Representing Concurrent Actions in Extended Logic Programming

Chitta Baral and Michael Gelfond
Department of Computer Science
University of Texas at El Paso
El Paso, Texas 79968, U.S.A.
{chitta,mgelfond}@cs.ep.utexas.edu

Abstract

Gelfond and Lifschitz introduce a declarative language *A* for describing effects of actions and define a translation of theories in this language into extended logic programs(ELP's). The purpose of this paper is to extend the language and the translation to allow reasoning about the effects of concurrent actions. Logic programming formalization of situation calculus with concurrent actions presented in the paper can be of independent interest and may serve as a test bed for the investigation of various transformations and logic programming inference mechanisms.

1 Introduction

Gelfond and Lifschitz [3] introduce a declarative language *A* for describing effects of actions and define the semantics of this language based on the notion of a finite automata. The simplicity of the language and its semantics makes it easier to describe the ontology of actions and contributes to establishing correctness (and sometimes completeness) of various logical formalizations of their effects. In particular, a theory of action stated in a language of extended logic programs(ELP's) [2] was described in [3] as a translation from a subset of *A* and proven to be sound w.r.t. the automata based semantics. Soundness and completeness w.r.t. this semantics of the approaches to formalizing actions proposed earlier by Pednault [II], Reiter [13] and Baker [I] was recently proved by Kartha [5].

Although the language *A* is adequate for formalizing several interesting domains, its expressive power is rather limited. In particular, every action is assumed to be executable in any situation and only one action can be performed at a time. In this paper we expand the syntax and semantics of *A* to remove these limitations and to allow for a representation of concurrent actions. As in [3], we translate theories in the resulting language *Ac* into ELP's and prove correctness of this translation. The translation can be viewed as a logic programming counterpart of situation calculus [0] and is interesting in its own right. The paper is organized as follows: In section two we define the syntax and semantics of the language *Ac*- Section three describes the translation of theories from *Ac* into ELP's while section four illustrates the translation by the way of examples. Due to space limitations the proof of correctness of the translation and other results will be presented in the full paper.

2 A language *Ac*

2.1 Syntax

First let us recall the syntax of language *A* from [3]. The alphabet of *A* consists of two disjoint nonempty sets £i and E2 of symbols, called fl*uent names* and *action names.* A fl*uent literal* is a fluent name possibly preceded by -. A *v-proposition* is an expression of the form

$$F \text{ after } A_1, \ldots, A_m \qquad (1)$$

where *F* is a fluent literal, and $A_1, \ldots, A_m$ ($m > 0$) are action names. If $m = 0$, (1) is written as

initially *F.*

An *e-proposition* is an expression of the form

$$A \text{ causes } F \text{ if } P_1, \ldots, P_n \qquad (2)$$

where *A* is an action name, and each of $F, P_1, \ldots, P_n$ ($n > 0$) is a fluent literal. $P_1, \ldots, P_n$ are called *preconditions* of (2). If $n = 0$, we write this proposition as

A causes *F.*

A *domain description* in *A* is a set of propositions.

The syntax of *Ac* differ from the syntax of *A* only in the definition of action names. By an action name of *Ac* we mean an arbitrary finite set *{a1,..., an}* of elements of £2. Intuitively, an action name {a,-} denotes a *unit action* while an action name $A = \{a_1, \ldots, a_n\}$ where $n > 1$ denotes a *compound action* - a set of unit actions which are performed concurrently and which start and stop cotemporaneously. For simplicity we will often identify a unit action name {a,} with a. To illustrate the notion of a domain description in *Ac* let us consider the following examples from 14]:

Example 1. Mary is lifting a bowl of soup from the kitchen table, while John is opening the door to the dining room.

