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 $do([a|L], s) = do(L, do(a, s))$.

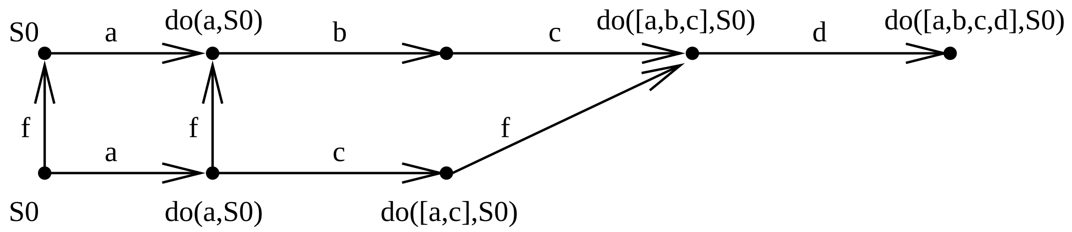

Figure 1: A function f as required by the axiom (6)

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

$$\begin{array}{lll} A_1(\vec{x}): & F(\vec{x}) & \texttt{:-}\ G_1 \\ \vdots & \vdots & \\ A_k(\vec{x}): & F(\vec{x}) & \texttt{:-}\ G_k \end{array}$$

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:

$$\begin{array}{l} (\exists \vec{y_1})H(G_1,s) \supset H(F(\vec{x}), do(A_1(\vec{x}),s)), \\ \quad \vdots \\ (\exists \vec{y_k})H(G_k,s) \supset H(F(\vec{x}), do(A_k(\vec{x}),s)), \end{array}$$

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:

$$\begin{array}{l} H(F(\vec{x}), do(a,s)) \equiv \{a = A_1(\vec{x}) \wedge (\exists \vec{y_1})H(G_1,s) \vee \cdots \vee \\ \quad (a = A_k(\vec{x}) \wedge (\exists \vec{y_k})H(G_k,s)) \vee H(F(\vec{x}),s)\}. \end{array} \tag{7}$$

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:*

$$\{H(F(\vec{x}), S_0) \equiv false \mid F \textit{ is a fluent}\}.$$

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:

$$
\begin{aligned}
&H(max(x, y, z), do(a, s)) \equiv \\
&\qquad a = A_1(x, y, z) \wedge H(leq(x, y), s) \wedge z = y \vee \\
&\qquad (a = A_2(x, y, z) \wedge z = x) \vee H(max(x, y, z), s), \\
&H(leq(x, y), do(a, s)) \equiv \{a = B_1(x, y) \wedge x = 1 \wedge y = 2 \vee \\
&\qquad a = B_2(x, y) \wedge x = 1 \wedge y = 1 \vee \\
&\qquad (a = B_3(x, y) \wedge x = 2 \wedge y = 2) \vee H(leq(x, y), s)\}.
\end{aligned}
$$

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.

$$cut(\vec{x}): \quad F(\vec{x}) \texttt{ :- } G_1 \texttt{ \& ! \& } G_2$$

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:*

$$\begin{aligned} s \prec s' \equiv (\exists a, b, s_1, s_2)\{s = do(a, s_1) \wedge s' = do(b, s_2) \wedge \\ [a \prec b \vee (a = b \wedge s_1 \prec s_2)]\}. \end{aligned} \tag{10}$$

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* $\texttt{minimal}(G, S)$ *the following formula:*

$$H(G, S) \wedge \neg(\exists s)[s \subset S \wedge H(G, s)].$$

Intuitively, $\mathtt{minimal}(G, S)$ 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

$$\mathcal{D} \models \mathtt{minimal}(leq(1,2), do(B_1(1,2), S_0)).$$

But

$$\mathcal{D} \not\models \mathtt{minimal}(leq(1,2), do([B_1(1,2), A_1(1,2,2)], S_0)),$$

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:

$$\Sigma \models (\forall s).H(G, s) \supset (\exists s')(s' \subseteq s \wedge \mathtt{minimal}(G, s')).$$

