Extending Answer Set Planning with Sequence, Conditional, Loop, Non-Deterministic Choice, and Procedure Constructs

Tran Cao Son*
New Mexico State University
Computer Science Department
PO Box 30001
Las Cruces, NM 88003-8001
tson@cs.nmsu.edu

Chitta Baral
Computer Science and Engineering
Arizona State University
Tempe, AZ 85287, USA
chitta@asu.edu

Sheila McIlraith
Knowledge Systems Laboratory
Computer Science
Stanford University
Stanford, CA 94305
sam@ksl.stanford.edu

Abstract

We extend answer set programming of dynamical systems with more expressive programming constructs such sequence, conditional, loop, non-deterministic choice of actions/arguments, and procedures. We discuss its relevance to the problem of answer set planning. We present an SMODELS encoding of these constructs and formally prove the correctness of our encoding.

Introduction

In (Lifschitz 1999), Lifschitz showed how answer set programming (ASP) (Marek & Truszczyński 1999; Niemelä 1999) can be used to do planning and coined the term *answer set planning*. It combines the advancements of research in reasoning about actions using logic programming (see e.g. (Gelfond & Lifschitz 1993) and the papers in (Lifschitz 1997)) and answer set programming (Marek & Truszczyński 1999; Niemelä 1999). While the latter proposes a general framework to solve constraint satisfaction problems in logic programming, the former supplies the logic programming axioms for representing and reasoning about actions and their effects, which can be used for a variety of tasks, including plan verification.

The key idea of answer set programming is to represent the solution to a problem by an *answer set*, or a *model* of the program. That is, to solve a problem, one first needs to represent it as a logic program whose answer sets correspond one-to-one to the solutions of the problem; next, to find a solution, one uses available tools such as SMODELS (Niemelä & Simons 1997), DLV (Citrigno *et al.* 1997), DeRes (Cholewinski, Marek, & Truszczynski 1996), or CCALC (McCain & Turner 1997) to compute an answer set of the program; finally, some translation from the answer set to the solution of the problem may be necessary. As its name suggests, answer set planning focuses on solving planning problems using the answer set programming paradigm.

Even though logic programming is regarded as suitable for use as both the representation language and the implementation language in answer set programming, it is argued that in many cases, a richer language for representation is useful in answer set programming (Simons 1999; Niemelä, Simons, & Soininen 1999). Simons introduced new types of rules called *choice rules, constraint rules,* and *weight rules* and Niemelä *et al.*demonstrated its usefulness in different applications (Niemelä, Simons, & Soininen 1999). The stable model semantics of logic programs containing these rules is given in (Simons 1999; Niemelä, Simons, & Soininen 1999). These features are now implemented in SMODELS, an efficient implementation of stable model semantics of logic programs.

In this paper, we extend answer set programming by adding more expressive constructs such as sequence, conditional, loop, non-deterministic choice of actions/arguments, and procedures, for representing and reasoning with dynamical domains. These constructs are derivative of constructs in procedural programming languages such as ALGOL, and of constructs in the logic programming language GOLOG (Levesque *et al.* 1997). We define the semantics of these constructs using a predicate called *trans* which is adapted from the *Trans* and *Final* predicates used to define the computational semantics of ConGolog (De Giacomo, Lesperance, & Levesque 1997; De Giacomo, Lespérance, & Levesque 2000). We implement an interpreter for these constructs in SMODELS and formally prove its correctness. We discuss the relevance of this work to answer set planning.

The paper is organized as follows. We begin with a review of the basics of stable model semantics, the action description language $\mathcal{B}$ and answer set planning. We then introduce the new constructs, present their encoding in logic programming, and prove the correctness of the implementation. We relate our work to GOLOG and discuss some desirable extensions of the current work in the last section.

Preliminaries

Stable Models of Logic Programs

We review the basic notion of a stable model for extended logic programs, introduced by Gelfond and Lifschitz in (Gelfond & Lifschitz 1988). A program Π is defined over a first-order language $\mathcal{LP}$, extended with a special unary predicate – the negation-as-failure operator – denoted by *not*. A negation-as-failure literal (or naf-literal) is of the form $not\ l$ where l is an atom of the language $\mathcal{LP}$. A program Π is a set of rules of the form

$$l_0 \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n, \qquad (1)$$

*Work done while at the Knowledge Systems Laboratory, Stanford University.

where $0 \leq m \leq n$, each l_i is an atom, and, *not* represents the negation-as-failure operator.

Let $lit(\Pi)$ denote the set of ground atoms in the language of the program Π.

