The Leland Stanford Junior University

to all to whom these Letters shall come Greeting

The Trustees and Faculty of the University, by virtue of the authority vested in them, have conferred on

Fangzhen Lin

who has satisfactorily pursued the Studies and passed the Examinations required therefor the Degree of

Doctor of Philosophy

in

Computer Science

with all the Rights, Privileges, and Honors thereunto appertaining

Given at Stanford University in the State of California on the Twenty-Sixth Day of September in the Year One Thousand Nine Hundred and Ninety-One the Two Hundred-Sixteenth Year of the Republic and the One Hundred-First Academic Year of the University.

LELAND STANFORD JUNIOR UNIVERSITY INCORPORATED 1901

James C. Gaither
President of the Board of Trustees

President of the University

James F. Gibbons
School of Engineering

H0440058100



林 方 真
LIN, FANGZHEN

性别
男

出生日期
1963-02-18

港澳证件号码
P630028(A)

签发日期
2004-03-12

本证有效期至
2014-03-12


港澳居民来往内地
通行证
C<CHN335423295370<<<<<<<<<00<<2
6302186M1403125HO4400581001129
LIN<<FANGZHEN<<<<<<<<<<<<<<<<<<

中国科学技术大学

“千人计划”短期项目聘用协议书

根据国家有关文件精神规定，结合中国科学技术大学实际情况，为保障“千人计划”短期项目的顺利实施，经中国科学技术大学（简称甲方）和受聘人员（简称乙方）共同协商，同意签署如下协议书:

一、甲方根据学科发展和人才培养的需要，聘请乙方每年定期来校工作。

二、甲乙双方根据所在院系学科发展的实际需要，协商确定具体的岗位职责和工作目标。乙方为甲方相关学科的发展积极工作，在学科建设、人才引进、青年教师培养、学术团队建设等方面展开工作。

三、甲方为乙方提供科研经费、工作场所、科研设备、科研助手以及其他工作、生活需要的条件，积极协助乙方申报国家“千人计划”短期项目。

四、乙方若入选国家“千人计划”短期项目，甲方将聘用乙方为“千人计划”教授岗位，聘期三年，每年在校工作不少于2个月。

五、乙方在聘期内，依托甲方所取得的教学、科研等成果均属职务成果，其发表有关论文、著作或申报有关奖励、专利等，均须同时署乙方及甲方名称，且甲方必须为作者第一单位。

六、乙方在甲方受聘期间，因履行职务或者主要是利用甲方的物质技术条件、科研信息等产生的发明创造、技术秘密等有关的知识产权均属于甲方享有，乙方享有署名权。

七、本协议未尽事宜，由甲乙双方共同协商确定。

八、本协议自甲乙双方签字盖章之日起生效。

九、本协议一式两份，甲乙双方各执一份。

甲方（盖章）:　　乙方（签字）:

年　　月　　日　　年　　月　　日

香港科技大學
THE HONG KONG
UNIVERSITY OF SCIENCE
AND TECHNOLOGY

人力資源處
HUMAN RESOURCES OFFICE

Our Ref.: PR/LIN/F

Tel. No.: 2358 6581

2 December 2010

To Whom It May Concern

This is to certify that Professor Fangzhen LIN, holder of Hong Kong Identity Card No. P630028(A), is currently employed by The Hong Kong University of Science and Technology as Professor of Computer Science and Engineering on a substantive basis.

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY
Human Resources Office
人力資源處
香港科技大學

Rebecca LAI (Ms)
for Director of Human Resources

Games and Economic Behavior ••• (••••) •••–•••

Contents lists available at ScienceDirect

Games and Economic Behavior

www.elsevier.com/locate/geb

Two equivalence results for two-person strict games

Pingzhong Tang *, Fangzhen Lin

Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong

ARTICLE INFO

Article history:
Received 21 April 2008
Available online xxxx

JEL classification:
C72

Keywords:
Strictly competitive games
Ordinal potential games
Quasi-supermodular games
Best response equivalence
Strict games

ABSTRACT

A game is strict if for both players, different profiles have different payoffs. Two games are best response equivalent if their best response functions are the same. We prove that a two-person strict game has at most one pure Nash equilibrium if and only if it is best response equivalent to a strictly competitive game, and that it is best response equivalent to an ordinal potential game if and only if it is best response equivalent to a quasi-supermodular game.

© 2010 Elsevier Inc. All rights reserved.

1. Introduction

In this paper we prove two results about pure Nash equilibria (PNEs) for 2-person *strict* games, which are those where payoff functions are one-to-one. The first result says that a 2-person strict game has at most one PNE iff it is *best-response equivalent* (Rosenthal, 1974) to a strictly competitive game (Moulin, 1976; Friedman, 1983). The second result says that a 2-person strict game is best-response equivalent to an ordinal potential game (Monderer and Shapley, 1996) iff it is best-response equivalent to a quasi-supermodular game (Topkis, 1998).

These results came out of our project on using computers to discover theorems in game theory (see, e.g. Tang and Lin, 2009). We were looking for conditions that imply the uniqueness and existence of the PNEs, and were using computers to run through a large set of conjectures. Given the number of all possible games, it made sense to test these conditions first on some special classes of games, and the class of strict games is a natural choice as in these games, the payoff functions are one-to-one, thus easier to generate. As it turned out, some interesting conditions hold only for this class of games. We have described some of them in Tang and Lin (2009). The two results that we report here are special in that their proofs are non-trivial and had to be done manually.

2. Basic definitions

A 2-person game is a tuple (A, B, u_1, u_2), where A and B are sets of pure strategies of players 1 and 2, respectively, and u_1 and u_2 are functions from $A \times B$ to reals called the *payoff functions* for players 1 and 2, respectively. In this paper we assume both A and B are finite.

A game is said to be *strict* if both players' payoff functions are one-to-one: for any (a_1, b_1) and (a_2, b_2) in $A \times B$, if $(a_1, b_1) \neq (a_2, b_2)$ then $u_i(a_1, b_1) \neq u_i(a_2, b_2)$, $i = 1, 2$. A related notion in the literature is *non-degenerate games* (see, e.g.

* Corresponding author.
E-mail address: kenshin@cse.ust.hk (P. Tang).

0899-8256/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.geb.2010.04.007

Please cite this article in press as: Tang, P., Lin, F. Two equivalence results for two-person strict games. Games Econ. Behav. (2010), doi:10.1016/j.geb.2010.04.007

Berger, 2007): a game is non-degenerate if for any $a_1, a_2 \in A$ and $b_1, b_2 \in B$, $u_1(a_1, b_1) \neq u_1(a_2, b_1)$ whenever $a_1 \neq a_2$, and $u_2(a_1, b_1) \neq u_2(a_1, b_2)$ whenever $b_1 \neq b_2$. Clearly, a strict game is also non-degenerate, but the converse is not true in general.

For each $b \in B$, let $B_1(b)$ be the set of best responses by player one to the action b by player two:

$$B_1(b) = \{a \mid a \in A, \text{ and for all } a' \in A, u_1(a', b) \leqslant u_1(a, b)\}.$$

Similarly, for each $a \in A$, let

$$B_2(a) = \{b \mid b \in B, \text{ and for all } b' \in N, u_2(a, b') \leqslant u_2(a, b)\}.$$

A profile $(a, b) \in A \times B$ is a pure Nash equilibrium (PNE) if $a \in B_1(b)$ and $b \in B_2(a)$.

In this paper we consider only PNEs. Our initial motivation for this work was to capture the class of games with at most one PNE. We started with strictly competitive games (see below). While every strictly competitive game can have at most one PNE, the converse is not true in general. This led us to consider notions of equivalence between games that will preserve PNEs. The one that suits our purpose here is that of *best-response equivalence* (Rosenthal, 1974): two 2-person games $G_1 = (A, B, u_1, u_2)$ and $G_2 = (A, B, u_1', u_2')$ are said to be best-response equivalent if for each $a \in A$, $B_2(a)$ in G_1 and G_2 are the same, and for each $b \in B$, $B_1(b)$ in G_1 and G_2 are the same.

It is easy to see that best-response equivalent games have the same sets of PNEs.

3. Strictly competitive games and at most one PNE

A game (A, B, u_1, u_2) is strictly competitive if for every pair of profiles s_1 and s_2 in $A \times B$, we have that $u_1(s_1) \leqslant u_1(s_2)$ iff $u_2(s_2) \leqslant u_2(s_1)$. We have the following result for strict games.

Theorem 1. *A strict 2-person game has at most one PNE if and only if it is best-response equivalent to a strictly competitive game.*

The "if" part of the theorem follows from the well-known fact that all PNEs in a strictly competitive game have the same payoff for each player (cf. p. 22, Osborne and Rubinstein, 1994), and that for strict games, profiles with same payoffs must be identical.

To prove the "only if" part of the theorem, for any given game $G = (A, B, u_1, u_2)$, we associate with it the following directed graph R: the vertices of R are profiles of the game and the set of arcs is:

$$\{((a, b), (a', b)) \mid a' \in B_1(b), a \notin B_1(b)\} \cup$$
$$\{((a, b), (a, b')) \mid b \in B_2(a), b' \notin B_2(a)\}.$$

Informally, this is a graph where player 1 moves to her best response, and player 2 moves away from her best response. Thus for a strictly competitive game, player 1's payoffs increase while player 2's payoffs decrease along every path of this graph.

The "only if" part of the theorem follows directly from the following two lemmas that relate strictly competitive games and acyclicity of the associated graphs.

Lemma 3.1. *A 2-person game is best-response equivalent to a strictly competitive game if and only if its directed graph has no cycle.*

Proof. Let $G = (A, B, u_1, u_2)$ be a 2-person game, and R its directed graph as defined above. If R has a cycle, then G cannot be equivalent to a strictly competitive game because if $G' = (A, B, u_1', u_2')$ is such a game, then for any profiles s_1 and s_2, if there is an arc from s_1 to s_2 in R, then $u_1'(s_1) < u_1'(s_2)$. So along the cycle, there must be a sequence $u_1'(t_1) < u_1'(t_2) < \cdots < u_1'(t_k) < u_1'(t_1)$, which is a contradiction.

Now if R has no cycle, construct a game $G' = (A, B, u_1', u_2')$ where u_1' and u_2' are defined as follows:

- $R_0 = \{(s, s') \mid \text{there is an arc from } s \text{ to } s' \text{ in } R\}$, $S_0 = \{s \mid s \in A \times B, \text{ there is no } s' \text{ such that } (s', s) \in R_0\}$.
- Suppose that R_k and S_k is defined, let

$$R_{k+1} = \{(s, s') \mid (s, s') \in R_k, \text{ and } s, s' \notin S_k\},$$
$$S_{k+1} = \{s \mid \text{for some } s', (s, s') \in R_k \text{ but there is no } s' \text{ such that } (s', s) \in R_{k+1}\}.$$

- Since R has no cycles, there is a finite number n such that for all $k > n$, $R_k = S_k = \emptyset$, and $A \times B = S_0 \cup \cdots \cup S_n$.
- Let u_1' be a one-to-one function from $A \times B$ to the set of positive integers such that if $i < j$, then for any $s \in S_i$ and $s' \in S_j$, $u_1'(s) < u_1'(s')$.
- Let $u_2' = -u_1'$.

Clearly, G' is strictly competitive, and best-response equivalent to G. □

Please cite this article in press as: Tang, P., Lin, F. Two equivalence results for two-person strict games. Games Econ. Behav. (2010), doi:10.1016/j.geb.2010.04.007

Lemma 3.2. *If the directed graph of a 2-person game has a cycle, then the game has at least two PNEs.*

Proof. Let $G = (A, B, u_1, u_2)$ be a 2-person game, and R its directed graph as defined above. Suppose $s_1, \ldots, s_k, s_{k+1}$ is a cycle in R. Suppose $s_1 = (a, b)$. Then either $s_2 = (a', b)$ for some $a' \neq a$ or $s_2 = (a, b')$ for some $b' \neq b$. Suppose it is the first case, $s_2 = (a', b)$. The proof for the second case is similar. Then by our construction of R, $s_3 = (a', b')$ for some $b' \neq b$, $s_4 = (a'', b')$ for some $a'' \neq a'$. We show that s_2 and s_4 are both PNEs of G. Because there is an arc from (a, b) to (a', b) in R, $a' \in B_1(b)$. Because there is an arc from (a', b) to (a', b') in R, $b \in B_2(a')$. Thus $s_2 = (a', b)$ is a PNE. Similarly, because there is arc from (a', b') to (a'', b') and an arc from (a'', b') to (a'', b'') in R for some b'' (it is possible that $a = a''$ and $b = b''$) $a'' \in B_1(b')$ and $b' \in B_2(a'')$. Thus $s_4 = (a'', b')$ is a PNE as well. □

One way to visualize this lemma[1] is to think of a path in this directed graph as a sequence of connected arrows that go either horizontally (player 1's move) or vertically (player 2's move). Thus a cycle must have an even number of arcs and vertices (profiles), and half of those profiles, the ones at the end of the horizontal arrows, must be PNEs.

3.1. Discussion

- The "if" part of Theorem 1 does not hold for general 2-person games – one can easily find a strictly competitive game that is not strict and has more than one PNE.
- The "only if" part of Theorem 1 holds for general 2-person games as well. This is because our proofs of the two lemmas above do not use the assumption that the games are strict, thus hold for general 2-person games as well. However, for general 2-person games, the "only if" part does not hold if we replace "at most one PNE" by "at most one payoff for the PNE" (meaning all the PNEs have the same payoff for each player). For instance, in the following game

1, 1	2, 2
2, 2	1, 1

the two PNEs have the same payoff pair $(2, 2)$. But this game is not best-response equivalent to any strictly competitive game as the clockwise movement between the profiles is a cycle in its directed graph.

4. Ordinal potential and quasi-supermodular games

We first review *ordinal potential games* (Monderer and Shapley, 1996). Given a game $G = (A, B, u_1, u_2)$, a function P from $A \times B$ to reals is an ordinal potential for G, if for every $y \in B$ and every $x, z \in A$,

$$u_1(x, y) - u_1(z, y) > 0 \quad \text{iff} \quad p(x, y) - p(z, y) > 0,$$

and for every $x \in A$ and every $y, w \in B$,

$$u_2(x, y) - u_2(x, w) > 0 \quad \text{iff} \quad p(x, y) - p(x, w) > 0.$$

G is called an ordinal potential game if it admits an ordinal potential function.

Monderer and Shapley (1996) proved that every finite ordinal potential game has a PNE.

Another class of games that always have PNEs is the so-called *quasi-supermodular* game (Topkis, 1998). A finite game (A, B, u_1, u_2) is quasi-supermodular if there are two linear orderings $<_1$ and $<_2$ of A and B, respectively, such that u_1 and u_2 satisfies the so-called single crossing property: $\forall (a_1 <_1 a_2) \in A^2$ and $\forall (b_1 <_2 b_2) \in B^2$, we have

$$u_1(a_2, b_1) > u_1(a_1, b_1) \quad \Rightarrow \quad u_1(a_2, b_2) > u_1(a_1, b_2),$$
$$u_2(a_1, b_2) > u_2(a_1, b_1) \quad \Rightarrow \quad u_2(a_2, b_2) > u_2(a_2, b_1)$$

and

$$u_1(a_2, b_1) \geqslant u_1(a_1, b_1) \quad \Rightarrow \quad u_1(a_2, b_2) \geqslant u_1(a_1, b_2),$$
$$u_2(a_1, b_2) \geqslant u_2(a_1, b_1) \quad \Rightarrow \quad u_2(a_2, b_2) \geqslant u_2(a_2, b_1).$$

For strict games, these two pairs of conditions are equivalent.

Given a 2-person game, a *best-response path* (Voorneveld, 2000) of the game is a sequence of profiles $s_1, \ldots, s_n$ such that for each $1 \leqslant i < n$, s_i and s_{i+1} differ on exactly one coordinate with the deviating player moving to a best response: if $s_i = (a, b)$ and $s_{i+1} = (a', b')$, then either $a = a'$ and $b' \in B_2(a)$ or $b = b'$ and $a' \in B_1(b)$. A best-response cycle is a best-response path whose two end points are the same.

Voorneveld (2000) introduced a slightly different notion of best-response cycles, and showed that a game with countable strategy sets is best-response equivalent to a *potential game* iff it has no best-response cycles under his definition. The

[1] This was suggested by one of the reviewers.

Please cite this article in press as: Tang, P., Lin, F. Two equivalence results for two-person strict games. Games Econ. Behav. (2010), doi:10.1016/j.geb.2010.04.007

difference between Voorneveld's notion and ours is that Voorneveld requires his best-response cycle to include at least one edge that strictly increases the payoff of the deviating player. The difference disappears for strict games, and without going into details, we remark here that while an ordinal potential game may not be a potential game, every ordinal potential game is best-response equivalent to a potential game. Thus from Voorneveld's result, we see that a 2-person finite strict game is best-response equivalent to an ordinal potential game iff it has no best-response cycles.

It also follows from the weak finite best response improvement path property[2] (cf. e.g. Kukushkin et al., 2005) of quasi-supermodular games that if a finite 2-person strict game is best-response equivalent to a quasi-supermodular game, then it has no best-response cycle. In this section, we show the other direction: if a finite 2-person strict game has no best-response cycle, then it is best-response equivalent to a quasi-supermodular game. Thus, a finite strict 2-person game is best-response equivalent to an ordinal potential game iff it is best-response equivalent to a quasi-supermodular game:

Theorem 2. *A 2-person strict game is best-response equivalent to a quasi-supermodular game if and only of it has no best-response cycle.*

Corollary 2.1. *A 2-person strict game is best-response equivalent to a quasi-supermodular game iff it is best-response equivalent to an ordinal potential game.*

Notice that best-response paths model Cournot dynamics, thus Theorem 2 also implies that if Cournot dynamics does not cycle, then it must be best-response equivalent to a quasi-supermodular game.

We now prove Theorem 2 through two lemmas. The first one relates quasi-supermodular games to the complementarity of best response functions, and the second the complementarity of best-response function to the acyclicity of best-response paths. While the "only if" part of Theorem 2 is already entailed by the current results, these two lemmas also provide an interesting alternative proof.

Given a 2-person game (A, B, u_1, u_2), and two linear orderings $<_1$ and $<_2$ of A and B, respectively, we say that the best-response function B_1 for player 1 is *non-decreasing* with respect to $<_1$ and $<_2$, if for each pair $b_i <_2 b_j$ $a_i = B_1(b_i)$ and $a_j = B_1(b_j)$, we have $a_i \leqslant_1 a_j$ and similarly for player 2, the best-response function B_2 is non-decreasing with respect to $<_1$ and $<_2$, if for each pair $a_i < a_j$ $1 \leqslant i < j \leqslant m$, $b_i = B_2(a_i)$ and $s'_j = (a_j, b_j)$, we have $b_i \leqslant_2 b_j$.

Essentially, if both players' best-response functions are non-decreasing, then they are strategic complementary.

Lemma 4.1. *A strict game is best-response equivalent to a quasi-supermodular game iff there exist two linear orderings under which the best-response functions for both players are non-decreasing.*

Non-decreasing best-response functions for quasi-supermodular game is a well-known intuition (the only if part), although we still prove it as follows since it is simple.

Proof of Lemma 4.1. Let $G = (A, B, u_1, u_2)$ be a strict game.

$\Rightarrow$: Suppose G is best-response equivalent to a quasi-supermodular game G'. Clearly, the best-response functions of G and G' are the same under any linear orderings. Now let $G' = (A, B, u'_1, u'_2)$. Since G' is quasi-supermodular, there are two linear orderings $<_1$ of A and $<_2$ of B such that u'_1 and u'_2 satisfy the single crossing properties. Then in G', the best-response functions under $<_1$ and $<_2$ are non-decreasing. For otherwise, suppose that B_1 is not non-decreasing in G'. Then there are two profiles (a_1, b_1) and (a_2, b_2) such that

$$a_1 \in B_1(b_1), \qquad a_2 \in B_1(b_2), \qquad b_1 <_2 b_2,$$
$$a_2 <_1 a_1.$$

Thus $u'_1(a_1, b_1) > u'_1(a_2, b_1)$ and $u'_1(a_2, b_2) > u'_1(a_1, b_2)$, which violate the single crossing conditions.

$\Leftarrow$: Suppose the best-response functions of the two players in G are non-decreasing under $<_1$ and $<_2$. We denote in this part that for $a_i, a_j \in A$, $a_i <_1 a_j$ if $i < j$. Similar for $b_i, b_j \in B$. Now consider the following game $G' = (A, B, u'_1, u'_2)$:

- if $s < t$, then $u'_1(a_i, b_s) < u'_1(a_j, b_t)$ for any a_i and a_j in A;
- for any $b_s \in B$, if $B_1(b_s) = \{a_k\}$ in G, then
  - if $i \neq a_k$, then $u'_1(a_i, b_s) < u'_1(a_k, b_s)$;
  - if $i > j > k$, then $u'_1(a_i, b_s) < u'_1(a_j, b_s)$;
  - if $i < j < k$, then $u'_1(a_j, b_s) > u'_1(a_i, b_s)$;
  - if $i < k < j$, then $u'_1(a_j, b_s) > u'_1(a_i, b)$;
- similarly for u'_2.

[2] A game has weak finite best-response improvement path property if every best-response path, in which the deviating player moves to a strictly better best response, is finite.

Please cite this article in press as: Tang, P., Lin, F. Two equivalence results for two-person strict games. Games Econ. Behav. (2010), doi:10.1016/j.geb.2010.04.007

Clearly, one can find two such functions u_1' and u_2', and that G' is best-response equivalent to G. We show that G' is quasi-supermodular on $<_1$ and $<_2$. Suppose $i < j$, $s < t$, and $u_1'(a_j, b_s) > u_1'(a_i, b_s)$. We show that $u_1'(a_j, b_t) > u_1'(a_i, b_t)$. Let $B_1(b_s) = \{a_k\}$ and $B_1(b_t) = \{a_l\}$. Then $k \leqslant l$ according to the non-decreasing property of player one's best-response function. Given $u_1'(a_j, b_s) > u_1'(a_i, b_s)$, by our construction, there are three cases:

1. $j = k$. We have $i < j \leqslant l$, thus $u_1'(a_j, b_t) > u_1'(a_i, b_t)$.
2. $i < j < k$. We have $i < j < k \leqslant l$, thus $u_1'(a_j, b_t) > u_1'(a_i, b_t)$.
3. $i < k < j$. We have either $i < l \leqslant j$ or $i < j < l$. Either way, we have $u_1'(a_j, b_t) > u_1'(a_i, b_t)$ by our construction.

Thus u_1' satisfies the single crossing condition. Similarly, u_2' satisfies the single crossing condition. So G is a quasi-supermodular game. □

Given Lemma 4.1, Theorem 2 then follows from the following lemma, which says non-decreasing best response functions are further equivalent to absence of best-response cycle. Before we start stating the lemma and the proof, we have some brief remarks on a related work.

At first sight, the "only if" part of the lemma below is closely related to an existing result, Echenique's (2004) Proposition 11, which says every non-degenerate ordinal potential game (thus without best-response cycle) is a *game of strategic complementarities*.[3] However, the definition of *game of strategic complementarities* differs from ours in that the orders on strategies are partial and complete lattice while we further require them to be linear. In other words, the set of games that are best-response equivalent to a quasi-supermodular game in our definition is a proper subset of the set of games of strategic complementarities. Therefore, our result generalizes Echenique's result but not vice versa. In addition, Echenique's (2004) proof relies on Theorem 5, which says a game with at least two PNEs is a GSC defined on lattices. However, there are games with at least two PNEs that are not GSC under any linear orderings (see the game given in Fig. 4 in Echenique's paper, for example). This means that Echenique's proof does not work on the games that we consider in our paper.

Lemma 4.2. *A strict 2-person game has no best-response cycle iff there are two linear orderings under which both players' best response functions are non-decreasing.*

Proof. Let $G = (A, B, u_1, u_2)$ be a strict game.

$\Leftarrow$: Let $<_1$ and $<_2$ be two linear orderings of A and B, respectively, under which the best-response functions B_1 and B_2 for player 1 and 2, respectively, are non-decreasing. We show that G has no best-response cycle. We begin with a profile (a_1, b_1) where, with loss of generality, $\{a_1\} = B_1(b_1)$. Let then player 2 and player 1 deviates to each own best-response alternatively in the following rounds. If in the first round, player 2 deviates to an action b_2, there are three cases:

- Case 1: $b_1 <_2 b_2$, then according to the non-decreasing property of B_1, we have $a_1 \leqslant_1 a_2$ where $\{a_2\} = B_1(b_2)$. If $a_1 = a_2$, it means that (a_1, b_2) is a PNE, where the best response path terminates. Otherwise $a_1 <_1 a_2$, this process will continue generating new profiles that alternatively increase in each player's action until it meets a PNE and terminates. This shows that the best-response path generated this way cannot return to where it started, therefore in this case there is no best-response cycle.
- Case 2: $b_2 <_2 b_1$. It is not hard to see that this process will continue generating new profiles that strictly increases in one of its players rank until it meets a PNE. This shows again that there is no best-response cycle.
- Case 3: $b_1 = b_2$, this means that (a_1, b_1) is a PNE, where the best-response path terminates. No cycle exists.

We have proved the $\Leftarrow$ part of the lemma.

$\Rightarrow$: We prove this part constructively. Suppose there is no best-response cycle in G. Our following procedure will produce two ranking functions, $rank_A$ that maps each $a \in A$ to an integer between 1 and $|A|$, and $rank_B$ that maps each $b \in B$ to an integer between 1 and $|B|$. From these two rankings, we get two linear orderings on A and B: $a <_1 a'$ if $rank_A(a) > rank_A(a')$, and $b <_2 b'$ if $rank_B(b) > rank_B(b')$. We show that under these two linear orderings, the best-response functions for both players are non-decreasing. For an informal description of how the following algorithm works, see an illustrative example below:

1. Initially, let $A_r = A$, $B_r = B$, $I_A = I_B = 1$, $A_c = B_c = []$ (the empty list), $rank_A(a) = 0$ for any $a \in A$, and $rank_B(b) = 0$ for any $b \in B$.
2. **while** $A_r \neq \emptyset$ or $B_r \neq \emptyset$ **do**
   2.1. Let $(a, b) \in A_r \times B_r$ be a Nash equilibrium of G (there must be such a Nash equilibrium as we show below), and let $A_c = [a]$, $B_c = [b]$.
   2.2. **while** $A_c \neq []$ or $B_c \neq []$ **do**
      (i) **If** $A_c \neq []$ **then**

[3] We thank the advisory editor for exposing us to this work.

Please cite this article in press as: Tang, P., Lin, F. Two equivalence results for two-person strict games. Games Econ. Behav. (2010), doi:10.1016/j.geb.2010.04.007

(a) Let a^* be the first element in A_c.
(b) $rank_A(a^*) = I_A$; $I_A = I_A + 1$.
(c) Delete a^* from A_c.
(d) Delete a^* from A_r.
(e) For each $b' \in B$ such that $B_1(b') = \{a^*\}$, add b' to the end of B_c. (If there are more than one such b', the order by which they are added to B_c does not matter.)

(ii) **If** $B_c \neq []$ **then**
(a) Let b^* be the first element in B_c.
(b) $rank_B(b^*) = I_B$; $I_B = I_B + 1$.
(c) Delete b^* from B_c.
(d) Delete b^* from B_r.
(e) For each $a' \in A$ such that $B_2(a') = \{b^*\}$, add a' to the end of A_c. (If there are more than one such a', the order by which they are added to A_c does not matter.)

We now show the correctness of step 2.1 and that the best-response sequences for both players under the orderings output by the procedure are non-decreasing.

Let A_{r_i} and B_{r_i} be the A_r and B_r at the beginning of the ith loop and let $G_i = (A_{r_i}, B_{r_i}, u_1, u_2)$, we have the following properties of our procedure that guarantee the precondition of 2.1 can always be satisfied.

Proposition 4.1. *For each $G_i = (A_{r_i}, B_{r_i}, u_1, u_2)$, we have*:

1. *For all $b_j \in B_{r_i}$, $B_1(b_j) \subseteq A_{r_i}$.*
2. *For all $a_j \in A_{r_i}$, $B_2(a_j) \subseteq B_{r_i}$.*
3. *G_i has no best-response cycle.*
4. *Every Nash equilibrium of G_i is also one of the original game G.*

Proof. We prove this proposition by induction on i.

- Base case: It is easy to verify that $G_1 = G$ satisfies 1–4.
- Inductive case: Suppose G_i satisfies 1–4, we now verify 1–4 for G_{i+1}.
  1. Suppose otherwise, there exists $b_j \in B_{r_{i+1}}$ such that $B_1(b_j) = \{a\} \subseteq A_{r_i} \setminus A_{r_{i+1}}$. According to our procedure, a will be deleted during this loop, which means b_j will been added to B_c and deleted during this loop because B_c will be empty at the end of the loop. This means $b_j \in B_{r_i} \setminus B_{r_{i+1}}$, a contradiction.
  2. Similar to above.
  3. 1–2 tells us the best-responses of G_{i+1} are also best-responses of G. Suppose G_{i+1} has a best-response cycle, this cycle would still be one in G, a contradiction.
  4. This part also follows directly from 1 to 2. □

Since G_i has no best-response cycle $\Leftrightarrow$ G_i is best-response equivalent to an ordinal potential game $\Rightarrow$ G_i has a Nash equilibrium. According to 1 and 4 of the above proposition, G_i always has a Nash equilibrium. Up to now, we have proved that the precondition of 2.1 in our procedure can always be satisfied.

Proposition 4.2. *The best-response functions under the orderings generated by our procedure are non-decreasing.*

Proof. For B_2 and for all $a_1, a_2 \in A$ with $a_2 <_1 a_1$, let $B_2(a_1) = \{b_1\}$ and $B_2(a_2) = \{b_2\}$, we show in the following that $b_2 \leqslant_1 b_1$.

If $b_1 = b_2$, we get to the conclusion immediately.

Now consider $b_1 \neq b_2$, when b_1 is the first element of B_c, a_1 is added to the end of A_c because $B_2(a_1) = \{b_1\}$. Similarly, when b_2 is the first element of B_c, a_2 is added to the end of A_c because $B_2(a_2) = \{b_2\}$. Since $a_2 <_1 a_1$, we must have a_2 is added to A_c later than a_1. This means b_2 appears as the first element of B_c later than b_1. So we get $b_2 <_2 b_1$.

Similar for player B_1. □

A few more words about the proof above:

- As we can see, our proof is algorithmic thus provides a means to sort acyclic best-response functions into non-decreasing order. In other words, given a strict (more generally, non-degenerate) game without best-response cycle, our procedure tells how to order the strategies to make the game quasi-supermodular.
- Also as a byproduct of our proof, in two-person strict quasi-supermodular game, the set of PNEs is actually linearly ordered (of course, this is partially caused by the fact that we require the orders on the strategy sets to be linear), in contrast to a previous result (Zhou, 1994) that says the set of PNEs is a complete lattice in a supermodular game.

Please cite this article in press as: Tang, P., Lin, F. Two equivalence results for two-person strict games. Games Econ. Behav. (2010), doi:10.1016/j.geb.2010.04.007

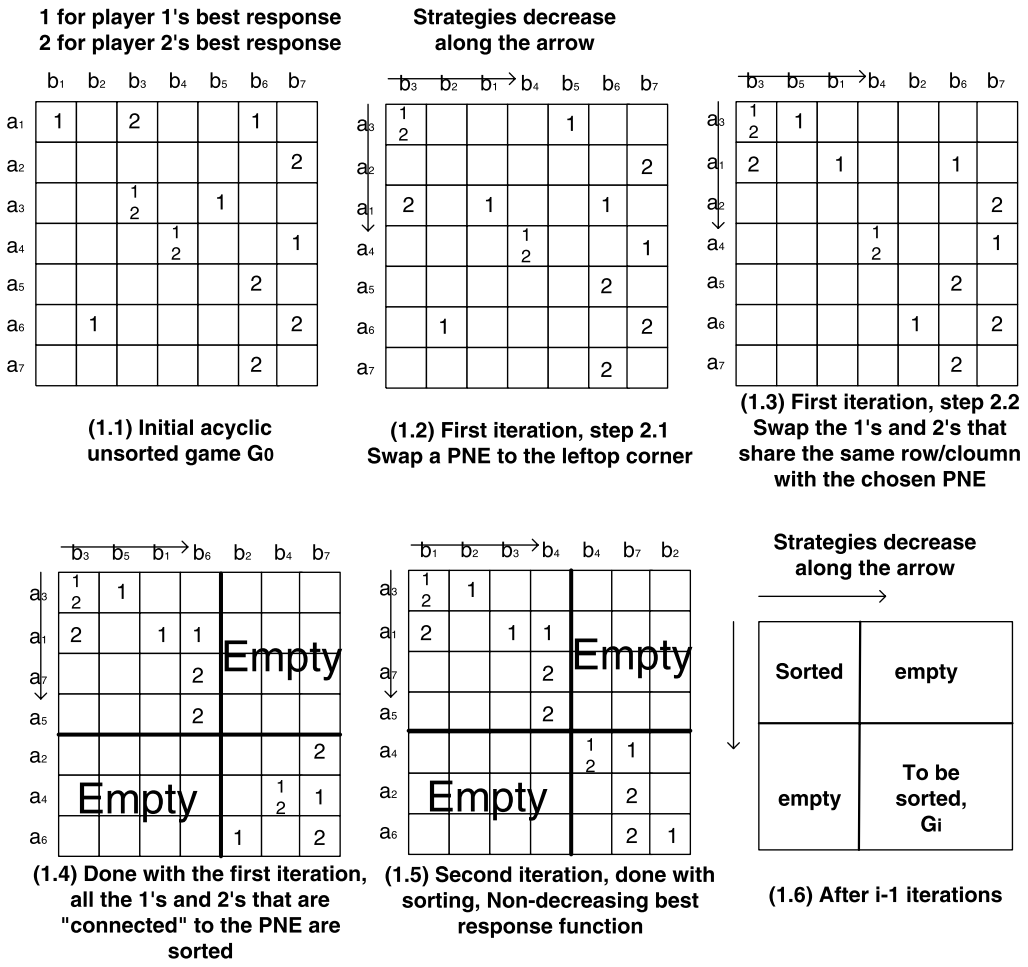


Fig. 1. Sorting the best-response matrix of a 7×7 acyclic game.

4.1. An illustrating example of the algorithm

We illustrate the algorithm for proving Lemma 4.2 using an example.

Consider a 7×7 2-player strict game whose best-response matrix is given in Fig. 1(1.1). In the matrix, the 1's indicate the player 1's best response, and 2's the player 2's best response. So the number 1 in the cell (a_1, b_1) means that the player 1's best response to b_1 is a_1, and the number 2 in (a_1, b_3) means that the player 2's best response to a_1 is b_3, and so on. This is a game that has no best-response cycles. However, if we order player 1's strategies from top to down, and player 2's strategies from left to right, then the best response functions are not non-decreasing. For instance, $b_3 <_2 b_6$, but $B_1(b_3) = a_3 >_1 a_1 = B_1(b_6)$. We now show how the procedure in our proof of Lemma 4.2 will re-arrange the two players' strategies so make the best response functions non-decreasing. The procedure is iterative. Step 2.1 finds a PNE, and moves it to the left top corner, as shown in Fig. 1(1.2). Step 2.2, as shown in Fig. 1(1.3) and (1.4), keeps swapping rows and columns until the matrix is divided into four regions: The top-left one is sorted, the top-right and the bottom-left ones have no best response entries in them and the bottom-right one is a sub-matrix that needs to be sorted recursively. For this particular game, this sub-matrix is sorted in one more iteration as shown in Fig. 1(1.5). In general, the number of iterations needed is the same as the number of PNEs in the game.

4.2. Discussions

Theorem 2 still holds for non-degenerate games, since the only assumption we made is that the best-response functions are single valued.

For general two person games, there are three ways to define a best response cycle:

1. Normal cycle. In this definition, as long as any best-response function is multi-valued, there is always a trivial cycle in which one player deviates among multiple best-responses.

Please cite this article in press as: Tang, P., Lin, F. Two equivalence results for two-person strict games. Games Econ. Behav. (2010), doi:10.1016/j.geb.2010.04.007

2. Voorneveld's cycle. This definition is essentially the same to above except that it requires at least one edge in the cycle that strictly increases the payoff for the deviating player. In this way, it rules out the trivial cycle in our original definition.
3. Strict cycle. In this definition, every edge in the cycle must strictly increases the payoff for the deviating player.

Three definitions are equivalent when consider strict games only. We now discuss the extension of Theorem 2 to general 2-person games with respect to three types of cycles.

The "only if" part of Theorem 2 does not hold for general 2-person games for normal cycle and Voorneveld's cycle but still holds for strict cycle. For instance, the following general game:

1, 1	2, 1
2, 2	1, 2

is a quasi-supermodular game under the ordering that orders player 1's (row player) strategies from top to down and player 2's action from left to right. However, going counterclockwise starting in any cell will form both a normal cycle as well as Voorneveld's cycle. Thus this game is not best-response equivalent to any ordinal potential game. However, according to Theorem 3 in Kukushkin et al. (2005), there is no strict cycle if a game is quasi-supermodular (therefore no strict cycle for a game that is best-response equivalent to it).

The "if" part of theorem 2 does not hold for general 2-person games for Voorneveld's as well as strict cycles but holds for normal cycles. To show this, consider the following general game:

2, 2	2, 1	2, 2
1, 1	1, 2	1, 2
1, 2	1, 2	1, 1

It has no Voorneveld's nor strict cycles. However it is not best-response equivalent to any quasi-supermodular game since there is no ordering of strategies under which player 2's best-response function satisfies single crossing condition. On the other hand, if we assume that a game has no normal cycle, it implies that its best-response functions are single-valued. Therefore, the if part holds for normal cycles.

Acknowledgments

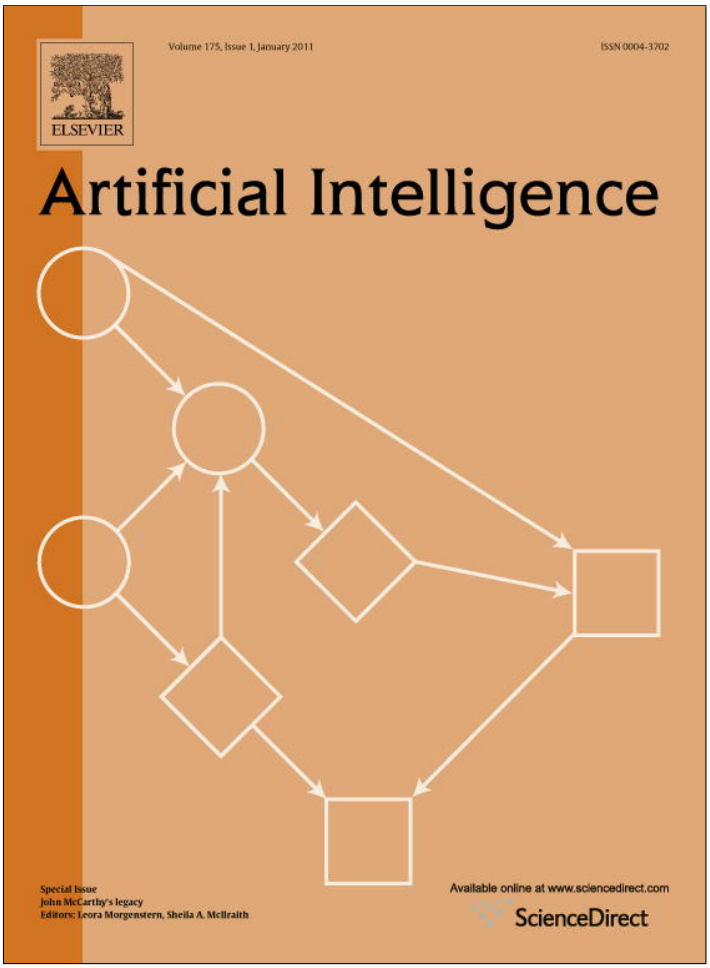
We are grateful to the GEB advisory editor and three reviewers for their helpful comments and suggestions. This work was supported in part by HK RGC GRFs 616707 and 616208.

References

Berger, U., 2007. Two more classes of games with the continuous-time fictitious play property. Games Econ. Behav. 60 (2), 247–261.
Echenique, F., 2004. A characterization of strategic complementarities. Games Econ. Behav. 46 (2), 325–347.
Friedman, J., 1983. On characterizing equilibrium points in two person strictly competitive games. Int. J. Game Theory 12, 245–247.
Kukushkin, N.S., Takahashi, S., Yamamori, T., 2005. Improvement dynamics in games with strategic complementarities. Int. J. Game Theory 33 (2), 229–238.
Monderer, D., Shapley, L.S., 1996. Potential games. Games Econ. Behav. 14, 124–143.
Moulin, H., 1976. Cooperation in mixed equilibrium. Math. Methods Operations Res. 1, 273–286.
Osborne, M.J., Rubinstein, A., 1994. A Course in Game Theory. MIT Press.
Rosenthal, R., 1974. Correlated equilibria in some classes of two-person games. Int. J. Game Theory 3, 119–128.
Tang, P. Lin, F., 2009. Discovering theorems in game theory: Two-person games with unique pure Nash equilibrium payoffs. In: IJCAI, pp. 312–317.
Topkis, D.M., 1998. Supermodularity and Complementarity. Princeton Univ. Press.
Voorneveld, M., 2000. Best-response potential games. Econ. Letters 66 (3), 289–295.
Zhou, L., 1994. The set of Nash equilibria of a supermodular game is a complete lattice. Games Econ. Behav. 7 (2), 295–300.

Please cite this article in press as: Tang, P., Lin, F. Two equivalence results for two-person strict games. Games Econ. Behav. (2010), doi:10.1016/j.geb.2010.04.007

Provided for non-commercial research and education use.
Not for reproduction, distribution or commercial use.



This article appeared in a journal published by Elsevier. The attached copy is furnished to the author for internal non-commercial research and education use, including for instruction at the authors institution and sharing with colleagues.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

http://www.elsevier.com/copyright

Author's personal copy

Artificial Intelligence 175 (2011) 264–277

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

From answer set logic programming to circumscription via logic of GK

Fangzhen Lin [a,*], Yi Zhou [b]

[a] Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong
[b] School of Computing and Mathematics, University of Western Sydney, Penrith South DC, NSW 1797, Australia

ARTICLE INFO

Article history:
Available online 3 April 2010

Keywords:
Logic programming
Answer set programming
Logic of GK
Circumscription
Nonmonotonic reasoning
Knowledge representation and reasoning

ABSTRACT

We first embed Pearce's equilibrium logic and Ferraris's propositional general logic programs in Lin and Shoham's logic of GK, a nonmonotonic modal logic that has been shown to include as special cases both Reiter's default logic in the propositional case and Moore's autoepistemic logic. From this embedding, we obtain a mapping from Ferraris's propositional general logic programs to circumscription, and show that this mapping can be used to check the strong equivalence between two propositional logic programs in classical logic. We also show that Ferraris's propositional general logic programs can be extended to the first-order case, and our mapping from Ferraris's propositional general logic programs to circumscription can be extended to the first-order case as well to provide a semantics for these first-order general logic programs.

© 2010 Elsevier B.V. All rights reserved.

Prologue

It gives us great pleasure to be able to contribute this work to this special issue of *Artificial Intelligence* in honor of John McCarthy. Like so many others, we have been influenced greatly by McCarthy and his work for as long as we have known AI. This particular work relates McCarthy's circumscription to several other nonmonotonic logics, and obviously could not have been done without McCarthy's pioneering work on nonmonotonic reasoning in general and circumscription in particular.

1. Introduction

Answer Set Programming (ASP) is a new paradigm of constraint-based programming based on logic programming with answer set semantics [9,13,17]. It started out with normal logic programs, which are programs that can have negation but not disjunction. Driven by the need of applications, various extensions have been proposed. These include disjunctive logic programs [5,6], nested expressions [7], cardinality and weight constraints [16], and others. Recently, Ferraris [2] proposed to view formulas in propositional logic as logic programs and showed that they include as special cases all the above mentioned classes of logic programs. In particular, Ferraris [2] provided a stable model semantics for these formulas using a transformation similar to the original Gelfond–Lifschitz transformation, and showed that this semantics coincides with Pearce's equilibrium logic [19].

In this paper, we show that this general stable model semantics can be embedded in Lin and Shoham's logic of GK (Grounded Knowledge) [11]. Besides showing the generality of Lin and Shoham's logic, which was proposed as a general logic for nonmonotonic reasoning, this embedding allows us to obtain a way to check in classical propositional logic whether any given two logic programs are strongly equivalent in almost the same way as in [12]. It also allows us to obtain a

* Corresponding author.
E-mail address: flin@cs.ust.hk (F. Lin).

0004-3702/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2010.04.001

Author's personal copy

mapping from general logic programs to propositional circumscription in the same way as Lin and Shoham [11] did for mapping normal logic programs to circumscription. As it turned out, this mapping, when extended to first-order case, yields a semantics to first-order general logic programs that is similar to the one proposed recently by Ferraris et al. [4].

We first briefly review Lin and Shoham's logic of GK, Ferraris's general logic programs, and Pearce's equilibrium logic.

2. Logic of GK

The language of the logic of GK is a modal propositional language with two modal operators, K, for knowledge, and A, for assumption. Given a set *Atom* of atoms (also called variables or primitive propositions), formulas in the logic of GK are defined inductively below in BNF notation:

$$F ::= \bot \mid p \mid K(F) \mid A(F) \mid \neg F \mid F \wedge F \mid F \vee F \mid F \rightarrow F,$$

where $p \in Atom$, and $\bot$ is a constant standing for falsity. Formulas without modal operators are called *base formulas*.

The semantics of the logic of GK is defined through *Kripke interpretations*. A Kripke interpretation M is a tuple $\langle W, \pi, R_K, R_A, s\rangle$, where W is a nonempty set whose elements are called *possible worlds*, π is a function that maps a possible world to a truth assignment on *Atom*, R_K and R_A are binary relations over W representing the accessibility relations for K and A, respectively, and $s \in W$, called the *actual world* of M. The *satisfaction relation* $\models$ between a Kripke interpretation $M = \langle W, \pi, R_K, R_A, s\rangle$ and a formula F is defined inductively as follows:

- $M \not\models \bot$;
- $M \models p$ if $\pi(s)(p) = 1$, where $p \in Atom$;
- $M \models \neg F$ iff $M \not\models F$;
- $M \models F \wedge G$ iff $M \models F$ and $M \models G$;
- $M \models F \vee G$ iff $M \models F$ or $M \models G$;
- $M \models F \rightarrow G$ iff $M \not\models F$ or $M \models G$;
- $M \models K(F)$ iff $\langle W, \pi, R_K, R_A, w\rangle \models F$ for any $w \in W$, such that $(s, w) \in R_K$;
- $M \models A(F)$ iff $\langle W, \pi, R_K, R_A, w\rangle \models F$ for any $w \in W$, such that $(s, w) \in R_A$.

We say that a Kripke interpretation M is a *model* of a formula F if M satisfies F. In the following, given a Kripke interpretation M, we let

$$K(M) = \{F \mid F \text{ is a base formula and } M \models K(F)\},$$
$$A(M) = \{F \mid F \text{ is a base formula and } M \models A(F)\}.$$

Notice that $K(M)$ and $A(M)$ are always closed under classical logical entailment. In the following, for any set X of formulas, we let $Th(X)$ be the logical closure of X under classical logic.

Informally in GK, one assumes $A(M)$ and minimizes $K(M)$. When the assumed $A(M)$ turns out to be the same as the minimal $K(M)$, an equilibrium is reached, and the assumption is said to be "justified" or the knowledge is said to be "grounded". Formally, GK models are defined as follows.

Definition 2.1 *(GK models).* Given a formula F, a Kripke interpretation M is a *minimal model* of F if M is a model of F and there does not exist another model M_1 of F such that $A(M_1) = A(M)$ and $K(M_1) \subset K(M)$. We say that M is a *GK model*[1] if M is a minimal model of F and $K(M) = A(M)$.

Lin and Shoham showed that the logic of GK can be used to capture Reiter's default logic [20] and Moore's auto-epistemic logic [15]. As a consequence, normal logic programs under stable model semantics can be captured in the logic of GK as well. Specifically, they showed that a normal rule

$$r \leftarrow p_1, \ldots, p_n, \text{not}\, q_1, \ldots, \text{not}\, q_m$$

can be translated into the following sentence in the logic of GK:

$$Kp_1 \wedge \cdots \wedge Kp_n \wedge \neg Aq_1 \wedge \cdots \wedge \neg Aq_m \rightarrow Kr. \tag{1}$$

They also showed that this translation extends to disjunctive logic programs.

In this paper, we shall show that general logic programs proposed by Ferraris [2] can be captured in the logic of GK as well.

[1] In [11], GK models are called preferred models.

Author's personal copy

3. General logic programs

Given a set *Atom* of atoms, general logic programs [3] are formulas defined inductively below in BNF notation:

$$F ::= \bot \mid p \mid F \wedge F \mid F \vee F \mid F \to F,$$

where $p \in Atom$. Notice that there is no negation in the language. Instead, for any formula F, $\neg F$ is considered to be a shorthand for $F \to \bot$.

A set $X \subseteq Atom$ of atoms can be considered as a truth assignment in the straightforward way:

$$X \not\models \bot, \qquad X \models p \quad \text{iff} \quad p \in X,$$

and the usual definition for the logical connectives.

The stable models of a formula (general logic program) are defined by a modified extended Gelfond–Lifschitz transformation. Given a general logic program F, and a set X of atoms, the *reduct of F under X* [2], written F^X, is the formula obtained from F by replacing each maximal subformula that is not classically satisfied by X with $\bot$. Thus for example,

$$(\neg F)^X = \begin{cases} \top, & X \models \neg F, \\ \bot, & \text{otherwise.} \end{cases}$$

Now a set X of atoms is a stable model of a general logic program F if:

(i) $X \models F^X$;
(ii) there is no proper subset X_1 of X, such that $X_1 \models F^X$.

Example 3.1. Consider the following three general logic programs.

$$P = \neg p \to q,$$
$$Q = \neg p \vee p,$$
$$R = p \to \neg\neg p,$$

where p, q are atoms. The maximal subformula in P that is false under $\{q\}$ is p, thus $P^{\{q\}}$ is $\neg\bot \to q$, which is satisfied by $\{q\}$, but not by $\emptyset$. Therefore, $\{q\}$ is a stable model of P. On the other hand, $P^{\{p\}}$ is $\bot \to \bot$, which is satisfied by $\{p\}$ as well as its subset $\emptyset$. Therefore, $\{p\}$ is not a stable model of P. It can be seen that $\{q\}$ is the only stable model of P. Similarly, it can be shown that Q has two stable models, $\{p\}$ and $\emptyset$, and R has exactly one stable model $\emptyset$.

4. Pearce's equilibrium logic

Pearce's equilibrium logic [19] is based on the logic of here-and-there, a non-classical logic. Given a set *Atom* of atoms, formulas of *Atom* are exactly the same as in the case of general logic programs. Thus, negation in equilibrium logic is considered a shorthand as well.

The semantics of the logic of here-and-there is defined in terms of *HT-interpretations*, which are pairs $\langle X, Y\rangle$ of sets of atoms such that $X \subseteq Y$. The *HT satisfaction relation*[2] $\models$ between an HT-interpretation $\langle X, Y\rangle$ and a formula F is defined recursively as follows:

- For $p \in Atom$, $\langle X, Y\rangle \models p$ if $p \in X$;
- $\langle X, Y\rangle \not\models \bot$;
- $\langle X, Y\rangle \models F \wedge G$ if $\langle X, Y\rangle \models F$ and $\langle X, Y\rangle \models G$;
- $\langle X, Y\rangle \models F \vee G$ if $\langle X, Y\rangle \models F$ or $\langle X, Y\rangle \models G$;
- $\langle X, Y\rangle \models F \to G$ if
  (i) $\langle X, Y\rangle \not\models F$ or $\langle X, Y\rangle \models G$, and
  (ii) $Y \models F \to G$.

An HT interpretation $\langle X, Y\rangle$ is an *equilibrium model* of a formula F if $X = Y$, $\langle X, Y\rangle \models F$, and there is no proper subset X_1 of X, such that $\langle X_1, Y\rangle \models F$. Ferraris [2] showed that the stable models of a formula are essentially the same as its equilibrium models.

Theorem 1 *(Ferraris). Let X be a set of atoms and F a general logic program, X is a stable model of F iff $\langle X, X\rangle$ is an equilibrium model of F.*

[2] We overload $\models$, and use it to stand for satisfaction relations for modal logic, classical logic, and logic of here-and-there. Which one it stands for should be clear from the context.

Author's personal copy

5. From general logic programs and equilibrium logic to the logic of GK

In this section, we present a translation from a general logic program (also a formula in equilibrium logic) to a formula in the logic of GK, and show that under the translation, stable models (thus equilibrium models) coincide with GK models in the logic of GK.

Given a general logic program F, we define two formulas F_A and F_{GK} in the logic of GK as follows:

(1) F_A is obtained from F by simultaneously replacing each atom p by Ap.
(2) F_{GK} is defined inductively as follows:
- $\bot_{GK} = \bot$;
- For $p \in Atom$, $p_{GK} = Kp$;
- $(F \odot G)_{GK} = F_{GK} \odot G_{GK}$ ($\odot$ is $\wedge$ or $\vee$);
- $(F \to G)_{GK} = (F_{GK} \to G_{GK}) \wedge (F \to G)_A$.

It can be shown that for a normal logic program F, F_{GK} is equivalent to the translation by Lin and Shoham [11] given in Section 2 under

$$\bigwedge_{p \in Atom} Kp \to Ap, \tag{2}$$

and that for any formula W in the logic of GK, M is a GK model of W iff M is a GK model of $W \wedge (2)$.

This translation is also similar to the mapping from formulas in equilibrium logic to quantified boolean formulas given in [18]. We shall discuss this in more detail in a later section.

To illustrate, consider the three programs in Example 3.1. P_{GK} is

$$\big((\neg p)_{GK} \to q_{GK}\big) \wedge (\neg p \to q)_A,$$

which is

$$\big((\neg p_{GK} \wedge \neg p_A) \to Kq\big) \wedge (\neg p_A \to q_A),$$

which is

$$\big((\neg Kp \wedge \neg Ap) \to Kq\big) \wedge (\neg Ap \to Aq). \tag{3}$$

Now let M be a model of the above sentence. If $p \in A(M)$, then (3) holds no matter what $K(M)$ is, thus its minimal model is $K(M) = Th(\emptyset)$, so cannot be a GK model. Now if $p \notin A(M)$, then (3) is equivalent to $(\neg Kp \to Kq) \wedge Aq$. Thus $q \in A(M)$. Thus if M is a minimal model, then $K(M) = Th(\{q\})$. And if $A(M) = K(M)$, then M is a GK model. What we have shown here is that in any GK model M of (3), $K(M) = A(M) = Th(\{q\})$. The existence of such a model is apparent.

It can be similarly shown that Q_{GK} is equivalent to $Ap \to Kp$, and that M is a GK model of Q_{GK} iff $K(M) = A(M) = Th(\{p\})$ or $K(M) = A(M) = Th(\{\top\})$. And R_{GK} is equivalent to $Kp \to Ap$, and that M is a GK model of R_{GK} iff $K(M) = A(M) = Th(\{\top\})$. Thus for these three programs, their GK models correspond one-to-one with their stable models. In general, we have the following result. Proofs of the theorems are in Appendix A.

Theorem 2. *Let X be a set of atoms and F a general logic program. The following two statements are equivalent.*

(1) *X is a stable model of F.*
(2) *There is a GK model M of F_{GK} such that $K(M) = A(M) = Th(X)$.*

6. From general logic programs and equilibrium logic to circumscription

Given their mapping (1) from normal logic program to the logic of GK, Lin and Shoham [11] showed that stable model semantics for normal logic programs can be captured in circumscription [14] as follows. Given a set $Atom = \{p, q, \ldots\}$ of atoms, let $Atom' = \{p', q', \ldots\}$ be a new set of atoms. Given a normal logic program F, let $C(F)$ be the conjunction of the sentences:

$$p_1 \wedge \cdots \wedge p_n \wedge \neg q'_1 \wedge \cdots \wedge \neg q'_m \to r,$$

for each rule

$$r \leftarrow p_1, \ldots, p_n, \mathrm{not}\, q_1, \ldots, \mathrm{not}\, q_m$$

in F. Lin and Shoham [11] showed that X is a stable model of F iff $X \cup X'$ is a model of

Author's personal copy

$$\bigwedge_{p\in Atom}(p\leftrightarrow p')\wedge Circum\big(C(F);Atom\big),$$

where $Circum(W;Q)$ is the circumscription of the atoms in Q in W (with all other atoms fixed). Lin and Shoham also showed that this result can be extended to disjunctive logic programs. Using the same idea, we can capture the stable model semantics of general logic program and equilibrium logic in circumscription as well.

Let $Atom$ be a set of atoms. Again let $Atom'=\{p'\mid p\in Atom\}$ be a set of new atoms. For any $X\subseteq Atom$, let $X'=\{p'\mid p\in X\}$. Given any general logic program F in the language $Atom$, let $C(F)$ be the result obtained from F_{GK} by replacing every Kp in it by p and every Ap in it by p', for every $p\in Atom$.

Theorem 3. *For any general logic program F in the language $Atom$, any set $X\subseteq Atom$, the following two statements are equivalent*

(1) *There is a GK model M of F_{GK} such that $K(M)=A(M)=Th(X)$.*
(2) *$X\cup X'$ is a model of*

$$\bigwedge_{p\in Atom}(p\leftrightarrow p')\wedge Circum\big(C(F);Atom\big). \tag{4}$$

Interestingly, our translation $C(F)$ that embeds general logic programs and equilibrium logic in circumscription is exactly the same as the one by Pearce et al. [18] for embedding equilibrium logic in quantified boolean formulas. They showed that $\langle X,X\rangle$ is an equilibrium model of a formula F in equilibrium logic iff X' is a model of the following quantified boolean formula:

$$F'\wedge\neg\exists Atom\big(\big(Atom<Atom'\big)\wedge C(F)\big),$$

where F' is the formula obtained from F by replacing every atom p by p', and $Atom<Atom'$ stands for

$$\bigwedge_{p\in Atom}(p\rightarrow p')\wedge\neg\bigwedge_{p\in Atom}(p'\rightarrow p).$$

While propositional circumscription is also a quantified boolean formula, it is a well studied formalism. There are many known results about circumscription that we can use. Mapping logic programs to circumscription can help us better understand both formalisms.

Notice that the formula $\bigwedge_{p\in Atom}(p\leftrightarrow p')$ is equivalent to

$$\Big[\bigwedge_{p\in Atom}(p\rightarrow p')\Big]\wedge\Big[\bigwedge_{p\in Atom}(p'\rightarrow p)\Big].$$

Thus (4) is equivalent to

$$\Big[\bigwedge_{p\in Atom}(p\rightarrow p')\Big]\wedge\Big[\bigwedge_{p\in Atom}(p'\rightarrow p)\Big]\wedge Circum\big(C(F);Atom\big),$$

which is equivalent to

$$\Big[\bigwedge_{p\in Atom}(p'\rightarrow p)\Big]\wedge Circum\Big(C(F)\wedge\bigwedge_{p\in Atom}(p\rightarrow p');Atom\Big),$$

as the atoms (predicates) to be minimized occur only negatively in $\bigwedge_{p\in Atom}(p\rightarrow p')$. Putting the formula $\bigwedge_{p\in Atom}(p\rightarrow p')$ into the theory in the circumscription is good as it can be used to simplify $C(F)$.

Proposition 6.1. *If $\bigwedge_{p\in Atom}(p\rightarrow p')\models C(F)\leftrightarrow W$, then (4) is equivalent to*

$$\bigwedge_{p\in Atom}(p\leftrightarrow p')\wedge Circum(W;Atom).$$

Consider again the three programs in Example 3.1. For P, $C(P)$ is equivalent to $\neg p\wedge\neg p'\rightarrow q$, which is equivalent to $\neg p'\rightarrow q$ under $(p\rightarrow p')\wedge(q\rightarrow q')$. Thus for this program, (4) is equivalent to

$$\big(p\leftrightarrow p'\big)\wedge\big(q\leftrightarrow q'\big)\wedge Circum\big(\neg p'\rightarrow q;\{p,q\}\big),$$

which is equivalent to

Author's personal copy

$$\big(p \leftrightarrow p'\big) \wedge \big(q \leftrightarrow q'\big) \wedge \neg p \wedge \big(\neg p' \leftrightarrow q\big),$$

which has a unique model $\{q, q'\}$.

For $Q = \neg p \vee p$, $C(Q)$ is equivalent to $p' \rightarrow p$. Thus for this program, (4) is equivalent to

$$\big(p \leftrightarrow p'\big) \wedge \mathit{Circum}\big(p' \rightarrow p; \{p\}\big),$$

which is equivalent to $p \leftrightarrow p'$, which has two models p and $\neg p$.

7. First-order general logic programs and circumscription

As in [11], we can extend the above mapping to the first-order case. First of all, we extend logic programs to the first-order case. Let L be a relational first-order language with equality, i.e. it has no proper functions. By an atom, we mean an atomic formula including equality atoms.

In the following, let Σ_{una} be the set of unique names assumptions on constants: $c_1 \neq c_2$ for any two distinct constants c_1 and c_2.

A first-order general logic program is a first-order sentence in the following set:

$$F ::= \bot \mid A \mid F \wedge F \mid F \vee F \mid F \rightarrow F \mid \forall x F \mid \exists x F,$$

where A is an atom, and x a variable. Again, for any general logic program F, $\neg F$ is considered to be a shorthand for $F \rightarrow \bot$.

Now let M be a finite model of Σ_{una} with domain D. Let σ be the mapping from constants to D under M. Clearly, for any distinct constants c_1 and c_2, $\sigma(c_1) \neq \sigma(c_2)$. We say that M is a stable model of a first-order general logic program F if $T(M)$, the set of ground facts true in M:

$$T(M) = \big\{P(\vec{u}) \bigm| P \text{ a predicate}, \vec{u} \in P^M\big\}$$

is a stable model of the general logic program F_M, the *grounding* of F on M obtained from F and M in two steps:

(1) First, replace every constant c in F by $\sigma(c)$, every subformula of the form $\forall x W$ in it by $\bigwedge_{u \in D} W(x/u)$, and every subformula of the form $\exists x W$ in it by $\bigvee_{u \in D} W(x/u)$, where $W(x/u)$ is the result of replacing every free occurrence of x in W by u. The order by which these subformulas are replaced does not matter.
(2) In the expression obtained by the first step, replace every equality atom $u = u$ by $\top$, and every $u = u'$ for distinct u and u' by $\bot$.

Example 7.1. Consider the following four programs:

$$F_1 = \exists x p(x) \wedge \exists x\big(\neg p(x) \rightarrow q\big),$$
$$F_2 = \exists x p(x) \wedge \big[\big(\exists x \neg p(x)\big) \rightarrow q\big],$$
$$F_3 = \exists x p(x) \wedge \big[\neg\big(\exists x p(x)\big) \rightarrow q\big],$$
$$F_4 = \exists x p(x) \wedge \forall x\big(\neg p(x) \rightarrow q\big).$$

Now consider a structure with two elements $\{a, b\}$. The grounding of the four programs on this domain are

$$\big(p(a) \vee p(b)\big) \wedge \big(\big(\neg p(a) \rightarrow q\big) \vee \big(\neg p(b) \rightarrow q\big)\big),$$
$$\big(p(a) \vee p(b)\big) \wedge \big(\big(\neg p(a) \vee \neg p(b)\big) \rightarrow q\big),$$
$$\big(p(a) \vee p(b)\big) \wedge \big(\neg\big(p(a) \vee p(b)\big) \rightarrow q\big),$$
$$\big(p(a) \vee p(b)\big) \wedge \big(\neg p(a) \rightarrow q\big) \wedge \big(\neg p(b) \rightarrow q\big),$$

respectively. So for this domain, F_1 and F_3 have the same stable models, $\{p(a)\}$ and $\{p(b)\}$, and F_2 and F_4 have the same stable models, $\{p(a), q\}$ and $\{p(b), q\}$. It is easy to see that this is the case for any given domain: F_1 and F_3 have the same stable models, and F_2 and F_4 have the same stable models.

We now show that the stable models of first-order general logic programs can be captured in circumscription as well.

Let Δ be the set of predicates in the language. Let Δ' be a set of new predicates, one for each P in Δ with the same arity and denoted by P'. Now given a first-order general logic program F, let $C(F)$ be the first-order formula defined inductively as follows.

- $C(\bot)$ is $\bot$.
- If W is an atomic formula, then $C(W)$ is W.

Author's personal copy

- $C(W_1 \odot W_2)$ is $C(W_1) \odot C(W_2)$, where $\odot \in \{\wedge, \vee\}$.
- $C(\forall x W)$ is $\forall x C(W)$, and $C(\exists x W)$ is $\exists x C(W)$.
- $C(W_1 \rightarrow W_2)$ is $(C(W_1) \rightarrow C(W_2)) \wedge (W_1' \rightarrow W_2')$, where W' is the result of replacing every predicate P in W by P'.

The stable models of F are then the models of the circumscription of all the predicates in Δ in $C(F)$, together with the following axiom

$$\bigwedge_{P \in \Delta} \forall \vec{x}\big(P(\vec{x}) \leftrightarrow P'(\vec{x})\big). \tag{5}$$

Theorem 4. *Let M be a finite model of Σ_{una}. M is a stable model of F iff M' is a model of*

$$Circum\big(C(F); \Delta\big) \wedge (5), \tag{6}$$

where M' is the conservative extension of M under (5)*.*

Similar to Proposition 6.1, we have

Proposition 7.1. *If $\bigwedge_{P \in \Delta} \forall \vec{x}(P(\vec{x}) \rightarrow P'(\vec{x})) \models C(F) \leftrightarrow W$, then* (6) *is equivalent to*

$$Circum(W; \Delta) \wedge (5).$$

Example 7.2. Consider the programs in Example 7.1.

- $C(F_1)$ is

  $$\exists x p(x) \wedge \exists x\big[\big(\neg p(x) \wedge \neg p'(x) \rightarrow q\big) \wedge \big(\neg p'(x) \rightarrow q'\big)\big],$$

  which is equivalent to $\exists x p(x) \wedge (\neg \exists x p'(x) \rightarrow q)$ under $\forall x.p(x) \rightarrow p'(x)$. Therefore, under (5), $Circum(C(F_1), \{p, q\})$ is equivalent to

  $$\exists! x p(x) \wedge \big(\big(\neg \exists x p'(x)\big) \leftrightarrow q\big),$$

  thus equivalent to

  $$\exists! x p(x) \wedge \neg q,$$

  which can be considered to be the first-order semantics of F_1. If $D = \{a, b\}$, then there are exactly two models of this sentence, $\{p(a)\}$ and $\{p(b)\}$.
- $C(F_2)$ is

  $$\exists x p(x) \wedge \big(\exists x\big(\neg p(x) \wedge \neg p'(x)\big) \rightarrow q\big) \wedge \big(\exists x \neg p'(x) \rightarrow q'\big),$$

  which is equivalent to

  $$\exists x p(x) \wedge \big(\exists x \neg p'(x) \rightarrow q\big)$$

  under $\forall x.p(x) \rightarrow p'(x)$. Therefore, $Circum(C(F_2), \{p, q\})$ is equivalent to

  $$\exists! x p(x) \wedge \big(\exists x \neg p'(x) \leftrightarrow q\big),$$

  under (5), thus equivalent to

  $$\exists! x p(x) \wedge \big(\big(\exists x \neg p(x)\big) \leftrightarrow q\big),$$

  which can be considered to be the first-order semantics of F_2. If $D = \{a, b\}$, then there are exactly two models of this sentence, $\{p(a), q\}$ and $\{p(b), q\}$.
- $C(F_3)$ is

  $$\exists x p(x) \wedge \big(\neg \exists x p(x) \wedge \neg \exists x p'(x) \rightarrow q\big) \wedge \big(\neg \exists x p'(x) \rightarrow q'\big),$$

  which is equivalent to $C(F_1)$ under (5). Thus F_1 and F_3 are equivalent under our semantics.
- $C(F_4)$ is

  $$\exists x p(x) \wedge \forall x\big[\big(\neg p(x) \wedge \neg p'(x) \rightarrow q\big) \wedge \big(\neg p'(x) \rightarrow q'\big)\big],$$

  which is equivalent to $C(F_2)$. Thus F_2 and F_4 are equivalent under our semantics.

Author's personal copy

We have defined the stable model semantics for first-order logic programs by reducing them to propositional logic programs through grounding. For this to work, we need to assume a finite domain for otherwise a first-order sentence such as $\exists x p(x)$ cannot be replaced by a propositional sentence. We also need the unique names assumption in order to eliminate equality literals such as $a = b$ or $a \neq b$.

Theorem 4 shows that this stable model semantics for first-order logic programs can be captured by a simple formula (6) using circumscription. However, while the stable model semantics makes the unique names and finite models assumptions, the formula (6) is completely general and makes sense without these assumptions. Thus one could use this formula to define the stable model semantics of first-order logic programs in the general case. As it turned out, this will yield a semantics that is essentially the same as the one proposed recently by Ferraris et al. [4]. More precisely, given a first-order sentence F, Ferraris et al. proposed a second-order sentence as its stable model semantics, and showed that this second-order sentence is equivalent to (6) with all new predicates existentially quantified: $\exists \vec{p} \varphi(\vec{P'}/\vec{p})$, where φ is formula (6), $\vec{p}$ a tuple of predicate variables, one for each predicate P' in φ and with the same arity, and $\varphi(\vec{P'}/\vec{p})$ is the result of replacing each P' in φ by its corresponding variable in $\vec{p}$.

8. Strong equivalence

The notion of strong equivalence [8] is important in logic programming. For disjunctive logic programs, research by Lin and Chen [10] and Eiter et al. [1] show that interesting programs transformation rules can be designed based on the notion.

According to Ferraris and Lifschitz [3], two general logic programs F and G are said to be *strongly equivalent* if for any formula F_1 that contains an occurrence of F, F_1 has the same stable models as the formula obtained from it by replacing an occurrence of F by G. They showed that for any F and G, they are strongly equivalent iff F and G are equivalent in the logic here-and-there.

As it turns out, our mapping from equilibrium logic to logic of GK also embeds logic of here-and-there to modal logic. Thus the problem of deciding whether two programs are strongly equivalent can be reduced to checking whether certain modal logic formulas are valid, and that, because of the special format of these modal formulas, can in turn be reduced to checking whether certain propositional formulas are tautologies.

Theorem 5. *Let F be a formula in equilibrium logic, X and Y two sets of atoms such that $X \subseteq Y$, and M a Kripke interpretation such that $K(M) = Th(X)$ and $A(M) = Th(Y)$. We have that $\langle X, Y \rangle \models F$ iff $M \models F_{GK}$.*

Theorem 6. *Let F and G be two general logic programs. The following conditions are equivalent.*

1. *F and G are strongly equivalent.*
2. $\bigwedge_{p \in Atom}(Kp \rightarrow Ap) \models (F \leftrightarrow G)_{GK}$.
3. $\bigwedge_{p \in Atom}(Kp \rightarrow Ap) \models F_{GK} \leftrightarrow G_{GK}$.
4. $\bigwedge_{p \in Atom}(p \rightarrow p') \models C(F \leftrightarrow G)$.
5. $\bigwedge_{p \in Atom}(p \rightarrow p') \models C(F) \leftrightarrow C(G)$.

Corollary 7. *The problem of deciding whether two general logic programs are strongly equivalent is co-NP complete.*

9. Conclusion

We showed that the logic of GK proposed by Lin and Shoham is flexible enough to handle stable model semantics of general logic programs. Because of this, the stable model semantics of general logic programs can also be formulated in circumscription, in both propositional and first-order cases. For future work, we plan to make use of the expressive power of GK in other applications.

Acknowledgements

We thank Vladimir Lifschitz for stimulating discussions on topics related to this paper, and for his helpful comments on earlier versions of this paper. This work was supported in part by HK RGC under grant HKUST6170/04E, and by China NSFC under grants 60703095 and 90718009.

Appendix A. Proofs of theorems

A.1. Proof of Theorem 2

Lemma 1. *Let F be a general logic program; X and Y two sets of atoms such that $X \subseteq Y$ and M a Kripke interpretation such that $K(M) = Th(X)$, $A(M) = Th(Y)$. We have that $X \models F^Y$ iff $M \models F_{GK}$.*

Author's personal copy

Proof. Let F and G be two general logic programs, $X \subseteq Y$ two sets of atoms. As shown in [3], $(F \wedge G)^X$ is equivalent to $F^X \wedge G^X$, and $(F \vee G)^X$ equivalent to $F^X \vee G^X$. Thus by induction on the structure of F, it's easy to see that $Y \models F$ iff $M \models F_A$, where M is as in the proposition.

We now prove that $X \models F^Y$ iff $M \models F_{GK}$ by induction on the structure of F.

- If F is $\bot$, then this assertion holds obviously.
- If F is an atom p, then p_{GK} is Kp. $X \models p^Y$ iff $p \in X$. On the other hand, $M \models Kp$ iff $p \in X$ too. Therefore, $X \models p^Y$ iff $M \models p_{GK}$.
- If F is $G \wedge H$, $X \models (G \wedge H)^Y$ iff $X \models G^Y$ and $X \models H^Y$. By induction hypothesis, this holds iff $M \models G_{GK}$ and $M \models H_{GK}$, which is equivalent to $M \models (G \wedge H)_{GK}$.
- If F is $G \vee H$, $X \models (G \vee H)^Y$ iff $X \models G^Y$ or $X \models H^Y$. By induction hypothesis, this holds iff $M \models G_{GK}$ or $M \models H_{GK}$, which is equivalent to $M \models (G \vee H)_{GK}$.
- Finally, if F is $G \to H$. $X \models (G \to H)^Y$ iff $Y \models G \to H$ and $X \models G^Y \to H^Y$ iff (i) $Y \models G \to H$ and (ii) $X \not\models G^Y$ or $X \models H^Y$. By induction hypothesis, this holds iff (i) $M \models (G \to H)_A$ and (ii) $M \not\models G_{GK}$ or $M \models H_{GK}$ iff $M \models (G \to H)_{GK}$.

This completes the induction proof. □

Theorem 2. *Let X be a set of atoms and F a general logic program. The following two statements are equivalent.*

(1) *X is a stable model of F.*
(2) *There is a GK model M of F_{GK} such that $K(M) = A(M) = Th(X)$.*

Proof. (1) $\Rightarrow$ (2): Suppose that X is a stable model of F. We can construct a Kripke interpretation M such that $K(M) = A(M) = Th(X)$. We now prove that M is a GK model of F_{GK}. Firstly, we have $X \models F^X$ since X is a stable model of F. By Lemma 1, $M \models F_{GK}$. We only need to prove M is a minimal model. We prove it by contradiction. Assume that M is not a minimal model of F_{GK}, then there exists another model M_1 of F_{GK} such that $A(M_1) = A(M) = Th(X)$ and $K(M_1) \subset K(M)$. We construct another Kripke interpretation M_2 such that $A(M_2) = Th(X)$ and $K(M_2) = Th(X_1)$, where $X_1 = \{p \mid M_1 \models Kp, p \in Atom\}$. For any general logic program G, by induction on the structure of G, it's easy to see that M_2 is a model of G_{GK} iff M_1 is a model of G_{GK}. Therefore, M_2 is a model of F_{GK}. By Lemma 1, $X_1 \models F^X$. Moreover, $X_1 \subset X$. This shows that X is not a stable model of F, a contradiction. Hence, M is a GK model of F_{GK}.

(1) $\Leftarrow$ (2): Suppose that there is a GK model M of F_{GK} such that $K(M) = A(M) = Th(X)$. By Lemma 1, $X \models F^X$. There is no proper subset X_1 of X such that $X_1 \models F^X$. Otherwise, we can construct a Kripke interpretation M_1 such that $K(M_1) = Th(X_1)$ and $A(M_1) = Th(X)$. By Lemma 1, M_1 is also a model of F_{GK}. Moreover $K(M_1) \subset K(M)$; $A(M_1) = A(M)$. This shows that M is not a GK model of F_{GK}, a contradiction. Hence, X is a stable model of F. □

A.2. Proof of Theorem 3

Notice that $C(F)$ can also be defined recursively as follows.

- $C(\bot)$ is $\bot$.
- If F is an atom, then $C(F)$ is F.
- $C(F_1 \odot F_2)$ is $C(F_1) \odot C(F_2)$, where $\odot$ is $\wedge$ or $\vee$.
- $C(F_1 \to F_2)$ is $(C(F_1) \to C(F_2)) \wedge (F_1' \to F_2')$, where F' is the expression obtained from F by replacing every atom p by p'.

Lemma 2. *Let F be a general logic program, X and Y two sets of atoms, and M a Kripke interpretation such that $K(M) = Th(X)$, $A(M) = Th(Y)$. We have that $X \cup Y'$ is a model of $C(F)$ iff $M \models F_{GK}$.*

Proof. We prove this by induction on the structure of F.

- If F is $\bot$, then this assertion holds obviously.
- If F is an atom p, then p_{GK} is Kp, $C(F)$ is p. $X \cup Y' \models p$ iff $p \in X$. On the other hand, $M \models Kp$ iff $p \in X$ too. Therefore, $X \cup Y' \models C(p)$ iff $M \models p_{GK}$.
- If F is $G \wedge H$, $X \cup Y' \models C(G \wedge H)$ iff $X \cup Y' \models C(G)$ and $X \cup Y' \models C(H)$. By induction hypothesis, this holds iff $M \models G_{GK}$ and $M \models H_{GK}$, which is equivalent to $M \models (G \wedge H)_{GK}$.
- If F is $G \vee H$, $X \cup Y' \models C(G \vee H)$ iff $X \cup Y' \models C(G)$ or $X \cup Y' \models C(H)$. By induction hypothesis, this holds iff $M \models G_{GK}$ or $M \models H_{GK}$, which is equivalent to $M \models (G \vee H)_{GK}$.
- Finally, if F is $G \to H$. $X \cup Y' \models C(G \to H)$ iff $X \cup Y' \models (C(G) \to C(H)) \wedge (G \to H)'$ iff $X \cup Y' \models (G \to H)'$ and $X \cup Y' \models C(G) \to C(H)$ iff (i) $X \cup Y' \models (G \to H)'$ and (ii) $X \cup Y' \not\models C(G)$ or $X \cup Y' \models C(H)$. Notice that $X \cup Y' \models (G \to H)'$ iff

Author's personal copy

$Y' \models (G \to H)'$ iff $Y \models G \to H$ iff $M \models (G \to H)_A$. By induction hypothesis, $X \cup Y' \models C(F)$ iff (i) $M \models (G \to H)_A$ and (ii) $M \not\models G_{GK}$ or $M \models H_{GK}$ iff $M \models (G \to H)_{GK}$.

This completes the induction proof. □

Theorem 3. *For any general logic program F in the language Atom, any set $X \subseteq Atom$, the following two statements are equivalent*

(1) *There is a GK model M of F_{GK} such that $K(M) = A(M) = Th(X)$.*
(2) *$X \cup X'$ is a model of*

$$\bigwedge_{p \in Atom} (p \leftrightarrow p') \wedge Circum\big(C(F); Atom\big). \tag{4}$$

Proof. (1) $\Rightarrow$ (2): Suppose that M is a GK model of F_{GK} and $K(M) = A(M) = Th(X)$. Then $M \models F_{GK}$, by Lemma 2, $X \cup X'$ is a model of $C(F)$. We now show that for any set of atoms such that $X_1 \subset X$, $X_1 \cup X'$ is not a model of $C(F)$. Suppose otherwise, $X_1 \cup X'$ is a model of $C(F)$. Construct a GK interpretation M_1 such that $K(M_1) = Th(X_1)$ and $A(M_1) = Th(X)$. By Lemma 2, $M_1 \models F_{GK}$. It's clear that $K(M_1) \subset K(M)$, $A(M_1) = A(M)$. This shows that M is not a GK model of F_{GK}, a contradiction. Therefore, $X \cup X'$ is a model of $Circum(C(F); Atom)$. Of course, $X \cup X'$ is a model of $\bigwedge_{p \in Atom}(p \leftrightarrow p')$. This shows that statement (1) implies statement (2).

(1) $\Leftarrow$ (2): Suppose that $X \cup X'$ is a model of $Circum(C(F); Atom)$. Thus $X \cup X'$ is a model of $C(F)$ and for every proper subset X_1 of X, $X_1 \cup X'$ is not a model of $C(F)$. Let M be a Kripke interpretation such that $K(M) = A(M) = Th(X)$. We now prove that M is a GK model of F_{GK}. According to Lemma 2, $M \models F_{GK}$. We only need to prove that M is a minimal model. We prove it by contradiction. Assume that M is not a minimal model of F_{GK}. Then there exists another model M_1 of F_{GK} such that $A(M_1) = A(M) = Th(X)$ and $K(M_1) \subset K(M)$. Construct another Kripke interpretation M_2 such that $A(M_2) = Th(X)$ and $K(M_2) = Th(X_1)$, where $X_1 = \{p \mid M_1 \models Kp, p \in Atom\}$. For any general logic program G, by induction on the structure of G, M_2 is a model of G_{GK} iff M_1 is a model of G_{GK}. Therefore, M_2 is a model of F_{GK}. By Lemma 2, $X_1 \cup X' \models C(F)$. Moreover, $X_1 \subset X$. $X \cup X'$ is not a model of $Circum(C(F); Atom)$, a contradiction. Hence, M is a GK model of F_{GK}. This shows that statement (2) implies statement (1). □

A.3. Proof of Theorem 4

Let L be a first order language with equality but without proper functions. Let C be the set of constants in L, and Δ the set of predicates in L. Let $\Delta' = \{P' \mid P \in \Delta\}$ be a set of new predicates, and L' the extension of L by the new predicates in Δ'.

A first-order structure M' of L' is said to be a conservative extension of a first-order structure M of L if M' and M have the same domain, and they interpret all symbols in L the same. Suppose that M' is a conservative extension of M. It is easy to see that if M is a model of Σ_{una}, then M' is also a model of Σ_{una}, and that for every sentence F, F_M and $F_{M'}$ are the same.

Let F be a first order general logic program in L, and M a finite model of Σ_{una}. We remark that F_M can also be defined recursively as follows (recall that σ is the mapping from constants in L to the domain of M under M).

- $\bot_M$ is $\bot$.
- If F is $a = b$, where a and b are two constants, then F_M is $\top$ when $\sigma(a)$ is the same as $\sigma(b)$, otherwise F_M is $\bot$.
- If F is an atomic formula $P(\vec{t})$, where $\vec{t}$ is a vector of constants, then F_M is $P(\sigma(\vec{t}))$, where $\sigma(\vec{t})$ is $\langle \sigma(t_1), \ldots, \sigma(t_n)\rangle$ when $\vec{t}$ is $\langle t_1, \ldots, t_n\rangle$.
- $(G \odot H)_M$ is $G_M \odot H_M$, where $\odot$ is $\wedge$, $\vee$ or $\to$.
- $(\exists x F)_M$ is $\bigvee_{u \in D} F(x/u)_M$.
- $(\forall x F)_M$ is $\bigwedge_{u \in D} F(x/u)_M$.

Thus given a first order general logic program F in L, F_M is a propositional formula in $A(M)$, the set of ground atoms in M:

$$A(M) = \big\{P(\vec{u}) \;\big|\; P \text{ an } n\text{-ary predicate}, \vec{u} \in D^n, D \text{ the domain of } M\big\},$$

and $C(F_M)$ is a propositional formula in $A(M) \cup A(M)'$.

Let M be a first-order structure. Recall that $T(M)$ is the set of ground atoms true in M:

$$T(M) = \big\{P(\vec{u}) \;\big|\; P \text{ a predicate}, \vec{u} \in P^M\big\}.$$

Lemma 3. *Let M be a finite model of Σ_{una}, and F a first order general logic program. We have that M is a model of F iff $T(M)$ is a model of F_M.*

Author's personal copy

Proof. We prove it by induction on the structure of F.

- If F is $\bot$, then this assertion holds obviously.
- If F is $a = b$, where a, b are constants, then F_M is $\top$ when a and b are the same; F_M is $\bot$ when a and b are two distinct constants. On the other hand, since M is a model of Σ_{una}, M is a model of F iff a is the same with b. Therefore this assertion holds.
- If F is an atomic formula $p(\vec{t})$, where $\vec{t}$ is a vector of constants, then F_M is $p(\sigma(\vec{t}))$. M is a model of F iff $\sigma(\vec{t}) \in p^M$ iff $p(\sigma(\vec{t})) \in T(M)$ iff $T(M)$ is a model of F_M.
- If F is $G \wedge H$, then F_M is $G_M \wedge H_M$. M is a model of F iff M is a model of $G \wedge H$ iff M is a model of G and F is a model of H iff $T(M)$ is a model of G_M and $T(M)$ is a model of H_M iff $T(M)$ is a model of $G_M \wedge H_M$ iff $T(M)$ is a model of F_M.
- If F is $G \vee H$, then F_M is $G_M \vee H_M$. M is a model of F iff M is a model of $G \vee H$ iff M is a model of G or F is a model of H iff $T(M)$ is a model of G_M or $T(M)$ is a model of H_M iff $T(M)$ is a model of $G_M \vee H_M$ iff $T(M)$ is a model of F_M.
- If F is $G \rightarrow H$, then F_M is $G_M \rightarrow H_M$. M is a model of F iff M is a model of $G \rightarrow H$ iff M is not a model of G or F is a model of H iff $T(M)$ is not a model of G_M or $T(M)$ is a model of H_M iff $T(M)$ is a model of $G_M \rightarrow H_M$ iff $T(M)$ is a model of F_M.
- If F is $\exists x G$, then F_M is $\bigvee_{u \in D}(G(x/u))_M$. M is a model of F iff there exists $u \in D$ such that M is a model of $G(x/u)$ iff there exists $u \in D$ such that $T(M)$ is a model of $(G(x/u))_M$ iff $T(M)$ is a model of $\bigvee_{u \in D}(G(x/u))_M$ iff $T(M)$ is a model of F_M.
- If F is $\forall x G$, then F_M is $\bigwedge_{u \in D}(G(x/u))_M$. M is a model of F iff for all $u \in D$, M is a model of $G(x/u)$ iff for all $u \in D$, $T(M)$ is a model of $(G(x/u))_M$ iff $T(M)$ is a model of $\bigwedge_{u \in D}(G(x/u))_M$ iff $T(M)$ is a model of F. □

Lemma 4. *Let M be a finite model of Σ_{una}, and F a first order general logic program. We have that $C(F_M)$ is equivalent to $C(F)_M$.*

Proof. We prove it by induction on the structure of F.

- If F is $\bot$, this assertion holds obviously.
- F is $a = b$, where a, b are constants. If a is the same with b, then both $C(F_M)$ and $(C(F))_M$ are $\top$, otherwise both of them are $\bot$.
- F is an atomic formula $p(\vec{t})$. Then F_M is $p(\sigma(\vec{t}))$, $C(F_M)$ is also $p(\sigma(\vec{t}))$. On the other hand, $C(F)$ is $p(\vec{t})$, $(C(F))_M$ is $p(\sigma(\vec{t}))$ too.
- F is $G \wedge H$. Then $C(F_M)$ is equivalent to $C(G_M) \wedge C(H_M)$, which is equivalent to $(C(G))_M \wedge (C(H))_M$, which is equivalent to $(C(G) \wedge C(H))_M$, which is equivalent to $(C(G \wedge H))_M$, which is $(C(F))_M$.
- F is $G \vee H$. Then $C(F_M)$ is equivalent to $C(G_M) \vee C(H_M)$, which is equivalent to $(C(G))_M \vee (C(H))_M$, which is equivalent to $(C(G) \vee C(H))_M$, which is equivalent to $(C(G \vee H))_M$, which is $(C(F))_M$.
- F is $G \rightarrow H$. Then $C(F_M)$ is $C(G_M \rightarrow H_M)$, which is equivalent to $(C(G_M) \rightarrow C(H_M)) \wedge (G'_M \rightarrow H'_M)$, which is equivalent to $(C(G)_M \rightarrow C(H)_M) \wedge (G'_M \rightarrow H'_M)$. On the other hand, $C(F)_M$ is equivalent to $((C(G) \rightarrow C(H)) \wedge (G' \rightarrow H'))_M$, which is equivalent to $(C(G)_M \rightarrow C(H)_M) \wedge (G'_M \rightarrow H'_M)$. Moreover, it's easy to see that for any first order sentence H and any structure M, H'_M is the same as $(H_M)'$. Hence, this assertion holds.
- If F is $\exists x G$, then F_M is $\bigvee_{u \in D}(G(x/u))_M$. $C(F_M)$ is $\bigvee_{u \in D} C((G(x/u))_M)$, which is equivalent to $\bigvee_{u \in D}(C(G(x/u))_M)$. On the other hand, $C(F)$ is $\exists x C(G)$, $(C(F))_M$ is $\bigvee_{u \in D}(C(G)(x/u))_M$. Moreover, by induction on the structure, it's easy to see that for any first order formula H, $C(H(x/u))$ is equivalent to $C(H)(x/u)$. Hence, this assertion holds.
- If F is $\forall x G$, then F_M is $\bigwedge_{u \in D}(G(x/u))_M$. $C(F_M)$ is $\bigwedge_{u \in D} C((G(x/u))_M)$, which is equivalent to $\bigwedge_{u \in D}(C(G(x/u))_M)$. On the other hand, $C(F)$ is $\forall x C(G)$, $(C(F))_M$ is $\bigwedge_{u \in D}(C(G)(x/u))_M$. Similarly, this assertion holds. □

Lemma 5. *Let M be a finite model of Σ_{una}, M' a conservative extension of M under (5), and F a first order sentence. Then $T(M) \cup T(M)'$ is a model of $C(F_M)$ iff M' is a model of $C(F)$.*

Proof. Since M' is a conservative extension of M under (5), $T(M') = T(M) \cup T(M)'$.

M' is a model of $C(F)$
iff
$T(M')$ is a model of $C(F)_{M'}$ (Lemma 3)
iff
$T(M) \cup T(M)'$ is a model of $C(F)_M$
iff
$T(M) \cup T(M)'$ is a model of $C(F_M)$ (Lemma 4). □

Let M and M^* be two first order structures on L' and Δ the set of predicates in L. We say that M^* is less than M on Δ, written by $M^* \subset_\Delta M$, if and only if:

Author's personal copy

(1) M and M^* have the same domain D.
(2) M and M^* map every constant c in C into the same element in D.
(3) For each $p' \in \Delta'$, $\vec{u} \in p'^{M^*}$ iff $\vec{u} \in p'^{M}$.
(4) For all $p \in \Delta$, if $\vec{u} \in p^{M^*}$, then $\vec{u} \in p^{M}$.
(5) There is a $p \in \Delta$ and some $\vec{u}$ such that $\vec{u} \in p^{M}$ and $\vec{u} \notin p_i^{M^*}$.

Thus M is a model of $Circum(F; \Delta)$ if and only if M is a model of F and there is no $M^* \subset_\Delta M$ such that M^* is also a model of F.

Lemma 6. *Let M be a finite model of Σ_{una}, M' a conservative extension of M under* (5)*, F a first order sentence, and $S \subseteq A(M)$. We have that*

(1) *$S \subset T(M)$ iff there is M^* such that $M^* \subset_\Delta M'$ and $T(M^*) = S \cup T(M)'$.*
(2) *For any M^* such that $M^* \subset_\Delta M'$ and $T(M^*) = S \cup T(M)'$, $S \cup T(M)'$ is a model of $C(F_M)$ iff M^* is a model of $C(F)$.*

Proof. (1) is obvious. Proof of (2):

M^* is a model of $C(F)$
iff
$T(M^*)$ is a model of $C(F)_{M^*}$ (Lemma 3)
iff
$S \cup T(M)'$ is a model of $C(F)_{M^*}$
iff
$S \cup T(M)'$ is a model of $C(F)_M$ (M and M^* have the same domain and interpret all constants the same)
iff
$S \cup T(M)'$ is a model of $C(F_M)$ (Lemma 4). □

Theorem 4. *Let M be a finite model of Σ_{una}. M is a stable model of F iff M' is a model of*

$$Circum\big(C(F); \Delta\big) \wedge (5), \tag{6}$$

where M' is the conservative extension of M under (5)*.*

Proof. M is a stable model of F
iff
$T(M)$ is a stable model of F_M (by the definition)
iff
$T(M) \cup T(M)'$ is a model of $\bigwedge_{p \in A(M)}(p \leftrightarrow p') \wedge Circum(C(F_M); A(M))$ (by Theorem 3)
iff

- $T(M) \cup T(M)'$ is a model of $\bigwedge_{p \in A(M)}(p \leftrightarrow p')$;
- $T(M) \cup T(M)'$ is a model of $C(F_M)$;
- for any $S \subseteq A(M)$, if $S \subset T(M)$, then $S \cup T(M)'$ is not a model of $C(F_M)$

iff

- M' is a model of (5);
- M' is a model of $C(F)$ (by Lemma 5);
- for any first order structure M^*, if $M^* \subset_\Delta M'$, then M^* is not a model of $C(F)$ (by Lemma 6)

iff M' is a model of (6). □

A.4. Proof of Theorems 5 and 6

Theorem 5. *Let F be a general logic program; X and Y two sets of atoms such that $X \subseteq Y$ and M a Kripke interpretation such that $K(M) = Th(X)$, $A(M) = Th(Y)$. We have that $\langle X, Y\rangle \models F$ iff $M \models F_{GK}$.*

Proof. As stated in [2], $\langle X, Y\rangle \models F$ iff $X \models F^Y$. According to Lemma 1, $X \models F^Y$ iff $M \models F_{GK}$, therefore, this assertion holds. □

Theorem 6. *Let F and G be two general logic programs. The following statements are equivalent.*

1. *F and G are strongly equivalent.*

Author's personal copy

2. $\bigwedge_{p\in Atom}(Kp \to Ap) \models (F \leftrightarrow G)_{GK}$.
3. $\bigwedge_{p\in Atom}(Kp \to Ap) \models F_{GK} \leftrightarrow G_{GK}$.
4. $\bigwedge_{p\in Atom}(p \to p') \models C(F \leftrightarrow G)$.
5. $\bigwedge_{p\in Atom}(p \to p') \models C(F) \leftrightarrow C(G)$.

Proof. $1 \Rightarrow 2$: Let M be a model of $\bigwedge_{p\in Atom}(Kp \to Ap)$. Construct a Kripke interpretation M_1 such that $K(M_1) = Th(X)$ and $A(M_1) = Th(Y)$, where $X = \{p \mid M \models Kp, p \in Atom\}$; $Y = \{p \mid M \models Ap, p \in Atom\}$. It's clear that $M_1 \models (F \leftrightarrow G)_{GK}$ iff $M \models (F \leftrightarrow G)_{GK}$. Since M is a model of $\bigwedge_{p\in Atom}(Kp \to Ap)$, $X \subseteq Y$. Therefore $\langle X, Y\rangle$ is an HT-interpretation. Since F and G are assumed to be strongly equivalent, they are equivalent in the logic of here-and-there. Thus $\langle X, Y\rangle \models F \leftrightarrow G$. Then by Theorem 5, $M_1 \models (F \leftrightarrow G)_{GK}$. Therefore, $M \models (F \leftrightarrow G)_{GK}$. This shows that $\bigwedge_{p\in Atom}(Kp \to Ap) \models (F \leftrightarrow G)_{GK}$.

$2 \Rightarrow 3$: $(F \leftrightarrow G)_{GK}$ is $((F \to G) \wedge (G \to F))_{GK}$, which is equivalent to $(F \to G)_{GK} \wedge (G \to F)_{GK}$, which is equivalent to

$$(F_{GK} \to G_{GK}) \wedge (F_A \to G_A) \wedge (G_{GK} \to F_{GK}) \wedge (G_A \to F_A),$$

which is equivalent to $(F_{GK} \leftrightarrow G_{GK}) \wedge (F_A \leftrightarrow G_A)$. This shows that $(F \leftrightarrow G)_{GK} \models F_{GK} \leftrightarrow G_{GK}$. Hence, this assertion holds.

$3 \Rightarrow 1$: We first show that if $\bigwedge_{p\in Atom}(Kp \to Ap) \models F_{GK} \leftrightarrow G_{GK}$, then F is classically equivalent to G. Suppose otherwise, without loss of generality, X is a model of F but not a model of G. By the definition of reduction, it's clear that X is a model of F^X. Construct a Kripke interpretation M such that $K(M) = A(M) = Th(X)$. By Lemma 1, M is a model of F_{GK}. Thus, M is also a model of G_{GK}. Again by Lemma 1, X is a model of G^X. Thus, X is a model of G, a contradiction. Hence, if $\bigwedge_{p\in Atom}(Kp \to Ap) \models F_{GK} \leftrightarrow G_{GK}$, then F is classically equivalent to G. Therefore, F_A is equivalent to G_A.

Suppose that H is a general logic program and it contains an occurrence of F, H_1 is the general logic program obtained from H by replacing an occurrence of F in H by G. We now show that H has the same set of stable models as H_1. By 3, $\bigwedge_{p\in Atom}(Kp \to Ap) \models F_{GK} \leftrightarrow G_{GK}$. Therefore, by induction on the structure of H, it's easy to see that $\bigwedge_{p\in Atom}(Kp \to Ap) \models H_{GK} \leftrightarrow (H_1)_{GK}$. Thus, H_{GK} and $(H_1)_{GK}$ has the same set of GK models. By Theorem 2, H has the same set of stable models as H_1.

$2 \Rightarrow 4$: Suppose $\bigwedge_{p\in Atom}(Kp \to Ap) \models (F \leftrightarrow G)_{GK}$. Now, suppose that X and Y are two sets of atoms and $X \cup Y'$ is a model of $\bigwedge_{p\in Atom}(p \to p')$. Therefore, $X \subseteq Y$. Construct a Kripke interpretation M such that $K(M) = Th(X)$ and $A(M) = Th(Y)$. We have that M is a model of $\bigwedge_{p\in Atom}(Kp \to Ap)$. By 2, it's also a model of $(F \leftrightarrow G)_{GK}$. By Lemma 2, $X \cup Y'$ is a model of $C(F \leftrightarrow G)$. This shows that $\bigwedge_{p\in Atom}(p \to p') \models C(F \leftrightarrow G)$.

$4 \Rightarrow 5$: Similar to the proof of $2 \Rightarrow 3$, $C(F \leftrightarrow G) \models C(F) \leftrightarrow C(G)$. Hence, this assertion holds.

$5 \Rightarrow 1$: Similar to the proof of $3 \Rightarrow 1$. □

Let F be a propositional formula constructed with atoms and basic connectives ($\bot$, $\wedge$, $\vee$ and $\to$). Let $Length(F)$ be the total number of atoms and $\bot$ occurring in F.

Lemma 7.

$$Length\big(C(F)\big) \leqslant \big(Length(F) + 1\big)\big(Length(F) + 2\big)/2 - 2.$$

Proof. We prove this assertion by induction on the structure of F.

- If F is $\bot$, then $Length(F) = Length(C(F)) = 1$. This assertion holds.
- If F is an atom p, then $Length(F) = Length(C(F)) = 1$. This assertion holds as well.
- If F is $G \wedge H$ or $G \vee H$, then $Length(C(F)) = Length(C(G)) + Length(C(H)) \leqslant (Length(G) + 1)(Length(G) + 2)/2 - 2 + (Length(H) + 1)(Length(H) + 2)/2 - 2 \leqslant (Length(G) + Length(H) + 1)(Length(G) + Length(H) + 2)/2 - 2$ (which is equivalent to $0 \leqslant Length(G)Length(H) + 1) \leqslant (Length(F) + 1)(Length(F) + 2)/2 - 2$.
- Finally, if F is $G \to H$, then $Length(C(F)) = Length(C(G)) + Length(C(H)) + Length(F) \leqslant (Length(G) + 1)(Length(G) + 2)/2 - 2 + (Length(H) + 1)(Length(H) + 2)/2 - 2 + Length(G) + Length(H) \leqslant (Length(G) + Length(H) + 1)(Length(G) + Length(H) + 2)/2 - 2$ (which is equivalent to $0 \leqslant (Length(G) - 1)(Length(H) - 1)) \leqslant (Length(F) + 1)(Length(F) + 2)/2 - 2$.