To represent this story in *Ac* let us consider an alphabet consisting of a fluent name *Lifted* and *Opened* and

two unit actions *Lift* and *Open*. The initial situation is described by v–propositions:

initially $\neg Lifted$ **initially** $\neg Open$

The effects of the actions can be described by the axioms:

$\{Lift\}$ **causes** *Lifted* $\{Open\}$ **causes** *Opened*

The resulting domain description will be denoted by D_1. Intuitively, the effects of the two actions of D_1 are completely independent and so both *Lifted* and *Opened* should hold after the execution of the compound action $\{Lift, Open\}$. The next example describes actions whose effects are mutually dependent.

Example 2. Whenever Mary tries to lift the bowl with one hand, she spills the soup. When she uses both hands, she does not spill the soup.

This time let us consider an alphabet consisting of a fluent name *Spilled* and two unit actions *Lift_l* and *Lift_r*. The initial situation may be described by a proposition:

initially $\neg$ *Spilled*

and the effects of actions are represented by propositions:

$\{Lift\_l\}$ **causes** *Spilled*

$\{Lift\_r\}$ **causes** *Spilled*

$\{Lift\_l,\ Lift\_r\}$ **causes** $\neg$ *Spilled*

The resulting domain description will be denoted by D_2.

2.2 Semantics

To describe the semantics of $\mathcal{A}_C$, we will define "models" of a domain description, and when a v-proposition is "true" in a model. If a v-proposition P is true in all models of a domain description D, we say that D *entails* P. As defined in [3], a *state* is a set of fluent names; given a fluent name F and a state σ, we say that F *holds* in σ if $F \in \sigma$; $\neg F$ *holds* in σ if $F \notin \sigma$.

A *transition function* is a mapping Φ of a subset of the set of pairs (A, σ), where A is an action name and σ is a state, into the set of states.[1] As in [3], a *structure* is a pair (σ_0, Φ), where σ_0 is a state (the *initial state* of the structure), and Φ is a transition function. We say that a sequence of action names $A_1, \ldots, A_m$ is *executable* in a structure $M = (\sigma_0, \Phi)$ if for every $1 \leq k \leq m$

$$\Phi(A_k, \Phi(A_{k-1}, \ldots, \Phi(A_1, \sigma_0)\ldots))$$

is defined. The resulting state will be denoted by $M^{(A_1,\ldots,A_m)}$.

We say that a v-proposition (1) is *true* (*false*) in a structure M if

1. $A_1, \ldots, A_m$ is executable in M,

2. F holds (does not hold) in $M^{(A_1,\ldots,A_m)}$.

In particular, the proposition "**initially** F" is true in M iff F holds in the initial state of M.

[1] Recall that in the definition of a transition function in the semantics of $\mathcal{A}$, Φ must be defined on the set of all such pairs.

We say that execution of an action A in a state σ *immediately causes* a fluent literal F if

There is an e–proposition "A **causes** F **if** $P_1, \ldots, P_n$" from the domain D such that for every i, $1 \leq i \leq n$, P_i holds in σ.

We say that execution of an action A in a state σ *causes* a fluent literal F if

1. A *immediately causes* F, or

2. There is a $B \subseteq A$, such that execution of B in σ *immediately causes* F and there is no C such that $B \subset C \subseteq A$ where execution of C in σ causes $\neg F$.

Let A be an action and σ be a state and consider:

$B_f(A, \sigma) = \{f : f$ is a fluent name and execution of A in σ causes $f\}$,

$B'_f(A, \sigma) = \{f : f$ is a fluent name and execution of A in σ causes $\neg f\}$.

A structure (σ_0, Φ) will be called a *model* of a domain description D if the following conditions are satisfied:

1. Every v-proposition from D is true in (σ_0, Φ);

2. For every action $A = \{a_1, \ldots, a_n\}$ and every state σ

(i) if $B_f(A, \sigma) \cap B'_f(A, \sigma) = \emptyset$ then $\Phi(A, \sigma)$ is defined and

$$\Phi(A, \sigma) = \sigma \cup B_f(A, \sigma) \setminus B'_f(A, \sigma).$$

(ii) otherwise $\Phi(A, \sigma)$ is undefined.

