Action Specifications in $\{log\}$

Alessandro Provetti and Gianfranco Rossi
Dipartimento di Matematica - Università di Parma
Via D'Azeglio, 85 Parma. I-43100, Italy.
{*provetti,gianfr*}*@prmat.math.unipr.it*

Abstract

Specifying the effect of actions is a relevant application of logic programming to knowledge representation. We argue that extensions of logic programming that support sets as first-class objects make it elegant and succint to describe actions and reason about plans and plan execution. This discussion is carried out in the framework of the language $\{log\}$ of Dovier et al..

1 Introduction

Extensions of logic programming that allow sets as first-class objects have recently been made available. This is an advancement over the standard alternative of using lists to represent sets, intuitively those collections where the order of components and possible repetitions are irrelevant.

One application of logic programming that can benefit from sets is commonsense reasoning about actions. This by now is a well established subfield of Knowledge Representation which aims to describing dynamical systems, planning, plan execution and reasoning with incomplete information. Several issues of independent interest have arisen in this field so far, viz. Circumscription and the Frame problem. In particular, automated reasoning about actions stimulated work on the theory and implementation of logic programming languages, such as those of Apt and Bezem [1], Turner [14], and Lifschitz et al. [11, 12].

There is a trend in reasoning about actions —started by Gelfond and Lifschitz's introduction of $\mathcal{A}$ in [9]— of having a *surface language* for describing actions proper, and then a translation to a concrete logic programming language, for which deduction mechanisms already exist. $\mathcal{A}$ and its followers[1] have several attractive features:

- a simple and concise syntax;
- a semantics based on the notion of automaton, which provides additional insight into the behavior of the corresponding dynamic system;
- non-commitment w.r.t. a specific set of logical connectives[2].

In this paper we discuss the question: *What is the added value of representing actions in a logic programming language equipped with set representation and manipulation facilities?*

[1] A recent developement is in [4].

[2] As [2] points out, this is desirable in view of representing and revising beliefs and in general for capturing commonsense reasoning.

The results we have had so far with the language $\{log\}$ of Dovier et al. [6] are promising. If used as a surface language, $\{log\}$ descriptions are easier to understand than corresponding list-based axiomatizations. If used for computing the consequences of an $\mathcal{A}$-style surface language, $\{log\}$ might turn out to be *more complete* than usual Prolog, i.e. certain non-ground queries now have terminating computations.

Example 1 *Consider a simple domain where A is the only action in the domain that causes fluent F to become false; in other words,* it switches F off. *In the $\mathcal{A}$ syntax, this gets written as*[3]*:*

A **causes** $\neg F$ **if** F

Although the initial situation is unknow, we would like to conclude that after executing A, F is false. In fact, in the initial situation, F is either true or false. In the former, the axiom above applies, so that we obtain $\neg F$ in the resulting state. In the latter, the axiom does not apply and –by the inertia principle– $\neg F$ is true in the resulting state also.

The two cases illustrated above are properly mirrored by the two models this theory yields under the $\mathcal{A}$ semantics. In fact, it is easy to show that

$$\{A \textbf{ causes } \neg F \textbf{ if } F\} \models_{\mathcal{A}} \neg F \textbf{ after } A$$

Gelfond and Lifschitz have defined a translation of $\mathcal{A}$ theories into extended logic programs which is proved sound. As a result, if Prolog (rather, an extension of it) is sound w.r.t. the answer set semantics then it is sound w.r.t. the $\mathcal{A}$ entailment.

Still, completeness does not hold in general even for the first step of this process and Example 1 is an immediate example of this. In fact, when the extended program obtained from translation of theory above is fed to Prolog, the interpreter is not able to derive any value of F in the initial situation. Then, causal law above is certainly not applicable since we do not have F. Inertia is not applicable, since we do not have $\neg F$, and Prolog is not able to compute the value of F after A is executed.