This completes the proof. □

Corollary 7. *The problem of deciding whether two general logic programs are strongly equivalent is co-NP complete.*

Proof. This assertion follows straightforwardly from Theorem 6 and Lemma 7. □

References

[1] T. Eiter, M. Fink, H. Tompits, P. Traxler, S. Woltran, Replacements in non-ground answer-set programming, in: KR'2006, 2006, pp. 340–351.
[2] P. Ferraris, Answer sets for propositional theories, in: LPNMR, 2005, pp. 119–131.

Author's personal copy

[3] P. Ferraris, V. Lifschitz, Mathematical foundations of answer set programming, in: We Will Show Them! (1), 2005, pp. 615–664.
[4] P. Ferraris, J. Lee, V. Lifschitz, A new perspective on stable models, in: Proceedings of IJCAI'07, 2007, pp. 372–379.
[5] M. Gelfond, V. Lifschitz, Classical negation in logic programs and disjunctive databases, New Generation Computing 9 (1991) 365–385.
[6] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The DLV system for knowledge representation and reasoning, ACM Transactions on Computational Logic 7 (3) (2006) 499–562.
[7] V. Lifschitz, L.R. Tang, H. Turner, Nested expressions in logic programs, Annals of Mathematics and Artificial Intelligence 25 (3–4) (1999) 369–389.
[8] V. Lifschitz, D. Pearce, A. Valverde, Strongly equivalent logic programs, ACM Transactions on Computational Logic 2 (4) (2001) 526–541.
[9] V. Lifschitz, Action languages, answer sets and planning, in: K.R. Apt, V.W. Marek, M. Truszczynski, D.S. Warren (Eds.), The Logic Programming Paradigm: A 25-Year Perspective, Springer-Verlag, 1999.
[10] F. Lin, Y. Chen, Discovering classes of strongly equivalent logic programs, in: Proc. of IJCAI'95, 2005, pp. 516–521.
[11] F. Lin, Y. Shoham, A logic of knowledge and justified assumptions, Artificial Intelligence 57 (1992) 271–289.
[12] F. Lin, Reducing strong equivalence of logic programs to entailment in classical propositional logic, in: Proc. of KR'02, 2002, pp. 170–176.
[13] V.W. Marek, M. Truszczynski, Stable logic programming – an alternative logic programming paradigm, in: K.R. Apt, V.W. Marek, M. Truszczynski, D.S. Warren (Eds.), The Logic Programming Paradigm: A 25-Year Perspective, Springer-Verlag, 1999.
[14] J. McCarthy, Applications of circumscription to formalizing commonsense knowledge, Artificial Intelligence 28 (1986) 89–118.
[15] R. Moore, Semantical considerations on nonmonotonic logic, Artificial Intelligence 25 (1) (1985) 75–94.
[16] I. Niemelä, P. Simons, Extending the smodels system with cardinality and weight constraints, in: Jack Minker (Ed.), Logic-Based Artificial Intelligence, Kluwer Academic Publishers, 2000, pp. 491–521.
[17] I. Niemelä, Logic programs with stable model semantics as a constraint programming paradigm, Annals of Mathematics and Artificial Intelligence 25 (3–4) (1999) 241–273.
[18] D. Pearce, H. Tompits, S. Woltran, Encodings for equilibrium logic and logic programs with nested expressions, in: Proc. EPIA-01, 2001, pp. 306–320.
[19] D. Pearce, A new logical characterisation of stable models and answer sets, in: Non-Monotonic Extensions of Logic Programming, 1997, pp. 57–70.
[20] R. Reiter, A logic for default reasoning, Artificial Intelligence 13 (1980) 81–132.

Ordered Completion for First-Order Logic Programs on Finite Structures

Vernon Asuncion
School of Computing & Mathematics
University of Western Sydney

Fangzhen Lin
Department of Computer Science
Hong Kong University of Sci. & Tech.

Yan Zhang and **Yi Zhou**
School of Computing & Mathematics
University of Western Sydney

Abstract

In this paper, we propose a translation from normal first-order logic programs under the answer set semantics to first-order theories on finite structures. Specifically, we introduce ordered completions which are modifications of Clark's completions with some extra predicates added to keep track of the derivation order, and show that on finite structures, classical models of the ordered-completion of a normal logic program correspond exactly to the answer sets (stable models) of the logic program.

Introduction

This work is about translating logic programs under the answer set semantics (Gelfond & Lifschitz 1988) to first-order logic. Viewed in the context of formalizing the semantics of logic programs in classical logic, work in this direction goes back to that of Clark (1978) who gave us what is now called Clark's completion semantics, on which our work, like almost all other work in this direction, is based.

In terms of the answer set semantics, Clark's completion semantics is too weak in the sense that not all models of Clark's completion are answer sets, unless the programs are "tight" (Fages 1994). Various ways to remedy this have been proposed, particularly in the propositional case (logic programs without variables) given the recent interest in Answer Set Programming (ASP) and the prospect of using SAT solvers to compute answer sets. This paper considers first-order logic programs, and the prospect of capturing the answer sets of these programs in first-order logic.

A crucial consideration in work of this kind is whether extra symbols (in the propositional case) or predicates (in the first-order case) can be used. For propositional logic programs, Ben-Eliyahu and Dechter's translation (1994) is polynomial in space but uses $O(n^2)$ extra variables, while Lin and Zhao's translation (2004) using loop formulas is exponential in the worst case but does not use any extra variables. Chen *et al.* (2006) extended loops and loop formulas to first-order case and showed that for finite domains, the answer sets of a first-order normal logic program can be captured by its completion and all its first-order loop formulas. However, in general, a program may have an infinite number of loops and loop formulas. But this seems to be the best that one can hope for if no extra predicates are used: it is well-known that transitive closure, which can be easily written as a first-order logic program, cannot be defined by any finite first-order theories on finite structures (Kolaitis 1990).

However, the situation is different if we introduce extra predicates. Our main technical result of this paper is that by using some additional predicates that keep track of the derivation order from bodies to heads in a program, we can modify Clark's completion into what we call *ordered completion* that captures exactly the answer set semantics on finite structures. While our emphasis in this paper is first-order logic programs, we nonetheless report some preliminary experimental results of using our ordered completion to compute answer sets of a ground logic program.

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Preliminaries

We assume a finite first-order language without function symbols but with equality. Given such a language, the notions of *terms*, *atoms*, *formulas* and *sentences* are defined as usual. In particular, an atom is called an *equality atom* if it is of the form $t_1 = t_2$, and a *proper atom* otherwise.

A normal logic program is a finite set of *rules* of the following form

$$\alpha \leftarrow \beta_1, \ldots, \beta_k, \mathsf{not}\, \gamma_1, \ldots, \mathsf{not}\, \gamma_l, \qquad (1)$$

where α is a proper atom, and $\beta_i, (1 \leq i \leq k), \gamma_j, (1 \leq j \leq l)$ are atoms. We call a variable in a rule a *local* variable if it occurs in the body but not the head of the rule.

Given a program Π, a predicate is called *intentional* if it occurs in the head of a rule in Π, and *extensional* otherwise. The *signature* of Π contains all intentional predicates, extensional predicates and constants occurring in Π.

For convenience and without loss of generality, in the following we assume that programs are *normalized* in the sense that for each intentional predicate P, there is a tuple $\vec{x}$ of distinct variables matching the arity of P such that for each rule, if its head mentions P, then the head must be $P(\vec{x})$. So the rules of P in the program can be enumerated as:

$$P(\vec{x}) \leftarrow Body_1, \cdots, P(\vec{x}) \leftarrow Body_k.$$

Clark's completion

Our following definition of Clark's completion is standard except that we do not make completions for extensional predicates.

Given a program Π, and a predicate P in it, *Clark's Completion* of P in Π is the following first-order sentence (Clark 1978):

$$\forall \overrightarrow{x}(P(\overrightarrow{x}) \leftrightarrow \bigvee_{1 \leq i \leq k} \exists \overrightarrow{y_i} \widehat{Body_i}), \tag{2}$$

where

- $P(\overrightarrow{x}) \leftarrow Body_1, \ldots, P(\overrightarrow{x}) \leftarrow Body_k$ are all the rules whose heads mention the predicate P (recall that we assume a program is normalized);
- $\overrightarrow{y_i}$ is the tuple of local variables in $P(\overrightarrow{x}) \leftarrow Body_i$;
- $\widehat{Body_i}$ is the conjunction of elements in $Body_i$ by simultaneously replacing the occurrences of not by $\neg$.

Clark's Completion (*completion* for short if clear from the context) of Π, denoted by $Comp(\Pi)$, is then the set of Clark's completions of all intentional predicates in Π.

Example 1 [Transitive Closure (TC)] The following normal logic program TC computes the paths of a given graph:

$$S(x,y) \leftarrow E(x,y)$$
$$S(x,y) \leftarrow E(x,z), S(z,y),$$

where E is the only extensional predicate of TC, representing the edges of a graph, and S is the only intentional predicate of TC. Ideally, the intentional predicate computes the transitive closure (i.e., all the paths) of a given graph. The Clark's Completion of TC is the following first-order sentence:

$$\forall xy(S(x,y) \leftrightarrow (E(x,y) \vee \exists z E(x,z) \wedge S(z,y))).$$

The answer set (stable model) semantics

The stable model semantics for normal propositional programs was proposed by Gelfond and Lifschitz (1988), and later extended to become answer set semantics for propositional programs that can have classical negation, constraints, disjunctions, and other operators. Due to space limitation, we assume familiarity with the answer set semantics for propositional logic programs.

The *answer set semantics* (or *stable model semantics*) for first-order normal logic programs *without extensional predicates* have been well studied as well. There are some different characterizations, for instance, in terms of grounding (Gelfond & Lifschitz 1988), in terms of loop formulas (Chen *et al.* 2006), in terms of circumscription (Lin & Zhou 2007), in terms of modified circumscription (Ferraris, Lee, & Lifschitz 2007), and in terms of first-order equilibrium logic (Pearce & Valverde 2004). It has been shown that all the above definitions coincide on finite structures (Ferraris, Lee, & Lifschitz 2007; Lin & Zhou 2007; Lee & Meng 2008).

Accounting for extensional predicates is straightforward (see e.g. (Chen *et al.* 2006)). Assuming one knows the answer set semantics of ground logic programs, the easiest way to define answer set semantics for a first-order logic program is by grounding, which is what we will do here. But one subtlety is the unique names assumption: whether distinct constants are interpreted differently. Here we do not need it, so we will not assume it.

Given a program Π, and a first-order structure $\mathcal{M}$ of the signature used in Π, we use the interpretations of $\mathcal{M}$ on the constants and extensional predicates to ground Π.

Definition 1 *The grounding of a program* Π *on a structure* $\mathcal{M}$*, written* $\Pi_{\mathcal{M}}$ *below, is the union of the following three sets:*

1. *The set of all instances of the rules in* Π *under* $\mathcal{M}$*, here an instance of a rule under* $\mathcal{M}$ *is the result of replacing all constants in the rule by their interpretations in* $\mathcal{M}$*, and all variables in the rule by some domain objects in* $\mathcal{M}$*;*
2. $EQ_{\mathcal{M}} = \{u = u \mid u$ *is a domain object in* $\mathcal{M}\}$*;*
3. $Ext_{\mathcal{M}} = \{Q(\vec{u}) \mid Q$ *is an extensional predicate and* $\vec{u} \in Q^{\mathcal{M}}\}$*, here* $Q^{\mathcal{M}}$ *is the interpretation of* Q *in* $\mathcal{M}$*.*

We now have the following definition:

Definition 2 *Let* Π *be a normal logic program and* $\mathcal{M}$ *a structure. We say that* $\mathcal{M}$ *is a* stable model *or an* answer set *of* Π *if the following set*

$$EQ_{\mathcal{M}} \cup Ext_{\mathcal{M}} \cup Int_{\mathcal{M}}$$

is an answer set of $\Pi_{\mathcal{M}}$ *in the propositional case, where* $Int_{\mathcal{M}}$ *is the following set*

$$\{P(\vec{u}) \mid P \text{ is an intentional predicate, and } \vec{u} \in P^{\mathcal{M}}\}.$$

Ordered Completion

It is well-known that Clark's completion does not fully capture the answer set semantics because of the cycles. For instance, the following program

$$p \leftarrow q$$
$$q \leftarrow p$$

has one answer set $\{\}$, but its completion completion $p \leftrightarrow q$ has two models $\{p, q\}$ and $\{\}$. Here, we propose a modification of Clark's completion to address this issue. The main technical property of our new translation is that for each finite first-order logic program, our translation yields a finite first-order theory that captures exactly the finite stable models of the program. The ideas behind our translation can be best illustrated by simple propositional programs. Consider the program mentioned above. We introduce four extra symbols $T_{pq}, T_{pp}, T_{qq}, T_{qp}$ (read, e.g. T_{pq} as from p to q), and translate this program into the following theory

$$(p \rightarrow q) \wedge (q \rightarrow p),$$
$$q \rightarrow (p \wedge T_{pq} \wedge \neg T_{qp}),$$
$$p \rightarrow (q \wedge T_{qp} \wedge \neg T_{pq}),$$
$$T_{pq} \wedge T_{qp} \rightarrow T_{pp},$$
$$T_{qp} \wedge T_{pq} \rightarrow T_{qq}.$$

The first sentence is the direct encoding of the two rules. The second one is similar to Clark's completion for q except that we add T_{pq} and $\neg T_{qp}$: for q to be true, p must be true and that it must be that p is used to derive q but not the other way around. The third sentence is similar, and the last two sentences are about the transitivity of the T atoms. It can be seen that in all models of the above sentences, both p and q must be false.

Definition of ordered completion

In general, let Π be a first-order normal logic program, and Ω_Π its set of intentional predicates. For each pair of predicates (P, Q) (might be the same) in Ω_Π, we introduce a new predicate T_{PQ}, called the *comparison predicate*, whose arity is the sum of the arities of P and Q. The intuitive meaning of $T_{PQ}(\overrightarrow{x}, \overrightarrow{y})$, read as from $P(\overrightarrow{x})$ to $Q(\overrightarrow{y})$, is that there is a derivation path from $P(\overrightarrow{x})$ to $Q(\overrightarrow{y})$.

Definition 3 *Let Π be a normal logic program. The* ordered completion *of Π, denoted by $OC(\Pi)$, is the set of following sentences:*

- *For each intentional predicate P, the following sentences:*

$$\forall \overrightarrow{x}(\bigvee_{1\le i\le k} \exists \overrightarrow{y_i} \widehat{Body_i} \rightarrow P(\overrightarrow{x})), \quad (3)$$

$$\forall \overrightarrow{x}(P(\overrightarrow{x}) \rightarrow \bigvee_{1\le i\le k} \exists \overrightarrow{y_i}(\widehat{Body_i} \wedge \bigwedge_{Q(\overrightarrow{z})\in Pos_i, Q\in\Omega_\Pi} (T_{QP}(\overrightarrow{z}, \overrightarrow{x}) \wedge \neg T_{PQ}(\overrightarrow{x}, \overrightarrow{z})))) \quad (4)$$

  where we have borrowed the notations used in the definition of Clark's completion, and further assume that Pos_i is the positive part of $Body_i$ and $Q(\overrightarrow{z})$ ranges over all the intentional atoms in the positive part of $Body_i$;

- *For each triple of intentional predicates P, Q, and R (two or all of them can be the same predicate) the following sentence:*

$$\bigwedge_{P,Q,R\in\Omega_\Pi} \forall \overrightarrow{x}\,\overrightarrow{y}\,\overrightarrow{z}(T_{PQ}(\overrightarrow{x}, \overrightarrow{y}) \wedge T_{QR}(\overrightarrow{y}, \overrightarrow{z}) \rightarrow T_{PR}(\overrightarrow{x}, \overrightarrow{z})), \quad (5)$$

In the following, we use $MComp(\Pi)$ to denote the set of the formulas (3) and (4), and $TranS(\Pi)$ the set of formulas (5). So $OC(\Pi) = MComp(\Pi) \cup Trans(\Pi)$.

Clearly, for finite programs, $OC(\Pi)$ is finite, and the predicates occurring in $OC(\Pi)$ are all the predicates occurring in Π together with all the comparison predicates $\{T_{PQ} \,|\, P, Q \in \Omega_\Pi\}$.

Notice that Clark's completion of a predicate consists of two parts:

$$\forall \overrightarrow{x}(\bigvee_{1\le i\le k} \exists \overrightarrow{y_i} \widehat{Body_i} \rightarrow P(\overrightarrow{x}))$$

$$\forall \overrightarrow{x}(P(\overrightarrow{x}) \rightarrow \bigvee_{1\le i\le k} \exists \overrightarrow{y_i} \widehat{Body_i}).$$

Thus the difference between $MComp(\Pi)$ and $Comp(\Pi)$ is that the former introduces some assertions on the comparison predicates, which intuitively mean that there exist derivation paths from the intentional atoms in the body to head but not the other way around (see Equation (4)). In addition, $TranS(\Pi)$ simply means that the comparison predicates satisfy "transitivity".

Proposition 1 *Let Π be a normal logic program. Then, $OC(\Pi)$ introduces m^2 new predicates whose arities are no more than $2s$, and the size of $OC(\Pi)$ is $O(s \times m^3 + s \times n)$, where m is the number of intentional predicates of Π, s the maximal arity of the intentional predicates of Π and n the length of Π.*

Example 2 [Transitive Closure continued] Recall the Transitive Closure program TC presented in Example 1. In this case, since the only intentional predicate is S, we only need to introduce one additional predicate T_{SS}, whose arity is 4. The ordered completion of TC consists of the following sentences:

$$\forall xy\ (E(x,y) \vee \exists z(E(x,z) \wedge S(z,y))) \rightarrow S(x,y),$$
$$\forall xy\ S(x,y) \rightarrow (E(x,y) \vee \exists z(E(x,z) \wedge S(z,y) \wedge T_{SS}(z,y,x,y) \wedge \neg T_{SS}(x,y,z,y))),$$
$$\forall xyuvzw\ T_{SS}(x,y,u,v) \wedge T_{SS}(u,v,z,w) \rightarrow T_{SS}(x,y,z,w).$$

Intuitively, one can understand $T_{SS}(x, y, u, v)$ to mean that $S(x, y)$ is used to establish $S(u, v)$. So the second sentence means that for $S(x, y)$ to be true, either $E(x, y)$ (the base case), or inductively, for some z, $E(x, z)$, $S(z, y)$, and $S(z, y)$ is used to establish $S(x, y)$ and not the other way around.

To see how these axioms work, consider the graph in Figure 1 with four vertices a, b, c, d, with E representing the edge relation: $E(a, b), E(b, a), E(a, c), E(c, d)$.

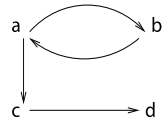


Figure 1: An example graph

Clearly, if there is a path from x to y, then $S(x, y)$ (by the first sentence above). We want to show that if there is no path from x to y, then $\neg S(x, y)$. Consider $S(d, a)$. If it is true, then since $\neg E(d, a)$, there must be an x such that

$$E(d, x) \wedge S(x, a) \wedge T_{SS}(x, a, d, a) \wedge \neg T_{SS}(d, a, x, a).$$

This is false as there is no edge going out of d.

Now consider $S(c, a)$. If it is true, then there must be an x such that

$$E(c, x) \wedge S(x, a) \wedge T_{SS}(x, a, c, a) \wedge \neg T_{SS}(c, a, x, a).$$

So x must be d, and

$$S(d, a) \wedge T_{SS}(d, b, a, b) \wedge \neg T_{SS}(a, b, d, b).$$

However, as shown above, $S(d, a)$ is false.

The main theorem

In this section, we prove the following main theorem.

Theorem 1 *Let Π be a normal logic program whose signature is σ, and $\mathcal{A}$ a finite σ-structure. Then, $\mathcal{A}$ is an answer*

set of Π *if and only if there exists a model* $\mathcal{M}$ *of* $OC(\Pi)$ *such that* $\mathcal{A}$ *is the reduct*[1] *of* $\mathcal{M}$ *on* σ.

Proof:(sketch) First we show that every finite answer set $\mathcal{A}$ of Π can be expanded to a model of $OC(\Pi)$. Construct a finite structure $\mathcal{M}$ by expanding $\mathcal{A}$ with the following interpretations on T_{PQ} for each pair (P, Q) of intentional predicates in Π:

$T_{PQ}(\overrightarrow{a}, \overrightarrow{b})$ iff there exists a path from $Q(\overrightarrow{b})$ to $P(\overrightarrow{a})$ in the dependency graph (see the definition in (Lin & Zhao 2003)) of the ground program $\Pi_{\mathcal{A}}$.

where $\overrightarrow{a}$ and $\overrightarrow{b}$ are two tuples of elements in the domain of $\mathcal{A}$ that match the arities of P and Q respectively. It can be proved that $\mathcal{M}$ is a model of $OC(\Pi)$.

Now we prove that the reduct of any finite model $\mathcal{M}$ of the ordered completion of Π on σ must be an answer set of Π. Clearly, $\mathcal{M} \uparrow \sigma$ is a model of $Comp(\Pi)$. Hence, according to the loop formula characterization of answer set semantics in the propositional case (Lin & Zhao 2003), it suffices to show that for all loops L of the ground program $\Pi_{\mathcal{M}\uparrow\sigma}$, the set of ground atoms $EQ_{\mathcal{M}\uparrow\sigma} \cup Ext_{\mathcal{M}\uparrow\sigma} \cup Int_{\mathcal{M}\uparrow\sigma}$ is a model of its loop formula.

Otherwise, since $\mathcal{M}$ is a model of $MComp(\Pi)$, we can get a sequence of ground atoms $P_0(\overrightarrow{a_0}), P_1(\overrightarrow{a_1}), P_2(\overrightarrow{a_2}), \ldots$, such that for all i, $P_i(\overrightarrow{a_i}) \in L$, $\overrightarrow{a_i} \in P_i^{\mathcal{M}}$, $T_{P_{i+1}P_i}(\overrightarrow{a_{i+1}}, \overrightarrow{a_i})$ holds in $\mathcal{M}$, and $T_{P_iP_{i+1}}(\overrightarrow{a_i}, \overrightarrow{a_{i+1}})$ does not hold in $\mathcal{M}$. Hence, for all $k < l$, $T_{P_lP_k}(\overrightarrow{a_l}, \overrightarrow{a_k})$ holds in $\mathcal{M}$ but $T_{P_kP_l}(\overrightarrow{a_k}, \overrightarrow{a_l})$ does not hold since T_{PQ} satisfy transitivity for all pairs of intentional predicates. However, since $\mathcal{M}$ is finite, there exist $k < l$ such that $P_k(\overrightarrow{a_k}) = P_l(\overrightarrow{a_l})$, a contradiction. ■

Normal logic program with constraints

Recall that we have required the head of a rule to be a proper atom. If we allow the head to be empty, then we have so-called *constraints*:

$$\leftarrow \beta_1, \ldots, \beta_k, \mathsf{not}\, \gamma_1, \ldots, \mathsf{not}\, \gamma_l, \tag{6}$$

where $\beta_i, (1 \leq i \leq k), \gamma_j, (1 \leq j \leq l)$ are atoms. A model is said to satisfy the above constraint if it satisfies the corresponding sentence:

$$\forall \overrightarrow{y} \neg(\beta_1 \wedge \ldots \wedge \beta_k \wedge \neg\gamma_1 \wedge \ldots \wedge \neg\gamma_l),$$

where $\overrightarrow{y}$ is the tuple of all variables occurring in (6). In the following, if c is a constraint of form (6), then we use $\widehat{c}$ to denote its corresponding formula above.

A normal logic program with constraints is then a finite set of rules and constraints. The answer set semantics can be extended to normal logic programs with constraints: a model is an answer set if it is an answer set of the set of the rules in the program and satisfies all the constraints in the program.

[1]A σ-structure is said to be a *reduct* of a σ'-structure ($\sigma \subseteq \sigma'$) $\mathcal{M}$ on σ, denoted by $\mathcal{M} \uparrow \sigma$, if it is the structure obtained from $\mathcal{M}$ by removing the interpretations of the symbols in $\sigma' \backslash \sigma$ (Ebbinghaus & Flum 1995).

Both Clark's completion and our ordered completion can be extended straightforwardly to normal logic programs with constraints: one simply adds the sentences corresponding to the constraints to the respective completions.

Proposition 2 *Let* Π *be a normal logic program whose signature is* σ*,* C *a set of constraints, and* $\mathcal{A}$ *a finite* σ*-structure. Then,* $\mathcal{A}$ *is an answer set of* $\Pi \cup C$ *iff there exists a model* $\mathcal{M}$ *of* $OC(\Pi) \cup \{\widehat{c} \mid c \in C\}$*, such that* $\mathcal{A}$ *is the reduct of* $\mathcal{M}$ *on* σ.

Ordered completion on maximal predicate loops (strongly connected components)

In our definition of ordered completions, we introduce a comparison predicate between each pair of predicates. This is not necessary. We only need to do so for pairs of predicates that belong to a same loop in the predicate dependency graph of the program.

Formally, the *predicate dependency graph* of a first-order program Π is a finite graph $PG_\Pi = \langle V, E\rangle$, where V is the set of all intentional predicates of Π and $(P, Q) \in E$ iff there is a rule whose head mentions P and whose positive body mentions Q.

Maximal predicate loops are then strongly connected components of PG_Π. Ordered completions on maximal predicate loops are the same as ordered completions except that the comparison predicates T_{PQ} are defined only when P and Q belong to a same maximal predicate loop. More precisely, the ordered completion of Π on maximal predicate loops, denoted by $OC^*(\Pi)$, is of the similar form as the ordered completion of Π (see Definition 3), except that

- $Q(\overrightarrow{z})$ in (4) ranges over all the intentional atoms in the positive part of $Body_i$ such that for some maximal predicate loop L, both P and Q are in L.
- P, Q and R in (5) are intentional predicates such that for some maximal predicate loop L, P, Q, and R are all in L.

The following proposition is a refinement of the main theorem.

Proposition 3 *Let* Π *be a normal logic program whose signature is* σ*, and* $\mathcal{A}$ *a finite* σ*-structure. Then,* $\mathcal{A}$ *is an answer set of* Π *if and only if there exists a model* $\mathcal{M}$ *of* $OC^*(\Pi)$ *such that* $\mathcal{A}$ *is the reduct of* $\mathcal{M}$ *on* σ.

In many cases, restricting comparison predicates on maximal predicate loops results in a much smaller ordered completion.

Example 3 [Hamiltonian Circuit (HC)] Consider the following normal program HC with constraints for computing Hamiltonian circuits of a graph:

$hc(x, y) \leftarrow arc(x, y), \mathsf{not}\, otherroute(x, y)$,
$otherroute(x, y) \leftarrow arc(x, y), arc(x, z), hc(x, z), y \neq z$,
$otherroute(x, y) \leftarrow arc(x, y), arc(z, y), hc(z, y), x \neq z$,
$reached(y) \leftarrow arc(x, y), hc(x, y), reached(x), \mathsf{not}\, init(x)$,
$reached(y) \leftarrow arc(x, y), hc(x, y), init(x)$,
$\leftarrow vertex(x), \mathsf{not}\, reached(x)$.

This program has three intentional predicates: hc, $otherroute$ and $reached$. According to the original version of ordered completion (see Definition 3), we need to

introduce 9 comparison predicates, and the maximal arity is 4.

However, by using maximal predicate loops, only one extra predicate is needed since HC has only one maximal predicate loop, namely $\{reached\}$. The only comparison predicate needed is $T_{RR}(x, y)$, which is binary. Hence, the new form of the ordered completion of HC is the following set of sentences:

$$\forall xy(hc(x,y) \leftrightarrow arc(x,y) \wedge \neg otherroute(x,y)),$$
$$\forall xy(otherroute(x,y) \leftrightarrow$$
$$\exists z(arc(x,y) \wedge arc(x,z) \wedge hc(x,z) \wedge y \neq z) \vee$$
$$\exists z(arc(x,y) \wedge arc(z,y) \wedge hc(z,y) \wedge x \neq z)),$$
$$\forall y((\exists x(arc(x,y) \wedge hc(x,y) \wedge reached(x) \wedge \neg init(x)) \vee$$
$$\exists x(arc(x,y) \wedge hc(x,y) \wedge init(x))) \rightarrow reached(y)),$$
$$\forall y(reached(y) \rightarrow (\exists x(arc(x,y) \wedge hc(x,y) \wedge init(x)) \vee$$
$$\exists x(arc(x,y) \wedge hc(x,y) \wedge reached(x) \wedge \neg init(x) \wedge$$
$$T_{RR}(x,y) \wedge \neg T_{RR}(y,x)))),$$
$$\forall x \neg(vertex(x) \wedge \neg reached(x)),$$
$$\forall xyz(T_{RR}(x,y) \wedge T_{RR}(y,z) \rightarrow T_{RR}(x,z)).$$

Arbitrary structures

Finally, we want to emphasize that the correspondence between classical first-order models of our ordered completions and stable models of a logic program holds only on finite structures. In general, the result doesn't hold on arbitrary structures. For instance, on arbitrary structures, the transitive closure program cannot be captured by a first-order sentence with or without using new predicates.

Related Work and Discussions

The only other translations from first-order logic programs under answer set semantics to first-order logic are based on loop formulas (Chen *et al.* 2006; Lee & Meng 2008). As mentioned earlier, the main difference between these translations and ours is that ours results in a finite first-order theory but uses extra predicates while the ones based on loop formulas do not use any extra predicates but in general result in an infinite first-order theory.

As we also mentioned, the basic intuitions behind almost all of the current translations from logic programs with answer set semantics to classical logic are similar. The main differences are in the ways these intuitions are formalized. In the following, we briefly review some of the closely related ones in the propositional case.

Propositional case

Ordered completion can be viewed as a propositional translation from normal logic programs to propositional theories by treating each propositional atom as a 0-ary predicate. Several proposals in this direction have been proposed (Ben-Eliyahu & Dechter 1994; Lin & Zhao 2003; Janhunen 2004; Lin & Zhao 2004; Niemelä 2008). Ben-Eliyahu and Dechter (1994) assigned an index (or level numbering) $\#x$ to each propositional atom x, and added the assertions $\#x < \#y$ to the Clark's completion for each pair (x, y), where x is the head of a rule and y is in the positive body of the rule. Janhunen (2004) proposed another similar translation and implemented an ASP solver called "lp2atomic|lp2sat". More recently, Niemelä (2008) proposed to capture the level mapping in difference logic, and based on it, designed an ASP solver called lp2diff using an SMT solver that integrates SAT with a difference logic module.

The main difference between Ben-Eliyahu and Dechter's translation and ours is that we use the comparison atoms T_{xy} instead of indices $\#x$. In fact, $T_{xy} \wedge \neg T_{yx}$ in ordered completion plays the role as $\#x < \#y$ in Ben-Eliyahu and Dechter 's translation. Although they look similar in the Clark's completion part, the ways to encode indices and comparison atoms in classical propositional logic are very different. As the difference to Niemelä's work, which formalized the notion of groundedness using the build-in predicate "$<$" in difference logic, we did it by introducing some new (comparison) predicates in classical logic. This difference shows up in the implementations as well: while Niemelä uses SMT, we use SAT.

Another translation, also sharing the basic idea of comparing stages, is due to Lin and Zhao (2003). However, they first translate a program equivalently to a tight program, and then use the Clark's completion of the new program to capture the original one. Also, the loop formula approach in the propositional case (Lin & Zhao 2004) can be regarded as a translation from logic programs to propositional theories. Different from the ones mentioned above, the loop formula approach requires no extra atoms but may be exponential. Based on this idea, two solvers called "assat" (Lin & Zhao 2004) and "cmodels" (Lierler & Maratea 2004) are implemented by adding loop formulas incrementally.

Fixed-point logic and Datalog

Another related work (Kolaitis 1990) is in the area of finite model theory and fixed-point logic. Although fixed-point logic and normal logic programming are not comparable, they have a common fragment, namely Datalog. Kolaitis (1990) showed that every fixed-point query is conjunctive definable on finite structures. That is, given any fixed-point query Q, there exists another fixed-point query Q' such that the conjunctive query (Q, Q') is implicit definable on finite structures. As a consequence, every datalog query is also conjunctive definable on finite structures. From this result, although tedious, one can actually derive a translation from datalog to first-order sentences using some new predicates not in the signatures of the original datalog programs.

We will not go into details comparing our translation and the one derived from Kolaitis' result since our focus here is on normal logic programs. Suffice to say here that the two are different in many ways, not the least is that ours is based on Clark's completion in the sense that some additional conditions are added to the necessary parts of intentional predicates, while the one derived from Kolaitis' result is not. We mention this work because Kolaitis's result did play an important role early on in our work. We speculated that if it is possible to translate datalog programs to first-order sentences using some new predicates, then it must also be possible for normal logic programs, and that if this is true, then

it must be doable by modifying Clark's completion. As it happened, this turned out to be the case.

Some Experimental Results

While our interest is mainly on first-order normal logic programs, and the possibility of constructing a first-order ASP solver, as an easy exercise, we implemented a prototype propositional ASP solver, called asp2sat, using our ordered completion. Our preliminary experimental results seem to indicate that while not as good as clasp[2], it is quite competitive with other SAT or SMT-based ones.

Table 1 contains some runtime data on Niemelä's Hamiltonian Circuit program with the particular instances taken from the assat website[3]. The following solvers were compared: our asp2sat using either zchaff (3.12)[4] or minisat (2.0)[5]; lp2atomic (1.12)[6] with minisat and lp2diff (1.10)[7] with z3; cmodels (3.79)[8] with zchaff, and clasp. In the table, "y" ("n") means that the corresponding graph has a (no) Hamiltonian Circuit. For each instance, we record the average time for 5 runs in seconds. We set the timeout threshold as 900 seconds, which is denoted by "-" in the table.

graph		asp2sat +zchaff	asp2sat +minisat	lp2atomic +minisat	lp2diff +z3	cmodels	clasp
nv60a356	y	1.04	0.35	0.27	0.13	3.01	0.02
nv60a526	y	3.21	3.11	1.06	0.21	1.82	0.09
nv60a554	y	7.33	16.06	1.95	0.26	0.39	0.04
nv70a396	y	0.43	0.95	0.55	0.16	7.26	0.04
nv70a428	y	1.52	2.08	0.67	0.20	0.68	0.03
nv70a511	y	1.20	15.57	1.52	0.28	2.27	0.04
nv70a549	y	3.05	17.12	1.91	0.24	0.33	0.04
nv70a571	y	4.35	5.35	5.52	0.24	0.33	0.04
2xp30.1	y	4.04	4.97	2.60	28.83	8.44	0.01
2xp30.3	y	2.56	8.40	580.93	2.29	17.31	0.04
2xp30.4	n	17.56	18.66	-	198.58	217.23	8.33
4xp20	n	0.00	0.09	118.27	2.82	0.14	0.00
4xp20.1	n	0.52	0.19	113.12	2.55	0.65	0.08
4xp20.2	y	1.40	1.26	126.51	0.28	0.90	0.02
4xp20.3	n	6.10	0.62	122.50	3.36	1.38	0.01

Table 1: Experimental results

Conclusion

The main contribution of this paper is to introduce a notion of ordered completion that captures the answer set semantics of normal logic programs on finite structures (See Theorem 1). Interestingly, this result fails if infinite structures are allowed. Theorem 1 is also surprising in the sense that many logic programs cannot be captured by first-order sentences without extra predicates (e.g. TC in Example 1).

Theorem 1 is important from both a theoretical and a practical point of view. To the best of our knowledge, our translation from first-order normal logic programs to first-order sentences is the first such one. It is also worth noting that this translation, when instantiated in the propositional case in an obvious way, yields an ASP solver that is competitive with other ASP solvers based on SAT and SMT solvers. More interestingly, we are looking at the possibility of using Theorem 1 to construct a first-order ASP solver. For us, this is the most important future direction of this work.

[2]http://www.cs.uni-potsdam.de/clasp/

[3]http://assat.cs.ust.hk/Assat-2.0/hc-2.0.html

[4]http://www.princeton.edu/∼chaff/zchaff.html

[5]http://www.minisat.se/

[6]http://www.tcs.hut.fi/Software/lp2sat/

[7]http://www.tcs.hut.fi/Software/lp2diff/

[8]http://www.cs.utexas.edu/∼tag/cmodels/

References

Ben-Eliyahu, R., and Dechter, R. 1994. Propositional semantics for disjunctive logic programs. *Ann. Math. Artif. Intell.* 12(1-2):53–87.

Chen, Y.; Lin, F.; Wang, Y.; and Zhang, M. 2006. First-order loop formulas for normal logic programs. In *KR'06*, 298–307.

Clark, K. L. 1978. Negation as failure. In Gallaire, H., and Minker, J., eds., *Logics and Databases*. New York: Plenum Press. 293–322.

Ebbinghaus, H. D., and Flum, J. 1995. *Finite Model Theory*. Springer-Verlag.

Fages, F. 1994. Consistency of Clark's completion and existence of stable of stable models. *Journal of Methods of Logic in Computer Science* 1:51–60.

Ferraris, P.; Lee, J.; and Lifschitz, V. 2007. A new perspective on stable models. In *IJCAI'07*, 372–379.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *ICLP'88*, 1070–1080.

Janhunen, T. 2004. Representing normal programs with clauses. In *ECAI'04*, 358–362.

Kolaitis, P. G. 1990. Implicit definability on finite structures and unambiguous computations (preliminary report). In *LICS'90*, 168–180.

Lee, J., and Meng, Y. 2008. On loop formulas with variables. In *KR'08*, 444–453.

Lierler, Y., and Maratea, M. 2004. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In *LPNMR'04*, 346–350.

Lin, F., and Zhao, J. 2003. On tight logic programs and yet another translation from normal logic programs to propositional logic. In *IJCAI'03*, 853–858.

Lin, F., and Zhao, Y. 2004. ASSAT: computing answer sets of a logic program by SAT solvers. *Artif. Intell.* 157(1-2):115–137.

Lin, F., and Zhou, Y. 2007. From answer set logic programming to circumscription via logic of GK. In *IJCAI'07*, 441–446.

Niemelä, I. 2008. Stable models and difference logic. *Ann. Math. Artif. Intell.* 53(1-4):313–329.

Pearce, D., and Valverde, A. 2004. Towards a first order equilibrium logic for nonmonotonic reasoning. In *JELIA'04*, 147–160.

Computing Loops with at Most One External Support Rule

Xiaoping Chen and **Jianmin Ji**
University of Science and Technology of China
P. R. China
xpchen@ustc.edu.cn, jizheng@mail.ustc.edu.cn

Fangzhen Lin
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
flin@cs.ust.hk

Abstract

If a loop has no external support rules, then its loop formula is equivalent to a set of unit clauses; and if it has exactly one external support rule, then its loop formula is equivalent to a set of binary clauses. In this paper, we consider how to compute these loops and their loop formulas in a normal logic program, and use them to derive consequences of a logic program. We show that an iterative procedure based on unit propagation, the program completion and the loop formulas of loops with no external support rules can compute the same consequences as the "Expand" operator in smodels, which is known to compute the well-founded model when the given normal logic program has no constraints. We also show that using the loop formulas of loops with at most one external support rule, the same procedure can compute more consequences, and these extra consequences can help ASP solvers such as cmodels to find answer sets of certain logic programs.

Introduction

The notions of loops and loop formulas were first proposed by Lin and Zhao (2004) for propositional normal logic programs. They have been extended to disjunctive logic programs (Lee & Lifschitz 2003), general logic programs (Ferraris, Lee, & Lifschitz 2006), normal logic programs with variables (Chen *et al.* 2006), and propositional circumscription (Lee & Lin 2006). They can be used for computing answer sets of a logic program by using an off-the-shelf SAT solver (Lin & Zhao 2004) or by modifying an existing SAT procedure (Giunchiglia, Lierler, & Maratea 2006). They are also useful for ASP (Answer Set Programming) solvers that use SAT-like strategies such as conflict-clause learning (Gebser *et al.* 2007).

In this paper, we propose to use loops and loop formulas to compute *consequences* of a logic program, which are formulas that are satisfied by every answer set of the logic program. In particular, we focus on consequences that are literals and can be computed efficiently. Our main motivation for this work is to use these consequences to speed-up ASP solvers.

Lin and Zhao (2004) showed that a set of atoms is an answer set of a normal logic program iff it satisfies the completion and the loop formulas of the program. Thus the consequences of a logic program are the logical consequences of its completion and loop formulas. However, deduction in propositional logic is coNP-hard and in general there may be an exponential number of loops (Lifschitz & Razborov 2006). One way to overcome these problems is to use some tractable inference rules and consider only those loop formulas that can be used effectively by these inference rules and at the same time can be computed efficiently.

In this paper, we choose unit propagation as the inference rule. To find loop formulas that are "unit propagation" friendly, let us look at the form of loop formulas.

According to (Lin & Zhao 2004), a loop L is a set of atoms, and its loop formula is a sentence of the form:

$$\bigvee L \supset \bigvee_{r \in R^-(L)} body(r),$$

where $R^-(L)$ is the set of so-called *external support rules* of L, and $body(r)$ is the conjunction of the literals in the body of the rule r. Without going into details about the definition of external support rules and how they are computed, we see that if a loop L has no external support rules, then its loop formula is equivalent to the following set of literals:

$$\{\neg a \mid a \in L\},$$

and if a loop L has exactly one external support rule, say r, then its loop formula is equivalent to the following set of binary clauses:

$$\{\neg a \vee l \mid a \in L,\ l \in body(r)\}.$$

Thus we see that loops that have at most one external support rule are special in that their loop formulas will yield unit or binary clauses that can be used effectively by unit propagation.

More generally, if we assume a set A of literals, then for any loop that has at most one external support rule whose body is not false under A, its loop formula is equivalent to either a set of literals or a set of binary clauses under A.

Since the completion of a logic program can be computed and converted to a set of clauses in linear time (by introducing new variables if necessary), if these loop formulas can also be computed in polynomial time, we then have a polynomial time algorithm for computing some consequences of a logic program. One such algorithm is as follows:

Copyright © 2008, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

1. compute the program completion and convert it to clauses;
2. compute the loop formulas of the loops with at most one external support rule;
3. apply unit propagation to the set of clauses obtained so far;
4. compute the loop formulas of the loops with at most one external support under the set of literals computed in the previous step, add them to the set of clauses, and go back to the previous step until it does not produce any new consequences.

We shall show that for normal logic programs, the loop formulas of loops with at most one external support rule can indeed be computed in polynomial time. Furthermore, if we consider only loops that do not have any external support rules in the above procedure, then it computes essentially the same set of literals as does the $Expand$ operator in smodels (Simons, Niemelä, & Soininen 2002). In particular, it computes the well-founded model when the given normal logic program has no constraints. In general, this procedure can be more powerful than the $Expand$ operator in smodels, when loops with external support rules are considered. As an example, consider the Hamiltonian Circuit (HC) problem. If the set of vertices of the given graph has two parts, A and B, such that there is exactly one arc from a node in A to a node in B, and one arc from a node in B to a node in A, then any HC of the graph must go through these two arcs. This means that every answer set of the logic program for this HC problem must contain these two arcs, and knowing this in advance should help in computing an answer set of the program. But none of these two "must in" arcs cannot be computed using the $Expand$ operator, but one of them can be computed using our procedure.

Notice that the above procedure works in principle for more general logic programs, such as disjunctive logic programs, for which notions like the program completion, loops, and loop formulas can be defined. Thus it may prove to be a useful way for extending the well-founded semantics to these more expressive logic programs, just as it does for normal logic programs.

This paper is organized as follows. We briefly review the basic notions of logic programming in the next section. We then define loops with at most one external support rule under a given set of literals, and consider how to compute their loop formulas. We then consider how to use these loop formulas to derive consequences of a logic program using unit propagation, and relate it to the $Expand$ operator in smodels. Finally, we discuss some other possible uses of these loop formulas.

Preliminaries

In this paper, we consider only fully grounded finite normal logic programs that may have constraints. That is, a logic program here is a finite set of rules of the form:

$$H \leftarrow p_1, \ldots, p_k, not\ q_1, \ldots, not\ q_m, \qquad (1)$$

where p_i, $1 \leq i \leq k$, and q_j, $1 \leq j \leq m$, are atoms, and H is either empty or an atom. If H is empty, then this rule is also called a *constraint*, and if H is an atom, it is a *proper rule*.

Given a logic program P, we denote by $Atoms(P)$ the set of atoms in it, and $Lit(P)$ the set of literals constructed from $Atoms(P)$:

$$Lit(P) = Atoms(P) \cup \{\neg a \mid a \in Atoms(P)\}.$$

Given a literal l, the *complement* of l, written $\bar{l}$ below, is $\neg a$ if l is a and a if l is $\neg a$, where a is an atom. For a set L of literals, we let $\overline{L} = \{\bar{l} \mid l \in L\}$.

For a rule r of the form (1), we let $head(r)$ be its head H, $body(r)$ the set $\{p_1, \ldots, p_k, \neg q_1, \ldots, \neg q_m\}$ of literals obtained from the body of the rule with "not" replaced by "$\neg$", $body^+(r)$ the set of atoms in its body, $\{p_1, \ldots, p_k\}$, and $body^-(r)$ the set of atoms under not in its body, $\{q_1, \ldots, q_m\}$.

To define the *answer sets* of a logic program with constraints, we first define the *stable models* of a logic program that does not have any constraints (Gelfond & Lifschitz 1988). Given a logic program P without constraints, and a set S of atoms, the Gelfond-Lifschitz transformation of P on S, written P_S, is obtained from P by deleting:

1. each rule that has a negative literal $not\ q$ in its body with $q \in S$, and
2. all negative literals in the bodies of the remaining rules.

Clearly for any S, P_S is a set of rules without any negative literals, so that P_S has a unique minimal model, denoted by $\Gamma_P(S)$. Now a set S of atoms is a stable model of P iff $S = \Gamma_P(S)$.

In general, given a logic program P that may have constraints, a set S of atoms is an answer set of P iff it is a stable model of the program obtained by deleting all the constraints in P, and it satisfies all the constraints in P, i.e. for any constraint of the form (1) such that H is empty, either $p_i \notin S$ for some $1 \leq i \leq k$ or $q_j \in S$ for some $1 \leq j \leq m$.

Notice that an answer set is a set of atoms, and can be considered as a logical model: the model assigns true to an atom iff it is in the answer set. In the following, when we say that an answer set satisfies a logical formula, we refer to the answer set as a logical model. Thus a sentence is a consequence of a logic program if it is true in every answer set of the logic program.

The well-founded semantics (Van Gelder, Ross, & Schlipf 1991) for programs without constraints can be defined using the fixed points of the operator Γ_P^2 (Van Gelder 1989), defined as follows: $\Gamma_P^2(S) = \Gamma_P(\Gamma_P(S))$. The operator Γ_P is anti-monotonic, so Γ_P^2 is monotonic. Thus it has a least and a greatest fixed point which we denote by $\text{lfp}(\Gamma_P^2)$ and $\text{gfp}(\Gamma_P^2)$, respectively. Informally, the atoms in the least fixed point are those that must be true, while those not in the greatest fixed point must be false. Formally, if P is a logic program without constraints, then its *well-founded model* is the set of literals

$$\text{lfp}(\Gamma_P^2) \cup \overline{Atoms(P) \setminus \text{gfp}(\Gamma_P^2)}.$$

As we mentioned, if a literal is in the well-founded model, then it is satisfied by every answer set. Of course, a logic program may not have any answer sets.

It is not very clear how the well-founded semantics can be extended to logic programs with constraints. One possibility is to take it to be the output of the $Expand$ operator of smodels as the operator outputs the well-founded model if the given program has no constraints, see (Baral 2003).

Another way of providing a semantics to logic programs is to transform them to theories in classical logic. This starts with Clark's (1978) completion semantics which says that an atom is true if and only if there is a rule with this atom as its head such that the body of this rule is true. Formally, given a program P without constraints, the completion of an atom p is the disjunction of the bodies of all rules in P that have p as their head. If there is no such rule, then the completion of p is $\neg p$. The Clark's completion of P is then the set of completions of all atoms in P. If P has constraints, then the completion of P is the union of Clark's completion and the set of sentences corresponding to the constraints in P: if $\leftarrow p_1, ..., p_n, not\ q_1, ..., not\ q_m$ is a constraint, then its corresponding sentence is $\neg(p_1 \wedge \cdots \wedge p_n \wedge \neg q_1 \wedge \cdots \wedge \neg q_m)$.

As we mentioned, we will convert the completion into a set of clauses. Since we are going to use inference rules that may not be logically complete, it matters how we convert it. In the following, let $comp(P)$ be the set of following clauses:

1. for each $a \in Atoms(P)$, if there is no rule in P with a as its head, then add $\neg a$;
2. if r is a proper rule, then add $head(r) \vee \bigvee \overline{body(r)}$;
3. if r is a constraint, then add the clause $\bigvee \overline{body(r)}$;
4. if a is an atom and $r_1, ..., r_n$, $n > 0$, are all the rules in P with a as their head, then introduce n new variables $v_1, ..., v_n$, and add the following clauses:

$$\neg a \vee v_1 \vee \cdots \vee v_n,$$
$$v_i \vee \bigvee \overline{body(r_i)}, \text{ for each } 1 \leq i \leq n,$$
$$\neg v_i \vee l, \text{ for each } l \in body(r_i) \text{ and } 1 \leq i \leq n.$$

Clearly, the size of $comp(P)$ is linear in the size of P.

It is known that if A is an answer set of P, then A is also a propositional model of the completion of P, but the converse may not be true in general as the completion semantics does not handle cycles. For instance, if the only rule about p in a program is $p \leftarrow p$, then the completion of p is $p \equiv p$, which is a tautology. But according to the answer set semantics p is false as one should not prove p by assuming p. To address this problem with the completion semantics, Lin and Zhao (2004) proposed adding so-called loop formulas to the completion of a program and show that the resulting propositional theory coincides with the answer set semantics.

To define loop formulas, we have to define loops, which are defined in terms of positive dependency graphs. Given a logic program P, the *positive dependency graph* of P, written G_P, is the directed graph whose vertices are atoms in P, and there is an arc from p to q if there is a rule $r \in P$ such that $p = head(r)$ and $q \in body^+(r)$. A set L of atoms is said to be a loop of P if for any p and q in L, there is a non-empty path from p to q in G_P such that all the vertices in the path are in L, i.e. the L-induced subgraph of G_P is strongly connected.

Given a loop, Lin and Zhao defined a formula which says that an atom in the loop cannot be proved by the atoms in the loop only. Thus atoms in the loop can be proved only by using some atoms and rules that are "outside" of the loop. Formally, a rule r is called an *external support* of a loop L if $head(r) \in L$ and $L \cap body^+(r) = \emptyset$. In the following, let $R^-(L)$ be the set of external support rules of L. Then the *loop formula* of L under P, written $LF(L, P)$, is the following implication:

$$\bigvee_{p \in L} p \supset \bigvee_{r \in R^-(L)} body(r), \tag{2}$$

where we have abused the notation and use $body(r)$ also for the conjunction of the literals in it.

Lin and Zhao showed that a set of atoms is an answer set of a logic program iff it is a propositional model of the completion and the loop formulas of all loops of the program. Thus the logical consequences of the program completion and loop formulas are identical to the consequences of the logic program. In particular, all literals in the well-founded model of a program are logical consequences of the program completion and loop formulas. However, checking logical entailment is coNP-complete, and in general, a program may have an exponential number of loops (Lifschitz & Razborov 2006). So to use program completion and loop formulas to derive consequences of a logic program efficiently, we have to use an efficient rule of inference such as unit propagation.

Given a set Γ of clauses, we let $UP(\Gamma)$ be the set of literals that can be derived from Γ by unit propagation. Formally, it can be defined as follows:

Function $UP(\Gamma)$

if $(\emptyset \in \Gamma)$ **then return** Lit;
$A := unit_clause(\Gamma)$;
if A is inconsistent **then return** Lit;
if $A \neq \emptyset$ **then return** $A \cup UP(assign(A, \Gamma))$
else return $\emptyset$;

where $unit_clause(\Gamma)$ returns the union of all unit clauses in Γ, and $assign(A, \Gamma)$ is

$$\{c \mid \text{ for some } c' \in \Gamma, c' \cap A = \emptyset, \text{ and } c = c' \setminus \overline{A}\}.$$

Loops with at Most One External Support

Consider first loops without any external support rules. If a loop L has no external support rules, i.e. $R^-(L) = \emptyset$, then its loop formula (2) is equivalent to $\bigwedge_{p \in L} \neg p$. More generally, if we already know that X is a set of literals that are true in all answer sets, then if X entails $\neg body(r)$, written $X \models \neg body(r)$, for every rule $r \in R^-(L)$, then under X, the loop formula of L is equivalent to $\bigwedge_{p \in L} \neg p$, where $X \models \neg body(r)$ means that the complement of one of the literals in $body(r)$ is in X.

Thus we extend the notion of external support rules, and have it conditioned on a given set of literals.

Let P be a logic program, and X a set of literals. We say that a rule $r \in R^-(L)$ is an *external support of L under X* if $X \not\models \neg body(r)$. In the following, we denote by $R^-(L, X)$

the set of external support rules of L under X. Now given a logic program P and a set X of literals, let

$$loop_0(P,X) = \{\neg a \mid a \in L \text{ for a loop } L \text{ of } P \text{ such that } R^-(L,X) = \emptyset\}.$$

Then $loop_0(P,X)$ is equivalent to the set of loop formulas of the loops that do not have any external support rules under X. In particular, the set of loop formulas of the loops that do not have any external support rules is equivalent to $loop_0(P,\emptyset)$.

Similarly, we can consider the set of loop formulas of the loops that have exactly one external support rule under a set X of literals:

$$loop_1(P,X) = \{\neg a \vee l \mid a \in L,\ l \in body(r),\ \text{for some loop } L \text{ and rule } r \text{ such that } R^-(L,X) = \{r\}\}$$

In particular, $loop_1(P,\emptyset)$ is equivalent to the set of loop formulas of the loops that have exactly one external support rule in P.

We now consider how to compute $loop_0(P,X)$ and $loop_1(P,X)$. We start with $loop_0(P,X)$. Let $ML_0(P,X)$ be the set of maximal loops that do not have any external support rules under X: a loop is in $ML_0(P,X)$ if it is a loop of P such that $R^-(L,X) = \emptyset$, and there does not exist any other such loop L' such that $L \subset L'$. Clearly,

$$loop_0(P,X) = \bigcup_{L \in ML_0(P,X)} \overline{L}.$$

Proposition 1 *Let P be a logic program and X a set of literals. If L_1 and L_2 are two loops of P that do not have any external support rules under X, and $L_1 \cap L_2 \neq \emptyset$, then $L_1 \cup L_2$ is also a loop of P that does not have any external support rules under X.*

Proof: Suppose otherwise, and let $r \in R^-(L_1 \cup L_2, X)$. Then $head(r) \in L_1 \cup L_2$, $body^+(r) \cap (L_1 \cup L_2) = \emptyset$, and $X \not\models \neg body(r)$. Thus either $head(r) \in L_1$ or $head(r) \in L_2$. So either $r \in R^-(L_1, X)$ or $r \in R^-(L_2, X)$, a contradiction with the assumption that L_1 and L_2 do not have any external support rules under X. ■

Thus for any P, and $X \subseteq Lit(P)$, loops in $ML_0(P,X)$ are pair-wise disjoint. This means that there can only be at most $|Atoms(P)|$ loops in $ML_0(P,X)$.

To compute $ML_0(P,X)$, consider G_P, the positive dependency graph of P. If L is a loop of P, then there must be a strongly connected component C of G_P such that $L \subseteq C$. For L to be in $ML_0(P,X)$, there are two cases: either $L = C$ and $R^-(C,X) = \emptyset$ or $L \subset C$, $R^-(C,X) \neq \emptyset$ and $R^-(L,X) = \emptyset$. If it is the latter, then for any $r \in R^-(C,X)$, it must be that $head(r) \notin L$, for otherwise, r must be in $R^-(L,X)$, a contradiction with $R^-(L,X) = \emptyset$. Thus if $R^-(C,X) \neq \emptyset$, then any subset of C that is in $ML_0(P,X)$ must also be a subset of $S = C \setminus \{head(r) \mid r \in R^-(C,X)\}$. Thus instead of G_P, we can recursively search the S induced subgraph of G_P. This motivates the following procedure for computing $ML_0(P,X)$.

$ML_0(P,X) := ML_0(P,X,Atoms(P))$;

Function $ML_0(P,X,S)$

$ML := \emptyset$; $G :=$ the S induced subgraph of G_P;

For each strongly connected component L of G:

if $R^-(L,X) = \emptyset$ **then** add L to ML **else** append $ML_0(P,X,L \setminus \{head(r) \mid r \in R^-(L,X)\})$ to ML.

return ML.

From our discussions above, the following result is immediate.

Theorem 1 *For any normal logic program P, any $X \subseteq Lit(P)$, and any $C \subseteq Atoms(P)$, the above function $ML_0(P,X,C)$ returns the following set of loops:*

$\{L \mid L \subseteq C$ is a loop of P such that $R^-(L,X) = \emptyset$, and there does not exist any other such loop L' such that $L \subset L'\}$

in $O(n^2)$, where n is the size of P as a set.

We now consider the problem of computing $loop_1(P,X)$. For any rule r, let $ML_1(P,X,r)$ be the set of maximal loops of P that have r as their only external support rule under X: L is in $ML_1(P,X,r)$ if it is a loop of P such that $R^-(L,X) = \{r\}$ and there is no other such loop L' such that $L \subset L'$. Notice that this definition is meaningful only if r is a proper rule of P. If r is a constraint, then it can never be an external support rule of any loop, thus $ML_1(P,X,r) = \emptyset$. Now let

$$ML_1(P,X) = \bigcup_{r \in P} ML_1(P,X,r).$$

It is easy to see that loops in $ML_1(P,X,r)$ are pair-wise disjoint. Thus the size of $ML_1(P,X,r)$ is bounded by $|Atoms(P)|$, and $ML_1(P,X)$ by $m|Atoms(P)|$, where m is the number of proper rules in P.

It is easy to see that $loop_1(P,X)$ is the following set:

$$\bigcup_{L \in ML_1(P,X)} \{\neg a \vee l \mid a \in L, l \in body(r), R^-(L,X) = \{r\}\}.$$

Thus to compute $loop_1(P,X)$, we need to compute only $ML_1(P,X)$, and for the latter, we only need to compute $ML_1(P,X,r)$ for all proper rules $r \in P$ such that $X \not\models \neg body(r)$. At first glance, this problem can be trivially reduced to that of computing $ML_0(P \setminus \{r\}, X)$, the set of maximal loops that do not have any external support rules under X in the program obtained from P by deleting r from it. However, while it is true that if $L \in ML_0(P \setminus \{r\}, X)$ and r is the only external support rule of L under X in P, then $L \in ML_1(P,X,r)$, the converse is not true in general.

Example 1 Consider the following logic program P:

$$\begin{aligned} a &\leftarrow b,\ c. \\ b &\leftarrow a. \\ b &\leftarrow c. \\ c &\leftarrow b. \end{aligned}$$

It is easy to see that $ML_1(P,\emptyset,b \leftarrow c) = \{\{a,b\}\}$, but $ML_0(P \setminus \{b \leftarrow c\}, \emptyset) = \{\{a,b,c\}\}$.

We do not yet know any efficient way of computing $ML_1(P, X, r)$, but for the purpose of computing $loop_1(P, X) \cup loop_0(P, X)$, $ML_0(P \setminus \{r\}, X)$ is enough.

Proposition 2 *For any normal logic program P and a set X of literals, $loop_0(P, X)$ implies that $loop_1(P, X)$ is equivalent to the following theory*

$$\bigcup_{X \not\models \neg body(r), L \in ML_0(P \setminus \{r\}, X)} \{\neg a \vee l \mid a \in L, l \in body(r)\}. \quad (3)$$

Proof: Suppose $loop_0(P, X)$ and $loop_1(P, X)$ are true. We show that for any proper rule r of P such that $X \not\models \neg body(r)$, and any $L \in ML_0(P \setminus \{r\}, X)$,

$$\{\neg a \vee l \mid a \in L, l \in body(r)\} \quad (4)$$

is true. Firstly, L is also a loop of P, and that either $R^-(L, X) = \emptyset$ or $R^-(L, X) = \{r\}$. For the first case, $\overline{L} \subseteq loop_0(P, X)$, thus (4) is true. For the second case, (4) is contained in $loop_1(P, X)$, thus true.

Now suppose $loop_0(P, X)$ and (3) are true. We show that $loop_1(P, X)$ is true, i.e. for any proper rule $r \in P$ such that $X \not\models \neg body(r)$ and any $L \in ML_1(P, X, r)$, (4) holds. Firstly, L is a loop of $P \setminus \{r\}$ that has no external support rules under X. Thus there exists $L' \in ML_0(P \setminus \{r\}, X)$ such that $L \subseteq L'$. Now L' is also a loop of P, and w.r.t. P, either $R^-(L', X) = \emptyset$ or $R^-(L', X) = \{r\}$. In the first case, $\overline{L} \subseteq \overline{L'} \subseteq loop_0(P, X)$, thus (4) holds. In the second case, L' must be in $ML_1(P, X, r)$ as well, thus $L = L'$ and (4) holds. ■

So to summarize, to compute $loop_0(P, X) \cup loop_1(P, X)$, we first compute $ML_0(P, X)$, and then for each proper rule $r \in P$ such that $X \not\models \neg body(r)$, we compute $ML_0(P \setminus \{r\}, X)$. The worst case complexity of this procedure is $O(n^3)$, where n is the size of P. There are a lot of redundancies in this procedure as described here as the computations of $ML_0(P, X)$ and $ML_0(P \setminus \{r\}, X)$ overlap a lot. These redundancies can and should be eliminated in the actual implementation.

Computing Consequences of a Logic Program

By Lin and Zhao's theorem on loop formulas, logical consequences of $comp(P) \cup loop_0(P, X) \cup loop_1(P, X)$ are also consequences of P. Since $comp(P)$, $loop_0(P, X)$, and $loop_1(P, X)$ can all be computed in polynomial time, with a polynomial time inference rule, we thus get a polynomial time algorithm for computing some consequences of a logic program. In this paper, we consider using $UP(C)$, the unit propagation.

Consider first loops without any external support rules. With $comp(P)$, $loop_0(P, X)$, and $UP(C)$, we can do the following:

$Y := comp(P) \cup loop_0(P, \emptyset)$; $X := \emptyset$;
while $X \neq UP(Y)$ **do**
  $X := UP(Y)$; $Y := Y \cup loop_0(P, X)$;
return $X \cap Lit(P)$.

Formally, the above procedure computes the least fixed point of the following operator:

$$f(X) = UP(comp(P) \cup X \cup loop_0(P, X)) \cap Lit(P).$$

As it turns out, this least fixed point is essentially the well-founded model when the given logic normal logic program has no constraints. This means that, surprisingly perhaps, the well-founded models amount to repeatedly applying unit propagation to the program completion and loop formulas of loops that do not have any "applicable" external support rules. In general, for normal logic programs that may have constraints, the least fixed point of the above operator is essentially $Expand(P, \emptyset)$ in smodels (Simons, Niemelä, & Soininen 2002; Baral 2003). More generally, $Expand(P, A)$ corresponds to the least fixed point of the following operator:

$$U_A^P(X) = UP(comp(P) \cup A \cup X \cup loop_0(P, X)) \cap Lit(P). \quad (5)$$

Now if we add in $loop_1(P, X)$, a more powerful operator can be defined:

$$T_A^P(X) = Lit(P) \cap UP(comp(P) \cup A \cup X \cup loop_0(P, X) \cup loop_1(P, X)).$$

In the following, we denote by $T(P, A)$ the least fixed point of the operator T_A^P. Clearly, $U_A^P(X) \subseteq T_A^P(X)$, and the least fixed point of U_A^P, denoted by $U(P, A)$, is contained in $T(P, A)$, the least fixed point of T_A^P. The following example shows that the containments can be proper.

Example 2 Consider the following logic program P:

$$\begin{array}{rcl} x & \leftarrow & not\ e. \\ e & \leftarrow & not\ x. \\ n & \leftarrow & x. \\ n & \leftarrow & m. \\ m & \leftarrow & n. \\ & \leftarrow & not\ n. \end{array}$$

Clearly, $x \in T(P, \emptyset)$ but $x \notin U(P, \emptyset)$.

The following proposition says that if an answer set of P satisfies A, then it also satisfies $T(P, A)$. In particular, every literal in $T(P, \emptyset)$ is a consequence of P.

Proposition 3 *Let P be a normal logic program and A a set of literals in P. If S is an answer set of P that satisfies A, then S also satisfies $T(P, A)$.*

Notice that $T(P, A)$, the least fixed point of the operator T_A^P, can be computed by an iterative procedure like the one described in Introduction.

Expand in smodels

We mentioned that the least fixed point of our operator U_A^P defined by (5) coincides with the output of the $Exapnd$ operator used in smodels. We now make this precise. Our following presentation of the $Expand$ operator in smodels follows that in (Baral 2003).

Given a logic program P and a set A of literals, the goal of $Expand(P, A)$ is to extend A as much as possible and as efficiently as possible so that all answer sets of P that agree with A also agree with $Expand(P, A)$. It is defined in terms of two functions named $Atleast(P, A)$ and $Atmost(P, A)$. They form the lower and upper bound of what can be derived from the program P based on A in the sense that those in $Atleast(P, A)$ must be in and those not in $Atmost(P, A)$ must be out. Formally, $Expand(P, A)$ is defined to be the least fixed point of the following operator:

$$E_A^P(X) = Atleast(P, A \cup X) \cup \overline{Atoms(P) \setminus Atmost(P, A \cup X)}. \quad (6)$$

The function $Atleast(P, A)$ is defined as the least fixed point of the operator F_A^P defined as follows:

$F_1(P, X) = \{a \in Atoms(P) \mid$ there is a rule r in P such that $a = head(r)$ and $X \models body(r)\}$,

$F_2(P, X) = \{\neg a \mid a \in Atoms(P)$ and for all $r \in P$, if $a = head(r)$, then $X \models \neg body(r)\}$,

$F_3(P, X) = \{x \mid$ there exists an atom $a \in X$ such that there is only one rule r in P such that $a = head(r)$, $x \in body(r)$, and $X \not\models \neg body(r)\}$,

$F_4(P, X) = \{\bar{x} \mid$ there exists $\neg a \in X$ such that there is a rule r in P such that $a = head(r)$ and $X \cup \{x\} \models body(r)\}$,

$$F_5(P, X) = \begin{cases} Lit(P) & \text{if } X \text{ is inconsistent,} \\ \emptyset & \text{otherwise,} \end{cases}$$

$F_A^P(X) = A \cup X \cup F_1(P, X) \cup F_2(P, X) \cup F_3(P, X) \cup F_4(P, X) \cup F_5(P, X)$.

The function $Atmost(P, A)$ is defined as the least fixed point of the following operator G_A^P:

$G_A^P(X) = \{a \mid$ there is a rule $r \in P$ such that $head(r) = a$, $X \setminus A^- \models body^+(r)$, and $body^-(r) \cap A^+ = \emptyset\} \setminus A^-$,

where $A^+ = \{a \mid a$ is an atom and $a \in A\}$, and $A^- = \{a \mid a$ is an atom and $\neg a \in A\}$.

In the following, a program P is said to be *simplified* if for any $r \in P$, $head(r) \notin body^+(r) \cup body^-(r)$. Notice that any logic program is strongly equivalent to a simplified logic program: if $head(r) \in body^+(r)$, then $\{r\}$ is strongly equivalent to the empty set, thus can be safely deleted from any logic program, and if $head(r) \in body^-(r)$, then $\{r\}$ is strongly equivalent to $\{\leftarrow body(r)\}$ (cf. (Lin & Chen 2007)).

The following theorem relates $U(P, A)$ and $Expand(P, A)$. The proof is given in Appendix.

Theorem 2 *For any normal logic program P, and any $A \subseteq Lit(P)$. $Expand(P, A) \subseteq U(P, A)$. If P is a simplified, then $Expand(P, A) = U(P, A)$.*

As we mentioned, if P has no constraint, then $Expand(P, \emptyset)$ is the same as the well-founded model of P (Baral 2003). Thus for simplified logic programs, the well-founded model can be computed by a bottom-up procedure using unit propagation on sets of clauses from the program completion and the loop formulas of the loops that do not have any external support rules.

The following example shows that if P has a rule such as $p \leftarrow not\ p$, U may be stronger than $Expand$.

Example 3 Consider the following program P:

$$\begin{aligned} p &\leftarrow not\ q. \\ q &\leftarrow not\ p. \\ f &\leftarrow not\ p. \\ f &\leftarrow not\ f. \end{aligned}$$

$U(P, \emptyset) = \{f, q, \neg p\}$, but $Expand(P, \emptyset) = \emptyset$.

Some Experiments

We have implemented a program that for any given normal logic program P, it first computes $T(P, \emptyset)$, and then adds the following set of constraints:

$$\{\leftarrow \bar{l} \mid l \in T(P, \emptyset)\}$$

to P. Our implementation is available at the following URL:

http://www.cs.ust.hk/cloop/

By Proposition 3, adding the above constraints to P does not change the answer sets, and intuitively, should help ASP solvers in computing the answer sets. To verify this, we tried our program on a number of benchmarks.

First, for the normal logic programs at the First Answer Set Programming System Competition[1], $T(P, \emptyset)$ does not return anything beyond the well-founded model of P. Thus for these programs, our program does not add anything new.

Next we tried Niemelä's (1999) encoding of the HC problem, and consider graphs with the special structure as mentioned in Introduction. Specifically, we create some copies of a complete graph, and then randomly add some arcs to connect these copies into a strongly connected graph such that any HC for this graph must go through these special arcs.

Table 1 contains the running times for these programs.[2] In this table, MxN.K stands for a graph with M copies of the complete graph with N nodes: $C_1, ..., C_M$, and with exactly one arc from C_i to C_{i+1} and exactly one arc from C_{i+1} to C_i, for each $1 \leq i \leq M$ (C_{M+1} is defined to be C_1). The extension K stands for a specific way of adding these arcs. The numbers under "cmodels with T" refers to the run times (in seconds) of cmodels (version 3.74 (Giunchiglia, Lierler, & Maratea 2006)) when the results from $T(P, \emptyset)$ are added to the original program as constraints, and those under "T" are the run times of our program for computing $T(P, \emptyset)$. As can be seen, information from $T(P, \emptyset)$ makes cmodels run much faster when looking for an answer set. In addition to cmodels, we also tried smodels and DLV (Leone *et al.* 2006), which unfortunately could not terminate even for the smallest graphs considered here. We also tried clasp (Gebser *et al.* 2007), which is very fast on these programs. On

[1] http://asparagus.cs.uni-potsdam.de/contest/.

[2] Our experiments were done on a Pentium 4 3.2Ghz CPU and 1GB RAM. The reported times are in CPU seconds as reported by Linux "/usr/bin/time" command.

average it returned a solution in a few seconds. Adding literals from $T(P, \emptyset)$ makes it run a little faster, but the speedup is not worth the overhead of computing $T(P, \emptyset)$.

Problem	cmodels	cmodels with T	T
20x12.1	67.62	8.20	2.18
20x12.2	5.02	5.45	1.98
20x12.3	17.33	5.69	2.00
20x12.4	29.91	7.07	1.82
20x12.5	5.56	6.90	1.78
20x12.6	8.76	6.45	2.08
20x12.7	76.11	4.86	2.13
20x12.8	281.31	4.93	1.84
20x12.9	233.56	5.08	2.10
20x12.10	57.98	7.51	2.11
20x20.1	$>$2h	87.01	8.24
20x20.2	114.39	129.31	8.05
20x20.3	$>$2h	59.12	6.49
20x20.4	97.24	63.06	8.22
20x20.5	123.17	86.93	8.30
20x20.6	$>$2h	105.15	8.36
20x20.7	1175.61	79.48	8.23
20x20.8	171.87	76.52	8.25
20x20.9	109.07	63.41	7.95
20x20.10	$>$2h	81.42	8.26

Table 1: Run-time Data.

Conclusion

We have looked at the loops that have at most one external support rule in this paper. These loops are special in that their loop formulas are equivalent to sets of unit or binary clauses. We have considered how they, together with the program completion, can be used to deduce useful consequences of a logic program under unit propagation and in the process relate them to the well-founded semantics and the $Expand$ operator in smodels.

We think that this work opens up a line of research that deserves further study:

- We have used unit propagation as the inference rule. One could use others as long as they are "efficient" enough. For example, in addition to unit propagation, one can consider adding the following rule: infer l from $l \vee a$ and $l \vee \neg a$, or even the full resolution on binary clauses.
- We have used these loop formulas for deriving consequences of a logic program. They can of course be used in ASP solvers such as ASSAT and cmodels directly. Whether this has any benefit requires further study.
- As we mentioned, what we have done here for normal logic programs can be extended to more expressive logic programs such as disjunctive logic programs. Among other things, this may provide a new perspective on how to extend the well-founded semantics to these more expressive logic programs. For instance, there have been several competing proposals for extending the well-founded semantics to disjunctive logic programs (Lobo, Minker, & Rajasekar 1992; Przymusinski 1995; Leone, Rullo, & Scarcello 1997; Brass & Dix 1999; Wang & Zhou 2005). It would be interesting to find out which one, if any, of these corresponds to the least fixed point of our operator U_A^P when applied to disjunctive logic programs.

Acknowledgments

This work has been supported in part by the Natural Science Foundations of China under grants 60496322, 60745002, 60573009, and 60703095, and by the Hong Kong RGC CERG 616806.

References

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Brass, S., and Dix, J. 1999. Semantics of (disjunctive) logic programs based on partial evaluation. *The Journal of Logic Programming* 40(1):1–46.

Chen, Y.; Lin, F.; Wang, Y.; and Zhang, M. 2006. First-order loop formulas for normal logic programs. In *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR2006)*, 298–307.

Clark, K. L. 1978. Negation as failure. In Gallaire, H., and Minker, J., eds., *Logic and Databases*. New York: Plenum Press. 293–322.

Ferraris, P.; Lee, J.; and Lifschitz, V. 2006. A generalization of the lin-zhao theorem. *Annals of Mathematics and Artificial Intelligence* 47(1-2):79–101.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007. Conflict-driven answer set solving. In *Proceddings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, 386–392.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference on Logic Programming (ICLP-88)*, 1070–1080.

Giunchiglia, E.; Lierler, Y.; and Maratea, M. 2006. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning* 36(4):345–377.

Lee, J., and Lifschitz, V. 2003. Loop formulas for disjunctive logic programs. In *Proceedings of the Nineteenth International Conference on Logic Programming (ICLP-03)*, 451–465.

Lee, J., and Lin, F. 2006. Loop formulas for circumscription. *Artificial Intelligence* 170(2):160–185.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic* 7(3):499–562.

Leone, N.; Rullo, P.; and Scarcello, F. 1997. Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Information and Computation* 135(2):69–112.

Lifschitz, V., and Razborov, A. 2006. Why are there so many loop formulas? *ACM Transactions on Computational Logic* 7(2):261–268.

Lin, F., and Chen, Y. 2007. Discovering Classes of Strongly Equivalent Logic Programs. *Journal of Artificial Intelligence Research* 28:431–451.

Lin, F., and Zhao, Y. 2004. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence* 157(1-2):115–137.

Lobo, J.; Minker, J.; and Rajasekar, A. 1992. *Foundations of Disjunctive Logic Programming*. MIT Press.

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

Przymusinski, T. 1995. Static semantics of logic programs. *Annals of Mathematics and Artificial Intelligence* 14:323–357.

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

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

Van Gelder, A. 1989. The alternating fixpoint of logic programs with negation. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS-89)*, 1–10.

Wang, K., and Zhou, L. 2005. Comparisons and computation of well-founded semantics for disjunctive logic programs. *ACM Transactions on Computational Logic* 6(2):295–327.

Proof of Theorem 2

We prove Theorem 2 by showing that for any logic program P and any set A of literals, $Expand(P, A) \subseteq U(P, A)$ (Lemma 2 below) and, for any simplified logic program P, $U(P, A) \subseteq Expand(P, A)$ (Lemma 3 below).

Before proving these two lemmas, we first prove that $\overline{Atoms(P) \setminus Atmost(P, A)}$ is equivalent to $M(P, A)$, the least fixed point of the operator M_P^A defined as follows:

$loop_0^A(P, X) = \{\neg a \mid$ there is a loop L of P such that $a \in L$ and $R^-(L, A \cup X) = \emptyset\}$,

$F_2^A(P, X) = \{\neg a \mid a \in Atoms(P)$ and for all $r \in P$, if $a = head(r)$ then $A \cup X \models \neg body(r)\}$,

$M_P^A(X) = \{\neg a \mid a \in A^-\} \cup X \cup loop_0^A(P, X) \cup F_2^A(P, X)$.

Lemma 1 *For any normal logic program P, and any $A \subseteq Lit(P)$. $M(P, A) = \overline{Atoms(P) \setminus Atmost(P, A)}$.*

Proof: Let $AM(P, A) = \overline{Atoms(P) \setminus Atmost(P, A)}$ and $X \subseteq AM(P, A)$, clearly $\overline{X} \cap Atmost(P, A) = \emptyset$. First we prove that $M(P, A) \subseteq AM(P, A)$. As $M(P, A)$ is the least fixed point of our operator M_P^A, we only need to prove that $M_P^A(X) \subseteq AM(P, A)$.

Let L be a loop of P and $R^-(L, A \cup X) = \emptyset$, clearly $L \cap Atmost(P, A \cup X) = \emptyset$. For any set of atoms Y such that $Y \subseteq Atmost(P, A)$ and $Y \subseteq Atmost(P, A \cup X)$, as $\overline{X} \cap Atmost(P, A) = \emptyset$, $Y \setminus A^- = Y \setminus (A^- \cup \overline{X})$, so $G_A^P(Y) = G_{A \cup X}^P(Y)$, hence $Atmost(P, A \cup X) = Atmost(P, A)$. Now we have $L \cap Atmost(P, A) = \emptyset$, then $\overline{loop_0^A(P, X)} \cap Atmost(P, A) = \emptyset$ and $loop_0^A(P, X) \subseteq AM(P, A)$.

Let a be an atom such that $a \in \overline{F_2^A(P, X)}$, then for all $r \in P$, if $a = head(r)$ then $A \cup X \models \neg body(r)$. Clearly, $a \notin Atmost(P, A \cup X)$ and $a \notin Atmost(P, A)$, hence, $F_2^A(P, X) \subseteq AM(P, A)$.

So $M_P^A(X) \subseteq AM(P, A)$ and $M(P, A) \subseteq AM(P, A)$. Now we prove that $AM(P, A) \subseteq M(P, A)$. Let $S = AM(P, A) \setminus M(P, A)$, we want to prove $S = \emptyset$. The sketch of the proof is: if $S \neq \emptyset$, then there exists a loop $L \subseteq \overline{S}$ and $R^-(L, M(P, A)) = \emptyset$, clearly, $\overline{L} \subseteq loop_0^A(P, M(P, A))$ and $\overline{L} \subseteq M(P, A)$, so $S \cap M(P, A) \neq \emptyset$, which conflicts to the definition of S. Now We give the detail of the proof.

For any atom a, if $\neg a \in S$, then there exists a rule r such that $head(r) = a$ and $Atmost(P, A) \setminus A^- \not\models body^+(r)$, which is equivalent to $((Atoms(P) \setminus Atmost(P, A)) \cup A^-) \cap body^+(r) \neq \emptyset$. As $Atoms(P) \setminus Atmost(P, A) = \overline{S} \cup \overline{M(P, A)}$, there exists a rule r such that $head(r) = a$ and $\overline{S} \cap body^+(r) \neq \emptyset$.

Let $G_P^{\overline{S}}$ be the $\overline{S}$ induced subgraph of G, $L = \{a \mid$ for each $b \in \overline{S}$, if there is a path from a to b in $G_P^{\overline{S}}$, then there is a path from b to a in $G_P^{\overline{S}}\}$. Now we prove that, $L \neq \emptyset$ and $R^-(L, M(P, A)) = \emptyset$.

For any atom $a \in \overline{S}$, let $H^-(a) = \{b \mid b \in \overline{S}$, there is a path from a to b and there is not any path from b to a in $G_P^{\overline{S}}\}$ and $H^+(a) = \{b \mid b \in \overline{S}$, there is a path from b to a in $G_P^{\overline{S}}\}$. If $L = \emptyset$, then for all atoms $a \in \overline{S}$, $H^-(a) \neq \emptyset$. If an atom

$b \in H^-(a)$, then $a \in H^+(b)$ and $H^-(b) \subseteq H^-(a) \setminus \{a\}$. If another atom $c \in H^-(b)$ then $H^-(c) \subseteq H^-(a) \setminus \{a, b\}$. So we can form an infinite list of atoms $\{a_1, a_2, \ldots\}$, where $a_{i+1} \in H^-(a_i)$ and $a_{i+1} \neq a_j$ $(1 \leq j \leq i, 1 \leq i \leq \infty)$. But $\overline{S}$ is finite, so $L \neq \emptyset$.

Clearly, L is a loop of P, $L \subseteq \overline{S}$, $L \neq \emptyset$, and for each external support rule r of L, $body^+(r) \cap \overline{S} = \emptyset$. Furthermore, all rules with heads belong to $\overline{S}$ are false under $M(P, A) \cup S$, so $R^-(L, M(P, A)) = \emptyset$. Then $\overline{L} \subseteq loop_0^A(P, M(P, A))$, $\overline{L} \subseteq M(P, A)$ and $\overline{L} \subseteq S$, which conflicts to the definition of S.

So $S = \emptyset$ and $M(P, A) = \overline{Atoms(P) \setminus Atmost(P, A)}$. ■

Lemma 2 *For any normal logic program P, and any $A \subseteq Lit(P)$. $Expand(P, A) \subseteq U(P, A)$.*

Proof: $Expand(P, A)$ is the least fixed point of our operator E_A^P defined by (6). For any set of literals X, such that $X \subseteq U(P, A)$, we want to prove $E_A^P(X) \subseteq U(P, A)$. First we prove that $Atleast(P, A \cup X) \subseteq U(P, A)$.

$Atleast(P, A \cup X)$ is the least fixed point of operator $F_{A \cup X}^P$. Let $Y \subseteq U(P, A)$, we consider $F_i(P, Y)$ $(1 \leq i \leq 5)$ respectively.

Let x be a literal, if $x \in F_1(P, Y)$, then there is a rule $x \leftarrow l_1, \ldots, l_n.$ and $\{l_1, \ldots, l_n\} \subseteq Y$. The corresponding clause $x \vee \bar{l}_1 \vee \cdots \vee \bar{l}_n$ is in $comp(P)$, then $x \in UP(comp(P) \cup Y)$. As $Y \subseteq U(P, A)$ and $U(P, A)$ is the least fixed point of our operator U_A^P defined by (5), $x \in UP(comp(P) \cup U(P, A))$, $x \in U(P, A)$. So $F_1(P, Y) \subseteq U(P, A)$.

If $\bar{x} \in F_2(P, Y)$, then for every rule r, if $head(r) = x$ then $body(r)$ is false under Y. The corresponding clause $\bar{x} \vee v_1 \vee v_2 \vee \cdots \vee v_n$ is in $comp(P)$ and $\neg v_1, \neg v_2, \ldots, \neg v_n \in UP(comp(P) \cup Y)$, where v_i is the new variable stands for the body of the rule. Clearly $\bar{x} \in UP(comp(P) \cup Y)$. In particular, if $n = 0$, there is a clause $\bar{x}$ in $comp(P)$, then $\bar{x} \in UP(comp(P) \cup Y)$. Similar to the prove for $F_1(P, Y)$, $F_2(P, Y) \subseteq U(P, A)$.

If $x \in F_3(P, Y)$, then there exists a clause $\neg a \vee v_1 \vee v_2 \vee \cdots \vee v_n$ in $comp(P)$, $a \in Y$ and for any $j \neq i$, $\neg v_j \in Y$, then $v_i \in UP(comp(P) \cup Y)$. As the clause $\neg v_i \vee x$ is in $comp(P)$, $x \in UP(comp(P) \cup Y)$, hence, $F_3(P, Y) \subseteq U(P, A)$.

If $\bar{x} \in F_4(P, Y)$, then there is a literal $\neg a \in Y$, a rule $a \leftarrow l_1, \ldots, l_n.$, and $\{l_1, \ldots, l_n\}$ is true under $Y \cup \{x\}$. As Y is consistent, then $\{l_1, \ldots, l_n\}$ is not true under Y, so x is equivalent to a literal l_i $(1 \leq i \leq n)$ and for any $j \neq i$, $1 \leq j \leq n$, $l_j \in Y$. The corresponding clause is $a \vee \bar{l}_1 \vee \cdots \vee \bar{l}_n$, then $\bar{x} \in UP(comp(P) \cup Y)$, so $F_4(P, Y) \subseteq U(P, A)$.

If Y is inconsistent, then $F_5(P, Y) = Lit(P)$. $U(P, A)$ is also inconsistent, then $U(P, A) = Lit(P)$, $F_5(P, Y) \subseteq U(P, A)$.

As for any $Y \subseteq U(P, A)$, $F_{A \cup X}^P(Y) \subseteq U(P, A)$, we have proved that $Atleast(P, A \cup X) \subseteq U(P, A)$.

Now we prove that $\overline{Atoms(P) \setminus Atmost(P, A)} \subseteq U(P, A)$.

From Lemma 1, $M(P, A \cup X) = \overline{Atoms(P) \setminus Atmost(P, A \cup X)}$. Clearly for any $Y \subseteq U(P, A)$, $loop_0^A(P, Y) \subseteq U(P, A)$ and $F_2^A(P, Y) \subseteq U(P, A)$, so $M_P^A(P, Y) \subseteq U(P, A)$, $M(P, A \cup X) \subseteq U(P, A)$.

So $E_A^P(X) \subseteq U(P, A)$ and $Expand(P, A) \subseteq U(P, A)$. ■

Lemma 3 *For any simplified logic program P, and any $A \subseteq Lit(P)$. $U(P, A) \subseteq Expand(P, A)$.*

Proof: $U(P, A)$ is the least fixed point of our operator U_A^P defined by (5). For any set of literals X such that $X \subseteq Expand(P, A)$, we want to prove $U_A^P(X) \subseteq Expand(P, A)$. X is consistent, if not, $Expand(P, A) = Lit(P)$ and $U_A^P(X) \subseteq Expand(P, A)$. First we prove that $UP(comp(P) \cup X) \cap Lit(P) \subseteq Expand(P, A)$.

From the definition of $comp(P)$ in Preliminaries, some new variables are introduced and there are four kinds of clauses in $comp(P)$. We consider them one by one. First we give some notions that will be used in the proof.

A set of literals $X \subseteq Lit(comp(P))$ is called a *proper submodel* of a set of clauses $comp(P)$, if for any new variable v_i which stands for the body $\{l_1, \ldots, l_m\}$, $v_i \in X$ implies $\{l_1, \ldots, l_m\} \subseteq X$ and $\neg v_i \in X$ implies there exists some j, $1 \leq j \leq m$ such that $\bar{l}_j \in X$. For any set of literals $Y \subseteq Lit(comp(P))$, $Sub(Y) = \{v_i \mid v_i \in Y, v_i$ stands for the body $\{l_1, \ldots, l_m\}$, and $\{l_1, \ldots, l_m\} \not\subseteq Y\}$.

For any set of literals $X \subseteq Lit(comp(P))$, such that X is a proper submodel of $comp(P)$ and $(X \cap Lit(P)) \subseteq Expand(P, A)$.

For type 1, the clause is $\neg a$ and $\neg a \in F_2(X \cap Lit(P))$.

For type 2, the clause is $\bar{l}_1 \vee \cdots \vee \bar{l}_n \vee l$. Assume that it can be reduced to a unit clause under X. As the rule is a simplified rule, the head of the rule does not belong to the atoms appeared in the body, then there are only two cases. If the remaining literal is l, then $\{l_1, \ldots, l_n\} \subseteq X$, so $l \in F_1(P, X \cap Lit(P))$, $l \in Expand(P, A)$. If the remaining literal is $\bar{l}_i$, then $\bar{l} \in X$, and for any $j \neq i, 1 \leq j \leq n, l_j \in X$, so $\bar{l}_i \in F_4(P, X \cap Lit(P))$, $\bar{l}_i \in Expand(P, A)$. So if the unit clause c is reduced from one of this kind of clauses, then $c \in Expand(P, A)$.

For type 3, the clauses are $\neg a \vee v_1 \vee \cdots \vee v_m$, $v_i \vee \bar{l}_{1_i} \vee \cdots \vee \bar{l}_{n_i}$, $\neg v_i \vee l_{1_i}, \ldots \neg v_i \vee l_{n_i}$ $(1 \leq i \leq m)$.

For the clause $\neg a \vee v_1 \vee \cdots \vee v_m$, assume that it can be reduced to a unit clause under X. There are only two cases. If the remaining literal is $\neg a$, then $\{\neg v_1, \ldots, \neg v_m\} \subseteq X$. As X is a proper submodel of $comp(P)$, for any rule $r \in P$ if $head(r) = a$ then $body(r)$ is false under $X \cap Lit(P)$, so $\neg a \in F_2(P, X \cap Lit(P))$, $\neg a \in Expand(P, A)$. If the remaining literal is v_i, $v_i \notin Lit(P)$, so we do not need to consider this case.

For the clause $v_i \vee \bar{l}_1 \vee \cdots \vee \bar{l}_n$, assume that it can be reduced to a unit clause under X. There are only two cases. If the remaining literal is $\bar{l}_j$, then $\neg v_i \in X$ and for any $k \neq j$, $1 \leq k \leq n$, $l_k \in X$. As X is a proper submodel of $comp(P)$, then $\bar{l}_j \in X$, so $\bar{l}_j \in Expand(P, A)$. If the remaining literal is v_i, $v_i \notin Lit(P)$.

For the clause $\neg v_i \vee l_j$, assume that it can be reduced to a unit clause under X. There are only two cases. If the remaining literal is l_j, then $v_i \in X$. As X is a proper submodel of $comp(P)$, $l_j \in X$. So $l_j \in Expand(P, A)$. If the remaining literal is $\neg v_i$, $v_i \notin Lit(P)$.

So if the unit clause c is reduced from the clause of type 3, then $\{c\} \cap Lit(P) \subseteq Expand(P, A)$.

For type 4, the clause is $\bar{l}_1 \vee \cdots \vee \bar{l}_n$. Assume that it can be reduced to a unit clause under X. If the remaining literal is $\bar{l}_i$, then for any $j \neq i, 1 \leq j \leq n, l_j \in X$, so $\bar{l}_i \in F_4(P, X \cap Lit(P))$. So if the unit clause c is reduced from one of this kind of clauses, then $c \in Expand(P, A)$.

$unit_clause(assign(X, comp(P)))$ returns the union of unit clauses reduced from $comp(P)$ under X, we denote $UPO(comp(P), X)$ for short. So for any $X \subseteq Lit(comp(P))$, X is a proper submodel of $comp(P)$ and $X \cap Lit(P) \subseteq Expand(P, A)$, we have $UPO(comp(P), X) \cap Lit(P) \subseteq Expand(P, A)$.

Let $Y = UPO(comp(P), X)$, if Y is not a proper submodel of $comp(P)$, then there exists some $v_i \in Y$ which stands for the body $\{l_1, \ldots, l_n\}$ and for some $1 \leq j \leq n$, $l_j \notin Y$. v_i can only come from the clause $\neg a \vee v_1 \vee \cdots \vee v_m$, so $a \in Y$ and for all $k \neq i$, $1 \leq k \leq m$, $\neg v_k \in Y$, then $\{l_1, \ldots, l_n\} \subseteq F_3(P, Y \cap Lit(P))$, $UPO(comp(P), Sub(Y)) \cap Lit(P) \subseteq Expand(P, A)$. Let $Z = Y \setminus Sub(Y)$, it is clear that Z is a proper submodel of $comp(P)$, then $UPO(comp(P), Z) \cap Lit(P) \subseteq Expand(P, A)$. So $UPO(comp(P), Y) \cap Lit(P) = (UPO(comp(P), Z) \cup UPO(comp(P), Sub(Y))) \cap Lit(P) \subseteq Expand(P, A)$.

So $UP(comp(P) \cup X) \cap Lit(P) \subseteq Expand(P, A)$.

Now we prove that $loop_0(P, X) \subseteq Expand(P, A)$.

From Lemma 1, for any $Y \subseteq Expand(P, A)$, $loop_0^A(P, Y) \subseteq Expand(P, A)$, then $loop_0(P, X) \subseteq Expand(P, A)$.

So $U_A^P(X) \subseteq Expand(P, A)$ and $U(P, A) \subseteq Expand(P, A)$. ■

Journal of Artificial Intelligence Research 28 (2007) 431-451 Submitted 07/06; published 04/07

Discovering Classes of Strongly Equivalent Logic Programs

Fangzhen Lin FLIN@CS.UST.HK
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Yin Chen GZCHENYIN@GMAIL.COM
Department of Computer Science
South China Normal University
Guangzhou, P.R. China

Abstract

In this paper we apply computer-aided theorem discovery technique to discover theorems about strongly equivalent logic programs under the answer set semantics. Our discovered theorems capture new classes of strongly equivalent logic programs that can lead to new program simplification rules that preserve strong equivalence. Specifically, with the help of computers, we discovered exact conditions that capture the strong equivalence between a rule and the empty set, between two rules, between two rules and one of the two rules, between two rules and another rule, and between three rules and two of the three rules.

1. Introduction

In this paper we apply computer-aided theorem discovery technique to discover theorems about strongly equivalent logic programs under the answer set semantics. Our discovered theorems capture new classes of strongly equivalent logic programs that can lead to new program simplification rules that preserve strong equivalence.

Theorem discovery is a highly creative human process. Generally speaking, we can divide it into two steps: (i) conjecture formulation, and (ii) conjecture verification, and computers can help in both of these two steps. For instance, machine learning tools can be used in the first step, i.e. in coming up with reasonable conjectures, and automated deduction tools can be used in the second step, i.e. in verifying the correctness of these conjectures.

While theorem discovery may make use of learning, these two tasks are fundamentally different. Theorem discovery starts with a theory, and aims at finding *interesting* consequences of the theory, while learning is mostly about induction, i.e. it starts with examples/consequences, and aims at finding a theory that would explain the given examples/consequences.

Using computers to discover theorems is an old aspiration. There have been some success stories. For instance, AM (Lenat, 1979) was reported to be able to come up with some interesting concepts and theorems in number theory, and the remarkable systems described by Petkovsek, Wilf, and Zeilberger (1996) can discover many identities, especially hypergeometric identities involving sums of binomial coefficients that are important for the analyses of algorithms. Yet another example where "interesting" theorems can be discovered

©2007 AI Access Foundation. All rights reserved.

almost fully automatically is a recent work by Lin (2004) on discovering state invariants in planning domains. Lin showed that there are ways to classify many state constraints that are useful in planning according to their syntactic properties, and enumerate them easily for many domains. Furthermore, for many of these constraints whether they are invariants can be checked automatically. As a result, the system described by Lin (2004) can discover many common constraints in planning domains, and for the logistics domain, it could even discover a set of "complete" state invariants.

Following this line of research, in this paper, we consider the problem of discovering classes of strongly equivalent sets of logic program rules under answer set semantics. As noted by Lifschitz, Pearce, and Valverde (2001), if two sets of rules are strongly equivalent, then we can replace one by the other in any logic program without changing the semantics of the program. Thus identifying strongly equivalent sets of logic program rules is a useful exercise that may have applications in program simplification.

This paper is organized as follows. In the next section, we briefly review the basic concepts of logic programming under answer set semantics. Then in section 3 we state in more precise terms the type of theorems that we want to discover. In section 4 we prove some general theorems that will help us prove these theorems, and in section 5, we describe some of the theorems that we discovered. We then discuss an application to logic program simplification in section 6, and finally we conclude this paper in section 7.

2. Answer Set Programming

Traditional logic programming systems like Prolog solve problems by query answering. The user encodes knowledge about a domain by a set of rules, and solves a problem by issuing queries to the set of rules. In contrast, Answer Set Programming (ASP) (Niemelä, 1999; Lifschitz, 1999; Marek & Truszczynski, 1999) is a constraint-based programming paradigm. It is based on logic programming with answer set semantics (Gelfond & Lifschitz, 1988, 1991). To solve a problem, the user encodes the domain knowledge as a logic program in such a way that the answer sets of the program will correspond to the solutions to the original problem. Compared to other constraint-based programming paradigms, ASP allows natural encodings of recursive relations, and has built-in facilities for default reasoning. Several ASP solvers have been developed (Niemelä, Simons, & Syrjänen, 2000; Leone, Pfeifer, Faber, Eiter, Gottlob, Perri, & Scarcello, 2006; Lin & Zhao, 2004; Lierler & Maratea, 2004). To date, ASP has been used in space shuttle planning (Nogueira, Balduccini, Gelfond, Watson, & Barry, 2001), evolutional linguistics (Erdem, Lifschitz, Nakhleh, & Ringe, 2003), and others. In the following, we briefly review some basic notions in ASP.

Let L be a propositional language, i.e. a set of atoms. In this paper we shall consider logic programs with rules of the following form:

$$h_1; \cdots; h_k \leftarrow p_1, \cdots, p_m, not\, p_{m+1}, \cdots, not\, p_n \tag{1}$$

where h_i's and p_i's are atoms in L. So a logic program here can have default negation (not), constraints (when $k = 0$), and disjunctions in the head of its rules. In the following, if r is a rule of the above form, we write Hd_r to denote the set $\{h_1, ..., h_k\}$, Ps_r the set $\{p_1, ..., p_m\}$, and Ng_r the set $\{p_{m+1}, ..., p_n\}$. Thus a rule r can also be written as $Hd_r \leftarrow Ps_r, not\, Ng_r$. The semantics of these programs are given by *answer sets* (Gelfond & Lifschitz, 1991), which

are defined by a fixed-point operator through what has been known as *Gelfond-Lifschitz transformation*. Let X be a subset of L, and P a logic program. The Gelfond-Lifschitz transformation of P on X, written P^X, is the set of rules obtained from P according to the following two rules:

1. If a rule of the form (1) is in P, and $p_i \in X$ for some $m+1 \leq i \leq n$, then delete this rule.

2. Delete all literals of the form $not\, p_i$ in the bodies of the remaining rules.

For instance, if P is the set of following rules:

$$a; b \leftarrow$$
$$c \leftarrow not\, a$$

then $P^{\{a\}}$ is $\{a; b \leftarrow\}$, and $P^{\{b\}}$ is $\{(a; b \leftarrow), (c \leftarrow)\}$.

Clearly, for any X and P, P^X is a set of rules which do not have the "*not*" operator. Now a set X is an answer set of P if X is a minimal set of atoms that *satisfies* every rule in P^X, where X satisfies a rule of the form

$$h_1; \cdots; h_k \leftarrow p_1, \cdots, p_m$$

if for some $1 \leq i \leq k$, $h_i \in X$ whenever $\{p_1, ..., p_m\} \subseteq X$. For instance, for the above program, both $\{a\}$ and $\{b, c\}$ are answer sets, and they are the only answer sets of the program.

Two logic programs P_1 and P_2 are said to be *equivalent* if they have the same answer sets, and *strongly equivalent* (Lifschitz et al., 2001) (under the language L), written $P_1 \simeq_{se} P_2$, if for any logic program P in L, $P \cup P_1$ and $P \cup P_2$ are equivalent (thus we write $P_1 \not\simeq_{se} P_2$ when P_1 and P_2 are not strongly equivalent). For example, $\{a \leftarrow b\}$ and $\{a \leftarrow c\}$ are equivalent, but not strongly equivalent. It can be shown that $\{a \leftarrow not\, a\} \simeq_{se} \{\leftarrow not\, a\}$. As in the abstract, we also say that a rule r is strongly equivalent to another rule r', written $r \simeq_{se} r'$, if $\{r\} \simeq_{se} \{r'\}$, and two rules r_1 and r_2 are strongly equivalent to a rule r, written $\{r_1, r_2\} \simeq_{se} r$, if $\{r_1, r_2\} \simeq_{se} \{r\}$, and so on.