It is clear that there can be at most one transition function Φ satisfying conditions (i)–(ii). Consequently, different models of the same domain description can differ only by their initial states.

Example 3. Consider the domain description D_1 from Example 1, the initial state $\sigma_0 = \emptyset$ and the transition function Φ defined as follows:

$\Phi(\emptyset, \sigma) = \sigma$

$\Phi(Open, \sigma) = \sigma \cup \{Opened\}$

$\Phi(Lift, \sigma) = \sigma \cup \{Lifted\}$

$\Phi(\{Open, Lift\}, \sigma) = \sigma \cup \{Opened, Lifted\}$

It is easy to see that the structure (σ_0, Φ) is the only model of the domain description D_1 and therefore D_1 entails v–propositions *Opened* **after** $\{Open, Lift\}$ and *Lifted* **after** $\{Open, Lift\}$.

Example 4. Consider a domain description D_3 containing three unit actions *Paint*, *Close* and *Open*, and two fluents, *Opened* and *Painted*. The effects of these actions are defined by the following e–propositions:

Close **causes** $\neg$*Opened*

Open **causes** *Opened*

Paint **causes** *Painted*.

Let a transition function Φ be defined as follows:

$\Phi(\emptyset, \sigma) = \sigma$

$\Phi(Paint, \sigma) = \sigma \cup \{Painted\}$

$\Phi(Close, \sigma) = \sigma \setminus \{Opened\}$

$\Phi(Open, \sigma) = \sigma \cup \{Opened\}$

$\Phi(\{Paint, Close\}, \sigma) = \sigma \cup \{Painted\} \setminus \{Opened\}$

$\Phi(\{Paint, Open\}, \sigma) = \sigma \cup \{Painted\} \cup \{Opened\}$

Notice, that for a pair (A, σ) where σ is an arbitrary state and $A = \{Open, Close\}$ or $A = \{Open, Close, Paint\}$, Φ is undefined.

It is easy to see that any structure $\{\sigma, \Phi\}$ where $\sigma \subset \{Opened, Painted\}$ is a model of D_3 and that D_3 has no other models.

A domain description is *consistent* if it has a model, and *complete* if it has exactly one model. For instance, domain descriptions D_1 and D_3 from Examples 1 and 4 are consistent, D_1 is complete, and a domain description containing the v–propositions **initially** F and **initially** $\neg F$ is inconsistent.

It is interesting to compare a new semantics with that defined in [3]. The comparison of course is only possible for the domain descriptions not containing names for compound actions. But, as demonstrated by the following example, even in this case the new semantics is somewhat more powerful than the old one.

Example 5. Consider a domain description D_4 containing an action name A, a fluent name F and two e–propositions

A **causes** F $\qquad$ A **causes** $\neg F$

According to the semantics from [3] D is inconsistent while it is easy to check that $M = (\emptyset, \Phi)$ where $\Phi(\emptyset, \sigma) = \sigma$ is a model of D.

The following proposition shows that for descriptions consistent in the sense of [3] both semantics coincide. Models of D in the sense of [3] will be called *s–models*.

Proposition 1. Let D be a domain description not containing compound actions and assume that D has an s–model. Let $M = (\sigma, \Phi)$ be a structure of M, and $M^* = (\sigma, \Phi^*)$ where Φ^* is Φ restricted to unit actions.