On the other hand, in Section 3.4 we describe a successful $\{log\}$ computation for this example, which illustrates the ability of $\{log\}$ constraint system to capture the 'reasoning with models' displayed in Example 1.

It is open whether a semantically correct and complete translation from $\mathcal{A}$ to $\{log\}$ exists; if so, this result would be in par with the Abductive logic programming solution of Denecker and De Schreye [5], who have established this result. Let us start our discussion by describing the features of $\{log\}$.

2 Overview of $\{log\}$

The language $\{log\}$ extends logic programs to allow a notation very close to standard Set theory. $\{log\}$ is an instance of the Constraint Logic Programming (CLP) scheme, and this feature supports the generation of answers qualified with constraints (see [7] for a discussion of this aspect). The main features are:

- the interpreted function symbol $\{\cdot/\cdot\}$, used to build sets;
- extended unification embedding a simple theory of sets;

[3]Refer to [9] or later works, e.g. [4, 5] on $\mathcal{A}$ for the syntax and semantics of $\mathcal{A}$.

- the predicates $=$, $\in$, $\neq$ and $\notin$ are treated as CLP constraints.

Let us describe below a fragment of $\{log\}$ which is used in the next examples.

2.1 Syntax

The standard language of logic programs is extended with the costant symbol $\{\}$ and function $\{\cdot/\cdot\}$[4]. The usual definition of well-formed term is extended by defining *set terms* as follows:

1. $\{\}$ is a set term;
2. $\{\tau_1, \tau_2, \ldots \tau_n\}$ is a set term if $\tau_1,\ \tau_2, \ldots \tau_n$ are terms;
3. $\{\tau_1/\tau_2\}$ is a set term if τ_1 is a term and τ_2 is a set term;
4. $\{\chi : \Gamma\}$ is a well-formed term (called *intensional set*) if χ is a variable appearing in Γ and Γ is any $\{log\}$ goal.

The primitive constraints in $\{log\}$ are atoms of the form $\tau = \sigma$, $\tau \neq \sigma$, $\tau \in \sigma$, and $\tau \notin \sigma$, where τ and σ are terms (in particular, set terms). Moreover, shorthand formulae of the form $forall(\chi \in \sigma, \Gamma)$, where Γ is any $\{log\}$ goal and χ is a variable appearing in Γ itself, can occur in place of atoms and are called *restricted universal quantifiers.*

2.2 Semantics

The semantics of $\{log\}$ programs is inherited from the CLP scheme, and it is formally defined in [7]. Let us proceed to informally describe what is the intended meaning of the new set-based structures.

Clearly, $\{\}$ is always interpreted as the empty set; term $\{\tau_1/\tau_2\}$ is interpreted as $\{\tau_1\} \cup \tau_2$, singleton $\{\tau\}$ is interpreted as $\{\tau\} \cup \{\}$ and so on. The intensional set $\{\chi : \pi(\tau_1, \ldots \tau_n)\}$ collects all and only the instances of χ such that $\pi(\tau_1, \ldots \tau_n)$ is true, i.e., $\sigma = \{\chi : \pi(\tau_1, \ldots \tau_n)\}$ is equivalent to $\chi \in \sigma \leftrightarrow \pi(\tau_1, \ldots \tau_n)$ is true. Similarly, a formula $forall(\chi \in \sigma, \Gamma)$ is considered equivalent to $\chi \in \sigma \rightarrow \Gamma$.

2.3 Set operations

In $\{log\}$ it is easy to define predicates like *union* and *intersection* and *complement* that capture the standard set-theoretic operations, e.g.

$$
\begin{array}{lll}
union(\{\}, S, S). & & \\
union(\{A/S\}, R, \{A/T\}) & :- & A \notin R \,\&\\
 & & union(S, R, T).\\
union(\{A/S\}, R, T) & :- & A \in R \,\&\\
 & & union(S, R, T).
\end{array}
$$

The first clause states $\emptyset \cup S = S$; the second *builds* the third parameter by adding to it the elements of the first set which do not belong to the second. To avoid repetitions in the result, those elements which belong to both sets are dropped, by the third clause.

