal{U}$ be a factual assert policy and $\boldsymbol{P} = (P_1, \ldots, P_n)$ be the compiled sequence obtained from KS by the simplified method described above. Then, $Bel_{\mathcal{A}}(KS) = Bel_{\mathcal{A}}((KB, P_{u_n}))$, where P_{u_n} is as in Lemma 2.*

Simple examples show that Theorem 2 does not hold in general. The investigation of classes of policies for which similar results hold are a subject for further research.

Computational Complexity

Finally, we briefly address the complexity of reasoning about a knowledge state KS. An update policy $\mathcal{U}$ is called *stratified* iff, for all EPI statements $cmd(\rho)$ **if** $C_1[\![C_2]\!] \in \mathcal{U}$, the associated rules $cmd_1(\rho) \leftarrow C_1'$ form a stratified logic program, where C_1' results from C_1 by replacing the EPI declaration **not** by default negation *not* .

For stratified $\mathcal{U}$, any $\Pi_i^{\mathcal{U}}$ has at most one answer set. Thus, the selection function $Sel(\cdot)$ is redundant. Otherwise, the complexity cost of $Sel(\cdot)$ must be taken into account. If $Sel(\cdot)$ is unknown, we consider all possible return values (i.e., all answer sets of $\Pi_i^{\mathcal{U}}$) and thus, in a cautious reasoning mode, all possible $Bel(KS) = Bel((P_0, \ldots, P_n))$ from Figure 1. Clearly, for update answer sets, deciding $r' \in Bel((Q_0, \ldots, Q_m))$ is in coNP; it is polynomial, if Q_0 is stratified and each Q_i, $1 \leq i \leq m$, contains only facts.

Theorem 3 *Let $Bel(\cdot)$ be the update answer set semantics, and $Sel(\cdot)$ polynomial-time computable with an* NP *oracle. Then, given a ground rule r and ground $KS = \langle KB; E_1, \ldots, E_n \rangle$, the complexity of deciding whether $r \in Bel(KS)$ is as follows (entries denote completeness results; the case of unknown $Sel(\cdot)$ is given at the right of "/"):*

$KB \setminus \mathcal{U}$	fact. assert & strat.	stratified	general
stratified	P	P^{NP}	$\mathrm{P}^{\mathrm{NP}}/\Pi_2^P$
general	P^{NP}	P^{NP}	$\mathrm{P}^{\mathrm{NP}}/\Pi_2^P$

Similar results hold, e.g., for dynamic logic programming.

The results can be intuitively explained as follows. Each U_i and P_i as in Figure 1 can be computed iteratively ($1 \leq i \leq n$), where at step i polynomially many problems $r' \in Bel((P_0, \ldots, P_{i-1}))$ must be solved to construct $\Pi_i^{\mathcal{U}}$. From $U_i = Sel(\Pi_i^{\mathcal{U}})$ and previous results, P_i is easily computed in polynomial time. Since P_i contains less than $|\mathcal{U}|$ rules, step i is feasible in polynomial time with an NP oracle. Thus, $\boldsymbol{P} = (P_0, \ldots, P_n)$ is polynomially computable with an NP oracle, and $r \in Bel(\boldsymbol{P})$ is decided with another oracle call. Updating a stratified P_0 such that only sets of facts P_1, $P_2, \ldots$ may be added preserves polynomial decidability of $r' \in Bel((P_0, \ldots, P_{i-1}))$; this explains the polynomial decidability result. In all other cases, P^{NP}-hard problems such as computing the lexicographically maximal model of a CNF formula are easily reduced to the problem.

If $Sel(\cdot)$ is unknown, each possible result of $Sel(\Pi_i^{\mathcal{U}})$ can be nondeterministically guessed and verified in polynomial time. This leads to $\mathrm{coNP}^{\mathrm{NP}} = \Pi_2^P$ complexity.

5 Implementational Issues

An elegant and straightforward realization of update policies is possible through IMPACT agent programs. IMPACT [Subrahmanian *et al.*, 2000] is a platform for developing software agents, which allows to build agents on top of legacy code, i.e., existing software packages, that operates on arbitrary data structures. Thus, in accordance with our approach, we can design a *generic implementation* of our framework, without committing ourselves to a particular update semantics $Bel(\cdot)$.

Since every update policy $\mathcal{U}$ is semantically reduced to a logic program, the corresponding executable commands can be computed using well-known logic programming engines like `smodels`, `DLV`, or `DeRes`. Hence, we may assume that a software package, $\mathcal{SP}$, for updating and querying a knowledge base KB is available, and that KB can be accessed through a function `bel()` returning the current belief set $Bel(KS)$. Moreover, we assume that $\mathcal{SP}$ has a function `event()`, which lists all rules of a current event. Then, an update policy $\mathcal{U}$ can be represented in IMPACT as follows.

(1) Conditions on the belief set and the event can be modeled by IMPACT *code call atoms*, i.e. atoms $\mathtt{in}(\mathtt{t}, \mathtt{bel}())$, $\mathtt{not_in}(\mathtt{t}, \mathtt{bel}())$, and $\mathtt{in}(\mathtt{t}, \mathtt{event}())$, where `t` is a constant r or a variable R. In IMPACT, $\mathtt{in}(r, \mathtt{f}())$ is true if constant r is in the result returned by `f()`; a variable R is bound to all r such that $\mathtt{in}(r, \mathtt{f}())$ is true; "`not_in`" is negation.

(2) Update commands can be easily represented as IMPACT *actions*. An action is implemented by a body of code in any programming language (e.g., C); its effects are specified in terms of add and delete lists (sets of code call atoms). Thus, actions like $\mathbf{assert}(R)$, $\mathbf{retract}(R)$, etc., where R is a parameter, are introduced.

(3) EPI statements are represented as IMPACT action rules

$$\texttt{Do}(cmd_1(\rho_1)) \leftarrow [\neg]\texttt{Do}(cmd_2(\rho')), \ldots, [\neg]\texttt{Do}(cmd_m(\rho')), \\ code_call_atoms(\texttt{cond}),$$

where $code_call_atoms(\texttt{cond})$ is the list of the code call atoms for the conditions on the belief set and the event in cond as described above.

The semantics of IMPACT agent programs is defined through *status sets*. A *reasonable status set* S is equivalent to a stable model of a logic program, and prescribes the agent to perform all actions α where $\texttt{Do}(\alpha)$ is in S. Thus, S represents the executable commands U_i of Figure 1 in accord with $\mathcal{U}$, and the respective action execution affects the computation of P_i via $tr(\cdot)$. For more details, cf. [Eiter *et al.*, 2001].

6 Related Work and Conclusion

Our approach is similar in spirit to the work in active databases (ADBs), where the dynamics of a database is specified through *event-condition-action* (*ECA*) *rules* triggered by events. However, ADBs have in general no declarative semantics, and only one rule at a time fires, possibly causing successive events. In [Baral and Lobo, 1996], a declarative characterization of ADBs is given, in terms of a reduction to logic programs, by using situation calculus notation.