Then M is a model of F iff M^* is an s–model of D and for every s-model N of D there is a model M of D such that $N = M^*$. □

3 From $\mathcal{A}_C$ to Extended Logic Programs

3.1 Extended Logic Programs

Extended logic programs were introduced in [2] (see also [10]) as a tool for reasoning in the presence of incomplete information. They are defined as collections of rules of the form

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

where each L_i is a literal, i.e. an atom possibly preceded by $\neg$, and *not* is the negation as failure. Intuitively the rule can be read as: if $L_1, \ldots, L_m$ are believed and it is not true that $L_{m+1}, \ldots, L_n$ are believed then L_0 is believed. A program determines a collection of *answer sets* – sets of ground literals representing possible beliefs of the program. A program is consistent if it has an answer set not containing contradictory literals. A ground literal L is *entailed* by an ELP if it belongs to all of its answer sets. The rules of ELP's can be identified with defaults [12]

$L_1 \wedge \ldots \wedge L_m : \overline{L_{m+1}}, \ldots, \overline{L_n} / L_0$

($\overline{L}$ stands for the literal complementary to L.) As was shown in [2] a literal L is entailed by a program Π iff it belongs to all extensions of the corresponding default theory.

In our further discussion we will need the following simple Lemma about ELP's:

Lemma 1. For any answer set A of an ELP Π:

(a) For any ground instance of rule from Π, if $\{L_1, \ldots, L_m\} \subseteq A$ and $\{L_{m+1}, \ldots, L_n\} \cap A = \emptyset$ then $L_0 \in A$.

(b) If A is consistent and $L_0 \in A$ then there exist a ground instance of a rule from Π such that $\{L_1, \ldots, L_m\} \subseteq A$ and $\{L_{m+1}, \ldots, L_n\} \cap A = \emptyset$.

3.2 The translation π

In this section we describe the translation π from domain descriptions to ELP's and prove the soundness of this translation.

The ELP πD, corresponding to a domain description D, uses variables of three sorts: *situation* variables $s, s', \ldots$, *fluent* variables $f, f', \ldots$, and *action* variables $a, a', \ldots$[2]. We also need a sort for *fluent literals* whose terms are of the form F or $\overline{F}$ where F is a term of the type fluent. Its language includes the situation constant S_0, and the fluent names and action names of D, that become object constants of the corresponding sorts. There are also some predicate and function symbols; their sorts will be clear from their use in the rules below. Of special importance is a function symbol $\{\}$ which will be used to form terms of the action type and a function *result* used to form the terms of the type situation.

The program πD will consist of the translations of the individual propositions from D along with other axioms.

1. **Inertia Axioms**:

(a) $Holds(f, Result(a, s)) \leftarrow Holds(f, s),$
$\qquad not\ Noninert(f, a, s),\ atomic(a)$

$\neg Holds(f, Result(a, s)) \leftarrow \neg Holds(f, s),$
$\qquad not\ Noninert(f, a, s),\ atomic(a)$

(b) $Holds(f, s) \leftarrow Holds(f, Result(a, s)),$
$\qquad not\ Noninert(f, a, s),\ atomic(a)$

$\neg Holds(f, s) \leftarrow \neg Holds(f, Result(a, s)),$
$\qquad not\ Noninert(f, a, s),\ atomic(a)$

[2] Using a sorted language implies, that all atoms in the program are formed in accordance with the syntax of sorted predicate logic. Moreover, when we speak of an *instance* of a rule, we assume that the terms substituted for variables are of the appropriate sorts.

i These rules are motivated by the "commonsense law of inertia," according to which fluents normally are not changed by actions. The rules 1(a) allow us to apply the law of inertia in reasoning "from the past to the future": The first—when a fluent is known to be true in the past, and the second—when it is known to be false. The rules 1(b) play the same role for reasoning 'from the future to the past." The auxiliary predicate *Nonineri* is essentially an "abnormality predicate" [8]. The axioms differ from those suggested in [3] only in the use of predicate "atomic" to restrict the inertia rules to unit actions.

2. Translating v-propositions:

The translation of a v-proposition "F **after** $A_1, \ldots, A_m$" is

$Holds(F, [A_1, \ldots, A_m])$.

where $[A_1, \ldots, A_m]$ stands for the ground term

$Result(A_m, Result(A_{m-1}, \ldots, Result(A_1, S_0)\ldots))$.

