From Causal Theories to Successor State Axioms

Bridging the Gap Between Nonmonotonic Action Theories and STRIPS-Like Formalisms

Area: Representation Formalisms – Action

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

In this paper we describe a system that can be used to generate action effect specifications. Unlike those like STRIPS commonly used in AI planning, our system uses an action description language that allows one to specify the effects of actions using domain rules, which are state constraints that can entail new action effects from old ones. Declaratively, an action domain in our language corresponds to a nonmonotonic causal theory in the situation calculus. Procedurally, such an action domain is compiled into a set of propositional theories, one for each action in the domain, from which fully instantiated successor state-like axioms and STRIPS-like systems are then generated. Theoretically, we show that the procedural semantics is sound with respect to the declarative semantics. Empirically, we have successfully tested our system on many benchmark planning domains, including most of those in McDermott's library of PDDL action domains.

1 Introduction

We describe a system that can be used to generate action effect specifications from a set of domain rules and direct action effect axioms, among other things. We expect the system to be a useful tool for knowledge engineers writing action specifications for classical AI planning systems, GOLOG system [4], and other systems where formal specifications of actions are needed.

One of our motivations for building such a system is to bridge the gap between formal nonmonotonic action theories on the one hand and STRIPS-like systems on the other. For years, researchers in nonmonotonic reasoning community have been proposing solutions to the frame and ramification problem, aiming for theories of actions that are more expressive than STRIPS-like systems. Until recently, however, these theories were of theoretical interest only because of their high computational complexity. The situation has since changed substantially due to the use of causality in representing domain constraints. For instance, McCain and Turner [8] showed that a competitive planner can be built directly on top of causal action theories. In this paper, we shall describe a system that takes as input a nonmonotonic action theory and returns as output a full action specification both in STRIPS-like format and as a set of successor state axioms.

The main difference between nonmonotonic action theories and STRIPS-like systems is in the former's use of domain constraints in deriving the indirect effects of actions. Specifying the effects of actions using domain constraints is like "engineering from first principle", and has many advantages. First of all, constraints are action independent, and work on all actions. Secondly, if the effects of actions derived from domain constraints agree with one's expectation, then this will be a good indication that one has axiomatized the domain correctly. Finally, domain constraints can be used for other purposes as well. For instance, they can be used to check the consistency of the initial situation database. In general, when a set of sentences violates a domain constraint, we know that no legal situation can satisfy this set of sentences. This idea can and has been used in planning to prune impossible states. Recently, there are even efforts at "reverse engineering" state constraints from STRIPS-like systems [10, 2], and use them in planning.

This paper is organized as follows. In section 2, we introduce an action domain description language which is in many ways similar to an $\mathcal{A}$-like language of Gelfond and Lifshitz [1]. A user describes an action domain in this language and submit it as the input to the system which will compile it into a

complete set of successor state axioms from which a STRIPS-like description similar to Pednault's ADL is then extracted. Section 3 describes a procedural semantics for this language, section 4 defines a formal semantics and proves that the procedural semantics defined in section 3 is sound under the formal semantics. Section 5 reports some experimental results, and section 6 discusses some related work, especially the work by Wilkins on causal reasoning in SIPE [9]. Section 7 makes some final remarks and points to directions of future work.

2 An action description language

Due to space limitation, we cannot describe the language in its full detail here. Rather, we shall illustrate it by the well-known blocks world.

The following lines (1) - (18) define a blocks world with three blocks (in the following, variables x, y, and z are assumed to be universally quantified, see section 4):

$$(block, \{1,2,3\}), \tag{1}$$
$$Fluent(on(x,y), block(x) \wedge block(y)), \tag{2}$$
$$Fluent(ontable(x), block(x)), \tag{3}$$
$$Complex(clear(x), block(x)), \tag{4}$$
$$Defined(clear(x), \neg\exists(y, block)on(y,x)), \tag{5}$$

$$Causes(on(x,y) \wedge x \neq z, \neg on(z,y)), \tag{6}$$
$$Causes(on(x,y) \wedge y \neq z, \neg on(x,z)), \tag{7}$$
$$Causes(on(x,y), \neg ontable(x)), \tag{8}$$
$$Causes(ontable(x), \neg on(x,y)), \tag{9}$$

$$Action(stack(x,y), block(x) \wedge block(y) \wedge x \neq y), \tag{10}$$
$$Precond(stack(x,y), ontable(x) \wedge clear(x) \wedge clear(y)), \tag{11}$$
$$Effect(stack(x,y), true, on(x,y)), \tag{12}$$