Our language for update policies is also related to *action languages*, which can be compiled to logic programs as well (cf., e.g., [Lifschitz and Turner, 1999]). A change to the knowledge base may be considered as an action, where the execution of actions may depend on other actions and conditions. However, action languages are tailored for planning and reasoning about actions, rather than reactive behavior specification; events would have to be emulated. Furthermore, a state is, essentially, a set of literals rather than a belief set as we define it. Investigating the relationships of our framework to these languages in detail—in particular concerning embeddings—is an interesting issue for further research.

A development in the area of action languages, with purposes similar to those of EPI, is the policy description language $\mathcal{PDL}$ [Lobo *et al.*, 1999]. It extends traditional action languages with the notion of *event sequences*, and serves for specifying actions as reactive behavior in response to events. A $\mathcal{PDL}$ policy is a collection of ECA rules, interpreted as a function associating with an event sequence a set of actions. $\mathcal{PDL}$ seems thus to be more expressive than EPI; possible embeddings of EPI into $\mathcal{PDL}$ remain to be explored.

The EPI language could be extended with several features:

(1) Special atoms $\mathbf{in}(r)$ telling whether r is actually part of KB (i.e., activated by $on(n_r)$), allowing to access the "extensional" part of KB.

(2) Rule terms involving literal constants and variables, e.g., "$H \leftarrow up(s_1), B$", where H, B are variables and $up(s_1)$ is a fixed atom, providing access to the structure of rules. Combined with (1), commands such as "remove all rules involving $up(s_1)$" can thus be conveniently expressed.

(3) More expressive conditions on the knowledge base are conceivable, requesting for more complex reasoning tasks, and possibly taking the temporal evolution into account. E.g., "$\mathbf{prev}(a)$" expressing that a was true at the previous stage.

In concluding, our generic framework, which extends other approaches to logic program updates, represents a convenient platform for declarative update specifications and could also be fruitfully used in several applications. Exploring these issues is part of our ongoing research.

Acknowledgements

This work was partially supported by the Austrian Science Fund (FWF) under grants P13871-INF and N Z29-INF.

References

[Alferes *et al.*, 1999a] J. Alferes, L. Pereira, H. Przymusinska, and T. Przymusinski. LUPS - A language for updating logic programs. In *Proc. LPNMR'99*, LNAI 1730, pp. 162-176. Springer, 1999.

[Alferes *et al.*, 2000] J. Alferes, J. Leite, L. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic updates of nonmonotonic knowledge bases. *J. Logic Programming*, 45(1-3):43-70, 2000.

[Baral and Lobo, 1996] C. Baral and J. Lobo. Formal characterization of active databases. In *Proc. LID'96*, LNCS 1154, pp. 175-195. Springer, 1996.

[Buccafurri *et al.*, 1999] F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. In *Proc. ICLP'99*, pp. 79-93. MIT Press, 1999.

[Eiter *et al.*, 2000] Th. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Considerations on updates of logic programs. In *Proc. JELIA'00*, LNAI 1919, pp. 2-20. Springer, 2000.

[Eiter *et al.*, 2001] Th. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Declarative knowledge updates through agents. In *Proc. AISB'01 Symp. on Adaptive Agents and Multi-Agent Systems, York, UK*, pp. 79-84. AISB, 2001.

[Gelfond and Lifschitz, 1988] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. ICSLP'88*, pp. 1070-1080. MIT Press, 1988.

[Gelfond and Lifschitz, 1991] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365-386, 1991.

[Inoue and Sakama, 1999] K. Inoue and C. Sakama. Updating extended logic programs through abduction. In *Proc. LPNMR'99*, LNAI 1730, pp. 147-161. Springer, 1999.

[Lifschitz and Turner, 1999] V. Lifschitz and H. Turner. Representing transition systems by logic programs. In *Proc. LPNMR'99*, LNAI 1730, pp. 92-106. Springer, 1999.

[Lobo *et al.*, 1999] J. Lobo, R. Bhatia, and S. Naqvi. A policy description language. In *Proc. AAAI/IAAI'99*, pp. 291-298. AAAI Press / MIT Press, 1999.

[Marek and Truszczyński, 1998] W. Marek and M. Truszczyński. Revision programming. *Theoretical Computer Science*, 190:241-277, 1998.

[Subrahmanian *et al.*, 2000] V.S. Subrahmanian, J. Dix, Th. Eiter, P. Bonatti, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems*. MIT Press, 2000.

[Williams and Antoniou, 1998] M.-A. Williams and G. Antoniou. A strategy for revising default theory extensions. In *Proc. KR'98*, pp. 24-35. Morgan Kaufmann, 1998.

[Zhang and Foo, 1998] Y. Zhang and N. Foo. Updating logic programs. In *Proc. ECAI'98*, pp. 403-407. 1998.

Abduction in Logic Programming: A New Definition and an Abductive Procedure Based on Rewriting

Fangzhen Lin
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Jia-Huai You
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1

Abstract

We propose a new definition of abduction in logic programming, and contrast it with that of Kakas and Mancarella's. We then introduce a rewriting system for answering queries and generating explanations, and show that it is both sound and complete under the partial stable model semantics and sound and complete under the answer set semantics when the underlying program is so-called odd-loop free. We discuss an application of the work to a problem in reasoning about actions and provide some experimental results.

1 Abduction in logic programming

In general, given a background theory T, and an observation q to explain, an abduction of q w.r.t. T is a theory Π such that $\Pi \cup T \models q$. Normally, we want to put some additional conditions on Π, such as that it is consistent with T and contains only those propositions called abducibles. For instance, in propositional logic, given a background theory T, a set A of assumptions or abducibles, and a proposition q, an explanation S of q is commonly defined (see [Reiter and de Kleer, 1987], [Poole, 1988], and [Konolige, 1992]) to be a minimal set of literals over A such that $T \cup S \models q$ and $T \cup S$ is consistent.

In the context of logic programming, abduction has been investigated from both proof-theoretic and model-theoretic perspectives (e.g. [Eshghi and Kowalski, 1989; Kakas and Mancarella, 1990; Satoh and Iwayama, 1992]). One of the most influential definitions of abduction in logic programming is that of Kakas and Mancarella's generalized stable model semantics [1990]. Given a logic program P, a set A of atoms standing for abducibles, and a query q, Kakas and Mancarella define an abductive explanation S to be a subset of A such that there is an answer set (also called a stable model) M of $P \cup S$ that satisfies q.

One can see the following two differences between this definition and the one that we defined above for propositional logic: In propositional logic, S is a set of literals, but in logic programming, it is just a set of atoms; In propositional logic, S must be minimal, in terms of the subset ordering relation; but there is no such requirement in the case of logic programming.

One could argue that these differences are due to the fact that under the answer set semantics, negation is considered to be "negation-as-failure." If none of the atoms in A appear in the head of a rule in the logic program P, then adding a set $S \subseteq A$ to P really means that we are adding the complete literal set, $S \cup \{\neg p \mid p \in A, p \notin S\}$, to P. This would also explain why there is no minimality condition in the definition: two complete sets of literals are never comparable in terms of the subset relation.