3. Translating e–propositions:

The translation of an e–proposition "A **causes** F if $P_1, \ldots, P_n$" consists of the rules:

(a) *Effect Axiom*:

$$Holds(F, Result(A, s)) \leftarrow Holds(P_1, s), \ldots, Holds(P_n, s)$$

It allows us to prove that F will hold after A, if the preconditions are satisfied.

(b) *Cancellation axiom for inertia*:

$$Noninert(|F|, A, s) \leftarrow not\ \overline{Holds(P_1, s)}, \ldots, not\ \overline{Holds(P_n, s)};$$

$\overline{Holds(P_i, s)}$ is a literal complementary to $Holds(P_i, s)$. For any fluent F, $|F| = F$ and $|\neg F| = F$.

The rule disables the inertia rules (1) in the cases when F can be affected by A.

(c) *Forward Reasoning Axioms*:

$$Holds(P_i, s) \leftarrow \overline{Holds(F, s)}, Holds(F, Result(A, s))$$

The above rules justify the following form of reasoning: If the value of F has changed after performing A, then we can conclude that the preconditions were satisfied when A was performed.

(d)*Backward Reasoning Axioms*:

$$\overline{Holds(P_i, s)} \leftarrow \overline{Holds(F, Result(A, s))}, Holds(P_1, s), \ldots, Holds(P_{i-1}, s), Holds(P_{i+1}, s), \ldots, Holds(P_n, s).$$

The above rules allow us to conclude that a precondition was false from the fact that performing an action did not lead to the result described by an effect axiom, and all other preconditions were true.

The axioms above differ from those suggested in [3] only by allowing terms for compound actions. The next axioms are new. They describe how the effects of individual actions are related to the effects of these actions performed concurrently.

4. Inheritance axioms:

(a) $Holds(f, Result(a, s)) \leftarrow subsetof(b, a), Holds(f, Result(b, s)), not\ Noninherit(f, a, b, s)$

(b) $\neg Holds(f, Result(a, s)) \leftarrow subsetof(b, a), \neg Holds(f, Result(b, s)), not\ Noninherit(\overline{f}, a, b, s)$

where $Noninherit(e, a, b, s)$ means "action a does not inherit fluent literal e from subaction b in situation s." According to these axioms compound actions normally inherit the effects of their components.

The next collection of axioms is concerned with cancellation of Inheritance axiom. For every e–proposition A **causes** F **if** $P_1, \ldots, P_n$, we introduce a rule

(c) $$Noninherit(\overline{F}, x, y, s) \leftarrow subsetof(y, x), subsetof(A, x), \neg subsetof(A, y), not\ \overline{Holds(P_1, s)}, \ldots, not\ \overline{Holds(P_n, s)}.$$

5. Defining subsetof and atomic:

For any two actions A and B we add $subsetof(A, B)$ if $A \subseteq B$, $\neg subsetof(A, B)$ if $A \not\subseteq B$ and $atomic(A)$ if A is a singleton and $\neg atomic(A)$ otherwise.

The noninheritance axiom in 4 (c) is essential for the correct treatment of concurrent actions and is *one of the major contribution* of this paper. It may be instructive to consider several weaker forms of this axiom.

(a) Consider an e-proposition A **causes** F **if** $P_1, \ldots, P_n$ The rule

$$Noninherit(\overline{F}, A, b, s) \leftarrow subsetof(b, A), not\ \overline{Holds(P_1, s)} \ldots not\ \overline{Holds(P_n, s)}.$$

says that if preconditions of A may be satisfied then A does not inherit from its subactions.

(b) Inheriting the non-inheritance

$$Noninherit(f, a, c, s) \leftarrow Noninherit(f, b, c, s), subsetof(b, a), subsetof(c, b)$$

This rule states that if b does not inherit f from c then any superset a of b does not inherit f from c.