The notion of strong equivalence is important for ASP for several reasons. First of all, it helps us understand the answer set semantics. For instance, Turner (2003) showed that the disjunctive rule $(a; b \leftarrow)$ is not strongly equivalent to any set of normal rules. This implies that there cannot be a modular translation from disjunctive logic programs to normal logic programs. However,

$$\{(a; b \leftarrow), (\leftarrow a, b)\}$$

is strongly equivalent to

$$\{(a \leftarrow not\, b), (b \leftarrow not\, a), (\leftarrow a, b)\}.$$

This means that under the constraint $(\leftarrow a, b)$, the disjunctive rule $(a; b \leftarrow)$ can be replaced by two rules without disjunction. Secondly, as we mentioned in the introduction, if P_1 and P_2 are strongly equivalent, then they are interchangeable regardless of where they occur. Thus if we have a large repertoire of pairs of strongly equivalent logic programs, we could

use them to transform a given program into one that is most suitable to the need in hand. In particular, it could help us simplify a program for the purpose of computing its answer sets. As we shall see, our discovered theorems will contribute significantly to this repertoire.

Lifschitz et al. (2001) showed that checking for strong equivalence between two logic programs can be done in the logic of here-and-there, a three-valued non-classical logic somewhere between classical logic and intuitionistic logic. Lin (2002) provided a mapping from logic programs to propositional theories and showed that two logic programs are strongly equivalent iff their corresponding theories in propositional logic are equivalent. This result will be used here both for generating example pairs of strongly equivalent logic programs, and for verifying a conjecture. We repeat it here.

Let P_1 and P_2 be two finite logic programs, and L the set of atoms in them.

Theorem 1 *(Lin, 2002) $P_1 \simeq_{se} P_2$ iff in propositional logic, the following sentence is valid:*

$$(\bigwedge_{p \in L} p \supset p') \supset [\bigwedge_{r \in P_1} \delta(r) \equiv \bigwedge_{r \in P_2} \delta(r)], \tag{2}$$

where for each $p \in L$, p' is a new atom, and for each rule r of the form (1), $\delta(r)$ is the conjunction of the following two sentences:

$$p_1 \wedge \cdots \wedge p_m \wedge \neg p'_{m+1} \wedge \cdots \wedge \neg p'_n \supset h_1 \vee \cdots \vee h_k, \tag{3}$$
$$p'_1 \wedge \cdots \wedge p'_m \wedge \neg p'_{m+1} \wedge \cdots \wedge \neg p'_n \supset h'_1 \vee \cdots \vee h'_k. \tag{4}$$

Notice that if $m = n = 0$, then the left sides of the implications in (3) and (4) are considered to be $true$, and if $k = 0$, then the right sides of the implications in (3) and (4) are considered to be $false$.

In general checking if two sets of rules are strongly equivalent is coNP-complete (c.f. Turner, 2001; Pearce, Tompits, & Woltran, 2001; Lin, 2002).

3. The Problem

As we mentioned above, one possible use of the notion of strongly equivalent logic programs is in program simplification. For instance, given a logic program, for each rule r in it, we may ask whether it can be deleted without knowing what other rules are in P, i.e. whether $\{r\}$ is strongly equivalent to the empty set. Or we may ask whether a rule r in P can be deleted if one knows that another rule r' is already in P, i.e. whether $\{r, r'\}$ is strongly equivalent to $\{r'\}$. In general, we may ask the following k-m-n question: Is $\{r_1, ..., r_k, u_1, ..., u_m\} \simeq_{se} \{r_1, ..., r_k, v_1, ..., v_n\}$? Thus our theorem discovery task is to come up, for a given k-m-n problem, a computationally effective condition that holds if and only if the answer to the k-m-n question is positive.

Now suppose we have such a condition C, and suppose that when

$$\{r_1, ..., r_k, u_1, ..., u_m\} \simeq_{se} \{r_1, ..., r_k, v_1, ..., v_n\},$$

it is better to replace $\{u_1, ..., u_m\}$ by $\{v_1, ..., v_n\}$ in the presence of $r_1, ..., r_k$ for the purpose of, say computing the answer sets of a program. One way to use this result to simplify a given program P is to first choose k rules in P, and for any other m rules in it, try to find

n rules so that the condition C holds, and then replace the m rules in P by the simpler n rules.

However, even if checking whether C holds would take a negligible constant time, using the above procedure to simplify a given logic program will be practical only when k, m, n are all very small or when k is almost the same as the number of the rules in the given program, and m and n are very small. Thus it seems to us that it is worthwhile to solve the k-m-n problem only when k, m, n are small. In particular, in this paper, we shall concentrate on the 0-1-0 problem (whether a rule can always be deleted), the 0-1-1 problem (whether a rule can always be replaced by another one), the 1-1-0 problem (in the presence of a rule, whether another rule can be deleted), the 2-1-0 problem (in the presence of two rules, whether a rule can always be deleted), and the 0-2-1 problem (if a pair of rules can be replaced by a single rule).

An example of theorems that we want to discover about these problems is as follows:

$$\text{For any rule } r,\ r \simeq_{se} \emptyset \text{ iff } (Hd_r \cup Ng_r) \cap Ps_r \neq \emptyset. \tag{5}$$

4. Some General Theorems

In this section, we prove some general theorems that will help us verify whether an assertion like (5) above is true.

Let L be a propositional language, i.e. a set of atoms. From L, construct a first-order language F_L with equality, two unary predicates H_1 and H_2, three unary predicates Hd_r, Ps_r, and Ng_r for each logic program rule r in L (we assume that each rule in L has a unique name), and three unary predicates X_i, Y_i, and Z_i for each positive number i.

Notice that we have used Hd_r, Ps_r, and Ng_r to denote sets of atoms previously, but now we overload them as unary predicates. Naturally, the intended interpretations of these unary predicates are their respective sets.

Definition 1 *Given a set L of atoms, an* intended model *of F_L is one whose domain is L, and for each rule r in L, the unary predicates Ps_r, Hd_r, and Ng_r are interpreted by their corresponding sets of atoms, Ps_r, Hd_r, and Ng_r, respectively.*

Conditions on rules in L, such as $Ps_r \cap Ng_r \neq \emptyset$, will be expressed by special sentences called *properties* in F_L.

Definition 2 *A sentence of F_L is a property about n rules if it is constructed from equality and predicates X_i, Y_i, and Z_i, $1 \leq i \leq n$. A property Φ about n rules is true (holds) on a sequence $P = [r_1, ..., r_n]$ of n rules if $\Phi[P]$ is true in an intended model of F_L, where $\Phi[P]$ is obtained from Φ by replacing each X_i by Hd_{r_i}, Y_i by Ps_{r_i}, and Z_i by Ng_{r_i}.*

Notice that since $\Phi[P]$ does not mention predicates X_i, Y_i, Z_i, H_1, and H_2, if it is true in one intended model, then it is true in all intended models.

As we have mentioned above, we are interested in capturing the strong equivalence between two programs by a computationally effective condition. More specifically, for some small k, m, and n, we are interested in finding a property Φ about $k + m + n$ rules such that for any sequence of $k + m + n$ rules, $P = [r_1, ..., r_k, u_1, ..., u_m, v_1, ..., v_n]$,

$$\{r_1, ..., r_k, u_1, ..., u_m\} \simeq_{se} \{r_1, ..., r_k, v_1, ..., v_n\} \text{ iff } \Phi \text{ is true on } P. \tag{6}$$

We shall now prove some general theorems that can help us verify the above assertion for a class of formulas Φ.

First of all, Theorem 1 can be reformulated in F_L as follows by reading $H_1(p)$ as "p holds", and $H_2(p)$ as "p' holds":

Theorem 2 *$P_1 \simeq_{se} P_2$ in L iff the following sentence*

$$\forall x(H_1(x) \supset H_2(x)) \supset [\bigwedge_{r \in P_1} \gamma(r) \equiv \bigwedge_{r \in P_2} \gamma(r)] \tag{7}$$

is true in all intended models of F_L, where $\gamma(r)$ is the conjunction of the following two sentences:

$$[\forall x(Ps_r(x) \supset H_1(x)) \wedge \forall x(Ng_r(x) \supset \neg H_2(x))] \supset \exists x(Hd_r(x) \wedge H_1(x)), \tag{8}$$

$$[\forall x(Ps_r(x) \supset H_2(x)) \wedge \forall x(Ng_r(x) \supset \neg H_2(x))] \supset \exists x(Hd_r(x) \wedge H_2(x)). \tag{9}$$

In first order logic, if a prenex formula of the form $\exists \vec{x} \forall \vec{y} B$ is satisfiable, then it is satisfiable in a structure with n elements, where B is a formula that contains no quantifiers, constants, or function symbols, and n is the length of $\vec{x}$ if it is non-empty, and 1 when $\vec{x}$ is empty. We can prove a similar result for our first-order languages and their intended models here.

Definition 3 *A sentence of F_L is an* extended *property about n rules if it is constructed from equality and predicates X_i, Y_i, and Z_i, $1 \leq i \leq n$, and H_1 and H_2. An extended property Φ about n rules is true (holds) on a sequence $P = [r_1, ..., r_n]$ of n rules in a model M if $\Phi[P]$ is true in M, where $\Phi[P]$ is obtained from Φ by replacing each X_i by Hd_{r_i}, Y_i by Ps_{r_i}, and Z_i by Ng_{r_i}.*

Definition 4 *In the following, if $P = [r_1, ..., r_n]$ is a tuple of rules in L, and L' is a subset of L, then we define the* restriction *of P on L' to be $[r'_1, ..., r'_n]$, where r'_i is*

$$Hd_{r_i} \cap L' \leftarrow Ps_{r_i} \cap L', not\,(Ng_{r_i} \cap L').$$

Lemma 1 *Let Φ be an extended property in F_L about n rules, and of the form $\exists \vec{x} \forall \vec{y} Q$, where $\vec{x}$ is a tuple of w variables, and Q a formula that does not have any quantifiers. If Φ holds on a sequence P of n rules in an intended model M of F_L, then there is a subset L' of L such that L' has at most w atoms (or one atom when $w = 0$), and Φ holds on the restriction of P on L' in an intended model of $F_{L'}$.*

Proof: Suppose M is an intended model of F_L such that $M \models \Phi[P]$. Thus there is a tuple $\vec{p}$ of w (or one when $w = 0$) atoms in L such that $M \models \forall \vec{y} Q[P](\vec{x}/\vec{p})$. Now let L' be the set of atoms in $\vec{p}$, and M' defined as follows:

- Each of the predicates H_1, H_2, X_i, Y_i, and Z_i, $i \geq 1$, is interpreted as the restriction of its interpretation in M on L'.
- For each rule r in L', the predicates Hd_r, Ps_r, and Ng_r are interpreted the same as they are in M. This is well-defined as r is also a rule in L,

Then M' is an intended model of $F_{L'}$. Let P' be the restriction of P on L'. Then P' is a tuple of rules in L'. Since Q has no quantifiers (and the language has no function symbols), for any instantiation $\vec{u}$ of $\vec{y}$ in L', $M \models Q[P](\vec{x}/\vec{p})(\vec{y}/\vec{u})$ iff $M' \models Q[P'](\vec{x}/\vec{p})(\vec{y}/\vec{u})$. Since $M \models \forall\vec{y}Q[P](\vec{x}/\vec{p})$, we have $M' \models \forall\vec{y}Q[P'](\vec{x}/\vec{p})$, Thus $M' \models \exists\vec{x}\forall\vec{y}Q[P']$. □

Using Theorem 2 and this lemma, we can show the following theorem which will enable us to automate the verification of the "if" part of (6) when the property Φ is in the prenex format.

Theorem 3 *Without loss of generality, suppose $m \geq n$. If Φ is a property about $k+m+n$ rules of the form $\exists\vec{x}\forall\vec{y}Q$, where $\vec{x}$ is a tuple of w variables, and Q a formula that does not have any quantifiers, then the following two assertions are equivalent:*

(a) For any sequence of $k+m+n$ rules, $P = [r_1, ..., r_k, u_1, ..., u_m, v_1, ..., v_n]$, if Φ is true on P, then $\{r_1, ..., r_k, u_1, ..., u_m\} \simeq_{se} \{r_1, ..., r_k, v_1, ..., v_n\}$.

(b) (b.1) If $n > 0$, then for any sequence $P = [r_1, ..., r_k, u_1, ..., u_m, v_1, ..., v_n]$ of rules with at most $w + 2(k+m)$ atoms, if Φ is true on P, then

$$\{r_1, ..., r_k, u_1, ..., u_m\} \simeq_{se} \{r_1, ..., r_k, v_1, ..., v_n\}.$$

(b.2) If $n = 0$, then for any sequence $P = [r_1, ..., r_k, u_1, ..., u_m]$ of rules with at most K atoms, if Φ is true on P, then

$$\{r_1, ..., r_k, u_1, ..., u_m\} \simeq_{se} \{r_1, ..., r_k\},$$

where K is $w + 2k$ if $w + 2k > 0$, and $K = 1$ otherwise.

Proof: If (a) then (b) is obvious. We assume that (b) is true, and show that (a) holds as well. Suppose first that $n > 0$. Suppose $P = [r_1, ..., r_k, u_1, ..., u_m, v_1, ..., v_n]$ is a sequence of $k+m+n$ rules in a language L such that Φ is true on P, and

$$\{r_1, ..., r_k, u_1, ..., u_m\} \not\simeq_{se} \{r_1, ..., r_k, v_1, ..., v_n\}.$$

Thus there is an intended model of F_L that satisfies $\Phi[P]$, and an intended model M of F_L that satisfies the following sentence:

$$(\forall x)H_1(x) \supset H_2(x) \wedge \neg[\bigwedge_{r\in P_1} \gamma(r) \equiv \bigwedge_{r\in P_2} \gamma(r)],$$

where $P_1 = \{r_1, ..., r_k, u_1, ..., u_m\}$, and $P_2 = \{r_1, ..., r_k, v_1, ..., v_n\}$. As we noted after Definition 2, M will also satisfy $\Phi[P]$. Thus M satisfies the following sentence

$$\Phi[P] \wedge (\forall x)H_1(x) \supset H_2(x) \wedge \{[\bigwedge_{r\in P_1} \gamma(r) \wedge \neg \bigwedge_{r\in P_3} \gamma(r)] \vee [\bigwedge_{r\in P_2} \gamma(r) \wedge \neg \bigwedge_{r\in P_4} \gamma(r)]\}, \quad (10)$$

where $P_3 = \{v_1, ..., v_n\}$, and $P_4 = \{u_1, ..., u_m\}$.

Now for any rule r, there is an extended property $\varphi(x, y)$ of one rule that does not mention any quantifiers such that $\gamma(r)$ is equivalent to $\exists x, y.\varphi[r]$. Thus for any tuple Q of t rules, there is an extended property φ of t rules that does not mention any quantifiers such that $\bigwedge_{r\in Q} \gamma(r)$ is equivalent to $\exists\vec{y}.\varphi[Q]$, where $\vec{y}$ is a tuple of $2t$ variables.

Thus there is

- a tuple $\vec{z_1}$ of $2(k+m)$ variables, a tuple $\vec{z_2}$ of variables, an extended property φ_1 of $k+m+n$ rules that does not have any quantifiers, and whose free variables are in $\vec{z_1}$ and $\vec{z_2}$; and
- a tuple $\vec{z_3}$ of $2(k+n)$ variables, a tuple $\vec{v_4}$ of variables, and an extended property φ_2 of $k+m+n$ rules that does not have any quantifiers, and whose free variables are in $\vec{z_3}$ and $\vec{z_4}$

such that $\vec{v_1}, \vec{v_2}, \vec{v_3}, \vec{v_4}$ do not have common variables in them, and (10) is equivalent to the following sentence:

$$\{\Phi \wedge \forall x(H_1(x) \supset H_2(x)) \wedge (\exists \vec{z_1} \forall \vec{z_2} \varphi_1 \vee \exists \vec{z_3} \forall \vec{z_4} \varphi_2)\}[P].$$

Since we have assumed that $m \geq n$, thus there is an extended property φ_3 about $k+m+n$ rules that does not mention any quantifiers and function symbols, and whose free variables are among $\vec{z_1}$, $\vec{z_2}$, and $\vec{z_4}$ such that the above sentence is equivalent to the following sentence:

$$(\Phi \wedge \forall x(H_1(x) \supset H_2(x)) \wedge \exists \vec{z_1}(\forall \vec{z_2}, \vec{z_4})\varphi_3)[P].$$

Now given the form of Φ assumed in the theorem, there is a tuple $\vec{z_5}$ of $w + 2(k+m)$ variables, a tuple $\vec{z_6}$ of variables, and an extended property Ψ of $k+m+n$ rules that does not mention any quantifiers, and whose free variables are among $\vec{z_5}$, and $\vec{z_6}$ such that the above sentence is equivalent to $(\exists \vec{z_5})(\forall \vec{z_6})\Psi[P]$.

By Lemma 1, there is a subset L' of L that has at most $w + 2(k+m)$ atoms such that $(\exists \vec{z_5})(\forall \vec{z_6})\Psi$ holds on P', where P' is the restriction of P on L'. If

$$P' = [r'_1, ..., r'_k, u'_1, ..., u'_m, v'_1, ..., v'_n],$$

then this will mean that Φ is true on P', and $\{r'_1, ..., r'_k, u'_1, ..., u'_m\} \not\approx_{se} \{r'_1, ..., r'_k, v'_1, ..., v'_n\}$. This shows that if (b.1), then (a).

The proof that if (b.2) then (a) is exactly the same except now that

$$[\bigwedge_{r \in P_1} \gamma(r) \equiv \bigwedge_{r \in P_2} \gamma(r)]$$

is equivalent to

$$[\bigwedge_{r \in P_2} \gamma(r) \supset \bigwedge_{r \in P_1} \gamma(r)].$$

□

The "only if" part of (6) can often be proved with the help of the following theorem.

Theorem 4 *Let L_1 and L_2 be two languages, and f a function from L_1 to L_2. If P_1 and P_2 are two programs in L_1 that are strongly equivalent, then $f(P_1)$ and $f(P_2)$ are two programs in L_2 that are also strongly equivalent. Here $f(P)$ is obtained from P by replacing each atom p in it by $f(p)$.*

Proof: By Theorem 1 and the fact that in propositional logic, if φ is a tautology, and f a function from L_1 to L_2, then $f(\varphi)$ is also a tautology, where $f(\varphi)$ is the formula obtained from φ by replacing each atom p in it by $f(p)$. □

For an example of using the theorems in this section for proving assertions of the form (6), see Section 5.1.

5. Computer-Aided Theorem Discovery

Given a *k-m-n* problem, our strategy for discovering theorems about it is as follows:

1. Choose a small language L;
2. Generate all possible triples
$$(\{r_1, ..., r_k\}, \{u_1, ..., u_m\}, \{v_1, ..., v_n\}) \tag{11}$$
of sets of rules in L such that $\{r_1, ..., r_k, u_1, ..., u_m\} \simeq_{se} \{r_1, ..., r_k, v_1, ..., v_n\}$ in L;
3. Formulate a conjecture on the *k-m-n* problem that holds in the language L, i.e. a condition that is true for a triple of the form (11) iff it is generated in Step 2;
4. Verify the correctness of this conjecture in the general case.

This process may have to be iterated. For instance, a conjecture formulated in Step 3 may fail to generalize in Step 4, so we either need to formulate a new conjecture or start all over again in step 1 using a larger language.

Ideally, we would like this process to be automatic. However, it is difficult to automate Steps 3 and 4 - the number of possible patterns that we need to examine in order to come up with a good conjecture in Step 3 is huge, and we do not have a general theorem that enables us to automate the verification part in Step 4. While Theorem 3 enables us to automate the proof of the sufficient part of the assertion (6) for a class of formulas Φ, we do not have a similar result for the necessary part - as we shall see below, Theorem 4 helps a lot here, but it does not provide an automated procedure. Nonetheless, computers play a crucial role in all steps, and in the following we report some of the theorems discovered using the above procedure.

5.1 The 0-1-0 Problem

This problem asks if a given rule is strongly equivalent to the empty set, thus can always be deleted from any program. We have the following experimental result:

Lemma 2 *If a rule r mentions at most three distinct atoms, then $r \simeq_{se} \emptyset$ iff*
$$(Hd_r \cup Ng_r) \cap Ps_r \neq \emptyset.$$

Using Theorem 4, we can show the following result:

Lemma 3 *If there is a rule r of the form (1) such that $r \simeq_{se} \emptyset$ and $(Hd_r \cup Ng_r) \cap Ps_r \neq \emptyset$ is not true, then there is such a rule that mentions at most three atoms.*

Proof: Suppose $r \simeq_{se} \emptyset$, $Hd_r \cap Ps_r = \emptyset$, and $Ps_r \cap Ng_r = \emptyset$. Suppose L is the set of atoms in r, and a, b, c are three new atoms. Let
$$f(p) = \begin{cases} a & p \in Hd_r \\ b & p \in Ps_r \\ c & otherwise \end{cases}$$

By Theorem 4, we also have $f(r) \simeq_{se} \emptyset$. By the construction of f, we also have $Hd_{f(r)} \cap Ps_{f(r)} = \emptyset$, and $Ps_{f(r)} \cap Ng_{f(r)} = \emptyset$, and that $f(r)$ mentions at most three distinct atoms. □

Theorem 5 (The 0-1-0 problem) *Lemma 2 holds in the general case, i.e. without any restriction on the number of atoms in r.*

Proof: We notice that the condition in Lemma 2, $(Hd_r \cup Ng_r) \cap Ps_r \neq \emptyset$, is equivalent to the following property

$$\exists x.(X_1(x) \vee Z_1(x)) \wedge Y_1(x)$$

being true on $[r]$. Thus the "if" part follows from Theorem 3 and Lemma 2. The "only if" part follows from Lemma 2 and Lemma 3. □

The "if" part of the theorem is already well-known, first proved by Osorio *et. al.* (2001). The "only if" part has also been proved recently by Inoue and Sakama (2004). While we did not discover anything new in this case, it is reassuring that the methodology works.

We notice here that there is no need to consider the 0-n-0 problem for $n > 1$, because for any n, $\{r_1, ..., r_n\}$ is strongly equivalent to $\emptyset$ iff for each $1 \leq i \leq n$, $\{r_i\}$ is strongly equivalent to $\emptyset$.

5.2 The 1-1-0 and the 0-1-1 Problems

The 1-1-0 problem asks if a rule can always be deleted in the presence of another rule, and the 0-1-1 problem asks if a rule can always be replaced by another one. We first solve the 1-1-0 problem, and the solution to the 0-1-1 problem will come as a corollary.

We have the following experimental result for the 1-1-0 problem:

Lemma 4 *For any two rules r_1 and r_2 that mentions at most three atoms, $\{r_1, r_2\}$ and $\{r_1\}$ are strongly equivalent iff one of the following two conditions is true:*

1. $r_2 \simeq_{se} \emptyset$.
2. $Ps_{r_1} \subseteq Ps_{r_2}$, $Ng_{r_1} \subseteq Ng_{r_2}$, *and* $Hd_{r_1} \subseteq Hd_{r_2} \cup Ng_{r_2}$.

Lemma 5 *If there are two rules r_1 and r_2 such that $\{r_1, r_2\} \simeq_{se} \{r_2\}$, but none of the two conditions in Lemma 4 hold, then there are two such rules that mention at most three atoms.*

Proof: Suppose there are two rules r_1, r_2 such that $\{r_1, r_2\} \simeq_{se} \{r_2\}$, and none of the two conditions in Lemma 4 hold. Let L be the set of atoms in r_1, r_2.

Without loss of generality, suppose a_1 is an atom that makes the condition (2) in Lemma 4 false. If $Ps_{r_2} \setminus \{a_1\}$ is not empty, let a_2 be an atom in it. Let $L' = \{a_1, a_2, a_3\}$, where a_3 is a new atom, and f be a function from L to L' as following:

$$f(a) = \begin{cases} a_1 & a = a_1 \\ a_2 & a \in Ps_{r_2} \setminus \{a_1\} \\ a_3 & otherwise \end{cases}$$

clearly, $f(r_1)$ and $f(r_2)$ mention at most three distinct atoms, and by Theorem 4,

$$\{f(r_1), f(r_2)\} \simeq_{se} f(r_1).$$

We show that none of the two conditions in Lemma 4 hold for $f(r_1)$ and $f(r_2)$ either.

We show first that $f(r_2) \not\simeq_{se} \emptyset$. By Theorem 5, we need to show that

$$S = Ps_{f(r_2)} \cap (Hd_{f(r_2)} \cup Ng_{f(r_2)})$$

is empty. If $a_1 \in S$, then by the construction of f, $a_1 \in Ps_{r_2} \cap (Hd_{r_2} \cup Ng_{r_2})$, a contradiction with the assumption that r_2 is not strongly equivalent to $\emptyset$. Similarly, if $a_2 \in S$, then by the construction of f, $a_2 \in Ps_{r_2} \cap (Hd_{r_2} \cup Ng_{r_2})$, a contradiction with the assumption that r_2 is not strongly equivalent to $\emptyset$. But then a_3 cannot be in S as a_3 cannot be in $Ps_{f(r_2)}$. Thus S must be empty.

We now show that it is not the case that $Ps_{f(r_1)} \subseteq Ps_{f(r_2)}$, $Ng_{f(r_1)} \subseteq Ng_{f(r_2)}$, and $Hd_{f(r_1)} \subseteq Hd_{f(r_2)} \cup Ng_{f(r_2)}$. By our assumption, a_1 is an atom that makes either $Ps_{r_1} \subseteq Ps_{r_2}$, $Ng_{r_1} \subseteq Ng_{r_2}$, or $Hd_{r_1} \subseteq Hd_{r_2} \cup Ng_{r_2}$ false. There are three cases here. Suppose a_1 makes $Ps_{r_1} \subseteq Ps_{r_2}$ false, i.e. $a_1 \in Ps_{r_1}$ but $a_1 \notin Ps_{r_2}$. Then by our construction of f, we also have that $a_1 \in Ps_{f(r_1)}$ but $a_1 \notin Ps_{f(r_2)}$. The other two cases are similar. □

Theorem 6 (The 1-1-0 problem) *Lemma 4 holds in the general case, without any restriction on the number of atoms in r_1 and r_2.*

Proof: The condition in Lemma 4 is equivalent to the following property

$$[\exists x.(X_2(x) \vee Z_2(x)) \wedge Y_2(x)] \vee$$
$$\{[\forall x.Y_1(x) \supset Y_2(x)] \wedge [\forall x.Z_1(x) \supset Z_2(x)] \wedge [\forall x.X_1(x) \supset (X_2(x) \vee Z_2(x))]\}$$

being true on $[r_1, r_2]$. Thus the "if" part follows from Theorem 3 and Lemma 4, by noticing that the above property can be written as $\exists x \forall \vec{y}.Q$ as required by Theorem 3. The "only if" part follows from Lemma 4 and Lemma 5. □

Thus if a rule r_2 cannot be deleted on its own but can be deleted in the presence of another rule r_1, then it must be the case that r_2 is redundant given r_1: if the body of r_2 is satisfied, then the body of r_1 is satisfied as well; furthermore, r_2 can entail no more than what can be entailed by r_1 ($Hd_{r_1} \subseteq Hd_{r_2} \cup Ng_{r_2}$).

Osorio et al. (2001) proved that $\{r_1, r_2\} \simeq_{se} r_1$ if either $Ps_{r_1} \cup Ng_{r_1} = \emptyset$ and $Hd_{r_1} \subseteq Ng_{r_2}$ or $Ps_{r_1} \subseteq Ps_{r_2}$, $Ng_{r_1} \subseteq Ng_{r_2}$, and $Hd_{r_1} \subseteq Hd_{r_2}$. More recently, Eiter et al. (2004) showed that $\{r_1, r_2\} \simeq_{se} r_1$ if r_1 *s-implies* r_2 (Wang & Zhou, 2005), i.e. if there exists a set $A \subseteq Ng_{r_2}$ such that $Hd_{r_1} \subseteq Hd_{r_2} \cup A$, $Ng_{r_1} \subseteq Ng_{r_2} \setminus A$, and $Ps_{r_1} \subseteq Ps_{r_2}$.

As one can see, these are all special cases of the "if" part of Theorem 6. Our result is actually more general. For instance, these special cases do not apply to

$$\{(c \leftarrow b, not\, c),\ (\leftarrow b, not\, c)\}$$

and

$$\{c \leftarrow b, not\, c\},$$

but one can easily show that these two sets are strongly equivalent using our theorem.

¿From our solution to the 1-1-0 problem, we can derive a solution to the 0-1-1 problem.

Theorem 7 (The 0-1-1 problem) *For any two rules r_1 and r_2, $r_1 \simeq_{se} r_2$ iff one of the following two conditions is true:*

1. $r_1 \simeq_{se} r_2 \simeq_{se} \emptyset$.
2. $Ps_{r_1} = Ps_{r_2}$, $Ng_{r_1} = Ng_{r_2}$, *and* $Hd_{r_1} \cup Ng_{r_1} = Hd_{r_2} \cup Ng_{r_2}$.

Proof: By Theorem 1, it is easy to see that $r_1 \simeq_{se} r_2$ iff $\{r_1, r_2\} \simeq_{se} r_1$ and $\{r_1, r_2\} \simeq_{se} r_2$. □

Thus two rules r_1 and r_2 can always be interchanged if either both of them can be deleted (strongly equivalent to the empty set) or they have the same body, and the same consequences when the body is true. For instance, we have $\{a \leftarrow B, not\ a\} \simeq_{se} \{\leftarrow B, not\ a\}$ no matter what B is, because the two rules have the same body, and when the body is true, the same consequence - a contradiction. As another example, we have

$$\{a; b \leftarrow not\ a\} \simeq_{se} \{b \leftarrow not\ a\},$$

because the two rules have the same body, and, when the body is true, the same consequence, b.

5.3 The 2-1-0, 0-2-1, and 0-2-2 Problems

The 2-1-0 problem asks if a rule can be deleted in the presence of another two rules, the 0-2-1 problem asks if two rules can be replaced by a single rule, and the 0-2-2 problem asks if two rules can be replaced by another two rules. Similar to the previous subsection, the solution to the 0-2-1 and 0-2-2 problems will follow from a solution to the 2-1-0 problem.

The experiment on the 2-1-0 problem was more difficult because as it turned out, we have to consider a language with six atoms in this case. In principle, given a language L, every subset of L can be the *Hd*, *Ps*, or *Ng* of a rule. Thus when the size of L is six, there are in principle $(2^6)^3 - 1 = 262,143$ possible rules, and $262,143^3$ triples of them. However, we can cut down the numbers significantly with the results that we already have proved.

First, we only have to consider rules that do not have common elements in any of the two sets in $\{Hd, Ps, Ng\}$: if either *Hd* and *Ps* or *Ps* and *Ng* have a common element, then by Theorem 5, this rule can be deleted; if *Hd* and *Ng* have common elements, then according to Theorem 7, we obtain a strongly equivalent rule by deleting the common elements in *Hd*. In the following, we call such rules *canonical*, that is, a rule r is canonical if

$$Hd_r \cap Ps_r = Hd_r \cap Ng_r = Ps_r \cap Ng_r = \emptyset.$$

Secondly, we do not have to consider isomorphic rules: if there is a one-to-one onto function from L to L that maps $\{r_1, r_2, r_3\}$ to $\{r'_1, r'_2, r'_3\}$, then these two sets of rules are essentially the same except for the names of atoms in them.

Thus by considering only canonical rules and using a certain normal form for triples of rules that avoids isomorphic rules, we ended up with roughly 120 million triples of rules to consider for verifying the following result, which took about 10 hours on a Solaris server consisting of 8 Sun Ultra-SPARC III 900Mhz CPUs with 8GB RAM.

For more details on the experiment on 2-1-0 problem, please refer to (Chen, Lin, & Li, 2005).

Lemma 6 *For any three canonical rules r_1, r_2 and r_3 that mention at most six atoms, $\{r_1, r_2, r_3\} \simeq_{se} \{r_1, r_2\}$ iff one of the following three conditions is true:*

1. $\{r_1, r_3\} \simeq_{se} r_1$.
2. $\{r_2, r_3\} \simeq_{se} r_2$.
3. *There is an atom p such that:*
   - 3.1 $p \in (Ps_{r_1} \cup Ps_{r_2}) \cap (Hd_{r_1} \cup Hd_{r_2} \cup Ng_{r_1} \cup Ng_{r_2})$
   - 3.2 $Hd_{r_i} \setminus \{p\} \subseteq Hd_{r_3} \cup Ng_{r_3}$ *and* $Ps_{r_i} \setminus \{p\} \subseteq Ps_{r_3}$ *and* $Ng_{r_i} \setminus \{p\} \subseteq Ng_{r_3}$, *where* $i = 1, 2$
   - 3.3 *If* $p \in Ps_{r_1} \cap Ng_{r_2}$, *then* $Hd_{r_1} \cap Hd_{r_3} = \emptyset$
   - 3.4 *If* $p \in Ps_{r_2} \cap Ng_{r_1}$, *then* $Hd_{r_2} \cap Hd_{r_3} = \emptyset$

The following lemma is the reason why we need to consider a language with six atoms for this problem.

Lemma 7 *If there are three canonical rules r_1,r_2 and r_3 such that $\{r_1, r_2, r_3\} \simeq_{se} \{r_1, r_2\}$, but none of the three conditions in Lemma 6 hold, then there are three such rules that mention at most six atoms.*

Proof: The proof of this lemma is tedious as we have to consider several cases. Consider the following statements about any three canonical rules r_1, r_2, r_3:

(I) $\{r_1, r_2, r_3\} \simeq_{se} \{r_1, r_2\}$.

(II) $\{r_1, r_3\} \not\simeq_{se} \{r_1\}$, i.e. $Ps_{r_1} \not\subseteq Ps_{r_3}$ or $Ng_{r_1} \not\subseteq Ng_{r_3}$ or $Hd_{r_1} \cup Ng_{r_1} \not\subseteq Hd_{r_3} \cup Ng_{r_3}$

(III) $\{r_2, r_3\} \not\simeq_{se} \{r_2\}$, i.e. $Ps_{r_2} \not\subseteq Ps_{r_3}$ or $Ng_{r_2} \not\subseteq Ng_{r_3}$ or $Hd_{r_2} \cup Ng_{r_2} \not\subseteq Hd_{r_3} \cup Ng_{r_3}$

(IV) $(Ps_{r_1} \cup Ps_{r_2}) \cap (Hd_{r_1} \cup Hd_{r_2} \cup Ng_{r_1} \cup Ng_{r_2}) = \emptyset$

(V) There is an atom p in the set $(Ps_{r_1} \cup Ps_{r_2}) \cap (Hd_{r_1} \cup Hd_{r_2} \cup Ng_{r_1} \cup Ng_{r_2})$, and another different atom q such that one of the following three conditions is true:

1. $q \in Hd_{r_1} \cup Ng_{r_1}$ and $q \notin Hd_{r_3} \cup Ng_{r_3}$.
2. $q \in Ps_{r_1}$ and $q \notin Ps_{r_3}$.
3. $q \in Ng_{r_1}$ and $q \notin Ng_{r_3}$.

Notice that this is the negation of condition (3.2) in Lemma 6.

(VI) $Hd_{r_1} \cap Hd_{r_3} \not\subseteq Ng_{r_3}$, and there is an atom $p \in Ps_{r_1} \cap Ng_{r_2}$ such that for $i = 1, 2$, $Hd_{r_i} \setminus \{p\} \subseteq Hd_{r_3} \cup Ng_{r_3}$, $Ps_{r_i} \setminus \{p\} \subseteq Ps_{r_3}$, and $Ng_{r_i} \setminus \{p\} \subseteq Ng_{r_3}$.

Since r_1 and r_2 are symmetric in the conditions in Lemma 6, to prove this lemma, we need only to prove the following three assertions:

(a) If there are three canonical rules r_1, r_2, r_3 which satisfy (I)-(IV), then there are three canonical rules r'_1, r'_2, r'_3 which mention at most six atoms, and satisfy (I)-(IV) as well.

(b) If there are three canonical rules r_1, r_2, r_3 which satisfy (I)-(III)(V), then there are three canonical rules r'_1, r'_2, r'_3 which mention at most six atoms, and satisfy (I)-(III)(V) as well.

(c) If there are three canonical rules r_1, r_2, r_3 which satisfy (I)-(III)(VI), then there are three canonical rules r'_1, r'_2, r'_3 which mention at most six atoms, and satisfy (I)-(III)(VI) as well.

We now prove the above three assertions one by one.

(a) Let a_1, a_2 be two atoms that make (II) and (III) true. If $(Ps_{r_3} \cap (Ps_{r_1} \cup Ps_{r_2})) \setminus \{a_1, a_2\}$ is not empty, let a_3 be an atom in it. If $Ps_{r_3} \setminus (Ps_{r_1} \cup Ps_{r_2} \cup \{a_1, a_2\})$ is not empty, let a_4 be an atom in it. If $(Ps_{r_1} \cup Ps_{r_2}) \setminus (Ps_{r_3} \cup \{a_1, a_2\})$ is not empty, let a_5 be an atom in it. Finally let a_6 be a new atom different from a_1 to a_5, and $L' = \{a_1, a_2, a_3, a_4, a_5, a_6\}$. Let f be a function from L to L' defined as following:

$$f(a) = \begin{cases} a_1 & a = a_1 \\ a_2 & a = a_2 \\ a_3 & a \in (Ps_{r_3} \cap (Ps_{r_1} \cup Ps_{r_2})) \setminus \{a_1, a_2\} \\ a_4 & a \in Ps_{r_3} \setminus (Ps_{r_1} \cup Ps_{r_2} \cup \{a_1, a_2\}) \\ a_5 & a \in (Ps_{r_1} \cup Ps_{r_2}) \setminus (Ps_{r_3} \cup \{a_1, a_2\}) \\ a_6 & otherwise \end{cases}$$

For each $1 \leq i \leq 3$, let r'_i be as follows:

$$Ps_{r'_i} = Ps_{f(r_i)}, Ng_{r'_i} = Ng_{f(r_i)}, Hd_{r'_i} = Hd_{f(r_i)} \setminus Ng_{f(r_i)}. \tag{12}$$

We have that

- For each $1 \leq i \leq 3$, r'_i is a canonical rule, and $r'_i \simeq_{se} f(r_i)$. For this, we only need to show $f(r_i) \not\simeq_{se} \emptyset$ for each $1 \leq i \leq 3$. To see this, notice that from the definition of f, atoms other than a_1 and a_2 in Ps_{r_3} are mapped to $\{a_3, a_4\}$, and atoms other than a_1 and a_2 in $Hd_{r_3} \cup Ng_{r_3}$ are mapped to $\{a_5, a_6\}$. Thus $Ps_{f(r_3)} \cap (Hd_{f(r_3)} \cup Ng_{f(r_3)}) = \emptyset$. By Theorem 5, $f(r_3) \not\simeq_{se} \emptyset$. Now $f(r_1) \not\simeq_{se} \emptyset$ and $f(r_2) \not\simeq_{se} \emptyset$, because (II) and (III) hold for $f(r_1), f(r_2), f(r_3)$ by definition of f.
- (I) holds for r'_1, r'_2, r'_3. This is because by Theorem 4,

  $$\{f(r_1), f(r_2), f(r_3)\} \not\simeq_{se} \{f(r_1), f(r_2)\},$$

  and for each $1 \leq i \leq 3$, $r'_i \simeq_{se} f(r_i)$.
- (II) and (III) hold for r'_1, r'_2, r'_3. As we mentioned, from the definition of f, (II) and (III) hold for $f(r_1), f(r_2), f(r_3)$.
- (IV) holds for r'_1, r'_2, r'_3. Again, we need only to show that (IV) holds for $f(r_1), f(r_2), f(r_3)$. To see this, notice that atoms other than a_1 and a_2 in $Ps_{r_1} \cup Ps_{r_2}$ are mapped to $\{a_3, a_5\}$, and atoms other than a_1 and a_2 in $Hd_{r_1} \cup Hd_{r_2} \cup Ng_{r_1} \cup Ng_{r_2}$ are mapped to $\{a_4, a_6\}$.

(b) Again let a_1, a_2 be two atoms that make (II) and (III) true. Let p, q be the two witness atoms in (V). If $Pos(r_3) \setminus \{a_1, a_2, p, q\}$ is not empty, let a_3 be an atom in it. Let a_4 be a new atom, and $L' = \{a_1, a_2, a_3, a_4, p, q\}$. Define f as follows:

$$f(a) = \begin{cases} a_1 & a = a_1 \\ a_2 & a = a_2 \\ p & a = p \\ q & a = q \\ a_3 & a \in Ps_{r_3} \setminus \{a_1, a_2, p, q\} \\ a_4 & otherwise \end{cases}$$

Define r'_i by (12) as well for each $1 \leq i \leq 3$.

- For each $1 \leq i \leq 3$, r'_i is a canonical rule, and $r'_i \simeq_{se} f(r_i)$. This can be seen in the same way as for (a) above.
- By Theorem 4, $\{f(r_1), f(r_2), f(r_3)\} \simeq_{se} \{f(r_1), f(r_2)\}$, and thus

  $$\{r'_1, r'_2, r'_3\} \simeq_{se} \{r'_1, r'_2\}.$$

  So (I) holds for r'_1, r'_2, r'_3.
- From definition of f, (II) and (III) hold for $f(r_1), f(r_2), f(r_3)$, and thus they hold for r'_1, r'_2, r'_3 as well.
- Again from the definition of f, (V) holds for $f(r_1), f(r_2), f(r_3)$: there is an atom p in the set $(Ps_{f(r_1)} \cup Ps_{f(r_2)}) \cap (Hd_{f(r_1)} \cup Hd_{f(r_2)} \cup Ng_{f(r_1)} \cup Ng_{f(r_2)})$, and another different atom q such that one of the following three conditions is true:
  1. $q \in Hd_{f(r_1)} \cup Ng_{f(r_1)}$ and $q \notin Hd_{f(r_3)} \cup Ng_{f(r_3)}$.
  2. $q \in Ps_{f(r_1)}$ and $q \notin Ps_{f(r_3)}$.
  3. $q \in Ng_{f(r_1)}$ and $q \notin Ng_{f(r_3)}$.

  (V) holds for r'_1, r'_2, r'_3 as well because for each $1 \leq i \leq 3$,

  $$Ps_{r'_i} = Ps_{f(r_i)}, Ng_{r'_i} = Ng_{f(r_i)}, Hd_{r'_i} \cup Ng_{r'_i} = Hd_{f(r_i)} \cup Ng_{f(r_i)}.$$

(c) Let a_1, a_2 be two atoms that make (II) and (III) true. Let p be the witness atom in (VI), and let $q \in Hd_{r_1} \cap Hd_{r_3}$ but $q \notin Ng_{r_3}$. If $Pos(r_3) \setminus \{a_1, a_2, p, q\}$ is not empty, let a_3 be an atom in it. Let a_4 is a new atom, and Let $L' = \{a_1, a_2, a_3, a_4, p, q\}$, Define f as follows:

$$f(a) = \begin{cases} a_1 & a = a_1 \\ a_2 & a = a_2 \\ p & a = p \\ q & a = q \\ a_3 & a \in Ps_{r_3} \setminus \{a_1, a_2, p, q\} \\ a_4 & otherwise \end{cases}$$

Again define r'_i by (12) as well for each $1 \leq i \leq 3$.

- For each $1 \leq i \leq 3$, r'_i is a canonical rule, and $r'_i \simeq_{se} f(r_i)$. This can be seen in the same way as for (a) above.

– Again by Theorem 4, $\{f(r_1), f(r_2), f(r_3)\} \simeq_{se} \{f(r_1), f(r_2)\}$, and thus

$$\{r'_1, r'_2, r'_3\} \simeq_{se} \{r'_1, r'_2\}.$$

So (I) holds for r'_1, r'_2, r'_3.

– Again from definition of f, (II) and (III) hold for $f(r_1), f(r_2), f(r_3)$, thus they hold for r'_1, r'_2, r'_3 as well.

– By the definition of f, (VI) holds for $f(r_1), f(r_2), f(r_3)$: $Hd_{f(r_1)} \cap Hd_{f(r_3)} \not\subseteq Ng_{f(r_3)}$, and there is an atom $p \in Ps_{f(r_1)} \cap Ng_{f(r_2)}$ such that for $i = 1, 2$, $Hd_{f(r_i)} \setminus \{p\} \subseteq Hd_{f(r_3)} \cup Ng_{f(r_3)}$, $Ps_{f(r_i)} \setminus \{p\} \subseteq Ps_{f(r_3)}$, and $Ng_{f(r_i)} \setminus \{p\} \subseteq Ng_{f(r_3)}$. (VI) holds for r'_1, r'_2, r'_3 as well because

$$Ps_{r'_i} = Ps_{f(r_i)}, Ng_{r'_i} = Ng_{f(r_i)}, Hd_{r'_i} \subseteq Hd_{f(r_i)}.$$

□

Theorem 8 (The 2-1-0 problem) *Lemma 6 holds in the general case, without any restriction on the number of atoms in r_1, r_2, r_3.*

Proof: The assertion that r_1, r_2, and r_3 are canonical rules and satisfy one of the three conditions in Lemma 6 is equivalent to the following property

$$\begin{aligned}
&[\forall x.((\neg(X_1(x) \wedge Y_1(x))) \wedge (\neg(X_1(x) \wedge Z_1(x))) \wedge (\neg(Y_1(x) \wedge Z_1(x))))] \wedge \\
&[\forall x.((\neg(X_2(x) \wedge Y_2(x))) \wedge (\neg(X_2(x) \wedge Z_2(x))) \wedge (\neg(Y_2(x) \wedge Z_2(x))))] \wedge \\
&[\forall x.((\neg(X_3(x) \wedge Y_3(x))) \wedge (\neg(X_3(x) \wedge Z_3(x))) \wedge (\neg(Y_3(x) \wedge Z_3(x))))] \wedge \\
&\{[(\forall x.Y_1(x) \supset Y_3(x)) \wedge (\forall x.Z_1(x) \supset Z_3(x)) \wedge (\forall x.X_1(x) \supset (X_3(x) \vee Z_3(x)))] \vee \\
&\quad [(\forall x.Y_2(x) \supset Y_3(x)) \wedge (\forall x.Z_2(x) \supset Z_3(x)) \wedge (\forall x.X_2(x) \supset (X_3(x) \vee Z_3(x)))] \vee \\
&\quad [\exists x.CON1(x) \wedge CON2(x) \wedge CON3(x) \wedge CON4(x)]\}
\end{aligned}$$

being true on $[r_1, r_2, r_3]$, where $CON1(x)$ stands for

$$(Y_1(x) \vee Y_2(x)) \wedge (X_1(x) \vee X_2(x) \vee Z_1(x) \vee Z_2(x))$$

$CON2(x)$ for

$$\begin{aligned}
\forall y.(x \neq y) \quad \supset \quad & [(X_1(y) \supset (X_3(y) \vee Z_3(y))) \wedge (Y_1(y) \supset Y_3(y)) \wedge (Z_1(y) \supset Z_3(y)) \wedge \\
& (X_2(y) \supset (X_3(y) \vee Z_3(y))) \wedge (Y_2(y) \supset Y_3(y)) \wedge (Z_2(y) \supset Z_3(y))]
\end{aligned}$$

$CON3(x)$ for

$$Y_1(x) \wedge Z_2(x) \supset \forall y.(\neg(X_1(y) \wedge X_3(y))),$$

and $CON4(x)$ for

$$Y_2(x) \wedge Z_1(x) \supset \forall y.(\neg(X_2(y) \wedge X_3(y))).$$

Thus the "if" part follows from Theorem 3 and Lemma 6, by noticing that the above property can be written as $\exists x \forall \vec{y}.Q$ as required by Theorem 3. The "only if" part follows from Lemma 6 and Lemma 7. □

The conditions in Lemma 6 (Theorem 8) are rather complex, and the reason why it is difficult to automate Step 3 of the procedure at the beginning of the section. These conditions capture all possible cases when r_3 is "subsumed" by r_1 and r_2, and are difficult to describe concisely by words. We give some examples.

Consider the following three rules:

$$r_1 : (a_2 \leftarrow a_1)$$
$$r_2 : (a_3 \leftarrow not\, a_1)$$
$$r_3 : (a_3 \leftarrow not\, a_2).$$

We have that $\{r_1, r_2, r_3\} \simeq_{se} \{r_1, r_2\}$ because the condition (4) in Lemma 6 holds.

However, if we change r_3 into $r_3' : \ a_2 \leftarrow not\, a_3$, then $P_1 = \{r_1, r_2, r_3'\}$ and $P_2 = \{r_1, r_2\}$ are not strongly equivalent: one could check that condition (4.3) in Lemma 6 does not hold, and indeed, while $P_2 \cup \{a_1 \leftarrow a_2\}$ has a unique answer set $\{a_3\}$, $P_1 \cup \{a_1 \leftarrow a_2\}$ has two answer sets $\{a_3\}$ and $\{a_1, a_2\}$.

It is also easy to show by Theorem 8 that $a_3 \leftarrow not\, a_2$ is "subsumed" by

$$\{(a_1; a_2; a_3 \leftarrow), (a_2; a_3 \leftarrow a_1)\},$$

and $a_2; a_3 \leftarrow$ is "subsumed" by

$$\{(a_2 \leftarrow a_1), (a_3 \leftarrow not\, a_1)\}.$$

With the results that we have, the following theorem will yield a solution to the 0-2-1 problem.

Theorem 9 (the 0-2-1 problem) *For any three rules r_1, r_2 and r_3, $\{r_1, r_2\}$ and $\{r_3\}$ are strongly equivalent iff the following three conditions are true:*

1. $\{r_1, r_2, r_3\} \simeq_{se} \{r_1, r_2\}$.
2. $\{r_1, r_3\} \simeq_{se} \{r_3\}$.
3. $\{r_2, r_3\} \simeq_{se} \{r_3\}$.

For example, We have

$$\{(a_2 \leftarrow a_1, not\, a_3), (a_1; a_2 \leftarrow not\, a_3)\} \simeq_{se} \{a_2 \leftarrow not\, a_3\}.$$

While we have

$$\{(\leftarrow a_2, a_3), (\leftarrow a_3, not\, a_2)\} \simeq_{se} \{\leftarrow a_3\},$$

we have

$$\{(a_1 \leftarrow a_2, a_3), (a_1 \leftarrow a_3, not\, a_2)\} \not\simeq_{se} \{a_1 \leftarrow a_3\}.$$

Similarly, we have the following theorem

Theorem 10 (the 0-2-2 problem) *For any four rules r_1, r_2, r_3, r_4, $\{r_1, r_2\}$ and $\{r_3, r_4\}$ are strongly equivalent iff the following four conditions are true:*

1. $\{r_1, r_2, r_3\} \simeq_{se} \{r_1, r_2\}$.
2. $\{r_1, r_2, r_4\} \simeq_{se} \{r_1, r_2\}$.
3. $\{r_3, r_4, r_1\} \simeq_{se} \{r_3, r_4\}$.
4. $\{r_3, r_4, r_2\} \simeq_{se} \{r_3, r_4\}$.

6. Program Simplification

We have mentioned that one possible use of the notion of strongly equivalent logic programs is in simplifying logic programs: if $P \simeq_{se} Q$, and that Q is "simpler" than P, we can then replace P in any program that contains it by Q.

Most answer set programming systems perform some program simplifications. However, only Smodels (Niemelä et al., 2000) has a stand-alone front-end called lparse that can be used to ground and simplify a given logic program. It seems that lparse simplifies a grounded logic program by computing first its well-founded model. It does not, however, perform any program simplification using the notion of strong equivalence. For instance, lparse-1.0.13, the current version of lparse, did nothing to the following set of rules:
$\{(a \leftarrow not\, b), (b \leftarrow not\, a), (a \leftarrow a)\}$. Nor does it replace the first rule in the following program $\{(a \leftarrow not\, a), (a \leftarrow not\, b), (b \leftarrow not\, a)\}$ by the constraint $\leftarrow not\, a$.

It is unlikely that anyone would be intentionally writing rules like $a \leftarrow a$ or $b \leftarrow a, not\, a$. But these type of rules can arise as a result of grounding some rules with variables. For instance, the following is a typical recursive rule used in logic programming encoding of the Hamiltonian Circuit problem (Niemelä, 1999; Marek & Truszczynski, 1999):

$$reached(X) \leftarrow arc(Y, X), hc(Y, X), reached(Y).$$

When instantiated on a graph with cyclic arcs like $arc(a, a)$, this rule generates cyclic rules of the form $reached(X) \leftarrow hc(X, X), reached(X)$. Unless deleted explicitly, these rules will slow down many systems, especially those based on SAT. For instance, none of the graphs tested using ASSAT have self-cycles consisting of an arc from a node to itself (Lin & Zhao, 2004). If these cycles are included, ASSAT would run significantly longer.

It is thus useful to consider using the results that we have here for program simplification. Indeed, transformation rules such as deleting those that contain common elements in their heads and positive bodies have been proposed (Brass & Dix, 1999), and studied from the perspective of strong equivalence (Osorio et al., 2001; Eiter et al., 2004). Our results add new such transformation rules. For instance, by Theorem 7, we can delete those elements in the head of a rule that also appear in the negation-as-failure part of the rule. Theorems 6, 8, and 9 can also be used to define some new transformation rules.

7. Concluding Remarks and Future Work

Donald Knuth, in his Forward to (Petkovsek et al., 1996), said

> "Science is what we understand well enough to explain to a computer. Art is everything else we do. ...Science advances whenever an Art becomes a Science.

> And the state of the Art advances too, because people always leap into new territory once they have understood more about the old."

We hope that with this work, we are one step closer to making discovering classes of strongly equivalent logic programs a Science.

We have mentioned that the methodology used in this paper is similar to that in (Lin, 2004). In both cases, plausible conjectures are generated by testing them in domains of small sizes, and general theorems are proved to aid the verification of these conjectures in the general case. However, while plausible conjectures are generated automatically in (Lin, 2004), they are done manually here. While the verifications of most conjectures in (Lin, 2004) are done automatically as well, they are done only semi-automatically here. Overcoming these two weaknesses is the focus of our future work. Specifically, we would like to make Step 3 of the procedure in Section 5 automatic, and prove a theorem similar to Theorem 3 to automate the proofs of the "only if" parts of theorems like Theorems 5 - 8, in the same way that Theorem 3 makes the proofs of the "if" parts of these theorems automatic. This way, we would be able to discover more interesting theorems in this area, and more easily!

Acknowledgments

An extended abstract of this paper appeared in *Proceedings of IJCAI'2005*. We thank Yan Zhang for his comments on an earlier version of the paper. We also thank the anonymous reviewers for their useful comments, especially one of them for pointing out an error in Lemma 4 in an earlier version of the paper. This work was supported in part by the Research Grants Council of Hong Kong under Competitive Earmarked Research Grant HKUST6170/04E. Part of the second author's work was done when he was a student at Sun Yat-Sen University, Guangzhou, China, and a visiting scholar in Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong.

References

Brass, S., & Dix, J. (1999). Semantics of (disjunctive) logic programs based on partial evaluation. *Journal of Logic Programming*, *40*(1), 1–46.

Chen, Y., Lin, F., & Li, L. (2005). SELP - a system for studying strong equivalence between logic programs. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning(LPNMR 2005)*, pp. 442–446.

Eiter, T., Fink, M., Tompits, H., & Woltran, S. (2004). Simplifying logic programs under uniform and strong equivalence. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning(LPNMR 2004)*, pp. 87–99.

Erdem, E., Lifschitz, V., Nakhleh, L., & Ringe, D. (2003). Reconstructing the evolutionary history of indo-european languages using answer set programming. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages(PADL 2003)*, pp. 160–176.

Gelfond, M., & Lifschitz, V. (1988). The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming(ICLP/SLP)*, pp. 1070–1080.

Gelfond, M., & Lifschitz, V. (1991). Classical negation in logic programs and disjunctive databases. *New Generation Computing*, *9*(3/4), 365–386.

Inoue, K., & Sakama, C. (2004). Equivalence of logic programs under updates. In *Proceedings of the 9th European Conference on Logics in Artificial Intelligence(JELIA)*, pp. 174–186.

Lenat, D. B. (1979). On automated scientific theory formation: A case study using the AM program. In *Machine Intelligence 9*, pp. 251–283. Jean Hayes, Donald Michie, and L. I. Mikulich, eds. Ellis Horwood.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., & Scarcello, F. (2006). The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, *7*(3).

Lierler, Y., & Maratea, M. (2004). Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning(LPNMR 2004)*, pp. 346–350.

Lifschitz, V. (1999). Action languages, answer sets and planning. In *The Logic Programming Paradigm: A 25-Year Perspective*. K.R. Apt, V.W. Marek, M. Truszczynski, D.S. Warren, eds, Springer-Verlag.

Lifschitz, V., Pearce, D., & Valverde, A. (2001). Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, *2*(4), 526–541.

Lin, F. (2002). Reducing strong equivalence of logic programs to entailment in classical propositional logic. In *Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning(KR2002)*, pp. 170–176.

Lin, F. (2004). Discovering state invariants. In *Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning(KR2004)*, pp. 536–544.

Lin, F., & Zhao, Y. (2004). ASSAT: computing answer sets of a logic program by sat solvers. *Artificial Intelligence*, *157*(1-2), 115–137.

Marek, V. W., & Truszczynski, M. (1999). Stable logic programming - an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*. K.R. Apt, V.W. Marek, M. Truszczynski, D.S. Warren, eds, Springer-Verlag.

Niemelä, I., Simons, P., & Syrjänen, T. (2000). Smodels: a system for answer set programming. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*. Breckenridge, Colorado, USA. (CoRR: arXiv:cs.AI/0003033) http://www.tcs.hut.fi/Software/smodels/.

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

Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., & Barry, M. (2001). An A-Prolog decision support system for the space shuttle. In *Proceedings of the 3rd International Symposium on Practical Aspects of Declarative Languages(PADL 2001)*, pp. 169–183.

Osorio, M., Navarro, J. A., & Arrazola, J. (2001). Equivalence in answer set programming. In *Selected Papers of the 11th International Workshop on Logic Based Program Synthesis and Transformation(LOPSTR 2001)*, pp. 57–75.

Pearce, D., Tompits, H., & Woltran, S. (2001). Encodings for equilibrium logic and logic programs with nested expressions. In *Proceedings of 10th Portuguese Conference on Artificial Intelligence(EPIA 2001)*, pp. 306–320.

Petkovsek, M., Wilf, H. S., & Zeilberger, D. (1996). $A = B$. Wellesley, Mass. : A K Peters.

Turner, H. (2001). Strong equivalence for logic programs and default theories (made easy). In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning(LPNMR 2001)*, pp. 81–92.

Turner, H. (2003). Strong equivalence made easy: nested expressions and weight constraints. *Theory and Practice of Logic Programming*, *3*(4-5), 609–622.

Wang, K., & Zhou, L. (2005). Comparisons and computation of well-founded semantics for disjunctive logic programs. *ACM Transactions on Computational Logic*, *6*(2), 295–327.

Recycling Computed Answers in Rewrite Systems for Abduction

FANGZHEN LIN
Hong Kong University of Science and Technology
and
JIA-HUAI YOU
University of Alberta

In rule-based systems, goal-oriented computations correspond naturally to the possible ways that an observation may be explained. In some applications, we need to compute explanations for a series of observations with the same domain. The question arises as to whether previously computed answers can be recycled. A "yes" answer could result in substantial savings of repeated computations. For systems based on classical logic, the answer *is* yes. For nonmonotonic systems, however, one tends to believe that the answer should be *no*, since recycling is a form of adding information. In this article, we show that computed answers can always be recycled, in a nontrivial way, for the class of rewrite procedures proposed earlier by the authors for logic programs with negation. We present some experimental results on an encoding of the logistics domain.

Categories and Subject Descriptors: I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Logic programming, nonmonotonic reasoning and brief revision*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Stable model semantics, partial stable model semantics, rewrite systems, recycling, logic programming, abduction

ACM Reference Format:

Lin, F. and You, J.-H. 2007. Recycling computed answers in rewrite systems for abduction. ACM Trans. Comput. Logic 8, 2, Article 9 (Apr. 2007), 24 pages. DOI = 10.1145/1227839.1227841 http://doi.acm.org/ 10.1145/1227839.1227841

An extended abstract of parts of this article appeared in *Proceedings of the Iiternational Joint Conference on Artificial Intelligence 2003* (Acapulco, Mexico).

F. Lin was supported in part by the Research Grants Council of Hong Kong under Competitive Earmarked Research Grant HKUST6205/02E.

Authors' addresses: F. Lin, Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong; email: flin@cs.ust.hk; J.-H. You (contact author), Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2E8; email: you@cs.ualberta.ca.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2007 ACM 1529-3785/2007/04-ART9 $5.00 DOI 10.1145/1227839.1227841 http://doi.acm.org/ 10.1145/1227839.1227841

1. INTRODUCTION

The question we shall address in this article is the following. With a sound and complete procedure for abduction, suppose we have computed explanations (conveniently represented as a disjunction) $Es = E_1 \vee \ldots \vee E_n$ for observation q. Suppose also that in the course of computing explanations for another observation p, we run into q again. Now, we may use the proofs Es for q without actually proving q again. The question is this: Will the use (recycling) of the proofs Es for q in the proof for p preserve the soundness and completeness of the procedure?

In this article, we answer this question positively, but in a nontrivial way, for the class of rewrite procedures proposed in Lin and You [2001] for abduction in logic programming under the (partial) stable model semantics [Gelfond and Lifschitz 1988], [Przymusinski 1990]. The main result is a theorem (Theorem 4.7) which says that recycling preserves the soundness and completeness.

The general idea of recycling is not new. Recycling in systems based on classical logic is always possible, since inferences in these systems can be viewed as transforming a logic theory to a logically equivalent one. In dynamic programming, it is the use of answers for previously computed subgoals that reduces the computational complexity. In some game playing programs, for example, in the World Champion checker program *Shinook* (www.cs.ualberta.ca/~chinook), the endgame database stores the computed results for endgame situations which can be efficiently referenced in real time.

However, the problem of recycling in a nonmonotonic proof system has rarely been investigated. We note that recycling is to use previous proofs. This differs from adding consequences. For example, it is known that the semantics based on answer sets or (maximal) partial stable models [Dung 1991]do not possess the *cautious nonmonotonicity* property. In other words, adding a consequence of a program could gain additional models, thus losing some consequences. The following example is due to [Dix 1991]:

$$P = \{a \leftarrow \mathsf{not}\, b. \quad b \leftarrow c, \mathsf{not}\, a. \quad c \leftarrow a.\} \tag{1}$$

Here, P has only one answer set: $\{a, c\}$. Thus, c is a consequence. When augmented with the rule $c \leftarrow$, the program gains a second answer set $\{b, c\}$ and loses a as a consequence.

Abduction in the framework of logic programming has been studied extensively, and a number of formalisms and top-down query answering procedures have been proposed [Console et al. 1991; Dung 1991, 1995; Eshghi and Kowalski 1989; Kakas and Mancarella 1990a, 1990b; Kakas et al. 1995, 2000; Lin and You 2001; Satoh and Iwayama 1991, 1992]. The class of rewrite procedures for abduction proposed in Lin and You [2001] is based on the idea of *abduction as confluent and terminating rewriting*. These are called *canonical systems* in the literature of rewrite systems [Dershowitz and Jouannaud 1990]. The confluence and termination properties guarantee that rewriting terminates at a unique normal form which is independent of the order of rewriting. Thus, each particular strategy of rewriting yields a rewrite procedure. It has been

shown in Lin and You [2001] that these rewrite procedures are sound and complete under the semantics based on partial stable models for *brave reasoning*; that is a query is true in a partial stable model if and only if there is a proof by such a rewrite procedure. Brave reasoning is particularly well suited for the task of generating explanations in abduction. In this article, we show that the soundness and completeness of these rewrite procedures can be preserved when they are extended with rules that "recycle" previous results.

This article is organized as follows. The next section defines logic program semantics. Section 3 reviews the rewriting framework. Then in Section 4 we formulate rewrite systems with computed rules and prove that recycling preserves soundness and completeness. Section 5 extends this result to rewrite systems with abduction. Recycling incurs overhead and sometimes the overhead can be substantial. A good recycling strategy makes computed answers likely to be used in later computations. In Section 6 we propose such a strategy, and Section 7 reports some experimental results. Section 8 includes extensions and related work, and Section 9 concludes.

2. LOGIC PROGRAM SEMANTICS

A rule is of the form

$$a \leftarrow b_1, \ldots, b_m, \mathsf{not}\, c_1, \ldots, \mathsf{not}\, c_n,$$

where a, b_i, and c_i are atoms of the underlying propositional language $\mathcal{L}$. Furthermore, $\mathsf{not}\, c_i$ are called *default negations*. A *literal* is an atom ϕ or its negation $\neg\phi$, while a *(normal) program* is a finite set of rules.

The *completion* of a program P, denoted $Comp(P)$, is a set of equivalences: For each atom $\phi \in \mathcal{L}$, if ϕ does not appear as the head of any rule in P, than $\phi \leftrightarrow F \in Comp(P)$; otherwise $\phi \leftrightarrow B_1 \vee \ldots \vee B_n \in Comp(P)$ (with default negations replaced by the corresponding negative literals) if there are exactly n rules $\phi \leftarrow B_i \in P$ with ϕ as the head; and these are the only equivalences in $Comp(P)$. We write T for B_i if B_i is empty.

The rewriting system of Lin and You [2001] is sound and complete with respect to the partial stable model semantics [Przymusinski 1990]. A simple way to define partial stable models without even introducing three-valued logic is by so-called *alternating fixpoints* [You and Yuan 1995]. Let P be a program and S a set of default negations. Define a function over sets S of default negations: $F_P(S) = \{\mathsf{not}\, a \mid P \cup S \nvdash a\}$. The relation $\vdash$ is the standard propositional derivation relation with each default negation $\mathsf{not}\, \phi$ being treated as a named atom $\mathsf{not_}\phi$.

A *partial stable model* M is defined by a fixpoint of the function that applies F_P twice, namely, $F_P^2(S) = S$, 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. An *answer set* E (also called *stable model*) is defined by a fixpoint S such that $F_P(S) = S$ and $E = \{\xi \in \mathcal{L} \mid P \cup S \vdash \xi\}$.

3. GOAL REWRITE SYSTEMS

In this section, we briefly review goal rewrite systems as formulated in Lin and You [2001].

A goal rewrite system is a rewrite system that consists of three types of rewrite rules: (1) program rules from $Comp(P)$ for literal rewriting; (2) simplification rules to transform and simplify goals; and (3) loop rules for handling loops.

A *program rule* is a completed definition $\phi \leftrightarrow B_1 \vee \ldots \vee B_n \in Comp(P)$ used from left-to-right: ϕ can be rewritten to $B_1 \vee \ldots \vee B_n$, and $\neg\phi$ to $\neg B_1 \wedge \ldots \wedge \neg B_n$. These are called *literal rewritings*.

A *goal*, also called a *goal formula*, is a formula which may involve $\neg$, $\vee$, and $\wedge$. A goal resulting from a literal rewriting from another goal is called a *derived goal*. Like a formula, a goal may be transformed to another goal without changing its semantics. This is carried out by simplification rules.

We assume that in all goals, negation appears only in front of a literal. This can be achieved by simple transformations like those used for converting a formula to clauses.

3.1 Simplification Rules

The simplification rules constitute a nondeterministic transformation system formulated with a mechanism of loop-handling in mind, which requires keeping track of literal sequences $g_0, \ldots, g_n$ where each g_i, $0 < i \leq n$ is in the goal formula resulting from rewriting g_{i-1}. Two central mechanisms in formalizing goal rewrite systems are *rewrite chains* and *contexts*.

—*Rewrite Chains:* 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 $l_1 \prec^+ l_n$.

—*Context:* A rewrite chain $g = g_0 \prec g_1 \prec \ldots \prec g_n = T$ records a set of literals $C = \{g_0, \ldots, g_{n-1}\}$ for proving g. We will write $T(\{g_0, \ldots, g_{n-1}\})$ and call C a *context*. A context will also be used to maintain consistency: If g can be proved via a conjunction, all of the conjuncts need be proved with contexts that are nonconflicting. 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 any literal in a 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 has the definitions: $a \leftrightarrow \neg b \wedge \neg c$ and $b \leftrightarrow q \vee \neg p$. Then, we get 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 from a:

$$a \prec \neg b \prec \neg q$$
$$a \prec \neg b \prec p$$
$$a \prec \neg c$$

Rules (Simplification). Let Φ and Φ_i be goal formulas, C be 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. $\Phi_1 \wedge (\Phi_2 \vee \Phi_3) \rightarrow (\Phi_1 \wedge \Phi_2) \vee (\Phi_1 \wedge \Phi_3)$
SR5'. $(\Phi_1 \vee \Phi_2) \wedge \Phi_3 \rightarrow (\Phi_1 \wedge \Phi_3) \vee (\Phi_2 \wedge \Phi_3)$

For any goal formula, repeated applications of SR5 and SR5' transform it to a disjunctive normal form (DNF).

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. Two rewrite rules are formulated to handle loops.

Definition 3.1. 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 preceding cases, l_n is called a *loop literal*.

Rules (Loop). 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. $g_n \rightarrow T(\{g_1, \ldots, g_n\})$:
  If $g_i \prec^+ g_n$, for some $1 \leq i < n$, is a negative loop or an even loop.

3.3 Goal Rewrite Systems

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, and for each $0 \leq i < k$, Q_{i+1} is obtained from Q_i by

—literal rewriting at a nonloop literal in Q_i,
—applying a simplification rule to a subformula of Q_i, or
—applying a loop rule to a loop literal in Q_i.

Definition 3.2. A *goal rewrite system* for a program P 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 loop rules, and $\rightarrow$ is the set of all rewrite sequences.

In Lin and You [2001], it is shown that goal rewrite systems are sound and complete with respect to the partial stable model semantics.

THEOREM 3.3. *Let P be a normal program and $\langle \mathcal{Q}_L, \mathcal{R}_P, \rightarrow \rangle$ a goal rewrite system.*

Soundness: *For any literal g and any rewrite sequence $g \rightarrow^* T(C_1) \vee \ldots \vee T(C_m)$, there exists a partial stable model M_i of P for each $i \in [1..m]$ such that $g \in C_i \subseteq M_i$.*

Completeness: *For any literal g which is true in a partial stable model M of P, there exists a rewrite sequence $g \rightarrow^* T(C_1) \vee \ldots \vee T(C_m)$ such that for some $i \in [1..m]$, $g \in C_i \subseteq M$.*

4. GOAL REWRITE SYSTEMS WITH COMPUTED RULES

We now consider how to recycle computation in these rewrite systems. Notice that what we eventually want is a theory of recycling in abductive goal rewrite systems. However, it is best to first study the basic case where there are no abducibles. As we shall see, just as abducible rewrite systems are a direct extension of rewrite systems, as we showed in Lin and You [2001], recycling in abducible rewrite systems is a direct extension of recycling in rewrite systems.

We first illustrate recycling with two examples.

Example 4.1. Given a rewrite system R^0, suppose, we have a rewrite sequence $\neg q \rightarrow a \rightarrow a \rightarrow F$. The failure is due to a positive loop on a. We may recycle the computed answer by replacing the rewrite rule for $\neg q$ by the new rule $\neg q \rightarrow F$. We thus get a new system, say R^1. Suppose that in trying to prove g we have

$$g \rightarrow a \rightarrow \neg q \rightarrow F,$$

where the last step makes use of the computed answer for $\neg q$. The question arises as to whether this way of using previously computed results guarantees, soundness and completeness. Theorem 4.7, to be proved later in this article answers this question positively. To see it for this example, assume that we have the following successful proof in R^0:

$$g \rightarrow a \rightarrow \neg q \rightarrow a \rightarrow T(\{g, a, \neg q\}),$$

where the termination is due the even loop on a. Had such a sequence existed, recycling would have produced a wrong result. However, we can see that the existence of the rewrite step $a \rightarrow \neg q$ implies the existence of a different means to prove $\neg q$:

$$\neg q \rightarrow a \rightarrow \neg q \vee \ldots \rightarrow T(\{\neg q, a\}) \vee \ldots$$

contradicting that $\neg q$ was rewritten to F in R^0.

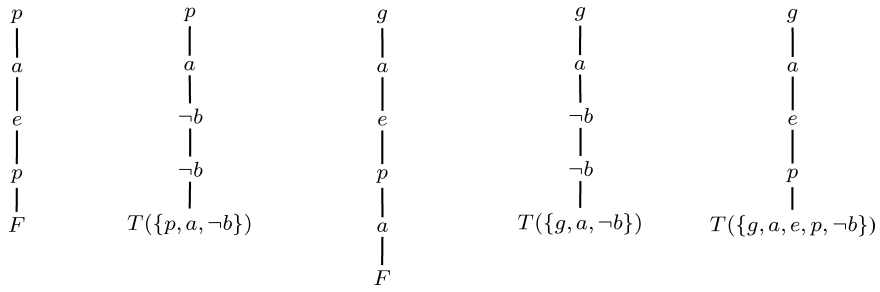


Fig. 1. Recycling may generate extra proofs.

Before giving the next example, we introduce a different way to understand rewrite sequences. Since any goal formula can always be transformed to a DNF using the distributive rules SR5 and SR5', and the order of rewriting does not matter, we can view rewriting as generating a sequence of DNFs. Thus, a rewrite sequence in DNF from an initial goal g

$$g \rightarrow^* N_1 \vee \ldots \vee N_n$$

can be conveniently represented by *derivation trees*, or *d-trees*, one for each N_i representing a possible way to prove g. For any i, the d-tree for N_i has g as its root node, wherein a branch from g to a leaf node corresponds to a rewrite chain from g that eventually ends with an F or some $T(C)$. As N_i is a conjunction, a successful proof requires each branch to succeed (i.e., end with $T(C)$ for some C) and the union of all resulting contexts to be consistent.

The next example is carefully constructed to illustrate that recycling may not yield the same answers as than no recycling the case where carried out. In particular, had been we can sometimes get additional answers.

Example 4.2. Consider the program

$$g \leftarrow a. \quad a \leftarrow \mathsf{not}\, b. \quad a \leftarrow e.$$
$$b \leftarrow b. \quad e \leftarrow p. \qquad p \leftarrow a.$$

In Figure 1 each d-tree consists of a single branch. The left two d-trees are expanded from goal p, corresponding to the following rewrite sequence:

$$p \rightarrow a \rightarrow e \vee \neg b \rightarrow p \vee \neg b \rightarrow F \vee \neg b \rightarrow \neg b \rightarrow \neg b \rightarrow T(\{p, a, \neg b\})$$

The next two d-trees are generated for goal g corresponding to the rewrite sequence

$$g \rightarrow a \rightarrow e \vee \neg b \rightarrow p \vee \neg b \rightarrow a \vee \neg b \rightarrow F \vee \neg b \rightarrow \neg b \rightarrow \neg b \rightarrow T(\{g, a, \neg b\}).$$

Now, we recycle the proof for p in the proof for g and compare it with the one without recycling. Clearly, the successful d-tree for g (the fourth from the left) will still succeed, as it doesn't involve any p. The focus is then on the d-tree in the middle, in particular, the node p within it; this d-tree fails when no recycling has been performed.

Since p is previously proved with context $\{g, a, \neg b\}$, recycling this proof amounts to terminating p with a context which is the union of this context with the rewrite chain leading to p (see the d-tree on the right). But this results in a successful proof that fails without recycling.

Though recycling appears to have generated a wrong result, we can verify that both generated contexts $\{g, a, \neg b\}$ and $\{g, a, e, p, \neg b\}$ belong to the same partial stable model. Thus, recycling in this example didn't lead to an incorrect answer but generated a redundant one. Theorem 4.7 shows that this is not incidental. Indeed, if p is true in a partial stable model, by derivation (look at the d-tree in the middle) so must be e, a, and g.

4.1 Rewrite Systems with Computed Rules

Given a goal rewrite system R, we may denote a rewrite sequence from a literal g by $g \rightarrow_R E$.

Definition 4.3 (*Computed Rule*). Let R be a goal rewrite system in which literal p is rewritten to its normal form. The *computed rule for* p is defined as: If $p \rightarrow_R F$, the computed rule for p is the rewrite rule $p \rightarrow F$; if $p \rightarrow_R T(C_1) \vee \ldots \vee T(C_n)$ then the computed rule for p is the rewrite rule $p \rightarrow T(C_1) \vee \ldots \vee T(C_n)$.

For the purpose of recycling, a computed rule $p \rightarrow E$ is meant to replace the existing literal rewrite rule for p. If E is F, that is, the goal p failed, then it can be used directly as the literal rewrite rule for p. Otherwise, we must combine the contexts in E with the rewrite chain leading to p, and keep only consistent ones.

Rule (*Recycling*). Let $g_1 \prec^+ g_n$ be a rewrite chain where g_n is a nonloop literal. Let $G = \{g_1, \ldots, g_n\}$, and $g_n \rightarrow T(D_1) \vee \ldots \vee T(D_k)$ be the computed rule for g_n. Further, let $\{D'_1, \ldots, D'_{k'}\}$ be the subset of $\{D_1, \ldots, D_k\}$ containing any D_i such that $D_i \cup G$ is consistent. Then, the *recycling rule* for $g_n$is defined as

RC. $g_n \rightarrow T(G \cup D'_1) \vee \ldots \vee T(G \cup D'_{k'})$.

Example 4.4. Consider the following program:

$$g \leftarrow a. \quad a \leftarrow p. \quad p \leftarrow \text{not } a.$$
$$a \leftarrow \text{not } p. \quad p \leftarrow \text{not } b. \quad b \leftarrow \text{not } a.$$

and the proof

$$p \rightarrow \neg a \vee \neg b \rightarrow p \vee \neg b \rightarrow T(\{p, \neg a\}) \vee \neg b$$
$$\rightarrow T(\{p, \neg a\}) \vee p \rightarrow T(\{p, \neg a\}) \vee T(\{p, \neg b\}).$$

We therefore have a computed rule for p:

$$p \rightarrow T(\{p, \neg a\}) \vee T(\{p, \neg b\})$$

Now, in the course of proving g we can recycle the computed rule for p by

$$g \rightarrow a \rightarrow p \vee \neg p \rightarrow T(\{g, a, p, \neg b\}) \vee \neg p \rightarrow \ldots$$

In the sequel, a rewrite system includes the recycling rule, as well as zero or more computed rules. We note that the termination and confluence properties still hold for the extended systems.

We are interested in the soundness and completeness of a series of rewrite systems, each of which recycles the computed answers generated on the previous one. For this purpose, given a program P, we use R_P^0 to denote the original goal rewrite system where literal rewrite rules are defined by the Clark completion of P. For all $i \geq 0$, R_P^{i+1} is defined in terms of R_P^i as follows: Let Δ_i be the set of computed rules (generated) on R_P^i for the set of literals $\mathcal{L}_{\Delta_i}$. Then, R_P^{i+1} is the rewrite system obtained from R_P^i by replacing the rewrite rules for the literals in $\mathcal{L}_{\Delta_i}$ by those in Δ_i. In the rest of this section, we will always refer to a fixed program P. Thus we may drop the subscript P and write R^i.

Definition 4.5. A rewrite system R^i is *sound* iff for any literal g and rewrite sequence $g \rightarrow_{R^i} T(C_1) \vee \ldots \vee T(C_n)$, and for each C_j, $j \in [1..n]$, there exists a partial stable model M of P such that $g \in C_j \subseteq M$. R^i is *complete* iff for any literal g such that $g \in M$ for some partial stable model M of P, there is a rewrite sequence $g \rightarrow_{R^i} T(C_1) \vee \ldots \vee T(C_n)$ such that for some C_j, $j \in [1..n]$, $g \in C_j \subseteq M$.

An important property of provability by rewriting is the so-called *loop rotation*, which is needed in order to prove the soundness and completeness of recycling; Specifically, a proof (a successful branch in a d-tree) terminated by a loop rule can be captured in rotated forms.

To describe this property, we need the following notation about rewrite chains: Any direct dependency relation $l \prec l'$ may be denoted by $l \cdot l'$, and we allow a segment (which may be empty) of a rewrite chain to be denoted by a Greek letter such as δ, θ, or ξ. Thus, we may write $x \cdot \delta \cdot y$ to denote a rewrite chain from x to y via δ, and $x \cdot \delta$ to mean a rewrite chain that begins with x followed by the segment denoted by δ. A rewrite chain may also be used to denote the set of the literals on it.

LEMMA 4.6 (LOOP ROTATION). *Let R^0 be a rewrite system without computed rules. Let Tr be a d-tree for literal g that succeeds with context C. Suppose a branch of Tr ends with a loop, $g \cdot \theta \cdot g$, for some θ. Then, for any literal $l \in \theta$, there is a proof of l that succeeds with the same context C.*

PROOF. A loop $\pi = g \cdot l_1 \cdot l_2 \cdot \cdots \cdot l_n \cdot g$, where g and l_i are literals, can always be rotated as

$$\begin{array}{l} l_1 \cdot l_2 \cdot \cdots \cdot l_n \cdot g \cdot l_1, \\ l_2 \cdot \cdots \cdot l_n \cdot g \cdot l_1 \cdot l_2, \\ \cdots\cdots \end{array}$$

and so on, so that if π is a negative loop (or an even loop, respectively) so is its rotated loop. Rotation over a d-tree can be performed as follows: Remove the top node n, and for any link from the top node $n \cdot q$, attach the link $n \cdot q$ to any occurrence of n. The assumption of the existence of loop $g \cdot \theta \cdot g$ ensures that in every round of rotation there is at least one occurrence of the top node (see Figure 2 for an illustration where rotation proceeds from left-to-right). It can

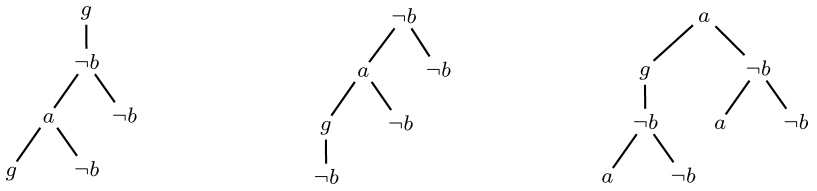


Fig. 2. Loop rotation.

be seen that the type of a loop is always preserved and the set of literals on the tree remains unchanged. □

4.2 Soundness and Completeness of Recycling

To follow, given a literal l, by *a proof of* l we mean a rewrite sequence from l to $T(C_1) \vee \ldots \vee T(C_n)$, where any C_i can be referred to as a proof of l.

THEOREM 4.7. *For any $i \geq 0$, R^i is sound and complete.*

PROOF. We prove the claim by induction on i. Specifically, R^0, the system without computed rules, is sound and complete [Lin and You 2001]. Now assume for all j with $0 \leq j \leq i$ that all R^j are sound and complete, and show that R^{i+1} is also sound and complete.

We only need to consider the situations where the rewriting in R^{i+1} differs from that of R^i. Let $\mathcal{L}_{\Delta_i}$ be the set of literals whose computed rules are generated on R^i. We can first carry out rewriting without rewriting the literals that are in $\mathcal{L}_{\Delta_i}$. In this case, rewriting from g in both R^i and R^{i+1} terminates at the same expression, which is either F or a DNF, say $N_1 \vee \ldots \vee N_m$. Each N_i can be representedby a d-tree.

—*Soundness:* Suppose $g \rightarrow_{R^{i+1}} T(D_1) \vee \ldots \vee T(D_s)$. For any $D \in \{D_1, \ldots, D_s\}$ we need to show that there is a partial stable model M such that $D \subseteq M$. Consider the d-tree Tr that generates D and suppose g is its root node. We show inductively in a bottom-up fashion that all the literals on Tr must be in the same partial stable model. For any leaf node p that is terminated by its computed rule

$$p \rightarrow \ldots \vee T(C) \vee \ldots$$

suppose Tr is the one that succeeded with context C. By the inductive hypothesis on R^j, we know that R^j is sound for all $j \leq i$, thus there is a partial stable model M such that $C \subseteq D \subseteq M$. If a leaf node q is terminated by a loop, by the Loop Rotation Lemma (Lemma 4.1), there is a proof of q in R^i using rotated loops. Otherwise we have an obvious case where a leaf node is rewritten to $True$ by its Clark completion.

In the inductive step, let $l_1, \ldots, l_n$ be the child nodes of some node l and assume that each l_i is proved in R^i, hence in some partial stable model. We first show that they belong to the same partial stable model M. Then, we show that l can also be proved in R^i, thus belonging to M as well. Without loss of generality, assume there are only two child nodes: $l_1 \rightarrow_{R^i} T(Q_1) \vee \ldots \vee T(Q_m)$, $l_2 \rightarrow_{R^i} T(W_1) \vee \ldots \vee T(W_n)$. Since D is constructed in R^{i+1} using computed

rules, by the definitions of computed and recycling rules, there are Q_i and W_j such that $Q_i \cup W_j \subseteq D$, and hence $Q_i \cup W_j$ is consistent. Then in R^i, the two contexts are merged by using simplification rule SR3, namely

$$l_1 \wedge l_2 \rightarrow_{R^i} \ldots \vee [T(Q_i) \wedge T(W_j)] \vee \ldots \rightarrow_{R^i} \ldots \vee T(Q_i \cup W_j) \vee \ldots.$$

Since R^i is sound, there is a partial stable model M such that $\{l_1, l_2\} \subseteq Q_i \cup W_j \subseteq M$. But l is derivable from l_1 and l_2. Using the definition of partial stable models, it can be shown that l must also be in M.

The induction allows us to conclude that for the top goal g and its proof D in R^{i+1}, we must have $g \in D \subseteq M$, for the same partial stable model M.

—*Completeness:* We show that for any context generated in R^i, the same context will be generated in R^{i+1}. Then, R^{i+1} is complete simply because R^i is complete.

Let $p \in \mathcal{L}_{\Delta_i}$, and consider a proof of g via p and its d-tree. Since each branch of this d-tree can be expanded and eventually terminated independent of others, for simplicity, we consider a proof of g by (an extension of) a branch $g \cdot \xi \cdot p$. In R^{i+1} the computed rule for p is used, while in R^i it is not. We only need to consider two cases of proof in R^i: either g is proved via p and a previously computed rule, or the proof is terminated due to a loop.

(i) *The case of loops.* In expanding the rewrite chain $g \cdot \xi \cdot p$ in R^i, we may form a loop, say $g \cdot \xi \cdot p \cdot \xi'$. If the loop is in ξ', exactly the same loop occurs in rewriting p as the top goal in R^i, so it is part of the computed rule for p. Otherwise it is a loop that *crosses over* p, in the general form

$$\pi = g \cdot \theta_1 \cdot l \cdot \theta_2 \cdot p \cdot \theta_3 \cdot l,$$

where l is the loop literal. As a special case of loop rotation over a branch (compare with Lemma 4.1), the same way of terminating a rewrite chain presents itself in proving p as the top goal in R^i, which is

$$\pi' = p \cdot \theta_3 \cdot l \cdot \theta_2 \cdot p.$$

If the loop on π is a negative loop (or an even loop, respectively), so is π'. Thus the same context will be generated in R^{i+1}.

(ii) *g is proved via p and a previously computed rule.* In other words, R^i gives a rewrite chain of the form $g \cdot \xi \cdot p \cdot \delta \cdot q$ where $q \rightarrow E$ is a computed rule generated on R^j for some $j < i$. Suppose that the context generated this way is C. Because of the existence of $p \cdot \delta \cdot q$, exactly the same computed rule $q \rightarrow E$ must be used in generating the computed rule for p in R^i. It can be seen that the context generated in R^{i+1} by recycling the computed answers for p (which is computed via q) is exactly the same as that which uses the computed answers for q, but not those for p. So, for any context generated this way in R^i, the same context will be generated in R^{i+1} as well. □

As given in the next corollary, if we only recycle failed proofs then exactly the same contexts will be generated.

COROLLARY 4.8. *Let R^i be a rewrite system where each computed rule is of the form $p \rightarrow F$. Let g be a literal and E be a normal form. Then, for any $i \geq 0$, $g \rightarrow_{R^0} E$ iff $g \rightarrow_{R^i} E$.*

PROOF. Let Δ be the set of literals whose rewrite rules are computed rules in R^i. By the condition in the corollary, each such computed rule is of the form $p \rightarrow F$. First, consider the process of rewriting goal g to a normal form E in which each literal in Δ is treated as a terminal node. Note that E is a DNF, or a d-tree where some of the leaf nodes are literals in Δ. Then, continue rewriting from E in R^0 where each literal in Δ is eventually rewritten to F. Let the normal form be E'. Also continue rewriting from E in R^i where each literal in Δ is replaced by F, and thus the eventual normal form is exactly the same as E'. □

5. RECYCLING IN ABDUCTIVE REWRITE SYSTEMS

Thus far we have shown that recycling can be done in rewrite systems that do not have any abducibles. As we mentioned, this is just the base case. Our aim is to do recycling in rewrite systems with abducibles. Without abducibles, the basic problem is simply that given a program P and a goal g, decide if there is a stable model that satisfies g. While the rewrite framework studied in this article may turn out to be a good means to solve this problem, currently the best approach is to use a stable model generator such as Smodels [Simons 2000] or ASSAT [Lin and Zhao 2004]. However, when there are many abducibles, and the problem is to find explanations of a goal in terms of these abducibles, then the rewrite systems discussed herein are better than answer set generators, as we discussed at the end of Section 7 in Lin and You [2001]. In the following, we first review an abductive framework as given in Lin and You [2001], and we then review how rewrite systems are extended to handle abduction as in Lin and You [2001]. Finally, we show how our results about recycling in rewrite systems can be similarly extended to recycling in abducible rewrite systems.

5.1 An Abductive Logic Programming Framework

In the following, let P be a normal logic program, and A a set of atoms called *abducibles*. We also assume that abducibles do not occur as the head of a rule. All the definitions and results in the following are from Lin and You [2001].

By a *hypothesis* α we mean a consistent set of literals over A, namely, it is not the case that p and $\neg p$ are both in α for some $p \in A$. We say that one 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 one hypothesis α is an *extension* of another β if $\beta \subseteq \alpha$, and a complete extension if it is an extension that is complete.

Definition 5.1. A complete hypothesis α is said to be an explanation of an atom q with respect to 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 5.2. 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

$$canCross \leftarrow boat, \text{not } leaking.$$
$$canCross \leftarrow boat, leaking, hasBucket.$$

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 elements can be deleted if it is to 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 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.

Computationally, it may be hard to compute minimal explanations from scratch. It is often easier to first compute a small "cover" of all explanations.

Definition 5.3. A set $\mathcal{S}$ of hypotheses is said to be a *cover* of q with respect to 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 5.4. *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.

PROPOSITION 5.5. *Let $\mathcal{S}$ be a cover of q. Then a hypothesis is a minimal explanation of q iff it is a prime implicant of $\bigvee_{\alpha\in\mathcal{S}} \alpha$.*

Finally, a remark here about the relationship between our notion of explanations and that of abductive explanations by Kakas and Mancarella [1990a]. It can be shown 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. So what is new here is a notion of minimal explanations. This is important, as the number of complete explanations can be significantly larger than the number of

minimal explanations. More importantly, minimal explanations capture a notion of "relevance" that is not there in complete explanations. More discussion about this can be found in Lin and You [2001].

5.2 Abductive Rewrite Systems

Again, all results in this subsection are from Lin and You [2001]. Firstly, as shown in Lin and You [2001], the rewriting framework can be extended to abduction in a straightforward way: The only difference in the extended framework is that we do not apply the Clark completion to abducibles. In other words, once an abducible appears in a goal, it will remain there unless eliminated by the simplification rules *SR2* or *SR2′*. In a similar way, the goal rewrite systems with computed rules in the previous section can be extended to abduction as well.

As an abducible may appear in a goal positively or negatively, we need a terminology to refer to both: An *abducible literal* is either an abducible ϕ or its negative counterpart $\neg\phi$. Just like a rewrite to T is written as $T(C)$, where C is the underlying rewrite chain (compare to Section 3.1), a rewrite to an abducible literal l will be written as $l(C)$ where C is the rewrite chain leading to, and including, l. Thus when we write $l(C)$, it is understood that C always contains l.

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. Recall that $\mathcal{Q}_L$ is the set of all goals. The following theorem from Lin and You [2001] shows that these rewrite systems are both sound and complete with respect to the partial stable models semantics.

THEOREM 5.6. *Let P be a finite program, A a set of abducibles, and $\langle \mathcal{Q}_L, \mathcal{R}_P, A, \rightarrow \rangle$ the goal rewrite system with respect to P and A.*

—Soundness: *For any literal g and any rewrite sequence*

$$g \rightarrow^* G \vee [l_1(C_1) \wedge \cdots \wedge l_k(C_k)] \vee G',$$

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 $P \cup \{l_1, \ldots, l_k\}^+$ such that $g \in \bigcup_{1 \leq j \leq k} C_i \subseteq M$.

—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*

$$g \rightarrow^* G \vee [l_1(C_1) \wedge \cdots \wedge l_k(C_k)] \vee G'$$

such that $g \in \bigcup_{1 \leq j \leq k} C_j \subseteq M$.

Notice that our soundness and completeness result is stated with respect to partial stable models, but our abductive framework in Section 5 is defined with the stable model semantics in mind. In general, a stable model is also a partial stable model, but the converse is not always true. Thus our rewrite systems may not be sound under the stable model semantics. The reason is that stable models check for global consistency, but our rewrite systems check only local consistencies. There are several ways to make the rewriting system also sound

and complete for stable models. One is that when a conjunction of abducibles

$$l_1(C_1) \wedge \cdots \wedge l_k(C_k)$$

is generated, we can check whether $C = C_1 \cup \cdots \cup C_k$ is consistent and complete. If it is, then $\{l_1, \ldots, l_k\}$ is an explanation. If it is consistent but not complete, then we can either call a stable model generator to see if C can be extended to a stable model or we can choose an atom p such that neither it nor its negation is in C, and continue the rewriting with either $p(C)$ or $\neg p(C)$, until a complete context is obtained.

There is, however, an important special class of logic programs where partial stable and stable models coincide. We say that a program has no odd loops (odd-loop-free) if there is no odd loop starting with any literal. Since goal rewrite systems are confluent, any odd loop in the program's dependency graph can replicate itself in a rewrite chain of some goal rewrite sequence. Therefore, there is no essential difference between our notion of odd-loop-free and the notion of *negative cycle free* in the literature [Fages 1994; Sato 1990].

The following theorem is proved in Lin and You [2001].

THEOREM 5.7. *Let P be a finite program, and $\langle \mathcal{Q}_L, \mathcal{R}_P, A, \rightarrow \rangle$ a goal rewrite system. Suppose that 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. 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.

5.3 Recycling in Abductive Rewrite Systems

We now show that recycling can be extended to abductive rewrite systems as well.

Definition 5.8. (*Computed Rule for Abduction*). Let R be a goal rewrite system for abduction. The *computed rule for a literal p* is defined as: If $p \rightarrow_R F$, the computed rule for p is the rewrite rule $p \rightarrow F$. If

$$p \rightarrow_R [l_{11}(C_{11}) \wedge \cdots \wedge l_{1k_1}(C_{1k_1})] \vee \ldots \vee [l_{m1}(C_{m1}) \wedge \cdots \wedge l_{mk_m}(C_{mk_m})]$$

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 the computed rule for p is the rewrite rule

$$p \rightarrow [l_{11}(C_{11}) \wedge \cdots \wedge l_{1k_1}(C_{1k_1})] \vee \ldots \vee [l_{m1}(C_{m1}) \wedge \cdots \wedge l_{mk_m}(C_{mk_m})]. \quad (2)$$

Rule (*Recycling*). Let $g_1 \prec^+ g_n \prec p$ be a rewrite chain where p is a nonloop literal. Let $G = \{g_1, \ldots, g_n, p\}$, and Eq. (2) be the computed rule for p. Then, the recycling rule for p is defined as

RC'.

$$p \rightarrow \big[l_{11}(C_{11} \cup G) \wedge \cdots \wedge l_{1k_1}(C_{1k_1} \cup G)\big] \vee \ldots \vee \big[l_{m1}(C_{m1} \cup G) \wedge \cdots \wedge l_{mk_m}(C_{mk_m} \cup G)\big].$$

Now, given an abductive rewrite system $R^0 = \langle \mathcal{Q}_L, \mathcal{R}_P, A, \rightarrow \rangle$, recursively, let R^n be the rewrite system obtained from R^{n-1} by replacing the rewrite rules for *some* of the literals by their corresponding recycling rules. Notice here that R^n is obtained from R^{n-1} nondeterministically by choosing some of the literals whose answers have been computed in R^{n-1}, and replace their rewrite rules by these computed answers. Both Theorems 5.6 and 5.7 can be extended to R^i for any $i \geq 0$.

THEOREM 5.9. *Let P be a finite program, A be a set of abducibles, and $R^0 = \langle \mathcal{Q}_L, \mathcal{R}_P, A, \rightarrow \rangle$ be the goal rewrite system with respect to P and A. For any $i > 0$, let R^i be defined as before.*

—Soundness: *For any literal g and any rewrite sequence*

$$g \rightarrow^*_{R^i} G \vee [l_1(C_1) \wedge \cdots \wedge l_k(C_k)] \vee G',$$

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 $P \cup \{l_1, \ldots, l_k\}^+$ such that $g \in \bigcup_{1 \leq j \leq k} C_i \subseteq M$.

—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*

$$g \rightarrow^*_{R^i} G \vee [l_1(C_1) \wedge \cdots \wedge l_k(C_k)] \vee G'$$

such that $g \in \bigcup_{1 \leq j \leq k} C_j \subseteq M$.

PROOF. Similar to the proof of Theorem 4.7. □

THEOREM 5.10. *Let P be a finite program, and $R^0 = \langle \mathcal{Q}_L, \mathcal{R}_P, A, \rightarrow \rangle$ a goal rewrite system. Let R^i be as defined earlier, $i > 0$. Suppose q is a proposition and*

$$q \rightarrow^*_{R^i} \big[l_{11}(C_{11}) \wedge \cdots \wedge l_{1k_1}(C_{1k_1})\big] \vee \ldots \vee \big[l_{m1}(C_{m1}) \wedge \cdots \wedge l_{mk_m}(C_{mk_m})\big]$$

is a rewrite sequence such that each l_{ij} is either T or an abd ucible literal, and $C_{i1} \cup \cdots \cup C_{ik_i}$ is consistent for each i. If P has no odd loops then

$$\big\{\{l_{11}, \cdots, l_{1k_1}\}, \cdots, \{l_{m1}, \cdots, l_{mk_m}\}\big\}$$

is a cover of q. In general, for arbitrary P we have

$$\bigvee_{\alpha \in \mathcal{S}} \alpha \supset \big[l_{11} \wedge \cdots \wedge l_{1k_1}\big] \vee \cdots \vee \big[l_{m1} \wedge \cdots \wedge l_{mk_m}\big],$$

where $\mathcal{S}$ is any cover of q.

PROOF. Just like the proof of Theorem 5.7 in Lin and You [2001]. □

6. A RECYCLING STRATEGY

We have shown that in theory, we can reuse the previously computed answers in our rewrite systems for abduction. To put the theory into practice, we need some effective strategies on how to recycle these computations.

First, we shall further motivate the need for good recycling strategies. Since a set of previously computed answers may not actually be used in answering a new goal, recycling in general does not change the complexity of deciding whether a goal is true in a partial stable model of a program. However, recycling sometimes can reduce the computation of a hard goal to trivial simply due to the fact that the significantly hard part of the current goal has already been computed and can be used directly in proving the current goal.

However, the benefit of recycling does not come free: Recycling incurs overhead, sometimes substantial overhead. In the worse case, since there can be an exponential number of (partial) stable models for a program, there can be an exponential number of explanations for an observation. Saving all of them can be expensive. Furthermore, we need to check against the list of computed literals during computation. So, it is easy to construct an example where computation becomes less efficient due to overhead. Thus, the effectiveness of recycling depends on a good strategy to ensuring that previously computed results are likely to be used. In this section we present such a strategy.

If we want to compute the abduction of all goals in a set without the framework of recycling introduced here, the only way is to compute them independently one-by-one. With the idea of recycling, we can try to recycle previously computed answers. The question is then which goals to compute first. This question arises even if we just want to compute the abduction of a single goal: Instead of computing it using the original program, it may sometimes be better if we first compute the abduction of some other goals and recycle the results.

Assuming that goals are literals, a simple strategy for deciding the order of goals to be computed is to find the dependency relations among the goals.

Definition 6.1. A literal l is said to be depending on a literal l' if the atom in l depends on the atom in l'. An atom p is said to be depending on an atom q if either q is inthe body of a rule whose head is p or inductively, there is another atom r such that p depends on r, and q is in the body of a rule whose head is r.

Suppose that l' depends on l, but l does not depend on l', written $l < l'$ to follow. It is easy to see that l will never be subgoaled to l' during rewriting, but l' could be subgoaled to l. Thus if we need to compute the abduction of both l and l', we should do this for l first.