However, while this notion of abductive explanations makes sense in theory, it is problematic in practice. For instance, if $A = \{a, b\}$, and $P = \{q \leftarrow a.\}$, then there are two abductive explanations for q according to Kakas and Mancarella's definition: $\{a\}$ and $\{a, b\}$. In general, if A has n elements, then there are 2^{n-1} abductive explanations for q, a number that is too big to manage.

Since in this case a is the explanation that we are looking for, it is tempting here to say that we should prefer minimal abductive explanations like what we did for propositional logic. As we mentioned above, this does not make sense if we take an abductive explanation to be a complete set of literals as implied by the answer set semantics. However, one can still try to minimize the set of atoms, in this case, preferring $\{a\}$ over $\{a, b\}$.

However, this minimization strategy is problematic when a program contains negation. Consider a situation in which a boat can be used to cross a river if it is not leaking or, if it is leaking, there is a bucket available to scoop the water out of the boat. This can be axiomatized by the following logic program P:

$$canCross \leftarrow boat, \mathsf{not}\, leaking.$$
$$canCross \leftarrow boat, leaking, hasBucket.$$

Now suppose that we saw someone crossed the river, how do we explain that? Clearly, there are two possible explanations: either the boat is not leaking or the person has a bucket with her. In terms of Kakas and Mancarella's definition, there are three abductive explanations for $canCross$, $\{boat\}$, $\{boat, hasBucket\}$, and $\{boat, leaking, hasBucket\}$, assuming that $A = \{boat, leaking, hasBucket\}$ is the set of abducibles. But only one of them, $\{boat\}$, is minimal.

On a closer look, we see that in our first example, when we say that $\{a\}$ is a preferred explanation over all the others, we do not mean the complete set of literals $\{a, \neg b\}$, is preferred

over all the others. While we want a to be part of the explanation, we don't necessarily want $\neg b$ because we do not want to apply negation as failure on abducibles, which are assumptions one can make one way or the other. What we want is for the set $\{a\}$ itself to be the best explanation for q.

One way to justify this is that all possible ways of completing this set into a complete set of literals, $\{a, \neg b\}$ and $\{a, b\}$, turn out to correspond to all the abductive explanations of q according to Kakas and Mancarella's definition. The same kind of justification turns out to work for our second example as well: the reason that $\{boat\}$ is not a preferred explanation is that while its completion according to negation-as-failure, $\{boat, \neg leaking, \neg hasBucket\}$ is an explanation, some of its other completions, for example $\{boat, leaking, \neg hasBucket\}$ is not an explanation. This motivates our following definitions.

2 Abduction in logic programming revisited

In this paper, we consider *(normal) logic programs* which are sets of rules of the form

$$a \leftarrow b_1, ..., b_m, \mathsf{not}\, c_1, ..., \mathsf{not}\, c_n$$

where a, b_i and c_i are atoms of the underlying propositional language L. Here "not" is the so-called *default negation*, and defined according to the answer set semantics [Gelfond and Lifschitz, 1988].

Let P be a logic program, A a set of propositions standing for abducibles, and q a proposition. In the following, without loss of generality, we shall assume that none of the abducibles in A occur in the head of a rule in P.[1]

In the following, by a *hypothesis* α we mean a consistent set of literals over A, i.e. it is not the case that p and $\neg p$ are both in α for some $p \in A$. We say that a hypothesis is *complete* if for each atom $p \in A$, either p or $\neg p$ is in α, but not both. Notice that a complete hypothesis is really a truth-value assignment over the language A. We say that a hypothesis α is an *extension* of another one β if $\beta \subseteq \alpha$, and a complete extension is an extension that is complete.

Definition 2.1 *A complete hypothesis α is said to be an explanation of q w.r.t. P and A iff there is an answer set M of $P \cup \alpha^+$ such that M contains q and for any $\neg p \in \alpha$, $p \notin M$, where α^+ is the set of atoms in α.*

Definition 2.2 *A hypothesis is said to be an explanation of q iff every complete extension of it is an explanation. A hypothesis α is said to be a minimal explanation if it is an explanation, and there is no other explanation α' such that $\alpha' \subset \alpha$.*

Consider the logic program P in the previous section about $canCross$. The following are the complete hypotheses that explain $canCross$:

$\{boat, \neg leaking, \neg hasBucket\}$,

$\{boat, \neg leaking, hasBucket\}$, $\{boat, leaking, hasBucket\}$.

Now consider $\{boat, hasBucket\}$. Clearly every complete extension of this set is an explanation, so it is an explanation as well. Furthermore, it is a minimal explanation as none of its element can be deleted for it continue to be an explanation. Similarly, $\{boat, \neg leaking\}$ is also a minimal explanation.

If we take a hypothesis to be the conjunction of its elements, then we have that in the propositional logic,

$$\bigvee_{\alpha \in \mathcal{S}_1} \alpha \equiv \bigvee_{\alpha \in \mathcal{S}_2} \alpha \equiv \bigvee_{\alpha \in \mathcal{S}_3} \alpha$$

where $\mathcal{S}_1$ is the set of all complete hypotheses that are explanations of q, $\mathcal{S}_2$ the set of all explanations of q, and $\mathcal{S}_3$ the set of all minimal explanations of q. Therefore the set of minimal explanations is a succinct representation of the set of all explanations.

It is clear from our definition that a complete hypothesis α is an explanation of q iff α^+ is an abductive explanation of q according to Kakas and Mancarella's definition. This implies that if none of the abducibles occur in the head of any clauses in P, then

$$\bigvee_{S \in \mathcal{S}_1} cl(S) \equiv \bigvee_{\alpha \in \mathcal{S}_2} \alpha,$$

where $\mathcal{S}_1$ is the set of all abductive explanation of q according to Kakas and Mancarella's definition, $cl(S) = S \cup \{\neg p \mid p \in A, p \notin S\}$, and $\mathcal{S}_2$ is the set of all minimal explanations of q.

So in a sense, the set of Kakas and Mancarella's abductive explanations and that of our minimal explanations are equivalent. However, as we have seen above, the number of abductive explanations can be very large. Enumerating them all is impossible even in simple, small domains. In contrast, the number of minimal explanations are much smaller. More importantly, just like explanations in propositional logic, they only include "relevant propositions."

But computationally, it may be hard to compute minimal explanations from scratch. It is often easier to compute first a small "cover" of all explanations.

Definition 2.3 *A set $\mathcal{S}$ of hypotheses is said to be a* cover *of q w.r.t. P and A iff*

$$\bigvee_{\alpha \in \mathcal{S}} \alpha \equiv \bigvee_{\alpha \in \mathcal{S}_0} \alpha,$$

where $\mathcal{S}_0$ is the set of minimal explanations of q.