(c) Taking care of any inconsistency that may arise due to inheritance:

Consider a pair of e–propositions

"A **causes** F **if** $P_1, \ldots, P_n$"
"B causes $\neg F$ **if** $Q_1, \ldots, Q_m$"

The rules

$$Noninherit(F, a, A, s) \leftarrow subsetof(A, a), subsetof(B, a), \neg subsetof(B, A), not\ \overline{Holds(Q_1, s)}, \ldots, not\ \overline{Holds(Q_m, s)}$$

$$Noninherit(\neg F, a, B, s) \leftarrow subsetof(A, a), subsetof(B, a), \neg subsetof(A, B), not\ \overline{Holds(P_1, s)}, \ldots, not\ \overline{Holds(P_n, s)}.$$

say that for any action a containing A and B if preconditions of A (B) may hold than a does not inherit $\neg F$ (F) from B (A).

The following proposition guarantees that the above rules are subsumed by the axiom 4(c).

Proposition 2. Let $\Pi_1 = \pi D$ be the translation of a domain description D and let Π_2 be an extension of Π_1 by the rules of the form (a)–(c) above. Then, A is an answer set of Π_1 iff it is an answer set of Π_2. □

The following theorem is the main technical result of this paper.

Soundness Theorem. For any v-proposition P = F **after** $A_1, \ldots, A_n$ and arbitrary domain description D such that $A_1, \ldots, A_n$ is executable in any model of D then if πD entails πP, then D entails P. □

The inheritance axioms do not contribute to the incompleteness which is caused by the incompleteness of the theory of atomic actions [3].

4 Examples

Example 6 *Independent Actions*
Consider the domain description D_1 from Example 1. The translation πD_1 of this domain consists of the Inertia and Inheritance axioms and the following axioms obtained from propositions of D_1:

X1 $\neg Holds(Lifted, S_0)$ (2)

X2 $\neg Holds(Opened, S_0)$ (2)

X3 $Holds(Lifted, [Lift])$ (3.a)

X4 $Holds(Opened, [Open])$ (3.a)

X5 $Noninert(Opened, Open, s) \leftarrow$ (3.b)

X6 $Noninert(Lifted, Lift, s) \leftarrow$ (3.b)

X7 $Noninherit(\neg Opened, x, y, S) \leftarrow subsetof(y, x)$, $subsetof(\{Open\}, x)$, $\neg subsetof(\{Open\}, y)$ (4.c)

X8 $Noninherit(\neg Lifted, x, y, S) \leftarrow subsetof(y, x)$, $subsetof(\{Lift\}, x)$, $\neg subsetof(\{Lift\}, y)$ (4.c)

In Example 3 we have shown that the domain description D_1 entails the v-propositions "Lifted **after** {Lift, Open}" and "Opened **after** {Lift, Open}". Let us demonstrate that the translation of these propositions is entailed by πD_1.

Let A be an arbitrary answer set of πD_1. According to the inheritance axiom 4.a and Lemma 1, to show that $Holds(Lifted, [\{Lift, Open\}]) \in A$ it suffices to show that

(a) $Holds(Lifted, [Lift]) \in A$ while

(b) $Noninherit(Lifted, \{Lift, Open\}, \{Lift\}, S_0) \notin A$.

(a) follows immediately from X3. To prove (b) recall that D_1 is consistent (see Example 3) and therefore, by the Soundness Theorem, πD_1 has a consistent answer set. Now (b) is the immediate consequence of Lemma 1, and the fact that $Noninherit(Lifted, \{Lift, Open\}, \{Lift\}, S_0)$ does not occur in the head of any ground instance of a rule from πD_1. Similar argument can be used to show that πD_1 entails $Holds(Opened, [\{Lift, Open\}])$.

In the following example we show how our formalism handles the case when the effect of a compound action cancels the effect of the atomic actions.

Example 7 *Dependent Actions: Cancellation*
Consider the domain description D_2 of Example 2. It is easy to see that D_2 entails

(a) $\neg$ *Spilled* **after** $\{Lift\_r, Lift\_l\}$.

