A System for Generating Action Specifications

Fangzhen Lin (flin@cs.ust.hk)
Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Draft of July 12, 1999

Abstract

In this paper we describe a system that can be used to generate a set of action specifications. Given a set of action effect axioms, domain rules, and some other axioms, the system computes for each action all of its effects, both those given explicitly by the effect axioms and those followed indirectly from the domain rules. The resulting action effect specifications are outputed in two forms: a complete set of fully instantiated successor state axioms and a STRIPS-like format similar to Pednault's ADL. We show that the procedure used by the system is sound under a translation to the situation calculus causal theories of (Lin [5]). This system can be used as a front-end to classical AI planners, Golog system, or any other system where a formal specification of the effects of actions is needed.

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 action effects are needed.

The system addresses some computational issues of the frame and ramification problems in reasoning about actions. In short, this problem is about how to reason about the effects of actions given a set of direct action effects and domain constraints. In the absence of domain constraints, this problem has been addressed using a frame-like system called STRIPS by Fikes and

Nilsson [1], and in the situation calculus by Pednault [9], Schubert [11], and Reiter [10]. The problem becomes more difficult and tricky when there are domain constraints. First of all, there is a question of what exactly these domain constraints are and how they should be represented. For this, we shall follow some of the more recent work in the area of formal theories of actions. The key idea here is that domain constraints as well as direct action effects should be written in a causal language. Specifically, the system that we shall describe in this paper will be based on causal situation calculus theories of actions proposed by Lin [5]. Basically, what it does is to compile causal theories in [5] into a format that is more readily usable by systems such as GOLOG and classical AI planners.

While domain constraints make the problem more difficult, they are desirable because they lead to a more principled way of axiomatizing an action domain. First of all, they are action independent, and work on all actions. Secondly, if the effects of actions derived from domain constraints conform with one's expectations, it 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 collection of facts violates a domain constraint, we know that no legal situation can satisfy these facts simultaneously. This idea can and has be used in planning to prune impossible states.

In a sense, our system can be regarded as a rational reconstruction of the causal reasoning module in Wilkins's SIPE system [12], the only planning system that we know of that uses domain constraints to generate the effects of actions. While our system is a stand-alone one, SIPE's causal reasoning model is embedded in the whole planning system, and it is not obvious how it can be used by other systems. More significantly, SIPE's causal rules are too complex and lack a declarative semantics. We shall elaborate these differences later in the paper.

Compared with the situation calculus, our system makes two simplifications: it assumes only two generic situations - the initial situation and the successor situation of doing a given action in the initial situation, and it currently works only in the propositional case, i.e. it assumes the number of fluents to be finite. With respect to action specifications commonly used in classical AI planning and by GOLOG, the first simplification does not seem to present any problem. The second simplification is more problematic, and is a limitation of our current system. But the limitation is not as bad as one would think at first glance, as we shall see later.

In the following, we first introduce an action domain description language which is in many ways similar to an $\mathcal{A}$-like language of Gelfond and Lifshitz [2]. 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.

2 An action description language

Our action description language include the following components.

2.1 Type definitions

A type definition is specified by expressions of the following form:

$$(p, \{a_1, ..., a_n\}),$$

where p is the type name, and $a_1, ..., a_n$ are all the objects belonging to this type. For instance, in the blocks world, we may have a type called *block*, and have, say, five blocks named numerically: $(block, \{1, 2, 3, 4, 5\})$. In a logistics domain, we may have a type called *loc* for locations, and have, say, 3 locations l_1, l_2, and l_3: $(loc, \{l_1, l_2, l_3\})$. We remark here that our current system requires the domain of a type to be finite and explicitly enumerated, in order to keep things at propositional level.

2.2 Primitive fluent definitions

Primitive fluents are defined by expressions of the following form:

$$Fluent(f(x_1, ..., x_n), p_1(x_1) \wedge \cdots \wedge p_n(x_n) \wedge e_1 \wedge \cdots \wedge e_m),$$

where f is a n-ary fluent name, p_i, $1 \leq i \leq n$, a type name, and e_i, $1 \leq i \leq m$, is $x_j \neq x_k$, for some $1 \leq j \leq k \leq n$. The intuitive meaning of this expression is that $f(x_1, ..., x_n)$ is a fluent if the second argument is true. For instance, in the blocks world, given the type definition $(block, \{1, 2, 3\})$, the following fluent specification:

$$Fluent(on(x, y), block(x) \wedge block(y) \wedge x \neq y)$$

would generate the following six fluents:

$$on(1, 2), on(1, 3), on(2, 1), on(2, 3), on(3, 1), on(3, 2).$$

Clearly, there should be only one fluent definition for each fluent.