This strategy, and as we have already mentioned in general rewrite with computed rules, is not guaranteed to be more efficient computationally:

—$l < l'$ does not imply that a proof of l' must go through l. For instance, it could be that all rewrite chains for l' terminate with failure before l is reached.

—Even if we have a computed rule for l, using it in the proof of l' does not always result in computational gain. For instance, suppose the computed rule

for l is

$$l \rightarrow_R T(C_1) \vee \cdots \vee T(C_n),$$

and a rewrite chain for l' is $l \rightarrow^* T(C) \wedge l$. If C is inconsistent with all C_i's, then this rewrite chain will terminate with F after trying all n different proofs of l using the computed rule for l. However, if we do not use the computed rule for l, it may well be that after only one step of expanding l, a contradiction is detected.

However, there is one case of recycling that is almost trivial to implement, yet whose effectiveness is independent of any strategy for rewriting. This is the case of recycling only failed proofs. In other words, if the computed rule for l is $l \rightarrow_R F$, then using this computed rule for l in the proof of another goal is always as good as the one without it. More precisely, if a terminating rewrite chain of q goes through l but does not use the computed rule for l, then there is another terminating rewrite chain of q that uses the computed rule for l, and has the same or fewer number of steps as the original rewrite chain. It is impossible to quantify the computational gains in terms of worst-case complexity analysis. In practice, we expect it to be very effective, as our following experiment shows.

7. EXPERIMENTS

We have implemented a depth-first search rewrite procedure with a branch-and-bound technique. The procedure can be used to compute explanations using a nonground program,under the condition that in each rule, a variable that appears in the body must also appear in the head. When this condition is not satisfied, we need only instantiate those variables that only appear in the body of a rule.

To check the effectiveness of the idea of recycling, we consider an application of abduction in logic programming, namely, the problem of computing successor state axioms from a causal action theory [Lin 1995, 2003; Lin and Wang 1999].

Consider a logistics domain in which we have a truck and a package. We know that the truck and 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 does the package. Suppose that we have the following propositions:

—ta(x): the truck is at location x initially;

—pa(x): the package is at location x initially;

—in: the package is in the truck initially;

—ta(x, y, z): the truck is at location x after the action of moving it from y to z is performed;[1]

—pa(x, y, z): the package is at location x after the action of moving the truck from y to z is performed; and

[1]Supposing that $move(y, z)$ stands for the action of driving the truck from locations y to z, and the current situation is s, then in situation calculus, this proposition can be written as $holds(ta(x), do(move(y, z), s))$. Our representation is sufficient here, since we only consider one type of action and two situations: the initial one, and that after the execution of the action.

—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:[2]

$$\begin{aligned}
&\mathrm{ta}(X, X1, X). && (3)\\
&\mathrm{pa}(X, X1, X2) \leftarrow \mathrm{ta}(X, X1, X2), \mathrm{in}(X1, X2).\\
&\mathrm{ta}(X, X1, X2) \leftarrow X \neq X2, \mathrm{ta}(X), \mathsf{not}\,\mathrm{taol}(X, X1, X2).\\
&\mathrm{taol}(X, X1, X2) \leftarrow Y \neq X, \mathrm{ta}(Y, X1, X2).\\
&\mathrm{pa}(X, X1, X2) \leftarrow \mathrm{pa}(X), \mathsf{not}\,\mathrm{paol}(X, X1, X2).\\
&\mathrm{paol}(X, X1, X2) \leftarrow Y \neq X, \mathrm{pa}(Y, X1, X2).\\
&\mathrm{in}(X, Y) \leftarrow \mathrm{in}. && (4)
\end{aligned}$$

The first rule is the effect axiom. The second 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 is the frame axiom about ta, with the help of a new predicate taol: If the truck is initially at X, and we cannot prove that it will be elsewhere after the action is performed, then it should still be at X.

As we can see, the preceding 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 $loc(X)$ for each variable X in the body of a rule (all the variables in the program refer to locations). Thus, the program is domain-restricted, and we need only instantiate 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:

$$\{\mathrm{in}\} \cup \{\mathrm{pa}(x), \mathrm{ta}(x) \mid x \in D\}.$$

Our job here is then to compute the abduction of successor state propositions

$$\{\mathrm{ta}(x, y, z), \mathrm{pa}(x, y, z), \mathrm{in}(y, z)\}$$

in terms of these abducibles, which are initial state propositions. For instance, given query pa(3, 2, 3), our system would compute its abduction as pa(3) $\vee$ in, meaning that for it to be true, either the package was initially at 3 or it was inside the truck.

According to the definition in the last section, literals that contain $\mathrm{pa}(X, Y, Z)$ depend on those that contain $\mathrm{in}(X, Y)$ and $\mathrm{ta}(X, Y, Z)$. But literals that contain $\mathrm{in}(X, Y)$ and those that contain $\mathrm{ta}(X, Y, Z)$ do not depend on each other. Thus we should first compute the abduction of $\mathrm{in}(X, Y)$ and $\mathrm{ta}(X, Y, Z)$. Now, $\mathrm{in}(X, Y)$ is solved by rule (4), $\mathrm{ta}(X, Y, X)$ by rule (3), and (as it turns out) when $X \neq Z$, $\mathrm{ta}(X, Y, Z)$ is always false, and its computation relatively easy. For instance, in our experiments for the domain with 9 locations, query

[2]See Lin and Wang [1999] for how logic programs of this kind can be used to encode a class of causal action theories.

Table I. Recycling in Logistics Domain

	9 Locations		10 Locations	
Query	NR	WR	NR	WR
pa(1,2,3)	0.71	0.41	1.50	0.89
-pa(1,2,3)	75.89	2.28	342.96	5.65
pa(3,2,3)	137.05	0.89	630.69	1.98
-pa(3,2,3)	2.97	1.98	7.64	5.03
pa(1,5,7)	122.87	0.75	278.07	1.31
-pa(1,5,7)	727.6	7.07	2534.09	19.08
pa(7,5,1)	108.66	17.82	188.50	30.72
-pa(7,5,1)	74.43	2.26	340.51	5.64
pa(7,1,7)	7619.72	20.78	29140.69	35.65
-pa(7,1,7)	2.98	2.01	7.71	5.05

Legends: NR—no recycling; WR—recycling ta(X, Y, Z) goals.
All times are in CPU seconds.

ta(7, 1, 6) took only 2.6 seconds. In comparison, query pa(7, 1, 7) took more than 7,000 seconds without recycling.

Table I contains runtime data for some representative queries.[3] For comparison purposes, each query is given two entries: the one under "NR" refers to a regular rewriting system without using recycling, and that under "WR" refers to rewriting system using computed rules about ta(X, Y, Z). As we can see, especially for hard queries like pa(7, 1, 7), recycling in this case significantly speeds-up the computation.

8. EXTENSIONS AND RELATED WORK

8.1 Some Extensions

So far we have assumed that the given logic program is propositional, and does not contain constraints. We discussed in Lin and You [2001] some possible extensions. Firstly, our rewrite systems work on a nonground program under the condition that in each rule, a variable that appears in the body must also appear in the head. Under this condition, an observation (a ground goal) is always rewritten to another ground goal, so that a rewriting mechanism designed for ground programs works just as well. When the condition is not satisfied, we need only instantiate those variables that only appear in the body of a rule. For example, domain-restricted programs [Niemelä 1999] can be instantiated only on domain predicates for variables that do not appear in the head. This is a significant departure from approaches that are based on ground computation, where a function-free program is first instantiated to a ground program with which the intended models are then computed.

Constraints of the form

$$\bot \leftarrow a_1, \ldots, a_i, \mathsf{not}\, b_1, \ldots, \mathsf{not}\, b_n$$

[3]Our implementation was written in Sicstus Prolog, and the experiments were done on a PIII 1GHz notebook with 512MB memory. This implementation is a significant improvement over the one in Lin and You [2001]. For instance, for a domain with seven locations, query pa(3, 2, 3) took more than 20 minutes for the implementation reported in Lin and You [2001], but required less than one second under our implementation running on a comparable machine.

can be handled in our rewriting procedure just like in other abductive procedures [Denecker and Schreye 1998; Fung and Kowalski 1997; Kakas et al. 2000]: Specifically, a goal is proved along with all the constraints. This ensures that all of the constraints are satisfied when the goal is proved.

Example 8.1. Consider the following program and constraint:

$$a \leftarrow \mathsf{not}\, b. \quad b \leftarrow \mathsf{not}\, a. \quad c \leftarrow c.$$
$$\bot \leftarrow b, \mathsf{not}\, c.$$

In trying to prove b, for example, the goal formula including the constraint is $b \wedge \neg(b \wedge \neg c)$, which reduces to $b \wedge (\neg b \vee c)$. The following rewrite sequence shows that there is no partial stable model that contains b while satisfying the given constraint.

$$\begin{aligned} & b \wedge (\neg b \vee c) \\ & \rightarrow (b \wedge \neg b) \vee (b \wedge c) \rightarrow \ldots\ldots \\ & \rightarrow F \vee (b \wedge c) \rightarrow b \wedge c \rightarrow \neg a \wedge c \rightarrow b \wedge c \\ & \rightarrow T(\{b, \neg a\}) \wedge c \rightarrow T(\{b, \neg a\}) \wedge c \rightarrow T(\{b, \neg a\}) \wedge F \rightarrow F \end{aligned}$$

However, computationally this is not a very effective approach when there are many constraints. Since a rewrite chain in our system carries a context, we could check the consistency of this context with the given constraints. This can be done by adding the following simplification rule:

$$T(C) \rightarrow F, \text{ if } C \text{ violates a constraint.}$$

Here we say that a context C violates a constraint $\bot \leftarrow G$ if G is a subset of C.

However, this approach may not be sound, for the same reason that our rewrite system is sound under partial stable model semantics, but not under the stable model semantics. Consider the following program:

$$p. \quad \bot \leftarrow p, q. \quad \bot \leftarrow p, \mathsf{not}\, q.$$

There is a rewrite of p to T under the context $\{p\}$ which does not violate any of the constraints according to our definition. However, p should not be "proved," as the context cannot be extended to one that satisfies both of the constraints. But this is the same reason that our rewrite system is not sound under the stable model semantics: A program may not have a stable model (e.g., with $a \leftarrow \mathsf{not}\, a$ the only rule whose head is a), but a goal may be written to T (e.g., a goal that does not mention a). So perhaps this strategy of dealing with constraints matches well with our rewrite systems, which are sound and complete under the partial stable model semantics.

8.2 Related Work

We discussed in Lin and You [2001] how our rewrite systems for abduction are related to other abductive systems, for example, Console et al. [1991], Denecker and Schreye [1998], Eshghi and Kowalski [1989], Fung and Kowalski [1997], Kakas et al. [1995] Kakas and Mancarella [1990a, 2000], and Satoh and

Iwayama [1992]. One useful way of looking at an abductive logic programming system is to consider the underlying semantics on which the system is based. For instance, the systems in Console et al. [1991], Fung and Kowalski [1997], Hayashi [1998] and Denecker and Schreye [1998] are based on Clark's completion semantics, the system of Kakas et al. [2001] is for well-founded models, and the procedure in Eshghi and Kowalski [1989] is sound and complete under the finite-failure three-valued semantics in which loops causing infinite failure are modeled by the truth value undefined [Giordano et al. 1996].

Our rewrite systems were designed with the stable model semantics in mind. As we saw in Section 5, we defined basic notions of abduction in logic programming, such as explanations, minimal explanations, and covers of explanations in terms of the stable model semantics. However, our rewrite systems are goal-directed, thus check for local consistency only. For instance, unless the atom a is "related to" to the query q, a rewrite chain starting at q would never reach a, thus would never check the consistency implicitly given by the odd loop $a \leftarrow \mathsf{not}\, a$. Thus our rewrite systems are sound and complete only under the partial stable model semantics.

According to Janhunen et al. [2004], given any normal or disjunctive program P, there is a polynomial-time translation from P to P' such that a query is true in an answer set of P if and only if it is true in a partial stable model of P' (see Corollaries 3.15–16 of Janhunen et al. [2004]). Hence, query answering under the answer set semantics can be carried out by query answering under the semantics based on partial stable models. Thus, our rewrite systems are applicable to the stable model semantics as well.

We can also apply the technique of rewriting for the answer set semantics without carrying out a transformation, as we outlined following Theorem 5.6.

We note that recycling in our case differs from the reuse of hypotheses in the abductive procedures in Eshghi and Kowalski [1989], Kakas and Mancarella [1990b], and Kakas et al. [2000]. In the last case, hypotheses are saved as individuals for the purpose of checking consistency, while in our case, the computed answers are saved. It is interesting to see that the consistency check in our rewrite procedures is already captured by the loop detection mechanism as defined in Section 3. The advantage of loop detection is not only that the completeness is guaranteed, but also that the procedure is terminating for propositional programs.

9. CONCLUDING REMARKS AND FUTURE WORK

We have considered the problem of how to reuse previously computed results for answering other queries in the abductive rewriting system of Lin and You [2001] for logic programs with negation, and showed that this can indeed be done. We have also described a methodology for using the recycling system in practice by analyzing the dependency relationship among propositions in logic programs. We applied this methodology to the problem of computing the effect of actions in a logistics domain, the same domain considered in Lin and You [2001], and our experimental results showed that recycling in this domain can indeed result in good performance gain.

For future work we are looking for more domains on which to try our system and to implement a system that can automatically analyze a program and decide how best to recycle previous computations.

ACKNOWLEDGMENTS

We would like to thank the reviewers of this article for their thoughtful comments. Fangzhen Lin's work was supported in part by the Research Grants Council of Hong Kong under Competitive Earmarked Research Grant HKUST6205/02E.

REFERENCES

CONSOLE, L., THESEIDER, D., AND PORASSO, P. 1991. On the relationship between abduction and deduction. *J. Logic Program. 2*, 5, 661–690.

DENECKER, M. AND SCHREYE, D. D. 1998. SLDNFA: An abductive procedure for normal abductive programs. *J. Logic Program. 34*, 2, 111–167.

DERSHOWITZ, N. AND JOUANNAUD, P. 1990. Rewrite systems. In *Handbook of Theoretical Computer Science: Formal Methods and Semantics*. North-Holland, vol. B, 243–320.

DIX, J. 1991. Classifying semantics of logic programs. In *Proceedings of the Ist Workshop on Logic Programming and Nonmonotonic Reasoning*. MIT Press, Cambridge, MA. 167–180.

DUNG, P. 1991. Negations as hypothesis: An abductive foundation for logic programming. In *Proceedings of the 8th International Conference on Logic Programming*. MIT Press, Cambridge, MA. 3–17.

DUNG, P. 1995. An argumentation theoretic foundation for logic programming. *J. Logic Program. 22*, 151–177.

ESHGHI, K. AND KOWALSKI, R. 1989. Abduction compared with negation by failure. In *Proceedings of the 6th International Conference on Logic Programming*. MIT Press, Cambridge, MA. 234–254.

FAGES, F. 1994. Consistency of Clark's completion and existence of stable models. *J. Methods Logic Comput. Sci. 1*, 51–60.

FUNG, T. AND KOWALSKI, R. 1997. The iff proof procedure for abductive logic programming. *J. Logic Program. 33*, 2, 151–164.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*. MIT Press, Cambridge, MA. 1070–1080.

GIORDANO, L., MARTELLI, A., AND SAPINO, M. 1996. Extending negation as failure by abduction: A three-valued stable model semantics. *J. Logic Program. 26,* 1, 31–67.

HAYASHI, H. 1998. Knowledge assimilation and proof restoration through the addition of goals. In *Proceedings of the 8th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*. Lacture Notes in Ccomputer Science, vol. 1480, Springer Verlag. 291–302.

JANHUNEN, T., NIEMELÄ, I., SEIPEL, D., SIMONS, P., AND YOU, J. 2004. Unfolding partiality and disjunctions in stable model semantics. In *ACM Trans. Comput. Logic*. To appear.

KAKAS, A., KOWALSKI, R., AND TONI, F. 1995. The role of abduction in logic programming. In *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University, Oxford, UK.

KAKAS, A. AND MANCARELLA, P. 1990a. Generalized stable models: A semantics for abduction. In *Proceedings of the 9th European Conference on Artificial Intelligence*. 285–291.

KAKAS, A. AND MANCARELLA, P. 1990b. On the relation between truth maintenance and abduction. In *Proceedings of the Ist Pacific Rim International and Conference on Artificial Intellgence (PRICAI)*. 438–443.

KAKAS, A., MICHAEL, A., AND MOURLAS, C. 2000. ACLP: Abductive constraint logic programming. *J. Logic Program. 44,* 1-3, 129–178.

KAKAS, A., VAN NUFFELEN, B., AND DENECKER, M. 2001. A-system: Problem solving through abduction. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*, New York. 591–596.

LIN, F. 1995. Embracing causality in specifying the indirect effects of actions. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann, New York. 1985–1993.

LIN, F. 2003. Compiling causal theories to successor state axioms and STRIPS-like systems. *J. Artif. Intell. Res. 19*, 279–314.

LIN, F. AND WANG, K. 1999. From causal theories to logic programs (sometimes). In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning.* Lacture Notes in Ccomputer Science, vol. 1730, Springer Verlag. 117–131.

LIN, F. AND YOU, J. 2001. Abduction in logic programming: A new definition and an abductive procedure based on rewriting. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 655–661. The full paper appears in *Artif. Intell. 140*, 1-2, 175–205 (2002).

LIN, F. AND ZHAO, Y. 2004. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artif. Intell. 157*, 1-2, 115–137.

NIEMELÄ, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annu. Math. Artif. Intell. 25,* 3-4, 241–273.

PRZYMUSINSKI, T. 1990. Extended stable semantics for normal and disjunctive logic programs. In *Proceedings of the 7th International Conference on Logic Programming*. MIT Press, Cambridge, MA. 459–477.

SATO, T. 1990. Completed logic programs and their consistency. *J. Logic Program. 9,* 1, 33–44.

SATOH, K. AND IWAYAMA, N. 1991. Computing abduction using the TMS. In *Proceedings of the 8th International Conference on Logic Programming*. MIT Press, Cambridge, MA. 505–518.

SATOH, K. AND IWAYAMA, R. 1992. A query evaluation method for abductive logic programming. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*. MIT Press, Cambridge, MA. 671–685.

SIMONS, P. 2000. Smodels: A system for computing the stable models of logic programs, version 2.26. `http://www.tcs.hut.fi/Software/smodels/.`

YOU, J. AND YUAN, L. 1995. On the equivalence of semantics for normal logic programs. *J. Logic Program. 22*, 212–221.

Received February 2004; revised November 2004; accepted April 2005

Proceedings of KR 2006: 298-307

First-Order Loop Formulas for Normal Logic Programs *

Yin Chen
South China Normal University
Guangzhou, P.R.China

Fangzhen Lin
Hong Kong University of Science and Technology
Hong Kong

Yisong Wang
Guizhou University
Guiyang, P.R.China

Mingyi Zhang
Guizhou Academy of Sciences
Guiyang, P.R.China

Abstract

In this paper we extend Lin and Zhao's notions of loops and loop formulas to normal logic programs that may contain variables. Under our definition, a loop formula of such a logic program is a first-order sentence. We show that together with Clark's completion, our notion of first-order loop formulas captures the answer set semantics on the instantiation-basis: for any finite set F of ground facts about the extensional relations of a program P, the answer sets of the ground program obtained by instantiating P using F are exactly the models of the propositional theory obtained by instantiating using F the first order theory consisting of the loop formulas of P and Clark's completion of the union of P and F. We also prove a theorem about how to check whether a normal logic program with variables has only a finite number of non-equivalent first-order loops.

Introduction

For a propositional normal logic program, Lin and Zhao (2004) showed that by adding what they called loop formulas to Clark's completion (Clark 1978), one obtains a one to one correspondence between the models of the resulting propositional theory and the answer sets of the logic program. There are several reasons why one wants to extend this result to the first-order case. For one, Clark's completion was originally defined to be a first-order theory on a set of rules with variables. More importantly, in answer set programming (ASP) applications (c.f. (Niemelä 1999; Marek & Truszczynski 1999; Lifschitz 1999; Nogueira *et al.* 2001; Baral 2003; Erdem *et al.* 2003; Eiter *et al.* 2004)), a logic program typically has two parts: a set of general rules with variables that encodes general knowledge about the application domain, and a set of facts that encodes the specific problem instance of the application domain and is used to ground the variables in the set of general rules. This means that when applying a logic program with variables to two problem instances, we have to compute the loops and loop formulas of two different propositional logic programs separately although these two programs, being obtained from grounding the same logic program on different domains, have basically the same structure thus should have the same kinds of loops and loop formulas. By defining loops and loop formulas directly on logic programs with variables, we can hopefully avoid this problem of having to compute similar loops and loop formulas every time a program is grounded on a domain. Thus extending loop formulas in logic programming to first-order case is not only theoretically interesting, but may also be of practical relevance.

Specifically, in this paper, we propose notions of first-order loops and loop formulas so that for any set P of rules with variables and any set F of ground facts about the extensional relations of P, the answer sets of the logic program obtained from grounding P using F will correspond to the propositional models of the theory obtained by instantiating $\mathit{COMP}(P \cup F) \cup LF(P)$ on D, where $\mathit{COMP}(P \cup F)$ is the completion of $P \cup F$, $LF(P)$ the set of loop formulas of P, D the set of constants in P and F, and an extensional relation is one that does not occur in the heads of any rules.

This paper is organized as follows. In the next section, we first define some basic notions such as the instantiations (groundings) of a first-order theory and a set of rules on a given finite domain. We then define the notions of positive dependency graph, loops, and loop formulas of a finite logic program with variables. We then prove some properties of our notions. Besides the correctness result as stated above, we also show some results about when a program has only a finite number of non-equivalent loops. Perhaps not surprisingly, given a finite set P of first-order rules, while the completion of P is a finite first-order theory, the set of loop formulas of P may be an infinite set of first-order formulas because there may be an infinite number of non-equivalent loops. Thus it is important to know whether a given program has only a finite number of non-equivalent loops. As we shall see, surprisingly perhaps, there is an algorithm for checking this.

Logical preliminaries

We assume a first-order language with equality but without proper functions. In the following, we shall call a formula of the form $t = t'$ an *equality atom*, and by an *atom* we mean an atomic formula in the language that is not an equality atom.

*Authors' names are listed according to the alphabetical order.
Copyright © 2006, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

Instantiating sentences on a finite domain

Given a first-order sentence φ, and a finite set D of constants, we define the *instantiation* of φ on D, written $\varphi|D$, to be a propositional formula defined inductively as follows:

- if φ does not have quantifications, then $\varphi|D$ is the result of replacing $d = d$ by $true$ and $d_1 = d_2$ by $false$ in φ, where d is any constant, and d_1 and d_2 are any two distinct constants.
- $\exists x.\varphi|D$ is $(\bigvee_{d \in D} \varphi(x/d))|D$, where $\varphi(x/d)$ is obtained from φ by replacing in it every free occurrence of x by d;
- $(\varphi_1 \vee \varphi_2)|D = \varphi_1|D \vee \varphi_2|D$;
- $(\neg\varphi)|D = \neg(\varphi|D)$.

Other connectives such as $\wedge$ and the universal quantification $\forall$ are treated as shorthands as usual. Notice that when instantiating a first-order sentence on a finite domain, we make unique names assumptions. So the instantiation of $a = a$ is $true$, the instantiation of $a = b$ is $false$, and the instantiation of $\forall x.p(x) \equiv x = a$ on $\{a, b\}$ is equivalent to $p(a) \wedge \neg p(b)$.

Now if T is a first-order theory (i.e. a set of first-order sentences) and D a finite set of constants, then we define the instantiation of T on D, written $T|D$, to be the set of the instantiations of the sentences in T on D.

In the following, we identify a propositional model of a propositional theory with the set of atoms true in the model. Given a first-order structure M with domain D, we define I_M to be following set:

$$\{p(\vec{t}) \mid \vec{t} \in p^M\},$$

where p^M is the interpretation of p in M. Symmetrically, given a set I of atoms, and a set D of constants, we define a first-order structure M_I^D as follows:

- The domain of M_I^D is D, and each constant in D is mapped to itself.
- For each predicate p, $\vec{t} \in p^{M_I^D}$ iff $p(\vec{t}) \in I$.

Clearly, if I is a set of ground atoms that mention only constants from D, then $I_{M_I^D} = I$.

Proposition 1 *Suppose φ is a first-order sentence, and D a finite domain that include all constants in φ. If I is a propositional model of $\varphi|D$, then M_I^D is a model of φ. Conversely, if M is a model of φ whose domain is D and it maps each constant in D to itself, then I_M is a propositional model of $\varphi|D$.*

Proof: We show that more generally, if D is a finite set that includes all of the constants in φ, and I a set of atoms such that

$$I \subseteq \{p(d_1, ..., d_k) \mid p \text{ a predicate and } d_1, ..., d_k \in D\},$$

then for any variable assignment σ, any formula ψ that mentions constants only in D, $M_I^D, \sigma \models \psi$ iff $I \models (\psi\sigma)|D$, where $\psi\sigma$ is the result of replacing every free variable in ψ by its value in σ. We prove this by induction.

- $M_I^D, \sigma \models t_1 = t_2$ iff $t_1\sigma = t_2\sigma$ (since all constants in t_1 and t_2 are in D, thus mapped to themselves) iff $[(t_1 = t_2)\sigma]|D$ is $true$.
- $M_I^D, \sigma \models p(\vec{t})$ iff $\vec{t}\sigma \in p^{M_I^D}$ iff $p(\vec{t})\sigma \in I$ iff $[p(\vec{t})\sigma]|D \in I$.
- $M_I^D, \sigma \models \exists x.\psi$ iff for some σ' that differs from σ only on x, we have that $M_I^D, \sigma' \models \psi$
  iff for some σ' that differs from σ only on x, we have that $(\psi\sigma')|D$ is true in I (by the inductive assumption)
  iff for some $d \in D$, $((\psi(x/d))\sigma)|D$ is true in I
  iff $\bigvee_{d \in D}((\psi(x/d))\sigma)|D$ is true in I
  iff $[(\bigvee_{d \in D} \psi(x/d))\sigma]|D$ is true in I
  iff $[(\exists x\psi)\sigma]|D$ is true in I.
- The cases for $\forall x\psi$, $\neg\psi$ and $\psi_1 \vee \psi_2$ are similar.

Similarly, we can show that if D is a finite set that includes all constants in φ, and M a first-order structure whose domain is D and such that it maps a constant in D to itself, then for any variable assignment σ, any formula ψ, $M, \sigma \models \psi$ iff $I_M \models (\psi\sigma)|D$. ■

The following is the key result about our notion of instantiation.

Proposition 2 *For any sentences φ and ψ, and any domain D that includes the constants in φ and ψ, if φ and ψ are logically equivalent in first-order logic (with equality), then the instantiations of φ and ψ on D are logically equivalent in propositional logic.*

Proof: Suppose I satisfies $\varphi|D$, then M_I^D is a model of φ. So M_I^D is a model of ψ. Thus $I_{M_I^D}$ satisfies $\psi|D$. But $I_{M_I^D} = I$. So I satisfies $\psi|D$ as well. ■

Notice that it is crucial for D to include all constants in φ and ψ. For instance, $\exists x(x = a)$ is logically equivalent to $true$, but $\exists x(x = a)|\{b\}$ is $false$.

Instantiating logic programs on a finite domain

A (first-order normal) logic program P is a set of (first-order normal) rules of the form:

$$h \leftarrow Body \tag{1}$$

where h is an atom, and $Body$ is a set of atoms, equality atoms, and expressions of the form $not\ A$, where A is either an atom or an equality atom. In the following, we use $t \neq t'$ as a shorthand for $not\ t = t'$. (However, as usual, when $t \neq t'$ occurs in a formula, it stands for $\neg t = t'$.)

According to this definition, a logic program can contain an infinite number of rules. However, in this paper, we consider only finite logic programs because Clark's completion may not be well-defined for an infinite set of rules. So in the following, unless otherwise specified, a logic program will be assumed to be finite.

In this paper, the semantics of a logic program with variables will be defined according to the answer set semantics (Gelfond & Lifschitz 1991) of propositional logic programs by grounding the first-order program on a domain.

Let r be a rule, we shall denote by $Var(r)$ the set of variables occurring in r, and $Const(r)$ the set of constants occurring in r. We call a variable $x \in Var(r)$ a *local* variable of r if it does not occur in the head of the rule. Similarly,

given a set P of rules, we shall denote by $Var(P)$ the set of variables in P, and $Const(P)$ the set of constants in P.

Given a set V of variables, and a set D of constants, a variable assignment of V on D is a mapping from V to D. If r is a rule, and σ a variable assignment, then $r\sigma$ is the result of replacing the variables in r by their values in σ.

Let r be a rule and D a set of constants. We define *the instantiation of r on D*, written $r|D$, to be the set of ground rules obtained from

$$R = \{r\sigma \mid \sigma \text{ is a variable assignment of } Var(r) \text{ on } D\}$$

by the following two transformations:

- if a rule in R contains either $a = b$ for some distinct constants a and b or $a \neq a$ for some constant a, then delete this rule;
- delete $a = a$ and $a \neq b$ for all constants a and b in the bodies of the remaining rules.

For instance, if r is the rule

$$p(x) \leftarrow q(x, y), x = a, x \neq y,$$

then its instantiation on $\{a, b\}$, $r|\{a, b\}$, is

$$\{p(a) \leftarrow q(a, b)\}.$$

The instantiation of a program P on D, written $P|D$, is then the union of the instantiations of all the rules in P on D.

Normal forms and program completions

Let r be a rule of the form (1), and suppose that h is $p(\vec{t})$ for some predicate p and tuple $\vec{t}$ of terms. If $\vec{x}$ is a tuple of variables not in r, and matches p's arity (so that $p(\vec{x})$ is well-formed), then the *normal form* of r on $\vec{x}$ is the following rule:

$$p(\vec{x}) \leftarrow \vec{x} = \vec{t} \cup Body,$$

where $(x_1, ..., x_k) = (t_1, ..., t_k)$ is the set:

$$\{x_1 = t_1, ..., x_k = t_k\}.$$

For instance, the normal form of the following rule

$$p(x, z) \leftarrow q(x, y), x \neq y$$

on (u, v) is

$$p(u, v) \leftarrow x = u, z = v, q(x, y), x \neq y$$

Notice that if none of the variables in $\vec{x}$ occur in a rule r, then all variables in r become local variables in the normal form of r on $\vec{x}$.

Using this notion of normal forms, we can define the *completion* of a predicate p under a program P, written $COMP(p, P)$, to be the following formula:

$$\forall \vec{x}.p(\vec{x}) \equiv \bigvee_{1 \leq i \leq k} \exists \vec{y_i} \widehat{Body_i},$$

where

- $\vec{x}$ is a tuple of distinct variables that are not in P, and matches p's arity;
- $(p(\vec{x}) \leftarrow Body_1), \cdots, (p(\vec{x}) \leftarrow Body_k)$ are the normal forms on $\vec{x}$ of all the rules in P whose heads mention the predicate p;
- for each $1 \leq i \leq k$, $\vec{y_i}$ is the tuple of local variables in the rule $(p(\vec{x}) \leftarrow Body_i)$, i.e., they occur in $Body_i$ but not in $\vec{x}$;
- $\widehat{Body_i}$ in the formula stands for the conjunction of all elements in $Body_i$ with "not" replaced by logical negation "$\neg$".

In particular, if a predicate $p(\vec{x})$ does not occur in the head of any rule in P, then its completion under P is equivalent to $\forall \vec{x} \neg p(\vec{x})$.

Now given a program P, we define the completion of P to be the set of the completions of all predicates in P.

Positive dependency graphs, loops, and loop formulas

As usual, a *binding* is an expression of the form x/t, where x is a variable, and t a term. A *substitution* is a set of bindings containing at most one binding for each variable. In the following, if φ is a first-order formula, and θ a substitution, we denote by $\varphi\theta$ the result of replacing every free variable in φ according to θ. In particular, a variable assignment can be considered a substitution, and if $\exists \vec{x} \varphi$ is a sentence, then $(\exists \vec{x} \varphi)|D$ is equivalent to the disjunction of the sentences in the set

$$\{(\varphi\sigma)|D \mid \sigma \text{ is a variable assignment of } \vec{x} \text{ on } D\},$$

and $(\forall \vec{x} \varphi)|D$ is equivalent to the conjunction of the sentences in the set

$$\{(\varphi\sigma)|D \mid \sigma \text{ is a variable assignment of } \vec{x} \text{ on } D\}.$$

Let P be a program. The *(first-order) positive dependency graph* of P, written G_P, is the infinite graph (V, E), where V is the set of atoms that do not mention any constants other than those in P, and for any $A, A' \in V$, $(A, A') \in E$ if there is a rule (1) in P and a substitution θ such that $h\theta = A$ and $b\theta = A'$ for some $b \in Body$.

A finite non-empty subset L of V is called a *(first-order) loop* of P if there is a non-zero length cycle that goes through only and all the nodes in L. This is the same as saying that for any A and A' in L, there is a non-zero length path from A to A' in the subgraph of G_P induced by L.

It is easy to see that if the given logic program P does not contain any variables, then the above definitions of positive dependency graph and loops are the same as those in (Lin & Zhao 2004).

Example 1 The following are some examples that illustrate our notion of loops.

- Let $P_1 = \{p(x) \leftarrow p(x)\}$. The vertices in P_1's positive dependency graph are atoms of the form $p(\xi)$, where ξ is a variable, and there is an arc from $p(\xi)$ to $p(\xi)$. Thus the loops of the program are singletons of the form $\{p(\xi)\}$, where ξ is a variable.

- Let P_2 be the following program:

$$\{(p(x) \leftarrow q(x)), (q(y) \leftarrow p(y)),$$
$$(p(x) \leftarrow r(x)), (q(y) \leftarrow not\ s(y))\}.$$

  The vertices in P_2's positive dependency graph are atoms of the forms $p(\xi)$, $q(\xi)$, $s(\xi)$ or $r(\xi)$, where ξ is a variable, and there is an arc from $p(\xi)$ to $q(\xi)$ and vice versa, and there is an arc from $p(\xi)$ to $r(\xi)$. Thus the loops of the program are sets of atoms of the form $\{p(\xi), q(\xi)\}$, where ξ is a variable.
- Let $P_3 = \{p(x) \leftarrow p(y)\}$. The vertices in P_3's positive dependency graph are atoms of the form $p(\xi)$, where ξ is a variable, and for any variables ξ and ζ, there is an arc from $p(\xi)$ to $p(\zeta)$. Thus any non-empty finite set of vertices is a loop. In terms of loops, this program has the same structure as Niemelä's program for Hamiltonian Circuit problem (Niemelä 1999).
- Let $P_4 = \{p(x, y) \leftarrow p(a, b)\}$, where a and b are constants. The vertices in P_4's positive dependency graph are atoms of the form $p(\xi, \zeta)$, where ξ and ζ are either a, b, or variables. There is an arc from $p(\xi_1, \zeta_1)$ to $p(\xi_2, \zeta_2)$ iff $\xi_2 = a$ and $\zeta_2 = b$. So the only loop of P_4 is $\{p(a, b)\}$.

To define loop formulas, we follow Lee (2005), define the *external support formula* of an atom in a loop first. Given a loop L of a program P, an atom $p(\vec{t})$, the external support formula of of $p(\vec{t})$ w.r.t. L, written $ES(p(\vec{t}), L, P)$, is the following formula:

$$\bigvee_{1 \le i \le k} \exists \vec{y_i} \left[\widehat{Body_i}\theta \wedge \bigwedge_{\substack{q(\vec{u}) \in Body_i\theta \\ q(\vec{v}) \in L}} \vec{u} \neq \vec{v} \right],$$

where

- $(p(\vec{x}) \leftarrow Body_1), \cdots, (p(\vec{x}) \leftarrow Body_k)$ are the normal forms on $\vec{x}$ of the rules in P whose heads mention the predicate p;
- $\vec{x}$ is a tuple of distinct variables that are not in P, and if $\vec{t} = (t_1, ..., t_n)$ and $\vec{x} = (x_1, ..., x_n)$, then

$$\theta = \{x_1/t_1, ..., x_n/t_n\}$$

  (so that $\vec{x}\theta = \vec{t}$);
- for each $1 \le i \le k$, $\vec{y_i}$ is the tuple of local variables in $p(\vec{x}) \leftarrow Body_i$. We assume that $\vec{y_i} \cap Var(L) = \emptyset$, by renaming local variables in $Body_i$ if necessary, where $Var(L)$ is the set of variables in L. In other words, if a rule has a local variable that is also in the loop, then we have to rename the local variable in the rule.

Notice that the only free variables in the formula $ES(p(\vec{t}), L, P)$ are those in $\vec{t}$. The choice of the variables in $\vec{x}$ is not material.

The (first-order) loop formula of L in P, written $LF(L, P)$, is then the following formula:

$$\forall \vec{x} \left[\bigvee_{A \in L} A \supset \bigvee_{A \in L} ES(A, L, P) \right], \qquad (2)$$

where $\vec{x}$ is the tuple of variables in L.

Example 2 We continue with the logic programs in Example 1.

- Consider $P_1 = \{p(x) \leftarrow p(x)\}$, and the loop $\{p(z)\}$. The normal form of the rule on y is $p(y) \leftarrow y = x, p(x)$. So the external support formula of $p(z)$ w.r.t. this loop is

$$\exists x.z = x \wedge p(x) \wedge x \neq z$$

  which is equivalent to $false$. Thus the loop formula for this loop is equivalent to $\forall x \neg p(x)$. The loop formulas of the other loops like $\{p(x)\}$ is equivalent to this formula as well.
- Consider

$$P_2 = \{(p(x) \leftarrow q(x)), (q(y) \leftarrow p(y)),$$
$$(p(x) \leftarrow r(x)), (q(y) \leftarrow not\ s(y))\}.$$

  and the loop $\{p(x), q(x)\}$. To compute the external support formula of $p(x)$, we first normalize all the rules about p on a new variable, say z:

$$p(z) \leftarrow z = x, q(x),$$
$$p(z) \leftarrow z = x, r(x).$$

  Since x is a local variable in the above rules, and it occurs in the loop, we replace it by another variable, say x_1:

$$p(z) \leftarrow z = x_1, q(x_1),$$
$$p(z) \leftarrow z = x_1, r(x_1).$$

  Thus the external support formula of $p(x)$ is

$$\exists x_1(x = x_1 \wedge q(x_1) \wedge x \neq x_1) \vee \exists x_1(x = x_1 \wedge r(x)),$$

  which is equivalent to $r(x)$. Similarly, the external support formula of $q(x)$ is equivalent to $\neg s(x)$. Thus the loop formula of this loop is equivalent to

$$\forall x.(p(x) \vee q(x)) \supset (r(x) \vee \neg s(x)).$$

  It can be seen that the loop formulas of all loops in P_2 are equivalent to this sentence.
- Consider $P_3 = \{p(x) \leftarrow p(y)\}$, and the loop $\{p(x_1), \cdots, p(x_k)\}$. The external support formula for each $p(x_i)$ is equivalent to

$$\exists y.p(y) \wedge \bigwedge_{1 \le i \le k} x_i \neq y.$$

  Thus the loop formula of this loop is equivalent to

$$\forall x_1, ..., x_k. \bigvee_{1 \le i \le k} p(x_i) \supset \exists y.p(y) \wedge \bigwedge_{1 \le i \le k} x_i \neq y.$$

Main theorem

In the following, we call a predicate in a program *extensional* if it does not occur in the head of any rule in the program, and the other predicates in the program are called *intensional*.

Theorem 1 *Let P be a finite set of rules, and F a finite set of ground facts about the extensional relations. Let D be the set of constants in $P \cup F$. A set M of ground facts is an answer set of $(P|D) \cup F$ iff M is a propositional model of $(COMP(P \cup F) \cup LF(P))|D$, where $COMP(P \cup F)$ is the completion of $P \cup F$ and $LF(P)$ the set of the loop formulas of P.*

Proof: See the next section. ■

Notice that $COMP(P \cup F)$ is actually the union of two sets: the set of the completions of the intensional predicates of P under P and the set of the completions of the extensional predicates of P under F. The first set can be computed before we are given the actual problem instance F.

We have defined a loop to be a set of vertices such that there is a *non-zero* length cycle that goes through all and only the vertices in the set. We can generalize this notion of loops by allowing zero length cycle, like what is done in (Lee & Lin 2006) and (Lee 2005). This will make every singleton set of nodes to be a loop, and the loop formulas of these singleton sets will correspond to Clark's completion. Under this more general notion of loops, Theorem 1 can be restated as follows:

> Let P be a finite set of rules, and F a finite set of ground facts about the extensional relations. Let D be the set of constants in $P \cup F$. A set M of ground facts is an answer set of $(P|D) \cup F$ iff M is a propositional model of $(\widehat{P} \cup F \cup LF'(P))|D$, where $LF'(P)$ is the set of the loop formulas for the (generalized) loops in P, and $\widehat{P}$ is the set of sentences obtained from P by replacing each rule $(F \leftarrow B)$ in it by the sentence $\forall \vec{x}.\widehat{B} \supset F$, where $\vec{x}$ is the tuple of variables in the rule.

Example 3 The following are some examples that illustrate the theorem.

- Consider P_2 from Examples 1 and 2.
  Suppose $F = \{r(a), s(b)\}$. Then $D = \{a, b\}$. By normalizing all rules on z, it is easy to see that the completion of $P_2 \cup F$ is equivalent to
  $$\forall z.p(z) \equiv \exists x(z = x \wedge q(x)) \vee \exists x(z = x \wedge r(x)),$$
  $$\forall z.q(z) \equiv \exists y(z = y \wedge p(y)) \vee \exists y(z = y \wedge \neg s(y)),$$
  $$\forall z.r(z) \equiv z = a, \quad \forall z.s(z) \equiv z = b,$$
  which is equivalent to
  $$\forall z.p(z) \equiv q(z) \vee r(z), \quad \forall z.q(z) \equiv p(z) \vee \neg s(z)),$$
  $$\forall z.r(z) \equiv z = a, \quad \forall z.s(z) \equiv z = b.$$
  Thus by Proposition 2, $COMP(P_2 \cup F)|D$ is equivalent to the set of the following set of sentences:
  $$p(a) \wedge q(a) \wedge p(b) \equiv q(b),$$
  $$r(a) \wedge \neg r(b) \wedge \neg s(a) \wedge s(b).$$
  As we have shown in Example 2, the loop formulas of all the loops are equivalent to
  $$\forall x.(p(x) \vee q(x)) \supset (r(x) \vee \neg s(x)).$$
  Thus $LF(P_2)|D$ is equivalent to $\neg p(b) \wedge \neg q(b)$. Thus the only propositional model of $COMP(P_2 \cup F) \cup LF(P_2)|D$ is $\{p(a), q(a), r(a), s(b)\}$.
  Now the instantiation of the program $P_2 \cup F$ on $\{a, b\}$ is:
  $$p(a) \leftarrow q(a).\ p(b) \leftarrow q(b).\ p(a) \leftarrow r(a).$$
  $$p(b) \leftarrow r(b).\ q(a) \leftarrow p(a).\ q(b) \leftarrow p(b).$$
  $$q(a) \leftarrow not\ s(a).\ q(b) \leftarrow not\ s(b).$$
  There is a unique answer set of this program, which is the same as the propositional model given above.

- Consider the program P_5 that contains the following rules:
  $$p(x) \leftarrow p(y)$$
  $$p(x) \leftarrow q(x, y), x \neq y.$$
  This program adds one more rule about p to the program P_3 in Examples 1 and 2. The loops of the program are the same as those of P_3, and of the form $\{p(\xi_1), ..., p(\xi_k)\}$. The loop formulas are equivalent to the following sentence:
  $$\forall x_1, ..., x_k. \bigvee_{1 \le i \le k} p(x_i) \supset$$
  $$[\exists y.p(y) \wedge \bigwedge_{1 \le i \le k} x_i \neq y] \vee [\bigvee_{1 \le i \le k} \exists y.q(x_i, y) \wedge x_i \neq y].$$
  Now let $F = \{q(a, b)\}$. Then $D = \{a, b\}$. If $k = 1$, then the instantiation of the above sentence is equivalent to
  $$(p(a) \supset p(b) \vee q(a, b)) \wedge (p(b) \supset p(a) \vee q(b, a)). \quad (3)$$
  For $k \ge 2$, the instantiations of the above sentences are all equivalent to the conjunction of (3) and the following sentence:
  $$(p(a) \vee p(b)) \supset (q(a, b) \vee q(b, a)).$$
  Now for the completion of $P_5 \cup F$, by normalizing all rules on u, v, we see that it is equivalent to
  $$\forall u.p(u) \equiv \exists x, y.(u = x \wedge p(y)) \vee$$
  $$\exists x, y.(u = x \wedge q(x, y) \wedge x \neq y),$$
  $$\forall u, v.q(u, v) \equiv (u = a \wedge v = b).$$
  Thus $COMP(P_5 \cup F)|D$ is equivalent to
  $$q(a, b) \wedge \neg q(a, a) \wedge \neg q(b, a) \wedge \neg q(b, b) \wedge p(a) \wedge p(b).$$
  Notice that this formula implies $LF(P_5)|D$, so the latter is not needed for this example. Now the instantiation of the program $P_5 \cup F$ on $\{a, b\}$ is:
  $$p(a) \leftarrow p(a).\ p(a) \leftarrow p(b).\ p(b) \leftarrow p(a).\ p(b) \leftarrow p(b).$$
  $$p(a) \leftarrow q(a, b).\ p(b) \leftarrow q(b, a).\ q(a, b).$$
  The unique answer set of this program is the same as the model of $COMP(P_5 \cup F)|D$.

Proof of the main theorem

The proof of Theorem 1 is tedious, but the basic idea is simple. Instead of $P|D$, we use an equivalent logic program $P_{eq}|D$ as defined below. The main task is then to show that there is a one-to-one correspondence between the loops of $P_{eq}|D$ and the instantiations of the first-order loops of P, and that this correspondence works between the loop formulas of $P_{eq}|D$ and the instantiations of the first-order loop formulas of P.

Notice that the instantiation of a program on a finite domain, $P|D$, eliminates all equality atoms. While doing this, it also delete rules whose bodies have an equality literal that is false. Thus the ground loops of $P|D$ and the instantiations of the first-order loops of P on D may not be in one-to-one

correspondence. So to faithfully preserve the loops of P, we introduce P_{eq} as defined below.

Let P be a program. Without loss of generality, suppose that P does not mention eq as a binary predicate. Let P_{eq} be the program obtained from P by replacing $t = t'$ in it by $eq(t, t')$. Notice that $t \neq t'$ will be replaced by $not\ eq(t, t')$ as the former is a shorthand for $not\ t = t'$.

Proposition 3 *For any finite domain D such that $Const(P) \subseteq D$, a set M of atoms occurring in $P|D$ is an answer set of $P|D$ iff $M \cup EQ_D$ is an answer set of $P_{eq}|D \cup EQ_D$, where $EQ_D = \{eq(d, d) \mid d \in D\}$.*

Proof: Since eq does not occur in the head of any rule in $P_{eq}|D$, $(P_{eq}|D) \cup EQ_D$ is equivalent to $(P|D) \cup EQ_D$. ■

So $P|D$ and $P_{eq} \cup EQ_D$ are essentially the same. But the latter is a more faithful way to ground P on D in the sense that it will not lose any loops of P. For instance, $\{p(x), p(y)\}$ is a loop of program

$$P = \{p(x) \leftarrow p(y), x \neq x\},$$

but for any D, $P|D$ is the empty program, and thus has no loop. But $P_{eq}|D$ has the same loops as those of P. Our next proposition shows that this is true in general.

Proposition 4 *Let P be a finite set of rules and D a finite set of constants containing $Const(P)$. If GL is a loop of $P_{eq}|D$ then for some loop L of P, and some variable assignment θ on D, $GL = L\theta$. Conversely, if L is a loop of P, and θ is a variable assignment on D, then $L\theta$ is a loop of $P_{eq}|D$.*

Proof: Let G_0 be the positive dependency graph of the propositional program $P_{eq}|D$, and G_1 the positive dependency graph of P.

Let A and B be two atoms in $P_{eq}|D$ that do not mention eq. Let $d_1, ..., d_k$ be the constants in A and B that are not in $Const(P)$, and A' and B' the results of replacing each d_i by x_i in A and B, respectively. Then we have that

(A, B) is an arc in G_0
iff
There is a rule $A \leftarrow B, Body$ in $P_{eq}|D$
iff
There is a rule $\alpha \leftarrow \beta, body$ in P_{eq} and a variable assignment σ such that $A = \alpha\sigma$ and $B = \beta\sigma$
iff
There is a rule $\alpha \leftarrow \beta, body'$ in P (since α and β do not mention eq) and a variable assignment σ such that $A = \alpha\sigma$ and $B = \beta\sigma$
iff
There is a rule $\alpha \leftarrow \beta, body'$ in P and a variable assignment σ' such that $A' = \alpha\sigma'$ and $B' = \beta\sigma'$. (σ' can be obtained from σ by replacing every $x/d_i (1 \leq i \leq k)$ in σ with x/x_i.)
iff
(A', B') is an arc in G_1.

Now if GL is a loop of $P_{eq}|D$, then there is a cycle $(A_1, ..., A_n, A_{n+1})$, where $A_{n+1} = A_1$, in G_0 such that $GL = \{A_1, ..., A_n\}$. Thus for each $1 \leq i \leq n$, A_i does not mention eq and (A_i, A_{i+1}) is an arc in G_0. Thus for each $1 \leq i \leq n$, (A'_i, A'_{i+1}) is an arc in G_1, where A'_i's are defined as above. Thus $\{A'_1, ..., A'_n\}$ is a loop of P.

Conversely, suppose L is a loop of P, and θ a variable assignment of $Var(L)$ on D. Then there is a cycle $(\alpha_1, ..., \alpha_n, \alpha_{n+1} = \alpha_1)$ in G_1 such that $L = \{\alpha_1, ..., \alpha_n\}$. Thus for each $1 \leq i \leq n$, (α_i, α_{i+1}) is an arc in G_1. Thus for each $1 \leq i \leq n$, $(\alpha_i\theta, \alpha_{i+1}\theta)$ is an arc in G_0 (the fact that each $\alpha_i\theta$ is a node in G_0 follows from the assumption that $Const(P) \subseteq D$). Thus $L\theta = \{\alpha_1\theta, ..., \alpha_n\theta\}$ is a loop of $P_{eq}|D$. ■

In the following, if P is a ground logic program, L a set of atoms in P, and A an atom in L, then we define the *ground support formula* of A w.r.t. L, written $GES(A, L, P)$, to be the following formula

$$\bigvee_{1 \leq i \leq k, B_i \cap L = \emptyset} \widehat{B_i}$$

where $(A \leftarrow B_1), ..., (A \leftarrow B_k)$ are the rules in P whose heads are A. Now if L is a loop of the ground program P, then the following formula, written $GLF(L, P)$,

$$\bigvee_{A \in L} A \supset \bigvee_{A \in L} GES(A, L, P)$$

is equivalent to the loop formula of L as defined in (Lin & Zhao 2004). In the following, we call $GLF(L, P)$ the *ground loop formula* of L in P.

Lemma 1 *Let P be a program, L be a loop of P_{eq} and D be a finite set of constants containing $Const(P)$. Then, for any an atom A in L and any substitution θ of $Var(L)$ on D, we have*

$$GES(A\theta, L\theta, P_{eq}|D) \equiv ES(A, L, P_{eq})\theta|D.$$

Proof: Let $A = p(\vec{t})$. Without loss of generality, we assume that there is only one rule in P whose head mentions p: $p(\vec{w}) \leftarrow B$, and does not contain variables in L (by renaming variables in the rule if necessary). Suppose that $\vec{x}$ is a tuple of variables not in P and L, and we normalize this rule on $\vec{x}$: $p(\vec{x}) \leftarrow B \cup \{\vec{x} = \vec{w}\}$. $GES(A\theta, L\theta, P_{eq}|D)$ is

$$\bigvee_{\vec{w}\sigma = \vec{t}\theta, B_{eq}\sigma \cap L\theta = \emptyset} \widehat{B_{eq}\sigma}, \tag{4}$$

where σ ranges over all variable assignments on D, and B_{eq} is the result of replacing equality in B by eq.

$ES(A, L, P_{eq})$ is

$$(\exists \vec{y})(\widehat{B_{eq}} \wedge \vec{t} = \vec{w} \wedge \bigwedge_{\substack{q(\vec{u}) \in B_{eq} \\ q(\vec{v}) \in L}} \vec{u} \neq \vec{v}),$$

where $\vec{y}$ is the tuple of variables in B and $\vec{w}$. Since none of the variables in B and $\vec{w}$ occur in L, and θ is a substitution of variables in L, so $ES(A, L, P_{eq})\theta$ is

$$(\exists \vec{y})(\widehat{B_{eq}} \wedge \vec{t}\theta = \vec{w} \wedge \bigwedge_{\substack{q(\vec{u}) \in B_{eq} \\ q(\vec{v}) \in L}} \vec{u} \neq \vec{v}\theta).$$

Thus $ES(A, L, P_{eq})\theta|D$ is equivalent to

$$\bigvee_{\sigma} \widehat{B_{eq}\sigma}, \tag{5}$$

where σ ranges over variable assignments on D such that

- $\vec{t}\theta$ and $\vec{w}\sigma$ are the same
- for every atom $q(\vec{v}) \in L$ and every atom $q(\vec{u}) \in B_{eq}$, $\vec{u}\sigma$ and $\vec{v}\theta$ are not the same, that is $L\theta \cap B_{eq}\sigma = \emptyset$.

Thus (4) and (5) are the same, and this proves the proposition. ■

Proposition 5 *Let P be a program and D be a finite set of constants containing $Const(P)$. Then $LF(P_{eq})|D$ and $GLF(P_{eq}|D)$ are logically equivalent, where for any program Q, $GLF(Q)$ is the set of ground loop formulas of Q.*

Proof: $LF(P_{eq})|D$ is

$$\{LF(L, P_{eq})|D \mid L \text{ is a loop of } P_{eq}\}.$$

For each loop L of P_{eq}, $LF(L, P_{eq})$ is

$$\forall\vec{x}\left[\bigvee_{A\in L} A \supset \bigvee_{A\in L} ES(A, L, P_{eq})\right].$$

Thus $LF(L, P_{eq})|D$ is equivalent to the set of following sentences:

$$\left[\bigvee_{A\in L} A \supset \bigvee_{A\in L} ES(A, L, P_{eq})\right]\sigma|D,$$

where σ is a variable assignment of $\vec{x}$ on D. The above sentence is equivalent to

$$\left[\bigvee_{A\in L} A\sigma \supset \bigvee_{A\in L} ES(A, L, P_{eq})\sigma\right]|D,$$

which is equivalent to the following sentence as for each $A \in L$, $A\sigma$ is a ground atom,

$$\bigvee_{A\in L} A\sigma \supset \bigvee_{A\in L} (ES(A, L, P_{eq})\sigma|D),$$

which, by Lemma 1, is equivalent to

$$\bigvee_{A\in L} A\sigma \supset \bigvee_{A\in L} GES(A\sigma, L\sigma, P_{eq}|D),$$

which is equivalent to

$$[\bigvee_{B\in L\sigma} B \supset \bigvee_{B\in L\sigma} GES(B, L\sigma, P_{eq}|D),$$

which is $GLP(L\sigma, P_{eq}|D)$. Thus by Proposition 4, $LF(P_{eq})|D$ is equivalent to $GLF(P_{eq}|D)$. ■

In the following, if P is a ground logic program, then we denote by $GCOMP(a, P)$ the ground completion of an atom a in P.

Proposition 6 *Let P be a program, and D a finite set of constants containing $Const(P)$. For each predicate p in P, $COMP(p, P_{eq})|D$ is equivalent to*

$$\{GCOMP(p(\vec{x})\sigma, P_{eq}|D) \mid \sigma \text{ is a variable assignment of } \vec{x} \text{ on } D\}.$$

Proof: Again without loss of generality, we assume that $p(\vec{w}) \leftarrow B$ is the only rule in P with p in its head, and $\vec{x}$ is a tuple of variables not in P. Then $COMP(p, P_{eq})$ is equivalent to

$$\forall\vec{x}p(\vec{x}) \equiv \exists\vec{y}\widehat{B_{eq}} \wedge \vec{x} = \vec{w},$$

where $\vec{y}$ is the tuple of variables in B and $\vec{w}$. So $COMP(p, P_{eq})|D$ is equivalent to the conjunction of the following sentences:

$$p(\vec{x})\sigma \equiv (\exists\vec{y}\widehat{B_{eq}} \wedge \vec{x}\sigma = \vec{w})|D,$$

where σ is a variable assignment on D, which is equivalent to

$$p(\vec{x})\sigma \equiv \bigvee_{\vec{x}\sigma=\vec{w}\tau} \widehat{B_{eq}}\tau,$$

where τ ranges over all variable assignments of $\vec{x}$ on D, which is exactly $GCOMP(p(\vec{x})\sigma, P_{eq}|D)$. ■

Finally, we can prove the main theorem:

Theorem 1 *Let P be a finite set of rules, and F a finite set of ground facts about the extensional relations. Let D be the set of constants in $P \cup F$. A set M of ground facts is an answer set of $(P|D) \cup F$ iff M is a propositional model of $(COMP(P \cup F) \cup LF(P))|D$, where $COMP(P \cup F)$ is the completion of $P \cup F$ and $LF(P)$ the set of the loop formulas of P.*

Proof: In the following, we denote by $GCOMP_1(Q)$ the set of ground completions of the atoms not mentioning "eq" in Q:

$$GCOMP_1(Q) = \{GCOMP(A, Q)|A \text{ does not mention } eq\}.$$

Similarly, we let

$$COMP_1(Q) = \{COMP(p, Q) \mid p \text{ is not } eq\}.$$

We also let

$$IU(D) = \{eq(d, d)|d \in D\} \cup \{\neg eq(d, d')|d \text{ and } d' \text{ are two distinct constants of } D\}.$$

M is an answer set of $(P|D) \cup F$
iff
M is an answer set of $(P \cup F)|D$
iff
$M \cup EQ_D$ is an answer set of $(P \cup F)_{eq}|D \cup EQ_D$ (Proposition 3)
iff
$M \cup EQ_D$ is a propositional model of

$$GCOMP((P\cup F)_{eq}|D\cup EQ_D)\cup GLF((P\cup F)_{eq}|D\cup EQ_D)$$

(Theorem 1 of (Lin & Zhao 2004))
iff
$M \cup EQ_D$ is a propositional model of

$$GCOMP_1((P\cup F)_{eq}|D)\cup IU(D)\cup GLF(P_{eq}|D\cup F\cup EQ_D)$$

($COMP(eq, P_{eq} \cup F \cup EQ_D)$ is equivalent to $IU(D)$)
iff
$M \cup EQ_D$ is a propositional model of

$$GCOMP_1((P \cup F)_{eq}|D) \cup IU(D) \cup GLF(P_{eq}|D)$$

iff
$M \cup EQ_D$ is a propositional model of

$$COMP_1((P \cup F)_{eq})|D \cup IU(D) \cup LF(P_{eq})|D$$

(by Proposition 6 and Proposition 5)
iff
$M \cup EQ_D$ is a propositional model of

$$[COMP_1((P \cup F)_{eq}) \cup \{\forall x, y(eq(x,y) \equiv x = y)\} \cup LF(P_{eq})]|D$$

iff
$M \cup EQ_D$ is a propositional model of

$$[COMP(P \cup F) \cup \{\forall x, y(eq(x,y) \equiv x = y)\} \cup LF(P)]|D$$

(by Proposition 2 and that in first-order logic,

$$\{\forall x, y(eq(x,y) \equiv x = y)\} \models COMP(p, Q_{eq}) \equiv COMP(p, Q),$$

for any program Q and predicate p in Q, and

$$\{\forall x, y(eq(x,y) \equiv x = y)\} \models LF(L, Q_{eq}) \equiv LF(L, Q),$$

for any loop L)
iff
$M \cup EQ_D$ is a propositional model of

$$[COMP(P \cup F) \cup LF(P)]|D \cup \{\forall x, y(eq(x,y) \equiv x = y)\}|D$$

iff
$M \cup EQ_D$ is a propositional model of

$$[COMP(P \cup F) \cup LF(P)]|D \cup IU(D)$$

iff
M is a propositional model of

$$[COMP(P \cup F) \cup LF(P)]|D$$

as M and $[COMP(P \cup F) \cup LF(P)]|D$ do not mention eq. ■

Some properties of loops and loop formulas

It is clear from the definition that if L is a loop of P, and θ is a substitution such that $Const(\theta) \subseteq Const(P)$, then $L\theta$ is also a loop of P, where $Const(\theta)$ is the set of constants occurring in θ. Thus if P has a loop that contains a variable, then P has an infinite number of loops. But as we have seen from the examples many of these loops are "equivalent" in the sense that their loop formulas are logically equivalent in first-order logic. This motivates our following definition.

Let L_1 and L_2 be two sets of atoms. We say L_1 *subsumes* L_2 if there is substitution θ such that $L_1\theta = L_2$. We say that L_1 and L_2 are *equivalent* if they subsume each other. For instance, $\{p(x)\}$ and $\{p(y)\}$ are equivalent. The set $\{p(x_1), p(x_2)\}$ subsumes $\{p(x)\}$, but not the other way around. If L_1 subsumes L_2, then the loop formula of L_1 also entails the loop formula of L_2.

Proposition 7 *Let L_1 be a loop of P, and L_2 a set of atoms such that $Const(L_2) \subseteq Const(P)$. If L_1 subsumes L_2, then L_2 is also a loop of P. Furthermore,*

$$\models LF(L_1, P) \supset LF(L_2, P).$$

Proof: Let θ be a substitution of $Var(L_1)$ such that $L_1\theta = L_2$. Since L_1 is a loop, there is a cycle $A_1, ..., A_n, A_{n+1} = A_1$ in G_P, the dependency graph of P, such that $L_1 = \{A_1, ..., A_n\}$. By the definition of G_P, $A_1\theta, ..., A_n\theta, A_{n+1}\theta$ is also a cycle of G_P. Thus $L_2 = \{A_1\theta, ..., A_n\theta\}$ is also a loop of P.

Now $LF(L_1, P)$ is $\forall\vec{\xi}\Phi(\vec{\xi})$, where $\vec{\xi}$ is the tuple of variables in L_1, and $\Phi(\vec{\xi})$ is

$$\left[\bigvee_{A \in L_1} A \supset \bigvee_{A \in L_1} ES(A, L_1, P)\right]. \tag{6}$$

We show that $LF(L_2, P)$ is equivalent to $\forall\vec{\zeta}\Phi(\vec{\xi})\theta$, where $\vec{\zeta}$ is the tuple of variables in $L_2 = L_1\theta$, i.e. $\vec{\xi}\theta$.

Notice that $LF(L_2, P)$ is

$$\forall\vec{\zeta}\left[\bigvee_{A \in L_1} A\theta \supset \bigvee_{A \in L_1} ES(A\theta, L_1\theta, P)\right]. \tag{7}$$

Thus it is enough to show that, for any $p(\vec{t}) \in L_1$, $ES(p(\vec{t}), L_1, P)\theta \equiv ES(p(\vec{t})\theta, L_1\theta, P)$. That is

$$\bigvee_{1 \le i \le k} \exists\vec{y_i}\left[\widehat{B_i}\delta_1 \wedge \bigwedge_{\substack{q(\vec{u}) \in B_i\delta_1 \\ q(\vec{v}) \in L_1}} \vec{u} \neq \vec{v}\right]\theta$$

$$\equiv \bigvee_{1 \le i \le k} \exists\vec{y_i}\left[\widehat{B_i}\delta_2 \wedge \bigwedge_{\substack{q(\vec{u}) \in B_i\delta_2 \\ q(\vec{v}) \in L_1\theta}} \vec{u} \neq \vec{v}\right]$$

where

- $\vec{x}$ is a tuple of distinct variables that are not in P, and
  $$(p(\vec{x}) \leftarrow B_1), \cdots, (p(\vec{x}) \leftarrow B_k)$$
  are the normal forms on $\vec{x}$ of all the rules in P whose heads mention the predicate p.
- Let $\vec{t} = (t_1, ..., t_n)$ and $\vec{x} = (x_1, ..., x_n)$. Then $\delta_1 = \{x_1/t_1, ..., x_n/t_n\}$ (so that $\vec{x}\delta_1 = \vec{t}$), and similarly for δ_2. So $p(\vec{x})\delta_1 = p(\vec{t})$ and $p(\vec{x})\delta_2 = p(\vec{t})\theta$.
- For each $1 \le i \le k$, $\vec{y_i}$ is the tuple of local variables in $p(\vec{x}) \leftarrow B_i$. We assume that $\vec{y_i} \cap Var(L_1) = \emptyset$, by renaming local variables in B_i if necessary. For the same reason, we also assume that $\vec{y_i} \cap Var(L_2) = \emptyset$.

The equivalence follows because

$$\bigvee_{1 \le i \le k} \exists\vec{y_i}\left[\widehat{B_i}\delta_1 \wedge \bigwedge_{\substack{q(\vec{u}) \in B_i\delta_1 \\ q(\vec{v}) \in L_1}} \vec{u} \neq \vec{v}\right]\theta$$

$$\equiv \bigvee_{1 \le i \le k} \exists\vec{y_i}\left[\widehat{B_i}\delta_1\theta \wedge \bigwedge_{\substack{q(\vec{u}) \in B_i\delta_1 \\ q(\vec{v}) \in L_1}} \vec{u}\theta \neq \vec{v}\theta\right],$$

$\widehat{B_i}\delta_1\theta$ is the same as $\widehat{B_i}\delta_2$ and $\bigwedge_{\substack{q(\vec{u}) \in B_i\delta_1 \\ q(\vec{v}) \in L_1}} \vec{u}\theta \neq \vec{v}\theta$ is the same as $\bigwedge_{\substack{q(\vec{u}) \in B_i\delta_2 \\ q(\vec{v}) \in L_1\theta}} \vec{u} \neq \vec{v}$. ■

Given a program P, a set Δ of loops of P is said to be *complete* if for any loop L of P, there is a loop $L' \in \Delta$ such that L' subsumes L. Thus by Proposition 7, if Δ is a complete set of loops of P, then the set of loop formulas of the loops in Δ is logically equivalent to the set of the loop formulas of all the loops of P. Of special interests are programs that have finite complete sets of loops. For the programs in Example 1, $\{\{p(x)\}\}$ is a complete set of loops for P_1, $\{\{p(x), q(x)\}\}$ is a complete set of loops for P_2. But P_3 has no finite complete set of loops. One of its complete sets of loops is $\{\{p(x_1)\}, \cdots, \{p(x_1), \cdots, p(x_k)\}, \cdots\}$.

An interesting question then is how to decide whether a program has a finite complete set of loops. The following theorem answers this question.

Theorem 2 *Let P be a finite logic program, and*

$$D = \{c_1, c_2\} \cup Const(P),$$

where c_1 and c_2 are two new constants not in P. The following five assertions are equivalent:

1. *P has a finite complete set of loops.*
2. *There is a natural number N such that, for any loop L of P, the size of $Var(L)$ is smaller than N.*
3. *For any loop L of P, and any atoms A_1 and A_2 in L, $Var(A_1) = Var(A_2)$.*
4. *For any loop L of $P_{eq}|D$, there are no two atoms A_1 and A_2 in L such that either A_1 mentions c_1 but A_2 does not or A_1 mentions c_2 but A_2 does not.*
5. *For any maximal loop (w.r.t. to subset ordering) L of $P_{eq}|D$, there are no two atoms A_1 and A_2 in L such that either A_1 mentions c_1 but A_2 does not or A_1 mentions c_2 but A_2 does not.*

Proof: "1 $\Leftarrow$ 2": Let Δ be the set of loops of P that mention only variables in $V = \{x_1, ..., x_N\}$. Clearly, Δ is finite. We show that it is complete. Let L be any loop of P. Let L' be obtained from L by replacing in it variables in $Var(L) \setminus V$ by distinct variables in $V \setminus Var(L)$ - this is possible as L mentions at most N variables. Clearly $L' \in \Delta$, and subsumes L.

"2 $\Leftarrow$ 3": P is a finite program, so there are at most finite predicates occurring in it. Let N be the maximum arity of all the predicates in P. For any loop L in P, the number of variables in L is at most N.

"3 $\Leftarrow$ 4": Suppose otherwise, and L is a loop of P such that there are two atoms A_1 and A_2 in L such that $Var(A_1) \neq Var(A_2)$. Without loss of generality, let x be a variable in A_1 and not in A_2. Let θ be the substitution $\{x/c_1\} \cup \{x'/c_2 \mid x' \in Var(L) \setminus \{x\}\}$. Then $L\theta$ is a loop of $P_{eq}|D$. Moreover, we can see $A_1\theta$ mentions c_1 and $A_2\theta$ does not, and $A_1\theta$ and $A_2\theta$ are two atoms in $L\theta$, which is a contradiction.

"4 $\Leftarrow$ 1": Let $\Delta = \{L_1, ..., L_n\}$ be a finite complete set of loops of P. For every $L_i \in \Delta$, $Var(L_i)$ is finite since L_i is finite. Now let $N = max\{|Var(L_1)|, ..., |Var(L_n)|\}$. For any loop L of P, there is a loop $L^* \in \Delta$ such that L^* subsumes L, and consequently,

$$|Var(L)| \leq |Var(L^*)| \leq N.$$

Now suppose condition 4 does not hold, and let $L = \{A_1, A_2, \ldots, A_n\}$ be a ground loop such that, without loss of generality, c_1 occurs in A_1 and not in A_2. Let x and y be two new variables not in L, and for each i, A'_i be obtained from A_i by replacing c_1 with x and c_2 with y. Let $L^{[1]} = \{A'_1, A'_2, \ldots, A'_n\}$. Clearly, A'_1 mentions x but A'_2 does not. But by Proposition 4 (more precisely as shown in the proof of this proposition), $L^{[1]}$ is loop of P.

From loop $L^{[1]}$, we can construct another loop $L^{[2]}$ of P as following. Let $\delta = \{x/x^*\}$ be a substitution, where x^* is variable not mentioned in $L^{[1]}$. Then $L^{[1]}\delta$ is also a loop of P. Notice that A'_2 does not mention x, so $A'_2 = A'_2\delta$, thus $A'_2 \in L^{[1]} \cap L^{[1]}\delta$. Let $L^{[2]} = L^{[1]} \cup L^{[1]}\delta$, then $L^{[2]}$ is also a loop of P as the union of any two loops that have a common element is also a loop. Clearly,

$$|Var(L^{[2]})| = |Var(L^{[1]})| + 1.$$

This procedure can be iterated, and for any $L^{[i]}$, we can construct a loop $L^{[i+1]}$ such that

$$|Var(L^{[i+1]})| = |Var(L^{[i]})| + 1.$$

So, for any given nature number N^*, we can construct a loop L^* of P such that $|Var(L^*)| > N^*$, a contradiction with condition 1.

Finally, "4 $\Leftrightarrow$ 5" as $P_{eq}|D$ is a finite program, thus for any loop of this program, there is a maximal one containing it. ■

Notice that for the purpose of computing loops, EQ_D is not needed: $P_{eq}|D$ and $P_{eq}|D \cup EQ_D$ have the same loops. Notice also that since P is a finite set of rules, the set D is finite, thus $P_{eq}|D$ in the theorem is also finite. In this case, since $P_{eq}|D$ has no variables, its loops cannot have variables either. In fact, for $P_{eq}|D$, the definition of loops given here is the same as that in the propositional case given in (Lin & Zhao 2004). In particular, the set of loops of $P_{eq}|D$ is finite. From Theorem 2.5, we have the following proposition:

Proposition 8 *Let P be a finite program, M the number of rules in P, N the maximum length of the rules in P, n the number of predicates (including equality) in P, k the maximum arity of the predicates (counting equality) in P, and c the number of constants in P. Then there is a $O(MNn^2(c+2)^{2k})$ algorithm for checking whether P has a finite complete set of loops.*

Proof: $n(c+2)^k$ is the maximum number of atoms in $P_{eq}|D$, $n^2(c+2)^{2k}$ the maximum number of arcs in the dependency graph of $P_{eq}|D$, $O(MNn^2(c+2)^{2k})$ the worst case time for constructing the graph, and there is a linear time algorithm in the size of the dependency graph of $P_{eq}|D$ for computing all maximal loops of the program. ■

The exact complexity of deciding whether P has a finite complete set of loops is an open question.

Recall that a variable in a rule is said to be a local one if it occurs in the body of the rule but not in the head. A variable of a program is said to be a local one if it is a local variable in one of the rules in the program. From Theorem 2, we can easily prove the following result.

Proposition 9 *If P does not have any local variables, then P has a finite complete set of loops.*

Proof: If P has no local variables in any of its rules, then by the definition of dependency graph G_P, if there is an arc from A to B in G_P, then $Var(A) = Var(B)$. The result then follows from the definition of the loop and Theorem 2. ■

For instance, programs P_1 and P_2 in Example 1 do not have any local variables.

Concluding remarks

We have proposed notions of loops and loop formulas for normal logic programs with variables. Our main result is Theorem 1, which basically says that for a logic program with variables, we can first compute a first-order theory consisting of its completion and loop formulas so that when given a finite domain, the instantiation of the first-order theory on this domain yields a propositional theory whose models are the same as the answer sets of the ground logic program obtained from instantiating the first-order logic program.

This means that in principle, we can pre-compute the completion and loop formulas of a logic program with variables before we know the actual problem domain, and once the domain is known, all that is needed is to instantiate the first-order theory on the domain to yield a set of clauses to be given to a SAT solver. It is likely that for logic programs that have finite complete sets of loops, pre-computing all the loops is more efficient than ASSAT-like strategies that have to compute loop formulas on each ground program, especially when the performance is averaged over a large number of domains. However, for logic programs such as Niemelä's program for Hamiltonian Circuit problem (Niemelä 1999) that do not have a finite complete set of loops, it is not clear whether pre-computing first-order loops will be more effective computationally. To answer this question, an empirical study is needed.

Another future work is that as we have mentioned, in this paper, the semantics of logic programs with variables are defined according to the propositional answer set semantics of the the grounded programs. In other words, variables in programs are just place-holders, and rules with variables are schemas. In this sense, the logic programs considered in this paper are not truly first-order. Recently, Lifschitz[1] introduced a semantics of logic programs with variables that does not depend on grounding. Also the formulation of answer set semantics for normal logic programs in circumscription (Lin 1991) can be applied to programs with variables as well. How the notions of loops and loop formulas proposed in this paper relate to these semantics of logic programs with variables is an open question that is worth studying.

Acknowledgments

Yisong Wang and Mingyi Zhang were partially supported by the Natural Science Foundation of China under grant NSFC 60573009 and the stadholder foundation of Guizhou Province under grant 2005(212).

[1]Invited talk at ASP'05, Bath, UK, July 2005.

References

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Clark, K. L. 1978. Negation as failure. In Gallaire, H., and Minker, J., eds., *Logics and Databases*. New York: Plenum Press. 293–322.

Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2004. A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM Trans. Comput. Log. 5(2)* 206–263.

Erdem, E.; Lifschitz, V.; Nakhleh, L.; and Ringe, D. 2003. Reconstructing the evolutionary history of indo-european languages using answer set programming. In *Proceedings of PADL 2003*, 160–176.

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

Lee, J., and Lin, F. 2006. Loop formulas for circumscription. *Artificial Intelligence* 170(2):160–185.

Lee, J. 2005. A model-theoretic counterpart of loop formulas. In *Proceedings of IJCAI-05*, 503–508.

Lifschitz, V. 1999. Action languages, answer sets and planning. In *The Logic Programming Paradigm: A 25-Year Perspective*. K.R. Apt, V.W. Marek, M. Truszczynski, D.S. Warren, eds, Springer-Verlag.

Lin, F., and Zhao, Y. 2004. ASSAT: Computing answer sets of a logic program by sat solvers. *Artificial Intelligence* 157(1-2):115–137.

Lin, F. 1991. *A Study of Nonmonotonic Reasoning*. Department of Computer Science, Stanford University, Stanford, CA: PhD thesis.

Marek, V. W., and Truszczynski, M. 1999. Stable logic programming - an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*. K.R. Apt, V.W. Marek, M. Truszczynski, D.S. Warren, eds, Springer-Verlag.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. and AI* 25(3-4):241–273.

Nogueira, M.; Balduccini, M.; Gelfond, M.; Watson, R.; and Barry, M. 2001. An A-Prolog decision support system for the space shuttle. In *Proceedings of PADL 2001*, 169–183.

Available online at www.sciencedirect.com
SCIENCE DIRECT®
Artificial Intelligence 157 (2004) 115–137
Artificial Intelligence
www.elsevier.com/locate/artint

ASSAT: computing answer sets of a logic program by SAT solvers

Fangzhen Lin *, Yuting Zhao

Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong

Received 28 December 2002; accepted 14 April 2004

Abstract

We propose a new translation from normal logic programs with constraints under the answer set semantics to propositional logic. Given a normal logic program, we show that by adding, for each loop in the program, a corresponding loop formula to the program's completion, we obtain a one-to-one correspondence between the answer sets of the program and the models of the resulting propositional theory. In the worst case, there may be an exponential number of loops in a logic program. To address this problem, we propose an approach that adds loop formulas a few at a time, selectively. Based on these results, we implement a system called ASSAT(X), depending on the SAT solver X used, for computing one answer set of a normal logic program with constraints. We test the system on a variety of benchmarks including the graph coloring, the blocks world planning, and Hamiltonian Circuit domains. Our experimental results show that in these domains, for the task of generating one answer set of a normal logic program, our system has a clear edge over the state-of-art answer set programming systems Smodels and DLV.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Answer set programming; Answer set semantics; Stable model semantics; Logic programming; SAT; SAT solvers

1. Introduction

Logic programming with answer sets semantics [7] and propositional logic are closely related. It is well known that there is a local and modular translation from clauses to

* Corresponding author.
E-mail addresses: flin@cs.ust.hk (F. Lin), yzhao@cs.ust.hk (Y. Zhao).

0004-3702/$ – see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2004.04.004

logic program rules such that the models of a set of clauses and the answer sets of its corresponding logic program are in one-to-one correspondence [18,24].

The other direction is more difficult and interesting. Niemelä [18] showed that there cannot be a modular translation from normal logic programs to sets of clauses, in the sense that for any programs P_1 and P_2, the translation of $P_1 \cup P_2$ is the union of the translations of P_1 and P_2. However, the problem becomes interesting when we drop the requirement of modularity. In the special case when all rules in a program have at most one literal in their bodies (so-called 2-literal programs), Huang et al. [8] showed that there is an efficient translation to sets of clauses that do not need to use any extra variables. They also observed that many of the logic programs in answer set programming applications are essentially 2-literal ones. For these programs, their experiments showed the advantage of computing answer sets using SAT solvers such as SATO over Smodels [21].

In the general case, Ben-Eliyahu and Dechter [3] gave a translation for a class of disjunctive logic programs, which includes all normal logic programs. However, one problem with their translation is that it may need to use a quadratic number of extra propositional variables. While the number of variables is not always a reliable indicator of the hardness of a SAT problem, it nonetheless exerts a heavy toll on current SAT solvers when the number gets too big (for instance, the default setting in SATO allows only a maximum of 30,000 variables). In the worst case, adding one more variable could double the search space.

In this paper we shall propose a new translation. To motivate, consider the completion of a logic program. It is well known that every answer set of a logic program is also a model of the completion of the program, but the converse is not true in general. Fages [6] essentially showed that if a logic program has no positive loops, then every model of its completion is also an answer set. Recently, Babovich et al. [2] extended Fages' result and showed that this continues to be true if the logic program is what they called "tight" on every model of its completion.

Intuitively, the completion semantics is too weak because it does not handle positive cycles properly. For instance, the program $\{p \leftarrow q,\ q \leftarrow p\}$ has a cycle $p \rightarrow q \rightarrow p$. Its completion is $\{p \equiv q, q \equiv p\}$, which has two models. But the program has a unique answer set in which both p and q are false.

To address this problem, we define a loop in a program to be a set of atoms such that for any pair of atoms in the set, there is a positive path from one to the other in the dependency graph of the program [1]. The basic idea of our translation is then to associate a formula with each loop in a program. The formula captures the logical conditions under which the atoms in the loop can be in an answer set. For instance, for the above program, $\{p, q\}$ is a loop. The formula associated with it is $(p \vee q) \supset \mathit{false}$, meaning that none of them can be in any answer set of the program.

In this paper, we show that if we add these formulas for loops to the completion of a program, we obtain a propositional theory whose models are in one-to-one correspondence with the answer sets of the logic program.

The advantages of our translation over the one by Ben-Eliyahu and Dechter mentioned above are that it does not use any extra variables[1] and is intuitive and easy to understand as one can easily work it out by hand for typical "textbook" example programs.

However, one problem with our translation is that in the worst case, there may be an exponential number of loops in a logic program. To overcome this, we propose an implementation strategy that does not compute all loops of a logic program at once, but iteratively computes a few with certain properties.

Our work contributes to both the areas of answer set logic programming and propositional satisfiability. It provides a basis for an alternative implementation of answer set logic programming by leveraging on existing extensive work on SAT with a choice of variety of SAT solvers ranging from complete systematic ones to incomplete randomized ones. Indeed, our experiments on some well known benchmarks such as graph coloring, planning, and Hamiltonian Circuit (HC) show that for the problem of generating one answer set of a logic program, our system has a clear advantage over the two popular specialized answer set generators, Smodels [18,21] and DLV [12]. On the other hand, this work also benefits SAT in providing some hard instances: we have encountered some relatively small SAT problems (about 720 variables and 4500 clauses) that could not be solved using any of the SAT solvers that we tried.

This paper is organized as follows. We first introduce some basic concepts and notations used in the paper. We then define a notion of loops and their associated loop formulas, and show that a set is an answer set of a logic program iff it satisfies its completion and the set of all loop formulas. Based on this result, we propose an algorithm and implement a system called ASSAT for computing the answer sets of a logic program using SAT solvers. We then report some experimental results of running ASSAT on graph coloring, blocks world planning, and HC domains, and compare them with those using Smodels and DLV.

2. Logical preliminaries

In this paper, we consider only fully grounded finite normal logic programs that may have constraints. That is, a logic program here is a finite set consisting of rules of the form:

$$p \leftarrow p_1, \ldots, p_k, \mathsf{not}\, q_1, \ldots, \mathsf{not}\, q_m, \tag{1}$$

and constraints of the form:

$$\leftarrow p_1, \ldots, p_k, \mathsf{not}\, q_1, \ldots, \mathsf{not}\, q_m, \tag{2}$$

where $k \geqslant 0, m \geqslant 0$, and p, $p_1, \ldots, p_k$, $q_1, \ldots, q_m$ are atoms without variables. Notice that the order of literals in the body of a rule or a constraint is not important under the answer set semantics, and we have written negative literals after positive ones. In effect, this means that a body is the conjunction of a set of literals, i.e., a rule of the form (1) can also be denoted by $p \leftarrow G$, where G is the set of literals in the body.

[1] This refers to the translation from logic programs to propositional theories. It is well known that to avoid generating an exponential number of clauses, one may have to introduce some new variables when converting a propositional theory to a set of clauses. See Section 5.

To define the *answer sets* of a logic program with constraints, we first define the *stable models* of a logic program that does not have any constraints [7]. Given a logic program P without constraints, and a set S of atoms, the Gelfond–Lifschitz transformation of P on S, written P_S, is obtained from P as follows:

- For each negative literal $\mathsf{not}\, q$ in the body of any rule in P, if $q \notin S$, then delete this literal from this body.
- In the resulting set of rules, delete all those that still contain a negative literal in their bodies, i.e., if a rule contains a $\mathsf{not}\, q$ in its body such that $q \in S$, then delete this rule.

Clearly for any S, P_S is a set of rules without any negative literals. This means that there is a unique minimal model of P_S, which is the same as the set of atoms that can be derived from the program using resolution when rules are interpreted as implications. In the following, we shall denote this set by $Cons(P_S)$. Now a set S is a stable model [7] of P iff $S = Cons(P_S)$.

In general, given a logic program P with constraints, a set S of atoms is an answer set if it is a stable model of the program obtained by deleting all the constraints in P, and it satisfies all the constraints in P, i.e., for any constraint of the form (2) in P, either $p_i \notin S$ for some $1 \leqslant i \leqslant k$ or $q_j \in S$ for some $1 \leqslant j \leqslant m$.

In the following, given a logic program P, we denote by $atom(P)$ the set of atoms appearing in the program P. Given a logic program P, its *completion*, written $Comp(P)$, is the union of the constraints in P and the Clark completion [5] of the set of rules in P, that is, it consists of following sentences:

- For each $p \in atom(P)$, let $p \leftarrow G_1, \ldots, p \leftarrow G_n$ be all the rules about p in P, then $p \equiv G_1 \vee \cdots \vee G_n$ is in $Comp(P)$. In particular, if $n = 0$, then the equivalence is $p \equiv \mathit{false}$, which is equivalent to $\neg p$.
- If $\leftarrow G$ is a constraint in P, then $\neg G$ is in $Comp(P)$.

Here we have somewhat abused the notation and write the body of a rule in a formula as well. Its intended meaning is as follows: if the body G is empty, then it is understood to be *true* in a formula, otherwise, it is the conjunction of the literals in G with not replaced by $\neg$. For example, the completion of the program:

$$a \leftarrow b, c, \mathsf{not}\, d.$$
$$a \leftarrow b, \mathsf{not}\, c, \mathsf{not}\, d.$$
$$\leftarrow b, c, \mathsf{not}\, d.$$

is

$$\{a \equiv (b \wedge c \wedge \neg d) \vee (b \wedge \neg c \wedge \neg d), \neg b, \neg c, \neg d, \neg(b \wedge c \wedge \neg d)\}.$$

In this paper, we shall identify a truth assignment with the set of atoms true in this assignment, and conversely, identify a set of atoms with the truth assignment that assigns an atom true iff it is in the set. Under this convention, it is well known that if S is an answer set of P, then S is also a model of $Comp(P)$, but the converse is not true in general.

In this paper we shall consider how to strengthen the completion so that a set is an answer set of a logic program iff it is a model of the strengthened theory. The key concepts are loops and their associated formulas. For these, it is most convenient to define the *positive dependency graph* of a logic program.

Given a logic program P, the positive dependency graph of P, written G_P, is the following directed graph: the set of vertices is $atom(P)$, and for any two vertices p, q, there is an arc from p to q if there is a rule of the form $p \leftarrow G$ in P such that $q \in G$ (recall that we can treat the body of a rule as a set of literals). Informally, an arc from p to q means that p positively depends on q. Notice that $\mathsf{not}\, q \in G$ does not imply an arc from p to q.

Recall that a directed graph is said to be *strongly connected* if for any two vertices in the graph, there is a (directed) path from one to the other. Given a directed graph, a *strongly connected component* is a set S of vertices such that for any $u, v \in S$, there is a path from u to v, and that S is not a subset of any other such set.

3. Loops and their formulas

As we mentioned, cycles are the reason why models of a logic program's completion may not be answer sets. Consider again the program $\{a \leftarrow b.\ b \leftarrow a.\}$. The set $\{a, b\}$ is a "loop" in the sense that a positively depends on b by the first rule, and b on a by the second rule. In propositional logic, if there is a "loop" like this, one is free to make any assumptions about the truth values of the atoms in the "loop", as long as the constraints, in this case the sentences in the completion of the program, are satisfied. But in the answer set semantics, one cannot assume that an atom is true without a justification or a proof. In this case, since there is no way to prove that either a or b is true, so they are not true by the default negation-as-failure assumption used by the answer set semantics.

Definition 1. A nonempty subset L of $atom(P)$ is called a loop in a logic program P if for any p, q in L, there is a path in G_P from p to q of length > 0. In other words, a nonempty subset L of $atom(P)$ is a loop in P if the subgraph of G_P induced by L is strongly connected. In the following, atoms in a loop are sometimes called *loop atoms*.

Given a logic program P, and a loop L in it, we associate two sets of rules with it:

$$R^+(L, P) = \{p \leftarrow G \mid (p \leftarrow G) \in P, p \in L, (\exists q).q \in G \wedge q \in L\},$$
$$R^-(L, P) = \{p \leftarrow G \mid (p \leftarrow G) \in P, p \in L, \neg(\exists q).q \in G \wedge q \in L\}.$$

In the following, when the program P is clear from the context, we will write $R^+(L, P)$ as $R^+(L)$, and $R^-(L, P)$ as $R^-(L)$.

It is clear that these two sets are disjoint, and every rule whose head is in L is in exactly one of the sets. Intuitively, $R^+(L)$ contains rules *in the loop*, and they give rise to arcs connecting vertices in L in P's positive dependency graph; on the other hand, $R^-(L)$ contains those rules about atoms in L that are *out of the loop*. For instance, for the program $\{a \leftarrow b.\ b \leftarrow a.\ a.\}$, the only loop is $\{a, b\}$, and for this loop, R^+ is $\{a \leftarrow b.\ b \leftarrow a.\}$, and R^- is $\{a.\}$.

Example 1. As a simple example, consider P below:

$$a \leftarrow b.\ \ b \leftarrow a.\ \ a \leftarrow \mathsf{not}\, c.$$
$$c \leftarrow d.\ \ d \leftarrow c.\ \ c \leftarrow \mathsf{not}\, a.$$

There are two loops in this program: $L_1 = \{a, b\}$ and $L_2 = \{c, d\}$. For these two loops, we have:

$$R^+(L_1) = \{a \leftarrow b.\ b \leftarrow a.\}, \quad R^-(L_1) = \{a \leftarrow \mathsf{not}\, c.\},$$
$$R^+(L_2) = \{c \leftarrow d.\ d \leftarrow c.\}, \quad R^-(L_2) = \{c \leftarrow \mathsf{not}\, a.\}.$$

While L_1 and L_2 above are disjoint, this is not always the case. However, if two loops have a common element, then their union is also a loop.

For any given logic program P and any loop L in P, one can observe that $\emptyset$ is the only answer set of $R^+(L)$. Therefore an atom in the loop cannot be in any answer set unless it is derived using some other rules, i.e., those from R^-. This motivates our definition of *loop formulas.*

Definition 2. Let P be a logic program, and L a loop in it. Let $R^-(L)$ be the following set of rules:

$$p_1 \leftarrow G_{11}, \ldots, p_1 \leftarrow G_{1k_1},$$
$$\vdots$$
$$p_n \leftarrow G_{n1}, \ldots, p_n \leftarrow G_{nk_n}.$$

Then the (*loop*) *formula* associated with L (under P), denoted by $LF(L, P)$, or simply $LF(L)$ when P is clear from the context, is the following implication:

$$\neg[G_{11} \vee \cdots G_{1k_1} \vee \cdots \vee G_{n1} \vee \cdots \vee G_{nk_n}] \supset \bigwedge_{p \in L} \neg p. \tag{3}$$

Example 2. Consider again the program and loops in Example 1 above. $LF(L_1)$ is $c \supset (\neg a \wedge \neg b)$, and $LF(L_2)$ is $a \supset (\neg c \wedge \neg d)$. Notice that the completion of P, $Comp(P)$, is:

$$a \equiv \neg c \vee b,$$
$$b \equiv a,$$
$$c \equiv \neg a \vee d,$$
$$d \equiv c,$$

which has three models: $\{a, b\}$, $\{c, d\}$, and $\{a, b, c, d\}$. However if we add the above two loop formulas to $Comp(P)$, it will eliminate the last model, and the remaining two are exactly the answer sets of P. The following theorem shows that this is always the case.

Theorem 1. *Let P be a logic program, $Comp(P)$ its completion, and LF the set of loop formulas associated with the loops of P. We have that for any set of atoms, it is an answer set of P iff it is a model of $Comp(P) \cup LF$.*

Proof. We prove this for programs that do not have constraints. From this, the result for the ones with constraints follows straightforwardly.

Let A be a set of atoms that satisfies $Comp(P) \cup LF$. Let P_A be the Gelfond–Lifschitz reduction of P on A. We need to show that A is exactly the set of atoms that can be derived from P_A. We use the convention that a rule with empty body is considered to have the tautology *true* as its body.

Let S_0 be the set of rules in P_A such that both their head and body are satisfied by A:

$$S_0 = \{p \leftarrow G \mid p \leftarrow G \in P_A \text{ and both } p \text{ and } G \text{ are true in } A\}.$$

Now because A is a model of $Comp(P)$, for every $p \in A$, there is a rule in S_0 whose head is p. Suppose we have S_i, then construct S_{i+1} as follows:

- If there is no loop in S_i, then let $S_{i+1} = S_i$.
- If there is a loop in S_i, suppose that $L_i = (p_1, \ldots, p_n)$ is a maximal one (in terms of subset relation), and $R^+(L_i, S_i)$ is the following set of rules:

$$r_1 : p_1 \leftarrow G_1,$$
$$\vdots$$
$$r_m : p_n \leftarrow G_m.$$

  where $n \leqslant m$. Then L_i must be a loop in P as well. Now since $A \models p_1$ and A is a model of $LF(L_i, P)$, there must be some $1 \leqslant k \leqslant n$, and a rule $p_k \leftarrow G, G'$ in $R^-(L_i, P)$ such that $A \models G \wedge G'$, where G is a set of atoms and G' a set of negative literals. By the definition of $R^-(L_i, P)$, $G \cap L_i = \emptyset$, so the reduct of this rule, $r : p_k \leftarrow G$, is not equal to any of $r_1, \ldots, r_n$. Now let S_{i+1} be the result of deleting all those rules in $r_1, \ldots, r_n$ whose head is p_k.

We can show the following properties about S_i:

- For some finite n, $S_k = S_n$ for all $k > n$, and for such n, S_n does not have any loops. This is quite obvious as there are only finite number of loops in S_0, and every S_i is a subset of S_0. In the following. let S be this S_n: $S = \bigcap_{i=1,2,\ldots} S_i$.
- For any i, if there is a loop in S_i, then the rule $r : p_k \leftarrow G$ used in the construction of S_{i+1} from S_i is in S_{i+1}. By the construction of S_{i+1}, we only need to show that it is in S_i. We prove this by induction. It is clear that $r \in S_0$. Suppose that $r \in S_j$ for some $j < i$, we show that $r \in S_{j+1}$, that is r cannot be deleted from S_j. Suppose otherwise, there is a maximal loop L in S_j for which $r \in R^+(L, S_j)$. Now let L' be the maximal loop in S_i used in the construction of S_{i+1}. Then since the head of r, p_k, is an element in both L and L', $L \cup L'$ is also a loop in S_j. So $L' \subseteq L$ as L is maximal. Now since r is deleted from S_j, by the construction of S_{j+1}, all rules in $R^+(L, S_j)$ whose head is p_k

are also deleted. So there are no rules in $R^+(L', S_i)$ whose head is p_k, a contradiction with the selection of r in the construction of S_{i+1}.

- For each atom $p \in A$, there is a rule in S whose head is p. We prove this by induction. We have shown this for S_0 as A is a model of $Comp(P)$. Suppose that this is true for S_i, we prove it below for S_{i+1}. If there is no loop in S_i, then this is trivially true. Suppose it has a loop. By the construction of S_{i+1}, it is the result of deleting some of the rules in S_i whose heads are identical to the head of the rule $r : p_k \leftarrow G$ used in the construction. So for any atom $p \in A$ that is different from p_k, there is a rule for it in S_i by our inductive assumption, and the same rule is also in S_{i+1} by our construction. Now for this atom p_k, as we have shown above, $r \in S_{i+1}$, which is a rule for p_k.
- For every $p \in A$, $S \models p$. We prove this by contradiction using induction. Notice that we have proved the following by now: (1) for each $p \in A$, there is a rule in S whose head is p; (2) for each rule in S, both the head and the body are true in A; and (3) S does not have any loops. Assume that there exists $p_0 \in A$, s.t. $S \not\models p_0$. Construct a set of sequences of atoms in A inductively as follows. First of all, let $T_0 = \{p_0\}$. Clearly, T_0 satisfies the following properties for T:
  - T is a sequence of distinct atoms in A.
  - None of atoms in T is entailed by S.
  - If p_i and p_{i+1} are two consecutive elements in T, then p_i depends on p_{i+1} in the sense that there is a rule $p_i \leftarrow G$ in S such that $p_{i+1} \in G$.

  Suppose that we have a sequence $T_k = [p_0, \ldots, p_k]$ with the above properties, construct T_{k+1} as follows: Let $p_k \leftarrow G_k$ be a rule in S. Since $S \not\models p_k$, $G_k \neq \emptyset$ and $S \not\models G_k$. Thus there must exist $p_{k+1} \in G_k$, s.t. $S \not\models p_{k+1}$. Since S is loop free, and each of the atoms in T_k depends on the next one, so for any i, $0 \leqslant i \leqslant k$, $p_{k+1} \neq p_i$. Now let $T_{k+1} = [p_0, \ldots, p_k, p_{k+1}]$. One can see that T_{k+1} so constructed also satisfies the two properties about T above. But this is impossible as A is finite and we could construct T_k this way for arbitrary k.

Thus $P_A \models p$ as $S \subseteq P_A$. Now if $P_A \models p$, then it must be the case that $p \in A$ as A is a model of $Comp(P)$, thus a model of P_A taken as a set of clauses. This proves that if A is a model of $Comp(P) \cup LF$, then A is an answer set of P.

Now suppose A is an answer set of P, then clearly A is a model of $Comp(P)$. We need to show that A is also a model of LF. We prove this by contradiction. Suppose L is a loop in P, and $LF(L)$ its loop formula. Suppose $R^-(L)$ in P is:

$$r_1 : p_1 \leftarrow G_1, G_1',$$
$$\ldots$$
$$r_n : p_n \leftarrow G_n, G_n',$$

where G_i is a set of atoms and G_i' a set of negative literals. If A does not satisfy $LF(L)$, then there must be a $p \in L$ such that $p \in A$, and for each rule r_i in $R^-(L)$, $A \not\models G_i \wedge G_i'$. Now if the following set of rules is the Gelfond–Lifschitz reduct of $R^-(L)$ on A:

$$r_{a_1}: p_{a_1} \leftarrow G_{a_1},$$
$$\ldots$$
$$r_{a_k}: p_{a_k} \leftarrow G_{a_k},$$

$1 \leqslant a_i \leqslant n$, then for each $1 \leqslant i \leqslant k$, $A \not\models G_{a_i}$. Since $p \in A$, $Cons(P_A) \models p$. So there must be a sequence of rules in P_A:

$$r''_1: q_1 \leftarrow Q_1,$$
$$\ldots$$
$$r''_u: q_u \leftarrow Q_u$$

s.t. $q_u = p$, $Q_1 = \emptyset$, and for each $1 < i \leqslant u$, $Q_i \subseteq \{q_1, \ldots, q_{i-1}\}$. In this sequence, there must be a v, $1 \leqslant v \leqslant u$, s.t. $\{q_1, \ldots, q_{v-1}\} \cap L = \emptyset$ and $q_v \in L$. Since $Q_v \subseteq \{q_1, \ldots, q_{v-1}\}$, $Q_v \cap L = \emptyset$. So r''_v must be a reduct of a rule in $R^-(L)$, i.e., for some $1 \leqslant i \leqslant k$, $r''_v = r_{a_i}$. But this is a contradiction as A must satisfy the body of r''_v but not that of r_{a_i}. This proves that if A is an answer set of P, then A is a model of $Comp(P) \cup LF$. □

The proof of the theorem also shows the following result:

Proposition 1. *If S is a model of $Comp(P)$, then S is an answer set of P iff there is a $P' \subseteq P$ such that P' does not have any loops, S satisfies the body of every rule in P', and there is a rule in P' for every atom in S.*

4. Computing loops

By Theorem 1, a straightforward approach of using SAT solvers to compute the answer sets of a logic program is to first compute all loop formulas, add them to its completion, and call a SAT solver. Unfortunately this may not be practical as there may be an exponential number of loops in a logic program. For an example, consider the following program by Niemelä [18] for finding Hamiltonian cycles of a graph:

r_1: $hc(V1, V2) \leftarrow arc(V1, V2),$ not $otherroute(V1, V2)$.

r_2: $otherroute(V1, V2) \leftarrow arc(V1, V2),\ arc(V1, V3),$
$hc(V1, V3),\ V2 \neq V3$.

r_3: $otherroute(V1, V2) \leftarrow arc(V1, V2),\ arc(V3, V2),$
$hc(V3, V2),\ V1 \neq V3$.

r_4: $reached(V2) \leftarrow arc(V1, V2),\ hc(V1, V2),\ reached(V1),$
not $initialnode(V1)$.

r_5: $reached(V2) \leftarrow arc(V1, V2),\ hc(V1, V2),\ initialnode(V1)$.
$initialnode(0)$.

r_6: $\leftarrow vertex(V),$ not $reached(V)$.