1. If Π does not contain any naf-literal (i.e. $m = n$ in every rule of Π) a stable models of Π is defined as the smallest set S, $S \subseteq lit(\Pi)$ such that for any ground instance $l_0 \leftarrow l_1, \ldots, l_m$ of a rule from Π, if $l_1, \ldots, l_m \in S$, then $l_0 \in S$.
2. If the program Π does contain some naf-literals ($m < n$ in some rule of Π), $S \subseteq lit(\Pi)$ is a stable model of Π if S is the stable model of the program Π' obtained from the set of all ground instances of Π by deleting
   (a) each rule that has a formula *not* l in its body with $l \in S$, and
   (b) all formulas of the form *not* l in the bodies of the remaining clauses.

We note that in ASP, a new type of rules called *constrains* with empty head (or $l_0 =$ *false*) is extremely useful. Stable models of logic programs with constraints are defined formally in (Niemelä, Simons, & Soininen 1999; Lifschitz & Turner 1999).

Representing Action Theories in $\mathcal{B}$

We use the high-level action description language $\mathcal{B}$ of (Gelfond & Lifschitz 1998) to represent action theories. In such a language, an action theory consists of two finite, disjoint sets of names called *actions* and *fluents* and a set of propositions of the following form:

$$\textbf{caused}(\{p_1, \ldots, p_n\}, f) \tag{2}$$

$$\textbf{causes}(a, f, \{p_1, \ldots, p_n\}) \tag{3}$$

$$\textbf{executable}(a, \{p_1, \ldots, p_n\}) \tag{4}$$

$$\textbf{initially } f \tag{5}$$

where f and p_i's are fluent literals (a *fluent literal* is either a fluent g or its negation $\neg g$, written as $neg(g)$) and a is an action. (2) represents a static causal law, i.e., a ramification constraint. (3) represents the (conditional) effect of a, while (4) denotes executability condition of a. Propositions of the form (5) are used to describe the initial state. Often, an action theory is given by a pair (D, Γ) where D consists of propositions of the form (2)-(4) and Γ consists of propositions of the form (5). For the purpose of this paper, it suffices to note that the semantics of such an action theory is given by a transition graph, represented by a relation t, whose nodes are the states of the action theory and whose links (labeled with actions) represent the transition between its states (see (Gelfond & Lifschitz 1998) for details). That is, if $\langle s, a, s' \rangle \in t$, then there exists a link with label a from state s to state s'. A *trajectory* of the system is denoted by a sequence $s_0 a_1 s_1 \ldots a_n s_n$ where s_i's are states and a_i's are actions and $\langle s_i, a_{i+1}, s_{i+1} \rangle \in t$ for $i \in \{0, \ldots, n-1\}$.

Answer Set Planning

A *planning problem* is specified by a triple $\langle D, \Gamma, \Delta \rangle$ where (D, Γ) is an action theory and Δ is a fluent formula (or *goal*), representing the (partial) goal state. A sequence of actions $a_1, \ldots, a_m$ is a *possible plan for* Δ if there exists a trajectory $s_0 a_1 s_1 \ldots a_m s_m$ such that s_0 and s_m satisfies Γ and Δ, respectively.

Observe that the notion of plan employed here is weaker than the conventional one where the goal must be achieved on every possible trajectories. This is because an action theory with causal laws can be non-deterministic. Generating plans for non-deterministic action theories is an interesting topic but is beyond the scope of this paper. If D is deterministic, i.e., for every pair of a state s and action a there exists at most one state s' such that $\langle s, a, s' \rangle \in t$, then every possible plan for Δ is also a plan for Δ.

Given a planning problem $\langle D, \Gamma, \Delta \rangle$, answer set planning solves it by translating it into a logic program $\Pi(D, \Gamma, \Delta)$ (or Π, for short) consisting of *domain-dependent* rules that describe D, Γ, and Δ respectively, and *domain-independent* rules that generate action occurrences and represent the transitions between states.

Action theory representation. We use two predicates *set* and in to define the sort *set* and to represent the set membership function, respectively. We assign to each set of fluent literals that occurs in a proposition of D a distinguished name. The constant nil denotes the set $\{\}$. A set of literals $\{p_1, \ldots, p_n\}$ will be replaced by the set of atoms $Y = \{set(s), in(p_1, s), \ldots, in(p_n, s)\}$ where s is the name assigned to $\{p_1, \ldots, p_n\}$. With this representation, propositions in D can be easily translated into a set of facts of Π. For example, a proposition $causes(a, f, \{p_1, \ldots, p_n\})$ with $n > 0$ is encoded as a set of atoms consisting of $causes(a, f, s)$ and the set Y (s is the name assigned to $\{p_1, \ldots, p_n\}$).