$$Action(unstack(x,y), block(x) \wedge block(y) \wedge x \neq y), \tag{13}$$
$$Precond(unstack(x,y), clear(x) \wedge on(x,y)), \tag{14}$$
$$Effect(unstack(x,y), true, ontable(x)), \tag{15}$$

$$Action(move(x,y,z), \\ block(x) \wedge block(y) \wedge block(z) \wedge x \neq y \wedge x \neq z \wedge y \neq z), \tag{16}$$

$$Precond(move(x,y,z), on(x,y) \wedge clear(x) \wedge clear(z)), \quad (17)$$
$$Effect(move(x,y,z), true, on(x,z)), \quad (18)$$

where

- Line (1) is an example of *type definitions*. It defines a type called *block* whose domain is the set $\{1,2,3\}$.
- Lines (2) and (3) are examples of *primitive fluent definitions*. Line (2) defines a binary (primitive) fluent called *on* which, under the type definition (1), yields the following set of fluent constants:

  $$\{on(1,2), on(1,2), on(1,3), on(2,1), on(2,2), on(2,3), on(3,1), on(3,2), on(3,3)\}.$$

  Line (3) is interpreted similarly.
- Lines (4) and (5) together is an example of *complex fluent definitions*. These are fluents that are defined in terms of primitive fluents. In this case, line (4) defines the syntax of the complex fluent *clear*, and line (5) defines its semantics. In line (5), $\exists(y, block)on(y,x)$ stands for $(\exists y).block(y) \wedge on(y,x)$. Under line (1), it will be expanded to:

  $$Defined(clear(1), \neg(on(1,1) \vee on(2,1) \vee on(3,1))),$$
  $$Defined(clear(2), \neg(on(1,2) \vee on(2,2) \vee on(3,2))),$$
  $$Defined(clear(3), \neg(on(1,3) \vee on(2,3) \vee on(3,3))).$$

- Lines (6) - (9) are examples of *domain rules*. In general, domain rules are specified by expressions of one of the following forms:

  $$Causes(\varphi, f(x_1, ..., x_n)),$$
  $$Causes(\varphi, \neg f(x_1, ..., x_n)),$$

  where f is a primitive fluent, and φ a fluent formula[1] that has no other unbound variables than those in $x_1, ..., x_n$. The intuitive meaning of a domain rule is that in any situation, if φ holds, then the fluent $f(x_1, ..., x_n)$ will be true as well. A domain rule is stronger than material implication. Its formal semantics is given by mapping it to a causal rule in [5], thus the name "causes" in it.

[1] A fluent formula is one that is constructed from fluents (both primitive and complex) and equalities.

- Lines (10), (13), and (16) are examples of *action definitions*. For instance, line (10) defines a binary action called *stack*. Under line (1), it generates the following set of action constants:

  $\{stack(1,2), stack(1,3), stack(2,1), stack(2,3), stack(3,1), stack(3,2)\}$.

- Lines (11), (14), and (17) are examples of *action precondition definitions*. For instance, line (11) says that for the action $stack(x,y)$ to be executable in a situation, $clear(x)$, $clear(y)$, and $ontable(x)$ must be true in it.
- Lines (12), (15), and (18) are examples of *action effect specifications*. In general, action effects are specified by expressions of one of the following forms:

  $$Effect(a(x_1,...,x_n), \varphi, f(y_1,...,y_k)),$$
  $$Effect(a(x_1,...,x_n), \varphi, \neg f(y_1,...,y_k)),$$

  where f is a primitive fluent, and φ a fluent formula that has no other unbound variables than those in $x_1,...,x_n,y_1,...,y_k$. The intuitive meaning of these expressions is that if φ is true in the initial situation, then action $a(x_1,...,x_n)$ will cause $f(y_1,...,y_k)$ to be true (false).

2.1 Action domain descriptions

While not applicable to the blocks world, in general, an action domain description can also include static proposition definitions and domain axioms. The former are for propositions that are not changed by any actions in the domain, and the latter are constraints about these static propositions. For instance, in the robot navigation domain, we may have a static proposition called $connected(d, r_1, r_2)$ meaning that door d connects rooms r_1 and r_2. The truth value of this proposition cannot be changed by the navigating robot which just rolls from rooms to rooms, but we may have a constraint on it saying that if d connects r_1 and r_2, then it also connects r_2 and r_1.