For complete graphs, rule r_4, when fully instantiated, will give rise to a loop for every set of vertices.

Thus, it seems more practical to add loop formulas one by one, selectively. This motivates the following procedure.

Procedure 1.

1. Let T be $Comp(P)$.
2. Find a model M of T. If there is no such model, then terminate with failure.
3. If M is an answer set, then exit with it (go back to step 2 when more than one answer sets are needed).
4. If M is not an answer set, then find a loop L such that its loop formula Φ_L is not satisfied by M.
5. Let T be $T \cup \{\Phi_L\}$ and go back to step 2.

By Theorem 1, this procedure is sound and complete, provided a sound and complete SAT solver is used. The two key questions regarding this procedure are as follows:

1. Are SAT solvers suitable for this purpose?
2. How to find a loop such that its loop formula is not satisfied by the current model in step 4?

Question 1 is empirical. For programs without loops, experiments done by Babovich et al. [2] and Huang et al. [8] pointed to a positive answer to this question. Our experiments on logic programs, including those with loops, seem to confirm this.

Question 2 is really the key to this procedure. If a set M of atoms is a model of $Comp(P)$ but not an answer set of P, then by Theorem 1, there must be a loop whose loop formula is not satisfied by M. But how hard it is to find such a loop? As it turns out, such a loop can be found in polynomial time. The key lies in the following set:

$$M^- = M - Cons(P_M).$$

(Recall that P_M is the Gelfond–Lifschitz reduct of P on M, and $Cons(P_M)$ is the set of consequences of P_M.)

Lemma 1. *If M is a model of $Comp(P)$, then $Cons(P_M) \subseteq M$.*

Proof. We prove this by induction. Suppose $Cons(P_M)$ is not empty, then for any $p \in Cons(P_M)$, there must be a sequence of rules in P_M:

$$p_1 \leftarrow G_1,$$
$$\cdots$$
$$p_k \leftarrow G_k$$

s.t. $p_k = p$, $G_1 = \emptyset$, and for $1 < i \leqslant k$, $G_i \subseteq \{p_1, \ldots, p_{i-1}\}$.

Let $S_0 = \{p \mid p \leftarrow . \in P_M\}$. Since $Cons(P_M) \neq \emptyset$, $S_0 \neq \emptyset$. Now for any $p \in S_0$, there must be a corresponding rule $r : p \leftarrow G^-$ in P, s.t. G^- is either empty or a set of negative literals which are satisfied by M. Since M is a model of P, $p \in M$. So we have $S_0 \subseteq M$. Suppose we have S_k, and $S_k \subseteq M$. Construct S_{k+1} as: $S_{k+1} = S_k \cup \{p \mid r' : p \leftarrow G_{ki} \in P_M$ and $G_{ki} \subseteq S_k\}$. For any $p \in S_{k+1} - S_k$, let the rule about p in P_M be $r' : p \leftarrow G_{ki}$ and the one in P be $r : p \leftarrow G_{ki}, G_{ki}^-$, since $G_{ki} \subseteq S_k$ and $S_k \subseteq M$, $G_{ki} \subseteq M$. Considering that r' is the Gelfond–Lifschitz reduct of r, G_{ki}^- is satisfied by M, so M satisfies the body of r. Since M is a model of $Comp(P)$, $p \in M$. So we have $S_{k+1} \subseteq M$.

Let $S = \bigcup_{i=0,1,\ldots} S_i$, then $S = Cons(P_M)$. So we have $Cons(P_M) \subseteq M$. □

Lemma 2. *For any $p \in M^-$, there must be a rule $p \leftarrow G$ about p in P_M such that G contains some $q \in M^-$. Furthermore, any such rule about p in P_M must have this property, i.e., its body contains an atom in M^-.*

Proof. Since p is in M, and M satisfies $Comp(P)$, there must be a rule about p in P whose body is satisfied by M. If G is the positive literals in the body of this rule, then $p \leftarrow G$ must be in P_M. Now G must have an atom in M^-, otherwise, they must be all in $Cons(P_M)$, which means that p must be in $Cons(P_M)$ as well, a contradiction with $p \in M^-$. □

From this lemma, the following proposition follows easily.

Proposition 2. *There is at least one loop L in P such that $L \subseteq M^-$. Furthermore, for any $p \in M^-$, there must be a loop $L \subseteq M^-$ such that for some $q \in L$, there is a (directed) path from p to q in G_P.*

Proof. Let $p \in M^-$, then by Lemma 2 and the construction of Δ, there must be a $q \in M^-$ such that (p, q) is an arc in Δ. Since p is any node, and Δ has only finite number of nodes, so this must lead to a cycle, thus a strongly connected component reachable from p. □

Definition 3. Let P be a program, and G_P its positive dependency graph. Let M be a model of $Comp(P)$. We say that a loop L of P is a *maximal loop under M* if L satisfies the following two conditions:

1. $L \subseteq M^-$, and
2. L is maximal in M^-, i.e., there is no other loop $L' \subseteq M^-$ such that $L \subset L'$.

In other words, L is a strongly connected component of the subgraph of G_P induced by M^-. A maximal loop L under M is called a *terminating* one if there does not exist another maximal loop L_1 under M such that for some $p \in L$ and $q \in L_1$, there is a path from p to q such that all vertices in the path are in M^-.

Theorem 2. *If M is a model of $Comp(P)$ but not an answer set of P, then there must be a terminating loop of P under M. Furthermore, M does not satisfy the loop formula of any of the terminating loops of P under M.*

Proof. By Proposition 2, there is a loop in M^-. Since M^- is finite, there must be a maximal loop under M. Suppose now there are no terminating loops. Since G_P is finite, there must be more than one maximal loops under M, and there will be a path from any maximal loop under M to any other maximal loop under M in the subgraph of G_P induced by M^-. This means that the union of all these maximal loops is also a maximal loop under M, a contradiction with the fact that there are more than one maximal loops. So there must be a terminating loop under M.

Now let α be a terminating loop on M. Its loop formula in P is of the form:

$$(\neg G_1 \wedge \cdots \wedge \neg G_n) \supset (\neg p_1 \wedge \cdots \wedge \neg p_k),$$

where p_i's are atoms in α, and G_i's are the bodies of rules in $R^-(\alpha, P)$. Since $\alpha \subseteq M^- \subseteq M$, to show that M does not satisfy this formula, we only need to show that for any $p \in \alpha$, and any rule $p \leftarrow G$ that is in $R^-(\alpha)$, $M \models \neg G$. Suppose $M \models G$, then $p \leftarrow G^+$ is in P_M, where G^+ is the set of positive literals in G. Now there are two cases: Case 1: G^+ does not contain any atom in M^-, in this case G^+ must be in $Cons(P_M)$, this means that p must be in $Cons(P_M)$ as well, a contradiction with our assumption that $p \in M^-$; Case 2: G^+ contains a $q \in M^-$, in this case, by Proposition 2 there must be a loop, thus a maximal loop β under M such that there is a path from q to an atom in β. This means that there is a path from p to an atom in β as well. Since $p \leftarrow G$ is not in $R^+(\alpha)$, q is not in α, so α must be different from β, a contradiction with the assumption that α is a terminating and maximal loop. □

Thus given a model M of $Comp(P)$, if it is not an answer set, then the problem of finding a loop whose loop formula is not satisfied by M can be reduced to the problem of finding a terminating loop under M. Notice that a terminating loop under M is a strongly connected component in the subgraph of G_P induced by M^-, and that the strongly connected components of a graph and their dependency chains can be computed in $O(n+e)$ time [22], where n is the number of nodes and e the number of arcs in the graph. So a terminating loop under M can be found in $O(m+k)$ time, where m is the size of M^-, and k the number of arcs in the subgraph of G_P induced by M^-. Normally M^- is quite small compared to the number of atoms in P. Once we have identified a loop, its loop formula can be computed in time similar to that for computing the completion of a program.

5. ASSAT(X)

ASSAT(X), where X is a SAT solver, is an implemented system based on Procedure 1 in the last section:

ASSAT(X)—X a SAT solver

1. Instantiate a given program using lparse, the grounding system of Smodels.
2. Do some simple simplifications to the instantiated program, such as deleting all rules $p \leftarrow G$ such that $p \in G$.

3. Compute the completion of the resulting program and convert it to clauses.[2]
4. **Repeat**
   (a) Find a model M of the clauses using X.
   (b) If no such M exists, then exit with failure.
   (c) Compute $M^- = M - Cons(P_M)$.
   (d) If $M^- = \emptyset$, then return with M for in this case it is an answer set.
   (e) Compute all maximal loops under M.
   (f) For each of these loops, compute its loop formula, convert it to clauses, and add them to the clausal set.

Notice that in the procedure above, when M is not an answer set, we will add the loop formula of every maximal loop under M to the current clausal set, instead of adding just the loop formula of one of the terminating loops if we want to follow Procedure 1 strictly using Theorem 2. The procedure above has the advantage of not having to check whether a loop is terminating. This is a feasible strategy as we have found from our experiments that there are usually not many such maximal loops.

6. Some experimental results

We experimented on a variety of benchmark domains. We report some of our results here for the following three domains: graph coloring, the blocks world planning, and Hamiltonian Circuit (HC) domains. More experimental data can be found on our web site http://www.cs.ust.hk/assat.

For these three domains, we used Niemelä's [18] logic program encodings that can be downloaded from Smodels' web site.[3] Among the three domains, only HC requires adding loop formulas to program completions. The graph coloring programs are always loop-free, and while the logic programs for the blocks world planning problems have loops, Babovich et al. [2] showed that all models of the programs' completions are answer sets. For graph coloring and the blocks world planning, our results confirmed the findings of [8], but we did it with many more and much larger instances.

For our system, ASSAT 2.0, we tried the following SAT solvers: Chaff2 (Mar 23, 2001 version) [17], Walksat 41 [19], GRASP (Feb, 2000 version) [20], Satz 215.2 [13], and SATO 4.1 (Zhang) [25]. In our experiments, we compare the performance of our system with Smodels version 2.27 [21] and DLV (May 16, 2003 version) [12] on the problem of computing one answer set of a normal logic program. More precisely, given a logic program, we measure the performance of each system by how fast it returns the first answer set of the program or reports that the program has no answer set.

[2] When converting a program's completion and loop formulas to clauses, $O(r)$ number of extra variables may have to be used in order to avoid combinatorial explosion, where r is the number of rules. So far, we do not find this to be a problem. For instance, for graph coloring and HC problems, no extra variables are needed. Notice that the approach in [3] also needs a program's completion as the base case.

[3] http://www.tcs.hut.fi/Software/smodels/.

For Smodels and ASSAT, we used lparse 1.0.13, the grounding module of Smodels, to ground a logic program (DLV has its own built-in grounding routine). We have noticed that for both Smodels and ASSAT, their behaviors are sometimes influenced by the parameters that one calls lparse with. For our experiments, we use "lparse -d none" which seemed to optimize the performance of both Smodels and ASSAT.

Our experiments were done on Sun Ultra 5 machines with 256M memory running Solaris. The reported times are in CPU seconds as reported by Unix "time" command, and include all pre-processing and post-processing time, if any. For instance, for ASSAT, they include the time for grounding the input program, for computing the completion and converting it to clauses, for computing loop formulas, for checking if a model returned by the SAT solver is an answer set, as well as the time for the SAT solver to return a model. To make the experiments feasible, we set a 2-hour cut off limit. So in the following tables, if a cell is marked by "–", it means that the system in question did not return after it had used up 2 hours of the CPU time. Also in the following tables, if a system is not included, that means it is not competitive on the problems.

We want to emphasize here that the bulk of the experiments here were done using Niemelä's early encodings for these benchmark domains. They are not the optimal ones for Smodels, and certainly not for DLV. As one of the referees for an extended abstract of this paper that we submitted to AAAI'02 pointed out, DLV is specialized in disjunctive logic programs. There are encodings of graph coloring and HC problems in disjunctive logic programs for which DLV will run faster. The newest version of Smodels also has some special constructs such as mGn that can be used to encode the problems in a more efficient way. One can also think of some encodings that are better suited for ASSAT. It is an interesting question as how all these systems will fare with each other with each using its own "best possible" encodings. While no one knows what the best encoding for each of the systems is, nonetheless for the HC domain, we shall also compare ASSAT using Niemelä's early encoding with DLV using the disjunctive encoding found on the DLV's web page, and Smodels using an encoding that uses mGn constructs.

6.1. The blocks world planning domain

This is a domain where a planning problem in the blocks world is solved by a sequence of logic programs such that every answer set of the nth program in the sequence corresponds to a plan of the original planning problem with exactly n steps. So if a planning problem requires n steps, then in the sequence of logic programs corresponding to the planning problem, the first $n-1$ logic programs do not have any answer sets. For details, we refer the read to Niemelä [18].

For this domain, we tested the systems on 16 planning problems, ranging from one with 15 blocks to one with 32 blocks. Table 1 contains some run time data on these instances. In the tables, *atoms* and *rules* are the number of atoms and rules, respectively, in the grounded logic program returned by lparse,[4] and *steps* is the number of steps in the

[4] As we have mentioned, DLV has its own grounding module. There is no information available about the size of the grounded program in DLV.

Table 1
The blocks world planning domain

Problem	Steps	Atoms	Rules	Smodels	DLV	ASSAT (Chaff2)	ASSAT (Satz)	ASSAT (SATO)
bw.19	9	12202	174099	17.58	–	13.69	22.3	12.33
	10	13422	191621	57.93	150.44	15.59	32.63	14.76
bw.21	10	16216	253241	24.95	236.1	19.47	33.25	17.5
	11	17690	276387	71.86	–	22.76	49.89	21.15
bw.23	11	21026	356663	36.62	621.33	27	50.81	24.58
	12	22778	386521	2284.19	890.51	31.48	76.29	29.5
bw.25	13	28758	526631	57.47	1899.95	40.62	90.07	36.31
	14	30812	564385	–	2613.85	49.34	153.66	44.04
bw.32	17	59402	1366664	172.12	–	106.87	–	96.86
	18	62702	1442764	–	4681.02	117.84	–	114.84

Legends: bw.*n*—a problem with *n* blocks. bw.15, bw.17, and bw.19 correspond to bw-large.c, bw-large.d, and bw-large.e on Smodels' web site, respectively.

plan. Two numbers are given for *steps*, the shortest step and the one immediately before. For instance, the first two rows are about *bw*.19, a planning problem in a blocks world with 19 blocks. The shortest plan for this problem needs 10 steps. The logic program that corresponds to the first row has no answer set, meaning that there is no plan with 9 steps. The one that corresponds to the second row has an answer set, meaning that there is a plan with exactly 10 steps.

As one can see, ASSAT performed very well here. Among the SAT solvers used with ASSAT, SATO and Chaff2 performed best. ASSAT(Satz) also did very well for problems with $\leqslant 26$ blocks. After that, it suddenly degraded, perhaps because the problem sizes were too big for it to handle now.

We notice that for all problems that we had tested, if a shortest plan requires n steps, then Smodels did very well in verifying that there does not exist a plan with $n-1$ steps.

6.2. The graph coloring domain

We tested the systems on over 50 randomly generated large graphs for both 3-coloring and 4-coloring problems. Table 2 is the results for some of them for 4-coloring, and Table 3 for 3-coloring. Again ASSAT(Chaff2) was the clear winner. Smodels was more competitive on 3-coloring problems. But on 4-coloring ones, it could not return within our time limit after p10000e100000, except for p10000e2100 which is not colorable. In general, we have observed that Smodels and ASSAT(Chaff2) had similar performance on graphs which are not colorable.

6.3. The Hamiltonian Circuit (HC) domain

This is the only benchmark domain that we could find which requires adding loop formulas to program completions. We thus did some extensive testing in this domain. We tested three classes of problems: randomly generated graphs, hand-coded hard graphs, and complete graphs. All these are directed graphs that do not have any arc that goes from a

Table 2
4-coloring

Problem	Atoms	Rules	Colorable?	Smodels	DLV	ASSAT(Chaff2)
p100e570	801	3880	y	1.89	1919.61	1.35
p300e1760	2401	11840	y	9.85	–	2.05
p600e3554	4801	23816	y	38.49	72.46	3.08
p6000e35946	48001	239784	y	4722.56	–	144.06
p10000e10000	80001	200000	y	–	–	58.98
p10000e11000	80001	203996	y	–	–	58.09
p10000e21000	80001	244000	n	31.12	3975.25	22.70
p10000e22000	80001	247992	y	–	–	49.33

Legends: pnem—a graph with n nodes and m edges.
p100e570, p300e1760, p600e3554, p6000e35946 are the same as p100, p300, p600, p6000 on Smodels' web site, respectively.

Table 3
3-coloring

Problem	Atoms	Rules	Colorable?	Smodels	DLV	ASSAT(Chaff2)
p100e570	601	2610	n	1.16	1.53	1.21
p300e1760	1801	7980	n	1.8	6	1.72
p600e3554	3601	16062	n	2.88	20.57	2.56
p6000e35946	36001	161838	n	24.15	2084.58	16.71
p10000e10000	60001	120000	y	2949.31	–	28.98
p10000e11000	60001	122997	y	2730.38	–	28.06
p10000e21000	60001	152997	n	20.65	2191.69	15.88
p10000e22000	60001	155994	?	–	–	–
p10000e23000	60001	158991	?	–	–	–

Legends: Same as in Table 2.

vertex to itself, as is usually assumed in work on HC. In this domain, we found Walksat performed surprisingly well, sometimes even better than Chaff2. However, one problem with Walksat is that it is incomplete: when it could not find an assignment, we don't know if the clauses are satisfiable or not. To address this, we invent W-C (Walksat+Chaff2): given a SAT instance, try Walksat on it first, if it does not return an assignment, then try Chaff2 on it. Another problem with Walksat is that it is a randomized system, so its performance may vary from run to run. We address this problem by running it 10 times, and taking the average. Thus in all the tables below, the data on ASSAT(W-C) are the averages over 10 runs.

Table 4 contains some statistics on 43 randomly generated Hamiltonian graphs (those with Hamiltonian Circuits). The numbers of nodes in these graphs range from 50 to 70 and numbers of arcs from 238 to 571. For those which ran out of 2 hours of CPU time, we use 7200 seconds (2 hours) in the calculation of sum, average, and standard deviation. The full set of data is available on our ASSAT web site.

Smodels could not solve 10 of them (did not return after 2 hours of CPU time), which amounts to a 23% failure rate, DLV could not solve 19 of them (44%). It is interesting to notice that compared with the other two domains, DLV fared better here. While overall it was still not as good as Smodels, there were 4 problems which Smodels could not solve but

Table 4
HC on randomly generated graphs

Problem	Smodels	DLV	ASSAT (Chaff2)	LFs	SATs	ASSAT (W-C)	LFs	SATs
SUM	74234	148705	4426	710	716	721	863	857
Average	1726.39	3458.27	102.95	16.51	16.65	17	20.09	19.94
STDEV	3060.72	3513.84	238.75	13.11	12.51	10.48	8.94	8.34
Unsolved	10	19	0			0		

Legends: SATs—number of calls to the SAT solver; LFs—number of loop formulas added totally; STDEV—standard deviation.

DLV could in a few seconds. ASSAT with both Chaff2 and W-C solved all of the problems. So far we had not run into any randomly generated graph which is Hamiltonian, either DLV or Smodels could solve it, but ASSAT could not. It is interesting to notice that the average number of calls to the SAT solver and the average number of loop formulas added are very close, both in ASSAT(Chaff2) and in ASSAT(W-C), so are the STDEVs. Indeed, we have found that for randomly generated graphs, if M is not an answer set, then often M^- is a loop by itself, i.e., M^- is the only maximal loop on M. Also the cost of ASSAT(X) is directly proportional to the number of calls made to X. One reason that ASSAT(W-C) outperformed ASSAT(Chaff2) is that walksat (W-C is really walksat here because it always returned a model for this group of graphs) is a bit "luckier" than Chaff2 in returning the "right" models. Also notice that on average, each call to Chaff2 took 6.2 seconds, and W-C 1 seconds.

We have found that it was difficult to come up with randomly generated non-Hamiltonian graphs which are hard. Most of them were really easy for all the systems and occurred when the number of edges is relatively small compared to that of vertices. There were reports (e.g., [4]) that randomly generated graphs with m close to $(\log n + \log\log n)n/2$, where n is the number of vertices and m the edges, have a 50% chance of being Hamiltonian, and was believed to be the "phase-transition" area of HC. However, Vandegriend [23] observed that for his heuristic algorithms, these graphs were very easy. Our experiments seemed to confirm this. For instance, a randomly generated graph with 70 vertices and 230 edges would have roughly 50% chance of being Hamiltonian. But for these graphs, all the systems could solve them very quickly, in a few seconds. We are still puzzled by this. For the systems that we have tested at least, the harder instances seem to be those graphs with more arcs, thus are likely to be Hamiltonian.

We did stumble on two graphs which are not Hamiltonian, but none of the systems that we tested (Smodels, DLV, ASSAT(X)) could solve them. They are not Hamiltonian for the obvious reason that some of the vertices in them do not have an arc going out. They both have 60 vertices, and one has 348 arcs and the other 358. The completions of the logic programs corresponding to them, when converted to clauses, have only about 720 variables and 4500 clauses. But none of the SAT solvers that we tested could tell us whether they are satisfiable.[5]

[5] Following the requests of some SAT researchers, we generated some more hard instances like these. They can be found on our web site, http://www.cs.ust.hk/assat/hardsat/.

Table 5
Hand-coded graphs

Graph	Vertex /arc	HC ?	Smodels	DLV	ASSAT (Chaff2)	LFs	SATs	ASSAT (W-C)	LFs	SATs
2xp30	60/316	n	1.28	1.39	1.57	2	2	4.69	2.2	2
2xp30.1	60/318	y	2.07	–	205.9	254	99	732.05	369.4	149.3
2xp30.2	60/318	y	–	–	70.41	213	128	1041.6	378.1	222.6
2xp30.3	60/318	y	–	–	70.54	213	128	774.45	322.5	187.5
2xp30.4	60/318	n	–	–	–			5983.48	18	12.7
4xp20	80/392	n	1.27	1.51	1.54	5	2	4.52	4.1	2
4xp20.1	80/395	n	–	–	8.3	4	2	18.76	4.1	2
4xp20.2	80/396	y	2.43	–	39.08	248	95	471.63	333.8	126.7
4xp20.3	80/396	n	1.24	1.79	9.12	9	5	19.84	17.7	13.2

Legends: SATs—number of calls to a SAT solver; LFs—number of loop formulas added totally; 2xp30—2 copies of p30; 2xp30.i—2xp30 + two new arcs; 4xp20—4 copies of p20; 4xp20.i—4xp40 + 3–4 new arcs.

More interesting are some hand-coded hard problems. One strategy is to take the union of several copies of a small graph, and then add some arcs that connect these components. To experiment with this strategy, we took as bases p30 (a graph with 30 vertices) and p20 (a graph with 20 vertices), both taken from Smodels' distribution. The results are shown in Table 5.

Notice that SATs and LFs for ASSAT(W-C) are in general larger than the corresponding ones for ASSAT(Chaff2) this time. It is clear that ASSAT(Chaff2) performed the best here. It is interesting to notice that some of these graphs are also very hard for specialized heuristic search algorithm. For instance, for graph 2xp30.4, the HC algorithm (no.559, written in Fortran) in ACM Collection of Algorithms did not return after running for more than 60 hours. ASSAT(Chaff2) could not solve it using lparse 1.0.13 in 2 hours. But with lparse 0.99.43, ASSAT(Chaff2) solved it in about 1.5 hours.

Complete graphs are of special interest for ASSAT because when instantiated on these graphs, Niemelä's logic program for HC has an exponential number of loops. So one would expect that these graphs, while trivial for heuristic search algorithms, could be hard for ASSAT. Our experiments confirmed this. But interestingly, these graphs are also very hard for Smodels and DLV. The results are given in Table 6.

Complete graphs are difficult using Niemelä's encoding also because of the sheer sizes of the programs they produce. For instance, after grounding, the complete graph with 50 nodes (c50) produces a program with about 5000 atoms and 240K rules, and needs 4.5M to store it in a file. For c60, the number of atoms is about 7K and rules about 420K.

We also compared ASSAT with an implementation[6] of Ben-Eliyahu and Dechter's translation [3]. As we mentioned earlier, their translation needs n^2 extra variables, and these extra variables seemed to exert a heavy toll on current SAT solvers. For complete graphs, it could only handle those up to 30 vertices using Chaff2. It caused Chaff2 to run into bus error after running for over 2 hours on graph 2xp30. Perhaps more importantly,

[6] Done by Jicheng Zhao.

Table 6
HC on complete graphs

Complete graph	Smodels	DLV	ASSAT (Chaff2)	LFs	SATs	ASSAT (W-C)	LFs	SATs
c10	1.14	1.26	1.52	2	3	1.26	3.3	4.3
c20	5.15	8.12	5.92	11	12	3.02	3.3	4.3
c30	29.48	58.69	6.24	2	3	16.43	8.9	9.9
c40	115.43	242.72	11.2	1	2	77.13	18.3	19.3
c50	414.62	850.12	21.07	1	2	399.04	54	55
c60	1091.13	2075.26	275.5	37	38	1488.31	76.8	77.8
c70	2598.65	4831.81	1170.44	94	95	2281.4	87.6	88.6
c80	5173.11	–	5905.4	249	250	4396.5	116.4	117.4
c90	–	–	208.84	4	5	2844.21	57.9	58.7
c100	–	–	2732.49	66	67	3162.76	46.875	47.125

Legends: SATs—number of calls to the SAT solvers; LFs—number of loop formulas added totally; cN—a complete graph with N vertices.

Table 7
Statistics on random graphs using different encodings

Problem	Smodels (new)	Smodels	DLV (new)	DLV	ASSAT
SUM	81807	74234	122802	148705	4426
Average	1902.49	1726.39	2855.87	3458.27	102.95
STDEV	3166.16	3060.72	3453.47	3513.84	238.75
Unsolved	11	10	16	19	0

Smodels (new) refers to the performance of Smodels using the new encoding, similarly for DLV (new). The entries for Smodels, DLV and ASSAT(Chaff2) are the same as in Table 4.

while Walksat was very effective on HC problems using our translation, it was totally ineffective with their translation as it failed to find an HC on even some of the simplest graphs such as p20. We speculate that the reason could be that the extra variables somehow confuse Walksat and make its local hill-climbing strategy ineffective.

So far we have compared the three systems: ASSAT, Smodels, and DLV, using the same encoding for each of the benchmark problems. As we mentioned earlier, an interesting question is how these systems will compare when they use different encodings. To find out, we did some experiments using the HC benchmark domain. We chose for Smodels a recent short encoding using cardinality constraints of the form mGn,[7] and for DLV an encoding found on its web site that uses disjunctive rules.[8] These two programs are given in Appendix A. For ASSAT, since it currently cannot handle cardinality constraints and disjunctive rule, we continued to use Niemelä's earlier encoding.

Table 7 contains some statistics of run time data on the same set of randomly generated HC problems as in Table 4. Again, the full set of data is available on the ASSAT web site. For ease of comparison, we also repeated some of the data from Table 4. As one can see, with its new encoding, DLV performed a bit better, though overall it was still not as good

[7] Downloaded from http://www.cs.engr.uky.edu/ai/benchmark-suite/ham-cyc.sm.
[8] Downloaded from http://www.dbai.tuwien.ac.at/proj/dlv/examples/hamcycle.

as Smodels and ASSAT. Surprisingly though for Smodels, its overall performance seemed a little worse using the new encoding. For instance, earlier, there were 10 problems it could not solve within our 2-hour time limit. Now there are 11. However, a big advantage of the new encoding is that when grounded, it yields a much smaller program. This seemed to have a big impact on the HC problems on complete graphs. For instance, for the complete graph with 80 vertices, Smodels with the new encoding returned a model in 100 seconds, and for the complete graphs with 100 vertices, it returned in 280 seconds. In comparison, for the complete graph with 100 vertices, using the old encoding, Smodels could not return in 2 hours. This also seems to confirm that for HC problems, one of the main reasons that the trivial complete graphs are hard for all the systems is because of the sizes of the grounded programs.

7. Conclusions

We have proposed a new translation from logic programs to propositional theories. Compared with the one in [3], ours has the advantage that it does not use any extra variables. We believe it is also more intuitive and simpler, thus easier to understand. However, in the worst case, it requires computing an exponential number of loop formulas. To address this problem, we have proposed an approach that adds loop formulas a few at a time, selectively. We have implemented a system called ASSAT based on this approach, and experimented it on some benchmark domains using various SAT solvers. While we were satisfied that so far our experimental results showed a clear edge of ASSAT over Smodels and DLV, we want to emphasize that the real advantage that we can see of ASSAT over specialized answer set generators lies in its ability to make use of the best and a variety of SAT solvers as they become available. For instance, with Chaff, we were able to run much larger problems than using others like SATO, and while Chaff has been consistently good on all of the benchmark problems that we have tested, other SAT solvers, like the randomized incomplete SAT solver walksat, performed surprisingly good on HC problems.

We also want to emphasize that by no means do we take this work to imply that specialized stable model generators such as Smodels are not needed anymore. For one thing, so far we have only considered the problem of finding one answer set of a logic program. It is not clear what would happen if we want to look for all the answer sets. More importantly, we hope this work, especially our new translation of logic programs to propositional logic, will lead to a cross fertilization between SAT solvers and specialized answer set solvers that will benefit both areas.

Since this work was first published [15], it has been extended in several directions. Lierler and Maratea [14] extended it to handle so-called cardinality constraints and choice rules in their second version of CMODELS. Lee and Lifschitz [10] extended our Theorem 1 and the notion of loops and loop formulas to disjunctive logic programs, Lee [9] showed that a similar result holds for McCain and Turner's causal logic [16], and finally Lee and Lin [11] showed that the idea works for propositional circumscription as well.

Finally, ASSAT as well as the experimental results reported in this paper can be found at http://www.cs.ust.hk/assat/.

Acknowledgements

We thank Jia-Huai You and Jicheng Zhao for many useful discussions about the topics and their comments on earlier versions of this paper. We especially thank Jicheng for suggesting to define loops as sets of atoms rather than sets of rules as we initially did, and for implementing a version of Ben-Eliyahu and Dechter's algorithm. We also thank Jia-Huai for making us aware of Chaff. Without it, we would not be able to make many claims that we made in the paper. He also suggested to try complete graphs for HC. We thank the reviewers of AAAI-2002 and this special issue of AIJ for this paper for their insightful and useful comments about earlier versions of this paper, which led to many improvements on the presentation of this paper. Any remaining errors in this paper are of course the authors'.

This work was supported in part by the Research Grants Council of Hong Kong under Competitive Earmarked Research Grants HKUST6061/00E and HKUST6205/02E.

Appendix A

A.1. A logic program encoding of HC using cardinality constraints

Below is the "new" encoding for HC that we used in Table 7 for Smodels. This program contains choice rules, and cardinality constrains. It was modified from the program named http://www.cs.engr.uky.edu/ai/benchmark-suite/ham-cyc.sm on the University of Kentucky ASP benchmark site[9] by renaming some atoms to be consistent with Niemelä's encoding that we used in the paper.

As usual, a graph is given by a collection of facts of the forms $vertex(X)$ and $arc(X, Y)$. One vertex v, 0 here, is chosen as the initial vertex $initialvtx(v)$.

$$\{hc(X, Y)\} :- arc(X, Y).$$
$$:- 2\{hc(X, Y) : arc(X, Y)\},\ vertex(Y).$$
$$:- 2\{hc(X, Y) : arc(X, Y)\},\ vertex(X).$$
$$:- vertex(X),\ \text{not}\, r(X).$$
$$r(Y) :- hc(X, Y),\ arc(X, Y),\ initialvtx(X).$$
$$r(Y) :- hc(X, Y),\ arc(X, Y),\ r(X),\ \text{not}\, initialvtx(X).$$
$$initialvtx(0).$$

A.2. A disjunctive logic program encoding of HC

Below is the disjunctive logic program encoding of the HC problem for directed graphs that we used in Table 7 for DLV. It was downloaded from DLV's web site.[10] Again, a graph

[9] http://www.cs.engr.uky.edu/ai/benchmarks.html.

[10] http://www.dbai.tuwien.ac.at/proj/dlv/examples/hamcycle.

is given by a collection of facts of the forms $vertex(X)$ and $arc(X, Y)$, and one vertex X, 0 here, is chosen as the initial vertex $start(X)$. In the program, $in_hc(X, Y)$ represents the fact that $arc(X, Y)$ is in the Hamiltonian Circuit to be constructed, and $out_hm(X, Y)$ the fact that the arc (X, Y) is not.

$$reached(X) :\!- in_hm(_\,, X).$$

$$in_hm(X, Y) \vee out_hm(X, Y) :\!- start(X), arc(X, Y).$$

$$in_hm(X, Y) \vee out_hm(X, Y) :\!- reached(X), arc(X, Y).$$

$$:\!- in_hm(X, Y),\ in_hm(X, Y1), Y! = Y1.$$

$$:\!- in_hm(X, Y),\ in_hm(X1, Y), X! = X1.$$

$$:\!- vertex(X),\ notreached(X).$$

$$start(0).$$

References

[1] K.R. Apt, H.A. Blair, A. Walker, Towards a theory of declarative knowledge, in: J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, Los Altos, CA, 1988, pp. 293–322.

[2] Y. Babovich, E. Erdem, V. Lifschitz, Fages' theorem and answer set programming, in: Proc. of NMR-2000, Breckenridge, CO, 2000.

[3] R. Ben-Eliyahu, R. Dechter, Propositional semantics for disjunctive logic programs, Ann. Math. Artificial Intelligence 12 (1996) 53–87.

[4] P. Cheeseman, B. Kanefsky, W.M. Taylor, Where the really hard problems are, in: Proc. IJCAI-91, Sydney, Australia, 1991, pp. 331–337.

[5] K.L. Clark, Negation as failure, in: H. Gallaire, J. Minker (Eds.), Logics and Databases, Plenum Press, New York, 1978, pp. 293–322.

[6] F. Fages, Consistency of Clark's completion and existence of stable of stable models, J. Methods Logic Comput. Sci. 1 (1994) 51–60.

[7] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: Proc. Fifth International Conference and Symposium on Logic Programming, Seattle, WA, 1988, pp. 1070–1080.

[8] G.-S. Huang, X. Jia, C.-J. Liau, J.-H. You, Two-literal logic programs and satisfiability representation of stable models: a comparison, in: Proc. 15th Canadian Conference on AI, Calgary, AB, in: Lecture Notes in Comput. Sci., vol. 2338, Springer, Berlin, 2002, pp. 119–131.

[9] J. Lee, Nondefinite vs. definite causal theories, in: Proc. 7th Internat. Conference on Logic Programming and Nonmonotonic Reasoning, Fort Lauderdale, FL, in: Lecture Notes in Comput. Sci., vol. 2923, Springer, Berlin, 2004, pp. 141–153.

[10] J. Lee, V. Lifschitz, Loop formulas for disjunctive logic programs, in: Proceedings of ICLP-2003, Mumbai, India, in: Lecture Notes in Comput. Sci., vol. 2916, Springer, Berlin, 2003, pp. 451–465.

[11] J. Lee, F. Lin, Loop formulas for circumscription, in: Proc. AAAI-2004, San Jose, CA, AAAI Press, Menlo Park, CA, 2004, in press.

[12] N. Leone, et al., DLV: a disjunctive datalog system, release 2001-6-11. At http://www.dbai.tuwien.ac.at/proj/dlv/, 2001.

[13] C.M. Li, Anbulagan, Heuristics based on unit propagation for satisfiability problems, in: Proc. IJCAI-97, Nagoya, Japan, 1997, pp. 366–371.

[14] Y. Lierler, M. Maratea, Cmodels-2: SAT-based answer set solver enhanced to non-tight programs, in: Proc. of LPNMR-2004, Fort Lauderdale, FL, in: Lecture Notes in Comput. Sci., vol. 2923, Springer, Berlin, 2004, pp. 346–350.

[15] F. Lin, Y. Zhao, ASSAT: Computing answer sets of a logic program by sat solvers, in: Proc. AAAI-2002, Edmonton, AB, AAAI Press, Menlo Park, CA, 2002, pp. 112–118.
[16] N. McCain, H. Turner, Causal theories of action and change, in: Proc. AAAI-97, Providence, RI, AAAI Press, Menlo Park, CA, 1997, pp. 460–465.
[17] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: engineering an efficient SAT solver, in: Proc. 39th Design Automation Conference, Las Vegas, NV, 2001, pp. 530–535.
[18] I. Niemelä, Logic programs with stable model semantics as a constraint programming paradigm, Ann. Math. Artificial Intelligence 25 (3–4) (1999) 241–273.
[19] B. Selman, H.A. Kautz, B. Cohen, Noise strategies for improving local search, in: Proc. AAAI-94, Seattle, WA, AAAI Press, Menlo Park, CA, 1994, pp. 337–343.
[20] J.P.M. Silva, K.A. Sakallah, GRASP—A new search algorithm for satisfiability, in: Proceedings of the International Conference on Computer-Aided Design (ICCAD'96), San Jose, CA, 1996, pp. 220–227.
[21] P. Simons, Smodels: A system for computing the stable models of logic programs, version 2.25. At http://www.tcs.hut.fi/Software/smodels/, 2000.
[22] R. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (1972) 146–160.
[23] B. Vandegriend, Finding Hamiltonian Cycles: algorithms, graphs and performance, MSc Thesis, Department of Computer Science, University of Alberta, 1998.
[24] J. You, R. Cartwright, M. Li, Iterative belief revision in extended logic programs, Theoret. Comput. Sci. 170 (1996).
[25] H. Zhang, M.E. Stickel, Implementing the Davis–Putnam method, J. Automat. Reasoning 24 (1/2) (2000) 277–296.

ELSEVIER

Artificial Intelligence 140 (2002) 175–205

Artificial Intelligence

www.elsevier.com/locate/artint

Abduction in logic programming: A new definition and an abductive procedure based on rewriting ☆

Fangzhen Lin [a,*], Jia-Huai You [b]

[a] *Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong*

[b] *Department of Computing Science, University of Alberta, Edmonton, AB, Canada T6G 2E8*

Received 5 September 2001

Abstract

A long outstanding problem for abduction in logic programming has been on how minimality might be defined. Without minimality, an abductive procedure is often required to generate exponentially many subsumed explanations for a given observation. In this paper, we propose a new definition of abduction in logic programming where the set of minimal explanations can be viewed as a succinct representation of the set of all explanations. We then propose an abductive procedure where the problem of generating explanations is formalized as rewriting with confluent and terminating rewrite systems. We show that these rewrite systems are 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 abduction in logic programming to a problem in reasoning about actions and provide some experimental results.
© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Abduction; Answer set programming; Rewrite systems

1. Introduction

Abductive reasoning subscribes to reasoning processes where explanatory hypotheses are formed and evaluated with respect to a knowledge base and an observation. Many

☆ This paper is extended from the short version that appeared in the proceedings of IJCAI-2001, Seattle, WA, USA.

* Corresponding author.
E-mail address: flin@cs.ust.hk (F. Lin).

0004-3702/02/$ – see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S0004-3702(02)00227-8

intelligent tasks, including medical diagnosis, fault diagnosis, scientific discovery, legal reasoning, and natural language understanding, have been characterized as abduction.

In its most general form, the problem of abduction is this: given a background theory T and an observation q to explain, find an explanation theory Π such that $\Pi \cup T \models q$. Normally, we also 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 [18,28,30]) to be a minimal set of literals over A such that $T \cup S \models q$ and $T \cup S$ is consistent.

Abductive reasoning may be carried out in non-classic logics as well. Logic programming with stable models or answer sets as the underlying semantics has been considered particularly appealing for abduction, due to its applications in solving constraint satisfaction and other combinatorial problems, in expressing the frame axioms, in reasoning with actions and causality, and in representing the history of a plan [21,26,27].

In the context of logic programming, abduction has been investigated from both proof-theoretic and model-theoretic perspectives (e.g., [7,14–16,34]). One of the most followed definitions of abduction in logic programming is that of Kakas and Mancarella's generalized stable model semantics [15]. Given a logic program P, a set A of atoms standing for abducibles, and a query q, Kakas and Mancarella defined an abductive explanation S to be a subset of A such that there is an answer set (also called a stable model) 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 $P = \{q \leftarrow a.\ r \leftarrow b.\}$, 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, and in these exponentially many explanations, only a is relevant.

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:

$$canCross \leftarrow boat, \text{not}\, leaking.$$

$$canCross \leftarrow boat, leaking, hasBucket.$$

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 simple observation that for a set of literals to be an explanation, all of its possible extensions must also be an explanation will be the basis for our new definition of abduction in logic programming, given in Section 3.

With a new definition of abductive explanations in hands, we next address the computational problem of how to generate these explanations. To simplify our presentation, in Section 4, we shall consider first the special case when the set of abducibles is empty. In this case, abduction becomes query answering. Briefly speaking, given a logic program, we introduce a rewrite system consisting of its Clark completion as rewrite rules for literal rewriting, and formula transformations as simplification rules, along with two loop rules to handle loops. It turns out that rewriting by a rewrite system of this kind always terminates at a unique formula, independent of any order of rewriting. In the literature of rewrite systems (see, e.g., [5,13]), this latter property is called the confluence property, and confluent and terminating systems are called canonical systems. A canonical system guarantees the termination at a unique expression independent of the order of rewriting. It is interesting to remark here that we could implement a form of nonmonotonic reasoning by a rewriting system, the two areas of research that had little connection previously. We show that these rewrite systems are sound and complete under the partial stable model semantics, in the sense that for any query, if it is written into *True*, then there must be a partial stable model containing this query, and if it is written into *False*, then there cannot be any partial stable model containing it. Since stable models are special cases of partial stable models, this means that if a query is written into *False*, there cannot be any stable model containing it.

However, if it is written into *True*, in general, it could still happen that there may not be any stable model containing it. But our rewrite systems are of such nature that when a query is written into *True*, there will be a context, which is a set of literals, associated with the rewriting. To see whether there will be a stable model containing this query, one then only has to check whether this context can be extended to a stable model, a task that is normally much easier than finding a stable model from scratch. There is a special case, however, when the given propositional logic program is finite and so-called *odd-loop* free. For these programs, partial stable models coincide with stable models. Thus our rewrite systems are also sound and complete for these programs.

Then in Section 5, we extend the system to logic programs with abducibles, and again show that it is sound and complete under the partial stable model semantics. Again this implies that for any query, if a program is odd-loop free, the rewriting system generates a set of explanations that "covers" all possible explanations of the query. In the general case, the rewriting system generates an approximation of such a cover. Section 6 compares our rewriting system with other abductive procedures. In particular, we will see why SLDNF-like procedures cannot be adopted in a simple way for the kind of answer set programming advocated in this paper. The rewriting system presented in this paper has been implemented in Prolog. In Section 7 we discuss an application of our system to reasoning about actions and present some experimental results using our Prolog implementation.

Although our technical development is based on propositional programs, we will comment on how our rewriting framework can be used for classes of function-free programs for proving ground goals. One such class is the so-called domain restricted programs [27]. This material is given in Section 4.4.

2. Logic programming semantics

Here, we consider (*normal*) *logic programs* which are sets of rules of the form

$$a \leftarrow b_1, \ldots, b_m, \mathsf{not}\, c_1, \ldots, \mathsf{not}\, c_n$$

where $m, n \geqslant 0$, and a, b_i and c_i are atoms of a underlying propositional language L. Here an atom with "not" in front is called a *default negation*. As usual, a is called the *head* of the rule and the rest the *body* of the rule. For clarity of presentation, we may place a period at the end of a rule.

We sometimes write a rule of the above form as

$$a \leftarrow D, \mathsf{not_}C$$

where D denotes the set of positive literals in the rule and $\mathsf{not_}C$ the set of default negations. Given such D and $\mathsf{not_}C$, we use the notation, $\neg D = \{\neg\phi \mid \phi \in D\}$ and $C = \{\phi \mid \mathsf{not}\,\phi \in \mathsf{not_}C\}$.

We sometimes also write a rule as $a \leftarrow B$, with B denoting the body of the rule. In this case, for convenience, we may also write B in a formula, as in the case of computing the Clark completion of a program, which will then stand for the conjunction of all the literals in B with the default negation not replaced by the negation operator "¬" in our

propositional language. When no confusion arises, the word *literal* may refer to a classic literal as well as a default negation.

In this paper, we mainly deal with three semantics: the completion semantics [2], the stable model semantics [10], and the partial stable model semantics [29]. The stable model semantics has been generalized to the answer set semantics [11]. For the class of normal logic programs, which is the one considered in this paper, the two terminologies are interchangeable.

Given a propositional 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, $\phi \leftrightarrow F \in Comp(P)$ (F stands for falsity here);
- otherwise, $\phi \leftrightarrow B_1 \vee \cdots \vee B_n \in Comp(P)$ (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 (tautology) for B_i if B_i is empty.

We now proceed to define partial stable models, which are originally defined in [29] under 3-valued logic. It is known that they can be defined equivalently in a number of different ways. Here, we adopt the definition given in [37] that does not rely on 3-valued logic (which will be introduced briefly at the end of this section).

We define some notations. Let E be a set of (classic) literals (or E may be a conjunction of literals in syntax). E^+ and E^- denote the subsets of positive literals and negative literals in E, respectively, and $E^N = \{\mathsf{not}\,\phi \mid \neg\phi \in E\}$.

Let P be a program and ξ denote a proposition. For any set S of default negations, let

$$F_P(S) = \{\mathsf{not}\,\xi \mid P \cup S \nvdash \xi\}$$

where $\vdash$ is the standard propositional derivation relation with each default negation $\mathsf{not}\,\phi$ being treated as a named atom $\mathsf{not_}\phi$. It is easy to check that the function that applies F_P twice, denoted F_P^2, is monotonic: $S_1 \subseteq S_2$ implies $F_P^2(S_1) \subseteq F_P^2(S_2)$.

A set of literals M is a *partial stable model* of P iff it satisfies $F_P^2(M^N) = M^N$, $M^N \subseteq F_P(M^N)$, and $M^+ = \{\xi \mid P \cup M^N \vdash \xi\}$. Any atom ξ such that $\xi \notin M$ and $\neg\xi \notin M$ represents that ξ is undefined in M. The condition $M^N \subseteq F_P(M^N)$ is to guarantee consistency. For example, with $P = \{p \leftarrow \mathsf{not}\,q.\ q \leftarrow \mathsf{not}\,p.\}$, the set of default negations $S = \{\mathsf{not}\,p, \mathsf{not}\,q\}$ satisfies $F_P^2(S) = S$, but not $S \subseteq F_P(S)$.

Let M be a partial stable model. M^+ is called an *answer set* (or a *stable model*) iff $F_P(M^N) = M^N$.

In general, a program 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 the values *true* or *false* to p. However, P has a partial stable model, $M = \{q\}$, in which q is true and p is undefined. This can be seen as follows (note that $M^N = \emptyset$): $F_P(M^N) = \{\mathsf{not}\,p\}$, $F_P(\{\mathsf{not}\,p\}) = M^N$, $M^N \subseteq F_P(M^N)$, and $M^+ = \{\xi \mid P \cup M^N \vdash \xi\}$.

As our discussion sometimes refers to 3-valued logic, it is convenient to introduce it here briefly. This material may be skipped if the reader is not going to read the proofs of the soundness and completeness theorems.

In 3-valued logic [17], there are three truth values: *true*, *false*, and *undefined*, denoted $\boldsymbol{t}$, $\boldsymbol{f}$, and $\boldsymbol{u}$ respectively. An interpretation I is a consistent set of literals: for any atom ϕ in the underlying language L, ϕ is true in I if $\phi \in I$, ϕ is false in I if $\neg\phi \in I$, and ϕ is undefined otherwise. The order of the truth values is defined as: $\boldsymbol{f} < \boldsymbol{u} < \boldsymbol{t}$. The connective $\neg$ (as well as not) is defined as: $\neg\boldsymbol{t} = \boldsymbol{f}$ ($\mathsf{not}\,\boldsymbol{t} = \boldsymbol{f}$), $\neg\boldsymbol{f} = \boldsymbol{t}$ ($\mathsf{not}\,\boldsymbol{f} = \boldsymbol{t}$), and $\neg\boldsymbol{u} = \boldsymbol{u}$ ($\mathsf{not}\,\boldsymbol{u} = \boldsymbol{u}$). The truth value of a conjunction is defined as the minimum value among the truth values of literals in the conjunction whereas the truth value of a disjunction is defined as the maximum value of the literals in the disjunction. The truth value of an implication $A \leftarrow B$ is $\boldsymbol{t}$ if the truth value of A is greater than or equal to that of B, otherwise it is $\boldsymbol{f}$. Logic equivalence is defined as: $A \leftrightarrow B$ iff $(A \rightarrow B) \wedge (B \rightarrow A)$.

We say that a 3-valued interpretation I satisfies a formula if the truth value of the formula is $\boldsymbol{t}$ in I. Thus a program rule $a \leftarrow B$ is satisfied by a 3-valued interpretation I if it is satisfied as an implication. A model of a program is an interpretation in which all the program rules are satisfied. We will use $I(Q)$ to denote the truth valuation of formula Q under 3-valued interpretation I. A 3-valued interpretation I reduces to a 2-valued interpretation if for every atom $\phi \in L$, either $\phi \in I$ or $\neg\phi \in I$. As usual, we use a set of atoms to denote a 2-valued interpretation.

3. Abduction in logic programming revisited

In this section, we present a new definition of abduction in logic programming based on the answer set semantics. 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 if it is an extension that is complete.

Definition 3.1. A complete hypothesis α is said to be an explanation of q with respect to 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 3.2. 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 Section 1 about *canCross*. The following are the complete hypotheses that explain *canCross*:

[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.

$\{boat, \neg leaking, \neg hasBucket\}$,

$\{boat, \neg leaking, hasBucket\}$, $\{boat, leaking, hasBucket\}$.

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 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 is 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 3.3. A set $\mathcal{S}$ of hypotheses is said to be a *cover* of q with respect to 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 3.4. *If $\mathcal{S}$ is a cover of q, then each $\alpha \in \mathcal{S}$ must be an explanation of q.*

Proof. If α' is a completion of α, then α' must be a completion of a minimal explanation of q, so must be an explanation of q. So it follows from the definition that α is an explanation. □

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.

To that end, we first prove the following result:

Proposition 3.5. *Let α be a hypothesis, and $\mathcal{S}$ a cover of q. We have that $\alpha \models \bigvee_{\beta\in\mathcal{S}} \beta$ iff α is an explanation of q.*

Proof. Only if case: suppose $\alpha \models \bigvee_{\beta\in\mathcal{S}} \beta$. This means that any complete extension of α must be an extension of one of the explanations in $\mathcal{S}$, so an explanation of q. It follows then that α is an explanation.
If case: if α is an explanation, and α' a complete extension of it. Then α' is an explanation. So it must be an extension of a minimal explanation. Because $\mathcal{S}$ is a cover, so α' must entail one of the explanations in $\mathcal{S}$. Thus $\alpha' \models \bigvee_{\beta\in\mathcal{S}} \beta$. Since α' is an arbitrary complete extension of α, so α must entail $\bigvee_{\beta\in\mathcal{S}} \beta$ as well. □

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.

From the last proposition, we immediately have the following:

Proposition 3.6. *Let $\mathcal{S}$ be a cover of q. Then a hypothesis is a minimal explanation of q iff it is a prime implicant of $\bigvee_{\alpha\in\mathcal{S}} \alpha$.*

4. Goal rewrite systems

Having defined the notion of abductive explanations and covers of a query, we now consider the computational problem of how to actually generate them. To this end, we shall formulate a goal-oriented proof procedure based on confluent and terminating rewriting, and show that the procedure is sound and complete under the partial stable model semantics. To simplify our presentation, we shall consider first the case where there are no abducibles.

The idea of goal rewriting is simple. Given a program P, a completed definition $\phi \leftrightarrow B_1 \vee \cdots \vee B_n \in \mathit{Comp}(P)$ can be used as a rewrite rule from left to right: ϕ is rewritten to $B_1 \vee \cdots \vee B_n$, and $\neg\phi$ to $\neg B_1 \wedge \cdots \wedge \neg B_n$. We call these *literal rewriting*, and the completed definitions *program* (*rewrite*) *rules*.

In general, a *goal* (*formula*) is just a formula in our propositional language. However, in the following, all goals are assumed to be *signed*, which means that in these formulas, negation occurs only in front of atoms.[2] As we shall see, this restriction is necessary in the rewrite rules that we have for handling loops. It is easy to see that we do not lose any generality with this restriction as any goal can be equivalently transformed into a signed goal using the following rewrite rules: for any formulas Φ and Ψ,

[2] The notion of signed goals was introduced in [19] for a similar purpose.

$$\neg\neg\Phi \to \Phi$$
$$\neg(\Phi \vee \Psi) \to \neg\Phi \wedge \neg\Psi$$
$$\neg(\Phi \wedge \Psi) \to \neg\Phi \vee \neg\Psi.$$

Note that, since we are dealing with partial stable models, 3-valued logic as introduced in Section 2 will serve as the underlying logic in goal rewrite systems. For example, it can be verified easily that the two sides of each rule above are logically equivalent under 3-valued logic.

In addition to program rules from $Comp(P)$ for literal rewriting, we also have the following two types of rewrite rules:

- simplification rules to transform and simplify goals, and
- loop rules for handling loops.

We introduce these two types of rules in the next two subsections, following which goal rewrite systems are defined and their properties investigated.

4.1. Simplification rules

The simplification subsystem is formulated with a mechanism of loop handling in mind, which requires keeping track of literal sequences $g_0, \ldots, g_n$ that have been chosen for rewriting, i.e., each g_i, $0 < i \leqslant n$, is in the goal formula resulted from rewriting g_{i-1}. These sequences are what we called *rewrite chains* below, and are checked for loops. In addition to rewrite chains, we also need to keep track of the *context* when a literal is written into T (tautology). This is necessary in order to maintain the consistency of a derivation: For a conjunction to be provable, not only each conjunct needs to be proved, the contexts under which these conjuncts are proved need to be consistent. These two notions are central in formulating our goal rewrite systems, and are defined as follows:

- *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 Φ is generated in order to prove l. This ancestor-descendant relation is denoted by $l \prec l'$. A sequence $l_1 \prec \cdots \prec l_n$ is then called a *rewrite chain*, abbreviated as $l_1 \prec^+ l_n$. Notice that it is essential here that Φ be in the form of a signed goal, and that when $\neg p$ is in Φ, we have that $l \prec \neg p$ but not $l \prec p$.
- *Context*: A rewrite chain $g = g_0 \prec g_1 \prec \cdots \prec g_n = T$ records a set of literals $C = \{g_0, \ldots, g_{n-1}\}$ for proving g. We will write $T(\{g_0, \ldots, g_{n-1}\})$ 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 we have the following equivalences: $\{a \leftrightarrow \neg b \wedge \neg c,\ b \leftrightarrow q \vee \neg p\}$. We then have a rewrite sequence

$$a \to \neg b \wedge \neg c \to \neg q \wedge p \wedge \neg c.$$

For the three literals in the last goal, we have the following rewrite chains from a:

$$a \prec \neg b \prec \neg q, \quad a \prec \neg b \prec p, \quad a \prec \neg c.$$

A rewrite chain should not be confused with a *rewrite sequence*. The former describes a dependency relationship between literals for book keeping purposes whereas the latter is a sequence of rewrite steps between goal formulas.

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. Rules SR1, SR1′, SR2, SR2′, SR6, and SR6′ are about the logical equivalence between the two sides of a rule (in 2-valued as well as 3-valued logic). 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.

It can be seen intuitively that the effect of SR5 and SR5′ is similar to that of SR4. In a sequential implementation, if no inconsistent literals are ever allowed by SR5 or SR5′ in individual steps, the condition for SR4 is never met, rendering SR4 redundant. On the other hand, SR4 alone is sufficient to safeguard the consistency of generated contexts, and it is not restricted to a sequential implementation. These rules represent different ways to guarantee consistency providing flexibility for implementation.

Note that, in general, the proof-theoretic meaning of a goal formula may not be the same as the logical meaning of the formula. For example, 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 4.1. 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 beginning part of a goal.

$$
\begin{aligned}
& g \to \neg a \\
& \to (b \vee c) \wedge (\neg b \vee d) \\
& \to (q \vee \neg p \vee c) \wedge (\neg b \vee d) \\
& \to (F \vee \neg p \vee c) \wedge (\neg b \vee d) \\
& \to (\neg p \vee c) \wedge (\neg b \vee d) \\
& \to (T(C) \vee c) \wedge (\neg b \vee d) \quad \text{where } C = \{g, \neg a, b, \neg p\} \\
& \to \big(T(C) \wedge (\neg b \vee d)\big) \vee \big(c \wedge (\neg b \vee d)\big) \\
& \to \big(T(C) \wedge \neg b\big) \vee \big(T(C) \wedge d\big) \vee \big(c \wedge (\neg b \vee d)\big) \quad \% \text{ apply SR5} \\
& \to F \vee \big(T(C) \wedge d\big) \vee \big(c \wedge (\neg b \vee d)\big) \\
& \to \big(T(C) \wedge d\big) \vee \big(c \wedge (\neg b \vee d)\big) \\
& \to \big(T(C) \wedge F\big) \vee \big(c \wedge (\neg b \vee d)\big) \\
& \to F \vee \big(c \wedge (\neg b \vee d)\big) \\
& \to c \wedge (\neg b \vee d) \\
& \to F \wedge (\neg b \vee d) \\
& \to F.
\end{aligned}
$$

4.2. 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 $\{l_1, \ldots, l_n\}$ where $l_1 = l_n$, $l_n = \neg l_1$, 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 to the underlying semantics.

To understand the effect of loop rules, it is convenient to construct a *dependency graph* of a program: for each rule $a \leftarrow b_1, \ldots, b_m, \mathsf{not}\, c_1, \ldots, \mathsf{not}\, c_n$ in the program, there is a positive edge from a to each b_i, $1 \leqslant i \leqslant m$, and a negative edge from a to each c_j, $1 \leqslant j \leqslant n$.

For the problem at hand, there are only four cases of loops. When $l_1 = \neg l_n$ (or $l_n = \neg l_1$), the sign has changed from the beginning of the loop to the end. This loop yields a path from l_1 to l_1 (or from l_n to l_n) in the program's dependency graph that has an *odd* number of negative edges. So we shall call them odd loops. Odd loops must fail as one cannot prove a proposition by assuming its complement.

When $l_1 = l_n$, either every l_i, $1 < i < n$, has the same sign as those of l_1 and l_n, or not. When all l_i have the same sign, this sign is either positive or negative. They are identified as two different cases here since they must be treated differently according to the semantics. Otherwise, there is at least one l_i, $1 < i < n$, whose sign differs from those of l_1 and l_n. In this case, from the program's dependency graph, a loop is formed that has an *even* number of negative edges. In answer set programming, even loops are often used as a mechanism to generate alternative candidate answer sets.

Definition 4.2. 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*.

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. $g_n \to F$
if $g_i \prec^+ g_n$, for some $1 \leqslant i < n$, is a positive loop or an odd loop.
LR2. $g_n \to T(\{g_1, \ldots, g_n\})$
if $g_i \prec^+ g_n$, for some $1 \leqslant i < n$, is a negative loop or an even loop.

Apparently, a loop literal should always be rewritten by a loop rule. Our definition of a rewrite sequence below will ensure that for each loop literal, there is exactly one loop rule that can be applied to it.

Example 4.3. $P_1 = \{b \leftarrow \mathsf{not}\, c.\ c \leftarrow c.\}$. Below, b is proved due to a negative loop, and the proof for $\neg b$ is failed due to a positive loop:

$$b \to \neg c \to \neg c \to T(\{b, \neg c\})$$
$$\neg b \to c \to c \to F$$

$P_2 = \{d \leftarrow \mathsf{not}\, a.\ a \leftarrow \mathsf{not}\, b.\ b \leftarrow \mathsf{not}\, a.\}$. Both d and $\neg d$ are proved, due to an even loop in either case:

$$d \to \neg a \to b \to \neg a \to T(\{d, \neg a, b\})$$
$$\neg d \to a \to \neg b \to a \to T(\{\neg d, a, \neg b\})$$

$P_3 = \{a \leftarrow \mathsf{not}\, b.\ \ b \leftarrow \mathsf{not}\, b.\}$. Neither a nor $\neg a$ is proved due to an odd loop:

$$a \to \neg b \to b \to F$$
$$\neg a \to b \to \neg b \to F.$$

4.3. Goal rewrite systems and their properties

In this subsection, we show that the goal rewrite systems defined above are confluent and terminating, and they are sound and complete under the partial stable model semantics.

Consider a propositional language L that consists of a (finite or infinite) number of propositions and their negative counterparts, all of which have been called *literals*, and special symbols F and $T(C)$ for each consistent set of literals C. Programs P in the language and their completions $Comp(P)$ are defined as usual, with the exception that any T appearing in $Comp(P)$ is replaced by $T(\emptyset)$. The set of *goals* $\mathcal{Q}_L$ consists of formulas constructed inductively from literals and special symbols by $\vee$ and $\wedge$. An *initial goal*, or *given goal*, is one without special symbols.

A *rewrite sequence* is a sequence of zero or more rewrite steps $Q_0 \to \cdots \to Q_k$, denoted $Q_0 \to^* Q_k$, such that Q_0 is an initial goal, and for each $0 \leqslant i < k$, 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 Q_i or a subformula in Q_i, or
- applying a loop rule to a loop literal in Q_i.

Notice that since literal rewriting is done only on non-loop literals, once a loop is formed during a rewriting process, it has to be eliminated by a loop rule. So it is not possible to have a rewrite chain that contains more than one loop in any rewrite sequence.

Without loss of generality, we often assume that an initial goal is just a literal. In addition, we may call a subsequence $Q_i \to^* Q_k$ a rewrite sequence in the understanding that it is part of some rewrite sequence $Q_0 \to^* Q_i \to^* Q_k$ from an initial goal Q_0.

Definition 4.4. A *goal rewrite system* is a triple $\langle \mathcal{Q}_L, \mathcal{R}_P, \to \rangle$, 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 $\to$ is the set of all rewrite sequences.

Goal rewrite systems are like term rewriting systems [5] everywhere except at terminating steps: a terminating step at a subgoal may depend on the history of rewriting.

A set of rewrite sequences defines a binary relation, say R, on the set of goal formulas: $R(Q, Q')$ iff $Q \to^* Q'$. Hence, a set of rewrite sequences corresponds to a binary relation.

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.

Definition 4.5. A goal rewrite system $\langle \mathcal{Q}_L, \mathcal{R}_P, \to \rangle$ is *terminating* iff there exists no endless rewrite sequence $Q_1 \to Q_2 \to Q_3 \to \cdots$ in $\to$.

Definition 4.6. Given a goal rewrite system, a goal is called a *normal form* if it cannot be rewritten by any rewrite rule.

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. Because any literal has a program rule in $Comp(P)$, no literal will appear in any normal form. Furthermore, as the simplification system transforms a goal eventually

to a disjunctive normal form (by SR6 and SR6′), goal rewriting always terminates at either F, or $T(C_1) \vee \cdots \vee T(C_m)$ for some $m \geqslant 1$. The latter indicates one or more ways by which the given goal is proved. We therefore have

Proposition 4.7. *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 $Q_k = T(C_1) \vee \cdots \vee T(C_m)$, for some $m \geqslant 1$.*

The confluence property is less obvious.

Definition 4.8. 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 4.9. *Any goal rewrite system $\langle \mathcal{Q}_L, \mathcal{R}_P, \rightarrow \rangle$ with a finite P is confluent.*

Since the techniques in proving this theorem are of little relevance to the rest of this paper, we postpone the proof to Appendix A.

The next theorem states the soundness and completeness of goal rewrite systems.

Theorem 4.10. *Let P be a finite program and $\langle \mathcal{Q}_L, \mathcal{R}_P, \rightarrow \rangle$ a goal rewrite system.*

Soundness: *For any literal g and any rewrite sequence $g \rightarrow^* T(C_1) \vee \cdots \vee T(C_m)$, there exists a partial stable model M_i of P, for each $i \in [1..m]$, such that $g \in C_i \subseteq M_i$.*

Completeness: *For any literal g true in a partial stable model M of P, there exists a rewrite sequence $g \rightarrow^* T(C_1) \vee \cdots \vee T(C_m)$ such that there exists $i \in [1..m]$, $g \in C_i \subseteq M$.*

Intuitively, the soundness says that whenever a literal g is proved, there exists a partial stable model containing g, and the completeness says that if g is true in one partial stable model, then there always exists a demonstrating proof. Notice that since stable models are a special case of partial stable models, this means that if g is true in some stable model, then there is a proof of it in the rewrite system. However, the converse is not true in general because stable model is a global notion and our rewrite system only checks local consistency. We shall have more to say about this after we generalize the theorem to include abducibles.

Before we prove the theorem, let's discuss the relationship between goal rewriting and derivability. Given a program P and a set of default negations Δ, $P \cup \Delta$ is viewed as a positive program. By a derivation of an atom ϕ using program P, we mean the usual least fixpoint construction of $P \cup \Delta$. In the following discussion, P refers to some fixed, finite program, and $\langle \mathcal{Q}_L, \mathcal{R}_P, \rightarrow \rangle$ is the goal rewrite system with respect to P. Δ and Δ' denote sets of default negations.

We consider rewriting without using loop rules, in which case loop literals are just left as terminating nodes. Any generated $T(C)$ is viewed as a conjunction of the literals in C. Goal rewriting in this case preserves logical equivalence under 2-valued as well as 3-valued logic—any generated equivalence logically follows from $Comp(P)$. For this reason, we are free to use $\leftrightarrow$ to express a rewrite sequence generated without using loop rules.

Suppose there are n rules in P with atom a as the head: $a \leftarrow B_i, \mathsf{not_}C_i$, $i \in [1..n]$ (recall that B_i denotes the set of positive literals in the body and $\mathsf{not_}C_i$ the set of default negations). The completion of a is then:

$$a \leftrightarrow (B_1 \wedge \neg C_1) \vee \cdots \vee (B_n \wedge \neg C_n)$$

The disjunctive normal form (DNF) at the right hand side of $\leftrightarrow$ describes all the possible ways to derive a using program P. For negative literal $\neg a$, we have

$$\neg a \leftrightarrow (\neg B_1 \vee C_1) \wedge \cdots \wedge (\neg B_n \vee C_n) \leftrightarrow \Psi_1 \vee \cdots \vee \Psi_m$$

where each Ψ_j is a conjunction of n literals $l_1 \wedge \cdots \wedge l_n$ where l_i is taken from $(\neg B_i \vee C_i)$. By the confluence property, such a DNF can always be obtained by repeatedly applying the distribution rules SR6 and SR6′. Intuitively, such a DNF expresses that any possibility of deriving a using P is *blocked* if we can demonstrate a derivation of each positive literal in Ψ_j, and show that it is impossible to derive l for any negative literal $\neg l$ in Ψ_j. In addition, we must ensure the consistency of Ψ_j. Without it, blocking in the sense above is ineffective, e.g., for the program $\{p \leftarrow q.\ p \leftarrow \mathsf{not}\, q.\}$.

Since any literal l can be expressed equivalently by a DNF, $l \leftrightarrow \Psi_1 \vee \cdots \vee \Psi_m$, in the following, we say that l is defined by $\Psi_1 \vee \cdots \vee \Psi_m$ (or, $\Psi_1 \vee \cdots \vee \Psi_m$ is the definition of l).

Because of the termination and confluence properties, goal rewriting can be carried out in any order. Here, we consider a particular order of rewriting under which the semantical implications are easier to understand:

> Repeatedly, perform a literal rewriting and then transform the derived goal to a DNF.

Any $T(C)$ in such a DNF is viewed as a conjunction of the literals in C. As an illustration, suppose we have $g \leftrightarrow \Psi_1 \vee \cdots \vee \Psi_m$. Without loss of generality, suppose the next literal rewriting takes place at l_1 of $\Psi_1 = l_1 \wedge \cdots \wedge l_n$. Let the definition of l_1 be $\Phi_1 \vee \cdots \vee \Phi_s$. We then get the next DNF

$$\begin{aligned}
&\Psi_1 \vee \cdots \vee \Psi_m \\
&\quad \leftrightarrow \left[(\Phi_1 \vee \cdots \vee \Phi_s) \wedge \bigwedge_{i \in [2..n]} l_i \right] \vee \Psi_2 \vee \cdots \vee \Psi_m \\
&\quad \leftrightarrow \left(\Phi_1 \wedge \bigwedge_{i \in [2..n]} l_i \right) \vee \cdots \vee \left(\Phi_s \wedge \bigwedge_{i \in [2..n]} l_i \right) \vee \Psi_2 \vee \cdots \vee \Psi_m.
\end{aligned}$$

Goal rewriting in this fashion terminates at a DNF where every literal therein is a loop literal (after removing any disjunct that contains an F). To complete the process, we only need to apply loop rules, deal with conjunctive contexts according to simplification rules SR3 and SR4, and remove any conjunction that contains an F according to rules SR1-2 and SR1′-2′. Let us call such a DNF a *pre-normal form*. Clearly, a pre-normal form is unique.

Along with rewrite chains, a DNF represents a collection of *derivation trees*.

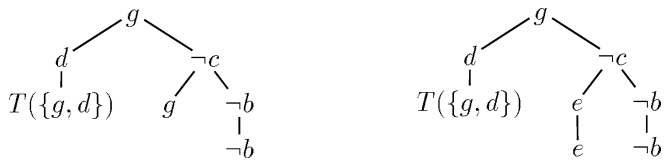


Fig. 1. Derivation trees.

Definition 4.11. Let g be a literal, and $g \leftrightarrow \Psi_1 \vee \cdots \vee \Psi_m$ be a rewrite sequence (without using loop rules) where each Ψ_i, $i \in [1..m]$, is a conjunction of literals and $T(C)$'s. The *derivation tree* (*d-tree* for abbreviation) of Ψ_i is a tree with g as the root node, and literals and $T(C)$'s in Ψ_i as the leaf nodes, where l' is a child node of l iff $l \prec l'$, or $l' = T(C)$ and $l \rightarrow T(C)$.

In a derivation tree, each branch from g to a leaf node l corresponds to the rewrite chain of l. It's called a *derivation* tree because the relation between a positive node p and the collection of its child nodes Φ corresponds to a derivation of p, i.e., $P \cup \Phi^+ \cup \Phi^N \vdash p$. This derivation relation is transitive over positive literals. For a negative literal $\neg p$, the collection of its child nodes Φ' expresses blocking of the derivation of p.

Note that a derivation tree of Ψ_i is unique, up to re-ordering of child nodes.

Example 4.12. Consider the following program:

$g \leftarrow d, \text{not}\, c.$ $\quad b \leftarrow b.$
$c \leftarrow \text{not}\, g, \text{not}\, e.$ $\quad e \leftarrow e.$
$c \leftarrow b.$ $\quad d.$

The pre-normal form from g is generated as:

$$g \leftrightarrow \big(T(\{g, d\}) \wedge g \wedge \neg b\big) \vee \big(T(\{g, d\}) \wedge e \wedge \neg b\big).$$

Fig. 1 depicts the two derivation trees of g, one for each conjunction in the pre-normal form. For instance, the derivation tree at the left has two loop literals, one of which is on an even loop and the other on a negative loop. After applying the loop rules, the tree at the left produces $T(\{g, d, \neg c, \neg b\})$, whereas the one at the right generates an F, due to a positive loop.

In the following, let $g \leftrightarrow \Psi_1 \vee \cdots \vee \Psi_m \rightarrow^* T(D_1) \vee \cdots \vee T(D_s)$ where $\Psi_1 \vee \cdots \vee \Psi_m$ is a pre-normal form. Each Ψ_j, $j \in [1..m]$, is reduced either to an F, or to a $T(D_i)$ if each loop literal in Ψ_j is either on a negative loop or on an even loop, and the union of all conjunctive contexts is consistent. Clearly, since some Ψ_j may have reduced to an F, we have $s \leqslant m$. By the assumption of the above rewrite sequence, we know $s \geqslant 1$. Now suppose $\Psi_j \rightarrow^* T(D_i)$. We then have the following two lemmas.

Lemma 4.13. *For each* $i \in [1..s]$, $D_i^N \subseteq F_P(D_i^N)$, $P \cup D_i^N \vdash D_i^+$, *and* $P \cup F_P(D_i^N) \vdash D_i^+$.

Proof. Assume $\text{not}\, \xi \in D_i^N$. Since D_i is consistent, $P \cup D_i^N \nvdash \xi$, hence $\text{not}\, \xi \in F_P(D_i^N)$, and $D_i^N \subseteq F_P(D_i^N)$. It is clear that for any positive literal p in D_i^+, $P \cup D_i^N \vdash p$, as, in

the derivation tree of Ψ_j, every rewrite chain (branch) from p either ends at a $T(C)$ where $C \subseteq D_i$, or is supported by a negative literal $\neg\xi$ for which $\mathsf{not}\,\xi \in D_i^N$, i.e., $p \prec^+ \neg\xi$. It then follows from $D_i^N \subseteq F_P(D_i^N)$ that $P \cup F_P(D_i^N) \vdash p$, for any positive literal p. □

Lemma 4.14. *For each $i \in [1..s]$, $D_i^N \subseteq F_P^2(D_i^N)$.*

Proof. Suppose there are n rules in P with atom a as the head: $a \leftarrow B_i, \mathsf{not_}C_i$, $i \in [1..n]$. For any $\mathsf{not}\,a \in D_i^N$, to show $\mathsf{not}\,a \in F_P^2(D_i^N)$, we need to show $P \cup F_P(D_i^N) \nvdash a$.

For any Δ such that $P \cup \Delta \vdash a$, consider any minimal $\Delta' \subseteq \Delta$ such that $P \cup \Delta' \vdash a$. Δ' being minimal means that $P \cup (\Delta' - \{\mathsf{not}\,\xi\}) \nvdash a$, for any $\mathsf{not}\,\xi \in \Delta'$. Thus, each of such Δ' corresponds to a particular derivation of a. Our goal is to show that for each of such Δ', there is at least one default negation $\mathsf{not}\,\xi \in \Delta'$ which is needed for the derivation of a but is not in $F_P(D_i^N)$.

It is clear that $\Delta' \neq \emptyset$, as otherwise $P \vdash a$, and thus there exists a rewrite chain, $\neg a \prec^+ F$, in the d-tree of Ψ_j, contradicting the assumption that $\Psi_j \rightarrow^* T(D_i)$.

Then, there is a derivation of a via some rule with a as the head relying on Δ', say $a \leftarrow B_k, \mathsf{not_}C_k$, such that $\Delta' = \mathsf{not_}C_k \cup \Delta''$ for some minimal Δ'' satisfying $P \cup \Delta'' \vdash B_k$. That Δ' is non-empty implies either $\mathsf{not_}C_k$ is non-empty or Δ'' is non-empty. Let the definition of $\neg a$ be $\Pi_1 \vee \cdots \vee \Pi_s$, where each Π_j is of the form $l_1 \wedge \cdots \wedge l_n$, and for each $i \in [1..n]$, $l_i \in \neg B_i$ or $l_i \in C_i$. Consider the kth rule above with a as the head. If $l_k \in C_k$, l_k is positive and $\neg a \prec l_k$ is in the d-tree of Ψ_j. Otherwise, l_k is negative, say $l_k = \neg b$, and $l_k \in \neg B_k$. In this case, $\neg a \prec \neg b$ is in the d-tree of Ψ_j. By repeating the same argument for $\neg b$, and so on, since the d-tree of Ψ_j is finite, there are only two possibilities: every literal l such that $\neg b \prec^+ l$ is negative, or there exists a positive literal ξ such that $\neg b \prec^+ \xi$. Clearly, the former case implies $P \cup \Delta' \nvdash b$. This is a contradiction to $P \cup \Delta' \vdash B_k$, where $b \in B_k$. Otherwise, there exists a positive literal ξ such that $\neg a \prec^+ \xi$. From Lemma 4.13, we know $P \cup D_i^N \vdash \xi$, hence $\mathsf{not}\,\xi \notin F_P(D_i^N)$. Since for each minimal Δ' such that $P \cup \Delta' \vdash a$, there is at least one $\mathsf{not}\,\xi \in \Delta'$ that is needed for the particular derivation of a but is not in $F_P(D_i^N)$, we conclude $P \cup F_P(D_i^N) \nvdash a$. This completes the proof. □

We are now ready to prove the soundness.

Proof of Soundness. Suppose $g \rightarrow^* T(D_1) \vee \cdots \vee T(D_k)$. We show that, for each $i \in [1..k]$, D_i can be extended to a partial stable model. We know $D_i^N \subseteq F_P(D_i^N)$ (Lemma 4.13), and $D_i^N \subseteq F_P^2(D_i^N)$ (Lemma 4.14). We also know that F_P^2 is monotonic and F_P is anti-monotonic (i.e., $S_1 \subseteq S_2$ implies $F_P(S_1) \supseteq F_P(S_2)$). We therefore have the following two sequences

$$D_i^N \subseteq F_P^2\big(D_i^N\big) \subseteq F_P^4\big(D_i^N\big) \ldots$$
$$F_P\big(D_i^N\big) \supseteq F_P^3\big(D_i^N\big) \supseteq F_P^5\big(D_i^N\big) \ldots$$

such that $F_P^k(D_i^N) \subseteq F_P^{k+1}(D_i^N)$, for any even number $k \geqslant 0$, and $F_P^j(D_i^N) \subseteq F_P^{j+1}(D_i^N)$, for any odd number $j \geqslant 1$. Because P is finite, we can restrict the function F_P to the finite domain consisting of atoms appearing in P (any other atom in the underlying language is false in any partial stable model). Thus, the two sequences above converge

at $F_P^n(D_i^N)$, for some even number $n \geqslant 0$, such that $F_P^n(D_i^N) = F_P^{n+2}(D_i^N)$, $F_P^{n+1}(D_i^N) = F_P^{n+3}(D_i^N)$, and $F_P^n(D_i^N) \subseteq F_P^{n+1}(D_i^N)$. By definition, the 3-valued interpretation M that corresponds to the fixpoint $F_P^n(D_i^N)$, namely $M^- = \{\neg\phi \mid \text{not}\phi \in F_P^n(D_i^N)\}$ and $M^+ = \{\phi \mid P \cup F_P^n(D_i^N) \vdash \phi\}$, is a partial stable model of P. □

We need the following lemma for the proof of completeness.

Lemma 4.15. *A partial stable model M of a program P is a* 3*-valued model of Comp(P).*

Proof. For any atom a defined by $a \leftarrow Body_i$, $i \in [1..m]$, we show M satisfies its completion: $a \leftrightarrow Body_1 \vee \cdots \vee Body_m$. This can be proved by considering all three cases: $M(a) = \boldsymbol{t}$, $M(a) = \boldsymbol{f}$, and $M(a) = \boldsymbol{u}$. The first two cases are straightforward. For the last case, since M is a model of P, we know $M(Body_i) \neq \boldsymbol{t}$, for any i. We show $M(Body_i) = \boldsymbol{u}$, for some i. Now assume $M(Body_i) = \boldsymbol{f}$ for all i. We show that this leads to a contradiction. Under this assumption, there is $\phi \in Body_i$ such that $M(\phi) = \boldsymbol{f}$, for each i. ϕ is either an atom or a default negation. Suppose ϕ is an atom. Then that $M(\phi) = \boldsymbol{f}$ implies $\text{not}\phi \in M^N$. Since M is a partial stable model of P, we have $M^N = F_P^2(M^N)$ hence $\text{not}\phi \in F_P^2(M^N)$. By the definition of F_P, we have $P \cup F_P(M^N) \nvdash \phi$. If ϕ is a default negation $\text{not}q$, then $M(q) = \boldsymbol{t}$ hence $P \cup M^N \vdash q$. As a partial stable model, M satisfies $M^N \subseteq F_P(M^N)$. Thus, $P \cup F_P(M^N) \vdash q$. In either case, we have $P \cup F_P(M^N) \nvdash a$ hence $\text{not}a \in F_P^2(M^N)$, i.e., $\text{not}a \in M^N$ and $\neg a \in M$. This contradicts the assumption that $M(a) = \boldsymbol{u}$. □

Proof of Completeness. Assume $g \in M$ for some partial stable model M. By Lemma 4.15, M is a 3-valued model of $Comp(P)$. Due to the confluence and termination properties, any literal can be reduced to a pre-normal form such that the equivalence $g \leftrightarrow \Psi_1 \vee \cdots \vee \Psi_m$ logically follows from $Comp(P)$. Below, $RC(\Psi_i)$ denotes the set of literals appearing on the rewrite chains of the literals in Ψ_i (and is viewed as a conjunction when appropriate). It follows from the transitivity of $\leftrightarrow$ that the following are equivalent:

- M satisfies g;
- M satisfies $RC(\Psi_1) \vee \cdots \vee RC(\Psi_m)$;
- M satisfies some consistent $RC(\Psi_i)$ (any inconsistent $RC(\Psi_j)$ can be removed);
- M satisfies some $RC(\Psi_i)$ for which Ψ_i is odd-loop free (any Ψ_j on an odd loop implies $RC(\Psi_j)$ is inconsistent).

Now suppose $\Psi_1 \vee \cdots \vee \Psi_m \rightarrow^* T(D_1) \vee \cdots \vee T(D_{m'})$. Then, M satisfies $D_1 \vee \cdots \vee D_{m'} \vee U$ where U is the disjunction of those Ψ_j that are consistent (i.e., each $RC(\Psi_j)$ is consistent) but on a positive loop. We are done if M satisfies $D_1 \vee \cdots \vee D_{m'}$. Assume M does not satisfy $D_1 \vee \cdots \vee D_{m'}$. Then M only satisfies U. Consider any Φ in U that is satisfied by M. For any loop literal ξ in Φ, since ξ is on a positive loop, we know $P \cup RC(\Phi)^N \nvdash \xi$. As M satisfies Φ, we have $RC(\Phi) \subseteq M$, hence $\xi \in M$ and $P \cup M^N \vdash \xi$. It follows there exists a Δ such that $\Delta \cup RC(\Phi)^N \subseteq M^N$ and $P \cup \Delta \vdash \xi$. That is, ξ can be reduced, consistently with M, using an alternative rule with ξ as the head. From the

definition of ξ, we know that no possibility of deriving ξ is missed. As each loop literal in Φ can be extended this way, we can replace the derivations from (the first occurrences of) such loop literals by these alternatives. Let Φ' be the conjunction whose d-tree is that of Φ except the derivations from such literals are removed. Then, there exists a rewrite sequence $\Phi' \rightarrow^* \Psi'$ such that Ψ' is in the pre-normal form and not on a positive loop, and $RC(\Psi')$ is consistent due to $RC(\Psi') \subseteq M$. Thus, $\Psi' \rightarrow^* T(C)$ where $C = RC(\Psi')$. Therefore, $T(C)$ must be one of the $T(D_i)$. This contradicts the assumption that M does not satisfy $D_1 \vee \cdots \vee D_{m'}$. We are done. □

A partial stable model M is *maximal* if there is no other partial stable model M' such that $M \subset M'$. Maximal partial stable models minimize the undefined. A stable model is clearly a maximal partial stable model that has no undefined. Maximal partial stable models are also known as *regular models* and *preferred extensions* [6,31,37]. Clearly, any literal g is true in a partial stable model if and only if g is true in a maximal partial stable model. We therefore have

Corollary 4.16. *Goal rewrite systems are sound and complete with respect to the regular model semantics.*

4.4. Rewriting with non-ground programs

The rewriting framework introduced here is defined for ground programs. It however can be applied to function-free programs for proving ground goals. In abduction, observations are usually formulated as ground goals. The idea is that if every derived goal is ground, then all the mechanisms given in this paper become applicable by adding an unification algorithm (in fact, a simple instantiation process is sufficient). 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.

One class of programs that can easily satisfy this property is the so-called *domain restricted programs* [27]. The idea of domain restriction is that if every variable that appears in a rule appears in a positive body literal of the rule, and draws its value from a finite domain, then the instantiation of the rule can be restricted to these domain values. The interest in [27] is to instantiate a function-free program to a possibly smaller ground program while preserving the stable model semantics. For our purpose of non-ground rewriting over ground goals, a rule can be instantiated only on domain predicates for variables that do not appear in the head. For example, to describe the reachability from a node s in a graph we may write

$$reached(U) \leftarrow arc(s, U).$$
$$reached(U) \leftarrow arc(V, U), reached(V).$$

along with some facts about the predicate $arc(X, Y)$, which can be considered a domain predicate. Since in the second rule the variable V appears only in the body, we instantiate the rule to

$$reached(U) \leftarrow arc(a_i, U), reached(a_i).$$

for each node a_i such that $arc(a_i, U)$ is true for some U. Suppose there are n such nodes $a_1, \ldots, a_n$. Then, the completion of the predicate *reached* is:

$$reached(X) \leftrightarrow$$
$$arc(s, X) \vee \big[arc(a_1, X) \wedge reached(a_1)\big] \vee \cdots \vee \big[arc(a_n, X) \wedge reached(a_n)\big].$$

Thus, a ground goal, say $reached(t)$ (to prove that t can be reached from s), is always rewritten to another ground goal.

5. Rewrite systems for abduction

The rewriting framework that we defined in the preceding section 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 $l(C)$ where C is the rewrite chain leading to, including l. Thus when we write $l(C)$, it is understood that C always contains l.

An abducible may appear in a goal positively or negatively, we thus need a terminology to refer to both of them: an *abducible literal* is either an abducible ϕ or its negative counterpart $\neg\phi$.

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. These rewrite systems are both sound and complete with respect to the partial stable models semantics.

Theorem 5.1. *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 with respect to P and A.*

Soundness: *For any literal g and any rewrite sequence*

$$g \rightarrow^* G \vee \big[l_1(C_1) \wedge \cdots \wedge l_k(C_k)\big] \vee G',$$

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 $P \cup \{l_1, \ldots, l_k\}^+$ such that $g \in \bigcup_{1 \leqslant j \leqslant k} C_i \subseteq M$.

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*

$$g \rightarrow^* G \vee \big[l_1(C_1) \wedge \cdots \wedge l_k(C_k)\big] \vee G',$$

such that $g \in \bigcup_{1 \leqslant j \leqslant k} C_j \subseteq M$.

Proof. Since none of the abducibles appear in the head of any program rule, the statements here about soundness and completeness follow directly from those of Theorem 4.10. □

We have again stated our results in terms of the partial stable model semantics. Again the reason that our rewriting system may not be sound under the stable model semantics

is that stable models check for global consistency, but our system checks only local ones. There are several ways to make the rewriting system also sound and complete for stable models. When a conjunction of abducibles

$$l_1(C_1) \wedge \cdots \wedge l_k(C_k)$$

is generated, one can check if $C = C_1 \cup \cdots \cup C_k$ is consistent and complete. If it is, then $\{l_1, \ldots, l_k\}$ is an explanation. If it is consistent but not complete, then we can either call a stable model generator to see if C can be extended to a stable model or we can choose an atom p such that neither it nor its negation is in C, and continue the rewriting with either $p(C)$ or $\neg p(C)$, until a complete context is obtained.

There is however an important special class of logic programs where partial stable models and stable models coincide. We say that a program has no odd loops (odd-loop free) if there is no odd loop starting with any literal. Since goal rewrite systems are confluent, any odd-loop in the program's dependency graph can replicate itself in a rewrite chain of some goal rewrite sequence. Therefore, there is no essential difference between our notion of odd-loop free and the notion of *negative cycle free* in the literature [8,32].

It has been shown in [36] that for any nonground, negative cycle free program with a well-founded stratification, its partial stable models are all 2-valued and thus coincide with its stable models.[3] A stratification in this case is a partial order of strata each of which contains ground atoms that are involved in some loops among themselves. In a well-founded stratification, there is no infinite descending chain in the partial order. That is, every such chain must have a base stratum.[4] This property allows us to construct, along the well-founded stratification in the bottom-up fashion, a 2-valued justifiable model (which is known to be a stable model) from any 3-valued justifiable model. In this way one can show there is no partial stable model with undefined atoms. Since finite propositional programs all have a well-founded stratification, for these programs our rewriting system becomes sound and complete under the stable model semantics. We thus have the following results.

Theorem 5.2. *Let P be a finite program, and $\langle \mathcal{Q}_L, \mathcal{R}_P, A, \rightarrow \rangle$ a goal rewrite system. Suppose q is a proposition and*

$$q \rightarrow^* \left[l_{11}(C_{11}) \wedge \cdots \wedge l_{1k_1}(C_{1k_1})\right] \vee \cdots \vee \left[l_{m1}(C_{m1}) \wedge \cdots \wedge l_{mk_m}(C_{mk_m})\right]$$

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. If P has no odd loops then

$$\left\{\{l_{11}, \ldots, l_{1k_1}\}, \ldots, \{l_{m1}, \ldots, l_{mk_m}\}\right\}$$

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.

[3] The result was stated for maximal partial stable models. However, the claim can be extended to all partial stable models by exactly the same proof.

[4] Here is a program that has no well-founded stratification: $\{p(a).\ \ p(x) \leftarrow \text{not } p(f(x)).\}$.

Proof. We show that for each i, $\{l_{i1}, \ldots, l_{ik_i}\}$ is an explanation of q. Let's denote the set by α, and let β be any complete hypothesis that extends α. We need to show that there is an answer set of $P \cup \beta^+$ that includes q. (Notice again that we have assumed that none of the propositions in A appear in the head of any rule in P.) Since there is a rewrite sequence of the form:

$$q \to^* G \vee \left[l_{i1}(C_{i1}) \wedge \cdots \wedge l_{ik_i}(C_{ik_i})\right] \vee G'$$

in $\langle \mathcal{Q}_L, \mathcal{R}_P, A, \to\rangle$, there is a rewrite sequence of the form $q \to^* T(C) \vee G$ in $\langle \mathcal{Q}_L, \mathcal{R}_{P\cup\beta+}, \to\rangle$, where

$$C = C_{i1} \cup \cdots \cup C_{ik_i},$$

because $C_{i1} \cup \cdots \cup C_{ik_i}$ is consistent, and $\{l_{i1}, \ldots, l_{ik_i}\}$ is a subset of β. Thus by Theorem 4.10, there is a partial stable model M of $P \cup \beta^+$ such that $q \in M$. Now since P does not have any odd loops, neither does $P \cup \beta^+$. So M^+ is an answer set of $P \cup \beta^+$. This shows that β is an explanation of q. So α is an explanation.

Now let β be any complete explanation of q. We need to show that for some i, $\{l_{i1}, \ldots, l_{ik_i}\}$ is a subset of β. Let M be an answer set of $P \cup \beta^+$ that includes q. By Theorem 4.10, there is a rewrite sequence of the form $q \to^* T(C) \vee G$ in $\langle \mathcal{Q}_L, \mathcal{R}_{P\cup\beta+}, \to\rangle$. Then there must be a rewrite sequence of the form: $q \to^* [l_1(C_1) \wedge \cdots \wedge l_j(C_j)] \vee G'$ in $\langle \mathcal{Q}_L, \mathcal{R}_P, A, \to\rangle$ such that $\{l_1, \ldots, l_j\} \subseteq T \cap \beta$, and $C_1, \ldots, C_j$ are consistent. So by Theorem 4.9, it must be the case that

$$l_1 \wedge \cdots \wedge l_j \models [l_{11} \wedge \cdots \wedge l_{1k_1}] \vee \cdots \vee [l_{m1} \wedge \cdots \wedge l_{mk_m}]$$

in propositional logic. But $\{l_1, \ldots, l_j\}$ is contained in β, which is a complete hypothesis, so there must be an i such that $\beta \models l_{i1} \wedge \cdots \wedge l_{ik_i}$. □

Consider again the boat example in Section 1. The Clark completion of *canCross* is:

$$\mathit{canCross} \equiv (\mathit{boat} \wedge \neg \mathit{leak}) \vee (\mathit{boat} \wedge \mathit{leak} \wedge \mathit{hasBucket}).$$

Since *boat*, *leak* and *hasBucket* are abducibles, rewriting for *canCross* terminates in one step, and produces the following cover:

$$\{\mathit{boat} \wedge \neg \mathit{leak},\ \mathit{boat} \wedge \mathit{leak} \wedge \mathit{hasBucket}\}.$$

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 $\mathit{boat} \wedge \neg \mathit{leak}$ and $\mathit{boat} \wedge \mathit{hasBucket}$.

6. 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 semantics that these procedures compute allow 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 [1]). However, for answer set programming where each answer set is taken as a solution to a given problem, loop handling becomes a semantic issue—a sound and complete backward chaining procedure cannot be defined without it.

A number of abductive procedures have been proposed for the two-valued as well as three-valued completion semantics [3,4,9], of which the system by Console et al. [3] and the IFF procedure by Fung and Kowalski [9] also use rewriting as the main mechanism to compute explanations. Console et al. show that, for non-recursive programs (called hierarchical programs), abductive explanations can be computed as a deduction by rewriting using iff-definitions. Fung and Kowalski extend this idea to the class of all normal programs, and get completeness results that can be stated without resorting to termination. This improves the completeness theorems by Denecker and De Schreye [4] which rely on termination as a condition. All of these procedures are defined for *cautious 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. For example, with the program $\{a \leftarrow \mathsf{not}\, b.\ b \leftarrow \mathsf{not}\, a.\}$, the answer to query a should be *no* in their case (however, none of these procedures actually terminates and returns this answer), and *true* in a stable model in our case. Apparently, the differences between the proof methods for consequence finding and those for brave reasoning lie in the correct handling of loops in the latter in order to capture each of the intended models.

Our work is closely related to another abductive procedure, the Eshghi–Kowalski procedure (EKP) [7] (also see [6]), which is sound and complete for ground programs under the finite-failure three-valued stable model semantics in which loops causing infinite failure are modeled by the truth value *undefined* [12]. It is known that with an appropriate handling of positive loops (distinguished as positive and negative loops in this paper), EKP can be made complete for the partial stable model semantics. 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 rewrite systems in the literature. (This further distinguishes our use of rewrite systems from the literature, e.g., in [9].) To illustrate these features, consider the following program

$$r1.\ g \leftarrow \mathsf{not}\, a. \qquad r3.\ b \leftarrow a.$$
$$r2.\ a \leftarrow \mathsf{not}\, b, \mathsf{not}\, c. \qquad r4.\ c \leftarrow a.$$

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. Therefore, g cannot

be proved. Note that in this reasoning we need to remember what was required previously (not a in this case). This is exactly how the proof is done by GRS:

$$g \to \neg a \to b \vee c \to a \vee c \to F \vee c \to c \to a \to F.$$

However, EKP will go through *six* nested levels, and do it twice through backtracking, before the same conclusion can be reached. That a single derivation branch in EKP could be deeply nested brings no surprise that any attempt to lay out a proof presents some challenge, even for small programs. We note that the mechanism of rewrite chain is indispensable in any implementation of a top-down procedure if termination and completeness are to be preserved. For GRS, such a mechanism is used both for termination and for the implementation of the semantics, resulting in a much simpler yet more natural proof structure.

Our goal rewriting procedure departures from the traditional SLDNF-like procedures also in the use of a *computational rule*. Recall that a computational rule is a function that returns a subgoal form a goal. Its interest originates from the so called *independence* of computational rules for Horn clause programs, which says that the commitment to any subgoal can be made without the need of looking back, because no solutions will be missed. For normal programs, such a computational rule must be *fair* which requires generation of an SLD-tree that is either finite, or every subgoal in it is *eventually* selected [25]. These conditions require a fair computation rule to be implemented by a form of breadth-first search in order to find a finitely failed tree. In contrast, since goal rewrite systems are confluent and terminating, literal selection can be arbitrary and is guaranteed to be fair. As an example, consider the following program:

$$\begin{array}{ll} g \leftarrow \mathsf{not}\, a. & \\ a \leftarrow \mathsf{not}\, b, \mathsf{not}\, c. & a \leftarrow \mathsf{not}\, d. \\ b \leftarrow \mathsf{not}\, d. & d \leftarrow \mathsf{not}\, e. \\ e \leftarrow \mathsf{not}\, d. & c. \end{array}$$

The program has two stable models, $\{c, e, a\}$ and $\{a, b, c, e\}$. A complete procedure for brave reasoning should generate a proof for g. Such a proof is reflected naturally, and logically, in goal rewriting

$$g \to \neg a \to (b \vee c) \wedge d \to (\neg d \vee c) \wedge d \to \cdots.$$

Any of the literals in the last goal above can be selected for literal rewriting, or the goal can be transformed to $(\neg d \wedge d) \vee (c \wedge d)$. Even if we can prove $\neg d$, its conflict with d will fail this alternative.[5] Thus $(\neg d \wedge d) \vee (c \wedge d)$ will be rewritten to $F \vee (c \wedge d)$ and then to $(c \wedge d)$. Continuing, we have

[5] As a technical note, this example also explains why in general we cannot have a rewrite rule of the form $T(C) \vee \Phi \to T(C)$, even if we content with one proof.

$$\to c \wedge d$$
$$\to T(\{g, \neg a, c\}) \wedge d$$
$$\to T(\{g, \neg a, c\}) \wedge \neg e$$
$$\to T(\{g, \neg a, c\}) \wedge d$$
$$\to T(\{g, \neg a, c\}) \wedge T(\{g, \neg a, d, \neg e\})$$
$$\to T(\{g, \neg a, c, d, \neg e\}).$$

In an SLDNF-procedure, for instance, in the Eshghi–Kowalski procedure, to prove g we need to prove that any attempt to prove a fails. The two possibilities of proving a are kept in a goal set $\{\leftarrow \mathsf{not}\, b, \mathsf{not}\, c;\ \leftarrow \mathsf{not}\, d\}$. Both should fail in order for g to succeed. From the first goal, suppose we choose $\mathsf{not}\, b$. To fail this goal, we can get a derivation of b using $\mathsf{not}\, d$; so far we have succeeded in choosing $\mathsf{not}\, b$ for the current goal. However, this proof causes a conflict in order to fail the second goal by proving d. Thus, the choice of $\mathsf{not}\, b$ in the first goal does not give us a proof that both goals fail. In fact, we must choose $\mathsf{not}\, c$ in the first goal in order to fail both. Since in general we do not know which subgoal leads to a proof, if a procedure is non-terminating, a fair computational rule must explore all alternatives in an interleaving fashion in order not to miss an answer to a query.

7. Applications and experimental results

We have implemented the writing framework in SWI-Prolog. Our implementation adopts a strategy based on eager literal rewriting and lazy expansion, which resembles the familiar depth-first strategy. The main idea is to delay applying distribution rules SR6 and SR6′ as much as possible to avoid an exponential blow up in goal size. We thus fix the order of rewriting by focusing on the leftmost literal of a goal. We say that a literal l in a goal Q is *rewritable* (for literal rewriting) if it is either at the leftmost position of Q, or at the second leftmost position with a conjunct $T(C)$ at the left where l is not a loop literal and neither l nor $\neg l$ is in C. That is, a literal l is rewritable only in goals that begin with one of the three forms: $l \vee \Phi$, $l \wedge \Phi$, and $T(C) \wedge l$, where Φ is a formula. If l is a loop literal then a loop rule is applied; if $\neg l \in C$ then rule SR5 is applied to produce an F; if $l \in C$, l is already proved, thus the rewrite chain of l is merged with C. Being lazy means that a goal is simplified only when doing so is necessary to make the goal rewritable. In particular, the distribution rules SR6 and SR6′ will be applied only when a goal is not rewritable, and none of the above applies.

Under this strategy, we keep rewriting the literal at the leftmost position of a goal. Eventually, it will become an F or a $T(C)$. If it is a $T(C)$, we will keep rewriting the leftmost literal l in conjunction with $T(C)$, during which $\neg l \in C$ is checked for consistency. When this l is rewritten to $T(C')$, C and C' are merged; and recursively, we continue to focus on the literal at the leftmost position of the goal, or the one in conjunction with $T(C \cup C')$. The reader may refer to the rewrite sequence in Example 4.1 as an example, which is generated using this strategy.

In our current implementation, the input is required to be a set of Clark completion sentences, one for each non-abducible proposition. 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 [22–24].