Goal representation. To encode Δ, we define formulas and provide a set of rules for formula evaluation. Due to the typing requirement of SMODELS, we consider formulas which are bounded classical formulas with each bound variable associated with a sort. They are formally defined as follows.

- A literal is a formula.
- The negation of a formula is a formula.
- A finite conjunction of formulas is a formula.
- A finite disjunction of formulas is a formula.
- If $X_1, \ldots, X_n$ are variables that can have values from the sorts $s_1, \ldots, s_n$, and $f_1(X_1, \ldots, X_n)$ is a formula then $\forall X_1, \ldots, X_n . f_1(X_1, \ldots, X_n)$ is a formula.
- If $X_1, \ldots, X_n$ are variables that can have values from the sorts $s_1, \ldots, s_n$, and $f_1(X_1, \ldots, X_n)$ is a formula then $\exists X_1, \ldots, X_n . f_1(X_1, \ldots, X_n)$ is a formula.

The encoding of formulas in SMODELS is done similarly to the encoding of sets. A sort called *formula* is introduced and each non-atomic formula will be associated with a unique name and defined by (possibly) a set of rules. For example, the formula in the fifth item can be represented by the rule $formula(forall(f, f_1(X_1, \ldots, X_n))) \leftarrow in(X_1, s_1), \ldots, in(X_n, s_n)$ where f is the name assigned to it. As with literal, negation will be represented by the function symbol neg. For example, if f is the name of a formula then $neg(f)$ is a formula representing its negation. Rules to check when a formula holds or does not hold can be written in a straightforward manner and are omitted here to save space.

Domain-independent rules. The domain-independent rules of Π are adapted mainly from (Gelfond & Lifschitz 1992). The key difference is the representation of time that has been used previously in (Dimopoulos, Nebel, & Koehler 1997; Lifschitz 1999; Lifschitz & Turner 1999). The main predicates in these rules are:

- $holds(L, T)$: L holds at time T,
- $possible(A, T)$: action A is executable at time T,
- $occ(A, T)$: action A occurs at time T,
- $holds\_formula(\varphi, T)$: formula φ holds at time T, and
- $holds\_set(S, T)$: S - a set of literals - holds at time T.

The main domain-independent rules are given next. In these rules, T is a variable of the sort $time$, L, G are variables denoting *fluent literals* (written as F or $neg(F)$ for some fluent F), S is a variable set of the sort set, and A, B are variables of the sort $action$.

```
holds(L, T+1):-
    occ(A, T), causes(A, L, S),
    holds_set(S, T).
holds(L, T):-
    caused(S, L), holds_set(S, T).
holds(L, T+1):-
    contrary(L, G),
    holds(L, T), not holds(G, T+1).
possible(A,T):-
    executable(A, S),
    holds_set(S, T).
holds(L, 0):-
    literal(L),
    initially(L).
nocc(A,T):-
    A =/= B, occ(B,T), T < length.
occ(A,T):-
    T < length,
    possible(A,T), not nocc(A,T).
```

Here, the first rule encodes the effects of action, the second rule encodes the effects of static causal laws, and the third rule is the inertial rule. The fourth rule defines a predicate that determines when an action can occur and the fifth encodes the initial situation. The last two rules are used to generate action occurrences, one at a time. $length$ is used to stipulate the maximum length of the resulting program. We omit most of the auxiliary rules such as rules for defining contradictory literals etc. The source codes and examples can be retrieved from our web-site[1].

Let $\Pi_n(D, \Gamma, \Delta)$ (or Π_n when it is clear from the context what D, Γ, and Δ are) be the logic program consisting of

- the set of domain-independent rules in which the domain of T is $\{0, \ldots, n\}$,
- the set of atoms encoding D and Γ, and
- the rule $\leftarrow$ not $hf(\Delta, n)$ that encodes the requirement that Δ holds at n.

The following result (adapted from (Lifschitz & Turner 1999)) shows the equivalence between trajectories of $\langle D, \Gamma, \Delta\rangle$ and stable models of $\Pi_n(D, \Gamma, \Delta)$.