2.3 Complex fluent definitions

Given a set of primitive fluents, one may want to define some new ones. For instance, in the blocks world, given the primitive fluent *on*, we can define *clear* in terms of *on* as: $(\forall x) clear(x) \equiv \neg(\exists y) on(y, x)$.

To specify complex fluents like *clear*, we first define *fluent formulas* as follows:

- $t_1 = t_2$ is a fluent formula, where t_1 and t_2 are terms, i.e. either a constant in the domain of a type or a variable.
- $f(t_1, ..., t_n)$ is a fluent formula, where $t_1, ..., t_n$ are terms, and f is either an n-ary primitive fluent, complex fluent, or static proposition (see below).
- If φ and φ' are fluent formula, then $\neg\varphi$, $\varphi \vee \varphi'$, $\varphi \wedge \varphi'$, $\varphi \supset \varphi'$, and $\varphi \equiv \varphi'$ are also fluent formulas.
- If φ is a fluent formula, x a variable, and p a type name, then $\forall(x, p)\varphi$ (for all x of type p, φ holds) and $\exists(x, p)\varphi$ (for some x of type p, φ holds) are fluent formulas. Notice that since we require types to have finite domains, so these quantifications are really shorthands.

A complex fluent is then specified in our language by a pair of expressions of the following form:

$$Complex(f(x_1, ..., x_n), p_1(x_1) \wedge \cdots p_n(x) \wedge e_1 \cdots \wedge e_m),$$
$$Defined(f(x_1, ..., x_n), \varphi),$$

where p_i's are types, e_i's equality literals, and φ is a fluent formula that does not mention any complex fluents and whose free variables are among $x_1, ..., x_n$. The first expression specifies the syntax and the second the semantics of the complex fluent.

For instance, the complex fluent *clear* in the blocks world can be specified as:

$$Complex(clear(x), block(x)),$$
$$Defined(clear(x), \neg\exists(y, block) on(y, x)).$$

As we mentioned above, quantifiers here are just shorthands because each type must have a finite domain. For instance, given the following specification:

$$(block, \{1, 2, 3\}),$$
$$Fluent(on(x, y), block(x) \wedge block(y))$$

the above fluent definition for *clear* will be expanded to:

$$\begin{aligned}&Defined(clear(1), \neg(on(1,1) \vee on(2,1) \vee on(3,1))),\\&Defined(clear(2), \neg(on(1,2) \vee on(2,2) \vee on(3,2))),\\&Defined(clear(3), \neg(on(1,3) \vee on(2,3) \vee on(3,3))).\end{aligned}$$

2.4 Static proposition definitions

A static proposition is one that is not changed by any action in the domain. For instance, in the robot navigation domain, we may have a proposition $connected(d, r1, r2)$ meaning that door d connects rooms $r1$ and $r2$. The truth value of this proposition cannot be changed by the navigating robot that just rolls from rooms to rooms.

In our language, a static proposition is defined by an expression of the following form:

$$Static(g(x_1, ..., x_n), p_1(x_1) \wedge \cdots \wedge p_n(x_n) \wedge e_1 \wedge \cdots \wedge e_m),$$

where g is a n-ary predicate, p_i, $1 \leq i \leq n$, a type name, and e_i, $1 \leq i \leq m$, is $x_j \neq x_k$, for some $1 \leq j \leq k \leq n$. The meaning of this expression is similar to a fluent definition, and there should only be one definition for each static proposition.

2.5 Domain axioms

Domain axioms are constraints on static propositions. For instance, for the static proposition $connected(d, r1, r2)$, we may want to impose the following constraint: $connected(d, r1, r2) \equiv connected(d, r2, r1)$. In our language, domain axioms are specified by expressions of the form:

$$Axiom(\varphi),$$

where φ is a fluent formula that does not mention any fluents, i.e. it mentions only static propositions and equality. For instance, the above constraint on *connected* is written as:

$$\begin{aligned}&Axiom(\forall(d, door)\forall(r_1, room)\forall(r_2, room)connected(d, r_1, r_2) \equiv\\&\quad connected(d, r_2, r_1)),\end{aligned}$$

where *door* and *room* are types.

2.6 Action definitions

Actions are defined by expressions of the following form:

$$Action(a(x_1, ..., x_n), p_1(x_1) \wedge \cdots \wedge p_n(x_n) \wedge e_1 \wedge \cdots \wedge e_m),$$

where a is an n-ary action name, and p's and e's are the same as in primitive fluent definitions. For instance, in the blocks world, given the type definition $(block, \{1, 2, 3\})$, the following action specification:

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

would generate the following six actions:

$$stack(1, 2), stack(1, 3), stack(2, 1), stack(2, 3), stack(3, 1), stack(3, 2).$$

There should be only one action definition for each action.