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:

$$\mathrm{ta}(X, X1, X).$$

$$\mathrm{pa}(X, X1, X2) \leftarrow \mathrm{ta}(X, X1, X2), \mathrm{in}(X1, X2).$$

$$\mathrm{ta}(X, X1, X2) \leftarrow X \neq X2, \mathrm{ta}(X), \mathrm{not}\,\mathrm{taol}(X, X1, X2).$$

$$\mathrm{taol}(X, X1, X2) \leftarrow Y \neq X, \mathrm{ta}(Y, X1, X2).$$

$$\mathrm{pa}(X, X1, X2) \leftarrow \mathrm{pa}(X), \mathrm{not}\,\mathrm{paol}(X, X1, X2).$$

$$\mathrm{paol}(X, X1, X2) \leftarrow Y \neq X, \mathrm{pa}(Y, X1, X2).$$

$$\mathrm{in}(X, Y) \leftarrow \mathrm{in}.$$

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 $loc(X)$ for each variable X in the body of a rule (all the variables in the program refer to locations). Thus, the program is domain restricted, and we only need to instantiate 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 $D = \{1, 2, 3, 4\}$:[6]

Query	Result	Time
ta(1, 2, 3)	false	0.0
ta(3, 2, 3)	true	0.0
pa(1, 2, 3)	pa(1) $\wedge \neg$in	0.05
$\neg$pa(1, 2, 3)	$\neg$pa(1) $\vee$ in $\vee$ pa(2) $\vee$ pa(3) $\vee$ pa(4)	0.2
pa(2, 2, 3)	pa(2) $\wedge \neg$in	0.08
$\neg$pa(2, 2, 3)	$\neg$pa(2) $\vee$ in $\vee$ pa(1) $\vee$ pa(3) $\vee$ pa(4)	0.1
pa(3, 2, 3)	pa(3) $\vee$ in	0.25
$\neg$pa(3, 2, 3)	$\neg$in $\wedge \neg$pa(3) $\vee \neg$in $\wedge$ pa(1) $\vee$ $\neg$in $\wedge$ pa(2) $\vee \neg$in $\wedge$ pa(4)	0.1

For instance, the row on 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 0.05 seconds. The row on 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 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 3, 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 [33]: $p \leftarrow \mathsf{not} \neg p$ 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 **smodels** [35] or **dlv** [20] 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.

8. Final remarks

Without the minimality requirement, a sound and complete procedure for abduction is required to generate sometimes a large amount of essentially redundant explanations. In this paper, we have given a new definition of abduction for logic programming that resolves this problem. In practice, for efficiency reasons one need not always compute the

[6] On a PIII 1 GHz PC with 512 MB RAM running SWI-Prolog 3.2.9.

set of minimal explanations, but a cover, which may be considered a semantically adequate representation of all explanations.

Computationally, we have shown that explanations can be computed by confluent and terminating rewriting. On the one hand, our work explores the well-understood relationships among the completion semantics, the partial stable model semantics, and the answer set semantics. On the other hand, we combine several ideas that had only been studied previously in separate contexts. Namely, we build loop checking into rewrite systems that implement the completion semantics to obtain an abductive procedure for the partial stable model semantics.

There are several directions for extending this work. One of them is to consider rewriting for non-ground programs for some restricted yet decidable classes of non-ground goals. This would extend our rewriting procedure to a more general query-answering procedure for wider classes of applications. Another question is on the handling of constraints of the form:

$$\bot \leftarrow a_1, \ldots, a_i, \mathsf{not}\, b_1, \ldots, \mathsf{not}\, b_n.$$

Our new definition of abduction can be extended to include these constraints straightforwardly. Computationally, constraints may be handled in our rewriting procedure just like in other abductive procedures [4,9,16]: a goal is proved along with all the constraints. This ensures that all of the constraints are satisfied when the goal is proved. This approach actually produces a new semantics: partial stable model semantics with constraints. This is distinguished from the partial stable model semantics because a normal program under this new semantics is no longer guaranteed a partial stable model. The semantical implications of partial stable models with constraints worth further study.

To improve the efficiency of the goal rewriting procedure, space pruning techniques shall be investigated and incorporated. Scalability may be improved by considering different strategies of maintaining a goal so that the run time space usage can be reduced. For example, one possible strategy is not to expand a goal using the distribution rules SR6 and SR6′; instead, a collection of literals from the goal is selected, one at a time, that corresponds to a candidate solution. This requires book keeping mechanisms that should be closely related to maintaining backtrack points in implementing Prolog languages.

Acknowledgements

We would like to thank Ilkka Niemelä for helpful discussions related to the topics of this paper, and Ken Satoh for comments on an 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.

Appendix A

We prove that a goal rewrite system with a finite program is confluent. It is known in the literature [13] that, a terminating rewrite relation is confluent iff it is *locally confluent*. Local confluence is defined as: whenever $t_1 \rightarrow t_2$ and $t_1 \rightarrow t_3$, there exist t_4 and rewrite

sequences such that $t_2 \rightarrow^* t_4$ and $t_3 \rightarrow^* t_4$. It therefore suffices to show the property of local confluence.

Theorem A.1. *Any goal rewrite system* $\langle \mathcal{Q}_L, \mathcal{R}_P, \rightarrow \rangle$ *with a finite* P *is locally confluent.*

We introduce some notations.

A goal formula is viewed as a tree. A subformula is identified by a sequence of positive integers describing the path from the root symbol to the head of the subformula. These sequences are called *indices*. That an index ω identifies a subformula Φ is expressed by a mapping $m(\omega) = \Phi$. The empty sequence identifies the formula itself. For example, given $(l_1 \vee l_2) \wedge l_3$, we have $m(1) = l_1 \vee l_2$, and $m(1.2) = l_2$.

A rewrite sequence of zero or more steps is denoted as $Q_0 \rightarrow^* Q_k$. When we are interested in where in a given goal Q_i a rewrite occurs and which rule is applied, we write $Q_i \rightarrow_{\omega,r} Q_{i+1}$ to indicate that rule r is applied to the subformula at index ω.

Proof. Let $Q_0 \rightarrow^* Q_k$ be a rewrite sequence, where $k \geqslant 0$. Suppose $Q_k \rightarrow_{\omega,r} N$ and $Q_k \rightarrow_{\omega',r'} N'$. We consider all the cases of possibly different rewrites on Q_k.

If the two rewrite steps are independent of each other, i.e., if their indices are non-overlapping, then trivially, there exists a formula M such that $N \rightarrow_{\omega',r'} M$ and $N' \rightarrow_{\omega,r} M$.

The following are the overlapping cases.

Case 1. A loop rewrite at $m(\omega) = l$ and a rewrite by SR5 at $m(\omega') = T(C) \wedge l$ (the symmetric case of a rewrite by SR5′ is similar). Using SR5 followed by SR2, we have $T(C) \wedge l \rightarrow F \wedge l \rightarrow F$. That is, an F is generated at ω'. A loop rule produces either an F, in which case we have $T(C) \wedge F \rightarrow F$ so that an F is at ω', or $T(C')$ for some C' at ω. The latter leads to $T(C) \wedge T(C') \rightarrow F$ due to $l \in C'$ and $\neg l \in C$, so that at ω' is also an F.

Case 2. Literal rewriting at $m(\omega) = l$ and a rewrite by SR5 at $m(\omega') = T(C) \wedge l$ (the symmetric case of a rewrite by SR5′ is similar). Again SR5 leads to an F at ω'. Since P is finite, it follows from Proposition 4.7 that the sequence terminates at either an F, or a $T(C_1) \vee \cdots \vee T(C_m)$ for some $m \geqslant 1$. Hence, there exists an extension from $Q_k \rightarrow_{\omega,r} N$, say $Q_k \rightarrow_{\omega,r} M \rightarrow^* M'$ such that M' is the same as M except at ω, $m(\omega) = F$ or $m(\omega) = T(C_1) \vee \cdots \vee T(C_m)$, for some $m \geqslant 1$. That is, at ω' we have either $T(C) \wedge F$, or $T(C) \wedge (T(C_1) \vee \cdots \vee T(C_m))$ where $\neg l \in C$ and $l \in C_i$ for each $1 \leqslant i \leqslant m$. Clearly, in either case, the rewrite sequence can be extended to lead to an F at ω'.

Case 3. Any rewrite inside a subformula that is distributed by SR6 (the symmetric case is similar). SR6 is $\Phi_1 \wedge (\Phi_2 \vee \Phi_3) \rightarrow (\Phi_1 \wedge \Phi_2) \vee (\Phi_1 \wedge \Phi_3)$. Any rewrite at a subformula inside $\Phi_1 \wedge (\Phi_2 \vee \Phi_3)$ causes overlapping. Clearly, distribution after the rewrite and the rewrite (or the duplicated rewrites if inside Φ_1) after distribution lead to the same result. Note that distribution does not change a rewrite chain for any literal; it simply duplicates the rewrite chain for each occurrence of the same literal.

Case 4. A rule overlaps with its symmetric counterpart. This includes the following cases of a rule and its symmetric counterpart both being applicable: SR1 and SR1′ for goal $F \vee F$, SR2 and SR2′ for goal $F \wedge F$, and SR6 and SR6′ for goal $\Psi_1 \wedge \Psi_2$ where each Ψ_i is a disjunction. Clearly, in each case, there exist rewrite sequences leading to a common goal. □

References

[1] R. Bol, A. Krzysztof, J. Klop, An analysis of loop checking mechanisms for logic programs, Theoret. Comput. Sci. 86 (1) (1991) 35–39.
[2] K.L. Clark, Negation as failure, in: H. Gallaire, J. Minker (Eds.), Logic and Data Bases, Plenum Press, New York, 1978, pp. 293–322.
[3] L. Console, D. Theseider, P. Porasso, On the relationship between abduction and deduction, J. Logic Programming 2 (5) (1991) 661–690.
[4] M. Denecker, D. De Schreye, SLDNFA: An abductive procedure for normal abductive programs, J. Logic Programming 34 (2) (1998) 111–167.
[5] N. Dershowitz, P. Jouannaud, Rewrite systems, in: Handbook of Theoretical Computer Science, Vol B: Formal Methods and Semantics, North-Holland, Amsterdam, 1990, pp. 243–320.
[6] P. Dung, An argumentation theoretic foundation for logic programming, J. Logic Programming 22 (1995) 151–177.
[7] K. Eshghi, R.A. Kowalski, Abduction compared with negation by failure, in: Proc. 6th Internat. Conference on Logic Programming, MIT Press, Cambridge, MA, 1989, pp. 234–254.
[8] F. Fages, Consistency of Clark's completion and existence of stable models, J. Methods Logic Comput. Sci. 1 (1994) 51–60.
[9] T. Fung, R. Kowalski, The iff proof procedure for abductive logic programming, J. Logic Programming 33 (2) (1997) 151–164.
[10] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: Proc. 5th Internat. Conference on Logic Programming, MIT Press, Cambridge, MA, 1988, pp. 1070–1080.
[11] M. Gelfond, V. Lifschitz, Classic negation in logic programs and disjunctive database, New Generation Computing 9 (1991) 365–385.
[12] L. Giordano, A. Martelli, M. Sapino, Extending negation as failure by abduction: A three-valued stable model semantics, J. Logic Programming 26 (1) (1996) 31–67.
[13] G. Huet, Confluent reductions: Abstract properties and applications to term rewriting systems, J. ACM 27 (4) (1980) 797–821.
[14] A. Kakas, R. Kowalski, F. Toni, The role of abduction in logic programming, in: Handbook of Logic in Artificial Intelligence and Logic Programming, Oxford University, Oxford, 1995.
[15] A. Kakas, P. Mancarella, Generalized stable models: A semantics for abduction, in: Proc. 9th European Conference on Artificial Intelligence, 1990, pp. 285–291.
[16] A. Kakas, A. Michael, C. Mourlas, ACLP: Abductive constraint logic programming, J. Logic Programming 44 (1–3) (2000) 129–178.
[17] S. Kleene, Introduction to Metamathematics, Wolters-Noordhoff, 1971.
[18] K. Konolige, Abduction versus closure in causal theories, Artificial Intelligence 53 (1992) 255–272.
[19] K. Kunen, Signed data dependencies in logic programs, J. Logic Programming 7 (3) (1989) 231–245.
[20] N. Leone, et al., DLV: A disjunctive datalog system, release 2000-10-15, Vienna University of Technology, 2000.
[21] V. Lifschitz, Answer set programming, in: K.R. Apt et al. (Eds.), The Logic Programming Paradigm: A 25-Year Perspective, Springer, Berlin, 1999, pp. 357–371.
[22] F. Lin, Embracing causality in specifying the indirect effects of actions, in: Proc. IJCAI-95, Montreal, Quebec, Morgan Kaufmann, San Mateo, CA, 1995, pp. 1985–1993.
[23] F. Lin, From causal theories to successor state axioms: Bridging the gap between nonmonotonic action theories and STRIPS-like formalisms, in: Proc. AAAI-2000, Austin, TX, 2000, pp. 781–793.
[24] F. Lin, K. Wang, From causal theories to logic programs (sometimes), in: Proc. 5th LPNMR, El Paso, TX, 1999.
[25] J.W. Lloyd, Foundations of Logic Programming, Springer, Berlin, 1987.
[26] V. Marek, M. Truszczyński, Stable models and an alternative logic programming paradigm, in: K.R. Apt et al. (Eds.), The Logic Programming Paradigm: A 25-Year Perspective, Springer, Berlin, 1999, pp. 375–398.
[27] I. Niemelä, Logic programs with stable model semantics as a constraint programming paradigm, Ann. Math. Artificial Intelligence 25 (3–4) (1999) 241–273.
[28] D. Poole, Representing knowledge for logic-based diagnosis, in: Proc. Fifth Generation Computer Systems Conference, 1988, pp. 1282–1290.

[29] T.C. Przymusinski, Extended stable semantics for normal and disjunctive logic programs, in: Proc. 7th Internat. Conference on Logic Programming, MIT Press, Cambridge, MA, 1990, pp. 459–477.
[30] R. Reiter, J. de Kleer, Foundations of assumption-based truth maintenance systems: Preliminary report, in: Proc. AAAI-87, Seattle, WA, 1987, pp. 183–189.
[31] D. Saccà, C. Zaniolo, Stable models and non-determinism in logic programs with negation, in: Proc. 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1990, pp. 205–217.
[32] T. Sato, Completed logic programs and their consistency, J. Logic Programming 9 (1) (1990) 33–44.
[33] K. Satoh, N. Iwayama, Computing abduction using the TMS, in: Proc. 8th Internat. Conference on Logic Programming, 1991, pp. 505–518.
[34] K. Satoh, R. Iwayama, A query evaluation method for abductive logic programming, in: Proc. Joint Internat. Conference and Symposium on Logic Programming, 1992, pp. 671–685.
[35] P. Simons, I. Niemelä, T. Soininen, Extending and implementing the stable model semantics, Artificial Intelligence 138 (2002) 181–234.
[36] J. You, L. Yuan, A three-valued semantics for deductive databases and logic programs, J. Computer System Sci. 49 (1994) 334–361.
[37] J. You, L. Yuan, On the equivalence of semantics for normal logic programs, J. Logic Programming 22 (1995) 212–221.

ELSEVIER

Artificial Intelligence 128 (2001) 143–159

Artificial Intelligence

www.elsevier.com/locate/artint

On strongest necessary and weakest sufficient conditions ☆

Fangzhen Lin

Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong

Received 3 May 2000

Abstract

Given a propositional theory T and a proposition q, a sufficient condition of q is one that will make q true under T, and a necessary condition of q is one that has to be true for q to be true under T. In this paper, we propose a notion of strongest necessary and weakest sufficient conditions. Intuitively, the strongest necessary condition of a proposition is the most general consequence that we can deduce from the proposition under the given theory, and the weakest sufficient condition is the most general abduction that we can make from the proposition under the given theory. We show that these two conditions are dual ones, and can be naturally extended to arbitrary formulas. We investigate some computational properties of these two conditions and discuss some of their potential applications.
© 2001 Elsevier Science B.V. All rights reserved.

Keywords: Automated reasoning; Definability; Abduction

1. Introduction

Given a propositional theory T and a proposition q, a sufficient condition α of q is one that will make q true under T: $T \models \alpha \supset q$. Symmetrically, a necessary condition α of q is one that has to be true for q to be true under T: $T \models q \supset \alpha$.

For instance, consider the following theory T:

$$rain \supset grassWet,$$

$$sprinklerOn \supset grassWet.$$

☆ An earlier version of this paper was the co-winner of the Best Paper Award at KR2000.

E-mail address: flin@cs.ust.hk (F. Lin).

0004-3702/01/$ – see front matter © 2001 Elsevier Science B.V. All rights reserved.
PII: S0004-3702(01)00070-4

A sufficient condition for *grassWet* to be true under T is for *rain* to be true. Another (trivial) one is *grassWet* itself. Yet another one is *sprinklerOn*. And yet another one is $rain \vee sprinklerOn$.

In this paper we shall propose a notion of weakest sufficient and strongest necessary conditions, study their properties, and consider ways of computing them. There are many potential applications. The following are two examples:

- *Abduction.* As we can see from the above example, given an observation, there may be more than one abductive conclusions that we can draw. It should be useful to find the weakest of such conclusions, i.e., the weakest sufficient condition of the observation.
- *Definability.* It is often necessary to determine whether a given theory yields a definition of a proposition in terms of a set of other propositions. For example, such computation is essential in both Simon's [21] and Pearl's [15] approaches to causation. This is also what is needed in order to compute successor state axioms [18] from causal theories. As we shall see, the problem of whether a proposition can be defined in terms of other propositions, and if so, finding an explicit definition of this proposition can be reduced to that of computing the two conditions that we shall propose in this paper. Furthermore, we believe definability is best handled using these two conditions. While a proposition may or may not be definable in terms of a set of base propositions, our strongest necessary and weakest sufficient conditions, as we shall show, always exist and are unique up to logical equivalence under the given theory. Furthermore, these two conditions are useful to have even when a proposition cannot be defined in terms of others. For instance, in the context of reasoning about actions, this situation arises when an action has an indeterminate effect on a proposition. In this case, the strongest necessary and weakest sufficient conditions of this proposition can be used to completely capture the effect of this action on the proposition.

This paper is organized as follows. We include logical preliminaries in Section 2. In Section 3, we define the notion of strongest necessary and weakest sufficient conditions, and prove some results that characterize these conditions. In Section 4, we extend these two conditions to that of arbitrary formulas. In Section 5 we outline some algorithms for computing these two conditions and discuss some experimental results. In Section 6, we discuss some related work, and finally in Section 7, we conclude this paper.

2. Logical preliminaries

We assume a propositional language. As usual, a truth assignment is a function from the set of propositions in the language to $\{true, false\}$, and the satisfiability relation $M \models \varphi$ is defined as usual between a truth assignment M and a formula φ.

Given a set P of propositions, a formula that mentions only propositions in P is called a *formula of* P. It is clear that the truth value of such a formula is determined by the truth values of the propositions in P only.

Given a formula φ, and a proposition p, we denote by $\varphi(p/true)$ the result of replacing every p in φ by *true*. We extend this notation to a set of formulas as well: if T is a finite set of formulas, then by $T(p/true)$ we mean the formula

$$\bigwedge_{\varphi \in T} \varphi(p/true).$$

We define $\varphi(p/false)$ and $T(p/false)$ similarly. It is clear that for any truth assignment M, $M \models \varphi(p/true)$ iff $M' \models \varphi$, where M' is the truth assignment that is exactly like M but $M'(p) = true$. A similar result holds for $\varphi(p/false)$.

A notion of forgetting, or eliminations of "middle terms" as Boole [2, p. 99] called it, will be essential here. Given a formula φ, and a proposition p, the result of forgetting p in φ, written $forget(\varphi; p)$, is the following formula:

$$\varphi(p/true) \vee \varphi(p/false).$$

Now given a finite set of propositions P, the result of forgetting P in φ, written $forget(\varphi; P)$, is defined inductively as:

$$forget(\varphi; \emptyset) = \varphi,$$
$$forget(\varphi; P \cup \{q\}) = forget(forget(\varphi; q); P).$$

It can be shown that for any formula φ and any propositions p_1 and p_2, $forget(forget(\varphi; p_1); p_2)$ and $forget(forget(\varphi; p_2); p_1)$ are equivalent. So $forget(\varphi; P)$ is well defined.

Since Boole [2] first defined it in 1854, this notion of forgetting, or eliminations of middle terms, has been re-discovered several times. For instance, Weber [23] re-introduced it and used it for updating propositional knowledge bases. It is also a special case of a more general notion of forgetting defined on first-order languages by Lin and Reiter [13] for capturing database progression [12] and certain notions of relevance in problem solving. Lewis [9, p. 155] observed that "For purposes of application of the algebra [1] to ordinary reasoning, elimination is a process more important than solution, since most processes of reasoning take place through the elimination of 'middle' terms". As we shall see, this notion of "the elimination of middle terms" is certainly central for us here.

The following lemma is entailed by Proposition 3 in (Lin and Reiter [13]), and will play an important role in this paper.

Lemma 1. *For any formula φ and any set P of propositions, if ϕ does not mention any propositions in P, then*

$$\varphi \models \phi \Leftrightarrow forget(\varphi; P) \models \phi.$$

In particular, we have that $\varphi \models forget(\varphi; P)$.

[1] The author's note: "the algebra" = Boolean algebra as we know it today.

3. Strongest necessary and weakest sufficient conditions

We begin with strongest necessary conditions. As we shall see, weakest sufficient conditions are dual ones.

Definition 1. Let T be a theory, P a set of propositions in T, and q a proposition in T but not in P. A formula φ of P is said to be a *necessary condition* of q on P under T if $T \models q \supset \varphi$. It is said to be a *strongest necessary condition* if it is a necessary condition, and for any other necessary condition φ', we have that $T \models \varphi \supset \varphi'$.

In this paper, both the background theory T and the base set P of propositions are assumed to be finite. The following proposition is straightforward:

Proposition 1. *If both φ and φ' are strongest necessary conditions of q on P under T, then $T \models \varphi \equiv \varphi'$.*

This means that if there is a strongest necessary condition, then it is unique up to logical equivalence under the background theory. So sometimes, we also call a strongest necessary condition *the* strongest one.

Example 1. The following examples provide some intuitions about our notion of strongest necessary conditions.

(1) $T_1 = \{q \supset (p_1 \wedge p_2)\}$. The strongest necessary condition of q on $\{p_1, p_2\}$ under T_1 is $p_1 \wedge p_2$, and the strongest necessary condition of q on $\{p_1\}$ is p_1.
(2) $T_2 = \{q \supset (p_1 \vee p_2)\}$. The strongest necessary condition of q on $\{p_1, p_2\}$ is $p_1 \vee p_2$, and the strongest necessary condition of q on $\{p_1\}$ is *true* because neither p_1 nor $\neg p_1$ follow from q under T_2.
(3) $T_3 = \{q \supset p_1, q\}$. A strongest necessary condition of q on $\{p_1\}$ is p_1. Another one is *true*. Of course, $T_3 \models p_1 \equiv \mathit{true}$.

The following theorem captures this notion of strongest necessary conditions in terms of truth assignments. It says that a necessary condition φ of q is a strongest one on P if and only if for any model of T, if it satisfies φ, then it can be modified into a model of q without changing the truth values of the propositions in P:

Theorem 1. *Let T, P, and q be as in Definition* 1. *Let φ be a necessary condition of q on P under T. Then φ is a strongest necessary condition of q on P under T iff for any model M of T and φ, there is an assignment M' such that*:

(1) *for any $p \in P$, $M'(p) = M(p)$.*
(2) *$M' \models T \cup \{q\}$.*

Proof. The "if" part: suppose φ is the strongest necessary condition (SNC) of q, and $M \models T \cup \{\varphi\}$. We need to show the existence of a model M' that satisfies the two conditions in the theorem. We prove by contradiction. Suppose there is no such M', i.e., there is no model of $T \cup \{q, M(P)\}$, where $M(P)$ is the conjunction of p, if $M(p) = \mathit{true}$, or $\neg p$,

if $M(p) = \mathit{false}$, for every $p \in P$. Then $T \models q \supset \neg M(P)$. This means that $\neg M(P)$ is a necessary condition (NC) of q on P. Thus $T \models \varphi \supset \neg M(P)$, this contradicts with the assumption that $M \models T \cup \{\varphi\}$.

The "only if" part: suppose for any model M of T and φ, there is another model M' that satisfies the two conditions in the theorem. We need to show that φ is the SNC of q. Let φ' be another NC of q: it is a formula of P and $T \models q \supset \varphi'$. We need to prove that $T \models \varphi \supset \varphi'$. Let M be any model of φ and T. We show that M is also a model of φ'. By our assumption, there is another model M' that satisfies the two conditions in the theorem. This means that $M' \models q$ and $M' \models T$. So $M' \models \varphi'$. But $M(P) = M'(P)$, and φ' is a formula of P. So we have $M \models \varphi'$ as well. □

Notice that it is essential that the formula φ in Theorem 1 be a necessary condition of q. Otherwise, we could always let φ be *false*.

We can use this theorem to verify that a certain necessary condition is in fact the strongest. For instance, consider $T_1 = \{q \supset p_1 \wedge p_2\}$ in Example 1 above. Clearly, p_1 is a necessary condition of q on $\{p_1\}$. It's a strongest one because given any model of T_1, if it satisfies p_1, then we can modify it into another model of T_1 and q while preserving the truth value of p_1. However, although it is also a necessary condition of q on $\{p_1, p_2\}$, it is not the strongest one this time, because if $M \models p_1 \wedge \neg p_2 \wedge \neg q$, then $M \models T_1 \cup \{p_1\}$, but it cannot be modified into a model of q and T_1 without changing the truth value of p_2.

We can similarly define the notion of weakest sufficient conditions:

Definition 2. Let T be a theory, P a set of propositions in T, and q a proposition in T but not in P. A formula ψ of P is said to be a *sufficient condition* of q on P under T if $T \models \psi \supset q$. It is said to be a *weakest sufficient condition* if it is a sufficient condition, and for any other sufficient condition ψ', we have that $T \models \psi' \supset \psi$.

Strongest necessary and weakest sufficient conditions are in fact dual conditions. The easiest way to make this dual relationship precise is to extend these conditions from propositions to arbitrary formulas and to show that for any formula A, a formula φ is a strongest necessary (weakest sufficient) condition of A iff $\neg\varphi$ is a weakest sufficient (strongest necessary) condition of $\neg A$. This will be done in Section 4.

We now relate these two conditions to the notion of definability. We say that a theory T *defines a proposition* q on a set P of propositions iff there is a formula φ of P such that $T \models q \equiv \varphi$. The following proposition is straightforward:

Proposition 2. *A theory T defines a proposition q on P iff $T \models \varphi \supset \phi$, where φ is any strongest necessary condition of q on P and ϕ any weakest sufficient condition of q on P, both under the theory T.*

Given that φ is a strongest necessary condition and ϕ a weakest sufficient condition of q, when the condition $T \models \varphi \supset \phi$ holds, we have that $T \models \varphi \equiv \phi$ and $T \models q \equiv \varphi$. So this proposition reduces the problem of computing definability to that of computing strongest necessary and weakest sufficient conditions. An advantage of working with the latter is that

the two conditions always exist and are unique up to logical equivalence under any given theory. Furthermore, they are useful to have even when they do not yield a definition of a proposition. For instance, in logic of programs, Dijkstra's notion of weakest preconditions is a special case of our notion of weakest sufficient conditions, and as shown in [1,6], these conditions provide a good way to reason about the indeterminate effects of actions for which successor state axioms (definitions) do not exist.

4. Strongest necessary and weakest sufficient conditions of a formula

Our notion of strongest necessary and weakest sufficient conditions can be extended from a proposition to an arbitrary formula.

Definition 3. Let T be a propositional theory, α a formula, and P a set of propositions in $T \cup \{\alpha\}$. A formula φ of P is said to be a necessary condition of α on P iff $T \models \alpha \supset \varphi$. It is said to be a strongest necessary condition if for any other necessary condition φ', we have that $T \models \varphi \supset \varphi'$. Sufficient conditions and weakest sufficient conditions are defined similarly.

Computing the strongest necessary and the weakest sufficient conditions of a formula can be reduced to that of a proposition, as the following proposition shows.

Proposition 3. *Let T, P, and α be as in Definition* 3. *A formula φ of P is the strongest necessary* (*weakest sufficient*) *condition of α on P under the theory T iff it is the strongest necessary* (*weakest sufficient*) *condition of q on P under the theory $T' = T \cup \{q \equiv \alpha\}$, where q is a new proposition not in T and α.*

Proof. We prove this for SNC. The case for WSC is similar. The "if" part: suppose φ is the SNC of α. We show that it is also an SNC of q under T'. It is easy to see that it is an NC of q: $T' \models q \supset \varphi$. Let φ' be any other necessary condition: $T' \models q \supset \varphi'$. Thus $T' \models \alpha \supset \varphi'$. Since q does not occur in α and φ', by Lemma 1, we have that $\mathit{forget}(T', q) \models \alpha \supset \varphi'$. But $\mathit{forget}(T', q)$ is equivalent to T since q does not occur in T and α. So φ' is also a necessary condition of α on P under T. Thus $T \models \varphi \supset \varphi'$. So $T' \models \varphi \supset \varphi'$.

The "only if" part: suppose φ is the SNC of q under T'. First of all, it is an NC: $T' \models q \supset \varphi$. Similar to the reasoning above, we have $T \models \alpha \supset \varphi$. So φ is also an NC of α on P under T. Now if φ' is any other NC of α under T: $T \models \alpha \supset \varphi'$. Then it is also an NC of q under T'. Thus by our assumption that φ is the SNC of q, we have that $T' \models \varphi \supset \varphi'$. Thus $T \models \varphi \supset \varphi'$ since q does not occur in T. □

Although not necessary in principle, the notion of sufficient and necessary conditions of a formula is very useful in expressing many properties. The following are some interesting ones.

The first one says that, as expected, if two formulas are equivalent under T, then they have the same strongest necessary and weakest sufficient conditions:

Proposition 4. *If $T \models \alpha \equiv \beta$, then for any set of propositions P, a formula φ is a strongest necessary* (*weakest sufficient*) *condition of α on P iff it is a strongest necessary* (*weakest sufficient*) *condition of β on P.*

Proof. Suppose φ is the SNC of α on P under T. Then it is NC of β as well. Now let φ' be any SC of β: $T \models \beta \supset \varphi'$. Then we have $T \models \alpha \supset \varphi'$, so $T \models \varphi \supset \varphi'$. Therefore φ is also an SNC of β on P. The proof that if φ is an SNC of β, then it is also an SNC of α is symmetric.

The proof for WSC is similar. □

The following proposition makes precise the dual relation between strongest necessary and weakest sufficient conditions:

Proposition 5. *A formula φ is the strongest necessary* (*weakest sufficient*) *condition of q iff $\neg\varphi$ is the weakest sufficient* (*strongest necessary*) *condition of $\neg q$, where all the conditions are on a common set of propositions and under a common theory.*

Proof. Suppose φ is the SNC of q. Then $T \models q \supset \varphi$. Thus $T \models \neg\varphi \supset \neg q$. So $\neg\varphi$ is a SC of $\neg q$. Suppose φ' is any other SC of $\neg q$: $T \models \varphi' \supset \neg q$. Then $T \models q \supset \neg\varphi'$. Thus $T \models \varphi \supset \neg\varphi'$. So $T \models \varphi' \supset \neg\varphi$. This proves that $\neg\varphi$ is the WSC of $\neg q$.

The proof of the other part of the proposition is similar. □

Proposition 6. *If $\varphi_1, \ldots, \varphi_k$ are the strongest necessary conditions of $\alpha_1, \ldots, \alpha_k$, respectively, then $\varphi = \varphi_1 \vee \cdots \vee \varphi_k$ is the strongest necessary condition of $\alpha_1 \vee \cdots \vee \alpha_k$, where all the conditions are on a common set of propositions and under a common theory.*

Proof. First of all, φ is an NC: $T \models \alpha \supset \varphi$. We show that it is the SNC using Theorem 1, generalized to arbitrary formula using Proposition 3: we show that if $M \models \varphi$ and $M \models T$, then there is a M' such that $M' \models T$, $M' \models \alpha$, and $M'(P) = M(P)$. Since $M \models \varphi$, there must be an $1 \leqslant i \leqslant k$ such that $M \models \varphi_i$. By Theorem 1 and Proposition 3, there is an M' such that $M'(P) = M(P)$, $M' \models T$, and $M' \models \alpha_i$, thus $M' \models \alpha$. □

Symmetrically, for weakest sufficient conditions, we have:

Proposition 7. *If $\varphi_1, \ldots, \varphi_k$ are the weakest sufficient conditions of $\alpha_1, \ldots, \alpha_k$, respectively, then $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_k$ is the weakest sufficient condition of $\alpha_1 \wedge \cdots \wedge \alpha_k$, where all the conditions are on a common set of propositions and under a common theory.*

This proposition is particularly useful in planning when we do not have a successor state axiom for every fluent: given a conjunctive goal $g_1 \wedge \cdots \wedge g_n$, to achieve it using action A, we need to compute the weakest sufficient condition of this conjunction, and reduce the conjunctive goal to it. By this proposition, instead of having to compute the weakest sufficient condition for every possible conjunction, it is enough to compute the condition for each conjunct ahead of time.

5. Computing strongest necessary and weakest sufficient conditions

There are several ways to compute these two conditions. We begin with *prime implicates* [22]. As with most reasoning tasks, strongest necessary and weakest sufficient conditions can be easily computed using prime implicates.

As it is customary in theorem proving, we shall equate a clause, which is a disjunction of literals, with the set of disjuncts in it. Given a propositional theory T, a prime implicate of T is then a clause C such that $T \models C$ and there is no other clause C' such that $T \models C'$, $C \neq C'$, and C' subsumes C, i.e., a subset of C.

Proposition 8. *Let T be a theory, q a proposition, and P a set of propositions. Let Π be the set of prime implicates of T that mention only propositions in $P \cup \{q\}$.*

(1) *The strongest necessary condition of q on P under T is equivalent to the following conjunction*:

$$\bigwedge\{C \mid \neg q \vee C \in \Pi\},$$

where if there is no such prime implicate in Π, then the conjunction is considered to be true, and if $\neg q \in \Pi$, then the conjunction is considered to be false.

(2) *The weakest sufficient condition of q on P under T is equivalent to the following disjunction*:

$$\bigvee\{\neg C \mid q \vee C \in \Pi\},$$

where if there is no such prime implicate in Π, then the disjunction is considered to be false, and if $q \in \Pi$, then the conjunction is considered to be true.

Proof. We prove this proposition for SNC. Let W be the conjunction in the proposition. Clearly, $T \models q \supset W$, so W is an NC. Now let ψ be any other NC: $T \models q \supset \psi$, and Γ the set of prime implicates of ψ. Since ψ does not mention q, the following set

$$\Gamma' = \{\neg q \vee C \mid C \in \Gamma\}$$

is the set of prime implicates of $q \supset \psi$, where if $C = \mathit{true}$ (i.e., ψ is a tautology), then $\neg q \vee C$ is *true*, and if $C = \mathit{false}$ (i.e., ψ is not satisfiable), then $\neg q \vee C$ is $\neg q$. Now since $T \models q \supset \psi$, for each $\neg q \vee C$ in Γ', $T \models \neg q \vee C$, so there is a prime implicate $\neg q \vee C'$ in Π such that C is subsumed by C'. Thus each prime implicate in Γ is subsumed by a conjunct in W. So $W \models \psi$. □

However, computing the strongest necessary (similarly weakest sufficient) condition of a proposition using prime implicates is almost always a bad idea, unless one wants these conditions for all propositions on all possible sets of propositions and wants them in the form of prime implicates. This is because, in general, there is no viable way of computing the set Π in Proposition 8 short of computing the set of all prime implicates of a theory. But once having the set of prime implicates, one can just "read" out these two conditions for any proposition using Proposition 8.

Nevertheless, Proposition 8 is interesting in that it relates our notion of weakest sufficient conditions to Reiter and de Kleer's notion of *minimal support clauses* of a set

of clauses [19]. According to Corollary 4 in [19], a clause C is a minimal support clause for a proposition q with respect to T iff there is a prime implicate C' of T such that $q \in C'$ and $C = C' - \{q\}$. So from above proposition, for any theory T, and proposition q, if P is the set of propositions that are in T but different than q, then the weakest sufficient condition of a q on P under T is equivalent to the disjunction of the negation of each minimal support clause for q with respect to T, and the strongest necessary condition of q is equivalent to the conjunction of the minimal support clauses of $\neg q$.

However, our two conditions are defined independent of prime implicates. In fact, a better way of computing them is in terms of the notion of forgetting that we defined in Section 2, by using the following theorem: [2]

Theorem 2. *Let T be a theory, P a set of propositions, and q a proposition in T but not in P. Let P' be the set of propositions that are in T but not in $P \cup \{q\}$. Then we have*:

(1) *The strongest necessary condition of q on P is $forget(T(q/true); P')$.*

(2) *The weakest sufficient condition of q on P is $\neg forget(T(q/false); P')$.*

Proof. We prove the case for SNC. The case for WSC is similar. First of all, $forget(T(q/true); P')$ is an NC: $T \models forget(T; P')$, so $T \models q \supset forget(T; P')$, thus $T \models q \supset forget(T(q/true); P')$ because $q \notin P'$. It is the strongest: suppose α is an NC: α is a formula of P, and $T \models q \supset \alpha$. Thus, by Lemma 1, we have that $forget(T; P') \models q \supset \alpha$, Therefore $q \models forget(T; P') \supset \alpha$. So $q \models forget(T(q/true); P') \supset \alpha$. Thus $\models forget(T(q/true); P') \supset \alpha$ because q does not occur in $forget(T(q/true); P') \supset \alpha$. □

We illustrate the use of this theorem using the following theory

$$T = \{q \supset p_1 \vee p_2\}.$$

According to the theorem, the strongest necessary condition of q on $\{p_1, p_2\}$ is $forget(T(q/true); \emptyset)$, which is $T(q/true)$ and equivalent to $p_1 \vee p_2$, the same as given in Example 1. The strongest necessary condition of q on $\{p_1\}$ is $forget(T(q/true); \{p_2\})$, which is equivalent to $forget(p_1 \vee p_2; \{p_2\})$, which by definition is $(p_1 \vee true) \vee (p_1 \vee false)$, thus equivalent to *true*.

By its definition, for any formula φ and proposition p, $forget(\varphi; p)$ returns a formula that is twice the size as that of φ; and for a set P of propositions, $forget(\varphi; P)$ returns a formula whose size is 2^n times the size of φ, where n is the size of P. Indeed, computing forgetting in the worst case is expensive. For instance, it is easy to see that a formula is satisfiable iff forgetting all propositions in it returns *true*, and a formula is not satisfiable iff forgetting all propositions in it returns *false*. However, $forget(\varphi; p)$ can often be simplified. For instance, if φ is in disjunctive normal form, then $forget(\varphi; p)$ can be computed efficiently to yield a formula that is shorter than φ: it is the result of replacing both p and $\neg p$ in φ by *true*. More generally, as Darwiche [5] showed, if φ is what he called decomposable negation normal form, then $forget(\varphi; p)$ can be computed efficiently.

[2] Lang and Marquis [8] showed that a proposition q is definable on P under T iff $T \models q \equiv forget(T(q/true); P')$. By Proposition 2, our theorem generalizes this result.

In this paper, we consider two ways of computing forgetting: directly using its definition or through (again) prime implicates. They are embodied below as Algorithms 1 and 2 for computing strongest necessary conditions. The ones for weakest sufficient conditions are similar.

Input. A set T of axioms, a set P of propositions, and a proposition q not in P.

Output. A formula of P that is the strongest necessary condition of q on P under T.

Algorithm 1.

(1) Let $T_1 = \{\varphi \mid \varphi \in T$ is a sentence of $P\}$, and $T_2 = T - T_1$. Replace q in T_2 by *true*, and transform the resulting theory into a *minimal set of clauses* C (see below for a definition). Similarly, transform T_1 into a minimal set of clauses C_0.
(2) Let P' be the set of propositions in C but not in P.
(3) If P' is empty, then go to post-processing, otherwise select a proposition p in P', and delete it from P'.
(4) Transform $C(p/true) \vee C(p/false)$ into a minimal set of clauses, assign it to C, and go back to step (3).
(5) *Post-processing:* although the resulting C is a strongest necessary condition of q, it is often an unwieldy set of clauses; it can be simplified as follows:
   - eliminate those clauses in C that are subsumed by one of the clauses in C_0;
   - for each clause α in C, transform $(C - \{\alpha\}) \cup C_0$ into a minimal set of clauses C_α, and eliminate α from C if it is subsumed by one of the clauses in C_α.

Where a minimal set of clauses is one such that:

- all unit clauses are resolved;
- none of the clauses is subsumed by any other clause in the set.

Our experience with this algorithm has been that it spends the bulk of the time on computing minimal sets of clauses in step (4). But if we do not require a set of clauses to be minimal, then the program quickly runs out of space.

Notice that according to our definition, for a set of clauses C to be a minimal one, only unit clauses in it need to be resolved. So our notion of minimality is much weaker than that of prime implicates. For comparison, let us consider computing forgetting using prime implicates.

The following algorithm makes use of the fact that for any φ and P, *forget*$(\varphi; P)$ is equivalent to the conjunction of prime implicates of φ that do not mention any propositions in P.

Algorithm 2.

(1) Let $T_1 = \{\varphi \mid \varphi \in T$ is a sentence of $P\}$, and $T_2 = T - T_1$. Transform T_1 into a minimal set of clauses C_0.
(2) Generate the prime implicates of $T_2(q/true)$ using Tison's procedure [22], and let C be the set of those prime implicates that mention only propositions in P.
(3) Go to post-processing as in Algorithm 1.

We have implemented the above two algorithms in SWI-Prolog,[3] and run them on both random 3CNFs[4] and action theories used for computing successor state axioms in various action domains [11]. It turned out that Algorithm 2 is always slower than Algorithm 1, with on average a slow down of between 10 to 15 times. The reason is that computing prime implicates is expensive. Our notion of minimal sets of clauses seems to serve our purpose well here in cutting down the number and sizes of clauses without incurring too much overhead in computing them.

Notice that in step (4) of Algorithm 1, after we have chosen a proposition to forget, we compute immediately the result of forgetting this proposition, $C(p/true) \vee C(p/false)$, by a set of minimal clauses. This is like breadth-first search, and as we have mentioned, is the bottleneck of this algorithm. Alternatively, we could work on each disjunct separately, and compute disjunctions only after all those propositions that need to be forgotten have been eliminated. As it turned out, this variant of Algorithm 1 performed far better than Algorithm 1 on random 3CNFs. For random 3CNFs with 50 variables and 215 clauses, this variant of Algorithm 1 spent on average 15 minutes[5] to return a strongest necessary condition of a proposition on a set of 10 other propositions, while Algorithm 1 did not return a solution after running overnight. However, this variant of Algorithm 1 is about 20% slower on action theories. The key is with the disjunction $C(p/true) \vee C(p/false)$: it can be simplified a lot for many benchmark action theories, but not much for random theories.

Regardless of which approach one uses, we have found that sometimes, it is a lot easier to compute the strongest necessary condition than the weakest sufficient condition, and sometimes, it is the other way around. For instance, we have found that for many action theories, strongest necessary conditions are a lot easier to compute than weakest sufficient conditions. When one condition is much easier to compute, the following proposition will be very useful in computing the other one.

Proposition 9. *Let T be a theory, q a proposition, and P a set of propositions.*

(1) *If φ is a necessary condition of q on P under T, and ψ a weakest sufficient condition of q on P under $T \cup \{\varphi\}$, then $\varphi \wedge \psi$ is a weakest sufficient condition of q on P under T.*

(2) *If ψ is a sufficient condition of q on P under T, and φ a strongest necessary condition of q on P under $T \cup \{\neg\psi\}$, then $\varphi \vee \psi$ is a strongest necessary condition of q on P under T.*

Proof.

(1) First of all, $\varphi \wedge \psi$ is an SC: $T \cup \{\varphi\} \models \psi \supset q$, so $T \models \varphi \wedge \psi \supset q$. It is the weakest: suppose α is an SC: $T \models \alpha \supset q$. We need to show that $T \models \alpha \supset \varphi \wedge \psi$. Since $T \models q \supset \varphi$. We have $T \models \alpha \supset \varphi$. But α is also an SC of q under $T \cup \{\varphi\}$, so $T \cup \{\varphi\} \models \alpha \supset \psi$ because ψ is the WSC of q under $T \cup \{\varphi\}$. Therefore $T \models \alpha \supset (\varphi \wedge \psi)$.

[3] SWI-Prolog is developed by Jan Wielemaker at the University of Amsterdam.

[4] Generated using a program by Kautz and Selman.

[5] On a SPARC Ultra2 machine running SWI-Prolog.

(2) Suppose φ' is an NC of q under T: $T \models q \supset \varphi'$. From $T \models \psi \supset q$, we have $T \models \psi \supset \varphi'$. From $T \cup \{\neg\psi\} \models q \supset \varphi$ and $T \cup \{\neg\psi\} \models q \supset \varphi'$, we have $T \cup \{\neg\psi\} \models \varphi \supset \varphi'$. Thus $T \models \neg\psi \wedge \varphi \supset \varphi'$, this together with $T \models \psi \supset \varphi'$, we have $T \models \varphi \supset \varphi'$. □

We illustrate this phenomenon next with a problem in reasoning about action.

6. Generating successor state axioms from causal theories

Consider a robot domain in which a robot needs to move objects around in an office environment. To reason and plan in this domain, the robot needs to have a model of the environment and the effects of her actions. For simplicity, let us assume that there is just one object, and the action that the robot needs to reason about is *move*$(1, 2)$, meaning that it moves from a certain location 1 to location 2. We can then represent the domain by the following propositions:

- at_i—initially, the object is at location i, for each location i.
- $at1_i$—after the action *move*$(1, 2)$, the object is at location i.
- atR_i—initially, the robot is at location i.
- $atR1_i$—after the action, the robot is at location i.
- h—initially, the robot is holding the object.
- $h1$—after the action, the robot is holding the object.

One way of representing the effects of the action is to write a causal theory, which in this case includes the following rules:

(1) The action will cause $atR1_2$ to be true, and for it to be possible, atR_1 must be true;

(2) The object can be at only one location:

$$(at_i \supset at_j) \wedge (at1_i \supset at1_j),$$

for every pair of locations $i \neq j$;

(3) The robot can be at only one location:

$$(atR_i \supset atR_j) \wedge (atR1_i \supset atR1_j),$$

for every pair of locations $i \neq j$;

(4) If the robot is holding the object, then they are at the same location:

$$\big[h \supset (at_i \equiv atR_i)\big] \wedge \big[h1 \supset (at1_i \wedge atR1_i)\big],$$

for every location i.

From these rules and a frame axiom which says that a property's truth value will not be changed unless entailed by the causal rules, we have the following set of axioms:

$$atR_1 \wedge atR1_2,$$

$$atR1_i \equiv atR_i \wedge \neg \bigvee_{j \neq i} atR1_j,$$

$$at1_i \equiv [atR1_i \wedge h1] \vee \left[at_i \wedge \neg \bigvee_{j \neq i} at1_j\right],$$

$$h1 \equiv h,$$
$$atR_i \supset \neg atR_j, \quad \text{for any } j \neq i,$$
$$at_i \supset \neg at_j, \quad \text{for any } j \neq i,$$

where the disjunctions range over all locations. For instance, the third axioms says that the package is at location i after that the robot has moved from location 1 to 2 ($at1_i$) is true iff either the robot is at the same location ($atR1_i$) and holding it ($h1$) or the package was initially at location i (at_i) and is not at any other location now. The first disjunct comes from the fourth causal rule, and the second one by the frame axiom and the second causal rule. The exact procedure of deriving these axioms from a causal theory and the rational behind it are given in (Lin [10,11]).

With this set of axioms, one can determine for any proposition its truth value after the robot has moved from location 1 to 2. However, for computational reasons, it is often advantageous to compile the set of axioms into a set of *successor state axioms* (Reiter [18]). In terms of situation calculus notation, a successor state axiom expresses the truth value of a property in a successor situation in terms of the facts about the current situation, and has the following format:

$$Poss(a, s) \supset \big[F(do(a, s)) \equiv \Phi(s)\big],$$

where $F(do(a, s))$ stands for the truth value of a property F in the situation resulted from doing a in the situation s, and $\Phi(s)$ is a formula about the situation s. Having these axioms facilitates the computation of regression, a crucial mechanism used by most of the AI planners.

In this example, what we want is to derive from the above set of axioms definitions of propositions that are about the successor situations, for example $at1_i$ and $atR1_i$, in terms of propositions that are about the initial situation, for example, at_i and atR_i. This is where our procedures for computing the strongest necessary and weakest sufficient conditions come in, as by Proposition 2.

To illustrate, let q be $at1_2$, and P the set of propositions about the initial situation:

$$P = \{h\} \cup \{at_i \mid i \text{ is a location}\} \cup \{atR_i \mid i \text{ is a location}\}.$$

Then the weakest sufficient condition of q on P is the weakest condition about the initial situation that would ensure that the package is at location 2 after the action is performed, and the strongest necessary condition is the strongest conclusion that one can infer about the initial situation once one knows that the package is at location 2 after the action is performed. When the two conditions are equivalent, which turns out to be the case, we have a successor state axiom for this property.

Fig. 1 compares the performance of computing the weakest sufficient condition of q on P using Algorithm 1 with or without the aid of Proposition 9. In the figure, the x-axis is the number of locations, and y-axis the CPU run time in seconds on a SPARC Ultra 2 machine with 128 MB of memory running SWI-Prolog (version 3.2.7). The line labeled by *snc* corresponds to the strongest necessary condition of q on P,[6] the one labeled by *wsc* the weakest sufficient condition of q on P,[7] and *wsc*1 the weakest sufficient condition of q

[6] For which the program outputs $h \vee at_2$ no matter how many locations there are in the domain.

[7] For which the program always outputs $h \vee at_2$. So we have the successor state axiom: $at1_2 \equiv h \vee at_2$.

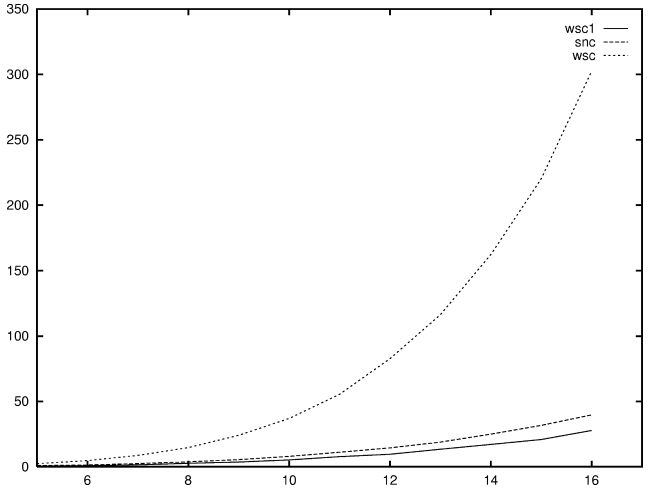


Fig. 1. A robot domain.

on P once the strongest sufficient condition has been added to the theory[8] according to Proposition 9. The speedup of *wsc*1 over *wsc* is quite obvious, and similar speedup was also achieved for other propositions and other action domains that we have experimented with, which include most of the benchmark planning domain in McDermott's PDDL library [14].

7. Related work

Our notion of strongest necessary and weakest sufficient conditions bear structural resemblance to Dijkstra's notion of strongest postconditions and weakest preconditions of programs (cf. [6]). However, Dijkstra was mainly concerned about compositional rules of these conditions over program operators, once these conditions are given for basic assignment statements. In contrast, our study of these two conditions are completely general. Even in applications to computing the effects of actions, our concerns for these two conditions differ from that of Dijkstra, and similarly recent applications of Dijkstra's notion to reasoning about action by Bjäreland and Karlsson [1], in that we are interested in computing the effect of *primitive* actions from a set of direct effect axioms and causal rules.

Our notion of weakest sufficient conditions is closely related to abduction and diagnosis. In propositional logic, given a theory T, a query q, and a finite set A of propositions called *abducibles*, an *explanation* is a finite set E of literals consisting of propositions in A such that (see, e.g. [7,16,17,19,20]):

- $T \cup E \models q$ and $T \cup E$ is consistent.
- E is a minimal such set, i.e., there is no E' that satisfies the above conditions and at the same time is a proper subset of E.

[8] For which the program always outputs *true*.

As is well-known, E is an explanation of q with respect to T and A iff the clause consisting of the complement of each element in E, i.e., $\bigvee_{l\in E}\tilde{l}$ is a minimal support clause of q with respect to T and A, where $\tilde{p}$ is $\neg p$ and $\tilde{\neg p}$ is p for any proposition p.[9] Thus by Proposition 8, the weakest sufficient condition of q is logically equivalent to the disjunction of all explanation of q. So much of the complexity results (e.g., [20]) for abduction carries over here.

However, while much of the work on abduction and diagnosis has been on computing individual explanations, we are more interested here in logical formulas that capture all possible explanations without committing ourself to the exact formats that these formulas ought to have. For instance, given the following T:

$$q \vee \bigvee_{1\leqslant i\leqslant n} (p_i \wedge q_i)$$

there are 2^n number of explanations of q with respect to T and $\{p_i, q_i \mid 1 \leqslant i \leqslant n\}$. However, the disjunction of these explanations has a logically equivalent formula, i.e., a weakest sufficient condition of q, whose length is proportional to n: $\neg \bigvee_{1\leqslant i\leqslant n}(p_i \wedge q_i)$.

But there are some exceptions. In the framework of logic programming, Console et al. ([3]) defined the *explanation formula* of an abduction problem to be *the most specific formula* in the language of abducible atoms such that the formula is entailed by the Clark completion of the program and the observation. Here the most specific formula is like our strongest condition, but with a difference. According to Console et al., a formula φ is more specific than φ' if $\varphi \models \varphi'$, while for us, this entailment only needs to be true under the given logical theory. This somewhat minor difference has some far reaching ramifications, as we shall see in connection with a more recent work by Darwiche [4] who proposed a very similar notion and applied it to diagnosis problems.

Given a theory T, a formula α, and a set of propositions P such that none of the propositions in P occur in α, Darwiche [4] defined the *consequence* of α on P to be a formula φ such that $T \cup \{\alpha\} \models \varphi$, and for any such φ', we have that $\varphi \models \varphi'$. Similarly, he defined the *argument* of α to be a formula φ such that $T \cup \{\varphi\} \models \alpha$, and for any such φ', we have that $\varphi' \models \varphi$. As one can see, Darwiche's notion of consequences generalizes that of explanation formulas by Console et al. above, and his notion of arguments corresponds to our notion of weakest sufficient conditions. But again, the crucial difference is in how formulas are compared. In Darwiche's case, as in the case of Console et al., when φ and φ' are compared, they are compared to under logical entailment. But in our case, they are compared to under logical entailment *relative* to the background theory T. The difference has some significant ramifications. First of all, if T defines α: $T \models \alpha \equiv \varphi$, then φ is both the strongest necessary and weakest conditions of α. But φ may be neither the consequence nor the argument of α according to Darwiche's definitions. Secondly, the post-processing step in Algorithm 1 will not be correct according to Darwiche's definitions. Perhaps most significantly, our Proposition 9 would not be true under Darwiche's definitions. For instance, consider the theory $T = \{p \supset q, q \supset \neg p\}$. According to Darwiche's definitions, the consequence of q on $\{p\}$ with respect to T is $\neg p$. So for Proposition 9 to hold in Darwiche's framework, the argument of q on $\{p\}$ with respect to T would have to be

[9] This result is an easy corollary of the results in [19].

equivalent to the conjunction of $\neg p$ and the argument of q on $\{p\}$ with respect to $T \cup \{\neg p\}$. Unfortunately, this is not true, as the conjunction is equivalent to *false*, which is not the argument of q on $\{p\}$ with respect to T. Notice that *false* is the weakest sufficient condition of q on $\{p\}$ under T, which means that under no condition on p can q be true as $\neg p$ is entailed by the theory. This last difference between our notion and Darwiche's is significant as Proposition 9 is the basis of one of the key strategies that we used for computing the weakest sufficient conditions in relation to action theories.

8. Concluding remarks and future work

We have proposed a notion of strongest necessary and weakest sufficient conditions of a proposition, and considered ways of computing them. We believe these conditions have many potential applications in various areas including abduction and reasoning about actions.

There are several directions for future work. One of them is to extend the results here to the first-order case. This can be a difficult task. For instance, a result in [12] shows that forgetting in the first-order case cannot in general be expressible in first-order logic. As a consequence, we expect that strongest necessary conditions of a proposition under a first-order theory cannot in general be expressible in first-order logic either. It seems that the best hope for dealing with the first-order case is to first reduce it to the propositional case, and then try to learn a first-order description from a set of propositional ones.

Acknowledgements

I would like to thank Maurice Pagnucco for his comments on an earlier version of this paper, and Adnan Darwiche for discussions related to the subject of this paper. I would also like to thank the anonymous reviewers for KR'2000 and this journal for their insightful comments. This work was supported in part by the Research Grants Council of Hong Kong under Competitive Earmarked Research Grants HKUST6061/00E.

References

[1] M. Bjäreland, L. Karlsson, Reasoning by regression: Pre- and postdiction procedures for logics of action and change with nondeterminism, in: Proc. IJCAI-97, Nagoya, Japan, 1997, pp. 1420–1425.
[2] G. Boole, An Investigation of the Laws of Thought, Walton, London, 1854 (Reprinted by Dover Books, New York, 1954).
[3] L. Console, D. Dupre, P. Torasso, On the relationship between abduction and deduction, J. Logic Comput. 1 (5) (1991) 661–690.
[4] A. Darwiche, A logical notion of conditional independence: Properties and applications, Artificial Intelligence 97 (1–2) (1997) 45–82.
[5] A. Darwiche, Compiling knowledge into decomposable negation normal form, in: Proc. IJCAI-99, Stockholm, Sweden, 1999.
[6] E.W. Dijkstra, C.S. Scholten, Predicate Calculus and Program Semantics, Springer, New York, 1990.
[7] K. Konolige, Abduction versus closure in causal theories, Artificial Intelligence 53 (1992) 255–272.

[8] J. Lang, P. Marquis, Two forms of dependence in propositional logic: Controllability and definability, in: Proc. AAAI-98, Madison, WI, AAAI Press, Menlo Park, CA, 1998.
[9] C.I. Lewis, A Survey of Symbolic Logic, University of California Press, Berkeley, CA, 1918 (Reprinted by Dover, New York, 1960. Chapter II in The Classic or Boole-Schroder Algebra of Logic).
[10] F. Lin, Embracing causality in specifying the indirect effects of actions, in: Proc. IJCAI-95, Montreal, Quebec, 1995, pp. 1985–1993.
[11] F. Lin, From causal theories to successor state axioms: Bridging the gap between nonmonotonic action theories and STRIPS-like formalisms, in: Proc. AAAI-2000, Austin, TX, AAAI Press, Menlo Park, CA, 2000, pp. 786–791.
[12] F. Lin, R. Reiter, How to progress a database, Artificial Intelligence 92 (1–2) (1997) 131–167.
[13] F. Lin, R. Reiter, Forget it!, in: R. Greiner, D. Subramanian (Eds.), Working Notes of AAAI Fall Symposium on Relevance, AAAI Press, Menlo Park, CA, 1994, pp. 154–159; also available at http://www.cs.toronto.edu/cogrobo/forgetting.ps.Z.
[14] D. McDermott et al. (AIPS-98 Planning Competition Committee), PDDL—The Planning Domain Definition Language, Technical Report TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, New Haven, CT, 1998.
[15] J. Pearl, Causal diagrams for empirical research, Biometrika 82 (1995) 669–709.
[16] D. Poole, Representing knowledge for logic-based diagnosis, in: Proc. 5th Generation Computer Systems Conference (FGCS'88), 1988, pp. 1282–1290.
[17] D. Poole, Explanation and prediction: An architecture for default and abductive reasoning, Computational Intelligence 5 (1992) 97–110.
[18] R. Reiter, The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression, in: V. Lifschitz (Ed.), Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy, Academic Press, San Diego, CA, 1991, pp. 418–420.
[19] R. Reiter, J. de Kleer, Foundations of ATMS, in: Proc. AAAI-87, Seattle, WA, 1987.
[20] B. Selman, H. Levesque, Abductive and default reasoning: A computational core, in: Proc. AAAI-90, Boston, MA, 1990, pp. 343–348.
[21] H. Simon, Causal ordering and identifiability, in: W. Hood, T. Koopmans (Eds.), Studies in Econometric Method, Wiley, New York, 1953.
[22] P. Tison, Generalized consensus theory and application to the minimization of boolean functions, IEEE Trans. on Electronic Computers 16 (4) (1967) 446–456.
[23] A. Weber, Updating propositional formulas, in: Proc. 1st Conference on Expert Database Systems, 1986, pp. 487–500.

Artificial Intelligence 103(1-2): 273-294 (1998)

Applications of the Situation Calculus To Formalizing Control and Strategic Information: The Prolog Cut Operator

Fangzhen Lin (flin@cs.ust.hk)
Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Abstract

We argue that the situation calculus is a natural formalism for representing and reasoning about control and strategic information. As a case study, in this paper we provide a situation calculus semantics for the Prolog cut operator, the central search control operator in Prolog. We show that our semantics is well-behaved when the programs are properly stratified, and that according to this semantics, the conventional implementation of the negation-as-failure operator using cut is provably correct with respect to the stable model semantics.

1 Introduction

The situation calculus (McCarthy and Hayes [8]) is a formalism for representing and reasoning about actions in dynamic domains. It is a many-sorted predicate calculus with some reserved predicate and function symbols. For example, in the blocks world, to say that block A is initially clear, we write:

$$H(clear(A), S_0),$$

where H is a reserved binary predicate that stands for "holds", and S_0 is a reserved constant symbol denoting the initial situation. As an another example, to say that action $stack(x, y)$ causes $on(x, y)$ to be true, we write:[1]

$$H(on(x, y), do(stack(x, y), s)),$$

where the reserved function $do(a, s)$ denotes the resulting situation of doing the action a in the situation s. This is an example of how the effects of an action can be represented in the situation calculus. Generally, in the situation calculus:

[1]In this paper, free variables in a displayed formula are assumed to be universally quantified.

- situations are first-order objects that can be quantified over;
- a situation carries information about its history, i.e the sequence of actions that have been performed so far. For example, the history of the situation

$$do(stack(A, B), do(stack(B, C), S_0))$$

is $[stack(B, C), stack(A, B)]$, i.e. the sequence of actions that have been performed in the initial situation to reach it. As we shall see later, our foundational axioms will enforce a one-to-one correspondence between situations and sequences of actions.

We believe that these two features of the situation calculus make it a natural formalism for representing and reasoning about control knowledge. For example, in AI planning, a plan is a sequence of actions, thus isomorphic to situations. So control knowledge in planning, which often are constraints on desirable plans, becomes constraints on situations (Lin [6]). Similarly, when we talk about control information in logic programming, we are referring to constraints on derivations, i.e. sequences of actions according to (Lin and Reiter [7]).

Although our long term goal is to develop a general framework for representing and reasoning about control knowledge in problem solving using the situation calculus, our focus in this paper is the Prolog cut operator, the central search control operator in Prolog. We shall provide a situation calculus semantics for logic programs with cut, and show that our semantics is well-behaved when the programs are properly stratified. We also show that according to this semantics, the conventional implementation of the negation-as-failure operator using cut is provably correct with respect to the stable model semantics of Gelfond and Lifschitz [3]. To the best of our knowledge, this is the first time a connection has been shown between a declarative semantics of negation and that of cut.

This paper is organized as follows. Section 2 briefly reviews the basic concepts in the situation calculus and logic programming. Section 3 reviews the situation calculus semantics of (Lin and Reiter [7]) for cut-free logic programs. For the purpose of this paper, the key property of this semantics is that derivations in logic programming are identified with situations. Section 4 extends this semantics to logic programs with cut. This is done by an axiom on *accessible* situations, that is, those situations whose corresponding derivations are not "cut off" by cut. Section 5 shows that our semantics is well-behaved for a class of stratified programs. Section 6 shows that the conventional implementation of negation-as-failure using cut is provably correct. Finally section 7 concludes this paper.

2 Logical Preliminaries

2.1 The Situation Calculus

The language of the situation calculus is a many-sorted second-order one with equality. We assume the following sorts: *situation* for situations, *action* for actions, *fluent* for propositional fluents such as *clear* whose truth values depend on situations, and *object* for everything else. As we mentioned above, we assume that S_0 is a reserved constant denoting the initial situation, H a reserved predicate for expressing properties about fluents in a situation, *do* a reserved binary function denoting the result of performing an action. In addition, we assume the following two partial orders on situations:

- $\sqsubset$: following convention, we write $\sqsubset$ in infix form. By $s \sqsubset s'$ we mean that s' can be obtained from s by a sequence of actions. As usual, $s \sqsubseteq s'$ will be a shorthand for $s \sqsubset s' \vee s = s'$.
- $\subset$: we also write $\subset$ in infix form. By $s \subset s'$ we mean that s can be obtained from s' by deleting some of its actions. Similarly, $s \subseteq s'$ stands for $s \subset s' \vee s = s'$.

We shall consider only the discrete situation calculus with the following foundational axioms:

$$S_0 \neq do(a, s), \tag{1}$$

$$do(a_1, s_1) = do(a_2, s_2) \supset (a_1 = a_2 \wedge s_1 = s_2), \tag{2}$$

$$(\forall P)[P(S_0) \wedge (\forall a, s)(P(s) \supset P(do(a, s))) \supset (\forall s)P(s)], \tag{3}$$

$$\neg s \sqsubset S_0, \tag{4}$$

$$s \sqsubset do(a, s') \equiv s \sqsubseteq s', \tag{5}$$

$$\begin{aligned} s \subset s' \equiv s \neq s' \wedge (\exists f)\{&(\forall s_1, s_2)(s_1 \sqsubseteq s_2 \supset f(s_1) \sqsubseteq f(s_2)) \wedge \\ &(\forall a, s_1)(do(a, s_1) \sqsubseteq s \supset do(a, f(s_1)) \sqsubseteq s')\}. \end{aligned} \tag{6}$$

The first two axioms are unique names assumptions for situations. The third axiom is second order induction. It amounts to the domain closure axiom that every situation has to be obtained from the initial one by repeatly applying the function *do*. In other words, it says that every situation is either the initial situation S_0 or the result of performing a sequence of actions in the initial situation, exactly the isomorphism between situations and sequences of actions that we mentioned earlier. As can be expected, induction will play an important role in this paper.

Axioms (4) and (5) define $\sqsubset$ inductively. The partial order $\sqsubset$ is really the "prefix" relation:[2] Given a situation $S = do([\alpha_1, ..., \alpha_n], S_0)$, $S' \sqsubseteq S$ iff there is a $0 \leq k \leq n$ such that $S' = do([\alpha_1, ..., \alpha_k], S_0)$. In particular, we have $(\forall s)S_0 \sqsubseteq s$.

[2]Given a sequence of actions $[a_1, ..., a_n]$, we use $do([a_1, ..., a_n], s)$ to denote the resulting situation of performing the sequence of actions in s. Inductively, $do([], s) = s$ and $do([a|L], s) = do(L, do(a, s))$.

Figure 1: A function f as required by the axiom (6)

Axiom (6) defines $\subset$. Informally, $s \subseteq s'$ iff s can be obtained from s' by deleting some of its actions. More precisely, suppose $S' = do([\alpha_1, ..., \alpha_n], S_0)$. Then $S \subseteq S'$ iff there are integers $1 \leq i_1 \leq \cdots \leq i_k \leq n$, $0 \leq k \leq n$, such that $S = do([\alpha_{i_1}, ..., \alpha_{i_k}], S_0)$. Figure 1 illustrates a function f as required by axiom (6) for proving the following relation:

$$do([a, c], S_0) \subset do([a, b, c, d], S_0).$$

Notice that $\sqsubset$ is a special case of $\subset$: if $s \sqsubset s'$ then $s \subset s'$. As we shall see, the partial order $\subset$ will play a crucial role in this paper.

In the following, we shall denote by Σ the set of the above axioms (1) - (6).

2.2 Logic Programs

We consider definite logic programs with cut. An *atom* p is a fluent term $F(t_1, ..., t_n)$, where F is a fluent of arity $object^n$, and $t_1, \ldots, t_n$ are terms of sort *object*. A *goal* G is an expression of the form

$$l_1 \ \texttt{\&} \ \cdots \ \texttt{\&} \ l_n$$

where $n \geq 0$, and for each $1 \leq i \leq n$, l_i is either an atom, an equality atom of the form $t = t'$, or `!`.

Without loss of generality, we assume that a *clause (rule)* is an expression of the form

$$F(x_1, ..., x_n) \ \texttt{:-} \ G$$

where F is a fluent of the arity $object^n$, $x_1, ..., x_m$ are distinct variables of sort *object*, and G is a goal. Notice that the more common form of a clause

$$F(t_1, ..., t_n) \ \texttt{:-} \ G\,.$$

can be taken to be a *shorthand* for the following clause:

$$F(x_1, ..., x_n) \ \texttt{:-} \ x_1 = t_1 \ \texttt{\&} \ \cdots \ \texttt{\&} \ x_n = t_n \ \texttt{\&} \ G\,.$$

where $x_1, ..., x_n$ are fresh variables not in G and $t_1, ..., t_n$.

Finally, a (definite) *program* is a finite set of clauses. The *definition* of a fluent symbol F in a program P is the set of clauses in P that have F in their heads.

Since a goal is not a situation calculus formulas, we need a way to refer to its truth values. Given a goal $G = l_1 \ \& \ \cdots \ \& \ l_n$, and a situation term S, we define $H(G, S)$, the truth value of G in the situation S, to be the situation calculus formula

$$H(l_1, S) \wedge \cdots \wedge H(l_n, S),$$

where for each $1 \leq i \leq n$:

1. If l_i is $t = t'$, then $H(l_i, S)$ is l_i.
2. If l_i is `!`, then $H(l_i, S)$ is the tautology $true$.

For example, $H(x = a \ \& \ parent(x, y) \ \& \ !, S_0)$ is

$$x = a \wedge H(parent(x, y), S_0) \wedge true.$$

3 A Logical Semantics

The cut operator in Prolog plays two roles. As a goal, it succeeds immediately. As a search control operator, it prevents a Prolog interpreter from backtracking past it. Consequently, our semantics for programs with cut will come in two stages. First, we consider the "pure logical" semantics of the programs when cut is taken to be a goal that succeeds immediately. For this purpose, we shall use the situation calculus semantics for logic programs without cut proposed by Lin and Reiter ([7]). For our purpose here, the key of this semantics is that program clauses are identified with the effects of actions in the situation calculus, so a branch in a search tree becomes a sequence of actions, thus isomorphically, a situation. This is important because the effect of a cut on the search tree can then be modeled by restrictions on situations. So our second step in formalizing the cut operator is to define a relation call Acc on situations so that $Acc(s)$ holds with respect to a logic program if the sequence of actions in s corresponds to a successful derivation according to the program.

The rest of this section is basically a review of [7] with a minor notational difference: while we reify fluents and use the special predicate H, Lin and Reiter [7] treat fluents as predicate symbols. For example, $H(broken, s)$ would be written as $broken(s)$ in [7].

According to [7], clauses are treated as rules, so that the application of such a rule in the process of answering a query is like performing an action. Formally, given a clause of the form

$$F(x_1, ..., x_n) \ \texttt{:-} \ G$$

Lin and Reiter [7] introduce a unique action A of the arity $object^n \rightarrow action$ to name this clause. The only effect of this action is the following:

$$(\exists \vec{\xi}) H(G, s) \supset H(F(\vec{x}), do(A(\vec{x}), s)),$$

where $\vec{x}$ is $(x_1, ..., x_n)$, and $\vec{\xi}$ is the tuple of variables that are in G but not in $\vec{x}$. For example, suppose $gp(x, y)$ is the action that names the following clause:

```
gparent(x,y) :- parent(x,z) & parent(z,y)
```

then we have the following effect axiom:

$$(\exists z)[H(parent(x, z), s) \wedge H(parent(z, y), s)] \supset H(gparent(x, y), do(gp(x, y), s)).$$

Now suppose that P is a program and F a fluent. Suppose the definition of F in P is

$$\begin{array}{ll} A_1(\vec{x}): & F(\vec{x}) \text{ :- } G_1 \\ \vdots & \vdots \\ A_k(\vec{x}): & F(\vec{x}) \text{ :- } G_k \end{array}$$

where $A_1, ..., A_k$ are the action names for the corresponding clauses. Then we have the following corresponding effect axioms for the fluent F:

$$\begin{array}{c} (\exists \vec{y_1}) H(G_1, s) \supset H(F(\vec{x}), do(A_1(\vec{x}), s)), \\ \vdots \\ (\exists \vec{y_k}) H(G_k, s) \supset H(F(\vec{x}), do(A_k(\vec{x}), s)), \end{array}$$

where $\vec{y_i}$, $1 \leq i \leq k$, is the tuple of variables in G_i which are not in $\vec{x}$. We then generate the following *successor state axiom* (Reiter [10]) for F:

$$\begin{array}{l} H(F(\vec{x}), do(a, s)) \equiv \{a = A_1(\vec{x}) \wedge (\exists \vec{y_1}) H(G_1, s) \vee \cdots \vee \\ \qquad (a = A_k(\vec{x}) \wedge (\exists \vec{y_k}) H(G_k, s)) \vee H(F(\vec{x}), s)\}. \end{array} \tag{7}$$

Intuitively, this axiom says that F is true in a successor situation iff either it is true initially or the action is one that corresponds to a clause in the definition of F and the body of the clause is satisfied initially. In particular, if the definition of F in the program P is empty, then (7) becomes $H(F(\vec{x}), do(a, s)) \equiv H(F(\vec{x}), s)$.

In the following, we call (7) the *successor state axiom for F wrt to P*.

Given a logic program P, the set of successor state axioms wrt P, together with some domain independent axioms, is then the "pure logical meaning" of P:

Definition 1 *The* basic action theory $\mathcal{D}$ for P *is*

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{ss} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}$$

where

- Σ *is the set of foundational axioms given in Section 2.1.*
- $\mathcal{D}_{ss}$ *is the set of successor state axioms for the fluents according to* P*.*
- $\mathcal{D}_{una}$ *is the following set of unique names axioms:*

$$f(\vec{x}) \neq g(\vec{y}) \tag{8}$$

for every pair f, g *of distinct function symbols, and*

$$f(\vec{x}) = f(\vec{y}) \supset \vec{x} = \vec{y} \tag{9}$$

for every function symbol f*. Notice that* $\mathcal{D}_{una}$ *includes unique names axioms for actions.*

- $\mathcal{D}_{S_0}$ *is:*

$$\{H(F(\vec{x}), S_0) \equiv false \mid F \text{ is a fluent}\}.$$

Definition 2 *Let* P *be a program, and* $\mathcal{D}$ *its corresponding basic action theory. A situation term* S *is called a* plan *for a goal* G *iff* $\mathcal{D} \models (\forall \vec{x})H(G, S)$*, where* $\vec{x}$ *are the variables in* G *and* S*.*

Given a query G, we then look for a tuple of terms $\vec{t}$ and a plan S for $G(\vec{t})$. Deductively, this amounts to find a constructive proof of the entailment: $\mathcal{D} \models (\exists \vec{x}, s)H(G, s)$, where $\vec{x}$ are the variables in G. Therefore query answering in logic programs literally becomes planning in the style of (Green [4]) in the situation calculus. This semantics has some nice properties. For example, it generalizes the Clark completion semantics:

Theorem 1 (Lin and Reiter [7]) *Let* P *be a program,* $\mathcal{D}$ *its action theory, and* F *a fluent. Suppose the successor state axiom for* F *in* $\mathcal{D}$ *is (7), and* G_i *is* l_{i1} *&* $\cdots$ *&* l_{ik_i} *for* $1 \leq i \leq n$*. Then* $\mathcal{D}$ *entails the Clark completion for* F*:*

$$\begin{aligned}(\exists s)H(F(\vec{x}), s) \equiv \{&(\exists \vec{y_1})\{(\exists s)H(l_{11}, s) \wedge \cdots \wedge (\exists s)H(l_{1k_1}, s)\} \vee \cdots \vee \\ &(\exists \vec{y_n})\{(\exists s)H(l_{n1}, s) \wedge \cdots \wedge (\exists s)H(l_{nk_n}, s)\}\}.\end{aligned}$$

This semantics is also closely related to a recent proposal by Wallace [12], and in the propositional case, equivalent to Gelfond and Lifschitz's stable model semantics [3]. For details, see (Lin and Reiter [7]).

Example 1 Consider the following program for $max(x, y, z)$, z is the maximum of $\{x, y\}$:[3]

[3]This is a typical example of improper uses of cut. We'll return to this point later.

$A_1(x,y,z)$: `max(x,y,z) :- leq(x,y) & ! & z=y`
$A_2(x,y,z)$: `max(x,y,z) :- z=x`
$B_1(x,y)$: `leq(x,y) :- x=1 & y=2`
$B_2(x,y)$: `leq(x,y) :- x=1 & y=1`
$B_3(x,y)$: `leq(x,y) :- x=2 & y=2`

Here $leq(x,y)$ means that x is less than or equal to y, and 1 and 2 are constants. We have the following successor state axioms for max and leq:

$$\begin{aligned}
&H(max(x,y,z),do(a,s)) \equiv \\
&\quad a = A_1(x,y,z) \wedge H(leq(x,y),s) \wedge z = y \vee \\
&\quad (a = A_2(x,y,z) \wedge z = x) \vee H(max(x,y,z),s), \\
&H(leq(x,y),do(a,s)) \equiv \{a = B_1(x,y) \wedge x = 1 \wedge y = 2 \vee \\
&\quad a = B_2(x,y) \wedge x = 1 \wedge y = 1 \vee \\
&\quad (a = B_3(x,y) \wedge x = 2 \wedge y = 2) \vee H(leq(x,y),s)\}.
\end{aligned}$$

From these successor state axioms, it is easy to see that performing first $B_1(1,2)$, then $A_1(1,2,2)$ in S_0 will result in a situation satisfying $max(1,2,2)$. Thus we have the following desirable conclusion:

$$\mathcal{D} \models (\exists s)H(max(1,2,2),s).$$

On the other hand, it is also clear that the action $A_2(x,y,x)$ will make $max(x,y,x)$ true, so we also have:

$$\mathcal{D} \models (\forall x,y)(\exists s)H(max(x,y,x),s),$$

which is not so desirable. Of course, the reason is that the basic action theory $\mathcal{D}$ does not take into account the effects of `!` on the search space. According to Prolog's search strategy, which attempts rules in the order as they are given, the second rule (A_2) for max will not be attempted if the subgoal $leq(x,y)$ before `!` in the first rule for max succeeds. Therefore the derivation corresponding to $do(A_2(x,y,x),S_0)$ will not be considered if there is a derivation for $leq(x,y)$, which is the case here if $x = 1$ and $y = 2$.

Let us call a situation *accessible* if the derivation corresponding to this situation[4] is legal, i.e. not ruled out by the cut. Our goal in defining a semantics for the cut operator is then to characterize the set of accessible situations. This is what we are going to do in the next section.

4 A Semantics for Cut

A clause containing cut:

[4]In the following, the terms situations and derivations will be used interchangeably.

$$cut(\vec{x}): \quad F(\vec{x}) \ \texttt{:-}\ G_1\ \texttt{\&}\ \texttt{!}\ \texttt{\&}\ G_2$$

means that if G_1 succeeds, then any derivation of $F(\vec{x})$ must use either

1. a rule before this one; or
2. this rule with the first derivation of G_1.

We now proceed to formalize this informal reading.

First, notice that we need two ordering relations: one on rules for deciding the precedence of rules, and the other on situations for defining "the first derivation".

In the following, we shall assume that we are given an ordering $\prec$ on actions (rules), and will define an ordering on situations using $\prec$. Intuitively, if $\alpha \prec \beta$, then during the process of search, the action α will be considered before the action β. For instance, according to Prolog's ordering rule, for our *max* example, $(\forall \vec{x}, \vec{y}) A_1(\vec{x}) \prec A_2(\vec{y})$.

Given a partial order on actions, there are many ways situations, i.e. sequences of actions, can be ordered, depending on particular problem solving strategies. In Prolog, a query is answered using a goal-directed search strategy, so if a plan $do([\alpha_1, ..., \alpha_n], S_0)$ is returned, then it must be the case that α_n is first decided, then α_{n-1}, ..., and finally α_1. If we read $s \prec s'$ as that the sequence of actions in s is considered before that in s', then we have the following definition:

Definition 3 *Given a partial order $\prec$ on actions, the derived partial order with the same name $\prec$ on situations is defined by the following axiom:*

$$s \prec s' \equiv (\exists a, b, s_1, s_2)\{s = do(a, s_1) \wedge s' = do(b, s_2) \wedge [a \prec b \vee (a = b \wedge s_1 \prec s_2)]\}. \tag{10}$$

With this partial order on situations, we can then say, roughly, that a situation is a "first derivation" of a goal G if it is a derivation of G that is *minimal* according to $\prec$. However, to make this precise we first need a space of derivations to define the notion of minimality. We do not want it to be the set of all plans for G because according to our definition, if $\mathcal{D}$ is a basic action theory for a logic program, then

$$\mathcal{D} \models (\forall a, s).H(G, s) \supset H(G, do(a, s)).$$

So if s is a plan for G, then for any action a, $do(a, s)$ is also a plan for G. This means that a plan for G can always be made "smaller" according to $\prec$ by appending to it some irrelevant actions. To avoid this problem, we define the notion of *minimal plans*:

Definition 4 *For any situation term S, any goal G, we denote by* `minimal`(G, S) *the following formula:*

$$H(G, S) \wedge \neg(\exists s)[s \subset S \wedge H(G, s)].$$

Intuitively, $\texttt{minimal}(G, S)$ holds if no actions in it can be deleted for it continue to be a plan of G. For our *max* program in section 3, it is easy to see that

$$\mathcal{D} \models \texttt{minimal}(leq(1,2), do(B_1(1,2), S_0)).$$

But

$$\mathcal{D} \not\models \texttt{minimal}(leq(1,2), do([B_1(1,2), A_1(1,2,2)], S_0)),$$

because $do(B_1(1,2), S_0) \subset do([B_1(1,2), A_1(1,2,2)], S_0)$. It can be seen that for any goal G, if there is a plan for it, then there is a minimal plan for it:

$$\Sigma \models (\forall s).H(G,s) \supset (\exists s')(s' \subseteq s \wedge \texttt{minimal}(G, s')).$$

We are now ready to formalize the informal reading of a clause containing ! at the beginning of this section. For this purpose, we introduce a new predicate $Acc(s)$, meaning that s is not "cut off" by cut.

If a program contains the cut rule $cut(\vec{x})$, and the goal G_1 succeeds, then for $Acc(s)$ to hold, it must be the case that for every minimal plan s_1 of F in s, either it uses a rule before this cut rule or it uses this cut rule with the first derivation of G_1:

$$\begin{array}{l} Acc(s) \supset (\forall \vec{x})\{(\exists s')[Acc(s') \wedge (\exists \vec{\xi}) H(G_1, s')] \supset \\ \quad (\forall s_1)[s_1 \subseteq s \wedge \texttt{minimal}(F(\vec{x}), s_1) \supset \\ \qquad (\exists a, s_2)(s_1 = do(a, s_2) \wedge a \prec cut(\vec{x})) \vee \\ \qquad (\exists s_2, s_3)(s_1 = do(cut(\vec{x}), s_2) \wedge s_3 \subseteq s_2 \wedge \texttt{first-der}(G_1, s_3))]\}, \end{array} \tag{11}$$

where $\vec{\xi}$ is the tuple of the free variables that are in G_1 but not in $\vec{x}$, and $\texttt{first-der}(G_1, s_3)$ stands for the following formula:

$$(\exists \vec{\xi})\texttt{minimal}(G_1, s_3) \wedge \neg(\exists s_3')[s_3' \prec s_3 \wedge Acc(s_3') \wedge (\exists \vec{\xi})\texttt{minimal}(G_1, s_3')],$$

where $\vec{\xi}$ is the tuple of variables that are in G_1 but not in $\vec{x}$. We remark that in the axiom (11):

- The formula
$$(\exists s')(Acc(s') \wedge (\exists \vec{\xi}) H(G_1, s'))$$
corresponds to "G_1 succeeds";

- The disjunct
$$(\exists a, s_2)(s_1 = do(a, s_2) \wedge a \prec cut(\vec{x}))$$
corresponds to "the derivation s_1 of $F(\vec{x})$ uses a rule before the cut rule";

- The disjunct

$$(\exists s_2, s_3)(s_1 = do(cut(\vec{x}), s_2) \wedge s_3 \subseteq s_2 \wedge \texttt{first-der}(G_1, s_3))$$

corresponds to "uses this rule with the first derivation of G_1". Notice that we cannot replace the above formula by the following more straightforward one:

$$(\exists s_2)(s_1 = do(cut(\vec{x}), s_2) \wedge \texttt{first-der}(G_1, s_2))$$

because s_2 may not be a minimal plan for G_1.

In the following, we call sentence (11) the *accessibility constraint* corresponding to the occurrence of `!` in clause $cut(\vec{x})$ after the subgoal G_1. Notice that a clause may have more than one occurrences of `!`, thus generate more than one accessibility constraints.

Now given a program P, suppose

$$\begin{aligned} Acc(s) &\supset \Psi_1(s), \\ &\vdots \\ Acc(s) &\supset \Psi_k(s), \end{aligned}$$

are all the accessibility constraints. We call the following sentence *the accessibility axiom* of P:

$$Acc(s) \equiv \Psi_1(s) \wedge \cdots \wedge \Psi_k(s). \tag{12}$$

Notice that this axiom attempts to define Acc recursively since the predicate also occurs in the right hand side of the equivalence. This should not be surprising. For example, the logical formalization of negation in logic programs normally requires fixed-point constructions, and negation is usually implemented by cut.

Definition 5 (Extended Action Theory) *Let P be a program, and Δ a given set of axioms about $\prec$ on actions.* The extended action theory $\mathcal{E}$ *of P is the the following set:*

$$\mathcal{E} = \mathcal{D} \cup \Delta \cup \{Acc, (10)\},$$

where $\mathcal{D}$ is the basic action theory for P, and Acc is the accessibility axiom of P of the form (12).

Definition 6 *Let P be a program, $\mathcal{E}$ its extended action theory, and G a goal. A situation term S is an* accessible *plan for G iff $\mathcal{E} \models (\forall \vec{x}).Acc(S) \wedge H(G, S)$, where $\vec{x}$ are the variables in G and S.*

Now given a query G, we answer it by looking for a constructive proof of the following entailment:

$$\mathcal{E} \models (\exists \vec{x}, s).Acc(s) \wedge H(G, s),$$

where $\vec{x}$ are the variables in G. Notice that when G contains variables, this method of answering queries does not exactly agree with Prolog's. It is well-known that in the presence of cut, according to most Prolog implementations, a program may answer a query like $F(a)$ positively but fail to return $x = a$ as a possible answer when presented with the query $F(x)$. However, this oddity of Prolog does not show up here: $x = a$ will be returned as an answer according to our construction because an accessible plan for $F(a)$ is a constructive proof of $(\exists x, s)Acc(s) \wedge H(F(x), s)$.

We illustrate the definitions with our max example. Suppose we use Prolog's search strategy, and order the actions as:

$$\Delta = \{a \prec b \equiv (\exists \vec{x})a = A_1(\vec{x}) \wedge (\exists \vec{y})b = A_2(\vec{y})\}.$$

Notice that we do not care about how the rules about leq are ordered. Since the only occurrence of `!` is in A_1, the accessibility axiom is

$$\begin{aligned} Acc(s) \equiv (\forall x, y, z)\{&(\exists s')(Acc(s') \wedge H(leq(x, y), s')) \supset \\ &(\forall s_1)[s_1 \subseteq s \wedge \texttt{minimal}(max(x, y, z), s_1) \supset \\ &\quad (\exists a, s_2)(s_1 = do(a, s_2) \wedge a \prec A_1(x, y, z)) \vee \\ &\quad (\exists s_2, s_3)(s_1 = do(A_1(x, y, z), s_2) \wedge s_3 \subseteq s_2 \wedge \texttt{first-der}(leq(x, y), s_3))]\}. \end{aligned} \tag{13}$$

Now let $\mathcal{E}$ be the extended action theory of the program. First of all, we can show that

$$\mathcal{E} \models (\forall x, y, z, s).\texttt{minimal}(leq(x, y), s) \supset [Acc(s) \wedge \neg H(max(x, y, z), s)],$$

thus

$$\mathcal{E} \models (\forall x, y)[(\exists s)H(leq(x, y), s) \equiv (\exists s)(Acc(s) \wedge H(leq(x, y), s))].$$

This is intuitively right since the only appearance of `!` is in the definition of max, and the presence of `!` has no effect on leq.

Now let $S_1 = do(B_1(1, 2), S_0)$, and $S_2 = do(A_1(1, 2, 2), S_1)$. Clearly

$$\mathcal{E} \models H(leq(1, 2), S_1) \wedge \texttt{minimal}(max(1, 2, 2), S_2).$$

We claim that $\mathcal{E} \models Acc(S_2)$ so that S_2 is an accessible plan for $max(1, 2, 2)$. Notice first that

$$\begin{aligned} &\mathcal{E} \models (\exists s).Acc(s) \wedge H(leq(1, 2), s), \\ &\mathcal{E} \models (\forall x, y, z).H(max(x, y, z), S_2) \supset (x = 1 \wedge y = 2 \wedge z = 2), \\ &\mathcal{E} \models \neg(\exists a, s).S_2 = do(a, s) \wedge a \prec A_1(1, 2, 2). \end{aligned}$$

Therefore by (13)

$$\mathcal{E} \models Acc(S_2) \equiv (\exists s_3)(s_3 \subseteq S_1 \wedge \texttt{first-der}(leq(1, 2), s_3)).$$

Let s_3 be S_1, then $\mathcal{E} \models Acc(S_2)$ if

$$\mathcal{E} \models \texttt{first-der}(leq(1,2), S_1),$$

that is

$$\mathcal{E} \models \texttt{minimal}(leq(1,2), S_1) \wedge \neg(\exists s).Acc(s) \wedge \texttt{minimal}(leq(1,2), s) \wedge s \prec S_1.$$

This follows from

$$\mathcal{E} \models (\forall s).\texttt{minimal}(leq(1,2), s) \equiv s = S_1.$$

So we have proved that $\mathcal{E} \models Acc(S_2)$.

On the other hand, there are no accessible plans for $max(1,2,1)$. The proof is as follows. Suppose for some situation term S, $\mathcal{E} \models max(1,2,1,S)$. Then it must be the case that $do(A_2(1,2,1), S_0) \subseteq S$. Since

$$\begin{aligned}
&\mathcal{E} \models (\exists s).Acc(s) \wedge H(leq(1,2), s),\\
&\mathcal{E} \models (\forall s).\texttt{minimal}(max(1,2,1), s) \equiv s = do(A_2(1,2,1), S_0),
\end{aligned}$$

it follows from (13) that for $Acc(S)$ to hold, it must be the case that

$$\begin{aligned}
&(\exists a, s)[do(A_2(1,2,1), S_0) = do(a, s) \wedge a \prec A_1(1,2,1)] \vee\\
&(\exists s, s')[do(A_2(1,2,1), S_0) = do(A_1(1,2,1), s) \wedge s' \subseteq s \wedge \texttt{first-der}(leq(1,2), s')],
\end{aligned}$$

which is obviously impossible.

Generalizing the above reasoning, we have:

$$\begin{aligned}
\mathcal{E} \models (\forall x, y)\{&(\exists s)H(leq(x,y), s) \wedge x \neq y \supset\\
&[(\exists s)(Acc(s) \wedge H(max(x,y,y), s)) \wedge \neg(\exists s)(Acc(s) \wedge H(max(x,y,x), s))]\}.
\end{aligned}$$

So the max program indeed defines max correctly. However since

$$\mathcal{E} \not\models (\forall x, y, z)[(\exists s)H(max(x,y,z), s) \equiv (\exists s)(Acc(s) \wedge H(max(x,y,z), s))],$$

this use of cut is not ideal in the sense that it does more than search control, i.e. it is a "red" cut.

5 Stratified Programs

As we have noticed, the accessibility axiom attempts to define Acc recursively. A natural question then is if the recursion will yield a unique solution for the predicate. In general, the answer is negative. However, if a program is properly stratified, then the axiom will yield a unique solution. To show this, we first present two simple lemmas about our axiomatization of cut.

The following lemma says that if s is not a derivation of F, then whether s is accessible has nothing to do with rules about F:

Lemma 1 *Let P be a program, and $\mathcal{E}$ its extended action theory. Suppose the accessibility axiom for P is as (12):*

$$Acc(s) \equiv \Psi_1(s) \wedge \cdots \wedge \Psi_n(s).$$

Let F be a fluent. Without loss generality, let

$$(\forall s)Acc(s) \supset \Psi_1,\ ...,\ (\forall s)Acc(s) \supset \Psi_k$$

be the accessibility constraints of the form (11) for all of the occurrences of ! *in the definition of F. Under $\mathcal{E}$, we have:*

$$\neg(\exists \vec{x})H(F(\vec{x}), s) \supset [Acc(s) \equiv \Psi_{k+1} \wedge \cdots \wedge \Psi_n]. \tag{14}$$

Proof: Under $\mathcal{E}$, if a situation is not a derivation for an atom, then no part of it can be a derivation:

$$\neg H(F(\vec{x}), s) \wedge s' \subseteq s \supset \neg H(F(\vec{x}), s').$$

Therefore if $\Psi(s)$ is the right hand side of (11), then

$$\neg(\exists \vec{x})H(F(\vec{x}), s) \supset \Psi(s).$$

From this, (14) follows. ■

The following lemma says that if there is an accessible plan for G, then there is a accessible minimal plan for G.

Lemma 2 *Let P be a program and $\mathcal{E}$ its extended action theory. For any goal G, we have:*

$$(\exists s)(Acc(s) \wedge H(G, s)) \equiv (\exists s)(Acc(s) \wedge \texttt{minimal}(G, s)).$$

Proof: By induction on situations, if there is a plan for G, then there is a minimal plan for it:

$$\Sigma \models (\forall s).H(G, s) \supset (\exists s')(s' \subseteq s \wedge \texttt{minimal}(G, s')).$$

From this and the fact that $(\forall s, s')(Acc(s) \wedge s' \subseteq s \supset Acc(s'))$, the lemma follows. ■

Let P be a program, and F a fluent. We say that the definition of F in P is *cut-free* if none of the clauses that are *relevant* to F contains !. Here a clause is relevant to F if, inductively, either it's in the definition of F or it's relevant to another fluent that appears in the definition of F. For example, the definition of *leq* in the *max* example is cut-free. For cut-free fluents, *Acc* does not play a role:

Proposition 1 *Let P be a program, and $\mathcal{E}$ its extended action theory. For any goal G, if the definition of each fluent F mentioned in G is cut-free, then under $\mathcal{E}$ we have:*

$$\texttt{minimal}(G, s) \supset Acc(s),$$
$$(\exists s)H(G, s) \equiv (\exists s)(Acc(s) \wedge H(G, s)).$$

Proof: If `minimal`(G, s), then s cannot be a derivation for any fluent that is not relevant to a fluent in G. Since every fluent in G is cut-free, this means that s cannot be a derivation for any fluent whose definition contains a rule that mentions cut. From this, the desired result follows directly from Lemma 1. ■

Cut-free fluents are the ground case of stratified programs:

Definition 7 *A program P is* stratified *if there is a function f from fluents in P to natural numbers such that*

1. *If F is cut-free in P, then $f(F) = 0$.*
2. *If F is not cut-free, then $f(F)$ is*

$$1 + max\{f(F') \mid F' \text{ appears in the definition of } F\}.$$

It is clear that if P is stratified, then for any fluent F in P, $f(F)$ is uniquely determined by the above two rules. In such case, we shall call the uniquely determined number the *rank* of F in P, and the maximum of the ranks of all the fluents in P the rank of the program P. For instance, in our *max* example, the rank of *leq* is 0, and the rank of *max* is 1. So the rank of the program is 1.

We now show that if P is stratified, then its accessibility axiom can be written equivalently in a recursion-free form. To that end, we introduce a new set of unary predicates $Acc_0, Acc_1, .., Acc_n, ...$ Intuitively, $Acc_n(s)$ holds if s is accessible (reachable) when all rules that mention a fluent of rank $> n$ are deleted from P. Formally, they are defined as follows:

$$Acc_0(s) \equiv true. \tag{15}$$

For $n > 0$, suppose

$$\begin{array}{ll} cut_1(\vec{x_1}): & F_1(\vec{x_1}) \texttt{ :- } G_1 \texttt{ \& ! \& } G_1'\texttt{.} \\ \vdots & \vdots \\ cut_k(\vec{x_k}): & F_k(\vec{x_1}) \texttt{ :- } G_k \texttt{ \& ! \& } G_k'\texttt{.} \end{array}$$

are all of the occurrences of `!` in the definitions of the fluents of rank n. Then Acc_n is defined by the following axiom:

$$Acc_n(s) \equiv Acc_{n-1}(s) \wedge \Phi_1(s) \wedge \cdots \wedge \Phi_k(s), \tag{16}$$

where for any $1 \leq i \leq k$, $\Phi_i(s)$ is obtained as follows: Suppose

$$(\forall s).Acc(s) \supset \Psi_i(s)$$

is the accessibility constraint of the form (11) for the occurrence of `!` in the rule $cut_i(\vec{x_i})$, then $\Phi_i(s)$ is the result of replacing every occurrence of Acc in $\Psi_i(s)$ by Acc_{n-1}.

Now the axiom (16) defines Acc_n inductively since its right hand mentions only Acc_{n-1}. We have:

Theorem 2 *Let P be a stratified program, M its rank, and $\mathcal{E}$ its corresponding extended action theory. We have*

$$\mathcal{E} \models Acc(s) \equiv Acc_M(s).$$

Proof: We prove by induction that for any goal G that mentions only fluents of rank $\leq i$, we have

$$(\exists s)[Acc(s) \wedge H(G,s)] \equiv (\exists s)[Acc_i(s) \wedge H(G,s)]. \tag{17}$$

For $i = 0$, this follows from Proposition 1. Inductively, suppose that this holds for i. We show that it holds for $i+1$ as well.