[4] As a result, the Herbrand universe of programs containing set terms is always infinite.

It is interesting to see how the $\{log\}$ interpreter handles non-ground queries against the definition above. For instance, the answers to query

$$?- union(\{a/X\}, \{b/Y\}, R).$$

are, in order of generation:

1. $R = \{a, b/Y\},\ X = \{\}$ with constraint $a \notin Y$;
2. $R = \{a, \_A, b/Y\},\ X = \{\_A\}$ with constraint $a \notin Y, \_A \notin Y, \_A \neq b$;
3. ...

The first answer corresponds to the minimal number of elements in the first operand, while the second remains unconstrained. The further answers consist in creating increasingly larger result sets. This seems to us acceptable in view of generate & test search. Moreover, we prospet making predicate *union* into a $\{log\}$ built-in constraint, giving an equivalent –but more concise– solution to the query above, namely

n $R = \{a, b/N\}$ with constraint $union(X, Y, N$.

Let us conclude the section by showing a more concise $\{log\}$ definition of basic set operations using Restricted Universal Quantifiers and intensional sets:

$$\begin{aligned}
&subseteq(S, R) &&:- \quad forall(X \in S, X \in R).\\
&union(S, R, T) &&:- \quad T = \{X :\ X \in S \ or\ X \in R\}.\\
&intersection(S, R, T) &&:- \quad T = \{X :\ X \in S\ \&\ X \in R\}.\\
&setminus(S, R, T) &&:- \quad T = \{X :\ X \in S\ \&\ X \notin R\}.
\end{aligned}$$

3 $\{log\}$ in the actions domain

In this section we adopt the familiar framework[5] of atomic actions and fluents, and discuss $\{log\}$ specifications of the effects of single actions and for evaluating the effect of executing plans.

3.1 Representing states and action effects

Let $\mathcal{F}$ be a finite set of ground terms representing fluents and let $\mathcal{S}$ be the collection of all set terms whose members belong to $\mathcal{F}$; these sets are called *state terms,* i.e., a state is just a collection of fluents. Intuitively, a non-ground state term:

$$\{f_1, f_2/X\}$$

stands for any ground states containing *at least* f_1 and f_2. As the cardinality of a state is bounded, so is the number of possible instatiations of the set above.

The meaning we shall associate to states is clear: they represent a *state of the domain* by the collection of all fluents that are true thereof. Now, the effect of executing an action is represented by the relation

[5]The difference to Situation Calculus is the absence of situations as such.

$$effect(< start\_state >, < action >, < end\_state >)$$

between the state before and the state after its execution. By defining predicate *effect,* the effect of an alphabet of actions $\mathcal{A}$ applied to a domain described by $\mathcal{F}$ remains defined.

For instance, suppose we want to describe in $\{log\}$ the effect of action *open_door:*

$$effect(S, open\_door, \{open/S\}) :- \quad locked \notin S.$$

By the axiom above, attempting *open_door* in a state where *locked* is –intuitively– false, succeeds and brings about *open* in the resulting state. Fluents belonging to the start state, i.e. S, are clearly *carried over* to the end state $\{open/S\}$. In the case *open* already belongs to S, subsequent operations on the resulting state (e.g., set unification) will ignore the repetition.

In the case *locked* belongs to S, a *Frame axiom* is needed to say that the state of the domain remains unvaried; such axiom can be written as follows:

$$\begin{array}{ll} apply(S, A, R) :- & effect(S, A, R). \\ apply(S, A, S) :- & \{R : effect(S, A, R)\} \ = \ \{\}. \end{array}$$

The definition of *apply* above expresses the assumption that the definition of predicate *effect* is complete. Thus, no applicable effect definition corresponds to no effect at all, and the end state is equated to the start state. In reasoning with non-ground states, we expect the first axiom to be applied the most times, possibly with generation of constraints. The second axiom is used only in case the interpretation of *effect* generates conflicting constraints. For instance, the interpretation of query $? - apply(\{alive/X), open\_door, Y)$ generates the answer:

$$locked \notin X, \ Y = \{open, alive/X\}$$

where the start state is constrained, and therefore also the end state is.