The following definition sums up our action description language:

Definition 1 *An* action domain description *is a set of type definitions, primitive fluent definitions, complex fluent definitions, static proposition definitions, domain axioms, action definitions, action precondition definitions, action effect specifications, and domain rules.*

3 A procedural semantics

Given an action domain description $\mathcal{D}$, we use the following procedure called CCP (a Causal Completion Procedure) to generate a complete action effect specifications:

1. Use primitive and complex fluent definitions to generate all fluents. In the following let $\mathcal{F}$ be the set of fluents so generated.
2. Use action definitions to generate all actions, and for each action A do the following:

   2.1. For each primitive fluent $F \in \mathcal{F}$, collect all A's positive effect about it:

   $$\mathit{Effect}(A, \varphi_1, F), \cdots, \mathit{Effect}(A, \varphi_n, F),$$

   all A's negative effect about it:

   $$\mathit{Effect}(A, \phi_1, \neg F), \cdots, \mathit{Effect}(A, \phi_m, \neg F),$$

   all positive domain rules about it:

   $$Causes(\varphi'_1, F), \cdots, Causes(\varphi'_k, F),$$

   all negative domain rules about it:

   $$Causes(\phi'_1, \neg F), \cdots, Causes(\phi'_l, \neg F),$$

   and generate the following pseudo successor state axiom for F:

   $$\begin{aligned} succ(F) \equiv\ & init(\varphi_1) \vee \cdots \vee init(\varphi_n) \vee succ(\varphi'_1) \vee \cdots \vee succ(\varphi'_l) \vee \\ & init(F) \wedge \neg[init(\phi_1) \vee \cdots \vee init(\phi_m) \vee succ(\phi'_1) \vee \cdots \vee succ(\phi'_k)], \end{aligned}$$

   where for any fluent formula φ, $init(\varphi)$ is the formula obtained from φ by replacing every fluent f in it by $init(f)$, and similarly $succ(\varphi)$ is the formula obtained from φ by replacing every fluent f in it by $succ(f)$. Intuitively, $init(f)$ means that f is true in the initial situation, and $succ(f)$ that f is true in the successor situation of performing the action A in the initial situation.

   2.2. Let $Succ$ be the set of pseudo successor state axioms generated from last step, $Succ1$ the following set of axioms:

   $$\begin{aligned} Succ1 = \{ & succ(F) \equiv succ(\varphi) \mid \\ & \mathit{Defined}(F, \varphi) \text{ is a complex fluent definition}\} \end{aligned}$$

and $Init$ the following set of axioms:

$$\begin{aligned} Init = \{&\varphi \mid Axiom(\varphi) \text{ is a domain axiom}\} \cup \\ &\{init(\varphi) \supset init(F) \mid Causes(\varphi, F) \text{ is a domain rule}\} \cup \\ &\{init(\varphi) \supset \neg init(F) \mid Causes(\varphi, \neg F) \text{ is a domain rule}\} \cup \\ &\{init(F) \equiv init(\varphi) \mid Defined(F, \varphi) \text{ is a complex fluent definition}\} \cup \\ &\{init(\phi_A) \mid Precond(A, \phi_A) \text{ is the precondition definition for } A\}. \end{aligned}$$

For each fluent F, if there is a formula Φ_F such that

$$Init \cup Succ \cup Succ1 \models succ(F) \equiv \Phi_F,$$

and Φ_F does not mention propositions of the form $succ(f)$, then output the axiom $succ(F) \equiv \Phi_F$. Otherwise, the action A's effect on F is indeterminate, so output the following two axioms: $succ(F) \supset \alpha_F$, and $\beta_F \supset succ(F)$, where α_F should be as strong as possible, and and β_F as weak as possible (see Section 5).

Conceptually, step 2.1 in the above procedure is most significant. In the next section, we shall prove that this step is provably correct under a translation to situation calculus causal theories of Lin [5]. Computationally, step 2.2 is most expensive.

Example 1 Consider the blocks world description in Section 2. Steps 1 and 2 use fluent and action definitions to generate all fluent and action constants. Steps 2.1 and 2.1 are then carried out for each action. For instance, for action $stack(1,2)$, we have:

2.1. For $on(1,2)$, there is one effect axiom: $Effect(stack(1,2), true, on(1,2))$, and five causal rules:

$$\begin{aligned} &Causes(on(1,1), \neg on(1,2)),\ Causes(on(1,3), \neg on(1,2)), \\ &Causes(on(2,2), \neg on(1,2)),\ Causes(on(3,2), \neg on(1,2)), \\ &Causes(ontable(1), \neg on(1,2)). \end{aligned}$$

Therefore step 2.1 generates the following pseudo-successor state axiom for $on(1,2)$:

$$\begin{aligned} succ(on(1,2)) \equiv\ & true \vee \\ & init(on(1,2)) \wedge \neg[succ(on(1,1)) \vee succ(on(1,3)) \vee \\ & \quad succ(on(2,2)) \vee succ(on(3,2) \vee succ(ontable(1))]. \end{aligned}$$

Pseudo-successor state axioms for other primitive fluents are generated similarly.

