From Causal Theories to Successor State Axioms and STRIPS-Like Systems

Abstract

We describe a system for specifying the effects of actions. Unlike those 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. We expect the system to be a useful tool for knowledge engineers writing action specifications for classical AI planning systems, GOLOG systems, and other systems where formal specifications of actions are needed.

Introduction

We describe a system for generating 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 systems [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 fully instantiated successor state axioms.

The main difference between nonmonotonic action theories and STRIPS-like systems is in the use of domain constraints by the former 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 [11, 2], and use them in planning.

We begin by introducing 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 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.

An action description language

Essentially, an action domain in our system is specified by a set of domain constraints, and for each action, an action precondition axiom and some direct effect axioms. There are a variety of syntax for writing such an action domain description. Since our system is implemented in Prolog, we choose a format that is most

Prolog-friendly.

The following lines (1) - (12) define a blocks world with three blocks (in the following, variables x, y, and z are assumed to be universally quantified, see the section on formal semantics):

$$domain(block, \{1,2,3\}), \quad (1)$$
$$Fluent(on(x,y), block(x) \wedge block(y)), \quad (2)$$
$$Fluent(ontable(x), block(x)), \quad (3)$$
$$Complex(clear(x), block(x)), \quad (4)$$
$$Defined(clear(x), \neg\exists(y, block) on(y,x)), \quad (5)$$
$$Causes(on(x,y) \wedge x \neq z, \neg on(z,y)), \quad (6)$$
$$Causes(on(x,y) \wedge y \neq z, \neg on(x,z)), \quad (7)$$
$$Causes(on(x,y), \neg ontable(x)), \quad (8)$$
$$Causes(ontable(x), \neg on(x,y)), \quad (9)$$
$$Action(stack(x,y), block(x) \wedge block(y) \wedge x \neq y) (10)$$
$$Precond(stack(x,y), ontable(x) \wedge clear(x) \wedge clear(y)), \quad (11)$$
$$Effect(stack(x,y), true, on(x,y)). \quad (12)$$

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*. For instance, under the type definition (1), (2) yields the following set of fluent constants: $\{on(1,2), on(1,2), on(1,3), on(2,1), on(2,2), on(2,3), on(3,1), on(3,2), on(3,3)\}$.
- 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 [6], thus the name "causes" in it.
- Line (10) defines a binary action called *stack*, and line (11) defines the precondition of this action: for the action $stack(x,y)$ to be executable in a situation, $clear(x)$, $clear(y)$, and $ontable(x)$ must be true in it. We can similarly define other actions in the blocks world, such as *unstack*.
- Line (12) is an example 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).

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

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.*

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 specification:

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:[2]

$$Effect(A, \varphi_1, F), \cdots, Effect(A, \varphi_n, F),$$

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:

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

where for any fluent formula φ, $init(\varphi)$ is the formula obtained from φ as follows: (1) eliminate first all the quantifiers in φ (this is possible because each type has a finite domain); (2) eliminate all equality literals using unique names assumptions; (3) replace every fluent f in it by $init(f)$. Similarly, $succ(\varphi)$ is the formula obtained from φ by the same procedure except here each fluent f in it is replaced 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:

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

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 the section on experimental results).

[2] Notice that when A and F are ground, no variables can occur in formulas φ_i, ϕ_i, φ'_i, and ϕ'_i below.

Conceptually, step 2.1 in the above procedure is most significant. 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 2.2 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 seven causal rules:

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

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

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

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

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

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

```
 stack(1, 2)          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)
Cond. effects:       Cond. effects:
Indet. effects:      Indet. 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.
- The STRIPS-like system so generated is best considered to be a shorthand for a set of fully instantiated successor state axioms. This view is consistent with that of [5, 7].
- As one can see, our CCP procedure crucially depends on the fact that each type has a finite domain so that all reasoning can be done in propositional logic. 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 often works in planning domains.

Formal semantics

The formal semantics of an action domain description is defined by a translation into a situation calculus causal theory in [6]. The translation is quite straightforward. For instance, a domain rule of the form $Causes(\varphi, f(x_1, ..., x_n))$ is translated to

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

where $H(\varphi, s)$ is the formula obtained from φ by replacing fluent atom f in it by $H(f, s)$ which stands for that f holds in s, $Caused(f, v, s)$ is another predicate in our version of the situation calculus and stands for that the fluent f is caused (by something unspecified) to have the truth value v in situation s, and $Fluent$ is a predicate constructed from primitive fluent definitions. For instance, corresponding to a primitive fluent definition like $Fluent(ontable(x), block(x))$, we have

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

where
$block$ is a type predicate. Similarly, an action effect axiom of the form: $Effect(a(x_1, ..., x_n), \varphi, f(y_1, ..., y_k))$ is translated to

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

where $Action$ is a predicate constructed from action definitions.

We have shown that under this translation, the procedural semantics given in the previous section is sound, and, under a condition similar to Reiter's consistency condition [9], complete as well. The precise statement of these results and its proof will be given in the full paper.

Summary of experimental results

Except for step 2.2, the procedure CCP in section 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 [12] 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.

Briefly, given a theory T, a proposition q, and a set of propositions P, a formula φ is a strongest necessary condition (weakest sufficient condition) of q on P if φ is a formula of P, $T \models q \supset \varphi$ ($T \models \varphi \supset q$), and for any such φ', we have that $T \models \varphi \supset \varphi'$ ($T \models \varphi' \supset \varphi$). Although there are some strategies that work particularly well in the action domains [12], these two conditions are in general expensive to compute. Thus, our strategy for step 2.2 is to first perform some simple simplification and rewriting, and then use a general procedure for computing these two conditions as a last resort:

1. for each pseudo successor state axiom $succ(f) \equiv \varphi$, do the following: eliminate all $succ(g)$, where g is a complex fluent, in φ using $Succ1$; if under $Init$ the new φ can be simplified into a formula φ' that does not mention any *succ*-propositions, then we have a successor state axiom for f, and we replace each occurrence of $succ(f)$ by φ' in other pseudo-successor state axioms;

2. this step tries to generate frame axioms: for each remaining pseudo-successor state axiom of the form $succ(f) \equiv init(f) \wedge \varphi$ do the following: first replace all *succ*-propositions in φ using their respective pseudo-successor state axioms; if the new φ is entailed by $init(f)$ and $Init$, then we have a frame axiom: $succ(f) \equiv init(f)$, and we replace each occurrence of $succ(f)$ by $init(f)$ in other pseudo-successor state axioms;

3. for each, say f, of the primitive fluents that we do not yet have a successor state axiom: compute first the strongest necessary condition φ of $succ(f)$ on *init*-propositions; if the weakest sufficient condition ϕ of $succ(f)$ under $Init \cup \{\varphi\}$ and the remaining pseudo-successor state axioms is equivalent to $true$, then we have a successor state axiom for f: $succ(f) \equiv \varphi$; otherwise, output $succ(f) \supset \varphi$ and $\phi \wedge \varphi \supset succ(f)$.

4. finally, process complex fluents using their definitions.

By the results in [12], this is a sound and complete procedure for step 2.2 of CCP. This is so even when we use a sound but incomplete propositional theorem prover for checking whether a formula can be simplified into one without mentioning any of the *succ*-propositions (step 1 above), or whether a formula is entailed by a propositional theory (step 2 above), as long as a sound and complete procedure is used for step 3 in computing the two conditions. Indeed, in our implemented system,[3] for steps 1 and 2 above, our implemented system uses unit resolution on clauses. Perhaps a bit surprisingly, this turns out to be adequate for many of the context free actions in benchmark planning domains such as the blocks world, logistics domain, and robot navigation domain. For these context free actions, our system finds all successor state axioms even before it reaches step 3, which is the most expensive step in the above procedure. Table 1 shows the performance of our system on action $stack(1,2)$ in the blocks world, and it is representative of the context free actions that we have experimented with. In the table, the "Time" column is the CPU time in seconds on a Sparc Ultra 2 machine running SWI-Prolog, the "Rate" column is the rate of increase of the time over the previous row, and the "Rate of n^6" is the rate of increase of n^6, where n equals to the number of blocks. As one can see, the rate of CPU times conforms well with that of n^6, which is the worst time complexity of our algorithm in the blocks world for a single action: given n blocks, there are $O(n^2)$ of fluents, and for each fluent, computing its successor state axiom needs to do a closure of unit resolution which is $O(m^2)$, where m is the size of clauses, which is in $O(n^2)$. Notice that the STRIPS-like description that our system outputs for the action $stack(1,2)$ is independent of the number of blocks, and is always the one given earlier.

No. of blocks	Time	Rate	Rate of n^6
5	1.63		
6	4.43	2.72	2.98
7	10.39	2.35	2.52
8	22.42	2.16	2.23
9	44.56	1.99	2.02
10	82.51	1.85	1.88
11	142.39	1.73	1.77
12	237.57	1.67	1.69
13	380.41	1.60	1.62
14	589.98	1.55	1.56
15	893.20	1.51	1.51
16	1314.66	1.47	1.47
17	1886.66	1.44	1.44
18	2655.81	1.41	1.41

Table 1: The blocks world

In addition to the blocks world, we have also successfully applied our system to generate many other benchmark planning domains, including most of the domains in McDermott's collection of action domains in PDDL. Some of the them are: 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

[3] Implemented in SWI-Prolog.

some of the common features:

- In all of these domains, 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, it is often straightforward for the user to generalize these propositional specifications to first-order ones, as we have illustrated it for the blocks world.

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 [10]. Wilkins remarked (page 85, [10]): "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, and are interpreted procedurally. In comparison, our domain rules are much simpler, and are interpreted declaratively. To a large degree, we can see our system as a rational reconstruction of the causal reasoning module in SIPE.

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, and under certain conditions, the completeness of the procedure used by the system and tested it successfully on many benchmark action domains.

There are many directions for future work. One of them is on generalizing the propositional STRIPS-like systems generated by our system to first-order case. As we have mentioned, there are some heuristics that seem to work well in many benchmark domains. But a systematic study is clearly needed,

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 *AAAI–98*, 1998.

[3] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *AAAI–98*, 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*, 31:59–84, 1997.

[5] V. Lifschitz. On the semantics of STRIPS. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pages 1–9. Morgan Kauffmann Publishers, Inc., 1986. June 30–July 2, Timberline, Oregon.

[6] F. Lin. Embracing causality in specifying the indirect effects of actions. In *IJCAI–95*, pages 1985–1993, 1995.

[7] F. Lin and R. Reiter. How to progress a database. *Artificial Intelligence*, (92)1-2:131–167, 1997.

[8] N. McCain and H. Turner. Satisfiability planning with causal theories. In *KR'98*, pages 212–221, 1998.

[9] 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.

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

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

[12] See the paper by the same author to be presented at the upcoming KR'2000.