By Lemma 2,

$$(\exists s)[Acc(s) \wedge H(G,s)] \equiv (\exists s)[Acc(s) \wedge \mathtt{minimal}(G,s)].$$

Let

$$(\forall s)(Acc(s) \supset \Psi_l(s)),\ ...,\ (\forall s)(Acc(s) \supset \Psi_m(s))$$

be all of the accessibility constraints for the occurrences of `!` in the definitions of fluents of rank $\leq n+1$. By Lemma 1 and the fact that if $\mathtt{minimal}(G,s)$, then s cannot be a derivation for any fluents of rank $> n+1$, we have

$$(\exists s)[Acc(s) \wedge \mathtt{minimal}(G,s)] \equiv (\exists s)[\Psi_l(s) \wedge \cdots \wedge \Psi_m(s) \wedge \mathtt{minimal}(G,s)].$$

According to the way that $\Phi_l(s), ..., \Phi_m(s)$ are obtained from $\Psi_l(s), ..., \Psi_l(s)$, respectively, by inductive assumption, we have

$$(\exists s)[\Psi_l(s) \wedge \cdots \wedge \Psi_m(s) \wedge \mathtt{minimal}(G,s)] \equiv (\exists s)[\Phi_l(s) \wedge \cdots \wedge \Phi_m(s) \wedge \mathtt{minimal}(G,s)].$$

But by the definition of $Acc_{n+1}(s)$,

$$Acc_{n+1}(s) \equiv \Phi_l(s) \wedge \cdots \wedge \Phi_m(s).$$

Therefore (17) holds for $i+1$ as well. This completes our inductive proof, and thus the proof that (17) holds for all i. From (17) the theorem follows immediately. ■

6 Negation-as-Failure by Cut

In last section we showed that for stratified programs, our axiom for *Acc* yields a unique solution. As can be expected, this does not hold in general. Consider the following program:

r_1: `p :- q'.`
r_2: `q :- p'.`
r_3: `p' :- p & ! & fail.`
r_4: `p'.`
r_5: `q' :- q & ! & fail.`
r_6: `q'.`

Notice that the definition of $fail$ is empty, so $(\forall s)\neg H(fail, s)$. This program is clearly not stratified. Its accessibility axiom is:

$$\begin{aligned}
&Acc(s) \equiv \\
&\quad (\exists s')(Acc(s') \wedge H(p, s')) \supset (\forall s_1)[s_1 \subseteq s \wedge \texttt{minimal}(p', s_1) \supset \\
&\qquad (\exists a, s_2)(s_1 = do(a, s_2) \wedge a \prec r_3) \vee \\
&\qquad (\exists s_2, s_3)(s_1 = do(r_3, s_2) \wedge s_3 \subseteq s_2 \wedge \texttt{first-der}(p, s_3))] \wedge \\
&\quad (\exists s')(Acc(s') \wedge H(q, s')) \supset (\forall s_1)[s_1 \subseteq s \wedge \texttt{minimal}(q', s_1) \supset \\
&\qquad (\exists a, s_2)(s_1 = do(a, s_2) \wedge a \prec r_5) \vee \\
&\qquad (\exists s_2, s_3)(s_1 = do(r_5, s_2) \wedge s_3 \subseteq s_2 \wedge \texttt{first-der}(q, s_3))].
\end{aligned}$$

Since r_4 is the only rule by which p' can be established, we have:

$$\texttt{minimal}(p', s) \supset s = do(r_4, S_0).$$

Similarly, $(\forall s)[\texttt{minimal}(q', s) \supset s = do(r_6, S_0)]$. Thus if we assume the following ordering on the rules:

$$\Delta = \{(\forall a, b)[a \prec b \equiv (a = r_3 \wedge b = r_4) \vee (a = r_5 \wedge b = r_6)]\},$$

then we have

$$\begin{aligned}
&Acc(s) \equiv \\
&\quad [H(p', s) \supset \neg(\exists s')(Acc(s') \wedge H(p, s'))] \wedge [H(q', s) \supset \neg(\exists s')(Acc(s') \wedge H(q, s'))].
\end{aligned}$$

Let $\mathcal{E}$ be the extended action theory of this program, we then have

$$\begin{aligned}
\mathcal{E} \models\ & (\exists s)(Acc(s) \wedge H(p, s)) \wedge \neg(\exists s)(Acc(s) \wedge H(q, s)) \vee \\
& (\exists s)(Acc(s) \wedge H(q, s)) \wedge \neg(\exists s)(Acc(s) \wedge H(p, s)),
\end{aligned}$$

that is, there is an accessible plan for p iff there is not an accessible plan for q, and the other way around as well.

Notice that the program is a rendition of the following logic program:

p `:- not` q`.`
q `:- not` p`.`

with negation implemented by *cut* as:

```
not F :- F & ! & fail.
not F.
```

If we identify an answer set (Gelfond and Lifschitz [3]) with a set of atoms that have accessible derivations, then for this program, our semantics agrees with that of ([3]). Our following theorem shows that this equivalence holds for arbitrary normal programs as well.

Let P be a logic program with negation (`not`) but without cut. Suppose that for each fluent F in P, F' is a new fluent of the same arity. Let P' be the logic program obtained by replacing every literal of the form `not` $F(\vec{t})$ in P by $F'(\vec{t})$, and by adding, for each new fluent F', the following two clauses:

$A_F(\vec{x})$: $\quad F'(\vec{x})$ `:-` $F(\vec{x})$ `& ! & fail.`
$A'_F(\vec{x})$: $\quad F'(\vec{x})$`.`

Suppose that for each fluent F, the action A_F is ordered before the action A'_F.

Theorem 3 *Let $\mathcal{E}$ be the extended action theory of P', and $\mathcal{D}$ the action theory for P as defined in (Lin and Reiter [7]). For any fluent F in P, and any tuple $\vec{t}$ of terms, we have:*

1. *If M is a model of $\mathcal{D}$, then there is a model M' of $\mathcal{E}$ such that $M \models (\exists s)H(F(\vec{t}), s)$ iff $M' \models (\exists s).Acc(s) \wedge H(F(\vec{t}), s)$.*
2. *If M' is a model of $\mathcal{E}$, then there is a model M' of $\mathcal{D}$ such that $M \models (\exists s)H(F(\vec{t}), s)$ iff $M' \models (\exists s).Acc(s) \wedge H(F(\vec{t}), s)$.*

Proof: The proof of this theorem is given in Appendix A because it is relatively long, and needs some results from [7]. ■

From this theorem, we conclude that the usual implementation of negation using cut is correct with respect to the semantics given in (Lin and Reiter [7]). As noted in (Lin and Reiter [7]), the semantics given there for logic programs with negation yields the same results as that given in (Wallace [12]), and the latter has been shown to be equivalent to the stable model semantics of (Gelfond and Lifschitz [3]) when only Herbrand models are considered. Therefore we can also conclude that the usual implementation of negation in terms of cut is correct with respect to the stable model semantics for logic programs with negation in the propositional case:

Corollary 4 *Let P be a propositional logic program with negation but without cut. Let P' be the logic program obtained from P as described above. For any set W of atoms, W is an answer set of P according to Gelfond and Lifschitz iff there is a model M of the extended action theory of P' such that $W = \{p \mid M \models (\exists s).Acc(s) \wedge H(p, s)\}$.*

7 Concluding Remarks

We have applied the situation calculus to logic programming by giving a semantics to programs with cut. We have shown that this semantics has some desirable properties: it is well-behaved when the program is stratified, and that according to this semantics, the usual implementation of negation-as-failure operator by cut is provably correct with respect to Gelfond and Lifschitz's stable model semantics.

There has been relatively few formal work on cut compared to that on negation in logic programming. Previous work on the semantics of cut includes an operational semantics by Podelski and Smolka [9], a denotational semantics by Jones and Mycroft [5], a dynamic algebra semantics by Börger [2], a paralogical semantics by Andrews [1] and a simple completion-style semantics by Stroetmann and Glaß[11] for a very special class of programs. Comparisons among these different semantics and between our semantics and the others have been difficult for two reasons: most of the semantics have been mathematically quite involved, and the formalisms used are very different. However, as part of our future work, we shall attempt to show that our semantics and the one given by Stroetmann and Glaß are equivalent for the special class of programs that they considered. More importantly, we shall try to use this semantics to clarify the proper roles of cut in logic programming, and to study the possibility of a better control mechanism in logic programming.

Our long term goal is to use the situation calculus as a general framework for representing and reasoning about control and strategic information in problem solving. In this regard, we have made some preliminary progress in applying the situation calculus to formalizing control knowledge in planning. As we mentioned in Section 1, in AI planning, a plan is a sequence of actions, thus isomorphic to situations. So control knowledge in planning, which often are constraints on desirable plans, becomes constraints on situations. Based on this idea, in (Lin [6]), we formulate precisely a subgoal ordering in planning in the situation calculus, and show how information about this subgoal ordering can be deduced from a background action theory, and, for both linear and nonlinear planners, how knowledge about this ordering can be used in a provably correct way to avoid unnecessary backtracking. We believe other control knowledge in planning can be similarly axiomatized and made into good use as well.

Acknowledgements

Much of this work was done while I was with the Cognitive Robotics Group in the Department of Computer Science at the University of Toronto, and I thank Ray Reiter and Hector Levesque for making this possible.

This work was also supported in part by the Research Grants Council of Hong Kong under Direct Allocation Grant DAG96/97.EG34 and under Competitive Earmarked Research Grant HKUST6091/97E.

I would like to thank Eyal Amir, Yves Lespérance, Hector Levesque, and especially Ray Reiter for helpful discussions relating to the subject of this paper and/or comments on earlier versions of this paper.

References

[1] J. Andrews. A paralogical semantics for the prolog cut. In *Proc. of Int. Logic Programming Symposium*, pages 591–605, 1995.

[2] E. Börger. A logical operational semantics of full Prolog. In *Computer Science Logic CSL'89*, pages 36–64. Springer-Verlag Lecture Notes in Computer Science 440, 1989.

[3] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Fifth International Conference and Symposium on Logic Programming*, pages 1070–1080, 1988.

[4] C. C. Green. Application of theorem proving to problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI–69)*, pages 219–239, 1969.

[5] N. D. Jones and A. Mycroft. stepwise development of operational and denotational semantics for Prolog. In *Proc. of 1984 International Symposium on Logic Programming*, pages 281–288, 1984.

[6] F. Lin. An ordering on subgoals for planning. *Annals of Mathematics and Artificial Intelligence*, 21:321–342, 1997.

[7] F. Lin and R. Reiter. Rules as actions: A situation calculus semantics for logic programs. *J. of Logic Programming*, 31(1-3):299–330, 1997.

[8] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

[9] A. Podelski and G. Smolka. operational semantics of constraint logic programs with cocoutining. In *Proc. of the 1995 International Conf. on Logic Programming*, pages 449–463. Kanagawa, Japan, 13-18 June 1995. The MIT Press, 1995.

[10] R. Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 418–420. Academic Press, San Diego, CA, 1991.

[11] K. Stroetmann and T. Glaß. A declarative semantics for the prolog cut operator. In *Proc. of 5th Int. Workshop on Extensions of Logic Programming*, pages 255–271, 1996.

[12] M. G. Wallace. Tight, consistent, and computable completions for unrestricted logic programs. *Journal of Logic Programming*, 15:243–273, 1993.

Appendix A

This appendix proves Theorem 3. Recall that in this theorem, P is a logic program with negation but without cut, and P' is the logic program obtained by replacing every literal of the form `not` $F(\vec{t})$ in P by $F'(\vec{t})$, and by adding, for each new fluent F', the following two clauses:

$A_F(\vec{x})$: $\quad F'(\vec{x})$ `:-` $F(\vec{x})$ `& ! & fail.`
$A'_F(\vec{x})$: $\quad F'(\vec{x})$.

We also assume that for each fluent F, the action A_F is ordered before the action A'_F.

In Section 3, we review the action theory semantics of ([7]) for logic programs without negation. Action theories for programs with negation have the same form but with the following semantics for negation: $H($`not` $p, s)$ is $\neg(\exists s')H(p, s)$. Notice that $H($`not` $p, s)$ is in fact independent of the situation argument. For example, given the following clauses:

$r_1(x)$: $\quad$ `p(x) :- not q(x)`
$r_2(x)$: $\quad$ `p(x) :- x=u`

the successor state axiom for F is:

$$H(p(x), do(a, s)) \equiv a = r_1(x) \wedge H(\texttt{not } q(x), s) \vee$$
$$(a = r_2(x) \wedge x = u) \vee H(p(x), s).$$

Theorem 3 *Let $\mathcal{E}$ be the extended action theory of P', and $\mathcal{D}$ the action theory for P as defined in (Lin and Reiter [7]). For any fluent F in P, and any tuple $\vec{t}$ of terms of sort object, we have:*

1. *If M is a model of $\mathcal{D}$, then there is a model M' of $\mathcal{E}$ such that $M \models (\exists s)H(F(\vec{t}), s)$ iff $M' \models (\exists s).Acc(s) \wedge H(F(\vec{t}), s)$.*
2. *If M' is a model of $\mathcal{E}$, then there is a model M' of $\mathcal{D}$ such that $M \models (\exists s)H(F(\vec{t}), s)$ iff $M' \models (\exists s).Acc(s) \wedge H(F(\vec{t}), s)$.*

Proof: Suppose that $F'_1, ..., F'_k$ are the new fluents in P'. Observe first that, as for the example in Section 6, we can show that

$$\mathcal{E} \models Acc(s) \equiv \tag{18}$$
$$(\forall \vec{x_1})\{H(F'_1(\vec{x_1}, s) \supset \neg(\exists s')[Acc(s') \wedge H(F_1(\vec{x_1}), s')]\} \wedge \cdots \wedge$$
$$(\forall \vec{x_k})\{H(F'_k(\vec{x_k}), s) \supset \neg(\exists s')[Acc(s') \wedge H(F_k(\vec{x_k}), s')]\}.$$

Suppose now that M is a model of $\mathcal{D}$. Construct an interpretation M' as follows:

1. The domain of M' for sort *object* is the same as that in M.

2. The domain of M' for sort *action* is the union of the action domain of M and the following set:

$$\{A_{F_i}(\vec{d}), A'_{F_i}(\vec{d}) \mid 1 \leq i \leq k, \vec{d} \text{ is a tuple of elements in the object domain}\}$$

3. The truth values of fluents are always false in S_0, and computed according to the successor state axioms of $\mathcal{E}$ in successor situations.
4. The interpretation for Acc is defined as follows: For any $\hat{s}$ in the situation domain of M',[5]

$$M' \models Acc(\hat{s})$$

iff for any $1 \leq i \leq k$ and any tuple $\vec{u_i}$ of elements in the object domain, whenever

$$M' \models H(F'_i(\vec{u_i}), \hat{s})$$

then

$$M \models \neg(\exists s')F_i(\vec{u_i}, s').$$

We need to show that M' is a model of $\mathcal{E}$, and that for any fluent F and any tuple $\vec{u}$ of objects,

$$\begin{array}{l} M \models (\exists s)H(F(\vec{u}), s) \\ \quad \Longleftrightarrow \\ M' \models (\exists s)Acc(s) \wedge H(F(\vec{u}), s) \end{array} \tag{19}$$

By the construction of M', the fact that M' is a model of $\mathcal{E}$ follows directly from (18) and the equivalence (19). We now prove the equivalence. Suppose

$$M \models (\exists s)H(F(\vec{u}), s).$$

Then for some $\hat{s}$ in the situation domain of M, $M \models \texttt{minimal}(F(\vec{u}), \hat{s})$. By induction on situations, for some actions $a_1, ..., a_n$, $\hat{s} = do([a_1, ..., a_n], S_0)$. Because none of the fluents are true in the initial situation S_0, by the form of the successor state axioms in $\mathcal{D}$, a_i must be the action name for some of the rules in P, for each $1 \leq i \leq n$. Furthermore, since $\hat{s}$ is a minimal plan for $F(\vec{u})$, for each i, the body of a_i must be satisfied in the situation $do([a_1, .., a_{i-1}], S_0)$. Now suppose

$$\texttt{not } F_{l_1}(\vec{u_{l_1}}), ..., \texttt{not } F_{l_m}(\vec{u_{l_m}})$$

[5]To simplify our presentation, if u is an element in the domain of a model M and $\varphi(x)$ is a formula, then we'll use $M \models \varphi(u)$ to stand for $M, \sigma \models \varphi(x)$, where σ is a variable assignment such that $\sigma(x) = u$.

are the negative literals appearing in the rules corresponding to $a_1, ..., a_n$. Then let $\hat{s'}$ be the situation that uses first the rules A'_F for the new fluents corresponding to the above literals, then the rules in s:

$$\hat{s'} = do([A'_{F_{l_1}}(\vec{u_{l_1}}), ..., A'_{F_{l_m}}(\vec{u_{l_m}}), a_1, .., a_n], S_0). \tag{20}$$

By the construction of M', it can be verified that $\hat{s'}$ is in the situation domain of M', and $M' \models Acc(\hat{s'}) \wedge H(F(\vec{u}), \hat{s'})$.

Conversely, suppose

$$M' \models (\exists s) Acc(s) \wedge H(F(\vec{u}), s).$$

Then there are $a_1, ..., a_n$ in the action domain of M' such that

$$M' \models Acc(do([a_1, ..., a_n], S_0)) \wedge \texttt{minimal}(F(\vec{u}), do([a_1, ..., a_n], S_0)).$$

Because we don't have any rules about $fail$, this means that a_i's cannot be $A_{F_l}(\vec{u})$ for any $1 \leq l \leq k$. If a_i is $A'_{F_l}(\vec{u})$, then $F'_l(\vec{u})$ must be true in $do([a_1, ..., a_n], S_0)$. Since this situation is accessible in M', by the construction of M', $M \models \neg(\exists s) H(F(\vec{u}), s)$. This means that if $\xi = [a_{l_1}, ..., a_{l_m}]$ is the list resulted from deleting the actions of the form $A'_{F_l}(\vec{u})$ in $[a_1, ..., a_n]$, then $do(\xi, S_0)$ is a situation in the domain of M, and $M \models H(F(\vec{u}), do(\xi, S_0))$. This proves the equivalence (19), and the first half of the theorem.

To prove the second half of the theorem, suppose now that M' is a model of $\mathcal{E}$. Construct an interpretation M as follows:

1. M and M' share the domains for all sorts.
2. The truth values of fluents are defined as follows: Initially, all fluents are false:

   $$M \models (\forall \vec{x}) \neg F(\vec{x}, S_0)$$

   for every fluent F. Inductively, for any $\hat{s}$ in the situation domain, any action $\hat{a}$ in the action domain, and any tuple $\vec{u}$ of objects, the truth value of $F(\vec{u})$ in $do(\hat{a}, \hat{s})$ is defined by the following two cases:

   Case 1. $\hat{a}$ is not an action that corresponds to one of clauses in the definition of F in P. In this case,

   $$M \models H(F(\vec{u}), do(\hat{a}, \hat{s}))$$

   iff

   $$M \models H(F(\vec{u}), \hat{s}).$$

   Case 2. $\hat{a}$ is an action that corresponds to one of the clauses in the definition of F in P. To simplify our presentation, suppose the instantiation of this clause is

$$F(\vec{u}) \texttt{ :- } G \texttt{ \& not } F_i(\vec{t})\texttt{.}$$

where G does not contain `not`. Then

$$M \models H(F(\vec{u}), do(\hat{a}, \hat{s}))$$

iff

$$M \models H(G, \hat{s}),$$

and

$$M' \models \neg(\exists s').Acc(s') \wedge H(F_i(\vec{t}), s').$$

Again, we need to show that the equivalence (19) holds, and that M is a model of $\mathcal{D}$, i.e. satisfies the successor state axioms in $\mathcal{D}$.

The proof of the equivalence (19) is similar to the one given above:

1. if $M \models (\exists s)H(F(\vec{u}), s)$, then there is a situation $\hat{s}$ such that $M \models \texttt{minimal}(F(\vec{u}), \hat{s})$. Let $\hat{s'}$ be the situation obtained from $\hat{s}$ in the way described by (20. Then from the construction of M and (18), it can be verified that $M' \models Acc(\hat{s'}) \wedge H(F(\vec{u}), \hat{s'})$.

2. if $M' \models (\exists s)Acc(s) \wedge H(F(\vec{u}), s)$, then there is a $\hat{s}$ in the situation domain such that $M' \models Acc(\hat{s}) \wedge \texttt{minimal}(F(\vec{u}), \hat{s})$. Let $\hat{s'}$ be the situation obtained from $\hat{s}$ by eliminating actions that do not correspond to actions in P. Then $M \models H(F(\vec{u}), \hat{s'})$.

From the equivalence (19), we have that for any fluent F,

$$M' \models (\exists s).Acc(s) \wedge H(F'(\vec{u}), s)$$

iff

$$M \models \neg(\exists s)H(F(\vec{u}), s).$$

From this, it is easy to show that M satisfies the successor state axioms in $\mathcal{D}$. ■

ELSEVIER

Artificial Intelligence 101 (1998) 201–226

Artificial Intelligence

What robots can do: robot programs and effective achievability

Fangzhen Lin [a,*], Hector J. Levesque [b,1]

[a] *Department of Computer Science, The Hong Kong University of Science and Technology, Hong Kong*
[b] *Department of Computer Science, University of Toronto, Toronto, Canada M5S 3H5*

Received 14 May 1997; received in revised form 19 March 1998

Abstract

In this paper, we propose a definition of goal achievability: given a basic action theory describing an initial state of the world and some primitive actions available to a robot, including some actions which return binary sensing information, what goals can be achieved by the robot? The main technical result of the paper is a proof that a simple robot programming language is universal, in that any effectively achievable goal can be achieved by getting the robot to execute one of the robot programs. The significance of this result is at least twofold. First, it is in many ways similar to the equivalence theorem between Turing machines and recursive functions, but applied to robots whose actions are specified by an action theory. Secondly, it provides formal justifications for using the simple robot programming language as a foundation for our work on robotics. © 1998 Elsevier Science B.V. All rights reserved.

Keywords: Robotics; Cognitive robotics; Robot programs; Achievability; Effective achievability; Abilities; Theories of actions; Situation calculus

1. Introduction

Imagine that in the not too distant future, you are given a robot of some sort, and that you want to figure out what it can do. Browsing through the manual that came with it, you discover that the robot is capable of performing under computer control any of a set of primitive actions $a_1, \ldots, a_n$. According to the manual, what each action a_i actually does depends on the state of the environment. First, to complete successfully, a precondition

* Corresponding author. Email: flin@cs.ust.hk.
[1] Email: hector@ai.toronto.edu.

0004-3702/98/$19.00 © 1998 Elsevier Science B.V. All rights reserved.
PII: S0004-3702(98)00041-1

of the action must hold in the environment. Next, assuming the action is successful, its effect on the environment may also depend on certain other conditions. Finally, some of the actions are connected to sensors and can return a binary value indicating when a certain condition holds. The question is: assuming we are willing to do some programming, what do we expect to be able to achieve with the robot?

In this paper, we propose an answer to this question. Specifically, we propose an abstract characterization of what goals are *effectively achievable* as a function of a given logical theory describing the initial state of the world and the primitive actions available to the robot. The main contribution of the paper is a precise technical framework where questions of goal achievability can be posed and answered. The main technical result is a proof of the *universality* of the simple robot programming language introduced in [7]: it will turn out that a goal is effectively achievable according to logical theory T iff there is a robot program that achieves it according to T.

1.1. A motivating example

To make the problem more concrete, imagine that you are also given a solid steel box that contains a treasure. There is a small robot-sized door on the box, which is currently closed, and there are two buttons beside it, a green one and a red one. The primitive actions available to the robot are *pressGreen*, *pressRed*, and *fetch*. The manual says that if the robot happens to be beside the closed door, *pressGreen* causes the green button to be pressed, and *pressRed* similarly. The manual also says that the robot has a heat sensor so that a press action returns 1 when the button pressed was hot, and 0 otherwise. The *fetch* action causes the robot to trundle inside the box and retrieve what's inside, provided that the robot is beside the door and the door is open.[2] The goal we are interested in here, obviously, is getting the treasure, under assumptions like the following:

(1) If we know nothing else about the environment, we want our account of achievability to predict that we cannot achieve the goal. Of course, we might end up eventually getting the treasure by forcing the door open with a crowbar, or by saying some magic words, or even by getting the robot to press the buttons in some order. But there is no reason to believe *a priori* that any of these methods will work.

(2) If we know that the red button opens the door of the box, we want our account of achievability to say that we can achieve the goal using the robot: we get it to do the sequence *pressRed*, then *fetch*. Of course, something might go wrong: the door might jam, lightning might strike the robot, a comet might hit the earth. But there is no reason to believe that the sequence will fail given what we have been told.

(3) If we know that one of the buttons opens the door of the box, and the other button locks the door permanently, but we don't know which is which, our account should predict that we cannot achieve the goal using the robot. As in (2), we know that there is a sequence of actions that will work—press one of the buttons then *fetch*—but here we do not know what that sequence is.

[2] In a more realistic setting, of course, there would be a large number of other preconditions for actions like these.

(4) But consider the following situation: we know that the door can be opened by first pressing the green button, and then pressing one more button, but we are not told which, and again, getting it wrong locks the door permanently. However, suppose that we know that the safe will lock forever iff the robot pushes a button that felt hot on the previous press. As in (3), we know that there is a sequence of actions that will work, and again we cannot say what that sequence is. This time, however, our account should predict that we can achieve the goal: we get the robot to *pressGreen*, and then *pressGreen* once more if the button was cold, but *pressRed* if it was hot.

(5) Finally, suppose we know that after pressing the green button some unspecified number of times and at least once, pressing the red button will open the door and pressing the green one will lock it forever. With no other information, we clearly cannot obtain the treasure. However, if we also know as in (4) that the door will lock forever iff the robot presses a button that was just hot, then we can once again achieve the goal: we get the robot to repeatedly press the green button until it feels hot, then press the red one to open the door, and then fetch the treasure.

To the best of our knowledge, there is as yet no formal framework that would give the intuitively correct answers for examples like these.

1.2. Relation to other work

There are, however, three areas of research that come close to providing these answers.

Planning. As the five examples above illustrate, the idea of a goal being achievable by a robot is clearly related to the concept of planning and especially, given the sensing, conditional planning, as in [2,13,19,23]. In all of the variants above, we ended up saying that the treasure was obtainable precisely when we could formulate some sort of plan to obtain it. Why then not simply define goal achievability in terms of the existence of a plan?

The problem with this involves characterizing exactly what we mean by a plan. An obvious case is when a fixed sequence of actions is sufficient. But in some of the variants above, we needed to consider conditional and iterative plans, which suggests a structure more like that of a program [14]. Clearly these would not be programs in a traditional language like C or LISP. For one thing, the primitive statements of the program would have to involve the actions a_i, rather than the usual variable assignment or read/write statements. What would we use as the conditions in an if-then-else or a while-loop statement? How should the execution of programs containing the a_i be defined?

We believe that these questions can be resolved and that it is possible to characterize achievability in terms of such programs (see Section 4 below) . However, to avoid making design decisions that might initially appear to be arbitrary or restrictive, we prefer to first define achievability in a general program-independent way, and then *prove* that a programming language is adequate according to this definition.

Computability. A second concept related to achievability is that of effective computability [16]. As will become clear, we will end up defining achievable goals as those where what to do next to achieve them, given what is known about the actions and the initial state of the world, can be "computed" as a function of what the sensors tell the robot.

However, we cannot simply use an existing account of computability for two reasons. First, we want to allow for incomplete information about the environment surrounding the robot. In contrast to typical accounts of computability, the initial state of the environment need only be partially specified by a collection of axioms. The second reason concerns the primitive actions. In typical computability models, the available actions are predefined and *internal* to the machine (or formalism). For instance, we might have actions to write and read a Turing machine tape, or to increment and decrement registers, or to assign values to variables, and so on. In our case, by contrast, the primitive actions for a robot are not predefined and are *external*, in that they have effects in the environment outside of the robot. These actions are also described by a collection of axioms, which specify the action preconditions and effects, and deal with the frame problem.

Thus our account of goal achievability depends crucially on what the given axioms say about the initial state and the available actions. In some of the examples above, we had two theories $T_1 \subseteq T_2$ describing the same initial state and set of actions. A goal was considered unachievable relative to the information provided by T_1, but achievable relative to T_2 where additional information was available. We would like to define a notion of goal achievability as a *relation* between a formal theory T and the goals we would like to achieve, and no existing account of computability does this.

Knowing how. Finally, the concept of achievability is very closely related to the concept of an agent knowing how (or being able to) achieve a goal or execute a plan, as discussed for example, in [1,24,25]. One difference between the two concepts concerns the issue of effectiveness. As far as we know, no existing account of knowing how or ability considers whether or not the know how of an agent would be effective, in the sense of allowing the agent to "compute" what to do. But putting effectiveness aside, there is also a difference in point of view: who has to know what and when. There may be conditions that we would consider to be achievable, but that the agent does not know how to bring about. For example, if the red button opens the door, *we* know that goal of getting the treasure is achievable by the agent/robot; but if the agent does not know which button is the correct one, we would not say that *it* knew how to get the treasure. Conversely, we can imagine a situation where we do not consider the goal to be achievable (in the sense of being able to produce a plan) because we do not know which buttons to use, but where we know that the agent does. We can also imagine situations where the agent initially knows less than we do, but after obtaining information from its sensors, knows as much or more than we do.

When reasoning about what one agent knows about another, the concept of knowing-how or ability may be the more useful one; when attempting to analyze what we can get an agent or robot to do for us, our notion of goal achievability may be the more appropriate. Moreover, it ought to be the case that the two notions coincide when the agent knows exactly what we do about the environment and the actions. The precise relation between the two concepts is subtle, however, and we will not explore it further here (see [5]).

In sum, while the concept of goal achievability is clearly related to the areas of planning, computability, and agent ability, none of these can give us the answers we want, for example, in the five situations above.

The rest of the paper is organized as follows. In the next section, we review the situation calculus, a formal logical language in terms of which the state of the environment and the

primitive actions can be described by a collection of axioms we call a basic action theory. In Section 3, we define precisely what we mean by effective achievability (and related notions) as a function of a given basic action theory. In Section 4, we review the syntax and semantics of a simple robot programming language first proposed in [7] as a language for plans. In Section 5, we present some results, including the main technical result of the paper: the universality of the robot programming language. This is a robot analogue of the classic universality result in computability theory: a function is computable iff there is a program/machine that computes it. Finally, in Section 6, we summarize the paper and suggest topics for further research.

2. The situation calculus and basic action theories

Since the goal of this research is to make the specification of goal achievability depend on a given action theory T describing the initial state of the world and the available actions, we need to describe the representation language we use to formulate the theories, which is a dialect of the situation calculus [15].

The language of the situation calculus is many-sorted. Normally, there is a sort for situations, a sort for actions, and a sort for objects like blocks and people that are elements in the domain of interest. We assume that there is a special constant S_0 used to denote the *initial situation*, namely that situation in which no actions have yet occurred; there is a distinguished binary function symbol *do* where $do(a, s)$ denotes the successor situation to s resulting from performing the action a; relations whose truth values vary from situation to situation, are called (relational) *fluents*, and are denoted by predicate symbols taking a situation term as their last argument; there is a special predicate $Poss(a, s)$ used to state that action a is executable in situation s; and finally, there is a special predicate $SF(a, s)$ used to state that the sensor associated with action a (if any) returns the value 1 in situation s. [3]

Within this language, we can formulate domain theories which describe the initial state of the world and the actions available to the robot. We specify the preconditions of actions, for example, by writing axioms that define *Poss*; we specify the condition measured by a sensor by writing axioms that define *SF*, and so on. Here, we use a theory which contains only the following axioms:

- Axioms describing the initial situation, S_0. Syntactically, these axioms cannot mention any other situation terms except S_0.
- Action precondition axioms, one for each primitive action A, characterizing $Poss(A, s)$. Syntactically, these axioms all have the following form:

$$Poss(A, s) \equiv \Psi_A(s), \tag{1}$$

where $\Psi_A(s)$ is a formula that does not mention any other situation terms except s, does not quantify over situation variables, and does not mention the special predicates like *Poss*, *SF*, or $<$ (introduced below).

[3] In [7], the predicate *SF* was used to characterize what an agent knew in a situation in terms of a fluent K. In this paper, we will not be concerned with the knowledge of agents.

- Sensed fluent axioms, one for each primitive action A, characterizing $SF(A, s)$. Syntactically, these axioms have the form:

$$SF(A, s) \equiv \Pi_A(s),$$

where $\Pi_A(s)$ satisfies the same conditions as those for $\Psi_A(s)$ above. For actions that have nothing to do with sensors, this should be $[SF(A, s) \equiv True]$.

- Successor state axioms, one for each fluent F, characterizing under what conditions $F(\vec{x}, do(a, s))$ holds as function of what holds in situation s. These take the place of the so-called effect axioms, but also provide a solution to the frame problem [20]. Syntactically, successor state axioms have the form:

$$Poss(a, s) \supset \left[F\left(\vec{x}, do(a, s)\right) \equiv \Phi_F\left(\vec{x}, a, s\right)\right], \tag{2}$$

where Φ_F satisfies the same conditions as those for $\Psi_A(s)$ above.

- Unique names axioms for the primitive actions: For any two different actions $A(\vec{x})$ and $A'(\vec{y})$, we have

$$A\left(\vec{x}\right) \neq A'\left(\vec{y}\right),$$

and for any action $A(x_1, \ldots, x_n)$, we have

$$A(x_1, \ldots, x_n) = A(y_1, \ldots, y_n) \supset x_1 = y_1 \wedge \cdots \wedge x_n = y_n.$$

- Foundational, domain-independent axioms that characterize the structure of the space of situations, and define a predicate $\leqslant$ so that $s_1 \leqslant s_2$ holds iff s_2 can be reached from s_1 by a sequence of executable actions, i.e., there are actions $a_1, \ldots, a_n$, $0 \leqslant n$, such that

$$s_2 = do\left([a_1, \ldots, a_n], s_1\right) \wedge$$
$$Poss(a_1, s_1) \wedge \cdots \wedge Poss\left(a_n, do\left([a_1, \ldots, a_{n-1}], s_1\right)\right)$$

holds, where for any situation s, $do([], s) = s$, and inductively, $do([a|L], s) = do(L, do(a, s))$. These axioms are:

$$S_0 \neq do(a, s),$$
$$do(a_1, s_1) = do(a_2, s_2) \supset (a_1 = a_2 \wedge s_1 = s_2),$$
$$(\forall P).P(S_0) \wedge (\forall a, s)[P(s) \supset P(do(a, s))] \supset (\forall s)P(s),$$
$$\neg s < S_0,$$
$$s < do(a, s') \equiv (Poss(a, s') \wedge s \leqslant s').$$

Notice the similarity between these axioms and Peano Arithmetic. The first two axioms are unique names assumptions; they eliminate finite cycles, and merging. The third axiom is second-order induction; it amounts to a domain closure axiom which says that every situation must be obtained by repeatedly applying do to S_0.[4] The last two axioms define $<$ inductively.

[4] For a discussion of the use of induction in the situation calculus, see (Reiter [21]).

Following [11], we call a theory of this form a *basic action theory*. In this paper we shall consider the question of effective achievability with respect to a basic action theory. Before turning to this task, we first make some remarks about the generality of our basic action theories and their relationships with more popularly used formalisms such as STRIPS [3] and ADL [17] used in AI planning.

If we do not consider sensing actions, then *as far as the action effects are concerned*, basic action theories have more or less the same expressive power as Pednault's ADL action description language, which is more expressive than STRIPS and used by UCPOP [18]. The main reason is that if there are no sensing actions, as far as the effects of actions are concerned, a basic action theory consists of a set of action precondition and successor state axioms of the forms (1) and (2), respectively. Under some reasonable conditions, these axioms can be reformulated as Pednault's ADL descriptions and vise versa following a procedure discussed in detail by Reiter [20] and Pednault [17]. In any event, for our purposes here, an action theory that does not have any sensing actions is not that interesting: if $(\forall a).SF(a, s) \equiv \mathit{True}$, then a goal $G(s)$ is achievable in a situation S iff there is an executable sequence Γ of ground actions such that $G(do(\Gamma, S))$ is entailed by the action theory. In this case, induction provides a powerful technique for proving what can and cannot be achieved, as shown by Reiter [21].

To the best of our knowledge, UWL [2] is the only STRIPS-like action description language that axiomatizes the effects of a sensing action on the agent's knowledge state. On the one hand, our action theories are more general than UWL domain descriptions in that we allow arbitrary formula to be sensed while UWL only allows a conjunction of literals. On the other hand, UWL is more general in that it allows a precondition of an operator to be a knowledge goal such as `(find-out (P . v))` (find out the truth value of P) or a maintenance goal such as `(hands-off P)` (do not change the truth value of P). These goals, although can be handled in the situation calculus,[5] are beyond the scope of this paper, and are interesting future research topics. Ignoring these differences, our account of effective achievability can then be used, for example, to check whether the planning algorithm for UWL given in [2] is sound and/or complete.

For further details on the generality of our approach, see also [7], where a number of examples are given of goals that are intuitively achievable/unachievable in the presence/absence of sensing. This paper contains examples of goals that can only be achieved by plans containing loops as well as sensing, and as far as we know, it is the only account that has this generality. It was this paper that inspired us to look for a definition of goal achievability that did not appeal to a predefined programming language of plans.

3. Effective achievability

To define in its most general form what a robot armed with primitive actions $a_1, \ldots a_n$. can achieve, it is useful to begin by looking at the problem from the point of view of a robot controller, for instance, an onboard computer.

[5] See [5,22] for a treatment of knowledge goals, and [8–10] for some example formalization of temporal constraints in the situation calculus.

What a robot controller needs to do at any given point in time is to select the primitive action to perform next (or to stop). We are willing to assume arbitrary amounts of computation and intelligence in making this decision, as well as full access to the given basic action theory. We do not want to assume, however, that the controller necessarily knows everything there is to know about the current state of the environment. For example, if it is part of the basic action theory that a door is open initially, the controller can use this fact; but if the action theory does not specify the state of the door, the robot may need to perform some (sensing) action to find out whether it is open, assuming such an action is available.

So what does a robot controller have access to beyond the given basic action theory? In its most general form, we might imagine that the robot controller remembers all of the actions it has selected until now, as well as the sensing results of all these actions. In general, these sensing results must be compatible with the given action theory, but will not be entailed by it, and so provide additional information to the controller.

Once we have specified what a robot controller is, we can then define the achievable goals. Roughly, a goal will be considered to be achievable if there exists a robot controller such that if we were to repeatedly do the primitive action it prescribes, then no matter how the sensing turns out, we would eventually terminate in a situation where, according to the action theory, the goal condition would hold. We now proceed to formally define the relevant notions.

3.1. Robot controllers and environments

We assume a finite set $\mathcal{A}$ of actions that are parameterless, and represented by constant symbols. At any point, the robot will be in some state determined by the actions it has performed so far and, in the event of sensing, the readings of its sensors. More precisely, we define:

Definition 1 (History). A *history* σ is an element of the set $\mathcal{R} = (\mathcal{A} \times \{0, 1\})^*$.

Intuitively, the history $(\alpha_1, \beta_1) \circ \cdots \circ (\alpha_n, \beta_n)$ means that $\alpha_1, \ldots, \alpha_n$ is the sequence of actions performed so far, and $\beta_1, \ldots, \beta_n$ are the respective sensing results of the actions: For any i, if the sensing fluent SF holds for α_i in the situation where the action is performed, then $\beta_i = 1$, else $\beta_i = 0$. Notice that by the form of basic action theories (cf. Section 2), if α_i is an action that has nothing to do with sensors, then $\beta_i = 1$. Notice also that the empty sequence ε is a history.

A robot controller is then a mapping from such a history to the next action to perform. In addition to the given primitive actions in $\mathcal{A}$, we assume some special symbols. In the following, let $\mathcal{A}^+ = \mathcal{A} \cup \{stop, abort, \perp\}$, where *stop*, *abort*, and $\perp$ are special symbols not in $\mathcal{A}$. Intuitively, *stop* will be used to denote termination, *abort* to signal exit before the end of the computation, and $\perp$ to denote an undefined computation.[6] Formally, we define:

[6] The reason we need *abort* and $\perp$ will be made clear later in the context of robot programs.

Definition 2 (Robot controller). A (*robot*) *controller* $\mathcal{C}$ is any function from histories to actions or special symbols, $\mathcal{C}: \mathcal{R} \Rightarrow \mathcal{A}^+$.

Definition 3 (Effective controller). A controller is *effective* if the function $\mathcal{C}$ is recursive.

It should be clear that the only feedback the robot gets from the environment is through its sensors. Just as a robot controller specifies the next action to perform, an environment specifies the sensing result of that action. More precisely, we define:

Definition 4 (Environment). An *environment* $\mathcal{E}$ is any function from histories and actions to the set $\{0, 1\}$, $\mathcal{E}: \mathcal{R} \times \mathcal{A} \Rightarrow \{0, 1\}$.

In other words, $\mathcal{E}(\sigma, \alpha)$ tells us what the sensor associated with action α will report given the history σ.

Intuitively, the picture is this. We start with the empty history ε; the robot controller $\mathcal{C}$ chooses an action to perform $\alpha_1 = \mathcal{C}(\varepsilon)$; the environment $\mathcal{E}$ determines the value returned by the α_1 sensor: $\beta_1 = \mathcal{E}(\varepsilon, \alpha_1)$; given this result, the robot then chooses another action to perform $\alpha_2 = \mathcal{C}((\alpha_1, \beta_1))$, and the environment determines the α_2 sensor value: $\beta_2 = \mathcal{E}((\alpha_1, \beta_1), \alpha_2)$; then $\alpha_3 = \mathcal{C}((\alpha_1, \beta_1) \circ (\alpha_2, \beta_2))$ and so on, until $\mathcal{C}$ says *stop*.

Definition 5 (System). A *system* is a pair $(\mathcal{C}, \mathcal{E})$, where $\mathcal{C}$ is a controller, and $\mathcal{E}$ an environment.

Frequently, we shall refer to the system $(\mathcal{C}, \mathcal{E})$ as the controller $\mathcal{C}$ under the environment $\mathcal{E}$.

Definition 6 ((Terminating) run). A history σ is a *run of a system* $(\mathcal{C}, \mathcal{E})$ if, inductively, either σ is the empty sequence ε or $\sigma = \sigma' \circ (\alpha, \beta)$ such that σ' is a run of the system, $\mathcal{C}(\sigma') = \alpha \in \mathcal{A}$, and $\mathcal{E}(\sigma', \alpha) = \beta$. A history σ is a *terminating run* of $(\mathcal{C}, \mathcal{E})$ if it is a run of $(\mathcal{C}, \mathcal{E})$ and $\mathcal{C}(\sigma) = stop$.

Clearly, a system can have at most one terminating run.

3.2. Achievability and effective achievability

Note that neither controllers nor environments are part of the situation calculus; they are simply abstract functions over the domain of histories. To make a connection with the situation calculus, we first relate histories to situations:

Definition 7 (Run and situation). Given any history σ, and any situation term s, we define another situation term, the *end situation* of σ on s, written $end(\sigma, s)$, as: $end(\varepsilon, s) = s$; and inductively, if $\sigma = \sigma' \circ (\alpha, \beta)$, then $end(\sigma, s) = do(\alpha, end(\sigma', s))$.

Next, we relate environments to logical interpretations of a basic action theory.

Definition 8 (Environment and interpretation). Given an interpretation I and a ground situation term S, an environment $\mathcal{E}$ is said to be *determined by I at S* iff for any history σ, and any action α, $\mathcal{E}(\sigma, \alpha) = 1$ iff $I \models SF(\alpha, end(\sigma, S))$.

It is clear that there is exactly one such environment for any given I and S. In other words, once we specify an interpretation of a basic action theory (and a starting situation), the interpretation of the SF predicate completely determines how the sensing will turn out, and hence the environment.

In general, we expect a basic action theory to be satisfied by many interpretations, corresponding to the various ways the environment could turn out. Goal achievability requires a controller to work in all such interpretations:

Definition 9 ((Effective) achievability). Given an action theory T, a goal $G(s)$ which is a formula with a single free situation variable s, and a ground situation term S, we say that G is *(effectively) achievable* in S according to T iff there is an (effective) controller $\mathcal{C}$ such that for any model I of T, there is a terminating run σ of $\mathcal{C}$ under the environment determined by I at S such that $I \models S \leqslant end(\sigma, S) \wedge G(end(\sigma, S))$.

Notice that the condition $S \leqslant end(\sigma, S)$ means that $end(\sigma, S)$ is reachable from S by a sequence of executable actions. This means that the actions prescribed by the run σ must be executable in their respective situations. This condition reflects our intuition that for a goal to be achievable, the sequence of actions that is used to achieve it must be at least executable.[7]

In general, there will be goals that are achievable but not effectively achievable. However, as we are going to show below, for *context-free* action theories, if a goal is achievable, then it is also effectively achievable.

3.3. Achievability in context-free action theories

By a context-free action theory we mean a theory in which all actions are context-free in the sense that their effects are independent of the state in which they are executed. For example, in the blocks world, the action $stack(x, y)$, that picks up block x on the table and puts in on top of block y, is context-free—as long as it is executable, it will always cause x to be on y. On the other hand, in the extended blocks world in which there may be more than one block on top of another block, the action $unstack(x, y)$, that removes x from y, is not context-free—whether block y will be clear afterwards, for example, depends on whether x was the only block on top of y.

[7] An interesting question motivated by a comment from an anonymous referee is whether basic action theories are strong enough to make this condition redundant in Definition 9. A simpler question asks whether for any goal G, whenever there is a situation S such that $T \models G(S)$, then there is another situation S' such that $T \models S_0 \leqslant S' \wedge G(S')$. These are presently open questions. However, except possibly making our definition of achievability a little simpler, the answers to them have otherwise no effect on the results of this paper.

Now an action theory is context-free if, according to the theory, all actions are context-free. Formally, following [12], we call an action theory T context-free if every successor state axiom in it has the following form:

$$Poss(a, s) \supset \left[F(\vec{x}, do(a, s)) \equiv \gamma_F^+(\vec{x}, a) \vee \left(F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a)\right)\right], \tag{3}$$

where $\gamma_F^+(\vec{x}, a)$ and $\gamma_F^-(\vec{x}, a)$ are situation-independent formulas whose free variables are among those in $\vec{x}, a$. Under the following *consistency condition* [20]:

$$\Sigma \models (\forall a, \vec{x}).\neg\left(\gamma_F^+(\vec{x}, s) \wedge \gamma_F^-(\vec{x}, s)\right), \tag{4}$$

the axiom (3) implies that for any action a, after the action is performed, F will be true (added) for tuples in $\{\vec{x} \mid \gamma_F^+(\vec{x}, a)\}$, false (deleted) for tuples in $\{\vec{x} \mid \gamma_F^-(\vec{x}, a)\}$, and persist for tuples in $\{\vec{x} \mid \neg\gamma_F^+(\vec{x}, a) \wedge \neg\gamma_F^-(\vec{x}, a)\}$. The action a is context-free because the conditions γ_F^- and γ_F^+ are situation-independent. Note that in the usual formulation of STRIPS, with add and delete lists, every action is considered to be context-free.

Theorem 1. *Let T be a context-free action theory, and the consistency condition* (4) *holds for every fluent F.* [8] *If a goal G is achievable in S, then it is also effectively achievable.*

Proof. See Appendix A. □

Informally, the theorem holds because a context-free action theory can only have finite number of possible legal states. One can read the theorem in two ways. On the one hand, it points to some potential computational advantages of working with context-free action theories. On the other hand, it also points out their expressive limitations. For example, this theorem implies that it is impossible to simulate an arbitrary context-sensitive action with a *finite* set of context-free actions.

4. A robot program language

In [7], the following question was considered: what should the output of a planning procedure be? In the absence of sensing, the answer is reasonably clear and dates back to Green [4]: a plan is a legally executable sequence of actions that results in a final situation where the goal condition holds. In the presence of sensing, however, a planner cannot simply return a sequence of actions since the actions to execute to satisfy the goal could depend on the runtime result of earlier sensing operations (as in the examples in the introduction).

Clearly, what is needed is something more like a program, with branches and loops. On the other hand, it would need to be a program that is not only legally executable (in the sense that the preconditions of the primitive actions at each step are satisfied), and leads to a goal state (in the sense that the program terminates and the goal condition holds in the terminating situation), but also a program that does not require more information to execute

[8] Technically, this condition is not necessary. However, it simplifies our proof, and is a reasonable condition to impose on action theories.

than what we expect the robot to have. For example, if all we know is that the door to the steel box opens by pushing either the red or the green button, then the program which says something like "if the red button is the one that opens the door then push it, else push the other one" might satisfy the first two conditions, but not the last one.

There are various ways to ensure this last requirement. The approach taken in [7] is to invent a simple language that contains branches and loops, but that does not mention any conditions involving fluents. The resulting programs are then trivial to execute since without such conditions, there is nothing for the robot executing the programs to know.

Consider the following simple programming language, defined as the least set of terms satisfying the following:

(1) *nil* and *exit* are programs.

(2) If a is an action and r_1 and r_2 are programs, then $branch(a, r_1, r_2)$ is a program.

(3) If r_1 and r_2 are programs, then $loop(r_1, r_2)$ is a program.

We will call such terms *robot programs* and the resulting set of terms $\mathcal{R}$, the robot programming language.

Informally, these programs are executed by an agent as follows: to execute *nil* the agent does nothing; to execute *exit* it must be executing a *loop*, in which case see below; to execute $branch(a, r_1, r_2)$ it first executes primitive action a, and then it executes r_1 if the sensor associated with a returns 1, and r_2 otherwise; to execute $loop(r_1, r_2)$, it executes the body r_1, and if it ends with *nil*, it repeats r_1 again, and continues doing so until it ends with *exit*, in which case it finishes by executing r_2.

Note that many actions will not have an associated sensor and will always return 1. We thus use the abbreviation $seq(a, r)$ for $branch(a, r, r)$ for those cases where the returned value is ignored.

Here are some robot programs for the examples in the introduction. (Recall that the *pressGreen* action returns 1 if the button is hot.)

(1) Sequence of actions:

$$seq(pressRed, seq(fetch, nil)).$$

(2) Conditional plan:

$$branch(pressGreen, seq(pressRed, seq(fetch, nil)), seq(pressGreen, seq(fetch, nil))).$$

(3) Iterative plan:

$$loop(branch(pressGreen, exit, nil), seq(pressRed, seq(fetch, nil))).$$

Intuitively at least, the following should be clear:

- An agent can always be assumed to know how to execute a robot program. These programs are completely deterministic, and do not mention any fluents. Assuming the binary sensing actions return a single bit of information to the agent, there is nothing else it should need to know.

- The example robot programs above, when executed, result in final situations where the goal conditions from the introduction are satisfied.

To be precise about this, we need to first define what situation is the final one resulting from executing a robot program r in an initial situation s. Because a robot program could conceivably loop forever (e.g., $\underline{loop}(\underline{nil}, \underline{nil})$), we will use a formula $Rdo(r, s, s')$ to mean that r terminates legally when started in s, and s' is the final situation. Formally, *Rdo* is an abbreviation for the following second-order formula:

$$Rdo(r, s_1, s_2) \stackrel{\text{def}}{=} \forall P\big[\ \cdots \supset\ P(r, s_1, s_2, 1)\big] \tag{5}$$

where the ellipsis is (the conjunction of the universal closure of) the following:

(1) Termination, normal case: $P(\underline{nil}, s, s, 1)$.

(2) Termination, loop body: $P(\underline{exit}, s, s, 0)$.

(3) Primitive actions returning 1:

$$Poss(a, s) \wedge SF(a, s) \wedge P(r', do(a, s), s', x) \supset P(\underline{branch}(a, r', r''), s, s', x).$$

(4) Primitive actions returning 0:

$$Poss(a, s) \wedge \neg SF(a, s) \wedge P(r'', do(a, s), s', x) \supset P(\underline{branch}(a, r', r''), s, s', x).$$

(5) Loops, exit case:

$$P(r', s, s'', 0) \wedge P(r'', s'', s', x) \supset P(\underline{loop}(r', r''), s, s', x).$$

(6) Loops, repeat case:

$$P(r', s, s'', 1) \wedge P(\underline{loop}(r', r''), s'', s', x) \supset P(\underline{loop}(r', r''), s, s', x).$$

By using second-order quantification in this way, we are defining *Rdo* recursively as the *least* predicate P satisfying the constraints in the ellipsis. Second-order logic is necessary here since there is no way to characterize the transitive closure implicit in unbounded iteration in first-order terms.

The relation $P(r, s, s', 0)$ in this definition is intended to hold when executing r starting in s terminates legally at s' with $\underline{exit}$; $P(r, s, s', 1)$ is the same but terminating with $\underline{nil}$. The difference shows up when executing $\underline{loop}(r, r')$: in the former case, we exit the loop and continue with r'; in the latter, we continue the iteration by repeating $\underline{loop}(r, r')$ once more.

With this definition in place, we can now characterize precisely the goals that are achievable using robot programs:

> Given an action theory T, a robot program r, a goal condition $G(s)$ and a ground situation term S, we say that r *achieves* G in S according to T iff
>
> $$T \models \exists s'.Rdo(r, S, s') \wedge G(s').$$

We now relate this definition to effective achievability.

5. Robot programs are universal

Our main technical result in this paper is that a goal is achievable by an (augmented) robot program iff it is achievable by an effective controller. We shall prove this in two parts. First, we show that for any robot program, there is a corresponding effective controller for it. We then show that if 5 special "Turing machine actions" are included, then any effective controller can be simulated by a robot program.

5.1. From robot programs to effective controllers

Theorem 2. *For any robot program r and any ground situation term S, there is an effective controller C such that for any interpretation I and any ground situation term S':*

(1) *If $I \models Rdo(r, S, S')$, then there is a terminating run σ of the system $(C, \mathcal{E})$ such that $S' = end(\sigma, S)$, where $\mathcal{E}$ is the environment determined by I at S.*

(2) *If there is a terminating run σ of the system $(C, \mathcal{E})$ such that $end(\sigma, S) = S'$, then $I \models S \leqslant S' \supset Rdo(r, S, S')$. Here $\mathcal{E}$ is the environment determined by I at S.*

Proof. See Appendix B. □

5.2. From effective controllers to robot programs

Given an effective controller, there may not always be a robot program that simulates it.[9] The easiest way to remedy this is to add some special Turing machine actions as in [7].

Formally, we assume that in addition to the actions in $\mathcal{A}$, we have five special actions, *left*, *right*, *mark*, *erase*, *read_mark*, and two special fluents *Marked*, *loc*, characterized by the following axioms:

(1) Precondition: the five actions are always possible

$$Poss(left, s) \wedge Poss(right, s) \wedge Poss(mark, s) \wedge Poss(erase, s) \wedge Poss(read_mark, s).$$

(2) Successor state: only *erase* and *mark* change the *Marked* fluent

$$Poss(a, s) \supset [Marked(n, do(a, s)) \equiv a = mark \wedge loc(s) = n \vee Marked(n, s) \wedge \neg[a = erase \wedge loc(s) = n]].$$

(3) Successor state: only *left* and *right* change the *loc* fluent

[9] The proof of this is somewhat laborious. Observe that despite the presence of loops, a robot program has no internal memory and so no way of counting. So consider an action theory that encodes a string problem: decide if a string of `0`s and `1`s has more `0`s than `1`s. A Turing machine can do this, but a finite-state automaton cannot. Similarly, an effective controller can achieve the corresponding goal, but a robot program in general cannot. We omit the details.

$$Poss(a, s) \supset \{loc(do(a, s)) = n \equiv$$
$$a = left \wedge loc(s) = n + 1 \quad \vee$$
$$a = right \wedge loc(s) = n - 1 \quad \vee$$
$$loc(s) = n \wedge a \neq left \wedge a \neq right\}.$$

(4) Sensed fluent: *read_mark* tells the agent whether the current location is marked

$$SF(left, s) \wedge SF(right, s) \wedge SF(erase, s) \wedge SF(mark, s) \wedge$$
$$[SF(read_mark, s) \equiv Marked(loc(s), s)].$$

These axioms ensure that the five special actions provide the robot with what amounts to a Turing machine tape.

In the following, we will be using what we will call a *TM basic action theory*. This is a basic action theory as before, but where $\mathcal{A}$ includes the five special actions, T contains the above axioms, and where the successor state axioms in T for fluents other than *loc* and *Marked* are such they are unaffected by the five actions. More precisely, for any fluent F that is different from *Marked* and *loc*, and when A is any of the five special actions, the theory T entails:

$$Poss(A, s) \supset \big[F\big(do(A, s)\big) \equiv F(s)\big].$$

In the following, for any ground situation term S, we define $clean(S)$ to be the situation term obtained from S by deleting all the special five actions: $clean(S_0) = S_0$, and

$$clean(do(\alpha, S)) = do(\alpha, clean(S))$$

if $\alpha \in \mathcal{A}$ is not a Turing action, and $clean(do(\alpha, S)) = clean(S)$ otherwise.

Theorem 3. *For any TM basic action theory T, any effective controller C, and any ground situation term S, there is a robot program r such that for any model I of T (as above) and any ground situation S', we have:*

(1) *If $I \models Rdo(r, S, S')$, then there is a terminating run σ of the system $(C, \mathcal{E})$ such that $clean(S') = end(\sigma, S)$, where $\mathcal{E}$ is the environment determined by I at S.*

(2) *If there is a terminating run σ of the system $(C, \mathcal{E})$ such that $S' = end(\sigma, S)$, then there is a situation S'' such that $S' = clean(S'')$, and $I \models S \leqslant end(\sigma, S) \supset Rdo(r, S, S'')$, where $\mathcal{E}$ is the environment determined by I at S.*

Proof. See Appendix C. □

5.3. The main theorem

By Theorems 2 and 3, we have the following result:

Theorem 4. *Let T be any TM basic action theory and G be any goal that does not mention the special fluents loc and Marked. Then G is effectively achievable in S according to T iff there is a robot program r such that r achieves G in S according to T.*

Proof. Suppose G is achieved by the effective controller $\mathcal{C}$. We show that the robot program r for $\mathcal{C}$ as in Theorem 3 achieves G. Suppose I is any model of T. Then, by the definition of achievability, there is a terminating run σ of $(\mathcal{C}, \mathcal{E})$ such that

$$I \models S \leqslant end(\sigma, S) \wedge G(end(\sigma, S)),$$

where $\mathcal{E}$ is the environment determined by I and S. According to Theorem 3, there is a situation S'' such that $clean(S'') = end(\sigma, S)$ and

$$I \models S \leqslant end(\sigma, S) \supset Rdo(r, S, S'').$$

Thus $I \models Rdo(r, S, S'')$. By $I \models G(end(\sigma, S))$ and $clean(S'') = end(\sigma, S)$, we have $I \models G(S'')$ (by a property about *clean*). So $I \models (\exists s')Rdo(r, S, s') \wedge G(s')$. Therefore $T \models (\exists s')Rdo(r, S, s') \wedge G(s')$, and so r achieves G.

Conversely, suppose r achieves G in S. We show that the effective controller $\mathcal{C}$ for r as in Theorem 2 achieves G in S, i.e., for any model I of T, there is a terminating run σ of the system $(\mathcal{C}, \mathcal{E})$ such that $I \models S \leqslant end(\sigma, S) \wedge G(end(\sigma, S))$, where $\mathcal{E}$ is the environment determined by I and S. Now suppose I is a model of T, by the assumption that r achieves G, i.e., $T \models (\exists s')Rdo(r, S, s') \wedge G(s')$, there is a ground situation term S' such that $I \models Rdo(r, S, S') \wedge G(S')$. By Theorem 2, there is a terminating run σ of $(\mathcal{C}, \mathcal{E})$ such that $S' = end(\sigma, S)$. So

$$I \models Rdo(r, S, end(\sigma, S)) \wedge G(end(\sigma, S)).$$

But by the definition of *Rdo*, we have $\models (\forall s_1, s_2)Rdo(r, s_1, s_2) \supset s_1 \leqslant s_2$. Thus we have $I \models S \leqslant end(\sigma, S)$. This shows that G is achieved by the controller $\mathcal{C}$. □

6. Conclusion

We have provided a definition of what it might mean for a condition to be achievable by a robot relative to a given action theory which describes the initial state of the world and the primitive actions available to the robot. Our main technical contribution is in showing that this notion of effective achievability coincides with a notion of achievability by a simple class of robot programs independently introduced in [7]. The significance of this result is at least twofold. First, it is in many ways similar to the equivalence theorem between Turing machines and recursive functions, but applied to robots whose actions are specified by an action theory. Secondly, it provides formal justifications for using the simple class of robot programs as a foundation for our work on robotics. For instance, [7] uses this class of robot programs as a basis for robot planning. We are also beginning work on compiling high-level GOLOG programs [6] into this class of robot programs.

There are some limitations with our current model that are worth mentioning here. First of all, we have assumed that there are only a finite number of parameterless actions. We have also assumed that the sensing actions are binary, characterized by the special *SF* predicate. Furthermore, we have assumed that the only feedback from the environment is the result of these sensing actions. In particular, we have not concerned ourselves here with possible action failure or exogenous actions. Some of these assumptions, such as the binary nature of sensing, are easy to relax; others will require more effort.

In concluding, we want to mention that we are working on relating this work to our other work on agent ability and knowing-how [5]. Another direction worth pursuing is investigating the "finite automaton" version of achievability, i.e., the power of robot programs without the special Turing machine actions.

Acknowledgements

This work was done as part of the Cognitive Robotics project at the University of Toronto, and we greatly benefited from discussions with Yves Lespérance, and Ray Reiter. Financial support for the second author was gratefully received from the Natural Sciences and Engineering Research Council of Canada, the Information Technology Research Centre of Ontario, and the Institute for Robotics and Intelligent Systems of Canada. The first author was also supported in part by the Research Grants Council of Hong Kong under Competitive Earmarked Research Grant HKUST6091/97E.

Appendix A. Proof of Theorem 1

Theorem 1. *Let T be a context-free action theory, and the consistency condition* (4) *holds for every fluent F. If a goal G is achievable in S, then it is also effectively achievable.*

We first prove a lemma which says that the set of *legal states* in a context-free action theory is finite. To formulate the lemma, we first introduce a few notations. Let $F_1(\vec{x}_1, s), \ldots, F_n(\vec{x}_n, s)$ be all the fluents in the language. We define $SameState(s, s')$, meaning that s and s' yield the same state, as follows:

$$SameState(s, s') \stackrel{\text{def}}{=} (\forall \vec{x}_1)\big(F_1(\vec{x}_1, s) \equiv F_1(\vec{x}_1, s')\big) \wedge \cdots \wedge (\forall \vec{x}_n)\big(F_n(\vec{x}_n, s) \equiv F_n(\vec{x}_n, s')\big).$$

The following are some simple properties about *SameState*:

Lemma A.1. *For any basic action theory T, we have*:

$$T \models (\forall a, s, s').SameState(s, s') \supset [Poss(a, s) \equiv Poss(a, s')],$$
$$T \models (\forall a, s, s').SameState(s, s') \supset [SF(a, s) \equiv SF(a, s')],$$
$$T \models (\forall a, s, s').Poss(a, s) \wedge SameState(s, s') \supset SameState(do(a, s), do(a, s')).$$

Proof. Trivially from the definition of basic action theories. □

Lemma A.2. *Let T be a context-free action theory with finite number of parameterless actions. Under the consistency condition* (4), *there is a natural number N such that the following set*

$$\{\|do(\xi, S_0)\| \mid \xi \text{ is a list of actions}\}$$

contains less than N elements, where for any list of actions ξ,

$$\|do(\xi, S_0)\| = \{do(\xi', S_0) \mid T \models S_0 \leqslant do(\xi, S_0) \wedge S_0 \leqslant do(\xi', S_0) \supset SameState(do(\xi, S_0), do(\xi', S_0))\}.$$

In other words, the number of possibly different legal states is bounded by N.

Proof. To simplify our proof, without loss of generality, we assume that the action theory has only two actions A and B. Consider an arbitrary fluent $F(\vec{x}, s)$. Suppose its successor state axiom is as (3). By the consistency condition (4), we have

$$T \models F(\vec{x}, do([], S_0)) \equiv F(\vec{x}, S_0),$$

$$T \models S_0 \leqslant do([A], S_0) \supset F(\vec{x}, do([A], S_0)) \equiv \begin{cases} True & \text{if } \gamma_F^+(\vec{x}, A), \\ False & \text{if } \gamma_F^-(\vec{x}, A), \\ F(\vec{x}, S_0) & \text{otherwise,} \end{cases}$$

$$T \models S_0 \leqslant do([B], S_0) \supset F(\vec{x}, do([B], S_0)) \equiv \begin{cases} True & \text{if } \gamma_F^+(\vec{x}, B), \\ False & \text{if } \gamma_F^-(\vec{x}, B), \\ F(\vec{x}, S_0) & \text{otherwise,} \end{cases}$$

$$T \models S_0 \leqslant do([A, B], S_0) \supset$$

$$F(\vec{x}, do([A, B], S_0)) \equiv \begin{cases} True & \text{if } \gamma_F^+(\vec{x}, A) \wedge \neg\gamma_F^-(\vec{x}, B), \\ True & \text{if } \gamma_F^+(\vec{x}, B), \\ False & \text{if } \gamma_F^-(\vec{x}, A) \wedge \neg\gamma_F^+(\vec{x}, B), \\ False & \text{if } \gamma_F^-(\vec{x}, B), \\ F(\vec{x}, S_0) & \text{otherwise,} \end{cases}$$

$$T \models S_0 \leqslant do([B, A], S_0) \supset$$

$$F(\vec{x}, do([B, A], S_0)) \equiv \begin{cases} True & \text{if } \gamma_F^+(\vec{x}, B) \wedge \neg\gamma_F^-(\vec{x}, A), \\ True & \text{if } \gamma_F^+(\vec{x}, A), \\ False & \text{if } \gamma_F^-(\vec{x}, B) \wedge \neg\gamma_F^+(\vec{x}, A), \\ False & \text{if } \gamma_F^-(\vec{x}, A), \\ F(\vec{x}, S_0) & \text{otherwise.} \end{cases}$$

Furthermore

$$T \models S_0 \leqslant do([A, A], S_0) \supset F(\vec{x}, do([A, A], S_0)) \equiv F(\vec{x}, do([A], S_0)),$$
$$T \models S_0 \leqslant do([B, B], S_0) \supset F(\vec{x}, do([B, B], S_0)) \equiv F(\vec{x}, do([B], S_0)),$$
$$T \models S_0 \leqslant do([A, B, A], S_0) \wedge S_0 \leqslant do([B, A], S_0) \supset$$

$$F(\vec{x}, do([A, B, A], S_0)) \equiv F(\vec{x}, do([B, A], S_0)),$$
$$T \models S_0 \leqslant do([A, B, B], S_0) \supset F(\vec{x}, do([A, B, B], S_0)) \equiv F(\vec{x}, do([A, B], S_0)),$$
$$T \models S_0 \leqslant do([B, A, A], S_0) \supset F(\vec{x}, do([B, A, A], S_0)) \equiv F(\vec{x}, do([B, A], S_0)),$$
$$T \models S_0 \leqslant do([B, A, B], S_0) \wedge S_0 \leqslant do([A, B], S_0) \supset$$
$$F(\vec{x}, do([B, A, B], S_0)) \equiv F(\vec{x}, do([A, B], S_0)).$$

Since the fluent F is arbitrary, the finiteness of the set in question follows. □

Proof of Theorem 1. Suppose G is achievable in S according to T. We need to show that there is a recursive controller that achieves G in S according to T. In fact, we can do better. We'll show that there is a *finite* controller that achieves G. Let C be a robot controller that achieves G according to T, and let

$$\mathcal{S} = \{\sigma \mid \sigma \text{ is a terminating run of } C \text{ under the environment determined by a model of } T \text{ at } S\}.$$

Then the following controller

$$C'(\sigma) = \begin{cases} \perp & \text{if } \sigma \text{ is not a prefix of any history in } \mathcal{S}, \\ stop & \text{if } \sigma \in \mathcal{S}, \\ \alpha & \text{if for some } \beta, \sigma \circ (\alpha, \beta) \text{ is a prefix of a history in } \mathcal{S}, \end{cases} \tag{A.1}$$

is clearly well (uniquely) defined, and achieves G in S as well. Furthermore, if $\mathcal{S}$ is finite, then C' is finite, thus recursive. We now show that $\mathcal{S}$ is indeed finite.

Observe, however, that even though there are only finitely many states, we cannot bound the length of a run by removing "loops" starting and ending in the same state (and guarantee the finiteness of $\mathcal{S}$ this way). This is because a controller may be using pure sensing actions which do not change the state to obtain information. So we need a slightly more complex approach.

Given a set of histories $\mathcal{H}$, and a history $\sigma \in \mathcal{H}$, a segment τ in σ: $\sigma = \sigma_1 \circ \tau \circ \sigma_2$ for some σ_1 and σ_2, is said to be *determinate* if for any σ' in $\mathcal{H}$, whenever σ_1 is a proper prefix of σ', then $\sigma_1 \circ \tau$ is a prefix of σ'. In other words, the underlying controller determined by $\mathcal{H}$ according to (A.1) with $\mathcal{S}$ replaced by $\mathcal{H}$, if any, does not need to consider the alternative outcome of the actions in τ. Notice that the empty sequence is trivially a determinate segment of any history σ with respect to any σ_1 and σ_2 above.

Given any run $\sigma \in \mathcal{S}$, we can decompose it into

$$\sigma = \sigma_1 \circ (\alpha_1, \beta_1) \circ \cdots \circ \sigma_k \circ (\alpha_k, \beta_k) \circ \sigma_{k+1} \tag{A.2}$$

such that

(1) $\sigma_1, \ldots, \sigma_{k+1}$ are determinate segments of σ in $\mathcal{S}$.

(2) $\sigma_1 \circ (\alpha_1, \beta_1), \ldots, \sigma_k \circ (\alpha_k, \beta_k)$ are not determinate segments of σ in $\mathcal{S}$.

Clearly, this decomposition is unique. Furthermore, it has the following properties:

(I) For any $\sigma' \in \mathcal{S}$, if

$$\sigma' = \sigma'_1 \circ (\alpha'_1, \beta'_1) \circ \cdots \circ \sigma'_m \circ (\alpha'_m, \beta'_m) \circ \sigma'_{m+1}$$

is a similar decomposition of σ', and

$$(\alpha_1, \beta_1) \circ \cdots \circ (\alpha_k, \beta_k) = (\alpha'_1, \beta'_1) \circ \cdots \circ (\alpha'_m, \beta'_m),$$

then $\sigma = \sigma'$. Suppose the above equation holds, then $m = k$, $\alpha_i = \alpha'_i$, and $\beta_i = \beta'_i$, $1 \leqslant i \leqslant k$. We show by induction on $0 \leqslant i \leqslant k$ that

$$\sigma_1 \circ (\alpha_1, \beta_1) \circ \cdots \circ \sigma_i \circ (\alpha_i, \beta_i) = \sigma'_1 \circ (\alpha'_1, \beta'_1) \circ \cdots \circ \sigma'_i \circ (\alpha'_i, \beta'_i).$$

The base case is trivial because both of them are empty sequence. Suppose

$$\sigma_1 \circ (\alpha_1, \beta_1) \circ \cdots \circ \sigma_i \circ (\alpha_i, \beta_i) = \sigma'_1 \circ (\alpha'_1, \beta'_1) \circ \cdots \circ \sigma'_i \circ (\alpha'_i, \beta'_i),$$

we show that

$$\begin{aligned}&\sigma_1 \circ (\alpha_1, \beta_1) \circ \cdots \circ \sigma_{i+1} \circ (\alpha_{i+1}, \beta_{i+1}) \\ &\quad = \sigma'_1 \circ (\alpha'_1, \beta'_1) \circ \cdots \circ \sigma'_{i+1} \circ (\alpha'_{i+1}, \beta'_{i+1}),\end{aligned}$$

that is, $\sigma_{i+1} = \sigma'_{i+1}$. Because σ_{i+1} is a determinate segment, by definition, we have that $\sigma_1 \circ (\alpha_1, \beta_1) \circ \cdots \circ \sigma_{i+1}$ must be a prefix of σ'. Similarly, $\sigma'_1 \circ (\alpha'_1, \beta'_1) \circ \cdots \circ \sigma'_{i+1}$ is a prefix of σ. So either $\sigma_{i+1} = \sigma'_{i+1}$ or one of them is a proper prefix of the other. The latter is impossible because otherwise, say σ_{i+1} is a proper prefix of σ'_{i+1}, then σ'_{i+1} cannot be a determinate segment of σ, which violates our assumptions. This proves that

$$\sigma_1 \circ (\alpha_1, \beta_1) \circ \cdots \circ \sigma_k \circ (\alpha_k, \beta_k) = \sigma'_1 \circ (\alpha'_1, \beta'_1) \circ \cdots \circ \sigma'_k \circ (\alpha'_k, \beta'_k).$$

Finally, $\sigma = \sigma'$. For otherwise, one of them would be a proper prefix of the other, which is impossible because they are both runs of $\mathcal{C}$ under some environment.

(II) For any $1 \leqslant i < j \leqslant k$, if $\alpha_i = \alpha_j$, then

$$\begin{aligned}&T \not\models S \leqslant \mathit{end}(\xi_{j-1} \circ \sigma_j, S) \supset \\ &\quad \mathit{SameState}(\mathit{end}(\xi_{i-1} \circ \sigma_i, S), \mathit{end}(\xi_{j-1} \circ \sigma_j, S)),\end{aligned}$$

where for any $0 \leqslant m \leqslant k$, ξ_m is the history:

$$\sigma_1 \circ (\alpha_1, \beta_1) \circ \cdots \circ \sigma_m \circ (\alpha_m, \beta_m).$$

For otherwise, suppose

$$\begin{aligned}&T \models S \leqslant \mathit{end}(\xi_{j-1} \circ \sigma_j, S) \supset \\ &\quad \mathit{SameState}\big(\mathit{end}(\xi_{i-1} \circ \sigma_i, S), \mathit{end}(\xi_{j-1} \circ \sigma_j, S)\big).\end{aligned} \tag{A.3}$$

We claim that $\sigma_j \circ (\alpha_j, \beta_j)$ would have to be a determinate segment of σ, a contradiction with the assumptions that we made about Eq. (A.2). To show that $\sigma_j \circ (\alpha_j, \beta_j)$ is a determinate segment of σ, suppose $\sigma' \in \mathcal{S}$, and ξ_{j-1} is a proper prefix of σ'. Then $\xi_{j-1} \circ \sigma_j$ must be a prefix of σ' for σ_j is a determinate segment of σ. Because both σ' and σ are terminating runs of the controller $\mathcal{C}$, for some β, $\xi_{j-1} \circ \sigma_j \circ (\alpha_j, \beta)$ must be a prefix of σ'. Now let I be the model of T under which σ' is the terminating run of the controller $\mathcal{C}$. Since $\sigma' \in \mathcal{S}$, by Definition 9, $I \models S \leqslant \mathit{end}(\sigma', S)$. Thus by (A.3),

$$I \models \mathit{SameState}\big(\mathit{end}(\xi_{i-1} \circ \sigma_i, S), \mathit{end}(\xi_{j-1} \circ \sigma_j, S)\big).$$

Since $\alpha_i = \alpha_j$, by Lemma A.1 $\beta = \beta_i$. Similarly, $\beta_j = \beta_i$. So $\beta = \beta_j$. Thus ξ_j is a prefix of σ'. This shows that $\sigma_j \circ (\alpha_j, \beta_j)$ is a determinate segment of σ.

Thus by Lemma A.2 and the above property (II), for any run σ in $\mathcal{S}$, if it is decomposed as (A.2), then $k \leqslant M \times N$, where M is the number of actions, and N is the upper bound of the number of possible equivalent classes given in Lemma A.2. By property (I), this implies that $\mathcal{S}$ must be finite.

Appendix B. Proof of Theorem 2

We shall prove a more general result than Theorem 2. To that end, we introduce a relation $Rexit(r, s, s')$ meaning that when started in s, the program r will exit (*abort*) in s'. It is defined in the way similar to *Rdo* in Section 4, as an abbreviation of the following second-order formula:

$$Rexit(r, s, s') \stackrel{\text{def}}{=} (\forall P)[\ \cdots \supset P(s, s', 0)].$$

where the ellipsis is exactly the same as in (5), i.e., the conjunction of six conditions given in Section 4. From this definition, it is easy to verify that the following are consequences of any basic action theory:

$$\neg Rexit(\underline{nil}, s, s'),$$
$$Rexit(\underline{exit}, s, s),$$
$$Rexit(\underline{branch}(a, r_1, r_2), s, s')$$
$$\equiv Poss(a, s) \wedge [SF(a, s) \supset Rexit(r_1, do(a, s), s')] \wedge$$
$$[\neg SF(a, s) \supset Rexit(r_2, do(a, s), s')].$$

The purpose of introducing *Rexit* is for the following lemma:

Lemma B.1. *Let T be an action theory, and I a model of T. For any ground situation terms S and S', $I \models Rdo(\underline{loop}(r_1, r_2), S, S')$ iff there are some ground situation terms $S_1, \ldots, S_n$, $n \geqslant 3$, such that*

(1) *$S = S_1$ and $S' = S_n$.*
(2) *For any $1 \leqslant i \leqslant n-3$, $I \models Rdo(r_1, S_i, S_{i+1})$.*
(3) *$I \models Rexit(r_1, S_{n-2}, S_{n-1})$, and $I \models Rdo(r_2, S_{n-1}, S_n)$.*

Similarly, $I \models Rexit(\underline{loop}(r_1, r_2), S, S')$ iff there are some $S_1, \ldots, S_n$, $n \geqslant 3$, such that

(1) *$S = S_1$ and $S' = S_n$.*
(2) *For any $1 \leqslant i \leqslant n-3$, $I \models Rdo(r_1, S_i, S_{i+1})$.*
(3) *$I \models Rexit(r_1, S_{n-2}, S_{n-1})$, and $I \models Rexit(r_2, S_{n-1}, S_n)$.*

Proof. By induction on the structure of r_1. □

In the following, a history σ is called an *exiting run* of a system $(\mathcal{C}, \mathcal{E})$ iff it is a run of the system, and $\mathcal{C}(\sigma) = abort$. The following theorem includes Theorem 2 as a special case.

Theorem B.2. *For any robot program r and any ground situation term S, there is an effective robot controller C such that for any interpretation I and any ground situation term S':*

(1) (a) *If $I \models Rdo(r, S, S')$, then there is a terminating run σ of $(C, \mathcal{E})$ such that $S' = end(\sigma, S)$;*

(b) *if $I \models Rexit(r, S, S')$, then there is an exiting run σ of $(C, \mathcal{E})$ such that $S' = end(\sigma, S)$, where $\mathcal{E}$ is the environment determined by I at S.*

(2) (a) *If there is a terminating run σ of $(C, \mathcal{E})$ such that $end(\sigma, S) = S'$, then $I \models S \leqslant S' \supset Rdo(r, S, S')$;*

(b) *if there is an exiting run σ of $(C, \mathcal{E})$ such that $end(\sigma, S) = S'$, then $I \models S \leqslant S' \supset Rexit(r, S, S')$, where $\mathcal{E}$ is the environment determined by I at S.*

Proof. We shall construct a recursive function $\Gamma : \mathcal{P} \times \mathcal{R} \to \mathcal{A}^+$, where $\mathcal{P}$ is the set of robot programs, such that for any robot program r, $\lambda\sigma\Gamma(r, \sigma)$ is a robot controller that satisfies the two conditions in the theorem with respect to r. We define Γ inductively on the structure of robot programs:

$$\Gamma(\underline{nil}, \sigma) = \begin{cases} stop & \text{if } \sigma = \varepsilon \\ \perp & \text{otherwise,} \end{cases}$$

$$\Gamma(\underline{exit}, \sigma) = \begin{cases} abort & \text{if } \sigma = \varepsilon \\ \perp & \text{otherwise,} \end{cases}$$

$$\Gamma(\underline{branch}(a, r_1, r_2), \sigma) = \begin{cases} a & \text{if } \sigma = \varepsilon \\ \Gamma(r_1, \sigma') & \text{if } \sigma = (a, 1) \circ \sigma' \\ \Gamma(r_2, \sigma') & \text{if } \sigma = (a, 0) \circ \sigma' \\ \perp & \text{otherwise,} \end{cases}$$

$$\Gamma(\underline{loop}(r_1, r_2), \sigma)$$

$$= \begin{cases} \Gamma(r_2, \sigma'') & \text{if for some } n \geqslant 0, \sigma = \sigma_1 \circ \cdots \circ \sigma_n \circ \sigma' \circ \sigma'' \text{ such that} \\ & \Gamma(r_1, \sigma_i) = stop \text{ for } 1 \leqslant i \leqslant n, \text{ and } \Gamma(r_1, \sigma') = abort. \\ \Gamma(r_1, \sigma') & \text{otherwise, where } \sigma' \text{ is a history} \\ & \text{such that for some } n \geqslant 0, \sigma = \sigma_1 \circ \cdots \circ \sigma_n \circ \sigma', \\ & \Gamma(r_1, \sigma_i) = stop \text{ for } 1 \leqslant i \leqslant n, \text{ and there is no} \\ & \text{proper prefix } \sigma'' \text{ of } \sigma' \text{ such that } \Gamma(r_1, \sigma'') = stop. \end{cases}$$

Clearly, for any program r, $\lambda\sigma\Gamma(r, \sigma)$ is a recursive function. We now show that $\lambda\sigma\Gamma(r, \sigma)$ satisfies the two conditions in Theorem B.2. We do so by induction over the structure of programs.

- r is $\underline{nil}$. For any ground situation term S, and any interpretation I:

$$I \models (\forall s)(Rdo(\underline{nil}, S, s) \equiv s = S) \wedge \neg(\exists s) Rexit(\underline{nil}, S, s).$$

From this, and the definition of $\Gamma(\underline{nil}, \sigma)$, the two conditions in the theorem are trivially satisfied.

- r is $\underline{exit}$. This case is analogous to the case of $\underline{nil}$: For any S, and any I,

$$I \models (\forall s)(Rexit(\underline{exit}, S, s) \equiv s = S) \wedge \neg(\exists s) Rdo(\underline{exit}, S, s).$$

- r is $\underline{branch}(a, r_1, r_2)$. Inductively, we assume that the two conditions are satisfied for $\lambda\sigma\Gamma(r_1, \sigma)$ and $\lambda\sigma\Gamma(r_2, \sigma)$. Let S be an arbitrary ground situation term, and I an arbitrary interpretation. Suppose $I \models Rdo(\underline{branch}(a, r_1, r_2), S, S')$. By the definition of Rdo, there are two cases:
  (1) $I \models SF(a, S) \wedge Rdo(r_1, do(a, S), S')$. By inductive assumption, there is a terminating run τ of $\lambda\sigma\Gamma(r_1, \sigma)$ under the environment determined by I at $do(a, S)$ such that $S' = end(\tau, do(a, S))$. By our construction of $\Gamma(\underline{branch}(a, r_1, r_2), \sigma)$, $(a, 1) \circ \tau$ is a terminating run of $\lambda\sigma\Gamma(\underline{branch}(a, r_1, r_2), \sigma)$ under the environment determined by I at S. [10] Furthermore,

$$end\big((a, 1) \circ \tau, S\big) = end\big(\tau, do(a, S)\big) = S'.$$

  (2) $I \models \neg SF(a, S) \wedge Rdo(r_2, do(a, S), S')$. Analogous to the previous case.
  This proves condition (1)(a) of Theorem B.2. The proof of (1)(b) is analogous.
  To prove (2)(a), suppose τ is a terminating run of $\lambda\sigma\Gamma(\underline{branch}(a, r_1, r_2), \sigma)$ under the environment $\mathcal{E}$ determined by I at S. Again, there are two cases:
  (1) $I \models SF(a, S)$. By our construction, $\tau = (a, 1) \circ \tau'$, and τ' is a terminating run of $\lambda\sigma\Gamma(r_1, \sigma)$ under the environment determined by I at $do(a, S)$. By inductive assumption,

$$I \models do(a, S) \leqslant S' \supset Rdo\big(r_1, do(a, S), S'\big),$$

  where $S' = end(\tau', do(a, S)) = end(\tau, S)$. By

$$I \models Poss(a, S) \wedge SF(a, S) \wedge Rdo\big(r_1, do(a, S), S'\big) \supset Rdo\big(\underline{branch}(a, r_1, r_2), S, S'\big),$$

  we have

$$I \models S \leqslant S' \supset Rdo\big(\underline{branch}(a, r_1, r_2), S, S'\big).$$

  (2) $I \models \neg SF(a, S)$. Analogous to the previous case.
  This proves (2)(a). The proof of (2)(b) is again analogous.
- r is $\underline{loop}(r_1, r_2)$. The proof for this case is exactly like that for the branch case, but using Lemma B.1. □

Appendix C. Proof of Theorem 3

We restate Theorem 3:

Theorem 3. *For any TM basic action theory T, any effective controller C, and any ground situation term S, there is a robot program r such that for any model I of T and any ground situation S', we have:*

(1) *If $I \models Rdo(r, S, S')$, then there is a terminating run σ of the system $(C, \mathcal{E})$ such that $clean(S') = end(\sigma, S)$, where $\mathcal{E}$ is the environment determined by I at S.*

[10] This makes use of easy lemma that if τ is a terminating run of C under the environment determined by I at $do(a, S)$, and $I \models SF(a, S)$, then $(a, 1) \circ \tau$ is a terminating run of C' under the environment determined by I at S, where C' is a controller such that $C'(\varepsilon) = a$ and $C'((a, 1) \circ \sigma) = C(\sigma)$.

(2) *If there is a terminating run* σ *of the system* $(\mathcal{C}, \mathcal{E})$ *such that* $S' = end(\sigma, S)$, *then there is a situation* S'' *such that* $S' = clean(S'')$, *and* $I \models S \leqslant end(\sigma, S) \supset Rdo(r, S, S'')$, *where* $\mathcal{E}$ *is the environment determined by* I *at* S.

Proof. The proof involves using the fluents *Marked* and *loc* to emulate a Turing machine tape. If $Marked(n, s)$ holds, then the nth position of the tape will be `1`, and otherwise, `0`. To indicate the state of some segment of the tape, we will show a sequence of `0`s and `1`s, with the current location (i.e., $loc(s)$) taken to be the position of the first digit of the underlined pair of `0`s and `1`s. Thus if the initial segment of the tape in S_0 is $\underline{00}1011$, then $loc(S_0) = 1$, $\neg Marked(1, S_0)$, $\neg Marked(2, S_0)$, $Marked(3, S_0)$, etc.

We assume without loss of generality that there are only two action A and B. We can encode a history as follows:

$$\underline{00}\alpha_1\beta_1 \ldots \alpha_n\beta_n 11,$$

where α_i is `01` or `10` for action A or action B respectively, and where β_i is `01` or `10` for outcome 0 or 1 respectively. Since C is a recursive function, there is a Turing machine M that computes it. Given a history encoded as above, it will terminate with the tape looking like

$$00\alpha_1\beta_1 \ldots \alpha_n\beta_n\underline{\alpha},$$

where α is either `00` (for *abort* or $\perp$), `11` (for *stop*) or `01` or `10` as above for A or B. Since the five special actions have the same power as a Turing machine, there is a robot program r_M that consists of only Turing actions that senses and marks the tape in exactly this way.

We can now describe the robot program that achieves the same goal as the controller C.

- For any fixed string z of `0`s and `1`s, let $write(z)$ be the robot program that writes z on the tape and sets *loc* to the position just right of z. It is defined inductively by:
  $write(\varepsilon) = \underline{nil}$;
  $write(0z) = \underline{seq}(erase, \underline{seq}(right, write(z)))$;
  $write(1z) = \underline{seq}(mark, \underline{seq}(right, write(z)))$.

- The robot program *home* is such that whenever the tape encodes some history like:

  $$00\alpha_1\beta_1 \ldots \alpha_n\beta_n\underline{\alpha},$$

  home will reset *loc* to 1:

  $\underline{loop}(\underline{seq}(left, \underline{branch}(read_mark,$
  $\underline{seq}(left, \underline{nil}),$
  $\underline{seq}(left, \underline{branch}(read_mark, \underline{nil}, \underline{exit})))),$
  $\underline{nil})$.

- The robot program *ini* is $[write(0011); home]$. [11]

[11] For any robot programs r and r', $r; r'$ means executing r followed by r'. Formally, the ";" notation is defined inductively:

$[\underline{nil}; r] = r$;
$[\underline{exit}; r] = \underline{exit}$;
$[\underline{branch}(a, r_1, r_2); r] = \underline{branch}(a, [r_1; r], [r_2; r])$;
$[\underline{loop}(r_1, r_2); r] = \underline{loop}(r_1, [r_2; r])$.

- The robot program r_A performs the action A, and writes either 0111 or 1011 on the tape depending on the sensing result returned:

  $\underline{branch}(A, [write(1011); home], [write(0111); home])$.

  The program r_B is analogous.
- The robot program *interpret* is

  $\underline{branch}(read_mark,$
  $\quad \underline{seq}(right, \underline{branch}(read_mark,$
  $\quad\quad \underline{seq}(right, \underline{exit}),$
  $\quad\quad \underline{seq}(right, r_B))),$
  $\quad \underline{seq}(right, \underline{branch}(read_mark,$
  $\quad\quad \underline{seq}(right, r_A),$
  $\quad\quad \underline{seq}(right, \underline{loop}(\underline{nil}, \underline{nil})))))$.

- Finally, the desired robot program r is $[ini; \underline{loop}([r_M; interpret], \underline{nil})]$ where r_M is the robot program associated with a single step of the effective controller $\mathcal{C}$.

To paraphrase, the robot program r, starting at some home position on the tape, first writes 0011 encoding the empty history, and then returns to the home position. Next, within a loop, it repeatedly uses r_M to place a suitable α at the end of the history, and then interprets this α: if it is 00 it goes into an infinite loop; if it is 11 it exits the loop (and so terminates); if it is 10, it performs action A, writes either 0111 or 1011 on the tape depending on the sensing result returned, and then returns to the home position; if it is 01, it does the same as above but for action B. Note that when α is 01 or 10, the effect of writing 0111 or 1011 on the tape depending on the result of action A or B ensures that the tape now encodes an extended history

$$\underline{00}\alpha_1\beta_1 \ldots \alpha_n\beta_n\alpha_{n+1}\beta_{n+1}11,$$

which is then ready for the next iteration.

It can be verified that from this construction the conditions in the theorem follow. □

References

[1] E. Davis, Knowledge preconditions for plans, J. Logic Comput. 4 (5) (1994) 721–766.

[2] O. Etzioni, S. Hanks, D. Weld, D. Draper, N. Lesh and M. Williamson, An approach to planning with incomplete information, in: Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference, Cambridge; MA, Morgan Kaufmann, San Mateo, CA, 1992, pp. 115–125.

[3] R.E. Fikes and N.J. Nilsson, STRIPS: a new approach to theorem proving in problem solving, Artificial Intelligence 2 (1971) 189–208.

[4] C. Green, Theorem proving by resolution as a basis for question-answering systems, in: B. Meltzer and D. Michie (Eds.), Machine Intelligence, Vol. 4, Edinburgh University Press, Edinburgh, 1969, pp. 183–205.

[5] Y. Lespérance, H. Levesque and F. Lin, Ability and knowing how in the situation calculus, in preparation.

[6] H. Levesque, R. Reiter, Y. Lespérance, F. Lin and R. Scherl, GOLOG: a logic programming language for dynamic domains, J. Logic Programming 31 (1–3) (1997) 59–83.

[7] H. Levesque, What is planning in the presence of sensing?, in: Proceedings AAAI-96, Portland, OR, 1996, pp. 1139–1146.

[8] F. Lin, Applications of the situation calculus to formalizing control and strategic information: the prolog cut operator, in: Proceedings Fifteen International Joint Conference on Artificial Intelligence (IJCAI-97), Nagoya, Japan, Morgan Kaufmann, San Mateo, CA, 1997, pp. 1412–1418.

[9] F. Lin, An ordering on subgoals for planning, Ann. Math. Artificial Intelligence 21 (1997) 321–342.
[10] F. Lin, On measuring plan quality, in: Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), 1998.
[11] F. Lin and R. Reiter, State constraints revisited, in: J. Logic Comput. 4 (5) (1994) 655–678.
[12] F. Lin and R. Reiter, How to progress a database, Artificial Intelligence 92 (1–2) (1997) 131–167.
[13] K. Krebsbach, D. Olawsky and M. Gini, An empirical study of sensing and defaulting in planning, in: Proceedings 1st Conference on AI Planning Systems, San Mateo, CA, 1992, pp. 136–144.
[14] Z. Manna and R. Waldinger, How to clear a block: a theory of plans, J. Automated Reasoning 3 (1987) 343–377.
[15] J. McCarthy and P.J. Hayes, Some philosophical problems from the standpoint of artificial intelligence, in: B. Meltzer and D. Michie (Eds.), Machine Intelligence, Vol. 4, Edinburgh University Press, Edinburgh, 1969, pp. 463–502.
[16] E. Mendelson, An Introduction to Mathematical Logic, Van Rostrand Reinhold Company, New York, 1964.
[17] E.P. Pednault, ADL: exploring the middle ground between STRIPS and the situation calculus, in: Proceedings First International Conference on Principles of Knowledge Representation and Reasoning (KR'89), Toronto, Ont., Morgan Kaufmann, San Mateo, CA, 1989, pp. 324–332.
[18] J.S. Penberthy and D.S. Weld, UCPOP: a sound, complete, partial order planner for ADL, in: Proceedings Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92), Cambridge, MA, Morgan Kaufmann, San Mateo, CA, 1992, pp. 103–114.
[19] M. Peot and D. Smith, Conditional nonlinear planning, in: Proceedings 1st Conference on AI Planning Systems, San Mateo, CA, 1992, pp. 189–197.
[20] R. Reiter, The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression, in: V. Lifschitz (Ed.), Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy, Academic Press, San Diego, CA, 1991, pp. 359–380.
[21] R. Reiter, Proving properties of states in the situation calculus, Artificial Intelligence 64 (1993) 337–351.
[22] R. Scherl and H. Levesque, The frame problem and knowledge-producing actions, in: Proceedings AAAI-93, Washington, DC, AAAI Press/The MIT Press, CA, 1993, pp. 689–695.
[23] M. Schoppers, Building plans to monitor and exploit open-loop and closed-loop dynamics, in: Proceedings 1st Conference on AI Planning Systems, San Mateo, CA, 1992, pp. 204–213.
[24] S. Thomas, PLACA, an agent oriented programming language, Ph.D. Thesis, Department of Computer Science, Stanford University, 1993.
[25] W. van der Hoek, B. van Linder and J.-J.Ch. Meyer, A logic of capabilities, in: A. Nerode and Y. Matiyasevitch (Eds.), Proceedings LFCS-94, the 3rd International Symposium on the Logical Foundations of Computer Science, Springer, Berlin, 1994, pp. 366–378.

NORTH-HOLLAND

GOLOG: A LOGIC PROGRAMMING LANGUAGE FOR DYNAMIC DOMAINS

HECTOR J. LEVESQUE, RAYMOND REITER, YVES LESPÉRANCE,* FANGZHEN LIN,† AND RICHARD B. SCHERL‡

▷ This paper proposes a new logic programming language called GOLOG whose interpreter automatically maintains an explicit representation of the dynamic world being modeled, on the basis of user supplied axioms about the preconditions and effects of actions and the initial state of the world. This allows programs to reason about the state of the world and consider the effects of various possible courses of action before committing to a particular behavior. The net effect is that programs may be written at a much higher level of abstraction than is usually possible. The language appears well suited for applications in high level control of robots and industrial processes, intelligent software agents, discrete event simulation, etc. It is based on a formal theory of action specified in an extended version of the situation calculus. A prototype implementation in Prolog has been developed. © Elsevier Science Inc., 1997 ◁

This research received financial support from the Information Technology Research Center (Ontario, Canada), the Institute for Robotics and Intelligent Systems (Canada), and the Natural Science and Engineering Research Council (Canada). Levesque and Reiter are fellows of the Canadian Institute for Advanced Research.

Address correspondence to Department of Computer Science, University of Toronto, Toronto, ON, M5S 1A4 Canada, Email: {hector, reiter, lesperan, fl, scherl}@cs.toronto.edu, http://www.cs.toronto.edu/~cogrobo/.

* Current address: Department of Computer Science, Glendon College, York University, 2275 Bayview Ave., Toronto, ON, Canada M4N 3M6.

† Current address: Department of Computer Science, Hong Kong University of Science and Technology, Kowloon, Hong Kong.

‡ Current address: Department of Computer and Information Science, New Jersey Institute of Technology, University Heights, Newark, NJ 07102, USA.

Received September 1995; accepted October 1996.

© Elsevier Science Inc., 1997
655 Avenue of the Americas, New York, NY 10010

0743-1066/97/$17.00
PII S0743-1066(96)00121-5

1. INTRODUCTION

Computer systems are often embedded in complex environments with which they interact. In programming such applications, the designer normally has an elaborate mental model of the environment and how the system's action will change the environment's state. Users of the system also have this kind of mental model. Typically, however, the system itself does not maintain an explicit model of the world it is operating in. This can make life difficult both for programmers and users —they may end up having to reconstruct the model being used, as there is no way for the system to explain or justify its behavior. But more importantly, this makes it difficult to reconfigure or extend the system by giving it "high-level" instructions, since it has no understanding at all of what it is doing.[1]

In this paper, we propose a programming language for such systems, whose design is based on a sophisticated logic of action. The interpreter for the language automatically maintains an explicit model of the system's environment and capabilities, which can be queried and reasoned with at run time. This allows complex behaviors to be defined at a much higher level of abstraction than would be possible otherwise. The language appears to be a distinct improvement over current technology for application such as: high-level control of robots and mechanical devices, programming intelligent software agents, modeling and simulation of discrete event systems, etc.

In the next section, we outline the theory of action on which our language is based. Then, we show how complex actions can be defined in the framework and explain how the resulting set of complex action expressions can be viewed as a programming language. In Section 4, we illustrate how our language is used through an example: a simple elevator controller. In the following section, we describe an implementation of the language, and sketch what experimental applications have been developed. Section 6 discusses the main distinguishing characteristics of the language. We conclude by summarizing the main features of our proposal, discussing its limitations, and outlining ongoing work that seeks to address these. The presentation throughout is informal in nature; in a companion paper [14], we explore the more formal aspects of this work.

2. INFORMAL INTRODUCTION TO THE SITUATION CALCULUS

To obtain the benefits mentioned in the Introduction, it is necessary to explicitly model how the world changes as a result of performing actions. There are a variety of ways of doing this, and we use the language of the situation calculus.

2.1. Intuitive Ontology for the Situation Calculus

The situation calculus (McCarthy [20]) is a first order language (with, as we shall see later, some second order features) specifically designed for representing dynamically changing worlds. All changes to the world are the result of named *actions*. A possible world history, which is simply a sequence of actions, is represented by a first order term called a *situation*. The constant S_0 is used to denote

[1] A similar view is advanced in Dixon [3].

the *initial situation*, namely that situation in which no actions have yet occurred. There is a distinguished binary function symbol *do*; $do(\alpha, s)$ denotes the successor situation to s resulting from performing the action α. Actions may be parameterized. For example, $put(x, y)$ might stand for the action of putting object x on object y, in which case $do(put(A, B), s)$ denotes that situation resulting from placing A on B when the world is in situation s. Notice that in the situation calculus, actions are denoted by first order terms, and situations (world histories) are also first order terms. For example, $do(putdown(A), do(walk(L), do(pickup(A), S_0)))$ is a situation denoting the world history consisting of the sequence of actions [pickup(A), walk(L), putdown(A)]. Notice that the sequence of actions in a history, in the order in which they occur, is obtained from a situation term by reading off its actions from right to left.

Relations whose truth values vary from situation to situation, called *relational fluents*, are denoted by predicate symbols taking a situation term as their last argument. For example, $is_carrying(robot, p, s)$, meaning that a robot is carrying package p in situation s, is a relational fluent. Functions whose denotations vary from situation to situation are called *functional fluents*. They are denoted by function symbols with an extra argument taking a situation term, as in $loc(robot, s)$, i.e., the robot's location in situation s.

2.2. Axiomatizing Actions and Their Effects in the Situation Calculus

Actions have *preconditions*—necessary and sufficient conditions that characterize when the action is physically possible. For example, in a blocks world, we might have:[2]

$$Poss(pickup(x), s) \equiv [(\forall z) \neg holding(z, s)] \wedge nexto(x, s) \wedge \neg heavy(x).$$

World dynamics are specified by *effect axioms*. These describe the effects of a given action on the fluents—the causal laws of the domain. For example, a robot dropping a fragile object causes it to be broken:

$$Poss(drop(r, x), s) \wedge fragile(x, s) \supset broken(x, do(drop(r, x), s)). \tag{2.1}$$

Exploding a bomb next to an object causes it to be broken:

$$Poss(explode(b), s) \wedge nexto(b, x, s) \supset broken(x, do(explode(b), s)). \tag{2.2}$$

A robot repairing an object causes it to be not broken:

$$Poss(repair(r, x), s) \supset \neg broken(x, do(repair(r, x), s)). \tag{2.3}$$

2.3. The Frame Problem

As first observed by McCarthy and Hayes [20], axiomatizing a dynamic world requires more than just action preconditions and effect axioms. So-called *frame axioms* are also necessary. These specify the action *invariants* of the domain,

[2] In formulas, free variables are considered to be universally quantified from the outside. This convention will be followed throughout the paper.

namely, those fluents which remain unaffected by a given action. For example, a robot dropping things does not affect an object's color:

$$Poss(drop(r,x),s) \wedge color(y,s) = c \supset color(y, do(drop(r,x),s)) = c.$$

A frame axiom describing how the fluent *broken* remains unaffected:

$$Poss(drop(r,x),s) \wedge \neg broken(y,s) \wedge [y \neq x \vee \neg fragile(y,s)]$$
$$\supset \neg broken(y, do(drop(r,x),s)).$$

The problem introduced by the need for such frame axioms is that we can expect a vast number of them. Only relatively few actions will affect the truth value of a given fluent; all other actions leave the fluent unchanged. For example, an object's color is not changed by picking things up, opening a door, going for a walk, electing a new prime minister of Canada, etc. This is problematic for the axiomatizer who must think of all these axioms; it is also problematic for the theorem proving system, as it must reason efficiently in the presence of so many frame axioms.