Theorem 1 *Given a planning problem, $\langle D, \Gamma, \Delta\rangle$. Let S be a stable model of $\Pi_n(D, \Gamma, \Delta)$, and define $s(i) = \{f \mid holds(f, i) \in S\}$ and $A[i, j] = [a_i, \ldots, a_j]$ where i, j are integers, f is a fluent, a_t is an action, and for every t, $i \le t \le j$, $occ(a_t, t) \in S$, then*

- *if $s_0 a_0 \ldots a_{n-1} s_n$ is a trajectory of $\langle D, \Gamma, \Delta\rangle$, then there exists a stable model S of Π_n such that $A[0, n-1] = [a_0, \ldots, a_{n-1}]$ and $s_i = s(i)$ for $i \in \{0, \ldots, n\}$, and*
- *if S is a stable model of Π_n with $A[0, n-1] = [a_0, \ldots, a_{n-1}]$ then $s(0)a_0 \ldots a_{n-1}s(n)$ is a trajectory of $\langle D, \Gamma, \Delta\rangle$.*

ALGOL-like Constructs in Answer Set Planning and Programming

GOLOG is a logic programming language developed by the Cognitive Robotics Group, University of Toronto, for reasoning about dynamical systems (Levesque *et al.* 1997). In GOLOG, ALGOL-like constructs such as sequence, loop, conditional, and nondeterministic choice of arguments/actions are added to the situation calculus language as macros that can be used to write programs. In ConGolog (De Giacomo, Lespérance, & Levesque 2000), a variant of Golog, these constructs are realized as terms within the language. Instead of planning, GOLOG and its variants are used to specify (non-deterministic) programs that constrain the evolution of the world. In the situation calculus, searching for a plan amounts to deductively instantiating the binding of the situation variable in the goal formula. The situation dictates the sequence of actions from the initial situation required to entail the goal. Similarly, a sequence of actions that realizes a GOLOG program is simply determined by searching for appropriate bindings of situation terms that satisfy situation calculus formulae established by our GOLOG program. PROLOG-based GOLOG-interpreters have been developed and used in a variety of applications (e.g., (Boutilier *et al.* 2000)).

We will show next that this feature can also be integrated easily into answer set programming, and used for answer set planning. First, we extend the language of answer set programming with the complex constructs to define programs. For an action theory (D, Γ) we define

- an action a is a program,
- a formula ϕ is a program,
- if p_i's are programs then $p_1; \ldots; p_n$ is a program,
- if p_i's are programs then $p_1 | \ldots | p_n$ is a program,
- if p_1 and p_2 are programs and ϕ is a formula then "**if** ϕ **then** p_1 **else** p_2" is a program,
- if p is a program and ϕ is a formula then "**while** ϕ **do** p" is a program, and

[1] http://www.ksl.stanford.edu/people/son/macros.html

- if X is a variable of sort s, $p(X)$ is a program, and $f(X)$ is a formula, then **pick**$(X, f(X), p(X))$ is a program.

The operational semantics of programs is defined as follows. A trajectory $s_0a_0s_1 \ldots a_{n-1}s_n$, denoted by α, is said to be *a trace of a program p* if

- for $p = a$ and a is an action, $n = 1$ and $a_0 = a$,
- for $p = \phi$, $n = 0$ and ϕ holds in s_0,
- for $p = p_1; p_2$, there exists an i such that $s_0a_0 \ldots s_i$ is a trace of p_1 and $s_ia_i \ldots s_n$ is a trace of p_2,
- for $p = p_1 | \ldots | p_n$, α is a trace of p_i for some $i \in \{1, \ldots, n\}$,
- for $p =$ **if** ϕ **then** p_1 **else** p_2, α is a trace of p_1 if ϕ holds in s_0 or α is a trace of p_2 if $neg(\phi)$ holds in s_0,
- for $p =$ **while** ϕ **do** p_1, $n = 0$ and $neg(\phi)$ holds in s_0 or ϕ holds in s_0 and there exists some i such that $s_0a_0 \ldots s_i$ is a trace of p_1 and $s_ia_i \ldots s_n$ is a trace of p, and
- for $p =$ **pick**$(X, f(X), q(X))$, then there exists a constant x of the sort of X such that $f(x)$ holds in s_0 and α is a trace of $q(x)$.

We next present the logic programming rules that realize this semantics. We define a predicate $trans(p, t_1, t_2)$ which holds in a stable model S iff $s(t_1)a_{t_1} \ldots s(t_2)$ is a trace of p[2]. As is customary with SMODELS, we will assign to each program a name (with the exceptions of action or formula), provide rules for the construction of programs, and use the prefix notations. More precisely,