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:

$$\begin{aligned} & Acc(s) \supset (\forall \vec{x})\{(\exists s')[Acc(s') \wedge (\exists \vec{\xi})H(G_1, s')] \supset \\ & \quad (\forall s_1)[s_1 \subseteq s \wedge \mathtt{minimal}(F(\vec{x}), s_1) \supset \\ & \qquad (\exists a, s_2)(s_1 = do(a, s_2) \wedge a \prec cut(\vec{x})) \vee \\ & \qquad (\exists s_2, s_3)(s_1 = do(cut(\vec{x}), s_2) \wedge s_3 \subseteq s_2 \wedge \mathtt{first\text{-}der}(G_1, s_3))]\}, \end{aligned} \tag{11}$$

where $\vec{\xi}$ is the tuple of the free variables that are in G_1 but not in $\vec{x}$, and $\mathtt{first\text{-}der}(G_1, s_3)$ stands for the following formula:

$$(\exists \vec{\xi})\mathtt{minimal}(G_1, s_3) \wedge \neg(\exists s_3')[s_3' \prec s_3 \wedge Acc(s_3') \wedge (\exists \vec{\xi})\mathtt{minimal}(G_1, s_3')],$$

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

$$\begin{gathered} Acc(s) \supset \Psi_1(s), \\ \vdots \\ Acc(s) \supset \Psi_k(s), \end{gathered}$$

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

$$\begin{aligned} & Acc(s) \equiv (\forall x, y, z)\{(\exists s')(Acc(s') \wedge H(leq(x,y), s')) \supset \\ & \quad (\forall s_1)[s_1 \subseteq s \wedge \texttt{minimal}(max(x,y,z), s_1) \supset \\ & \quad\quad (\exists a, s_2)(s_1 = do(a, s_2) \wedge a \prec A_1(x,y,z)) \vee \\ & \quad\quad (\exists s_2, s_3)(s_1 = do(A_1(x,y,z), s_2) \wedge s_3 \subseteq s_2 \wedge \texttt{first-der}(leq(x,y), s_3))]\}. \end{aligned} \tag{13}$$

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

$$\mathcal{E} \models \texttt{minimal}(leq(1,2), S_1) \land \neg(\exists s).Acc(s) \land \texttt{minimal}(leq(1,2), s) \land s \prec S_1.$$

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

$$\mathcal{E} \models (\exists s).Acc(s) \land H(leq(1,2), s),$$
$$\mathcal{E} \models (\forall s).\texttt{minimal}(max(1,2,1), s) \equiv s = do(A_2(1,2,1), S_0),$$

it follows from (13) that for $Acc(S)$ to hold, it must be the case that

$$(\exists a, s)[do(A_2(1,2,1), S_0) = do(a, s) \land a \prec A_1(1,2,1)] \lor$$
$$(\exists s, s')[do(A_2(1,2,1), S_0) = do(A_1(1,2,1), s) \land s' \subseteq s \land \texttt{first-der}(leq(1,2), s')],$$

which is obviously impossible.

Generalizing the above reasoning, we have:

$$\begin{aligned}\mathcal{E} \models (\forall x, y)\{&(\exists s)H(leq(x,y), s) \land x \neq y \supset \\ &[(\exists s)(Acc(s) \land H(max(x,y,y), s)) \land \neg(\exists s)(Acc(s) \land H(max(x,y,x), s))]\}.\end{aligned}$$

So the max program indeed defines max correctly. However since

$$\mathcal{E} \not\models (\forall x, y, z)[(\exists s)H(max(x,y,z), s) \equiv (\exists s)(Acc(s) \land H(max(x,y,z), s))],$$

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

$$(\forall s)Acc(s) \supset \Psi_1,\ \ldots,\ (\forall s)Acc(s) \supset \Psi_k$$

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:*

$$(\exists s)(Acc(s) \wedge H(G, s)) \equiv (\exists s)(Acc(s) \wedge \mathtt{minimal}(G, s)).$$

Proof: By induction on situations, if there is a plan for G, then there is a minimal plan for it:

$$\Sigma \models (\forall s).H(G, s) \supset (\exists s')(s' \subseteq s \wedge \mathtt{minimal}(G, s')).$$

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:*

$$\mathtt{minimal}(G, s) \supset Acc(s),$$
$$(\exists s)H(G, s) \equiv (\exists s)(Acc(s) \wedge H(G, s)).$$

Proof: If $\texttt{minimal}(G, s)$, 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*

$$1 + max\{f(F') \mid F' \textit{ appears in the definition of } F\}.$$

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

$$\begin{array}{ll} cut_1(\vec{x_1}): & F_1(\vec{x_1}) \texttt{ :- } G_1 \texttt{ \& ! \& } G_1'. \\ \vdots & \vdots \\ cut_k(\vec{x_k}): & F_k(\vec{x_1}) \texttt{ :- } G_k \texttt{ \& ! \& } G_k'. \end{array}$$

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,

$$(\exists s)[Acc(s) \wedge H(G,s)] \equiv (\exists s)[Acc(s) \wedge \texttt{minimal}(G,s)].$$

Let

$$(\forall s)(Acc(s) \supset \Psi_l(s)),\ \ldots,\ (\forall s)(Acc(s) \supset \Psi_m(s))$$