2.2. We then "solve" these pseudo-successor state axioms, and generate fully instantiated successor state axioms such as

$$succ(on(1,1)) \equiv false,\ succ(on(1,2)) \equiv true,\ succ(on(1,3)) \equiv false.$$

Once we have a set of these fully instantiated successor state axioms, we can generate the following STRIPS-like description:

```
Action stack(1, 2)          Action stack(1, 3)          ...
  Preconditions:              Preconditions:
    ontable(1)                  ontable(1)
    clear(1)                    clear(1)
    clear(2)                    clear(3)
  Add list:                   Add list:                 ...
    on(1, 2)                    on(1,3)
  Delete list:                Delete list:
    ontable(1)                  ontable(1)
    clear(2)                    clear(3)
  Conditional effects:        Conditional effects:
  Indeterminate effects:      Indeterminate effects:    ...
```

We have the following remarks:

- Although we generate the axiom $succ(on(1,3)) \equiv false$ for $stack(1,2)$, we do not put $on(1,3)$ into its delete list. This is because we can deduce $init(on(1,3)) \equiv false$ from $Init$ as well. A fluent is put into the add or the delete list of an action only if this fluent's truth value is definitely changed by the action.
- As one can see, our CCP procedure crucially depends on the fact that each type has a finite domain so that all reasoning is done at the propositional level. This is a limitation of our current system, and this limitation is not as bad as one might think. First of all, typical planning problems all assume finite domains, and changing the domain of a type in an action description is easy - all one need to do is to change the corresponding type definition. More significantly, a generic action domain description can often be obtained from one that assumes a finite domain. In our blocks world example, the numbers "1", "2", and "3" are generic names, and can be replaced by parameters. For instance, if we replace "1" by x and "2" by y in the above STRIPS-like description of $stack(1,2)$, we will get a STRIPS-like description for $stack(x,y)$ that works for any x and y. We have found that this is a strategy that always works in planning domains.

4 Formal semantics

The formal semantics of an action domain description is defined by a translation into a situation calculus causal theory in [5]. We first briefly review the language of the situation calculus.

4.1 Situation calculus

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 propositional fluents, *truth-value* for truth values $true$ and $false$, and *object* for everything else. We use the following domain independent predicates and functions:

- Binary function do - for any action a and any situation s, $do(a, s)$ is the situation resulting from performing a in s.
- Binary predicate $Holds$ - for any p and any situation s, $Holds(p, s)$ is true if p holds in s.
- Binary predicate $Poss$ - for any action a and any situation s, $Poss(a, s)$ is true if a is possible (executable) in s.
- 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.

4.2 A translation to the situation calculus

Let $\mathcal{D}$ be an action domain description. The translation of $\mathcal{D}$ into a situation calculus theory is defined as follows:

- A type definition like $(block, \{1, 2, 3\})$ is translated to:

  $$1 \neq 2 \neq 3 \wedge (\forall x).block(x) \equiv (x = 1 \vee x = 1 \vee x = 3).$$

  A primitive fluent definition like $Fluent(ontable(x), block(x))$ is translated to

  $$(\forall x)Fluent(ontable(x)) \equiv block(x).$$

  An action definition like $Action(stack(x, y), block(x) \wedge block(y) \wedge x \neq y)$ is translated to

  $$(\forall x, y).Action(stack(x, y)) \equiv block(x) \wedge block(y) \wedge x \neq y.$$

- An action precondition axiom $Precond(a(x_1, ..., x_n), \varphi)$ is translated to

$$(\forall \vec{x}, s).Action(a(x_1, ..., x_n)) \supset [Poss(a(x_1, ..., x_n), s) \equiv H(\varphi, s)],$$

where $H(\varphi, s)$ is the formula obtained from φ by replacing fluent atom f in it by $H(f, s)$. Notice that no other unbound variables except those in $\vec{x} = (x_1, ..., x_n)$ can occur in φ.