Proposition 2.4 *If $\mathcal{S}$ is a cover of q, then each $\alpha \in \mathcal{S}$ must be an explanation of q.*

So a cover is a set of explanations such that any complete explanation must be an extension of one of the explanations in the cover. Once we have a cover, then we can find all minimal explanations by propositional reasoning alone. Recall that a conjunction of literals α is a *prime implicant* of a formula φ if $\alpha \models \varphi$, and there is no other β such that $\beta \models \varphi$ and β is a subset of α, i.e. α is a minimal conjunction of literals that entails φ.

Proposition 2.5 *Let $\mathcal{S}$ be a cover of q. Then a hypothesis is a minimal explanation of q iff it is a prime implicant of $\bigvee_{\alpha \in \mathcal{S}} \alpha$.*

In the rest of this paper, we shall propose a rewriting system for generating explanations of a proposition in a logic program. We shall first define it for logic programs without abducibles. We will then extend the system to logic programs

[1] If $p \in A$ occurs in the head of a rule, then we can always introduce a new proposition, say p', add the rule $p \leftarrow p'$ to P, add p' to A and delete p from A.

with abducibles, and show that for any query, the rewriting system generates an approximation of a cover set in the general case, and exactly a cover set when the given logic program has no so-called "odd loops." We will then discuss an application of our system to reasoning about actions and present some experimental results.

3 Goal Rewrite Systems

Given a (ground) program P, the *Clark completion* of P, denoted $Comp(P)$, is the following set of equivalences: for each atom $\phi \in L$,

- if ϕ does not appear as the head of any rule in P, $\phi \leftrightarrow F \in Comp(P)$.
- otherwise, $\phi \leftrightarrow B_1 \vee ... \vee B_n \in Comp(P)$ (with default negations replaced by negative literals), if there are exactly n rules $\phi \leftarrow B_i \in P$ with ϕ as the head. We write T for B_i if B_i is empty.

The idea of goal rewriting is simple. A completed definition $\phi \leftrightarrow B_1 \vee \ldots \vee B_n \in Comp(P)$ can be used as a rewrite rule from left to right: ϕ is rewritten to $B_1 \vee \ldots \vee B_n$, and $\neg\phi$ to $\neg B_1 \wedge \ldots \wedge \neg B_n$. We call these *literal rewriting*, and the completed definitions *program (rewrite) rules*.

A goal is a formula which may involve $\neg$, $\vee$ and $\wedge$. A goal is also referred to as a *goal formula*. A goal resulted from a literal rewriting from another goal is called a *derived goal*. A goal with negation appearing only in front of atoms is said to be *signed*, a term introduced in [Kunen, 1989] for a similar purpose. For convenience, we assume that *all goals are signed*, which can be achieved easily by simple transformations using the following rules: for any formulas Φ and Ψ,

$$\neg\neg\Phi \rightarrow \Phi$$
$$\neg(\Phi \vee \Psi) \rightarrow \neg\Phi \wedge \neg\Psi$$
$$\neg(\Phi \wedge \Psi) \rightarrow \neg\Phi \vee \neg\Psi$$

Like a formula, a goal may be further transformed to a suitable form for literal rewriting without changing its semantics. With a mechanism of loop handling, rewriting of a goal Q terminates at either T meaning that Q is proved, or F meaning that Q is not proved. Hence, a goal rewrite system consists of three types of rewrite rules: (i) program rules from $Comp(P)$ for literal rewriting, (ii) simplification rules to transform and simplify goals, and (iii) loop rules for handling loops.

3.1 Simplification rules

The simplification subsystem is formulated with a mechanism of loop handling in mind, which requires keeping track of literal sequences $g_0, ..., g_n$ where each g_i, $0 < i \leq n$, is in the goal formula resulted from rewriting g_{i-1}. Two central mechanisms in formalizing goal rewrite systems are *rewrite chains* and *contexts*.

- *Rewrite Chain:* Suppose a literal l is written by its definition $\phi \leftrightarrow \Phi$ where $l = \phi$ or $l = \neg\phi$. Then, each literal l' in the derived goal is generated in order to prove l. This ancestor-descendant relation is denoted $l \prec l'$. A sequence $l_1 \prec \ldots \prec l_n$ is then called a *rewrite chain*, abbreviated as $l_1 \prec^+ l_n$. Notice that it is essential here that any goal G be in the form of a signed goal, and that when $\neg p$ is in G, we have that $l \prec \neg p$ but not $l \prec p$.
- *Context:* A rewrite chain $g = g_0 \prec g_1 \prec \ldots \prec g_n = T$ records a set of literals $C = \{g_0, ..., g_{n-1}\}$ for proving g. We will write $T(\{g_0, ..., g_{n-1}\})$ and call C a *context*.

  For simplicity, we assume that whenever $\neg F$ is generated, it is automatically replaced by $T(C)$, where C is the set of literals on the corresponding rewrite chain, and $\neg T$ is automatically replaced by F.

Note that for every literal in any derived goal, the rewrite chain leading to it from a literal in the given goal is uniquely determined. As an example, suppose the completion of a program is: $\{a \leftrightarrow \neg b \wedge \neg c, b \leftrightarrow q \vee \neg p\}$. We then have a rewrite sequence $a \rightarrow \neg b \wedge \neg c \rightarrow \neg q \wedge p \wedge \neg c$. For the three literals in the last goal, we have the following rewrite chains for them from a: $a \prec \neg b \prec \neg q$, $a \prec \neg b \prec p$, and $a \prec \neg c$.

Simplification Rules:

Let Φ_i's be any goal formulas, C a context, and l a literal.

SR1. $F \vee \Phi \rightarrow \Phi$

SR1'. $\Phi \vee F \rightarrow \Phi$

SR2. $F \wedge \Phi \rightarrow F$

SR2'. $\Phi \wedge F \rightarrow F$

SR3. $T(C_1) \wedge T(C_2) \rightarrow T(C_1 \cup C_2)$ if $C_1 \cup C_2$ is consistent

SR4. $T(C_1) \wedge T(C_2) \rightarrow F$ if $C_1 \cup C_2$ is inconsistent

SR5. $T(C) \wedge l \rightarrow F$ if $\neg l \in C$

SR5'. $l \wedge T(C) \rightarrow F$ if $\neg l \in C$

SR6. $\Phi_1 \wedge (\Phi_2 \vee \Phi_3) \rightarrow (\Phi_1 \wedge \Phi_2) \vee (\Phi_1 \wedge \Phi_3)$

SR6'. $(\Phi_1 \vee \Phi_2) \wedge \Phi_3 \rightarrow (\Phi_1 \wedge \Phi_3) \vee (\Phi_2 \wedge \Phi_3)$ □

The simplification system is a nondeterministic transformation system. The primed version of a rule is its symmetric case. Most of the rules are about the logical equivalence between the two sides of a rule. SR3 merges two contexts if they are consistent, otherwise SR4 makes it a failure to prove. SR5 and SR5' prevent generating an inconsistent context before literal l is even proved.