2.7 Action precondition definitions

Action precondition definitions are specified by expressions of the following form:

$$Precond(a(x_1, ..., x_n), \varphi),$$

where a is an n-ary action name, φ is a fluent formula whose free variables are among $x_1, ..., x_n$.

There should be only one precondition definition for each action. For instance, in the blocks world, we may have:

$$Precond(stack(x, y), clear(x) \wedge clear(y) \wedge ontable(x)),$$

which 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.

2.8 Action effect specifications

Action effects are specified by expressions of the following form:

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

or of the form:

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

where φ is a fluent formula, and f a primitive fluent. 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). For instance, in the blocks world, action $stack(x, y)$ causes x to be on y:

$$Effect(stack(x, y), true, on(x, y)).$$

For an example of context dependent effect, consider action $drop(x)$ that break an object only if it is fragile:

$$Effect(drop(x), fragile(x), broken(x)).$$

2.9 Domain rules

Domain rules are specified by expressions of the following form:

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

or of the following form:

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

where φ is a fluent formula, and f a primitive fluent. 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 (Lin [5]), thus the name "causes" in it. For instance, in the blocks world, a block can only be on one other block:

$$Causes(on(x, y) \wedge y \neq z, \neg on(x, z)).$$

In a logistic domain, if a package is inside a truck which is at location l, then the package is at location l as well:

$$Causes(in(x, y) \wedge at(y, l), at(x, l)).$$

2.10 Action domain descriptions

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.*

Example 1 The following action domain description defines a blocks world with three blocks:

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

$$Causes(on(x,y) \land x \neq z, \neg on(z,y)),$$
$$Causes(on(x,y) \land y \neq z, \neg on(x,z)),$$
$$Causes(on(x,y), \neg ontable(x)),$$
$$Causes(ontable(x), \neg on(x,y)),$$

$$Action(stack(x,y), block(x) \land block(y) \land x \neq y),$$
$$Precond(stack(x,y), ontable(x) \land clear(x) \land clear(y)),$$
$$Effect(stack(x,y), true, on(x,y)),$$

$$Action(unstack(x,y), block(x) \land block(y) \land x \neq y),$$
$$Precond(unstack(x,y), clear(x) \land on(x,y)),$$
$$Effect(unstack(x,y), true, ontable(x)),$$

$$Action(move(x,y,z), block(x) \land block(y) \land block(z) \land x \neq y \land x \neq z \land y \neq z),$$
$$Precond(move(x,y,z), on(x,y) \land clear(x) \land clear(z)),$$
$$Effect(move(x,y,z), true, on(x,z)).$$

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.

3. For each primitive fluent $F \in \mathcal{F}$, collect all A's positive effect about it:
$$Effect(A, \varphi_1, F), \cdots, Effect(A, \varphi_n, F),$$

all A's negative effect about it:

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

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:

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

where for any fluent formula φ, $init(\varphi)$ is the formula obtained from φ by replacing every fluent f in φ by $init(f)$, and similarly $succ(\varphi)$ is the formula obtained from φ by replacing every fluent f in φ 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 action A in the initial situation.

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

$$Succ1 = \{succ(F) \equiv succ(\varphi) | Defined(F, \varphi) \text{ is a complex fluent definition}\}$$

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. In this case, output two axioms:

$$succ(F) \supset \alpha_F,$$
$$\beta_F \supset succ(F),$$

where α_F is a strongest formula satisfying the first implication, and β_F a weakest formula satisfying the second implication.

Conceptually, step 2 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, the last step is most expensive. We shall describe the strategies that our system uses to implement it in section 5.

Example 2 Consider the blocks world description in Example 1. The set of fluents generated by step 1 is:

$$\begin{aligned}\mathcal{F} = \{&on(1,1), on(1,2), on(1,3), on(2,1), on(2,2), on(2,3),\\ &on(3,1), on(3,2), on(3,3), clear(1), clear(2), clear(3),\\ &ontable(1), ontable(2), ontable(3)\}.\end{aligned}$$

Step 2 generates the following actions:

$$\begin{aligned}&stack(1,2), stack(1,3), stack(2,1), stack(2,3), stack(3,1), stack(3,2),\\ &unstack(1,2), unstack(1,3), unstack(2,1), unstack(2,3), unstack(3,1), unstack(3,2),\\ &move(1,2,3), move(1,3,2), move(2,1,3), move(2,3,1), move(3,1,2), move(3,2,1)\end{aligned}$$

For each of these actions, we need to go through the remaining two steps. For instance, for $stack(1,2)$, there is only one effect axiom about $on(1,2)$:

$$Effect(stack(1,2), true, on(1,2)),$$

and the following causal rules about $on(1,2)$:

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

Therefore step 3 generates the following pseudo-successor state axiom for $on(1,2)$:

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

We can similarly generate the following pseudo-successor state axioms for other primitive fluents:

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

For complex fluent *clear*, its definition yields following axioms:

$$
\begin{aligned}
succ(clear(1)) &\equiv \neg succ(on(1,1)) \wedge \neg succ(on(2,1)) \wedge \neg succ(on(3,1)), \\
succ(clear(2)) &\equiv \neg succ(on(1,2)) \wedge \neg succ(on(2,2)) \wedge \neg succ(on(3,2)), \\
succ(clear(3)) &\equiv \neg succ(on(1,3)) \wedge \neg succ(on(2,3)) \wedge \neg succ(on(3,3)).
\end{aligned}
$$

We can then "solve" these pseudo-successor state axioms and generate the following successor state axioms:

$$
\begin{aligned}
succ(on(1,1)) &\equiv false, \\
succ(on(1,2)) &\equiv true, \\
succ(on(1,3)) &\equiv false, \\
succ(on(2,1)) &\equiv false, \\
succ(on(2,2)) &\equiv false,
\end{aligned}
$$

$$
\begin{aligned}
&succ(on(3,1)) \equiv false,\\
&succ(on(3,2)) \equiv false,\\
&succ(ontable(1)) \equiv false,\\
&succ(on(2,3)) \equiv init(on(2,3)),\\
&succ(on(3,3)) \equiv init(on(3,3)),\\
&succ(ontable(2)) \equiv init(ontable(2)),\\
&succ(ontable(3)) \equiv init(ontable(3)),\\
&succ(clear(1)) \equiv init(clear(1)),\\
&succ(clear(2)) \equiv false,\\
&succ(clear(3)) \equiv init(clear(3)).
\end{aligned}
$$

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.

In the last section, we introduce fluent formulas. We now extend H to these formulas: for any fluent formula φ and situation s, $H(\varphi,s)$ is defined as follows:

- $H(t_1 = t_2, s)$ is $t_1 = t_2$.

- if P is a static proposition, then $H(P, s)$ is P.
- if F is a fluent, either primitive or complex, then $H(F, s)$ stands for itself.
- inductively, $H(\neg\varphi, s)$ is $\neg H(\varphi, s)$, $H(\varphi \vee \varphi', s)$ is $H(\varphi, s) \vee H(\varphi', s)$, and similarly for other connectives.
- inductively, $H(\forall(x,p)\varphi, s)$ is $\forall s.p(x) \supset H(\varphi, s)$ and $H(\exists(x,p)\varphi, s)$ is $\exists x.p(x) \wedge H(\varphi, s)$.

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 $(p, \{a_1, ..., a_k\})$ is translated to:

$$(\forall x).p(x) \equiv (x = a_1 \vee \cdots \vee x = a_k),$$
$$a_1 \neq a_2 \neq \cdots \neq a_k.$$

- a primitive fluent definition

$$\mathit{Fluent}(f(x_1, ..., x_n), p_1(x_1) \wedge \cdots \wedge p_n(x_n) \wedge e_1 \wedge \cdots \wedge e_m)$$

  is translated to

$$(\forall x_1, ..., x_n).\mathit{Fluent}(f(x_1, ..., x_n)) \equiv p_1(x_1) \wedge \cdots \wedge p_n(x_n) \wedge e_1 \wedge \cdots \wedge e_m.$$

- a complex fluent definition

$$\begin{aligned}&\mathit{Complex}(f(x_1, ..., x_n), p_1(x_1) \wedge \cdots p_n(x) \wedge e_1 \cdots \wedge e_m),\\ &\mathit{Defined}(f(x_1, ..., x_n), \varphi),\end{aligned}$$

  is translated to

$$\begin{aligned}&(\forall x_1, ..., x_n).\mathit{Fluent}(f(x_1, ..., x_n)) \equiv p_1(x_1) \wedge \cdots p_n(x) \wedge e_1 \cdots \wedge e_m,\\ &(\forall x_1, ..., x_n, s).\mathit{Fluent}(f(x_1, ..., x_n)) \supset [H(f(x_1, ..., x_n), s) \equiv H(\varphi, s)].\end{aligned}$$

- a domain axiom about static propositions:

$$\mathit{Axiom}(\varphi)$$

  is translated to φ with quantifiers in it treated as shorthands:

$$\forall(x,p)\phi = \forall s.p(x) \supset \phi,$$
$$\exists(x,p)\phi = \exists x.p(x) \wedge \phi.$$

- an action definition

$$Action(a(x_1,...,x_n), p_1(x_1) \wedge \cdots \wedge p_n(x_n) \wedge e_1 \wedge \cdots \wedge e_m)$$

  is translated to

$$(\forall x_1,...,x_n).Action(a(x_1,...,x_n)) \equiv p_1(x_1) \wedge \cdots \wedge p_n(x_n) \wedge e_1 \wedge \cdots \wedge e_m.$$

  We assume here that the domain description has only one action definition for each action.