3.2 Results and limitations

The predicates discussed earlier in this section allow an elegant treatment of actions whose success depends on the domain. Several types of dependence are possible.

Example 2 *Loading a gun causes the gun to be loaded. Any other fluent keeps its value.*

$$\begin{array}{lll} effect(S, load, R) & :- & loaded \notin S \ \& \\ & & R \ = \ \{loaded/S\}. \end{array}$$

Let us comment on the first atom in the body: trying to load an already loaded gun has no effect; this contingence is taken care of by the definition of 'apply' above.

Example 3 *Shooting a gun which is loaded causes the gun itself to become unloaded and death of the target, i.e., the fluent 'alive' becomes false. This case is treated with the predicate 'setminus,' which implements the usual set difference operation*[6]*.*

$$\begin{array}{lll} effect(S, shoot, R) & :- & S = \{loaded/X\} \ \& \\ & & setminus(X, \{alive\}, R). \end{array}$$

Example 4 *Shooting and unloaded gun has no effect on the state of the domain. This is taken care of by the inertia axiom in the definition of predicate 'apply.'*

[6]Simply, $setminus(X, Y, Z) \equiv Z = X \setminus Y$.

Example 5 *Shooting without knowing about the state of the gun makes fluent 'alive' become undefined, while the gun ends up unloaded in any case.*

This case is not treated properly even by means of non-ground state terms. In fact, the effect of *alive* becoming undefined should be expressed by saying that *alive* neither belongs nor does not belong to the end state. In fact, there are two models of the evolution of the system, i.e., two alternative instantiations of the end state should be considered.

An alternative solution to Example 5 is the introduction of another type of terms, that we called *double state.*

3.3 Double states

Double states are terms:

$$state(S^+, S^-)$$

where, S^+ and S^- are set terms. Double states are convenient for a three-valued approach to state description. Intuitively, S^+ is a set of fluents which are *definitely true*, and S^- is a set of fluents which are *definitely false*, while fluents that belong to none of the two sets are demeed *unknown.* Clearly, when $S^+ \cap S^- \neq \emptyset$ the state is inconsistent. Now the effect of an action is specified by a quadruple of sets describing the state where the action takes place and the resulting state, respectively:

$$state(S^+_{start}, S^-_{start}) \stackrel{action}{\longrightarrow} state(S^+_{end}, S^-_{end})$$

This schema gets implemented in $\{log\}$ in a rather straightforward way. In what follows, we shorten the function constant *state* into *s.*

Example 6 *(Continuation of Example 2)*

$$\begin{array}{lll} effect(s(S,R), load, s(T,U)) & :- & setminus(R, \{loaded\}, U)\ \& \\ & & union(S, \{loaded\}, T). \end{array}$$

Example 7 *(Continuation of Example 3)*

$$\begin{array}{lll} effect(s(S,R), shoot, s(T,U)) & :- & S = T\ \& \\ & & setminus(R, \{alive, loaded\}, T). \end{array}$$

Of course, Example 4 is still taken care of in terms of inertia axiom. Let us proceed with the formalization with double states of "shooting without knowing about the state of the gun makes *alive* become undefined."

Example 8 *(Continuation of Example 5)*

$$\begin{array}{lll} effect(s(S,R), shoot, s(T,U)) & :- & loaded \notin S\ \& \\ & & loaded \notin R\ \& \\ & & setminus(S, \{alive\}, T)\ \& \\ & & union(R, \{loaded\}, U). \end{array}$$

Notice how a)the gun ends up unloaded in any case and b)if alive was already false its value would not be changed.