Note that the proof-theoretic meaning of a goal formula may not be the same as the logical meaning of the formula. E.g., the goal formula $a \vee \neg a$ (a tautology in classic logic) could well lead to an F if neither a nor $\neg a$ can be proved.

For goal rewriting that does not involve loops, the system described so far is sufficient.

Example 3.1 *Let P be*

$$g \leftarrow \mathsf{not}\, a$$
$$a \leftarrow \mathsf{not}\, b, \mathsf{not}\, c; \quad a \leftarrow b, \mathsf{not}\, d$$
$$b \leftarrow q; \quad b \leftarrow \mathsf{not}\, p$$

Then $Comp(P)$ is:

$$g \leftrightarrow \neg a; \quad a \leftrightarrow (\neg b \wedge \neg c) \vee (b \wedge \neg d); \quad b \leftrightarrow q \vee \neg p$$
$$q \leftrightarrow F; \quad p \leftrightarrow F; \quad d \leftrightarrow F; \quad c \leftrightarrow F$$

The rewrite sequence below is generated by focusing on the left part of a goal.

$g \rightarrow \neg a$
$\rightarrow (b \vee c) \wedge (\neg b \vee d)$
$\rightarrow (q \vee \neg p \vee c) \wedge (\neg b \vee d)$
$\rightarrow (F \vee \neg p \vee c) \wedge (\neg b \vee d)$
$\rightarrow (\neg p \vee c) \wedge (\neg b \vee d)$
$\rightarrow (T(C) \vee c) \wedge (\neg b \vee d) \quad$ *where* $C = \{g, \neg a, b, \neg p\}$
$\rightarrow (T(C) \wedge (\neg b \vee d)) \vee (c \wedge (\neg b \vee d))$
$\rightarrow (T(C) \wedge \neg b) \vee (T(C) \wedge d) \vee (c \wedge (\neg b \vee d))$ %apply SR5
$\rightarrow F \vee (T(C) \wedge d) \vee (c \wedge (\neg b \vee d))$
$\rightarrow (T(C) \wedge d) \vee (c \wedge (\neg b \vee d))$
$\rightarrow (T(C) \wedge F) \vee (c \wedge (\neg b \vee d))$
$\rightarrow F \vee (c \wedge (\neg b \vee d))$
$\rightarrow c \wedge (\neg b \vee d) \rightarrow F \wedge (\neg b \vee d) \rightarrow F$

3.2 Loop rules

After a literal l is rewritten, it is possible that at some later stage either l or $\neg l$ appears again in a goal on the same rewrite chain. Thus, a loop is a rewrite chain $\{l_1, ..., l_n\}$ such that $l_1 = l_n$, or $l_1 = \neg l_n$. A loop analysis involves classifying all the cases of loops, and for each one, determining the outcome of a rewrite according the underlying semantics. For the problem at hand, there are only four cases.

Definition 3.2 *Let* $S = l_1 \prec^+ l_n$ *be a rewrite chain.*

- *If* $\neg l_1 = l_n$ *or* $l_1 = \neg l_n$, *then* S *is called an* odd loop.
- *If* $l_1 = l_n$, *then*
  - S *is called* a positive loop *if* l_1 *and* l_n *are both atoms and each literal on* $l_1 \prec^+ l_n$ *is also an atom;*
  - S *is called* a negative loop *if* l_1 *and* l_n *are both negative literals and each literal on* $l_1 \prec^+ l_n$ *is also negative;*
  - *Otherwise,* S *is called an* even loop.

In all the cases above, l_n *is called a* loop literal.

Note that when $l_1 = l_n$, and all of l_i, $1 \leq i \leq n$, have the same sign, this sign is either positive or negative, though both types of loops are caused by "positive loops" in the given program. These two types of loops must be treated differently according to the semantics.

It turns out that we only need two rewrite rules to handle all four cases.

Loop Rules: Let $g_1 \prec^+ g_n$ be a rewrite chain.

LR1. $g_n \rightarrow F$
if $g_i \prec^+ g_n$, for some $1 \leq i < n$, is a positive loop or an odd loop.

LR2. $g_n \rightarrow T(\{g_1, ..., g_n\})$
if $g_i \prec^+ g_n$, for some $1 \leq i < n$, is a negative loop or an even loop. □

Apparently, a loop literal should always be rewritten by a loop rule.

Example 3.3 $P_1 = \{b \leftarrow \mathsf{not}\, c;\ c \leftarrow c\}$. *Below,* b *is proved and* $\neg b$ *is not.*

$$b \rightarrow \neg c \rightarrow \neg c \rightarrow T(\{b, \neg c\});\quad \neg b \rightarrow c \rightarrow c \rightarrow F$$

$P_2 = \{d \leftarrow \mathsf{not}\, a;\ a \leftarrow \mathsf{not}\, b;\ b \leftarrow \mathsf{not}\, a\}$. *Both* d *and* $\neg d$ *can be proved.*

$$d \rightarrow \neg a \rightarrow b \rightarrow \neg a \rightarrow T(\{d, \neg a, b\})$$
$$\neg d \rightarrow a \rightarrow \neg b \rightarrow a \rightarrow T(\{\neg d, a, \neg b\})$$

$P_3 = \{a \leftarrow \mathsf{not}\, b;\ b \leftarrow \mathsf{not}\, b\}$. *Neither* a *nor* $\neg a$ *is proved:*

$$a \rightarrow \neg b \rightarrow b \rightarrow F;\quad \neg a \rightarrow b \rightarrow \neg b \rightarrow F$$

3.3 Goal rewrite systems and their properties

To summarize, a *rewrite sequence* is a sequence of zero or more rewrite steps $Q_0 \rightarrow \ldots \rightarrow Q_k$ (denoted $Q_0 \rightarrow^* Q_k$) such that Q_0 is an *initial goal* (one involving no context), and for each $0 \leq i < k$, Q_{i+1} is obtained from Q_i by

- literal rewriting at a non-loop literal in Q_i; or
- applying a simplification rule to a subformula in Q_i; or
- applying a loop rule to a loop literal in Q_i.

We may call a subsequence $Q_i \rightarrow^* Q_k$ a rewrite sequence in the understanding that it is part of some rewrite sequence $Q_0 \rightarrow^* Q_i \rightarrow^* Q_k$ from an initial goal Q_0.

Definition 3.4 *A* goal rewrite system *is a triple* $\langle \mathcal{Q}_L, \mathcal{R}_P, \rightarrow \rangle$, *where* $\mathcal{Q}_L$ *is the set of all goals,* $\mathcal{R}_P$ *is a set of rewrite rules which consists of program rules from* $Comp(P)$, *the simplification rules, and the loop rules; and* $\rightarrow$ *is the set of all rewrite sequences.*

Goal rewrite systems are like term rewrite systems (see, e.g., [Dershowitz and Jouannaud, 1990]) everywhere except at terminating steps: a terminating step at a subgoal may depend on the history of rewriting.