- A program $p_1; \ldots; p_n$ is represented by the atoms $proc(p)$, $head(p, n_1)$, $tail(p, n_2)$ and the set of atoms representing $p_2; \ldots; p_n$, where p, n_1, and n_2 are the names assigned to $p_1; \ldots; p_n$, p_1 (if it is not a primitive action or a formula), and $p_2; \ldots; p_n$, respectively.
- A program $p_1 | \ldots | p_m$ is represented by the set of atoms $set(s)$, $in(n_1, s), \ldots, in(n_m, s)$, $choiceAction(s)$, and the set of atoms representing p_i's where s and n_i's are the names assigned to the program $p_1 | \ldots | p_m$ and p_i's, respectively.
- A program **if** ϕ **then** p_1 **else** p_2 is represented by the atom $if(n, n_\phi, n_1, n_2)$ and the set of atoms representing p_1 and p_2 where n, n_ϕ, n_1 and n_2 are the names assigned to the program **if** ϕ **then** p_1 **else** p_2, the formula ϕ, p_1, and p_2, respectively.
- A program **while** ϕ **do** p_1 is represented by the atom $while(n, n_\phi, n_1)$ and the set of atoms representing p_1 where n, n_ϕ, and n_1 are the names assigned to the program **while** ϕ **do** p_1, the formula ϕ, and p_1, respectively.
- A program **pick**(X, ϕ, p_1) is represented by the rule $choiceArgs(n, n_\phi, n_1) \leftarrow s(X)$ and the set of atoms representing p_1 where $s(X)$ is the type definition of X, and n, n_ϕ, and n_1 are the names assigned to the program **pick**(X, ϕ, p_1), the formula ϕ, and p_1, respectively.

The rules for the $trans$ predicate are listed below.

[2] Recall that we define $s(i) = \{holds(f, i) \in S \mid f \text{ is a fluent}\}$ and assume $occ(a_i, i) \in S$.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% axioms for programs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
trans(P,Tb,Te):-
    time(Tb), time(Te), time(Te1),
    Tb <= Te1, Te1 <= Te,
    proc(P), head(P,P1), tail(P,P2),
    trans(P1,Tb,Te1), trans(P2,Te1,Te).

trans(A,Tb,Tb+1):- time(Tb),
    action(A), neq(A, null), occ(A,Tb).

trans(null,Tb,Tb):- time(Tb).

trans(N,Tb,Te):- time(Tb), time(Te),
    Tb <= Te, in(P1,N),
    choiceAction(N),
    trans(P1,Tb,Te).

trans(F,Tb,Tb):- time(Tb),
    formula(F), holds_formula(F,Tb).

trans(I,Tb,Te):- time(Tb), time(Te),
    Tb <= Te, if(I,F,P1,P2),
    holds_formula(F,Tb), trans(P1,Tb,Te).

trans(I,Tb,Te):- time(Tb), time(Te),
    Tb <= Te, if(I,F,P1,P2),
    not holds_formula(F,Tb),
    trans(P2,Tb,Te).

trans(W,Tb,Te):- time(Tb), time(Te),
    while(W,F,P), holds_formula(F,Tb),
    time(Te1), Tb <= Te1, Te1 <= Te,
    Tb <= Te, trans(P,Tb,Te1),
    trans(W,Te1,Te).

trans(W,Tb,Tb):- time(Tb),
    while(W,F,P), not holds_formula(F,Tb).

trans(S, Tb, Te):-
    time(Tb), time(Te), Tb <= Te,
    choiceArgs(S,F,P), holds_formula(F, Tb),
    trans(P, Tb, Te).
```

Just as these constructs can be used to write non-deterministic programs for dynamical systems in GOLOG, so too can they be used to write non-deterministic programs within our answer set programming paradigm. As noted previously, these constructs serve to constrain the possible evolutions of a dynamical system, and hence the possible trajectories. As such they can be used in the context of answer set planning to specify domain-specific control knowledge that constrains the possible plans under consideration. Domain-specific control knowledge has been shown to by Bacchus and Kabanza and others (e.g., (Bacchus & Kabanza 2000)) to be an effective way of speeding up planning.

Example

In this section, we present our encoding of the canonical elevator example, as described in (Levesque *et al.* 1997), and show the results of some queries.

```
% Elevator example %%%%%%%%%%%%%%%%%%%%%%
floor(0..5).

% action declarations
action(up(N)):- floor(N).
action(down(N)):- floor(N).
action(turnoff(N)):- floor(N).
action(open).  action(close). action(null).

% fluents declarations
fluent(currentFloor(N)):- floor(N).
fluent(on(N)):- floor(N).
fluent(opened).

% actions
causes(up(N), currentFloor(N), nil):-
    floor(N).
causes(down(N), currentFloor(N), nil):-
    floor(N).
causes(turnoff(N), neg(on(N)), nil):-
    floor(N).
causes(open, opened, nil).
causes(close, neg(opened), nil).

% executable conditions of actions
executable(up(N), list1(N)):- floor(N).
executable(down(N), list2(N)):-
floor(N).
executable(turnoff(N), list3(N)):-
    floor(N).
executable(open, nil).
executable(close, nil).
executable(null, nil).

set(list1(N)):- floor(N).
set(list2(N)):- floor(N).
set(list3(N)):- floor(N).

in(above(N), list1(N)):- floor(N).
in(below(N), list2(N)):- floor(N).
in(on(N), list3(N)):- floor(N).