- An action effect axiom: $Effect(a(x_1, ..., x_n), \varphi, f(y_1, ..., y_k))$ is translated to

$$\begin{aligned}(\forall \vec{x}, \vec{y}).Action(a(x_1, ..., x_n)) \wedge Fluent(f(y_1, ..., y_k)) \supset \\ (\forall s).Poss(a(x_1, ..., x_n), s) \wedge H(\varphi, s) \supset \\ Caused(f(y_1, ..., y_k), true, do(a(x_1, ..., x_n), s)),\end{aligned}$$

Similar translation is done for an effect axiom of the form

$$Effect(a(x_1, ..., x_n), \varphi, \neg f(y_1, ..., y_k)).$$

- A domain rule of the form $Causes(\varphi, f(x_1, ..., x_n))$ is translated to

$$(\forall \vec{x}).Fluent(f(x_1, ..., x_n)) \supset (\forall s).H(\varphi, s) \supset Caused(f(x_1, ..., x_n), true, s),$$

Similar translation is done for a domain rule of the form

$$Causes(\varphi, \neg f(x_1, ..., x_n)).$$

The translation of complex fluents, static propositions, and domain axioms is straightforward, and is omitted here.

Now given an action domain description $\mathcal{D}$, let $\mathcal{T}$ be its translation in the situation calculus. The semantics of $\mathcal{T}$ is then determined by its completion $comp(\mathcal{T})$ which is defined as the set of following sentences (see [5] for more details):

1. The circumscription of $Caused$ in $\mathcal{T}$ with all other predicates fixed.
2. For each primitive fluent F, the following generic successor state axiom:

$$\begin{aligned}\forall a, s.Poss(a, s) \supset H(F, do(a, s)) \equiv \\ [Caused(F, true, do(a, s)) \vee H(F, s) \wedge \neg Caused(F, false, do(a, s))].\end{aligned}$$

3. The unique names assumptions for fluent, action names, and truth values, the basic axiom about causality which says that if a fluent is caused to have certain truth value, then it must be of that truth value, and the *foundational axioms* in [7] for the discrete situation calculus.

The following theorem shows that the procedural semantics given in the previous section is sound with respect to semantics given here.

Theorem 1 *Let $\mathcal{D}$ be an action domain description, and T its translation in the situation calculus. If the procedure CCP in the previous section outputs an axiom, then this axiom, when translated into the situation calculus, is entailed by comp(T): for any action A, and fluent F,*

- *if it outputs $succ(F) \equiv \Phi_F$, then*

$$comp(\mathcal{T}) \models \forall s.Poss(A,s) \supset H(F, do(A,s)) \equiv \Phi_F[s],$$

  where $\Phi_F[s]$ is the formula obtained from Φ_F by replacing every $init(f)$ in it by $H(f,s)$.

- *if it outputs $succ(F) \supset \alpha_F$ and $\beta_F \supset succ(F)$, then*

$$comp(\mathcal{T}) \models \forall s.Poss(A,s) \wedge H(F, do(A,s)) \supset \alpha_F[s],$$
$$comp(\mathcal{T}) \models \forall s.Poss(A,s) \wedge \beta_F[s] \supset H(F, do(A,s)).$$

5 Summary of experimental results

Except for step 2.2, the procedure CCP in section 3 is straightforward to implement. What step 2.2 does is to determine, for each proposition of the form $succ(F)$, whether it can be defined in terms of propositions of the form $init(p)$. If yes, we want an explicit definition, and if not, we want two most general implications: $succ(F) \supset \alpha_F$ and $\beta_F \supset succ(F)$. As it turned out, α_F and β_F are what we have called elsewhere [6] the *strongest necessary condition* and *weakest sufficient condition* of $succ(F)$, respectively, and they are also the key in determining whether we can have a successor state axiom for F. As one may suspect, these two conditions are in general expensive to compute, but there are some strategies that work particularly well in the action domains, see [6].