Two desirable properties of rewrite systems are the properties of *termination* and *confluence*. Rewrite systems that possess both of these properties are called *canonical* systems. A canonical system guarantees that the final result of rewriting from any given goal is unique, independent of any order of rewriting. It therefore allows an implementation to be based on any particular order of rewriting.

Since the simplification system is terminating and literal rewriting only generates non-repeated rewrite chains, it is clear that a goal rewrite system is terminating when the given program is finite. A goal is called a *normal form* if it cannot be rewritten by any rule.

Proposition 3.5 *Let* $\langle \mathcal{Q}_L, \mathcal{R}_P, \rightarrow \rangle$ *be a goal rewrite system. If* P *is finite then every rewrite sequence in* $\rightarrow$ *is finite. Further, for any rewrite sequence* $Q_0 \rightarrow^* Q_k$, *if* Q_k *is a normal form, then either* $Q_k = F$ *or* $Q_k = T(C_1) \vee \ldots \vee T(C_m)$ *for some* $m \geq 1$.

Definition 3.6 *A goal rewrite system* $\langle \mathcal{Q}_L, \mathcal{R}_P, \rightarrow \rangle$ *is confluent iff for any rewrite sequences* $t_1 \rightarrow^* t_2$ *and* $t_1 \rightarrow^* t_3$, *there exist* $t_4 \in \mathcal{Q}_L$ *and rewrite sequences* $t_2 \rightarrow^* t_4$ *and* $t_3 \rightarrow^* t_4$.

Theorem 3.7 *Any goal rewrite system* $\langle \mathcal{Q}_L, \mathcal{R}_P, \rightarrow \rangle$ *with a finite* P *is confluent.*

The goal rewrite systems described here are sound and complete w.r.t. *partial stable models*. These results are special cases of those for the rewrite systems for abduction given in the next section.

4 Rewrite systems for abduction

Let P be a program and A a set of abducibles. An *abducible literals* is either an abducible ϕ or its negative counterpart $\neg\phi$.

The rewriting framework that we defined earlier can be extended for abduction in a straightforward way: the only difference in the extended framework is that we do not apply the Clark completion to abducibles. That is, once an abducible appears in a goal, it will remain there unless it is eliminated by the simplification rule $SR2$ or $SR2'$.

Just like a rewrite to T is written as $T(C)$, where C is the underlying rewrite chain (cf. Section 4.1), a rewrite to an abducible literal l will be written as $l(C)$ for rewrite chain C.

In the following we shall denote by $\langle \mathcal{Q}_L, \mathcal{R}_P, A, \rightarrow \rangle$ the rewrite system obtained by the logic program P and the set A of abducibles. We shall show that it is both sound and complete w.r.t. the partial stable models semantics [Przymusinski, 1990], which is a three-valued generalization of answer set semantics. An answer set of P is also its partial stable model. But sometimes P may not have any answer sets. For instance, if $P = \{p \leftarrow \mathsf{not}\, p;\ q \leftarrow\}$, then P has no answer set, because there is no way to assign a truth value to p. However, P has a partial stable model in which q is true and p is undefined. The following definition is adopted from [You and Yuan, 1995].

Let P be a (ground) program. For any set S of default negations, let $F_P(S) = \{\mathsf{not}\, a \mid P \cup S \not\vdash a\}$, where $\vdash$ is the standard propositional derivation relation with each default negation $\mathsf{not}\, \phi$ being treated as a named atom not_ϕ. A *partial stable model* M of P is defined by a maximal fixpoint S of the function that applies F_P twice, $F_P^2(S) = S$, while satisfying $S \subseteq F_P(S)$, in the following way: for any atom ξ, $\neg\xi \in M$ if $\mathsf{not}\, \xi \in S$, $\xi \in M$ if $P \cup S \vdash \xi$, and ξ is *undefined* otherwise. Notationally, any ξ such that $\xi \notin M$ and $\neg\xi \notin M$ represents that ξ is undefined in M. An *answer set* E (also called a *stable model*) can then be defined as a special case by a fixpoint W such that $F_P(W) = W$ and $E = \{\xi \mid P \cup W \vdash \xi\}$.

Theorem 4.1 *Let P be a finite program, A a set of propositions, and $\langle \mathcal{Q}_L, \mathcal{R}_P, A, \rightarrow \rangle$ the goal rewrite system w.r.t. P and A.*

Soundness: *For any literal g and any rewrite sequence*

$$g \rightarrow^* [l_1(C_1) \wedge \cdots \wedge l_k(C_k)] \vee G,$$

where each l_i is either an abducible literal or T, if $C_1 \cup \cdots \cup C_k$ is consistent, then there exists a partial stable model M of $P \cup \{l_1, ..., l_k\}^+$ such that $g \in \bigcup_{1 \le j \le k} C_i \subseteq M$.

Completeness: *For any set of atoms $S \subseteq A$, and any literal g in a partial stable model M of $P \cup S$, there exists a rewrite sequence*

$$g \rightarrow^* [l_1(C_1) \wedge \cdots \wedge l_k(C_k)] \vee G,$$

such that $g \in \bigcup_{1 \le j \le k} C_j \subseteq M$, and $S \subseteq \{l_1, ..., l_k\}$.

We say that a program P has no odd loops if there is no odd loop starting with any literals (programs that have no odd loops are also called *call-consistent* [Dung, 1992]). It is well-known that if P has no odd loops, then the partial stable models of P and the answer sets of P coincide. We now show that for any goal, the underlying rewrite system will generate an approximation of a cover for any program P, and exactly a cover when P has no odd loops.

Theorem 4.2 *Let $\langle \mathcal{Q}_L, \mathcal{R}_P, A, \rightarrow \rangle$ be a goal rewrite system. Suppose q is a proposition and*

$$q \rightarrow^* [l_{11}(C_{11}) \wedge \cdots \wedge l_{1k_1}(C_{1k_1})] \vee \ldots \vee [l_{m1}(C_{m1}) \wedge \cdots \wedge l_{mk_m}(C_{mk_m})]$$

is a rewrite sequence such that each l_{ij} is either T or an abducible literal, and $C_{i1} \cup \cdots \cup C_{ik_i}$ is consistent for each i. Then, if P has no odd loops then

$$\{\{l_{11}, \cdots, l_{1k_1}\}, \cdots, \{l_{m1}, \cdots, l_{mk_m}\}\}$$

is a cover of q. In general, for arbitrary P we have

$$\bigvee_{\alpha \in \mathcal{S}} \alpha \supset [l_{11} \wedge \cdots \wedge l_{1k_1}] \vee \cdots \vee [l_{m1} \wedge \cdots \wedge l_{mk_m}]$$

where $\mathcal{S}$ is any cover of q.