% defined fluents
holds(neg(currentFloor(N)), T):-
    time(T), floor(N), floor(M),
    holds(currentFloor(M), T), neq(N,M).

holds(below(N), T):-
    time(T), floor(N), floor(M),
    holds(currentFloor(M), T), N < M,
    holds(neg(opened),T).

holds(above(N), T):-
    time(T), floor(N), floor(M),
    holds(currentFloor(M), T), N > M,
    holds(neg(opened),T).

% initial situation
initially(on(3)). initially(on(5)).
initially(currentFloor(2)).
initially(neg(opened)).
```

The following procedures were provided in (Levesque *et al.* 1997).

```
proc(serve(N),
    [go(N),turnoff(N),open,close]).
proc(go(N),
    up(N)|down(N)|in(currentFloor(N)).
proc(serve_a_floor,
    pick(N, on(N), serve(N))).
proc(control,[while(on(N),serve_a_floor),
    if(currentFloor(0),
            open, [down(0), open]).
```

We encode these procedures with the following rules.

```
% procedure serve(N)
proc(serve(N)):- floor(N).
head(serve(N), h1(N)):- floor(N).
tail(serve(N), t1((N)):- floor(N).

proc(t1((N)):- floor(N).
head(t1((N), turnoff(N)):- floor(N).
tail(t1((N), t2):- floor(N).

proc(t2). head(t2, open). tail(t2, close).

proc(h1(N)):- floor(N).
head(h1(N), p(N)):- floor(N).
tail(h1(N), null):- floor(N).
choiceAction(p(N)):- floor(N).

in(up(N), p(N)):- floor(N).
in(down(N), p(N)):- floor(N).
in(currentFloor(N), p(N)):- floor(N).

% procedure serve_a_floor
proc(serve_a_floor).
head(serve_a_floor, s(N)):- floor(N).
tail(serve_a_floor, null).

choiceArgs(s(N), on(N), serve(N)):-
    floor(N).

%procedure control
proc(control).
head(control, wloop).
tail(control, park).

while(wloop, existOn, serve_a_floor).

exists(existOn, on(N)):- floor(N).

%procedure park
```

```
proc(park).
head(park, if0). tail(park, null).

if(if0, currentFloor(0),
        open, elsePark).

proc(elsePark).
head(elsePark, down(0)).
tail(elsePark, open).
```

To find a trace of a program with the length less than or equal $length$, we add the following rule to Π_n.

```
goal(T):- time(T), trans(Program, 0, T).
goal(T+1):- time(T), T < length, goal(T).
:- not goal(length).
```

Sample runs:

- For $Program = control$ and $length = 11$, asking for one solution, the program returns:

  ```
  occ(up(5),0) occ(turnoff(5),1)
  occ(open,2) occ(close,3) occ(down(3),4)
  occ(turnoff(3),5) occ(open,6)
  occ(close,7) occ(down(0),8) occ(open,9)
  ```

- For $Program = serve\_a\_floor$ and $length = 11$, asking for all solutions, the program returns:

  ```
  occ(up(5),0) occ(turnoff(5),1)
  occ(open,2) occ(close,3)