The implemented system in SWI-Prolog has been successfully tested on many benchmark planning domains including most of the domains in McDermott's collection of action domains in PDDL. Some of the them are: the

blocks world, a scheduling domain that includes Pednault's dictionary and paycheck domain as a special case, the rocket domain, the SRI robot domain, the machine shop assembling domain, the ferry domain, the grid domain, the sokoban domain, and the gear domain. The following is a list of some of the common features:

- Our specifications of these benchmark planning domains seem very natural. In particular, in all cases, it is quite straightforward to decide what effects of an action should be encoded as direct effects (those given by the predicate *Effect*) and what effects as indirect effects (those derived from domain rules).
- The most common domain rules are functional dependency constraints. For instance, in the blocks world, the fluent $on(x, y)$ is functional on both arguments; in the logistics domain, the fluent $at(object, loc)$ is functional on the second argument (each object can be at only one location). It makes sense then that we should have a special shorthand for these domain rules, and perhaps a special procedure for handling them as well. But more significantly, given the prevalent of these functional dependency constraints in action domains, it is worthwhile to investigate the possibility of a general purpose planner making good uses of these constraints.
- Our system is basically propositional. The generated successor state axioms and STRIPS-like systems are all fully instantiated. However, in all the domains that we have tried, it is quite straightforward for the user to generalize these propositional specifications to first-order ones, as we have illustrated it for the blocks world.

6 Related work

In terms of the action description language, the most closely related work is $\mathcal{A}$-like languages, particularly the language $\mathcal{C}$ [3]. On the one hand, our language is more restrictive than $\mathcal{C}$ in that it does not provide facilities for expressing the truth value of a fluent in a particular, for instance the initial, situation. It is a language for specifying the generic effects of actions. On the other hand, it has more facilities, such as types and static propositions, than the $\mathcal{C}$ language.

In planning, the most closely related work is the causal reasoning module in Wilkins's SIPE system [9]. Wilkins remarked (page 85, [9]): "Deductive

causal theories are one of the most important mechanisms used by SIPE to alleviate problems in operator representation caused by the STRIPS assumption." Unfortunately, none of the more recent planning systems have anything like SIPE's causal reasoning module. In SIPE, domain rules have triggers, preconditions, conditions, and effects. Informally, when the triggers become true in the new situation, SIPE would then check in sequence to see if the preconditions were true in the old situation, and the conditions are true in the new situation. If all these conditions are true, it will then deduce the effects. For instance, the following is a SIPE causal rule in the blocks world:

```
Causal-rule: Not-on
Arguments: x, y, z;      Trigger: on(x,y);
Precondition: on(x,z);   Effects: not on(x,z);
```

In comparison, our domain rules are much simpler. For instance, our corresponding rule for the above one is simply: $Causes(on(x,y) \wedge y \neq z, on(x,z))$. We do not need procedural directives like triggers. To a large degree, we can see our system as a rational reconstruction of the causal reasoning module in SIPE. As we have shown in Theorem 1, the procedure used by our system is sound under a translation to causal theories in the situation calculus. While Wilkins also gave a translation of his causal rules to formulas in the situation calculus, he did not specify an underlying logic to reason about such formulas. In fact, as shown in [5], a translation like Wilkins' is unlikely to work.

7 Concluding remarks

We have described a system for generating the effects of actions from direct action effect axioms and domain rules, among other things. We have shown the soundness of the procedure used by the system and tested it successfully on many benchmark action domains. We plan to address the following issues in the future:

- As we have mentioned, functional dependency constraints are the most prevalent domain rules in the action domains that we have experimented with. It would be interesting to see if a planner can make use of this fact.
- While our system can handle actions with indeterminate effects, we have not yet tried it on large examples. What we would like to do is to experiment it with some practical examples, and more importantly, to design a planner that can handle indeterminate actions.

References

[1] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.

[2] A. Gerevini and L. Schubert. Inferring state constraints for domain-independent planning. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI–98), AAAI Press, Menlo Park, CA.*, 1998.

[3] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI–98), AAAI Press, Menlo Park, CA.*, 1998.

[4] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming, Special issue on Reasoning about Action and Change*, 31:59–84, 1997.

[5] F. Lin. Embracing causality in specifying the indirect effects of actions. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI–95), IJCAI Inc. Distributed by Morgan Kaufmann, San Mateo, CA.*, pages 1985–1993, 1995.

[6] F. Lin. On strongest necessary and weakest sufficient conditions. In *http://www.cs.ust.hk/faculty/flin*, 1999. Submitted.

[7] F. Lin and R. Reiter. State constraints revisited. *Journal of Logic and Computation, Special Issue on Actions and Processes*, 4(5):655–678, 1994.

[8] N. McCain and H. Turner. Satisfiability planning with causal theories. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 212–221, 1998.

[9] D. Wilkins. *Practical planning: extending the classical AI planning paradigm.* Morgan Kaufmann, San Mateo, CA, 1988.

[10] Y. Zhang and N. Foo. Deriving invariants and constraints from action theories. *Fundamenta Informaticae*, 30(1):109–123, 1997.