3.4 Formalization of Example 1

While describing the axioms for Example 1, we also describe the inertia mechanisms which is added to this type of action formalization. In Appendix A, a precise definition of the translation of $\mathcal{A}$ theories into $\{log\}$ programs is given; however, the only axiom we had in Ex. 1 gets translated into:

$$\begin{array}{lll} \mathit{effect}(s(S,T),a,s(U,V)) & :- & f \in S \ \& \\ & & \mathit{setminus}(S,\{f\},U) \ \& \\ & & \mathit{union}(T,\{\mathit{loaded}\},V). \end{array}$$

The following is the *cancelation axiom:*

$$\mathit{no\_effect}(s(S,T),a) \ :- \ f \in T.$$

Notice how the negation on the constraint is done: the opposite of $f \in S$ is $f \in T$. Finally, we need the inertia axiom:

$$\mathit{effect}(s(S,T),a,s(S,T)) \ :- \ \mathit{no\_effect}(s(S,T),a).$$

Now, the interpretation of query:

$? - \mathit{effect}(s(S,T),a,s(U,V))$

gets the following positive answers:

(1) $S = \{f/U\},\ T = \_X,\ U = \_Y,\ V = \{f/T\}$
$[f \notin T,\ f \notin U]$.

(2) $S = U,\ T = \{f/U\},\ V = \{f/U\}$
$[f \notin U]$.

These two results accurately reflect the two models of the theory discussed earlier. In answer (1), instantiation $S = \{f/U\}$ corresponds to *assuming* f initially true, while $V = \{f/T\}$ corresponds to *concluding* f is false in the resulting state. In (2), instantiation $T = \{f/U\}$ corresponds to assuming f initially false, which is carried over to the resulting state by $V = \{f/U\}$. In both cases, constraints are important inasmuch as they ensure consistency of both $s(S,T)$ and $s(U,V)$.

4 A Plan Checker in $\{log\}$

A simple plan checker is a program that verifies whether a list of actions is adequate to achieve a certain goal, usually expressed as a set (a conjunction) of fluents that must be true in the final situation[7]. The success of the plan depends –apart from the ordering and type of actions– on the state of the domain where it is carried out. The initial state is described by asserting which fluents are initially true (resp. false). Of course, the description can be incomplete and it should be easy to update by adding new descriptions.

[7]It is easy to allow a goal to specify that certain fluents must be false in the final situation.

4.1 Defining the initial situation

Knowledge about the initial situation is asserted by means of facts, for instance:

$initially(alive, true).$ $\quad$ $initially(loaded, false).$
$initially(wet, true).$ $\quad$ $initially(sunny, false).$
$\ldots$ $\quad$ $\ldots$

The predicate *initial*, which is used to collect the fluents for which the initial value is given, is defined as follows.

$$\begin{array}{lll} initial(s(St, Sf)) & :- & St\_1 = \{X \ : \ initially(X, true)\} \ \& \\ & & Sf\_1 = \{X \ : \ initially(X, false)\} \ \& \\ & & union(St\_1, \_R\_1, St) \ \& \\ & & union(Sf\_1, \_R\_2, Sf) \ \& \\ & & consistent(s(St, Sf)). \end{array}$$

$$\begin{array}{lll} consistent(s(St, Sf)) & :- & forall(X \in St, \ X \notin Sf) \ \& \\ & & forall(X \in Sf, \ X \notin St). \end{array}$$

Since knowledge of the initial value of fluents is incomplete, *St* and *Sf* are instantiated to terms of the form $\{f_1, f_2 \ldots / R\}$ where $f_1, f_2 \ldots$ are asserted true (resp. false) by *initially* facts and R allows more fluents to be constrained to be true (resp. false) as the computation progresses. The last condition is there for constraining the variable part of *St* not to contain fluents already belonging to *Sf* and viceversa.