  occ(up(3),0) occ(turnoff(3),1)
  occ(open,2) occ(close,3)
  ```

Comparison to GOLOG

In this section, we compare our implementation with PROLOG-based implementations of GOLOG. The following points differentiate our implementation from other GOLOG-interpreters[3]:

1. Our implementation is done in a purely declarative logic programming language while other GOLOG-interpreters use PROLOG. One advantage of using a purely declarative logic programming language over PROLOG is that it avoids the many non-declarative features of PROLOG such as left to right ordering of literals in the body of a rule, top to bottom processing of rules in a file, or infinite loops even in finite domains etc.
2. Our implementation can be used only with finite action theories. On the other hand, PROLOG-based GOLOG-interpreters could in principle be applied to infinite situation calculus theories, provided that the successor state axioms of fluents can be encoded in such a way that loops can be avoided[4].
3. Our implementation is elaboration tolerant with respect to the addition of temporal constraints to the planning problem. PROLOG implementation are not as elaboration tolerant. This can be attributed to the difference between answer set planning/programming and the PROLOG query answering procedure. Answer set programming/planning uses the *generate-test algorithm* to solve a planning problem. Thus, adding a constraint does not require extra work other than its specification. In PROLOG, the query answering procedure is *goal-directed*. As such, looking for a plan satisfying some *additional* constraints would require a query reformulation or a change to the problem specification. Discussions of the inclusion of simple constraints in GOLOG can be found in (McIlraith 2000b), however to the best of our knowledge, none of the current PROLOG-based GOLOG interpreters supports this feature.
4. Our action formalization is more elaboration tolerant with respect to the addition of state constraints than situation calculus formalizations. This is because state constraints are specified separately and can be added/removed easily. On the other hand, state constraints generally have to be incorporated into successor state axioms of fluents in the situation calculus framework. As such, adding/removing a state constraint would generally require a (simple) rewrite of the successor state axioms. For a treatment of state constraints in solitary stratified situation calculus theories, see (McIlraith 2000a).
5. By adding the cut operator at appropriate choice points, a PROLOG-based GOLOG interpreter can be easily converted into an online interpreter. This is not the case with our implementation. The main reason is again the difference between answer set planning and PROLOG query answering procedure.

As a final point in our comparison between our answer set programming implementation and Golog, we prove that for action theories without static causal laws, the logic programming implementation of the previous section can be viewed as a sound and complete GOLOG-interpreter.

Let (D, Γ) be an action theory without a static causal law and a complete initial state. It has been shown in (Kartha 1993) that there exists a situation calculus theory D^T which is equivalent to (D, Γ). It is easy to see that a program wrt. an action theory (D, Γ) defined in the previous section could be viewed as a GOLOG-program in D^T. Vice versa, if a GOLOG-program contains only constructs mentioned in this paper then it can be viewed as a program in this paper too. Let G_n be the logic program consisting of

[3]We are aware of at least two interpreters: one is written in Eclipse Prolog and the other in Quintus Prolog. A proof for the soundness and completeness of a PROLOG-based ConGolog-interpreter can be found in (De Giacomo, Lespérance, & Levesque 2000).

[4]E.g., it is not clear whether the infinite situation calculus theory consisting of an action A, the fluents G and $H(n)$ for integer n, the initial situation axioms $\neg G(S_0)$, $\forall n.[H(n, S_0)]$, and the successor state axioms $G(do(A, s)) \equiv (\forall n.[H(n, s)]) \vee G(s)$ and $H(n, do(A, s)) \equiv H(n, s)$ can be encoded in PROLOG such that the query $Do([A^*; (G)?], S_0, s)$ yields the correct answer, $s = do(A, S_0)$.

- the set of domain-independent rules (Section 2) in which the domain of T is $\{0, \ldots, n\}$,
- the set of atoms describing D and Γ, and
- the set of $trans$-rules in which the domain of T is $\{0, \ldots, n\}$.

The following theorems summarize the relationship between stable models of G_n and valid instantiations of a program P in D^T.

Theorem 2 *For every program P and a stable model S of G_n, if $trans(P, 0, n) \in S$ then $D^T \models Do(P, S_0, do(A[0, n], S_0))$*[5].

Theorem 3 *For every program P and action sequence $a_0, \ldots, a_n$ such that $D^T \models Do(P, S_0, do([a_0, \ldots, a_n], S_0))$, there exists a stable model S of G_n such that $trans(P, 0, n) \in S$ and $occ(a_i, i) \in S$.*

Conclusions

In this paper, we extended answer set programming of dynamical systems, as proposed for answer set planning, by introducing ALGOL-like constructs. These constructs can be used to provide domain-specific information that constrains the evolution of a dynamical system, as might be done in writing a non-deterministic program, or in specifying domain-specific control knowledge in a planning problem. In addition to extending the answer set programming paradigm, we also presented an SMODELS implementation of these constructs. The implementation can be used as a GOLOG-like interpreter for the class of GOLOG programs which are expressible by the constructs defined in this paper. This shows that GOLOG programming can be easily integrated into answer set programming/planning. One of our main goals in the future is to study the integration of other planning techniques into answer set planning.