Consider again the boat example in Section 1. The Clark completion of $canCross$ is:

$$canCross \equiv (boat \wedge \neg leak) \vee (boat \wedge leak \wedge hasBucket).$$

Since $boat$, $leak$ and $hasBucket$ are abducibles, so rewriting for $canCross$ terminates in one step, and produces the following cover:

$$\{boat \wedge \neg leak,\ boat \wedge leak \wedge hasBucket\}.$$

Notice that the second explanation is not minimal. To get minimal ones, we have to compute prime implicants of the disjunction of explanations in the cover, which are $boat \wedge \neg leak$ and $boat \wedge hasBucket$.

5 Related work

Traditionally, logic programming proof procedures have been defined abstractly in terms of derivation and refutation. Termination has been considered a separate, implementation issue. On the one hand, this separation is possible since the underlying semantics allows the completeness to be stated without resorting to termination. But completeness is rarely guaranteed in an implementation. On the other hand, the separation is also necessary since these procedures deal with non-ground programs for which the problem of loop-checking is undecidable (even for function-free programs [Bol *et al.*, 1991]). For answer set programming, however, loop handling has become a semantic issue: a sound and complete procedure cannot be defined without it. Thus, a distinct feature of our work is a mechanism of loop handling both for termination and for the implementation of the underlying semantics.

Completed programs have been used in query answering in abstract, abductive procedures in [Console *et al.*, 1991; Denecker and Schreye, 1998; Fung and Kowalski, 1997] for non-ground programs with constraints. These procedures are defined for the three-valued completion semantics under the *certainty* mode of reasoning – computing bindings for which an (existential) goal is true in all indented models. In our case the reasoning mode is *brave* – establishing whether a query is true in one of the intended models.

Our work is closely related to another abstract procedure, the Eshghi-Kowalski procedure (EKP) [Eshghi and Kowalski, 1989], which is sound and complete for ground programs under finite-failure three-valued stable models [Giordano *et al.*, 1996]. Besides loop handling and termination, to some extent, one can say that our goal rewriting system (GRS) simulates EKP in a nontrivial way.

1. GRS incurs no backtracking! Backtracking is simulated by rewriting disjunctions, e.g., $F \vee \Phi \rightarrow \Phi$.
2. Loops that go through negation are handled in EKP by nested structures while in GRS by a *flat* structure using rewrite chains.

These features plus loop handling made it possible to formalize our system as a rewriting system benefiting from the known properties of term rewriting in the literature. This also distinguishes our use of rewrite systems from that by [Fung and Kowalski, 1997]. It seems remarkable that a form of non-monotonic reasoning is just rewriting, two areas of research that had little connection previously.

To illustrate these feature, consider the following program

$$\begin{array}{ll} r1.\ g \leftarrow \mathsf{not}\, a & r3.\ b \leftarrow a \\ r2.\ a \leftarrow \mathsf{not}\, b, \mathsf{not}\, c & r4.\ c \leftarrow a \end{array}$$

and the question whether we can prove g. We may answer this question by the following reasoning: To have g we must have $\mathsf{not}\, a$ (r1); To have $\mathsf{not}\, a$ we must have either b or c (r2) which requires having a (r3 and r4). This results in a contradiction. Note that in this reasoning we need to remember what was required previously ($\mathsf{not}\, a$ in this case). This is exactly how the proof is done by GRS

$$g \rightarrow \neg a \rightarrow b \vee c \rightarrow a \vee c \rightarrow F \vee c \rightarrow c \rightarrow a \rightarrow F$$

However, EKP will go through *six* nested levels, and do it twice through backtracking, before the same conclusion can be reached.

Rewriting can be applied to function-free programs for proving ground goals. The idea is that if every derived goal is ground, then all the mechanisms given in this paper apply directly. Obviously, if for every rule in the given program a variable that appears in the body also appears in the head, then a ground goal will be rewritten to another ground goal. *Domain restricted* programs [Niemelä, 1999] can be instantiated only on domain predicates over variables that do not appear in the head so that the resulting non-ground programs also satisfy this requirement. For example, the program given in the next section is domain restricted.

6 Applications and experimental results

We have implemented the writing framework in SWI-Prolog. In the following, we discuss the performance of our implementation on one particular application of abduction in logic programming, which is the problem of computing successor state axioms from a causal action theory [Lin, 2000].

Consider a logistics domain in which we have a truck and a package. We know that the truck and the package can each be at only one location at any given time, and that if the package is in the truck, then when the truck moves to a new location, so is the package. Suppose that we have the following propositions: $\mathrm{ta}(x)$ – the truck is at location x initially; $\mathrm{pa}(x)$ – the package is at location x initially; in – the package is in the truck initially; $\mathrm{ta}(x,y,z)$ – the truck is at location x after the action of moving it from y to z is performed; $\mathrm{pa}(x,y,z)$ – the package is at location x after the action of moving the truck from y to z is performed; and $\mathrm{in}(y,z)$ – the package is in the truck after the action of moving the truck from y to z is performed. We then have the following logic program:

$$\begin{array}{l} \mathrm{ta}(X, X1, X). \\ \mathrm{pa}(X, X1, X2) \leftarrow \mathrm{ta}(X, X1, X2), \mathrm{in}(X1, X2). \\ \mathrm{ta}(X, X1, X2) \leftarrow X \neq X2, \mathrm{ta}(X), \mathsf{not}\,\mathrm{taol}(X, X1, X2). \\ \mathrm{taol}(X, X1, X2) \leftarrow Y \neq X, \mathrm{ta}(Y, X1, X2). \\ \mathrm{pa}(X, X1, X2) \leftarrow \mathrm{pa}(X), \mathsf{not}\,\mathrm{paol}(X, X1, X2). \\ \mathrm{paol}(X, X1, X2) \leftarrow Y \neq X, \mathrm{pa}(Y, X1, X2). \\ \mathrm{in}(X, Y) \leftarrow \mathrm{in}. \end{array}$$

The first rule is the effect axiom. The second rule is a causal rule which says that if a package is in the truck, then the package should be where the truck is. The rest are frame axioms. For instance, the third one is the frame axiom about ta, with the help of a new predicate taol: if the truck is initially at X, and if one cannot prove that it will be elsewhere after the action is performed, then it should still be at X.

As one can see, the above program, when fully instantiated over any given finite set D of locations, has no odd loops. So our rewrite system will generate a cover for any query. Note that in the program we have omitted domain predicate $loc(Y)$ for each variable Y in the body of a rule (all the variables in the program refer to locations). Thus, the program is domain restricted and needs only to be instantiated for the variable Y in the fourth and sixth rules over the domain of locations.