4.2 Architecture of the plan checker

The predicates defined so far are applied to the definition of the topmost predicate $holds(< goal >, < plan >)$ where *goal* is a set of fluents and *plan* is a list of actions. A query $?{-}holds(G, P)$ is true when all the fluents in G are true in the state reached by executing P. Predicate *exec* is used to simulate plan execution by applying each action of the plan, from left to right.

$$\begin{array}{lll} holds(Goal, Plan) & :- & initial(s(S, R)) \ \& \\ & & exec(s(S, R), Plan, s(T, U)) \ \& \\ & & subseteq(Goal, T). \end{array}$$

$$\begin{array}{l} exec(St, [\,], St). \\ exec(St\_in, [A|Rest], St\_out) :- \ apply(St\_in, A, St) \ \& \\ \qquad\qquad\qquad\qquad\qquad\quad exec(St, Rest, St\_out). \end{array}$$

While *holds* constraints plans that are successful starting from the initial situation, a plan simulation starting from a given initial state is made by simple changes:

$$\begin{array}{lll} holds(Domain, Goal, Plan) & :- & exec(Domain, Plan, s(T, U)) \ \& \\ & & subseteq(Goal, T). \end{array}$$

5 Final Remarks

The plan checker defined in the previous section may be of little practical utility, but it is important to notice how easily it can be expressed in $\{log\}$ ȧnd how it enjoys a straightforward declarative semantics. A realistic planner, e.g. one capable of *generating* the plan, could be defined starting from the schema above. In fact, with $\{log\}$ it is

easy to "animate" action specifications without compromising much on declarativeness. Our results on $\mathcal{A}$ are rather preliminary but suggest that $\{log\}$ has features apt to bring declarative programming into practice.

From the software engineering standpoint, theories of actions can be interesting as a testbed for building provably-correct classes of logic programs from rather abstract specifications. Our work aims to contribute to a better understanding of this process.

References

[1] K. Apt and M. Bezem, 1991. Acyclic Programs. *New Generation Computing,* 29(3):335–363.

[2] C. Baral, 1995. Reasoning about Actions : Non-deterministic effects, Constraints and Qualification In *Proc. of IJCAI95, 14th International Joint Conference on Artificial Intelligence,* pages 2017–2023.

[3] C. Baral and M. Gelfond, 1993. Representing concurrent actions in extended logic programming. In *Proc. of IJCAI93, 13th International Joint Conference on Artificial Intelligence,* pages 866–871.

[4] C. Baral, M. Gelfond and A. Provetti, 1997. Representing Actions: Laws, Observations and Hypothesis. *Journal of Logic Programming,* 31(1-3) special issue on actions. Available at *http://cs.utep.edu/csdept/students/aleWWW/papers.html*

[5] M. Denecker and D. De Schreye. Representing incomplete knowledge in abductive logic programming. In *Proc. of International Logic Programming Symposium 93*, pages 147–164, 1993.

[6] A. Dovier, E. Omodeo, E. Pontelli and G. Rossi, 1996. $\{log\}$: a Language For Programming in Logic with Finite Sets. *Journal of Logic Programming,* 28(1):1–44.

[7] A. Dovier and G. Rossi, 1993. Embedding extensional finite sets in CLP. In D. Miller (editor), *Proc. of ILPS93, the IVth Int'l Logic Programming Symposium,* pages 540–556, The MIT Press.

[8] M. Gelfond and V. Lifschitz, 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3,4):365–387.

[9] M. Gelfond and V. Lifschitz, 1992. Representing actions in extended logic programs. In *JICSLP92, Joint International Conference and Symposium on Logic Programming,* pages 559–573.

[10] R. Kowalski and M. Sergot, 1986. A logic-based calculus of events. *New Generation Computing,* 4:67–95.