There are many extensions to the original GOLOG described in 1997. These extensions include the concurrent actions, interrupts, and priorities of ConGolog (De Giacomo, Lespérance, & Levesque 2000), partial ordering (Baral & Son 1999), continuous change (cc-Golog) (Grosskreutz & Lakemeyer 2000), knowledge-producing actions (SGOLOG) (Lakemeyer 1999), and the probabilistic actions and decision-theoretic notions of DTGOLOG (Boutilier *et al.* 2000). In future work, we intend to extend our implementation of answer set programming of dynamical systems to include these additional features. We also plan to develop a better interface that converts programs in their standard form into eligible input of SMODELS.

[5]The intuitive meaning of

$$D^T \models Do(P, S_0, do([a_1, \ldots, a_m], s))$$

is that the program P, starting execution in situation S_0 will legally terminate in situation $do([a_1, \ldots, a_m], s)$ which is a shorthand of the situation $do(a_m, do(\ldots, do(a_1, S_0)))$. More on GOLOG axioms and definitions can be found in (Levesque *et al.* 1997; Reiter 1998) etc.

References

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116(1,2):123–191.

Baral, C., and Son, T. C. 1999. Extending ConGolog to allow partial ordering. In *Proceedings of the 6th International Workshop on Agent Theories, Architectures, and Languages (ATAL), LNCS, Vol. 1757*, 188–204.

Boutilier, C.; Reiter, R.; Soutchanski, M.; and Thrun, S. 2000. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the 17th National Conference on Artificial Intelligence*, 355–362.

Cholewinski, P.; Marek, V.; and Truszczynski, M. 1996. Default reasoning system DeReS. In *Proceedings of the 5th International Conference on Knowledge Representation and Reasoning*. Morgan Kaufmann.

Citrigno, S.; Eiter, T.; Faber, W.; Gottlob, G.; Koch, C.; Leone, N.; Mateis, C.; Pfeifer, G.; and Scarcello, F. 1997. The dlv system: Model generator and application frontends. In *Proceedings of the 12th Workshop on Logic Programming*, 128–137.

De Giacomo, G.; Lesperance, Y.; and Levesque, H. 1997. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In *Proceddings of the 15th International Joint Conference on Artificial Intelligence*, 1221–1226.

De Giacomo, G.; Lespérance, Y.; and Levesque, H. 2000. *ConGolog*, a concurrent programming language based on the situation calculus. *Artificial Intelligence* 121(1-2):109–169.

Dimopoulos, Y.; Nebel, B.; and Koehler, J. 1997. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of European Conference on Planning*, 169–181.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In Kowalski, R., and Bowen, K., eds., *Logic Programming: Proc. of the Fifth International Conf. and Symp.*, 1070–1080.

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

Gelfond, M., and Lifschitz, V. 1993. Representing actions and change by logic programs. *Journal of Logic Programming* 17(2,3,4):301–323.

Gelfond, M., and Lifschitz, V. 1998. Action languages. *ETAI* 3(6).

Grosskreutz, H., and Lakemeyer, G. 2000. cc-golog: Towards more realistic logic-based robot controllers. In *Proceedings of the 17th National Conference on Artificial Intelligence*, 476–482.

Kartha, G. 1993. Soundness and completeness theorems for three formalizations of action. In *Proceddings of the 13th International Joint Conference on Artificial Intelligence*, 724–729.

Lakemeyer, G. 1999. On sensing and off-line interpreting in golog. In *Logical Foundations for Cognitive Agents, Contributions in Honor of Ray Reiter*, 173–187.

Levesque, H.; Reiter, R.; Lesperance, Y.; Lin, F.; and Scherl, R. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming* 31(1-3):59–84.

Lifschitz, V., and Turner, H. 1999. Representing transition systems by logic programs. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning*, 92–106.

Lifschitz, V., ed. 1997. *Special issue of the Journal of Logic Programming on Reasoning about actions and change*, volume 31(1-3).

Lifschitz, V. 1999. Answer set planning. In *International Conf. on Logic Programming*, 23–37.

Marek, V., and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-year Perspective*, 375–398.

McCain, N., and Turner, M. 1997. Causal theories of action and change. In *Proceedings of the 14th National Conference on Artificial Intelligence*, 460–467.

McIlraith, S. 2000a. Intergrating actions and state constraints: A closed-form solution to the ramification problem (sometimes). *Artificial Intelligence* 116:87–121.

McIlraith, S. 2000b. Modeling and programming devices and web agents. In *Proceedings of the NASA Goddard Workshop on Formal Approaches to Agent-Based Systems, LNCS, Springer-Verlag*.

Niemelä, I., and Simons, P. 1997. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Proc. ICLP & LPNMR*, 420–429.

Niemelä, I.; Simons, P.; and Soininen, T. 1999. Stable model semantics for weight constraint rules. In *Proceedings of the 5th International Conference on on Logic Programming and Nonmonotonic Reasoning*, 315–332.

Niemelä, I. 1999. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25(3,4):241–273.

Reiter, R. 1998. KNOWLEDGE IN ACTION: Logical Foundations for Describing and Implementing Dynamical Systems. *http://www.cs.toronto.edu/~cogrobo*

Simons, P. 1999. Extending the stable model semantics with more expressive rules. In *Proceedings of the 5th International Conference on on Logic Programming and Nonmonotonic Reasoning*, 305–316.