πD_2 entails the translation of (a) since it contains the rule

$Holds(\neg Spilled, [\{Lift\_r, Lift\_l\}]) \leftarrow$

obtained from e–proposition

$\{Lift\_l, Lift\_r\}$ **causes** $\neg Spilled$

from D_2.

To see why the Inheritance axiom does not cause inconsistency by inheriting, say, $Holds(Spilled, [\{Lift\_l\}])$ as the result of the action [{Lift_l, Lift_r}] it suffices to notice that the rule

$$Noninherit(Spilled, \{Lift\_l, Lift\_r\}, Lift\_l, s) \leftarrow subsetof(Lift\_Left, \{Lift\_l, Lift\_r\}), subsetof(\{Lift\_l, Lift\_r\}, \{Lift\_l, Lift\_r\}), \neg subsetof(\{Lift\_r, Lift\_l\}, Lift\_l)$$

belongs to πD_2 as a ground instance of 4.c., therefore $Noninherit(Spilled, \{Lift\_l, Lift\_r\}, Lift\_l, s)$ is entailed by πD_2 for any situation s.

The effects of compound actions are cancelled in essentially the same way. Consider a domain description $D_{2.1}$ obtained from D_2 by adding an e–proposition

$\{Flip, Lift\_r, Lift\_l\}$ **causes** *Spilled*.

$\pi D_{2.1}$ will contain the rules:

$$Holds(Spilled, [\{Flip, Lift\_r, Lift\_l\}])$$
$$Noninherit(\neg Spilled, x, y, s) \leftarrow subsetof(y, x), subsetof(\{Flip, Lift\_r, Lift\_l\}, x), \neg subsetof(\{Flip, Lift\_r, Lift\_l\}, y)$$ (4.c)

These rules entail

$$Noninherit(\neg Spilled, \{Flip, Lift\_r, Lift\_l\}, \{Lift\_r, Lift\_l\}, s)$$

which blocks the inheritance axioms (4.b) and hence $\{Flip, Lift\_r, Lift\_l\}$ does not inherit "$\neg Spilled$" from $\{Lift\_r, Lift\_l\}$.

In the next example we show how to represent a compound action whose subactions have conflicting effects.

Example 8 *Conflicting Subactions*
Consider the domain description D_3 of Example 4. Since for any state σ the transition function Φ of D_3 is undefined on $(\{Open, Close\}, \sigma)$ the effect of performing "*Close*" and "*Open*" concurrently is unknown. Accordingly, no information about the state $[\{Open, Close\}]$ is entailed by πD_3. To show that this is indeed the case let us notice that the following rules belong to πD_3:

Z1 $\neg Holds(Opened, [\{Close\}])$ (3.a)

Z2 $Holds(Opened, [\{Open\}])$ (3.a)

Z3 $Noninherit(\neg Opened, x, y, s) \leftarrow subsetof(y, x)$, $subsetof(\{Open\}, x)$, $\neg subsetof(\{Open\}, y)$ (4.c)

Z4 $Noninherit(Opened, x, y, s) \leftarrow subsetof(y, x)$, $subsetof(\{Close\}, x)$, $\neg subsetof(\{Close\}, y)$ (4.c)

By instantiating Z4 with $x = \{Open, Close\}$ and $y = \{Open\}$ we obtain the clause:

$Noninherit(Opened, \{Open, Close\}, \{Open\}, s)$

Similarly, from Z3 we obtain the clause

$Noninherit(\neg Opened, \{Open, Close\}, \{Close\}, s)$

Hence, neither $Holds(Opened, [\{Open, Close\}])$ nor $\neg Holds(Opened, [\{Open, Close\}])$ can be derived from πD_3 using (4.b). From Lemma 1 and consistency of πD_3 we can conclude that it is unknown if "Opened" holds or does not hold in $[\{Open, Close\}]$. Notice however, that our program entails $Holds(Painted, [\{Open, Close, Paint\}])$ – a translation of the v–proposition *Painted* after $\{Open, Close, Paint\}$ not entailed by D. (since $\{Open, Close, Paint\}$ is not executable in models of D). This explains the executability condition in the Soundness Theorem.