- An action precondition axiom

$$Precond(a(x_1,...,x_n), \varphi)$$

  is translated to

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

  where $\vec{\xi}$ is the list of all free variables in $a(x_1,...,x_n)$ and φ. We mentioned earlier that one of the limitations of our current system is that action preconditions have to be given explicitly. This is reflected in the above translation.

- An action effect axiom:

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

  is translated to

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

  where $\vec{\xi}$ is the list of all free variables in $a(x_1,...,x_n)$, $f(y_1,...,y_k)$, and, φ.

  Similarly, an effect axiom

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

  is translated to

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

- A domain rule of the form

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

is translated to

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

where $\vec{\xi}$ is the list of all free variables in $f(x_1, ..., x_n)$ and φ. Similarly, a domain rule of the form

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

is translated to

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

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 $comp(\mathcal{T})$ that is defined as the set of following sentences:

1. The circumscription of $Caused$ in T with all other predicates fixed.

2. The following basic axioms about $Caused$ that says that if a fluent is caused to be true (false), then it is true (false):

$$Caused(p, true, s) \supset Holds(p, s), \tag{1}$$
$$Caused(p, false, s) \supset \neg Holds(p, s). \tag{2}$$

3. For the truth values, the following unique names and domain closure axiom:

$$true \neq false \wedge (\forall v)(v = true \vee v = false). \tag{3}$$

4. The unique names assumptions for fluent and action names. Specifically, if F_1,..., F_n are all the fluent names, then we have:

$$F_i(\vec{x}) \neq F_j(\vec{y}), \ i \text{ and } j \text{ are different},$$
$$F_i(\vec{x}) = F_i(\vec{y}) \supset \vec{x} = \vec{y}.$$

Similarly for action names.

5. For each primitive fluent F, the following generic successor state axiom:

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

6. The *foundational axioms* in [7] for the *discrete* situation calculus. These axioms characterize the structure of the space of situations. For the purpose of this paper, it is enough to mention that they include the following unique names axioms for situations:

$$s \neq do(a, s),$$
$$do(a, s) = do(a', s') \supset (a = a' \wedge s = s').$$

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 $succ(F) \equiv \Phi_F$ in step 4, 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*

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

Proof: This is proved by showing that the axioms used to generate outputs in step 4 are entailed by $comp(\mathcal{T})$:

- if $Precond(A, \phi_A)$ is the precondition axiom for A in $\mathcal{D}$, then

  $$comp(\mathcal{T}) \models (\forall s).Poss(A, s) \equiv H(\phi_A, s).$$

- if $Causes(\varphi, F)$ is a domain rule in $\mathcal{D}$, then

  $$comp(\mathcal{T}) \models (\forall s).H(\varphi, s) \supset H(F, s),$$

  and if $Causes(\varphi, \neg F)$ is a domain rule in $\mathcal{D}$, then

  $$comp(\mathcal{T}) \models (\forall s).H(\varphi, s) \supset \neg H(F, s).$$

- for any fluent f, if $succ(f) \equiv \Psi_f$ is a pseudo-successor state axiom in $Succ$, then

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

  where $\Psi_f[A,s]$ is the formula obtained by replacing formulas of the form $init(f')$ in Ψ_f by $H(f',s)$, and formulas of the form $succ(f')$ in Ψ_f by $H(f', do(A,s))$.

These follows from the way expressions in $\mathcal{D}$ is translated to sentences in $\mathcal{T}$ and the fact the minimizing $Caused$ in $\mathcal{T}$ entails, for each fluent f,

$$Caused(f, true, do(A,s)) \equiv H(\varphi_1, s) \vee \cdots H(\varphi_n, s) \vee H(\varphi'_1, do(A,s)) \vee \cdots H(\varphi'_l, do(A,s)),$$
$$Caused(f, false, do(A,s)) \equiv H(\phi_1, s) \vee \cdots H(\phi_m, s) \vee H(\phi'_1, do(A,s)) \vee \cdots H(\phi'_k, do(A,s)),$$

where φ_i, φ'_i, ϕ_i and ϕ'_i are as in step 3 in CCP, but for f (it was for F in the procedure). ■

While we have not been able to prove it, we believe that the converse of this theorem holds as well, i.e. the procedure CCP is also complete.

5 An implementation

Except for step 4, the procedure CCP in section 3 is straightforward to implement. This section describes the strategy that our system uses for implementing step 4.