2.3.1. What Counts as a Solution to the Frame Problem? Suppose the person responsible for axiomatizing an application domain has specified all of the causal laws for the world being axiomatized. More precisely, she has succeeded in writing down *all* the effect axioms, i.e., for each fluent F and each action A which can cause F's truth value to change, axioms of the form

$$Poss(A,s) \wedge R(\vec{x},s) \supset (\neg)F(\vec{x}, do(A,s)).$$

Here, R is a first order formula specifying the contextual conditions under which the action A will have its specified effect on F.

A solution to the frame problem is a systematic procedure for generating, from these effect axioms, all the frame axioms. If possible, we also want a *parsimonious* representation for these frame axioms (because in their simplest form, there are too many of them).

2.4. A Simple Solution to the Frame Problem

By appealing to earlier ideas of Haas [7], Schubert [29] and Pednault [21], Reiter [23] proposes a simple solution to the frame problem, which we illustrate with an example. Suppose that (2.1), (2.2), and (2.3) are all the effect axioms for the fluent *broken*, i.e., they describe all the ways that an action can change the truth value of *broken*. We can rewrite (2.1) and (2.2) in the logically equivalent form:

$$Poss(a,s) \wedge [(\exists r)\{a = drop(r,x) \wedge fragile(x,s)\}$$
$$\vee (\exists b)\{a = explode(b) \wedge nexto(b,x,s)\}] \qquad (2.4)$$
$$\supset broken(x, do(a,s)).$$

Similarly, consider the negative effect axiom (2.3) for *broken*; this can be rewritten as:

$$Poss(a,s) \wedge (\exists r)a = repair(r,x) \supset \neg broken(x, do(a,s)). \qquad (2.5)$$

In general, we can assume that the effect axioms for a fluent F have been written in the forms:

$$Poss(a,s) \wedge \gamma_F^+(\vec{x},a,s) \supset F(\vec{x},do(a,s)), \tag{2.6}$$

$$Poss(a,s) \wedge \gamma_F^-(\vec{x},a,s) \supset \neg F(\vec{x},do(a,s)). \tag{2.7}$$

Here $\gamma_F^+(\vec{x},a,s)$ is a formula describing under what conditions doing the action a in situation s leads the fluent F to become true in the successor situation $do(a,s)$; similarly $\gamma_F^-(\vec{x},a,s)$ describes the conditions under which performing a in s results in F becoming false in the successor situation. The solution to the frame problem of [23] rests on a *completeness assumption*, which is that the causal axioms (2.6) and (2.7) characterize all the conditions under which action a can lead to a fluent $F(\vec{x})$ becoming true (respectively, false) in the successor situation. In other words, axioms (2.6) and (2.7) describe all the causal laws affecting the truth values of the fluent F. Therefore, if action a is possible and $F(\vec{x})$'s truth value changes from *false* to *true* as a result of doing a, then $\gamma_F^+(\vec{x},a,s)$ must be *true* and similarly for a change from *true* to *false*. Reiter [23] shows how to derive a *successor state axiom* of the following form from the causal axioms (2.6) and (2.7) and the completeness assumption.

Successor State Axiom

$$Poss(a,s) \supset \left[F(\vec{x},do(a,s)) \equiv \gamma_F^+(\vec{x},a,s) \vee \left(F(\vec{x},s) \wedge \neg \gamma_F^-(\vec{x},a,s)\right)\right].$$

This simple axiom embodies a solution to the frame problem. Notice that this axiom universally quantifies over actions a. In fact, this is one way in which a parsimonious solution to the frame problem is obtained.

Applying this to our example about breaking things, we obtain the following successor state axiom:

$$\begin{aligned} Poss(a,s) \supset [&broken(x,do(a,s)) \\ &\equiv (\exists r)\{a = drop(r,x) \wedge fragile(x,s)\} \\ &\vee (\exists b)\{a = explode(b) \wedge nexto(b,x,s)\} \\ &\vee broken(x,s) \wedge \neg(\exists r)a = repair(r,x)]. \end{aligned}$$

It is important to note that the above solution to the frame problem presupposes that there are no *state constraints*, as for example in the blocks world constraint: $(\forall s).on(x,y,s) \supset \neg on(y,x,s)$. Such constraints sometimes implicitly contain effect axioms (so-called indirect effects), in which case the above completeness assumption will not be true. The assumption that there are no state constraints in the axiomatization of the domain will be made throughout this paper. In [17, 15], the approach discussed in this section is extended to deal with state constraints, by compiling their effects into the successor state axioms.

2.5. Axiomatizing an Application Domain in the Situation Calculus

In general, a particular domain of application will be specified by the union of the following sets of axioms:

- Action precondition axioms, one or each primitive action.
- Successor state axioms, one for each fluent.

- Unique names axioms for the primitive actions.
- Axioms describing the initial solution—what is true initially, before any actions have occurred. This is any finite set of sentences which mention no situation term, or only the situation term S_0.
- Foundational, domain independent axioms for the situation calculus. These include unique names axioms for situations, and an induction axiom. Since these play no special role in this paper, we omit them. For details, and for their metamathematical properties, see Lin and Reiter [17] and Reiter [24].

3. COMPLEX ACTIONS, PROCEDURES, AND GOLOG

The previous section outlines a situation calculus-based approach for representing, and reasoning about, simple actions. It fails to address the problem of expressing, and reasoning with, complex actions and procedures, for example:

- **if** *car_in_driveway* **then** *drive* **else** *walk* **endIf**
- **while** $(\exists block)$ *ontable*(*block*) **do** *remove_a_block* **endWhile**
- **proc** *remove_a_block* (πx)[*pickup*(x); *putaway*(x)] **endProc**

Here, we have introduced a procedure declaration (*remove_a_block*), and also the nondeterministic operator π; $(\pi x)[\delta(x)]$ means nondeterministically pick an individual x, and for that x, perform $\delta(x)$. We shall see later that this kind of nondeterminism is very useful for robotics and similar applications.

3.1. Complex Actions and Procedures in the Situation Calculus

Our approach will be to define complex action expressions using some additional extralogical symbols (e.g., **while**, **if**, etc.) which act as *abbreviations* for logical expressions in the language of the situation calculus. These extralogical expressions should be thought of as *macros* which expand into genuine formulas of the situation calculus. So below, we define the abbreviation $Do(\delta, s, s')$, where δ is a complex action expression; intuitively, $Do(\delta, s, s')$ will hold whenever the situation s' is a terminating situation of an execution of complex action δ starting in situation s. Note that our complex actions may be nondeterministic, that is, may have several different executions terminating in different situations.

Do is defined intuitively on the structure of its first argument as follows:

1. *Primitive actions*:

$$Do(a, s, s') \stackrel{def}{=} Poss(a[s], s) \wedge s' = do(a[s], s).$$

By the notation $a[s]$ we mean the result of restoring the situation arguments to any functional fluents mentioned by the action term a (see the next item immediately below). For example, if a is *read*(*favorite_book*(*John*)), and if *favorite_book* is a functional fluent (which means that its value is situation dependent) then $a[s]$ is *read*(*favorite_book*(*John*, s)).

2. *Test actions*:

$$Do(\phi?, s, s') \stackrel{def}{=} \phi[s] \wedge s = s'.$$

Here, ϕ is a pseudo-fluent expression (not a situation calculus formula) which stands for a formula in the language of the situation calculus, but with all situation arguments suppressed. $\phi[s]$ denotes the situation calculus formula obtained from ϕ by restoring situation variable s as the suppressed situation argument for all fluent names (relational and functional) mentioned in ϕ.

Examples: If ϕ is

$$(\forall x).ontable(x) \wedge \neg on(x, A),$$

then $\phi[s]$ stands for

$$(\forall x).ontable(x, s) \wedge \neg on(x, A, s).$$

If ϕ is

$$(\exists x) on(x, favorite_block(Mary)),$$

then $\phi[s]$ stands for

$$(\exists x) on(x, favorite_block(Mary, s), s).$$

3. *Sequence*:

$$Do([\delta_1; \delta_2], s, s') \stackrel{def}{=} (\exists s^*).(Do(\delta_1, s, s^*) \wedge Do(\delta_2, s^*, s')).$$

4. *Nondeterministic choice of two actions*:

$$Do((\delta_1 | \delta_2), s, s') \stackrel{def}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s').$$

5. *Nondeterministic choice of action arguments*:

$$Do((\pi x)\, \delta(x), s, s') \stackrel{def}{=} (\exists x)\, Do(\delta(x), s, s').$$

6. *Nondeterministic iteration*: Execute δ zero or more times.

$$\begin{aligned} &Do(\delta^*, s, s') \\ &\quad \stackrel{def}{=} (\forall P).\{(\forall s_1) P(s_1, s_1) \wedge \\ &\qquad\qquad (\forall s_1, s_2, s_3)[P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \supset P(s_1, s_3)]\} \\ &\quad \supset P(s, s'). \end{aligned}$$

In other words, doing action δ zero or more times takes you from s to s' iff (s, s') is in every set (and therefore, the smallest set) such that:
(a) (s_1, s_1) is in the set for all situations s_1.
(b) Whenever (s_1, s_2) is in the set, and doing δ in situation s_2 takes you to situation s_3, then (s_1, s_3) is in the set.
The above definition of nondeterministic iteration utilizes the standard second order way of expressing this set. Some appeal to second order logic appears necessary here because transitive closure is not first order definable, and nondeterministic iteration appeals to this closure.

Conditionals and white-loops can be defined in terms of the above constructs as follows:

if ϕ **then** δ_1 **else** δ_2 **endIf** $\stackrel{def}{=} [\phi?; \delta_1] | [\neg \phi?; \delta_2]$,

while ϕ **do** δ **endWhile** $\stackrel{def}{=} [[\phi?; \delta]^*; \neg \phi?]$.

Procedures:

The difficulty with giving a situation calculus semantics for recursive procedure calls using macro expansion is that there is no straightforward way to macro expand a procedure body when that body includes a recursive call to itself.

1. We begin with an auxiliary macro definition: For any predicate symbol P of arity $n+2$, taking a pair of situation arguments:

$$Do(P(t_1,\ldots,t_n),s,s') \stackrel{def}{=} P(t_1[s],\ldots,t_n[s],s,s').$$

In what follows, expressions of the form $P(t_1,\ldots,t_n)$ occurring in programs will serve as procedure calls, and we will understand $Do(P(t_1,\ldots,t_n),s,s')$ to mean that executing the procedure P on actual parameters $t_1,\ldots,t_n$ causes a transition from situation s to s'. Notice that in the macro expansion, the actual parameters (t_i) are first evaluated with respect to the current situation s $(t_i[s])$ before passing them to the procedure P, so the procedure mechanism we are defining is *call by value*. Because we now want to include procedure calls among our actions, we extend the definition of complex actions to consist of any expression that can be constructed from primitive actions and procedure calls using the complex action constructors of 1–6 above.

2. Next, we define a situation calculus semantics for *programs* involving (recursive) procedures. We suppose, in the standard block-structured programming style, that a GOLOG program consists of a sequence of declarations of procedures $P_1,\ldots,P_n$, with formal parameters $\vec{v}_1,\ldots,\vec{v}_n$ and procedure bodies $\delta_1,\ldots,\delta_n$ respectively, followed by a main program body δ_0. Here, $\delta_1,\ldots,\delta_n,\delta_0$ are complex actions, extended by actions for procedure calls, as described in 1 above. So a GOLOG program will have the form:

$$\textbf{proc } P_1(\vec{v}_1)\,\delta_1 \textbf{ endProc};\ldots;\textbf{ proc } P_n(\vec{v}_n)\,\delta_n \textbf{ endProc};\ \delta_0.$$

We define the result of evaluating a program of this form as follows:

$$Do\left(\{\textbf{proc } P_1(\vec{v}_1)\,\delta_1 \textbf{ endProc};\ldots;\textbf{ proc } P_n(\vec{v}_n)\,\delta_n \textbf{ endProc};\ \delta_0\},s,s'\right)$$
$$\stackrel{def}{=} (\forall P_1,\ldots,P_n).\left[\bigwedge_{i=1}^{n}(\forall s_1,s_2,\vec{v}_i).Do(\delta_i,s_1,s_2)\supset Do(P_i(\vec{v}_i),s_1,s_2)\right]$$
$$\supset Do(\delta_0,s,s').$$

This is the situation calculus definition corresponding to the more usual Scott-Strachey *least fixed-point* definition in standard programming language semantics (Stoy [32]).[3]

Examples:

1. Given that *down* means move an elevator down one floor, define $d(n)$, meaning move an elevator down n floors.

$$\textbf{proc } d(n)(n=0)?\,|\,d(n-1);\ \textit{down}\ \textbf{endProc}.$$

[3] By using programs as above within the bodies of other procedures, we obtain the tree-structured nesting of procedures typical of Algol-like languages. Moreover, we get the lexical scoping rules of these languages for free from our use of the quantifiers in the definition of *Do*.

2. Parking an elevator on the ground floor:

   proc *park* $(\pi m)[atfloor(m)?; d(m)]$ **endProc**.

3. Define *above* to be the test action which is the transitive closure of *on*.

   proc $above(x, y)(x = y)?|(\pi z)[on(x, z)?; above(z, y)]$ **endProc**.

4. *clean* means put away all the blocks into the box.

   proc *clean* $(\forall x)[block(x) \supset in(x, Box)]?|$

   $(\pi x)[(\forall y) \neg on(y, x)?; put(x, Box)]$; *clean* **endProc**.

5. A GOLOG blocks world program consisting of three procedure declarations devoted to creating towers of blocks, and a main program which makes a seven block tower, while ensuring that block A is clear in the final situation.

```
proc maketower(n)          % Make a tower of n blocks.
  (πx, m)[tower(x, m) ?;   % tower(x, m) means that there is a tower
                           % of m blocks, whose top block is x.
  if m ≤ n then stack(x, n − m)
    else unstack(x, m − n)
   endIf]
endProc;

proc stack(x, n)           % Place n blocks on the tower whose top block is x.
  n = 0?|(π y)[put(y, x); stack(y, n − 1)]
endProc;

proc unstack(x, n)         % Remove n blocks from the tower
                           % whose top block is x.
  n = 0?|(π y)[on(x, y)?; movetotable(x); unstack(y, n − 1)]
endProc;
% main: create a seven block tower, with A clear at the end.
maketower(7); ¬(∃x)on(x, A)?
```

Except for procedures, this formalization draws considerably from dynamic logic [5]. In effect, it reifies as situations in the object language of the situation calculus, the possible worlds with which the semantics of dynamics logic is defined. For a more technical treatment of this macro approach to complex actions, see Levesque, Lin, and Reiter [14].

3.2. Why Macros?

Programs and complex actions "macro expand" to (sometimes second order) formulas of the situation calculus; *complex behaviors are described by situation calculus formulas*. But why do we treat these as macros rather than as first class objects (terms) in the language of the situation calculus? To see why, consider the complex action

while $[(\exists block)ontable(block)]$ **do** *remove_a_block* **endWhile**.

Now ask what kind of thing is *ontable*(*block*)? It is not a fluent, since fluents take situations as arguments. But it is meant to stand for a fluent since the expression *ontable*(*block*) will be evaluated with respect to the current situation of the execution of the **while**-loop. To see what must happen if we avoid the macro approach, suppose we treat complex actions as genuine first order terms in the language of the situation calculus.

- We must augment this language with new distinguished function symbols ?, ;, |, π, and perhaps *while*, *if_then_else*.
- Moreover, since a **while**-loop is now a first order term, the p in *while*(p, a) must be a first order term also. But p can be any "formula" standing for a situation calculus formula, e.g., *ontable*(*block*), $(\exists x, y).$*ontable*$(x) \wedge \neg$ *red*$(x) \vee$ *on*(x, y).
- So we must introduce new function symbols into the language; $\widehat{on}$, $\widehat{ontable}$, *and*, *or*, *exists*, *not* etc. (We need $\widehat{on}$ to distinguish it from the fluent *on*.) Now these "formulas" look like genuine terms:

  $$\widehat{ontable}(block),$$

  $$exists\Big(X, exists\Big(Y, or\Big(and\big(\widehat{ontable}(X), not\big(\widehat{red}(X)\big)\big), \widehat{on}(X, y)\Big)\Big)\Big).$$

  Notice that X and Y here must be constants. In other words, we must *reify* fluents and formulas about fluents whose situation arguments have been suppressed. This makes the resulting first order language much more complicated.
- Even worse, we must *axiomatize* the correspondence between these reified formulas and the actual situation calculus formulas they stand for. In the axioms for *Do*, such reified formulas get evaluated as

  $$Do(p?, s, s') \equiv apply(p, s) \wedge s = s'.$$

  Here, *apply*(p, s) is true iff the reified formula p, with its situation argument s restored (so that it becomes a genuine situation calculus formula), is true. So we have to axiomatize *apply*. These axioms are schemas over fluents F and reified formulas p, p_1, p_2 and the quantified "variables" X of these expressions.

  $$apply\big(\hat{F}(t_1, \ldots, t_n), s\big) \equiv F(apply1(t_1, s), \ldots, apply1(t_n, s), s),$$

  where *apply*1 restores situation arguments to functional fluents. Also needed are:

  $$apply(and(p_1, p_2), s) \equiv apply(p_1, s) \wedge apply(p_2, s),$$

  $$apply(or(p_1, p_2), s) \equiv apply(p_1, s) \vee apply(p_2, s),$$

  etc.

All of this would result in a much more complex theory. To avoid this technical clutter, we have chosen to take the above macro route in defining complex actions, and to see just how far we can push this idea. As we shall see, it is possible to develop a very rich theory of actions this way.

3.3. Programs as Macros: What Price Do We Pay?

By opting to define programs as macros, we obtain a much simpler theory than if we were to reify these actions. The price we pay for this is a less expressive formalism. For example, we cannot *quantify* over complex actions, since these are not objects in the language of the situation calculus. This means, for example, that we cannot synthesize programs using conventional theorem proving techniques, as in Manna and Waldinger [19]. In their approach to program synthesis, one would obtain a program satisfying the goal formula *Goal* as a side effect of proving the following entailment:

$$Axioms \models (\exists \delta, s).Do(\delta, S_0, s) \wedge Goal(s).$$

Here, *Axioms* are those described in Section 2.5. But the program to be synthesized is being existentially quantified in the theorem, so that this theorem cannot even be expressed in our language.

On the other hand, many other program properties are, in principle, provable with our formalism. Moreover, doing so is (conceptually) straightforward precisely because program executions are formulas of the situation calculus.

1. *Correctness*: To show that, whenever program δ terminates, it leads to a world situation satisfying property P:

   $$Axioms \models (\forall s).Do(\delta, S_0, s) \supset P(s).$$

   Or, the stronger

   $$Axioms \models (\forall s_0, s).Do(\delta, s_0, s) \supset P(s).$$

2. *Termination*: To show that program δ terminates:

   $$Axioms \models (\exists s) Do(\delta, S_0, s).$$

   Or, the stronger

   $$Axioms \models (\forall s_0)(\exists s) Do(\delta, s_0, s).$$

In other words, our macro account is well-suited to applications where a program δ is *given*, and the job is to prove it has some property. As we will see, the main property we have been concerned with is execution: given δ and an initial situation, find a terminating situation for δ, if one exists. To do so, we prove the termination of δ as above, and then extract from the proof a binding for the terminating situation.

3.4. GOLOG

The program and complex action expressions defined above can be viewed as a programming language whose semantics is defined via macro-expansion into sentences of the situation calculus. We call this language GOLOG, for "alGOl in LOGic." GOLOG appears to offer significant advantages over current tools for applications in dynamic domains like the high-level programming of robots and software agents, process control, discrete event simulation, etc. In the next section, we present a simple example.

4. AN ELEVATOR CONTROLLER IN GOLOG

Here we show how to axiomatize the primitive actions and fluents for a simple elevator, and we write a GOLOG program to control this elevator.

Primitive actions:

- *up*(n)—Move the elevator up to floor n.
- *down*(n)—Move the elevator down to floor n.
- *turnoff*(n)—Turn off call button n.
- *open*—Open the elevator door.
- *close*—Close the elevator door.

Fluents:

- *current_floor*$(s) = n$—In situation s, the elevator is at floor n.
- *on*(n, s)—In situation s, call button n is on.
- *next_floor*(n, s)—In situation s, the next floor to be served is n.

Primitive action preconditions:

$Poss(up(n), s) \equiv current_floor(s) < n.$

$Poss(down(n), s) \equiv current_floor(s) > n.$

$Poss(open, s) \equiv true.$

$Poss(close, s) \equiv true.$

$Poss(turnoff(n), s) \equiv on(n, s).$

Successor state axioms:

$$Poss(a, s) \supset [current_floor(do(a, s)) = m \equiv \{a = up(m) \vee a = down(m) \vee current_loor(s) = m \wedge \neg(\exists n)a = up(n) \wedge \neg(\exists n)a = down(n)\}].$$

$$Poss(a, s) \supset [on(m, do(a, s)) \equiv on(m, s) \wedge a \neq turnoff(m)].$$

A defined fluent.

$$next_floor(n, s) \equiv on(n, s) \wedge (\forall m).on(m, s) \supset |m - current_floor(s)| \geq |n - current_floor(s)|.$$

This defines the next floor to be served as a nearest floor to the one where the elevator happens to be.

The GOLOG procedures:

proc *serve*(n) *go_floor*(n); *turnoff*(n); *open*; *close* **endProc**.

proc *go_floor*(n) $(current_floor = n)?|up(n)|down(n)$ **endProc**.

proc *serve_a_floor* $(\pi n)[next_floor(n)?;$ *serve* $(n)]$ **endProc**.

proc *control* **[while** $(\exists n)on(n)$ **do** *serve_a_floor* **endWhile]**; *park* **endProc**.

proc *park* **if** *current_floor* $= 0$ **then** *open* **else** *down*(0); *open* **endIf endProc**.

Initial situation:

$$current_floor(S_0) = 4, \qquad on(b, S_0) \equiv b = 3 \vee b = 5.$$

Notice that this last axioms specifies that, initially, buttons 3 and 5 are on, and moreover no other buttons are on. In other words, we have complete information initially about which call buttons are on. It is this completeness property of the initial situation which will justify, in part, the Prolog implementation described below in Section 5.

Running the program:

This is a theorem proving task; we need to establish the following entailment:

$$Axioms \models (\exists s)\, Do(control, S_0, s).^4$$

Here, *Axioms* are those of Section 2.5. Notice especially what this entailment says, and why it makes sense.

- Although the expression $Do(control, S_0, s)$ looks like an atomic formula, *Do* is a macro not a predicate, and the expression stands for a much longer second order situation calculus sentence. This will mention only the primitive actions *up*, *down*, *turnoff*, *open*, *close* and the fluents *current_floor*, *on*, *next_floor*, as well as the distinguished situation calculus symbols *do*, S_0, *Poss*.
- Because this macro-expanded sentence is legitimate situation calculus, it makes sense to seek a proof of it from *Axioms*, which characterize the fluents and actions of this elevator world.

A successful "execution" of the program, i.e., a successful proof, might return the following binding for s:

$$s = do\Big(open, do\Big(down(0), do\Big(close, do\Big(open, do\Big(turnoff(5),$$
$$do\Big(up(5), do\Big(close, do\Big(open, do\Big(turnoff(3), do(down(3), S_0)\Big)\Big)\Big)\Big)\Big)\Big)\Big)\Big)\Big).$$

Such a binding represents an execution trace of the GOLOG program for the given description of the initial situation. This trace, namely, the action sequence

$$[down(3), turnoff(3), open, close, up(5), turnoff(5), open, close, down(0), open],$$

would next be passed to the elevator's execution module for controlling it in the physical world.

As one can see from the example, GOLOG is a logic programming language in the following sense:

1. Its interpreter is a general-purpose theorem prover. In its most general form, this must be a theorem prover for second order logic; in practice (see Section 6 below, and Levesque, Lin, and Reiter [14]), first order logic is sufficient for most purposes.
2. Like Prolog, GOLOG programs are executed for their side effects, namely, to obtain bindings for the existentially quantified variables of the theorem.

[4] Strictly speaking, we must prove the sentence $(\exists s)Do(\Pi; control, S_0, s)$ where Π is the sequence of procedure declarations just given. The call to *control* in this sentence serves as the main program. See the definition of GOLOG programs and their semantics in Section 3.1 above.

```
:- op(950, xfy, [#]). /* Nondeterministic action choice.*/

do([],S,S).      /* This clause and the next are for sequences */
do([E|L],S,S1) :- do(E,S,S2), do(L,S2,S1).
do(?(P),S,S) :- holds(P,S).
do(E1 # E2,S,S1) :- do(E1,S,S1) ; do(E2,S,S1).
do(if(P,E1,E2),S,S1) :- do([?(P),E1] # [?(neg(P)),E2],S,S1).
do(star(E),S,S1) :- do([] # [E,star(E)],S,S1).
do(while(P,E),S,S1):- do([star([?(P),E]),?(neg(P))],S,S1).
do(pi(V,E),S,S1) :- sub(V,_,E,E1), do(E1,S,S1).
do(E,S,S1) :- proc(E,E1), do(E1,S,S1).
do(E,S,do(E,S)) :- primitive_action(E), poss(E,S).

/* sub(Name,New,Term1,Term2): Term2 is Term1 with Name replaced by New. */
sub(X1,X2,T1,T2) :- var(T1), T2 = T1.
sub(X1,X2,T1,T2) :- not var(T1), T1 = X1, T2 = X2.
sub(X1,X2,T1,T2) :- not T1 = X1, T1 =..[F|L1], sub_list(X1,X2,L1,L2),
                    T2 =..[F|L2].
sub_list(X1,X2,[],[]).
sub_list(X1,X2,[T1|L1],[T2|L2]) :- sub(X1,X2,T1,T2), sub_list(X1,X2,L1,L2).

holds(and(P1,P2),S) :- holds(P1,S), holds(P2,S).
holds(or(P1,P2),S) :- holds(P1,S); holds(P2,S).
holds(neg(P),S) :- not holds(P,S).   /* Negation by failure */
holds(some(V,P),S) :- sub(V,_,P,P1), holds(P1,S).
```

FIGURE 1. A GOLOG interpreter in CProlog.

5. IMPLEMENTATION AND EXPERIMENTATION

In this section, we discuss an implementation of the GOLOG language in Prolog. We begin by presenting a very simple version of this interpreter. We then show how the elevator example above would be written for this interpreter and some execution traces. We conclude by listing some of the applications currently being investigated in GOLOG.

5.1. An Interpreter

Given that the execution of GOLOG involves a finding a proof in second-order logic, it is perhaps somewhat surprising how easy it is to write a GOLOG interpreter. Figure 1 shows the entire program in CProlog.

The `do` predicate here takes 3 arguments: a GOLOG action expression, and terms standing for the initial and final situations. Normally, a query will be of the form $\texttt{do}(e, \mathbf{s0}, \mathbf{S})$, so that an answer will be a binding for the final situation S. In this implementation, a legal GOLOG action expression e is one of the following:

- $[e_1, \ldots, e_n]$, sequence.
- $?(p)$, where p is a condition (see below).
- $e_1 \# e_2$, nondeterministic choice of e_1 or e_2.
- $\texttt{if}(p, e_1, e_2)$, conditional.

- `star`(e), nondeterministic repetition.
- `while`(p, e), iteration.
- `pi`(v, e) nondeterministic assignment, where v is an atom (standing for a GOLOG variable) and e is a GOLOG action expression that uses v.
- a, where a is the name of a user-declared primitive action or defined procedure (see below).

A condition p in the above is either a fluent or an expression of the form `and`(p_1, p_2), `or`(p_1, p_2), `neg`(p), or `some`(v, p), where v is an atom and p is a condition using v. In evaluating these conditions, the interpreter uses negation as failure to handle `neg`, and consults the user-supplied `holds` predicate to determine which fluents are true.

In this implementation, a GOLOG application (like the elevator, below) is expected to have the following parts:

1. A collection of clauses of the form `primitive_action`(*act*), declaring each primitive action.
2. A collection of clauses of the form **proc**(*name*, *body*) declaring each defined procedure (which can be recursive). The *body* here can be legal GOLOG action expression.
3. A collection of clauses which define the predicate `poss`(*act*, *situation*) for every primitive action and situation. Typically, this requires one clause per action, using a variable to range over all situations.
4. A collection of clauses which define the predicate `holds`(*fluent*, *situation*) for every fluent and situation. Normally, this is done in two parts:
   (a) A collection of clauses defining `holds`(*fluent*, **s0**), characterizing which fluents are true in the initial situation. The clauses need not be atomic, and can involve arbitrary Prolog computation for determining entailments of the initial database. We make the usual Prolog closed world assumption on this database.
   (b) A collection of clauses defining `holds`(*fluent*, `do`(*act*, *situation*)) for every combination of fluent, primitive action, and situation. Typically, this is done with a single clause for each fluent, with variables for the actions and situations. This amounts to writing the successor state axiom for the fluent.

While this interpreter might appear intuitively to be doing the right thing, at least in cases where the closed world assumption (CWA) is made, it turns out to be non-trivial to state precisely in what sense it is correct. On the one hand, we have the specification of *Do* as a formula in second order logic, and on the other, we have the above `do` predicate, characterized by a set of Horn clauses. The exact correspondence between the two depends on a number of factors, and we do not intend to discuss them here. For a formal statement and proof of correctness of this interpreter, the interested reader should consult the companion paper [14].

Given the simplicity of the characterization of the `do` predicate (in first-order Horn clauses), and the complexity of the formula that results from *Do* (in second-order logic), a reasonable question to ask is why we even bother with the latter. The answer is that the definition of `do` is too weak: it is sufficient for finding

a terminating situation (when it exists) given an initial one,[5] but it cannot be used to show non-termination. Consider the program $\delta = [a^*; (x \neq x)?]$. For this program, we have that $\neg\, Do(\delta, s, s')$ is entailed for any s and s'; the do predicate, however, would simply run forever.

On the other hand, the semantics of Prolog is often formulated in terms of minimal models which, in the case of simple logic programs like the above interpreter, have a number of desirable features. Could we not use these ideas instead of second-order quantification to characterize GOLOG program execution? The answer is that we could, but only when the set of axioms characterizing the initial situation S_0 can be made part of a logic program. Our specification of *Do*, on the other hand, is fully general: it does exactly the right thing even when the axioms describing the initial situation contain disjunctions, existential quantifications, and so on. The semantics of logic programs can perhaps be generalized to accommodate such axioms, but it is not clear that the resulting specification would be much simpler than ours.

We emphasize that the above interpreter relies on the standard Prolog CWA that the initial database—the facts true in the initial situation S_0—is complete. This was the case for the logical specification of the elevator example of Section 4. For many applications, this is a reasonable assumption. For many others this is unrealistic, for example in a robotics setting in which the environment is not completely known to the robot. In such cases, a more general GOLOG interpreter is necessary. Such an interpreter might still make use of Prolog's backchaining mechanism to reduce queries about the current situation to queries about the initial situation. In other words, *regression*-based query evaluation (Waldinger [34], Pednault [21], Reiter [23]) can be implemented using Prolog. However, answering the regressed queries in the initial situation would require, in general, the full power of a first order theorem prover.

5.2. *The Elevator Example*

In Figure 2, we present clauses defining the previously discussed elevator example, and in Figure 3, we show some queries to the interpreter for this program.

In the first query, we ask the interpreter to pick a floor and turn off its call button. The answers show that there are only two ways to do this: either turn off floor 3 or turn off floor 5.

In the second query, we ask the interpreter to either turn off a call button or to go to a floor that satisfies the test **next_floor**. Since this predicate has been defined to hold only of those floors whose button is on, this gives us four choices: turn off floor 3 or 5, or go to floor 3 or 5.

In the final query, we call the main elevator controller, **control**, to serve all floors and then park the elevator. There are only two ways of doing this: serve floor 3 then 5 then park, or serve floor 5 then 3 then park. Note that we have not attempted to prune the backtracking to avoid duplicate answers.

[5] This needs to be hedged: the Prolog interpreter is sufficient only if we assume a breadth-first execution strategy. Otherwise, GOLOG programs like *park* in Section 3.1, which terminate according to *Do*, could cause do to run forever.

```
/* Primitive control actions */

primitive_action(turnoff(N)). /* Turn off call button N. */
primitive_action(open).       /* Open the elevator door. */
primitive_action(close).      /* Close the elevator door. */
primitive_action(up(N)).      /* Move the elevator up to floor N.*/
primitive_action(down(N)).    /* Move the elevator down to floor N.*/

/* Definitions of Complex Control Actions  */

proc(go_floor(N), ?(current_floor(N)) # up(N) # down(N)).
proc(serve(N), [go_floor(N), turnoff(N), open, close]).
proc(serve_a_floor, pi(n, [?(next_floor(n)), serve(n)])).
proc(park, if(current_floor(0), open, [down(0), open])).

/* control is the main loop. So long as there is an active call button,
   it serves one floor. When all buttons are off, it parks the elevator.   */

proc(control, [while(some(n, on(n)), serve_a_floor), park]).

/* Preconditions for Primitive Actions */

poss(up(N),S) :- holds(current_floor(M),S), M < N.
poss(down(N),S) :- holds(current_floor(M),S), M > N.
poss(open,S).
poss(close,S).
poss(turnoff(N),S) :- holds(on(N),S).

/* Successor state axioms for primitive fluents. */

holds(current_floor(M),do(E,S)) :- E = up(M) ; E = down(M) ;
              not E = up(N), not E = down(N), holds(current_floor(M),S).

holds(on(M),do(E,S)) :- holds(on(M),S), not E = turnoff(M).

/* Initial situation. Call buttons: 3 and 5. The elevator is at floor 4. */

holds(on(3),s0).   holds(on(5),s0).   holds(current_floor(4),s0).

/* next_floor(N) determines which of the active call buttons should be served
   next. Here, we simply choose an arbitrary active call button.   */

holds(next_floor(N),S) :- holds(on(N),S).
```

FIGURE 2. The elevator controller.

5.3. Experimentation

The actual implementation of GOLOG we have been using at the University of Toronto is in Quintus Prolog and incorporates a number of additional features for debugging and for efficiency beyond those of the simple interpreter presented here.

For example, one serious limitation of the style of interpreter presented here is the following: determining if some condition (like **current_floor(0)**) holds in a

```
?- do(pi(n,[?(on(n)),turnoff(n)]),s0,S).

S = do(turnoff(3),s0) ;

S = do(turnoff(5),s0) ;

no

- ----------------------------------------

?- do(pi(n, turnoff(n) # ([?(next_floor(n)),go_floor(n)])),s0,S).

S = do(turnoff(3),s0) ;

S = do(turnoff(5),s0) ;

S = do(down(3),s0) ;

S = do(up(5),s0) ;

no

- ----------------------------------------

?- do(control,s0,S).

S = do(open,do(down(0),do(close,do(open,do(turnoff(5),do(up(5),do(close,
do(open,do(turnoff(3),do(down(3),s0)))))))))) ;

S = do(open,do(down(0),do(close,do(open,do(turnoff(3),do(down(3),do(close,
do(open,do(turnoff(5),do(up(5),s0)))))))))) ;

S = do(open,do(down(0),do(close,do(open,do(turnoff(5),do(up(5),do(close,
do(open,do(turnoff(3),do(down(3),s0)))))))))) ;

S = do(open,do(down(0),do(close,do(open,do(turnoff(3),do(down(3),do(close,
do(open,do(turnoff(5),do(up(5),s0)))))))))) ;

no
```

FIGURE 3. Running the elevator program.

situation involves looking at what actions led to that situation, and unwinding these actions all the way back to the initial situation. This process is called *regression* in the AI planning literature. Doing this repeatedly with very long sequences of actions can take considerable time. Moreover, the Prolog terms representing situations that are far removed from the initial situation end up being gigantic.

However, it is possible in many cases to *progress* the initial database to handle this (Lin and Reiter [16, 18]). The idea is that the interpreter periodically "rolls the initial database forward" in response to the actions generated thus far during the evaluation of the program. This progressed database becomes the new initial database for the purposes of the continuing evaluation of the program. In this way,

the interpreter maintains a database of just the current value of all fluents, and the distance from the initial situation is no longer a problem.

To evaluate our interpreter and the entire GOLOG framework, we have been experimenting with various types of applications. The most advanced involves a robotics application—mail delivery in an office environment [9]. The high-level controller of the robot programmed in GOLOG is interfaced to an existing robotics package that supports path planning and local navigation. The system currently works in simulation mode; experiments with a real robot have begun in collaboration with the robotics group at the University of Bonn.

Another application involves tools for home banking [27]. In this case, a number of software agents written in GOLOG handle various parts of the banking process (responding to buttons on an ATM terminal, managing the accounts at a bank, monitoring account levels for a user, etc.), communicating over TCP/IP.

CONGOLOG, a version of the language supporting concurrency (including interrupts, priorities, and support for exogenous actions) is also being implemented, and experiments with various applications (meeting scheduling, multi-elevator coordination) are under way.

6. DISCUSSION

GOLOG is designed as a logic programming language for dynamic domains. As its full name (alGOl in LOGic) implies, GOLOG attempts to blend ALGOL programming style into logic. It borrows from ALGOL many well-known, and well-studied programming constructs such as sequence, conditionals, recursive procedures and loops.

However, unlike ALGOL and most other conventional programming languages, programs in GOLOG decompose into primitives that in most cases refer to actions in the external world (e.g., picking up an object or telling something to another agent), as opposed to commands which merely change machine states (e.g., assignments to registers). Furthermore, these primitives are formulated by axioms in first-order logic so their effects can be formally reasoned about. This feature of GOLOG supports the specification of dynamic systems at the right level of abstraction.

More importantly, GOLOG programs are evaluated with a theorem prover. The user supplies precondition axioms, one per action, successor state axioms, one per fluent, a specification of the initial situation of the world, and a GOLOG program specifying the behavior of the agents in the system. Executing a program amounts to finding a ground situation term σ such that

$$Axioms \models Do(program, S_0, \sigma).$$

This is done by trying to prove

$$Axioms \models (\exists s)\, Do(program, S_0, s),$$

and if a (constructive) proof is obtained, such a ground term

$$do(a_n, \ldots do(a_2, do(a_1, S_0)) \ldots)$$

is obtained as a binding for the variable s. Then the sequence of actions $[a_1, a_2, \ldots, a_n]$ is sent to the primitive action execution module. This looks very

much like logic programming languages such as Prolog. However, unlike such general purpose logic programming languages, GOLOG is designed specifically for specifying agents' behaviors and for modeling dynamic systems. In particular, in GOLOG, actions play a fundamental role.

There is a body of literature related to the GOLOG project:

1. Dixon's Amala [3]. Amala is a programming language in a conventional imperative style. It is designed after the observation that the semantics of embedded programs should reflect the assumptions about the environment as directly as possible. This is similar to our concern that language primitives should be user-defined, at a high level of abstraction. However, while GOLOG requires these primitives be formally specified within the language, Amala does not. One consequence of this is that programs in GOLOG can be executed by a theorem power, but not those in Amala.
2. Classical AI planning work (Green [6] and Fikes and Nilsson [4]). Like classical AI planning, GOLOG requires primitives and their effects to be formally specified. The major difference is that GOLOG focuses on high-level programming rather than plan synthesis at run-time. But sketchy plans are allowed; nondeterminism can be used to infer the missing details. In our elevator example, it was left to the GOLOG interpreter to find a legal sequence of actions to serve all active call buttons. But we can go well beyond this. As an extreme case, the program

   while $\neg$ *Goal* **do** $(\pi a)[\mathit{Appropriate}(a)?; a]$ **endWhile**,

   repeatedly selects an appropriate action and performs it until some goal is achieved. Finding a legal sequence of actions in this case is simply a reformulation of the planning problem.
3. Situated automata [26]. GOLOG shares with situated automata the same philosophy of designing agents using a high level language, and then compiling the high-level programs into low-level ones that can be immediately executed. In the framework considered here, the low-level programs are simply sequences of primitive actions. In [13], we also consider cases involving sensing (see below) where no such sequence exists, and it is necessary to compile to low-level programs containing loops and conditionals.
4. Shoham's AGENT-0 programming language [31]. This includes a model of commitments and capabilities, and has simple communication acts built-in; its agents all have a generic rule-based architecture; there is also a global clock and all beliefs are about time-stamped propositions. However, there is no automatic maintenance of the agents beliefs based on a specification of primitive actions as in GOLOG and only a few types of complex actions are handled; there also seems to be less emphasis on having a complete formal specification of the system.

   A number of other groups are also developing formalisms for the specification of artificial agents. See [35] for a detailed survey of this research.
5. Transaction logic (Bonner and Kifer [2]). This is a new logic for defining complex database transactions, and like GOLOG provides a rich repertoire of operators for defining new transactions in terms of old. These include sequence, nondeterministic choice, conditionals and iteration. The Bonner-Kifer approach focuses on the definition of complex transactions in terms of

elementary updates. On the assumption that these elementary updates successfully address the frame problem, any complex update defined in terms of these elementary ones will inherit a correct solution to the frame problem. Unfortunately, Bonner and Kifer do not address the frame problem for these elementary updates; this task is left to the person specifying the database.

6. The strategies of McCarthy and Hayes [20]. This is a surprisingly early proposal for representing complex actions (called *strategies*) in the situation calculus. McCarthy and Hayes even appeal to an Algol-like language for representing their strategies, and they include a mechanism for returning symbolic execution traces, as sequences of actions, of these strategies. Moreover, they sketch a method for proving properties of strategies. While McCarthy and Hayes provide no formal development of their proposal, it nevertheless anticipates much of the spirit and technical content of our GOLOG project.

The version of GOLOG presented here omits some important considerations. The following is a partial list:

1. Sensing and knowledge. When modeling an autonomous agent, it is necessary to consider the agent's *perceptual actions*, e.g., acts of seeing, hearing, etc. Unlike ordinary actions that affect the environment, perceptual actions affect an agent's mental state, i.e., its state of *knowledge*. Scherl and Levesque [28] provide a situation calculus account of knowledge, and within this setting, show how to solve the frame problem for perceptual actions.
2. Sensing and knowing how. In the presence of sensing actions, the method described above for executing a GOLOG program is no longer adequate. For example, suppose the sensing action SENSE_P reads the truth value of P, and the primitives a and b are always possible. Then the following program $\mathscr{P}$ is perfectly reasonable:

   SENSE_P; **if** P **then** a **else** b **endIf**

   and should be executable with respect to any initial situation. However, it is not the case that

   $$Axioms \models Do(\mathscr{P}, S_0, \sigma)$$

   for any ground situation term σ. That is, at compile time, the agent does not know the truth value of P and therefore does not know the exact sequence of primitive actions that corresponds to the execution of this program. We have considered several possible solutions to this problem. See [11, 13].
3. Exogenous actions. We have assumed that all events of importance are under the agent's control. That is why, in the elevator example, we did not include a primitive action *turnon*(n), meaning push call button n. Such an action can occur at any time, and is not under the elevator's control. *turnon*(n) is an example of an *exogenous* action. Other such examples are actions under nature's control—it starts to rain, a falling ball bounces on reaching the floor. In writing an elevator or robot controller, one would not include exogenous actions as part of the program, because the robot is in no position to cause such actions to happen.

4. Concurrency and reactivity. Once we allow for exogenous events, it becomes very useful to write programs which monitor certain conditions, and take appropriate actions when they become true. For example, in the middle of serving a floor, smoke might be detected by the elevator, in which case, normal operation should be suspended, and an alarm should be sounded until the alarm is reset. As mentioned earlier, we are investigating a concurrent version of GOLOG where a number of complex actions of this sort can be executed concurrently (at different priorities). We believe that this form of concurrency allows a much more natural specification of controllers that need to quickly react to their environment while following predetermined plans.
5. Continuous processes. It is widely believed that, by virtue of its reliance on discrete situations, the situation calculus cannot represent continuous processes and their evolution in time, like an object falling under the influence of gravity. However, as shown by Pinto [22] and also by Ternovskaia [33], one can view a process as a fluent—*falling*(s)—which becomes true at the time t that the instantaneous action *start_falling*(t) occurs, and becomes false at the time t of occurrence of the instantaneous action *end_falling*(t). One can then write axioms that describe the evolution in time of the falling object. Reiter [25] gives a situation calculus account of such natural events whose behaviors are described by known laws of physics. This means that one can write GOLOG simulators of such dynamical systems [8]. Moreover, although we have not yet explored this possibility, the GOLOG programmer can now write robot controllers which allow a robot to exploit such naturally occurring exogenous events in its environment.

7. CONCLUSIONS

GOLOG is a logic programming language for implementing applications in dynamic domains like robotics, process control, intelligent software agents, discrete event simulation, etc. Its basis is a formal theory of actions specified in an extended version of the situation calculus.

GOLOG has a number of novel features, both as a programming language, and as an implementation tool for dynamic modeling.

1. Formally, a GOLOG program is a macro which expands during the evaluation of the program to a (usually second order) sentence in the situation calculus. This sentence mentions only the primitive, user defined actions and fluents. The theorem proving task in the evaluation of the program is to prove this sentence relative to a background axiomatization consisting of the foundational axioms of the situation calculus, the action precondition axioms for the primitive actions, the successor state axioms for the fluents, and the axioms describing the initial situation.
2. GOLOG programs are normally evaluated to obtain a binding for the existentially quantified situation variable in the top-level call $(\exists s)Do($*program*$, S_0, s)$. The binding so obtained by a successful proof is a symbolic trace of the program's execution, and denotes that sequence of actions which is to be performed in the external world. At this point, the entire GOLOG computa-

tion has been performed off-line. To effect an actual change in the world, this program trace must be passed to an execution module which knows how to physically perform the sequence of primitive actions in the trace.

3. Because a GOLOG program macro-expands to a situation calculus sentence, we can prove properties of this program (termination, correctness, etc.) directly within the situation calculus.
4. Unlike conventional programming languages, whose primitive instruction set is fixed in advance (assignments to variables, pointer-changing, etc.), and whose primitive function and predicate set is also predefined (values and types of program variables, etc.), GOLOG primitive actions and fluents are user defined by action precondition and successor state axioms. In the simulation of dynamic systems, this facility allows the programmer to specify his primitives in accordance with the naturally occurring events in the world he is modeling. This, in turn, allows programs to be written at a very high level of abstraction, without concern for how the system's primitive architecture is actually implemented.
5. The GOLOG programmer can define complex action *schemas*—advice to a robot about how to achieve certain effects—*without specifying in detail how to perform these actions*. It becomes the theorem prover's responsibility to figure out one or more detailed *executable* sequences of primitive actions which will achieve the desired effects.

$$\textbf{while}[(\exists \mathit{block})\mathit{ontable}(\mathit{block})]\ \textbf{do}\ (\pi\, b)\mathit{remove}(b)\ \textbf{endWhile},$$

is such an action schema; it does not specify any particular sequence in which the blocks are to be removed. Similarly, the elevator program does not specify in which order the floors are to be served. On this view of describing complex behaviors, the GOLOG programmer specifies a skeleton plan; the evaluator uses deduction, in the context of a specific initial world situation, to fill in the details. Thus GOLOG allows the programmer to strike a compromise between the often computationally infeasible classical planning task, in which a plan must be deduced entirely from scratch, and detailed programming, in which every little step must be specified.

There are several limitations to the version of GOLOG that has been presented here. The implementation only works with completely known initial situations. Adapting GOLOG to work with non-Prolog theories in the initial situation will require some effort (see [16] for ideas on this). Handling sensing actions requires the system's knowledge state to be modeled explicitly [28] and complicates the representation and updating of the world model. Exogenous events also affect the picture as the system may no longer know what the actual history is. In many domains, it is also necessary to deal with sensor noise and "control error" (see [1] for some initial results).

We are also developing an extended version of the language called CONGOLOG that supports concurrent processes, interrupts, and differing priorities on processes (based on an interleaving semantics for concurrent processes) [12]. Techniques for representing and reasoning about continuous processes (e.g., filling a bathtub) are also under investigation [25]. Finally, work is also in progress on a multi-agent distributed version of CONGOLOG for agent-oriented programming

applications, which will support distinct world models for each agent and a library of high-level communication actions [10]. Notions like ability, goals, commitments, and rational choice become important in such domains and we are extending our model to deal with them [30].

REFERENCES

1. Bacchus, F., Halpern, J. Y., and Levesque, H. J., Reasoning about Noisy Sensors in the Situation Calculus, in: C. S. Mellish (ed.), *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, Montréal, Aug. 1995, Morgan Kaufmann Publishing, pp. 1933–1940.
2. Bonner, A. and Kifer, M., An Overview of Transaction Logic, *Theoretical Computer Science* 133:205–265 (1994).
3. Dixon, M., *Embedded Computation and the Semantics of Programs*, Ph.D. Thesis, Department of Computer Science, Stanford University, Stanford, CA, 1991. Also appeared as Xerox PARC Technical Report SSL-91-1.
4. Fikes, R. E. and Nilsson, N. J., STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving, *Artificial Intelligence* 2(3/4):189–208 (1971).
5. Goldblatt, R., *Logics of Time and Computation, CSLI Lecture Notes No.* 7, Center for the Study of Language and Information, Stanford University, Stanford, CA, 2nd edition, 1987.
6. Green, C. C., Theorem Proving by Resolution as a Basis for Question-Answering Systems, in: B. Meltzer and D. Michie (eds.), *Machine Intelligence 4*, American Elsevier, New York, 1969, pp. 183–205.
7. Haas, A. R., The Case for Domain-Specific Frame Axioms, in: F. M. Brown (ed.), *The Frame Problem in Artificial Intelligence: Proceedings of the 1987 Workshop*, Lawrence, KA, April 1987, Morgan Kaufmann Publishing, pp. 343–348.
8. Kelley, T. G., Modeling Complex Systems in the Situation Calculus: A Case Study Using the Dagstuhl Steam Boiler Problem, in: L. C. Aiello, J. Doyle, and S. C. Shapiro (eds.), *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR'96)*, Morgan Kaufmann Publishers, San Francisco, CA, pp. 26–37.
9. Lespérance, Y., Levesque, H. J., Lin, F., Marcu, D., Reiter, R., and Scherl, R. B., A Logical Approach to High-Level Robot Programming—A Progress Report, in: B. Kuipers (ed.), *Control of the Physical World by Intelligent Agents, Papers from the 1994 AAAI Fall Symposium*, New Orleans, LA, Nov. 1994, pp. 109–119.
10. Lespérance, Y., Levesque, H. J., Lin, F., Marcu, D., Reiter, R., and Scherl, R. B., Foundations of a Logical Approach to Agent Programming, in: M. Wooldridge, J. P. Müller, and M. Tambe (eds.), *Intelligent Agents, Volume II, Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages (ATAL-95), Lecture Notes in Artificial Intelligence*, Springer-Verlag, 1996, pp. 331–346.
11. Lespérance, Y., Levesque, H. J., Lin, F., and Scherl, R. B., Ability and Knowing How in the Situation Calculus, Unpublished manuscript, 1997.
12. Levesque, H. J., Concurrency in the Situation Calculus, Unpublished manuscript, 1997.
13. Levesque, H. J., What Is Planning in the Presence of Sensing?, in: *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR, Aug. 4–8, 1996, AAAI Press/MIT Press, pp. 1139–1146.
14. Levesque, H. J., Lin, F., and Reiter, R., Defining Complex Actions in the Situation Calculus, Technical Report, Department of Computer Science, University of Toronto, 1997, to appear.
15. Lin, F., Embracing Causality in Specifying the Indirect Effects of Actions, in: C. S. Mellish (ed.), *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, Montréal, Aug. 1995, Morgan Kaufmann Publishing, pp. 1933–1940.
16. Lin, F. and Reiter, R., How to Progress a Database (and Why) I. Logical Foundations, in: J. Doyle, E. Sandewall, and P. Torasso (eds.), *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourth International Conference*, Bonn, Germany, 1994, Morgan Kaufmann Publishing, pp. 425–436.

17. Lin, F. and Reiter, R., State Constraints Revisited, *Journal of Logic and Computation* 4(5):655–678 (1994).
18. Lin, F. and Reiter, R., How to Progress a Database II: The STRIPS Connection, in: *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, Montreal, Aug. 20–25, 1995, pp. 2001–2007.
19. Manna, Z. and Waldinger, R., How to Clear a Bock: A Theory of Plans, *Journal of Automated Reasoning* 3:343–377 (1987).
20. McCarthy, J. and Hayes, P., Some Philosophical Problems from the Standpoint of Artificial Intelligence, in: B. Meltzer and D. Michie (eds.), *Machine Intelligence 4*, Edinburgh University Press, Edinburgh, Scotland, 1969, pp. 463–502.
21. Pednault, Edwin P. D., ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus, in: R. J. Brachman, H. J. Levesque, and R. Reiter (eds.), *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, ON, May 1989, Morgan Kaufmann Publishing, pp. 324–332.
22. Pinto, J. A., *Temporal Reasoning in the Situation Calculus*, Ph.D. Thesis, Department of Computer Science, University of Toronto, Toronto, ON, Feb. 1994. Available as Technical Report KRR-TR-94-1.
23. Reiter, R., The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression, in: V. Lifschitz (ed.), *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, Academic Press, San Diego, CA, 1991, pp. 359–380.
24. Reiter, R., Proving Properties of States in the Situation Calculus, *Artificial Intelligence* 337–351 (Dec. 1993).
25. Reiter, R., Natural Actions, Concurrency and Continuous Time in the Situation Calculus, in: L. C. Aiello, J. Doyle, and S. C. Shapiro (eds.), *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR'96)*, 1996, Morgan Kaufmann Publishers, San Francisco, CA, pp. 2–13.
26. Rosenschein, S. J. and Kaelbling, L. P., The Synthesis of Digital Machines with Provable Epistemic Properties, in: J. Y. Halpern (ed.), *Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning about Knowledge*, Morgan Kaufmann Publishers, Monterey, CA, 1986, pp. 83–98.
27. Ruman, S., Golog as an Agent-Programming Language: Experiments in Developing Banking Applications, Master's Thesis, Department of Computer Science, University of Toronto, Toronto, ON, 1996.
28. Scherl, R. B. and Levesque, H. J., The Frame Problem and Knowledge-Producing Actions, in: *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Washington, DC, July 1993, AAAI Press/MIT Press, pp. 689–695.
29. Schubert, L. K., Monotonic Solution to the Frame Problem in the Situation Calculus: An Efficient Method for Worlds with Fully Specified Actions, in: H. E. Kyberg, R. P. Loui, and G. N. Carlson (eds.), *Knowledge Representation and Defeasible Reasoning*, Kluwer Academic Press, Boston, MA, 1990, pp. 23–67.
30. Shapiro, S., Lespérance, Y., and Levesque, H. J., Goals and Rational Action in the Situation Calculus—A Preliminary Report, in: *Working Notes of the AAAI Fall Symposium on Rational Agency: Concepts, Theories, Models, and Applications*, Cambridge, MA, Nov. 1995, pp. 117–122.
31. Shoham, Y., Agent-Oriented Programming, *Artificial Intelligence* 60(1):51–92 (1993).
32. Stoy, J. E., *Denotational Semantics*, MIT Press, 1977.
33. Ternovskaia, E., Interval Situation Calculus, in: *Proceedings of ECAI'94 Workshop W5 on Logic and Change*, Amsterdam, Aug. 8–12, 1994, pp. 153–164.
34. Waldinger, R., Achieving Several Goals Simultaneously, in: E. Elcock and D. Michie (eds.), *Machine Intelligence 8*, Ellis Horwood, Edinburgh, Scotland, 1977, pp. 94–136.
35. Wooldrige, M. J. and Jennings, N. R., Intelligent Agents: Theory and Practice, *Knowledge Engineering Review* 10(2) (1995).

Proceedings of IJCAI 1995: 1985-1993

Embracing Causality in Specifying the Indirect Effects of Actions*

Fangzhen Lin
Department of Computer Science
University of Toronto
Toronto, Canada M5S 1A4
Email:fl@cs.toronto.edu

Abstract

This paper considers the problem of specifying the effects of actions in the situation calculus using domain constraints. We argue that normal state constraints that refer to only the truth values of fluents are not strong enough for this purpose, and that a notion of causation needs to be employed explicitly. Technically, we introduce a new ternary predicate *Caused* into the situation calculus: *Caused(p,v,s)* if the proposition *p* is caused (by something unspecified) to have the truth value *v* in the state *s*. Using this predicate, we can represent not only *action-triggered* causal statements such as that the action *load* causes the gun to be loaded, but also *fluent-triggered* ones such as that the fact that the switch is in the up position causes the lamp to be on. The former is convenient for representing the direct effects of actions, and the latter the indirect effects.

1 Introduction

We consider the problem of formalizing the effects of actions in the situation calculus [McCarthy and Hayes, 1969]. To motivate the needs for a notion of causality in this endeavor, let us consider the following problem.

Imagine a suitcase with two locks and a spring loaded mechanism which will open the suitcase when both of the locks are in the up position. Apparently, because of the spring loaded mechanism, if an action changes the statuses of the locks, then this action may also cause, as an indirect effect, the suitcase to open.

The problem of how to describe the spring loaded mechanism by a constraint and use it to derive the indirect effects of actions is an instance of the *ramification* problem [Finger, 1986], which has been recognized as one of the central problems in reasoning about the effects of actions. In the situation calculus, the constraint that this spring loaded mechanism gives rise to can be represented as follows:[1]

$$up(L1, s) \wedge up(L2, s) \supset open(s). \quad (1)$$

Although summarizing concisely the relationship among the truth values of the three relevant propositions at any particular instance of time, this constraint is too weak to describe the indirect effects of actions. For instance, suppose that initially the suitcase is closed, the first lock in the down position, and the second lock in the up position. Suppose an action is then performed to turn up the first, lock. Then this constraint is ambiguous about what will happen next. According to it, either the suitcase may spring open or the second lock may get turned down. Although we have the intuition that the former is what will happen, this constraint is not strong enough to enforce that because there is a different mechanism that will yield a logically equivalent constraint. For instance, a mechanism that will turn down the second lock when the suitcase is closed and the first lock is up will yield the following logically equivalent one:

$$up(L1, s) \wedge \neg open(s) \supset \neg up(L2, s).$$

So to faithfully represent the ramification of the spring loaded mechanism on the effects of actions, something stronger than the constraint (1) is needed. The proposal of this paper is to appeal to causality: (through the spring loaded mechanism) the fact that both of the locks are in the up position *causes* the suitcase to open. The goal of this paper is then to make precise causal statements like this, and show how they can be used effectively to describe the effects of actions.

This paper is organized as follows. Section 2 briefly describes the version of the situation calculus used in this paper. Section 3 shows in detail how the suitcase example is solved using causality. Section 4 generalizes the method used in section 3 to a general class of theories. Section 5 applies the method of section 4 to an example to further illustrate some of the fine points of our approach. To simplify our presentation, we shall defer discussions of related work, in particular that of [Lifschitz, 1987] and that of [Haugh, 1987], to section 6. Finally section 7 concludes this paper.

"This research was supported by grants from the Government of Canada Institute for Robotics and Intelligent Systems, and from the National Science and Engineering Research Council of Canada.

[1] We use the convention that in displayed formulas, free variables are implicitly universally quantified. See the next section for a precise description of the version of the situation calculus used in this paper.

2 The Situation Calculus

We have already seen the situation calculus in action in the introduction section. We hope the notations used there are familiar and/or intuitive enough for everyone to follow. In any event, they are defined in this section.

The language of the situation calculus is a many sorted first order one. We assume the following sorts: *situation* for situations, *action* for actions, *fluent* for propositions! fluents, *truth-value* for truth values *true* and *false,* and *object* for everything else.

We use the following domain independent predicates and functions:

- The binary function *do* - for any action *a* and any situation s, *do(a,s)* is the situation resulting from performing *a* in *s.*
- The binary predicate *Holds* - for any propositional fluent *p* and any situation s, *Holds(p, s)* is true if p holds in *s.*
- The binary predicate *Poss* - for any action *a* and any situation s, *Poss(a,s)* is true if *a* is possible (executable) in s.
- The ternary predicate *Caused* - for any fluent p, any truth value v, and any situation s, *Caused(p, v, s)* is true if the fluent p is caused (by something unspecified) to have the truth value *v* in the situation *s.*

Notice that in the introduction, we wrote, for instance, $up(L1, s)$ instead of $Holds(up(L1), s)$. We consider the former to be a shorthand for the latter. We shall continue to do so in an effort to improve the readability of our formulas. Formally, if F is a fluent name of arity $object^n \rightarrow fluent$, then we define the expression $F(t_1, ..., t_n, t_s)$ to be a shorthand for the formula $Holds(F(t_1, ..., t_n), t_s)$, where $t_1, \cdots, t_n$ are terms of sort *object*, and t_s a term of sort *situation*.

We assume that all theories in this paper will include the following basic axioms:

- For the predicate *Caused*, the following two basic axioms:

$$Caused(p, true, s) \supset Holds(p, s), \quad (2)$$
$$Caused(p, false, s) \supset \neg Holds(p, s). \quad (3)$$

- For the truth values, the following unique names and domain closure axiom:

$$true \neq false \wedge (\forall v)(v = true \vee v = false). \quad (4)$$

- The unique names assumptions for fluent and action names (we assume there are only finitely many of them). Specifically, if F_1,..., F_n are all the fluent names, then we have:

$$F_i(\vec{x}) \neq F_j(\vec{y}), \ i \text{ and } j \text{ are different},$$
$$F_i(\vec{x}) = F_i(\vec{y}) \supset \vec{x} = \vec{y}.$$

  Similarly for action names.

- The *foundational axioms* in [Lin and Reiter, 1994] for the *discrete* situation calculus. These axioms characterize the structure of the space of situations. For the purpose of this paper, it is enough to mention that they include the following unique names axioms for situations:

$$s \neq do(a, s),$$
$$do(a, s) = do(a', s') \supset (a = a' \wedge s = s').$$

3 An Example

Consider again the suitcase example described in the introduction section. Suppose that *flip(x)* is an action that flips the status of the lock *x.* Its direct effect can be described by the following axioms:[2]

$$Poss(flip(x), s) \supset$$
$$up(x, s) \supset Caused(up(x), false, do(flip(x), s)), \quad (5)$$
$$Poss(flip(x), s) \supset$$
$$\neg up(x, s) \supset Caused(up(x), true, do(flip(x), s)). \quad (6)$$

To perform *flip(x),* the object *x* must be a lock:

$$Poss(flip(x), s) \supset lock(x), \quad (7)$$

where *lock* is a unary predicate symbol. We assume that LI and *L2* are two distinct locks:

$$L1 \neq L2 \wedge lock(L1) \wedge lock(L2) \quad (8)$$

The spring loaded mechanism is now represented by the following causal rule:

$$up(L1, s) \wedge up(L2, s) \supset Caused(open, true, s). \quad (9)$$

Notice that this causal rule, together with the basic axiom (2) about causality, entail the state constraint (1). Notice also that the physical, spring loaded mechanism behind the causal rule has been abstracted away. For all we care, it may just as well be that the device is not made of spring, but of bombs that will blow open the suitcase each time the two locks are in the up position. It then seems natural to say that the fluent *open* is caused to be true by the fact that the two locks are both in the up position. This is an instance of what we have called *fluent-triggered* causal statements in the abstract. In comparison, causal statements like the effect axioms (5) and (6) are *action-triggered.*

This finishes describing our starting theory for the domain. To describe fully the effects of the actions, we need to add suitable frame axioms. Our version of the frame axioms is: Unless caused otherwise, a fluent's truth value will persist:

$$Poss(a, s) \supset \{\neg(\exists v) Caused(p, v, do(a, s)) \supset$$
$$[Holds(p, do(a, s)) \equiv Holds(p, s)]\}. \quad (10)$$

For this frame axiom to make sense, obviously, we need to minimize the predicate *Caused.* We shall circumscribe the predicate *Caused* in the set of axioms introduced so far. We shall do so with all the other predicates *(Poss* and *Holds)* fixed, because we want to condition

[2]Recall that, for instance, *up(x, s)* is defined to be a shorthand for *Holds(up{x), s).*

Caused on them. It is easy to see that the circumscription [McCarthy, 1986] yields the following *causation axioms:*

$$Caused(open, v, s) \equiv$$
$$v = true \wedge up(L1, s) \wedge up(L2, s), \quad (11)$$
$$Caused(up(x), v, s) \equiv$$
$$\{v = true \wedge (\exists s')[s = do(flip(x), s') \wedge Poss(flip(x), s') \wedge \neg up(x, s')] \vee \quad (12)$$
$$v = false \wedge (\exists s')[s = do(flip(x), s') \wedge Poss(flip(x), s') \wedge up(x, s')]\}.$$

Notice that these axioms entail the two direct effect axioms (5, 6) and the causal rule (9). In fact, they are obtained by applying the Clark completion [1978] to the predicate *Caused* in these clauses. As we shall sec in the next section, this will be true for a general class of theories.

Having computed the causal relation, we now use the frame axiom (10) to compute the effects of actions. It is easy to see that from the frame axiom (10) and the two basic axioms (2, 3) about causality, we can infer the following pseudo *successor state axiom* [Reiter, 1991]:

$$Poss(a, s) \supset \{Holds(p, do(a, s)) \equiv$$
$$Caused(p, true, do(a, s)) \vee \quad (13)$$
$$Holds(p, s) \wedge \neg Caused(p, false, do(a, s))\}$$

From this axiom and the causation axiom (12) for the fluent *up,* we then obtain the following real successor state axiom for the fluent *up:*

$$Poss(a, s) \supset \{up(x, do(a, s)) \equiv$$
$$(a = flip(x) \wedge \neg up(x, s)) \vee (up(x, s) \wedge a \neq flip(x))\}$$

Similarly for the fluent *open,* we have

Poss(a,s)) {open(do(a, s)) =
[up(L1,do(a,s)) A up(L2, do{a, s))] V open(s)}.

Now from this axiom, first eliminating up(Ll,do(a,s)) and *up(L2,do(a,s))* using the successor state axiom for *up,* then using the unique names axioms for actions, and the constraint (1) which, as we pointed out earlier, is a consequence of our axioms, we can deduce the following successor state axiom for the fluent *open:*

$$Poss(a, s) \supset \{open(do(a, s)) \equiv$$
$$a = flip(L1) \wedge \neg up(L1, s) \wedge up(L2, s) \vee$$
$$a = flip(L2) \wedge \neg up(L2, s) \wedge up(L1, s) \vee$$
$$open(s)\}.$$

Obtaining these successor state axioms solves the frame and the ramification problems for the suitcase example.

Finally it remains to be shown how *Poss* is computed. This is an instance of the *qualification problem* [McCarthy, 1977], and the standard default assumption one uses is that an action is always executable unless explicitly ruled out by the theory. In other words, we want to maximize the predicate *Poss* in the theory we have so far, including the frame axiom and the causality axioms.

To determine what policy we shall use in maximizing *Poss,* we first have to be clear what we expect from the maximization. For us, after the maximization, we want, for each action *A(x),* an *action precondition axiom* [Reiter, 1991] of the following form:

$$Poss(A(\vec{x}), s) \equiv \Phi(\vec{x}, s),$$

where Φ(x, *s)* is a formula which refers to only the truth values of the fluents in s. In other words, the preconditions of an action should be expressed in terms of the truth values of the fluents in the current state. This means that we should maximize *Poss(a, s)* with *Holds(p, s)* fixed while allowing *Caused* and *Holds(p,s')* ^ ≠ *s'* to vary. Model-theoretically, this means that an intended model is one in which the extension of *Poss(a, s)* cannot be made larger by changing the interpretation of the predicate *Caused* and the truth values of the fluents at states other than *s.*

It turns out there is a procedure based on the Clark completion for computing the maximization. The procedure is outlined in the next section. For this example, it yields the dark completion of (7):

$$Poss(flip(x), s) \equiv lock(x),$$

Having this action precondition axiom solves the qualification problem for the suitcase example.

This concludes our solution to the suitcase problem. The method used in this example can be generalized to a general class of theories, as we shall see in the next section.

4 The Method

The procedure we followed in solving the suitcase problem can be summarized as follows:

1. Start with a theory *T* that includes all the effect axioms and state constraints.
2. Minimize *Caused* in *T.* Let T" be the resulting theory.
3. Add to *V* the frame axiom (10). Let *T"* be the resulting theory.[3]
4. Maximize *Poss* in *T"* to obtain the final action theory.

Clearly, the tractability of this approach depends crucially on the form of the initial theory *T.* In the following we consider a special class of theories for which the Clark completion is enough to compute the result of minimizing *Caused* and that of maximizing *Poss.*

Before we describe this special class of theories, we first define a terminology that will be used throughout this section. Let *s* be a situation variable. We call a formula *Φ($)* a *simple state formula about s* if Φ does not mention *Poss, Caused,* or any situation term other than possibly the variable *s.* For example, $lock(L1) \wedge L1 \neq L2$ (mentions no situation term) and *up(x, s)* A *a = flip(x)*

[3] Alternatively, instead of simply adding (10), one can minimize the formula:

->(Holds(p,s) = Holds(p,do(a,s)))

in the theory *T'.* See [Lin and Reiter, 1994].

(mentions only s) are simple state formulas about s, but $up(x, do(a, s))$ and $up(x, s')$ are not since the former mentions the compound situation term $do(a, s)$, and the latter mentions the situation variable s' which is different from s.

Step 1. For each action $A(\vec{x})$, formalize the *direct effects* of A by axioms of the form:

$$Poss(A(\vec{x}), s) \supset \Phi(s) \supset Caused(F(\vec{y}), v, do(A(\vec{x}), s)), \quad (14)$$

where F is a fluent name, and $\Phi(s)$ is a simple state formula about s. For instance, the direct effect axiom (5) for the action $flip$ can be rewritten as;

$$Poss(flip(x), s) \supset x = y \wedge v = false \wedge up(y, s) \supset Caused(up(y), v, do(flip(x), s)).$$

Step 2. For each action $A(\vec{x})$, formalize the *explicit qualifications* of A by axioms of the form:

$$Poss(A(\vec{x}), s) \supset \Phi(s) \quad (15)$$

where $\Phi(s)$ is a simple state formula about s. For instance, the qualification axiom (7) for the action $flip$ is of this form.

Step 3. Formalize all *causal rules*, the constraints that will be used to derive the indirect effects of actions, by axioms of the form:

$$\Phi(s) \wedge Caused(p_1, v_1, s) \wedge \cdots \wedge Caused(p_n, v_n, s) \supset Caused(F(\vec{x}), v, s), \quad (16)$$

where *F* is a fluent, and Φ(s) a simple state formula about *s.* The causal rule (9) in the suitcase example can be rewritten straightforwardly as axioms of this form. For an example of a causal rule in which the predicate *Caused* also appears in the left hand side of the implication, suppose that we add a new fluent called *down* to our suitcase example so that *down(x, s)* is true if the lock *x* is in the down position. Clearly this new fluent is an antonym of up, so one of them is caused to be true iff the other is caused to be false:

C aused(up(x), true, s) = Caused(down(x), false, s),
Caused(up(x), false, s) = Caused(down(x), true, s).

Notice that adding the fluent *down* and the above causal rules to the suitcase example will not affect the causation axiom for the fluent *up,* although the minimization will no longer be computable by the Clark completion (see Step 5).

Step 4. Formalize all other domain knowledge by axioms of the form:

$$(\forall s)C(s), \quad (17)$$

where *C(s)* is a simple state formula about *s.* In the suitcase example, the axiom (8) will be included in this step. State constraints that are used to derive implicit action qualifications will also be included here. For instance, in the blocks world, the state constraint $(\forall x)\neg on(x, x, s)$, which says that a block cannot be on top of itself, will be included in this step. This constraint implies that the action *stack(x, y),* which puts *x* on *y,* will not be executable when *x = y.* For more examples of such *qualification state constraints,* see [Ginsberg and Smith, 1988; Lin and Reiter, 1994].

The above four steps yield a starting theory *T* about the dynamic domain of interest. Of course, in addition to the axioms given in steps 1 to 4, *T* also contains the axioms given in section 2. The remaining steps apply nonmonotonic logic to *T* to solve the frame, the ramification, and the qualification problems.

Step 5. The goal of this step is to compute for each fluent *F* a *causation axiom* of the following form:

$$Caused(F(\vec{x}), v, s) \equiv \Psi, \quad (18)$$

where is a formula which does not mention the predicate *Caused.*

This is achieved by circumscribing the predicate *Caused* in *T* with all the other predicates fixed. However, since all the axioms in *T* other than those given in Step 1 and Step 3 either do not mention *Caused* or mention it negatively, by a standard result in circumscription, the circumscription of *Caused* in *T* is equivalent to the union of *T* and the circumscription of *Caused* in the set of axioms given in Step 1 and Step 3. But these axioms are definite clauses about *Caused.* So the circumscription always has a unique minimal model. Furthermore, if the theory *T* is stratified, then the circumscription can be computed using the Clark completion. Formally, we say that *T* is *stratified* if there are no fluents $F_0, F_1, \ldots, F_n$ such that F_0 -> F_1 -> •■--> F_n -▶ F_0, where for any fluents *F* and *F', F' —➤ F (F* depends on *F')* if there is a causal rule in *T* such that *F* appears in the right hand side, and *F'* appears in the left hand side of the causal rule. For instance, the theory in the suitcase example is stratified because *up —> open* is the only dependency relation that holds for this theory. However, if we add the fluent *down,* and the above causal rules relating *up* and *down,* then the resulting theory will no longer be stratified because we'll have *up —> down —> up.*

Proposition 4.1 *Let T' be the union of the axioms in section 2, the axioms given in Step 2 and Step 4, and the Clark completion of the predicate Caused in the clauses given in Step 1 and Step 3. If the theory T is stratified, then T' is the result of circumscribing Caused in T with the other predicates fixed, i.e. for any structure M, M is a minimal model of T iff M is a model of T'.*

However, as it is well known in logic programming community, the Clark completion may be too weak if there are cycles or recursion in the causal rule (16). The following is an example of cycles:

$$Caused(up(x), v, s) \supset Caused(up(x), v, s)$$

and the following an example of recursion:

Caused(heap(x), true, s) ⊃ Caused(heap(f(x)),true, s).

This is only natural because the Clark completion is first-order, but to capture cycles and recursion we need second-order logic.

Step 6. Assume that for each fluent *F* we have computed a causation axiom of the form (18), the goal of this step is to compute for each fluent *F* a *successor state axiom* [Reiter, 1991] of the form:

$$Poss(a,s) \supset F(\vec{x}, do(a,s)) \equiv (\Phi(s) \vee (F(\vec{x},s) \wedge \Phi'(s))), \quad (19)$$

where Φ and Φ are simple state formulas about *s.*

This is achieved by replacing the occurrences of *Caused* in the pseudo successor state axiom (13) according to (18), as we have done in the suitcase example. Generally, we can obtain a successor state axiom for each fluent this way if the theory *T* is stratified:

Proposition 4.2 *If T is stratified, then there is a simple rewriting procedure by which we can obtain a successor state axiom for each fluent using the causation axioms and the pseudo-successor state axiom.*

Having computed for each fluent a successor state axiom solves the frame and the ramification problems. Successor state axioms are desirable because they have many appealing computational properties (see [Reiter, 1991]), and are the foundations underlying much of the Cognitive Robotics research project at the University of Toronto.[4]

We want to point out that even if we have computed for each fluent *F* a causation axiom of the form (18), it is not guaranteed that we'll have for each fluent a successor state axiom of the form (19). This is due to cycles such as

$$open(s) \supset Caused(open, true, s), \quad (20)$$

and recursion such as the following transitive closure axiom:

$$R(x,z,s) \wedge R(z,y,s) \supset Caused(R(x,y), true, s).$$

To handle recursion, we'll have to use second-order successor state axioms which are ones still of the form (19) but with the formulas Φ and Φ' allowed to be second-order. However, cycles cause problems in this case. For instance, given (20), we'll have a causation axiom for *open* of the form:

$$Caused(open, v, s) \equiv v = true \wedge open(s) \vee \cdots$$

So the pseudo successor state axiom will yield something like

$$Poss(a,s) \supset \{open(do(a,s)) \equiv open(do(a,s)) \vee \cdots\}$$

which is not very useful. Fortunately, there seems to be a sense that these cycles should never arise. Some authors, for example Shoham [1990], have insisted that causation be anti-reflexive. In other words, the causes of a fact should never include the fact itself. So it is wrong to write causal rules like (20). In any event, we have yet to see an example where this kind of cycle arises naturally.

Step 7. Assume that for each fluent *F* we have computed a successor state axiom of the form (19), the goal of this final step is to compute for each action *A* an *action precondition axiom* [Reiter, 1991] of the form:

$$Poss(A(\vec{x}), s) \equiv \Phi(s), \quad (21)$$

where Φ is a simple state formula.

For each action *A,* this is achieved by maximizing the relation *Poss(A(x*),s)* with *Holds(p,s)* fixed but *Caused* and *Holds(p,s')* As' $\neq$ *s* allowed to vary. The precise definition of this maximization policy will be given in the full version of this paper.[5] It turns out that there is a procedure to compute the result of the maximization using regression and the Clark completion. The procedure is a generalization of that in [Lin and Reiter, 1994], and can only be outlined here due to the space limitation. But we'll see an example in the next section.

From each causation axiom of the form (18), deduce first the following two axioms by eliminating the predicate *Caused:*

$$\Psi(v/true) \supset F(\vec{x}, s),$$
$$\Psi(v/false) \supset \neg F(\vec{x}, s),$$

where $\Psi(v/true)$ is the result of replacing the free variable *v* in Ψ by *true,* and similarly for $\Psi(v/false)$. Now for each action *A,* apply regression to these axioms and the constraints (17) given in Step 4 to generate all (non-vacuous) qualification axioms of the following form:

$$Poss(A(\vec{x}, s) \supset \Phi(s),$$

where Φ is a simple state formula about s, and is not a consequence of the axioms we have so far. Finally apply the Clark completion to these axioms to obtain an action precondition axiom of the form (21).

So to summarize, we have:

Theorem 1 *// the theory T given by steps 1 to 4 is stratified, then there is a procedure by which we can obtain a successor state axiom for each fluent, and an action precondition axiom for each action. The procedure is based on simple syntactic manipulations, and is provably correct with respect to our nonmonotonic semantics.*

5 Another Example

Having described the general procedure in last section, we now apply it to an example involving walking and shooting. This example is of interest because it has a constraint which yields an indirect effect for one action, but an implicit qualification on another.

Imagine an agent who can perform the following three actions: *start-walk, end-walk,* and *shoot.* We follow the seven-step procedure given in the last section.

Step 1. *Direct effect axioms:*

Poss(start-walk, s) ⊃
Caused(walking, true, do(start-walk, s))

Poss (end-walk, s) ⊃
Caused(walking, false, do(end-walk, s)),

Poss(shoot, s) ⊃ *Caused(dead, true, do(shoot, s)).*

[4] See ftp://ftp.ca.toronto.edu/pub/cogrob/README.html

[5] But see [Lin and Shoham, 1991] or [Lin and Reiter, 1994] for the definition of a similar policy.

Step 2. *Explicit action qualification axioms:*

$$Poss(start\text{-}walk, s) \supset \neg walking(s),$$
$$Poss(end\text{-}walk, s) \supset walking(s).$$

Step 3. *Causal rules:*

$$dead(s) \supset Caused(walking, false, s). \quad (22)$$

Step 4. There are no other domain constraints.

Step 5. Clearly, the theory generated by the above steps is stratified, so Proposition 4.1 is applicable. The Clark completion of *Caused* in the clauses in steps 1 and 3 yields:

$$Caused(dead, v, s) \equiv$$
$$v = true \wedge (\exists s')[s = do(shoot, s') \wedge Poss(shoot, s')],$$
$$Caused(walking, v, s) \equiv$$
$$v = false \wedge dead(s) \vee$$
$$v = false \wedge (\exists s')[s = do(end\text{-}walk, s') \wedge Poss(end\text{-}walk, s')] \vee$$
$$v = true \wedge (\exists s')[s = do(start\text{-}walk, s') \wedge Poss(start\text{-}walk, s')].$$

Step 6. From the pseudo successor state axiom (13) and the above causation axiom for the fluent *dead*, we obtain the following successor state axiom:

$$Poss(a, s) \supset \{dead(do(a, s)) \equiv a = shoot \vee dead(s)\}.$$

For the fluent *walking*, similar reasoning yields the axiom:

$$Poss(a, s) \supset \{walking(do(a, s)) \equiv$$
$$a = start\text{-}walk \vee$$
$$walking(s) \wedge a \neq end\text{-}walk \wedge \neg dead(do(a, s))\}.$$

Using the successor state axiom for *dead*, we can obtain the following successor state axiom for *walking*:

$$Poss(a, s) \supset \{walking(do(a, s)) \equiv$$
$$a = start\text{-}walk \vee$$
$$walking(s) \wedge \neg dead(s) \wedge a \neq end\text{-}walk \wedge a \neq shoot\}$$

Notice that the causal rule (22) has been used to derive the following indirect effect of *shoot*:

$$Poss(shoot, s) \supset \neg walking(do(shoot, s)). \quad (23)$$

Step 7. From the causation axioms for *dead* and *walking*, we can deduce the following axioms:

$$(\exists s')[s = do(shoot, s') \wedge Poss(shoot, s')] \supset dead(s),$$
$$(\exists s')[s = do(end\text{-}walk, s') \wedge Poss(end\text{-}walk, s')] \supset \neg walking(s),$$
$$(\exists s')[s = do(start\text{-}walk, s') \wedge Poss(start\text{-}walk, s')] \supset walking(s),$$
$$dead(s) \supset \neg walking(s).$$

Consider the first one. This axiom is equivalent to

$$Poss(shoot, s) \supset dead(do(shoot, s)).$$

However, this is not an action qualification axiom since *dead(do(shoot, s))* is not a simple state formula about *s*. So we replace it using its successor state axiom (regression), and obtain:

$$Poss(shoot, s) \supset (shoot = shoot \vee dead(s)).$$

But this yields a vacuous action qualification axiom since the right hand side of the implication is a tautology.

Similar efforts on the second and the third axioms end up in vain as well. So we are left with the fourth one. To see if it entails any implicit action qualifications, we instantiate it to *do(a, s):*

$$dead(do(a, s)) \supset \neg walking(do(a, s)). \quad (24)$$

Now regress both *dead(do(a, s))* and *walking(do(a, $))* according to their respective successor state axioms, we get

$$Poss(a, s) \supset \{[a = shoot \vee dead(s)] \supset$$
$$\neg[a = start\text{-}walk \vee$$
$$walking(s) \wedge \neg dead(s) \wedge a \neq end\text{-}walk \wedge a \neq shoot]\}$$

So we have:

$$Poss(start\text{-}walk, s) \supset \neg dead(s). \quad (25)$$

But this is a non-vacuous qualification axiom on *start-walk*. It can be explained intuitively as follows: If *dead* holds in state *s,* then *start-walk* must not be possible, otherwise, since it does not affect the truth value of *dead* but makes *walking* true, the resulting state will violate the constraint (24). But this constraint is derived from the causal rule (22), so we see here that this causal rule is used to derive both the indirect effect (23) for the action *shoot,* and the implicit qualification (25) on the action *start-walk.*

Now since we have considered all four consequences of the causation axioms, and there are no other constraints in step 4, the correctness of the procedure in the last section guarantees that (25) is the only implicit action qualification axiom. So we obtain the following action precondition axioms using the Clark completion:

$$Poss(start\text{-}walk, s) \equiv \neg walking(s) \wedge \neg dead(s),$$
$$Poss(end\text{-}walk, s) \equiv walking(s),$$
$$Poss(shoot, s).$$

6 Related Work

Much of the work on reasoning about action concerns causality. We shall attempt to review only those that make explicit and formal use of this notion.

Before us, Lifschitz [1987] and Haugh [1987] have also proposed using causality to formalize the effects of actions in the situation calculus. However, they consider only what we called action-triggered causation, so can only represent the direct effects of actions. In fact, both of the causation predicates in [Lifschitz, 1987] and in [Haugh, 1987] take an action but no state argument, and both of the efforts have difficulties handling the ramification problem. Similar remarks apply to [Elkan, 1992] as well despite the fact that the causation predicate there has a state argument. This is because the causation

predicate in [Elkan, 1992] continues to have an action argument, and the state argument is introduced only to help expressing complex preconditions.

Whereas we treat causality as a predicate, Geffner [1990] and McCain and Turner [1995] treat it as modal operators. Nonetheless, it seems that this work and that of [McCain and Turner, 1995] have much in common. The causal theories of [Geffner, 1990], however, are aimed at, general default reasoning. Although [Geffner, 1990] includes an example of how the general framework can be applied to reasoning about action, it is not clear if this can be done in general. In particular, it is not clear if a distinction can be made between ramification and qualification constraints in this framework.

Whereas we use causality as a primitive notion, Shoham [1990], and Iwasaki and Simon [1986] attempt to derive it from an acausal theory. In particular, Iwasaki and Simon consider deriving the causal relations from a set of acausal equations. It is not clear if this approach can be ported into the situation calculus.

Pearl [1988] argues about the need for a primitive notion of causality in general default reasoning. This paper obviously echoes the same theme. In fact, the title of this paper follows that of [Pearl, 1988].

7 Conclusions

We have argued that acausal state constraints like (1) are not adequate for representing the indirect effects of actions, and proposed a solution using causal rules like (9). By embracing causality, we are able to use only simple nonmonotonic formalisms for solving the frame, the ramification, and the qualification problems. This enables us to describe a general class of theories for which our approach is computationally tractable. In fact, we are currently working on a planner that can take as input causal theories of the form specified by steps 1 to 4 in section 4.

Acknowledgements

My thanks to Alvaro del Val, Norman Foo, and Yan Zhang for stimulating discussions related to the subject of this work. This work owes much to Ray Reiter. In fact, it is an outgrowth of our ongoing discussions on the ramification problem and the situation calculus. I also thank G. Neelakantan Kartha, Yves Lesperance, Vladimir Lifschitz, Norman McCain, and Ray Reiter for very helpful comments on earlier versions of this paper.

References

[Clark, 1978] Keith L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logics and Databases,* pages 293-322. Plenum Press, New York, 1978.

[Elkan, 1992] Charles Elkan. Reasoning about action in first-order logic. In *Proc. of the 1992 Canadian Conf. on Artificial Intelligence,* 1992.

[Finger, 1986] Jeff Finger. *Exploiting Constraints in Design Synthesis.* PhD thesis, Department of Computer Science, Stanford University, Stanford, CA, 1986.

[Geffner, 1990] Hector Geffner. Causal theories for nonmonotonic reasoning. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90),* pages 524-530, 1990.

[Ginsberg and Smith, 1988] Matthew L. Ginsberg and David E. Smith. Reasoning about action II: the qualification problem. *Artificial Intelligence,* 35:311-342, 1988.

[Haugh, 1987] Brian Haugh. Simple causal minimizations for temporal persistence and projection. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87),* pages 218-223, 1987.

[Iwasaki and Simon, 1986] Yumi Iwasaki and H.A. Simon. Causality in device behavior. *Artificial Intelligence,* 29:3-32, 1986.

[Lifschitz, 1987] Vladimir Lifschitz. Formal theories of action. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87),* pages 966-972, 1987.

[Lin and Reiter, 1994] Fangzhen Lin and Raymond Reiter. State constraints revisited. *Journal of Logic and Computation, Special Issue on Actions and Processes,* 4(5):655-678, 1994.

[Lin and Shoham, 1991]
Fangzhen Lin and Yoav Shoham. Provably correct theories of action: Preliminary report. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91),* Anaheim, CA, 1991. Full paper to appear in *JACM.*

[McCain and Turner, 1995] Norman McCain and Hudson Turner. A causal theory of ramifications and qualifications. In *This Volume,* 1995.

[McCarthy and Hayes, 1969] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4,* pages 463-502. Edinburgh University Press, Edinburgh, 1969.

[McCarthy, 1977] John McCarthy. Epistemological problems of Artificial Intelligence. In *IJCAI-77,* pages 1038-1044. Cambridge, MA, 1977.

[McCarthy, 1986] John McCarthy. Applications of circumscription to formalizing commonsense knowledge. *Artificial Intelligence,* 28:89-118, 1986.

[Pearl, 1988] Judea Pearl. Embracing causality in default reasoning. *Artificial Intelligence,* 35:259-271, 1988.

[Reiter, 1991[1] Raymond Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy,* pages 418-420. Academic Press, San Diego, CA, 1991.

[Shoham, 1990] Yoav Shoham. Nonmonotonic reasoning and causation. *Cognitive Science,* 14:213-252, 1990.

Research Grants Council 研究資助局

General Research Fund

VIEW PROPOSAL RESULT

List of Proposals

Project Number :	616909
Exercise Year :	2009 / 2010
Project Title :	Deducing consequences of a logic program using loop formulas
PI Name :	Prof Lin, Fangzhen
Institution :	The Hong Kong University of Science and Technology
Project Fund :	HK$858,000
On-costs :	HK$128,700
Total Fund Awarded :	HK$986,700
Rank :	5.0
Approved Project duration :	24 months

View Comments Return

FAQ

SCREEN ID: SCRRM013_2

Research Grants Council 研究資助局

General Research Fund

VIEW PROPOSAL RESULT

List of Proposals

Project Number : 616208
Exercise Year : 2008 / 2009
Project Title : Proving the Termination of Certain Golog programs
PI Name : Prof Lin, Fangzhen
Institution : The Hong Kong University of Science and Technology
Project Fund : HK$327,937
Rank : 4.0
Approved Project duration : 30 months

View Comments Return

FAQ

SCREEN ID: SCRRM013_2

Research Grants Council 研究資助局

General Research Fund

VIEW PROPOSAL RESULT

List of Proposals

Project Number : 616707
Exercise Year : 2007 / 2008
Project Title : Discovering theorems in game theory
PI Name : Prof Lin, Fangzhen
Institution : The Hong Kong University of Science and Technology
Project Fund : HK$391,591
Rank : 4.0
Approved Project duration : 24 months

View Comments | Return

FAQ

SCREEN ID: SCRRM013_2

Research Grants Council 研究資助局

General Research Fund

VIEW PROPOSAL RESULT

List of Proposals

Project Number : 616806
Exercise Year : 2006 / 2007
Project Title : First-Order Loop Formulas
PI Name : Prof Lin, Fangzhen
Institution : The Hong Kong University of Science and Technology
Project Fund : HK$534,000
Rank : 5.0
Approved Project duration : 36 months

View Comments Return

FAQ

SCREEN ID: SCRRM013_2

KR Inc Home
KR2012
Past Conferences
Proceedings
Prizes
KR Inc Organization
Related Conferences
KR Mailing Lists
Resources
Contact Information

Ray Reiter Best Paper Prize

The Ray Reiter Best Paper Prize was introduced in 2004 in honour of the contributions made by Professor Ray Reiter. The prize is sponsored by Artificial Intelligence Journal.

2010

The Combined Approach to Query Answering in DL-Lite
R. Kontchakov, C. Lutz, D. Toman, F. Wolter, M. Zakharyaschev

2008

Can You Tell the Difference Between DL-Lite Ontologies?
Roman Kontchakov, Frank Wolter, and Michael Zakharyaschev

2006

First-Order Loop Formulas for Normal Logic Programs
Yin Chen, Fangzhen Lin, Yisong Wang, Mingyi Zhang

Redoing the Foundations of Decision Theory
Lawrence Blume, David Easley, Joseph Halpern

2004

Expressive power and succinctness of propositional languages for preference representation
Sylvie Coste-Marquis, Jerome Lang, Paolo Liberatore, Pierre Marquis

Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR2000)

Breckenridge, Colorado, USA 12-15 April 2000



This is to certify that the two following papers have been selected for the best paper award:

Spatio-temporal representation and reasoning based on RCC-8

Wolter F., Zakharyaschev, M.

On Strongest Necessary and Weakest Sufficient Conditions

Lin F.

Anthony G. Cohn, (Conference Chair)

Fausto Giunchiglia, (Program Chair)

Bart Selman, (Program Chair)

Fifteenth International Joint Conference on Artificial Intelligence

IJCAI-97 DISTINGUISHED PAPER AWARD

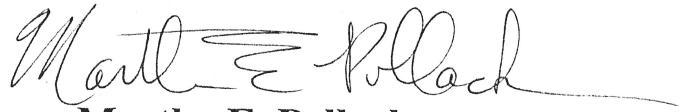
Awarded to

Fangzhen Lin

for his paper

"Applications of the Situation Calculus to Formalizing Control and Strategic Information: The PROLOG Cut Operator"

Presented at
IJCAI-97
Tuesday, 26 August 1997
Nagoya, Japan



Martha E. Pollack
IJCAI-97 Program Chair

THE CROUCHER FOUNDATION

Upon recommendations by distinguished members
of the international scientific community
The Trustees are pleased to acknowledge
excellence in the scientific work of

Professor Lin Fangzhen

with the award of
The Croucher Senior Research Fellowship
("The Croucher Award")
for the academic year 2006-07

This award was presented to the winner by
Professor Zhou Guangzhao
Chairman, China Association For Science and Technology
Formerly President, The Chinese Academy of Sciences
on 28th March 2006

THE CROUCHER FOUNDATION
Awards
Official
Certification

Yuet Wai Kan, F.R.S.
CHAIRMAN

The Fifth International Conference on
Artificial Intelligence Planning and Scheduling

presents a Celcorp Prize for performance
in the AIPS 2000 Planning Competition for

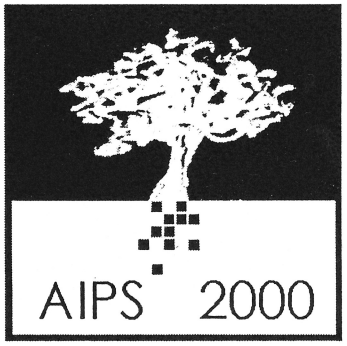
Honorable Mention

for the planning system

System R

to

Fangzhen Lin



Sponsored by Celcorp

Breckenridge, Colorado
April 16, 2000



National Conference on Artificial Intelligence

July 25 – 29, 2004 ♦ San Jose, California

The American Association for Artificial Intelligence presents an AAAI-04 Outstanding Paper Honorable Mention to:

Joohyung Lee
The University of Texas at Austin
Fangzhen Lin
Hong Kong University of Science and Technology

for the paper entitled:

Loop Formulas for Circumscription

Ronald J. Brachman

Ronald J. Brachman, President, AAAI

Subject: Your KR paper is selected for Artificial Intelligence Journal fast track reviewing
From: Didier DUBOIS <dubois@irit.fr>
Date: Tue, 18 May 2004 16:35:41 +0200
To: flin@cs.ust.hk
CC: welty@us.ibm.com, Mary-Anne@it.uts.edu.au

Dear Colleague

It is our pleasure to announce that your paper to be presented at the
KR2004 conference
Discovering State Invariants
Author Fangzhen Lin

has been selected by the Programme Committee of this conference to be
recommended for the Artificial Intelligence Journal fast track reviewing,
with a view to publish a long version thereof in this prestigious journal.

Congratulations! The list of 5 selected papers has been forwarded to the
Editor-in Chief who will extend an official invitation to contribute this
long version to the journal.

It is however important for you to note the following:

Acceptance to the journal subsequent to this invitation will NOT be
automatic. Under AIJ policy, the journal paper is supposed to be a
nontrivial extension
of the conference paper. The papers, which should be significantly
different from their conference versions, would be published as regular
papers but identified as revisions of papers first given at KR-04.This is
particularly important in the case of a conference such as KR where the
conference itself is considered by most to have an archival status (a
custom typical of the field, quite unusual in many other fields).

What the AI journal can offer is a rapid review process (first-round
reviewing in about 4 weeks), plus the honor of being in the distinguished

set of KR2004. And, due to the intrinsic quality of your work, it is clear
the chance of getting accepted is very high. However all submissons, be
they brilliant as we suppose yours will be, must comply with the journal
reviewing requirements.

Congratulations again for your paper standing among the five best ones
according the programme committee.

Looking forward to hearing your presentation in Whistler.

Best regards

Chris, Didier, Mary-Anne

Didier Dubois
Co-Editor-in Chief, Fuzzy Sets and Systems
Porte 308
IRIT, Université Paul Sabatier,
118 route de Narbonne,
31062, Toulouse, cedex 4, FRANCE.

Tel :+335 61.55.63.31 (05 61.55.63.31 en France)
E-mail : dubois@irit.fr
Fax : +335 61.55.62.39 (05 61.55.62.39 en France)

GOLOG: A logic programming language for dynamic domains [PDF] from psu.edu
HJ Levesque, R Reiter, Y Lespérance, F **Lin**... - The Journal of Logic ..., 1997 - Elsevier Find@HKUST
This paper proposes a new logic programming language called GOLOG whose interpreter automatically maintains an explicit representation of the dynamic world being modeled, on the basis of user supplied axioms about the preconditions and effects of actions and the initial state of the ...
Cited by 833 - Related articles - BL Direct - All 29 versions

ASSAT: Computing answer sets of a logic program by SAT solvers [PDF] from psu.edu
F **Lin**... - Artificial Intelligence, 2004 - Elsevier Find@HKUST
We propose a new translation from normal logic programs with constraints under the answer set semantics to propositional logic. Given a normal logic program, we show that by adding, for each loop in the program, a corresponding loop formula to the program's completion, we ...
Cited by 304 - Related articles - BL Direct - All 20 versions

[PDF] Embracing Causality in specifying the Indirect E ects of Actions [PDF] from psu.edu
F **Lin** - Proceedings of the International Joint Conference on ..., 1995 - Citeseer
This paper considers the problem of specify- ing the e ects of actions in the situation cal- culus using domain constraints. We argue that normal state constraints that refer to only the truth values of uents are not strong enough for this purpose, and that a notion of causation needs to ...
Cited by 276 - Related articles - View as HTML - BL Direct - All 22 versions

Provably correct theories of action [PDF] from toronto.e
F **Lin**... - Journal of the ACM (JACM), 1995 - portal.acm.org Find@HKUST
Page 1. Provably Correct Theories of Action **FANGZHEN LIN** UniuersiQof Toronto, Toronto, (hnada AND YOAV SHOHAM Stanford Unit>ersity, Stanford, California Abstract. We investigate logical formalization of the effects of actions in the situation calculus. ...
Cited by 139 - Related articles - BL Direct - All 10 versions

Foundations of a logical approach to agent programming [PDF] from psu.edu
Y Lespérance, H Levesque, F **Lin**, D Marcu... - Intelligent Agents II ..., 1996 - Springer
Abstract. This paper describes a novel approach to high-level agent program- ming based on a highly developed logical theory of action. The user provides a specification of the agents' basic actions (preconditions and effects) as well as of relevant aspects of the environment, in an ...
Cited by 123 - Related articles - Find@HKUST - BL Direct - All 28 versions

Forget it [PDF] from psu.edu
F **Lin**... - Working Notes of AAAI Fall Symposium on Relevance, 1994 - aaai.org
Introduction We encountered the need for a theory of forgetting in our work on Cognitive Robotics. 1 In our effort to control an autonomous robot, we imagine the robot has a knowledge base representing the initial world state. As the robot performs actions, she needs to bring ...
Cited by 127 - Related articles - All 20 versions

A logical approach to high-level robot programming–a progress report
Y Lespérance, HJ Levesque, F **Lin**... - Control of the physical ..., 1994 - aaai.org
Abstract This paper describes a novel approach to high-level robot programming based on a highly developed logi- ca] theory of action. The user provides a specification of the robot's basic actions (their preconditions and effects on the environment) as well as of relevant as- ...
Cited by 110 - Related articles - All 8 versions

How to progress a database* 1 Find@HKUST
F **Lin**... - Artificial Intelligence, 1997 - Elsevier
In this paper, we consider the general problem of progressing an initial database in response to a given sequence of actions. We appeal to the situation calculus and an axiomatization of actions which addresses the frame problem (Reiter (1991)). This setting is considerably ...
Cited by 109 - Related articles - All 8 versions

Concurrent actions in the situation calculus
F **Lin**... - Proceedings of the National Conference on Artificial ..., 1992 - aaai.org
We propose a representation of concurrent actions; rather than invent a new formalism, we model them within the standard situation calculus by introducing the notions of global actions and primitive actions, whose relationship is analogous to that between situations and fluents. The result ...
Cited by 94 - Related articles - BL Direct - All 5 versions

[CITATION] Argument systems: a uniform basis for nonmonotonic reasoning
F **Lin**... - Proceedings of the First ..., 1989 - Morgan Kaufmann Publishers
Cited by 83 - Related articles - Library Search

Create email alert

Gooooooooooogle

Result Page: 1 2 3 4 5 6 7 8 9 10 Next

fangzhen lin Search

Go to Google Home - About Google - About Google Scholar

©2010 Google