[11] V. Lifschitz, N. McCain, and H. Turner, 1993. Automation of reasoning about action: a logic programming approach. In *Posters of ILPS93, International Symposium on Logic Programming.*

[12] V. Lifschitz and H. Turner, 1994. Splitting a logic program. In P. Van Hentenryck, editor, *Proc. of ICLP94, the Eleventh Int'l Conf. on Logic Programming,* pages 23–38.

[13] A. Provetti, 1996. Hypothetical reasoning about actions: from situation calculus to event calculus. *Computational Intelligence,* 12(3):478–498. Available at *http://cs.utep.edu/csdept/students/aleWWW/papers.html*

[14] H. Turner, 1993. A monotonicity theorem for extended logic programs. In D. S. Warren (editor), *Proc. of ICLP93, the 10th International Conference on Logic Programming,* pages 567–585.

A From $\mathcal{A}$ theories to $\{log\}$ programs

In this Appendix we define a syntactic translation of $\mathcal{A}$ theories into $\{log\}$ programs. To do so, we restrict our consideration to those theories where *value propositions* are restricted to refer to the initial situation only. Therefore, each value proposition can be written as **initially** L where $L = F$ or $\neg F$. Moreover, we restrict to consider only those effect propositions whose conditions are expressed as a conjunction sof fluent literals. For any theory $T_{\mathcal{A}}$ we build the corresponding program P incrementally, following this recipe:

1. for each value proposition **initially** F (resp. **initially** $\neg F$) in $T_{\mathcal{A}}$ add the fact
   initially(f, true) (resp. *initially(f, false)*)
   to P;

2. for each *effect proposition*
   A **causes** F **if** $\bigwedge F_i$
   in $T_{\mathcal{A}}$ add the rules

$$\begin{array}{lll} \mathit{effect}(s(S,T),a,s(U,V)) & :- & f_1 \in S \;\&\\ & & \ldots \;\&\\ & & f_n \in S \;\&\\ & & \mathit{union}(S,\{f\},U) \;\&\\ & & \mathit{setminus}(T,\{f\},V). \end{array}$$

$$\begin{array}{lll} \mathit{no\_effect}(s(S,T),a) & :- & f_1 \in T \;\&\\ & & \ldots \;\&\\ & & f_n \in T. \end{array}$$

   to P; the case where the effect and or the conditions are negative fluents is similar, mutatis mutandis;

3. finally, these rules are always included in P:

$$\mathit{effect}(s(S,T),a,s(S,T)) \;:-\; \mathit{no\_effect}(s(S,T),a).$$

$$\begin{array}{lll} \mathit{initial}(s(St,Sf)) & :- & St\_1 = \{X \;:\; \mathit{initially}(X,\mathit{true})\} \;\&\\ & & Sf\_1 = \{X \;:\; \mathit{initially}(X,\mathit{false})\} \;\&\\ & & \mathit{union}(St\_1,\_R\_1,St) \;\&\\ & & \mathit{union}(Sf\_1,\_R\_2,Sf) \;\&\\ & & \mathit{consistent}(s(St,Sf)). \end{array}$$

$$\begin{array}{lll} \mathit{consistent}(s(St,Sf)) & :- & \mathit{forall}(X \in St,\; X \notin Sf) \;\&\\ & & \mathit{forall}(X \in Sf,\; X \notin St). \end{array}$$

$$\begin{array}{lll} \mathit{holds}(Goal,Plan) & :- & \mathit{initial}(s(S,R)) \;\&\\ & & \mathit{exec}(s(S,R),Plan,s(T,U)) \;\&\\ & & \mathit{subseteq}(Goal,T). \end{array}$$

$exec(St, [\,], St).$
$exec(St\_in, [A|Rest], St\_out) :- \quad apply(St\_in, A, St)\ \&$
$\qquad\qquad\qquad\qquad\qquad\qquad\quad exec(St, Rest, St\_out).$

$holds(Domain, Goal, Plan) :- \quad exec(Domain, Plan, s(T, U))\ \&$
$\qquad\qquad\qquad\qquad\qquad\quad subseteq(Goal, T).$