Given an action A, as in step 4, let $Succ$ be the set of pseudo-successor state axioms for primitive fluents, $Succ1$ the set of pseudo-successor state axioms for complex fluents, and $Init$ the set of initial situation axioms derived from action precondition axiom for A, domain axioms, domain rules, and complex fluent definitions. Our strategy for implementing step 4 is as follows (in the following, a $succ$ proposition is one of the form $succ(f)$, and an $init$-proposition is one of the form $init(f)$):

1. Transform $Init$ into a clausal form and derive from it a set of unit clauses $Unit$.

2. Use $Unit$ to simplify the axioms in $Succ$ and for each resulting axiom in it:

   $$succ(f) \equiv \Phi_f, \tag{4}$$

   if Φ_f does not mention $succ$-propositions, then delete it from $Succ$, output it and replace $succ(f)$ in the rest of the axioms by Φ_f.

3. For each remaining axiom (4) in $Succ$, if Φ_f has the form $init(f) \wedge ...$ (a candidate of a frame axiom), then check to see if $succ(f)$ can be derived from $Succ$, $Unit$, and $init(f)$ by unit resolution. If so, delete it from $Succ$, output $succ(f) \equiv init(f)$, and replace $succ(f)$ in $Succ$ by $init(f)$.

4. As a last resort, for each remaining (4) in $Succ$, compute a strongest necessary condition α_f of $succ(f)$ on the $init$-propositions under the theory $Init \cup Succ$ (see [6]), a weakest sufficient condition β_f of $succ(f)$ on the $init$-propositions under the theory $\{\alpha_f\} \cup Init \cup Succ$. If β is a tautology, then delete (4) from $Succ$, output $succ(f) \equiv \alpha_f$, and replace $succ(f)$ in $Succ$ by α_f. If β is not a tautology, then output $succ(f) \supset \alpha_f$ and $\alpha_f \wedge \beta_f \supset succ(f)$, but do not delete (4) from $Succ$. The correctness of this step can be found in (Lin [6]).

5. The previous steps would solve the equations in $Succ$, and generate appropriate output for primitive fluents. For each complex fluent F:

$$Defined(F, \varphi),$$

   if every primitive fluent in φ has a successor state axiom, then do the following:

   (a) if no primitive fluents in φ are changed by the action, then this complex fluent is not changed by the action either, so output $succ(F) \equiv init(F)$;

   (b) otherwise, output $succ(F) \equiv \Phi$, where Φ is obtained from $succ(\varphi)$ by replacing every $succ$-proposition in it by the right side of its successor state axiom.

   Otherwise, if some of the primitive fluents in φ do not have a successor state axiom, which means the action may have an indeterminate effect on them, then this action may have an indeterminate effect on F as well. Compute the strongest necessary and weakest sufficient conditions of $succ(F)$ under $Init \cup Succ \cup Succ1$ as in last step, and output them.

Step 4, and step 5 for complex fluents, are the bottleneck of this implementation strategy as in the worst case, computing the strongest necessary condition of a proposition is coNP-hard. However, it has been our experience that if action A has a context-free effect on fluent F, then its successor state axiom can be computed without using step 4.

We have implemented the procedure CCP using the above strategy in SWI-Prolog[1]. The following are some experimental results of our implementation.

6 Some experimental results

6.1 The meet-pass domain

This simple domain was adapted from McDermott's collection of PDDL action domains [8]. In this domain, there are trains traveling on tracks, and some of the tracks are connected. The problem is to schedule trains in such a way that at any time, no track will have more than one trains traveling on it. Assume that we have two trains and three tracks:

$$\begin{array}{l}(train, \{a, b\}),\\ (track, \{1, 2, 3\}).\end{array}$$

There is one primitive fluent: $at(x, y)$ - train x is on track y, one complex fluent: $clear(x)$ - track x is clear, i.e. no trains on it, and one static proposition: $connected(x, y)$ - tracks x and y are connected,

$$\begin{array}{l}Fluent(at(X, Y), train(X) \wedge track(Y)),\\ Complex(clear(X), track(X)),\\ Defined(clear(X), \neg\exists(y, track) at(y, X)),\\ Static(connected(X, Y), track(X) \wedge track(Y) \wedge X \neq Y.\end{array}$$

We can identify the following domain rules:

$$\begin{array}{l}Causes(at(X, Y) \wedge Z \neq X, \neg at(Z, Y)),\\ Causes(at(X, Y) \wedge Z \neq Y, \neg at(X, Z)).\end{array}$$

There is one type of action: $move(x, y, z)$ - move train x from track y to track z, and is specified by the following action definition, action precondition, and action effect axioms:

$$\begin{array}{l}Action(move(X, Y, Z), track(Y) \wedge track(Z) \wedge Y \neq Z \wedge train(X)),\\ Precond(move(X, Y, Z), at(X, Y) \wedge clear(Z) \wedge connected(Y, Z)),\\ Effect(move(X, Y, Z), true, at(X, Z)).\end{array}$$