5 Relation to other work.

The language *Ac* and the translation of domain descriptions in this language builds on the ideas from [3]. The treatment of concurrency in the language of situation calculus follows the lines suggested in [4j. The use of the syntax and semantics of ELP's instead of predicate calculus and circumscription allows us to come up with a more complete and computationally superior system of axiom. Another recent paper addressing the possibility of expressing the results of concurrent actions in situation calculus is [7]. The precise relationship between the two approaches is yet to be investigated. The important difference is again in the choice of the formalisms - the nonmonotonic approach of [7] seems to require combining two different non-monotonic formalisms - circumscription and default logic. In contrast our approaches use single formalisms of domain descriptions or that of ELP's. There are some other differences: for instance, in Example 8 expanded by a v-proposition "initially *Open*", the formalism of Lin and Shoham uses inertia to entail *Holds(Open, {Open.Close})* while we believe that "unknown" (produced by our systems) is the more intuitive answer. The nice feature of Lin and Shoham's formalization is so called epistemological completeness of their system [6]. Intuitively, a theory of a (deterministic) action is epistemologically complete if, given a complete description of the initial situation, the theory enables us to predict a complete description of the resulting situation when the action is performed. Since some of our actions are not executable we can not expect to have precisely this property but it is possible to suitably modify the notion and show that both our formalisms are epistemologically complete w.r.t. executable actions. This will be done in the full version of this paper, in which we will also elaborate on our treatment of non-executable and unknown actions.

Acknowledgement

We would like to acknowledge the grants NSF-IRI-92-11-662, NSF-CDA 90-15-006 and NSF-IRI 91-03-112. We also thank V. Lifschitz, G. Kartha and the anonymous referees for their valuable comments.

References

[I] Andrew Baker. Nonmonotonic reasoning in the framework of situation calculus. *Artificial Intelligence,* 49:5-23, 1991.

[2] Michael Gelfond and Vladimir Lifschitz. Logic programs with classical negation. In David Warren and Peter Szeredi, editors, *Logic Programming: Proc. of the Seventh Int'l Conf,* pages 579-597, 1990.

[3] Michael Gelfond and Vladimir Lifschitz. Representing actions in extended logic programs. In *Joint International Conference and Symposium on Logic Programming.,* 1992.

[4] Michael Gelfond, Vladimir Lifschitz, and Arkady Rabinov. What are the limitations of the situation calculus? In Robert Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe.* Kluwer Academic, Dordrecht, 1991.

[5] G. Kartha. Soundness and Completeness Theorems to Three Formalizations of Actions. IJCAI 93.

[6] Fangzhen Lin and Yoav Shoham. Provably correct theories of actions: preliminary report. In *Proc. of AAAI-91,* 1991.

[7] Fangzhen Lin and Yoav Shoham. Concurrent actions in the situation calculus. In *Proc. of AAAI-92,* pages 590-595, 1992.

[8] John McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence,* 26(3):89-116, 1986.

[9] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence,* volume 4, pages 463-502. Edinburgh University Press, Edinburgh, 1969.

[10] David Pearce and Gerd Wagner. Reasoning with negative information 1 - strong negation in logic programming. Technical report, Gruppe fur Logic, Wissentheorie and Information, Freie Universitat Berlin, 1989.

[II] Edwin Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In Ronald Brachman, Hector Levesque, and Raymond Reiter, editors, *Proc. of the First InVI Conf on Principles of Knowledge Representation and Reasoning,* pages 324-332,1989.

[12] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence,* 13(1,2):81-132,1980.

[13] Raymond Reiter. The frame problem in the situation calculus. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation,* pages 359-380. Academic Press, 1991.