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 $\texttt{minimal}(G,s)$, then s cannot be a derivation for any fluents of rank $> n+1$, we have

$$(\exists s)[Acc(s) \wedge \texttt{minimal}(G,s)] \equiv (\exists s)[\Psi_l(s) \wedge \cdots \wedge \Psi_m(s) \wedge \texttt{minimal}(G,s)].$$

According to the way that $\Phi_l(s), \ldots, \Phi_m(s)$ are obtained from $\Psi_l(s), \ldots, \Psi_l(s)$, respectively, by inductive assumption, we have

$$(\exists s)[\Psi_l(s) \wedge \cdots \wedge \Psi_m(s) \wedge \texttt{minimal}(G,s)] \equiv (\exists s)[\Phi_l(s) \wedge \cdots \wedge \Phi_m(s) \wedge \texttt{minimal}(G,s)].$$

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:

```
r1:    p :- q'.
r2:    q :- p'.
r3:    p' :- p & ! & fail.
r4:    p'.
r5:    q' :- q & ! & fail.
r6:    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:

$$
\begin{aligned}
&Acc(s) \equiv \\
&\quad (\exists s')(Acc(s') \wedge H(p, s')) \supset (\forall s_1)[s_1 \subseteq s \wedge \mathtt{minimal}(p', s_1) \supset \\
&\qquad (\exists a, s_2)(s_1 = do(a, s_2) \wedge a \prec r_3) \vee \\
&\qquad (\exists s_2, s_3)(s_1 = do(r_3, s_2) \wedge s_3 \subseteq s_2 \wedge \mathtt{first\text{-}der}(p, s_3))] \wedge \\
&\quad (\exists s')(Acc(s') \wedge H(q, s')) \supset (\forall s_1)[s_1 \subseteq s \wedge \mathtt{minimal}(q', s_1) \supset \\
&\qquad (\exists a, s_2)(s_1 = do(a, s_2) \wedge a \prec r_5) \vee \\
&\qquad (\exists s_2, s_3)(s_1 = do(r_5, s_2) \wedge s_3 \subseteq s_2 \wedge \mathtt{first\text{-}der}(q, s_3))].
\end{aligned}
$$

Since r_4 is the only rule by which p' can be established, we have:

$$\mathtt{minimal}(p', s) \supset s = do(r_4, S_0).$$

Similarly, $(\forall s)[\mathtt{minimal}(q', s) \supset s = do(r_6, S_0)]$. 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:

$A_F(\vec{x})$: $F'(\vec{x})$ `:-` $F(\vec{x})$ `& ! & fail.`
$A'_F(\vec{x})$: $F'(\vec{x})$.

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:

$$\begin{array}{ll} A_F(\vec{x}): & F'(\vec{x}) \texttt{ :- } F(\vec{x}) \texttt{ \& ! \& fail.} \\ A'_F(\vec{x}): & F'(\vec{x})\texttt{.} \end{array}$$

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: $H(\texttt{not}\ p, s)$ is $\neg(\exists s')H(p, s)$. Notice that $H(\texttt{not}\ p, s)$ is in fact independent of the situation argument. For example, given the following clauses:

$$\begin{array}{ll} r_1(x): & \texttt{p(x) :- not q(x)} \\ r_2(x): & \texttt{p(x) :- x=u} \end{array}$$

the successor state axiom for F is:

$$\begin{array}{c} H(p(x), do(a, s)) \equiv a = r_1(x) \wedge H(\texttt{not}\ q(x), s) \vee \\ (a = r_2(x) \wedge x = u) \vee H(p(x), s). \end{array}$$

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

$$\begin{array}{l} \mathcal{E} \models Acc(s) \equiv \\ \quad (\forall \vec{x_1})\{H(F'_1(\vec{x_1}, s) \supset \neg(\exists s')[Acc(s') \wedge H(F_1(\vec{x_1}), s')]\} \wedge \cdots \wedge \\ \quad (\forall \vec{x_k})\{H(F'_k(\vec{x_k}), s) \supset \neg(\exists s')[Acc(s') \wedge H(F_k(\vec{x_k}), s')]\}. \end{array} \quad (18)$$

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,

$$\begin{array}{c} M \models (\exists s)H(F(\vec{u}), s) \\ \Longleftrightarrow \\ M' \models (\exists s)Acc(s) \wedge H(F(\vec{u}), s) \end{array} \tag{19}$$

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

$$F(\vec{u}) \texttt{ :- } G \texttt{ \& not } F_i(\vec{t}).$$

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 $\mathcal{D}$. ∎