Now let the set A of abducibles be the following set:

$$\{\mathrm{in}\} \cup \{\mathrm{pa}(x), \mathrm{ta}(x) \mid x \in D\}.$$

The following table shows some of the results for $D = \{1,2,3,4\}$:[2]

Query	Result	Time
$\mathrm{ta}(1,2,3)$	false	0.0
$\mathrm{ta}(3,2,3)$	true	0.0
$\mathrm{pa}(1,2,3)$	$\mathrm{pa}(1) \wedge \neg\mathrm{in}$	0.05
$\neg\mathrm{pa}(1,2,3)$	$\neg\mathrm{pa}(1) \vee \mathrm{in} \vee \mathrm{pa}(2) \vee \mathrm{pa}(3) \vee \mathrm{pa}(4)$	0.2
$\mathrm{pa}(2,2,3)$	$\mathrm{pa}(2) \wedge \neg\mathrm{in}$	0.08
$\neg\mathrm{pa}(2,2,3)$	$\neg\mathrm{pa}(2) \vee \mathrm{in} \vee \mathrm{pa}(1) \vee \mathrm{pa}(3) \vee \mathrm{pa}(4)$	0.1
$\mathrm{pa}(3,2,3)$	$\mathrm{pa}(3) \vee \mathrm{in}$	0.25
$\neg\mathrm{pa}(3,2,3)$	$\neg\mathrm{in} \wedge \neg\mathrm{pa}(3) \vee \neg\mathrm{in} \wedge \mathrm{pa}(1) \vee$ $\neg\mathrm{in} \wedge \mathrm{pa}(2) \vee \neg\mathrm{in} \wedge \mathrm{pa}(4)$	0.1

For instance, the row on $\mathrm{pa}(1,2,3)$ says that for it to be true, the package must initially be at 1 and cannot be inside the truck (otherwise, it would be moved along with the truck), and the computation took 0.1 seconds.The row on $\mathrm{pa}(3,2,3)$ says that for it to be true, either the package was initially at 3 or it was inside the truck. The outputs for larger Ds are

[2]On a PIII 1GHz PC with 512MB RAM running SWI-Prolog 3.2.9.

similar. The performance varies for different queries. For simple queries like ta$(1,2,3)$, their covers can be computed almost in constant time. The hardest one is for pa$(3,2,3)$ which took 25 minutes when $|D| = 7$.

It is interesting to compare our system with an alternative for computing the cover of a query. As we mentioned in Section 2, the set of abductive explanations according to Kakas and Mancarella is actually a cover. One way of computing these abductive explanations is to add, for each proposition $p \in A$, the following two clauses ([Satoh and Iwayama, 1991]): $p \leftarrow \mathsf{not}\, \neg p$ and $\neg p \leftarrow \mathsf{not}\, p$ into the original program, and use the fact that there will be a one to one correspondence between abductive explanations of q under the original program and answer sets of the new program that contain q. So one can use an answer set generator, for example **smodel** or **dlv**, to compute a cover of query by generating all the answer sets in the new program that contain the query. However, the problem here is that there are too many such answer sets in this case. For instance, suppose there are n locations, then the number of answer sets that contain any particular query is in the order of 2^{2n}, roughly one half of the number of complete hypotheses, even for a very simple query like ta$(1,2,3)$. We do not know at the moment if there is any efficient way of using an answer set generator to compute a cover set of a query.

7 Future work

There are several directions for extending this work. One of them is to consider rewriting for non-ground logic programs for some restricted yet decidable classes of non-ground goals. Another important one is to extend this to programs with constraints of the form:

$$\bot \leftarrow a_1, ..., a_i, \mathsf{not}\, b_1, ..., \mathsf{not}\, b_n$$

Our new definition of abduction can be extended to include these constraints straightforwardly. The challenge is in extending our rewriting system accordingly.

8 Acknowledgements

We would like to thank Ilkka Niemelä for helpful discussions related to the topics of this paper, and Ken Satoh for comments on earlier version of this paper. The first author's work was supported in part by the Research Grants Council of Hong Kong under Competitive Earmarked Research Grant HKUST6145/98E. The work by the second author was carried out mainly during his visits to Hong Kong University of Science and Technology and the Institute of Information Science, Academia Sinica in Taiwan.

References

[Bol *et al.*, 1991] R. Bol, A. Krzysztof, and J. Klop. An analysis of loop checking mechanisms for logic programs. *Theoretical Computer Science*, 86(1):35–39, 1991.

[Console *et al.*, 1991] L. Console, D. Theseider, and P. Porasso. On the relationship between abduction and deduction. *J. Logic Programming*, 2(5):661–690, 1991.

[Denecker and Schreye, 1998] M. Denecker and D. De Schreye. Sldnfa: an abductive procedure for normal abductive programs. *J. Logic Programming*, 34(2):111–167, 1998.

[Dershowitz and Jouannaud, 1990] N. Dershowitz and P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Vol B: Formal Methods and Semantics*, pages 243–320. North-Holland, 1990.

[Dung, 1992] P.M. Dung. On the relation between stable and well-founded semantics of logic programs. *Theoretical Computer Science*, 105:7–25, 1992.

[Eshghi and Kowalski, 1989] K. Eshghi and R.A. Kowalski. Abduction compared with negation by failure. In *Proc. 6th ICLP*, pages 234–254. MIT Press, 1989.

[Fung and Kowalski, 1997] T. Fung and R. Kowalski. The iff proof procedure for abductive logic programming. *J. Logic Programming*, 33(2):151–164, 1997.

[Gelfond and Lifschitz, 1988] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. 5th ICLP*, pages 1070–1080. MIT Press, 1988.

[Giordano *et al.*, 1996] L. Giordano, A. Martelli, and M. Sapino. Extending negation as failure by abduction: A three-valued stable model semantics. *J. Logic Programming*, 26(1), 1996.

[Kakas and Mancarella, 1990] A. Kakas and P. Mancarella. Generalized stable models: a semantics for abduction. In *Proc. 9th European Conf. for AI*, 1990.

[Konolige, 1992] K. Konolige. Abduction versus closure in causal theories. *Artificial Intelligence*, 53:255–272, 1992.

[Kunen, 1989] K. Kunen. Signed data dependencies in logic programs. *J. Logic Programming*, 7(3):231–245, 1989.

[Lin, 2000] F. Lin. From causal theories to successor state axioms: bridging the gap between nonmonotonic action theories and STRIPS-like formalisms. In *Proc. AAAI '00*, 2000.

[Niemelä, 1999] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. and AI*, 25(3-4):241–273, 1999.

[Poole, 1988] D. Poole. Representing knowledge for logic-based diagnosis. In *Proc. Fifth Generation Computer Systems Conference*, pages 1282–1290, 1988.

[Przymusinski, 1990] T.C. Przymusinski. Extended stable semantics for normal and disjunctive logic programs. In *Proc. 7th ICLP*, pages 459–477. MIT Press, 1990.

[Reiter and de Kleer, 1987] R. Reiter and J. de Kleer. Foundations of ATMS. In *Proc. AAAI87*, 1987.

[Satoh and Iwayama, 1991] K. Satoh and N. Iwayama. Computing abduction using the tms. In *Proc. the 8th International Conference on Logic Programming*, 1991.

[Satoh and Iwayama, 1992] K. Satoh and R. Iwayama. A query evaluation method for abductive logic programming. In *Proc. JICSLP'92*, 1992.

[You and Yuan, 1995] J. You and L. Yuan. On the equivalence of semantics for normal logic programs. *J. Logic Programming*, 22:212–221, 1995.