For this domain, our system generates 9 fluents and 12 actions. For each action, it generates both a complete set of fully instantiated successor

[1] SWI-Prolog is developed by Jan Wielemaker at University of Amsterdam

state axioms and a STRIPS-like representation. For instance, for action $move(a, 1, 2)$ (move train a from track 1 to track 2), the STRIPS-like representation looks like:

```
move(a, 1, 2):

  Preconditions: at(a, 1), clear(2), connected(1, 2).
  Add list: clear(1), at(a, 2).
  Delete list: clear(2), at(a, 1).
```

The complete set of axioms is:

```
move(a, 1, 2):

succ(clear(1)) <-> true
succ(at(a, 2)) <-> true
succ(at(a, 3)) <-> false
succ(at(b, 1)) <-> false
succ(at(b, 2)) <-> false
succ(clear(2)) <-> false
succ(at(a, 1)) <-> false
succ(clear(3)) <-> init(clear(3))
succ(at(b, 3)) <-> init(at(b, 3))
```

The total computation time for all actions is 103K inferences and 2 seconds.[2] This is a pure STRIPS domain, i.e. all actions are context free. For this domain, as we mentioned earlier, step 4 in our implementation procedure is not needed, and step 5 is easy.

6.2 The monkey and banana domain

This domain is also adapted from McDermott's collection of PDDL action domains, which attributes it to University of Washington's UCPOP collection of action domains, which in turn attributes it to Prodigy. While some of the action effects generated by our system are context-dependent, they are all context-free in the other systems. We shall elaborate on this difference later.

In this domain, there are two types, loc for locations (we assume there are three locations here), and $object$ for things like monkey, banana, box, etc.:

$$(loc, \{1, 2, 3\}),$$
$$(object, \{monkey, box, banana, knife, glass, fountain\}).$$

[2]All times in this paper refers to CPU times on a IBM ThinkPad 315ED with Pentium 166MMX and 32MB RAM.

The following are fluent definitions:

$$\begin{aligned}
&Fluent(onFloor),\\
&Fluent(at(M,X), loc(X) \wedge object(M)),\\
&Fluent(hasknife),\\
&Fluent(onbox(X), loc(X)),\\
&Fluent(hasbanana),\\
&Fluent(haswater),\\
&Fluent(hasglass).
\end{aligned}$$

The following are domain rules about these fluents:

$$Causes(onbox(X), at(monkey, X)), \tag{5}$$
$$Causes(onbox(X), at(box, X)), \tag{6}$$
$$Causes(onbox(X), \neg onFloor), \tag{7}$$
$$Causes(onFloor, \neg onbox(X)), \tag{8}$$
$$Causes(at(M,X) \wedge X \neq Y, \neg at(M,Y)), \tag{9}$$
$$Causes(hasglass \wedge at(monkey, X), at(glass, X)), \tag{10}$$
$$Causes(hasknife \wedge at(monkey, X), at(knife, X)), \tag{11}$$
$$Causes(hasbanana \wedge at(monkey, X), at(banana, X)). \tag{12}$$

The following are action definitions along with their respective preconditions and effect axioms:

- $goto(x, y)$ - the monkey goes to x from y:

$$\begin{aligned}
&Action(goto(X,Y), loc(X) \wedge loc(Y) \wedge X \neq Y),\\
&Precond(goto(X,Y), at(monkey, Y) \wedge onFloor),\\
&Effect(goto(X,Y), true, at(monkey, X)).
\end{aligned}$$

- $climb(X)$ - the monkey climbs onto the box at location X:

$$\begin{aligned}
&Action(climb(X), loc(X)),\\
&Precond(climb(X), at(box, X) \wedge onFloor \wedge at(monkey, X)),\\
&Effect(climb(X), true, onbox(X)).
\end{aligned}$$

- $pushbox(X, Y)$ - the monkey pushes the box from Y to X.

$$\begin{aligned}
&Action(pushbox(X,Y), loc(X) \wedge loc(Y) \wedge X \neq Y),\\
&Precond(pushbox(X,Y), at(monkey, Y) \wedge at(box, Y)) \wedge onFloor),\\
&Effect(pushbox(X,Y), true, at(monkey, X)),\\
&Effect(pushbox(X,Y), true, at(box, X)).
\end{aligned}$$

- $getknife(X)$ - get knife at location X.

  $Action(getknife(X), loc(X))$,
  $Precond(getknife(X), at(knife, X) \wedge at(monkey, X) \wedge \neg hasknife)$,
  $Effect(getknife(X), true, hasknife)$.

- $getbanana(X)$ - grab banana at loc X, provided the money is on the box.

  $Action(getbanana(X), loc(X))$,
  $Precond(getbanana(X), onbox(X) \wedge at(banana, X) \wedge \neg hasbanana)$,
  $Effect(getbanana(X), true, hasbanana)$.

- $pickglass(X)$ - pick up glass at loc X.

  $Action(pickglass(X), loc(X))$,
  $Precond(getwater(X), at(fountain, X) \wedge onbox(X) \wedge hasglass \wedge \neg haswater)$,
  $Effect(getwater(X), true, haswater)$.

- $getwater(X)$ - get water from fountain at loc X, provided the monkey is on the box, and has a glass in hand.

  $Action(getwater(X), loc(X))$,
  $Precond(pickglass(X), at(glass, X) \wedge at(monkey, X) \wedge \neg hasglass)$,
  $Effect(pickglass(X), true, hasglass)$.

This domain has 27 actions and 26 fluents. Again, for each action, our system generates both a complete set of fully instantiated successor state axioms and a STRIPS-like representation. For instance, for action $goto(1, 2)$, the following is the STRIPS-like representation generated by the system:

```
Action goto(1, 2)

Preconditions: at(monkey, 2), onFloor

Add list: at(monkey, 1)

Delete list: at(monkey, 2)

Conditional effects:
```

```
succ(at(banana, 1)) <-> init(hasbanana) \/ init(at(banana, 1))
succ(at(knife, 1)) <-> init(hasknife) \/ init(at(knife, 1))
succ(at(glass, 1)) <-> init(hasglass) \/ init(at(glass, 1))
succ(at(banana, 2)) <-> - init(hasbanana) & init(at(banana, 2))
succ(at(knife, 2)) <-> - init(hasknife) & init(at(knife, 2))
succ(at(glass, 2)) <-> - init(hasglass) & init(at(glass, 2))
```

The total running time for all actions is 68 seconds while performing 18.5 million inferences. About 90 percent of time is spent on step 4, i.e. on computing the strongest necessary and weakest sufficient conditions of fluents on which the given action has a context-dependent effects. For instance, for action $goto(1,2)$ above, the majority of time was spent on generating the above 6 conditional effects.

We mentioned earlier that the specifications for this domain given in McDermott's collection as well as other are all context-free. For instance, the following is a specification for action *goto* in PDDL in McDermott's collection:

```
(:action GO-TO
   :parameters (?x ?y)
   :precondition (and (location ?x) (location ?y)
                   (not (= ?y ?x)) (on-floor) (at monkey ?y))
   :effect (and (at monkey ?x) (not (at monkey ?y))))
```

This corresponds to a context-free action that does not change any other fluent except *at*. It is clear that the design of this action does not take into account domain rules (10) - (12). With this specification, if initially banana is at location 1, then the goal of having banana at location 2 would not be achievable.

6.3 Some other domains and summary

We have axiomatized in our language many other 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 monkey and banana 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 would 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.

- As we mentioned earlier, our system is 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. In this sense, it's best to regard this system as a semi-automatic generator of action domain specifications.

7 Related work

In terms of the action description language, the most closely related work is Gelfond and Lifschitz's $\mathcal{A}$-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 [12]. Wilkins writes (page 85, [12]): "Use of the STRIPS assumptions has made operators unacceptably difficult to describe in previous classical planners... One of the primary reasons for this is that all effects of an action must be explicitly stated... Deductive causal theories are one of the most important mechanisms used by SIPE to alleviate problems in operator representation caused by the STRIPS assumption." This is certainly one of the motivations for our system as well. 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, a SIPE causal rule for $on(x, y)$ in the blocks world would look like:

```
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 demonstrated in [5], such translations would not work.

8 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 in many benchmark action domains used by current AI planners. We plan to address the following issues in the future:

- As we mentioned, in many cases, it is easy to generalize the output of the system into first order case. It would be interesting to see if a general theorem can be proved to confirm this formally.
- As we also mentioned, functional dependency constraints are the most prevalent domain rules in the action domains that we have experimented. 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 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] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to theorem proving in problem solving. *Artificial Intelligence*, 2:189–208, 1971.

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

[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. Strongest necessary conditions, weakest sufficient conditions, and their applications. In *Draft*, 1999.

[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] D. McDermott et al. (AIPS-98 Planning Competition Committee). PDDL – the planning domain definition language. Technical Report Tech Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[9] E. P. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pages 324–332. Morgan Kaufmann Publishers, Inc., 1989.

[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] L. K. Schubert. Monotonic solution to the frame problem in the situation calculus: an efficient method for worlds with fully specified actions. In H. Kyberg, R. Loui, and G. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer Academic Press, Boston, MA, 1990.

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