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 as 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 framework to solve constraint satisfaction problems in logic programming (Marek & Truszczyński 1999; Niemelä 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 planning 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 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 II is defined over a first-order language $L_P$, 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 $L_P$. A program II is a set of rules of the form

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

(1)
where \( 0 \leq m \leq n \), each \( l_i \) is an atom, and, \( \neg \) represents the negation-as-failure operator.

Let \( \mathcal{l}(\Pi) \) denote the set of ground atoms in the language of the program \( \Pi \).

1. If \( \Pi \) does not contain any naf-literal (i.e. \( m = n \) in every rule of \( \Pi \)) a stable models of \( \Pi \) is defined as the smallest set \( S \), \( S \subseteq \mathcal{l}(\Pi) \) such that for any ground instance \( l_0 \leftarrow l_1, \ldots, l_m \) of a rule from \( \Pi \), if \( l_1, \ldots, l_m \in S \), then \( l_0 \in S \).

2. If the program \( \Pi \) does contain some naf-literals (\( m < n \) in some rule of \( \Pi \)), \( S \subseteq \mathcal{l}(\Pi) \) is a stable model of \( \Pi \) if \( S \) is the stable model of the program \( \Pi' \) obtained from the set of all ground instances of \( \Pi \) by deleting

(a) each rule that has a formula \( \neg l \) in its body with \( l \in S \), and

(b) all formulas of the form \( \neg l \) in the bodies of the remaining clauses.

We note that in ASP, a new type of rules called constraints with empty head (or \( l_0 = \text{false} \)) 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 \( B \)

We use the high-level action description language \( B \) 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:

\[
\text{caused}\{p_1, \ldots, p_n\}, f \quad (2) \\
\text{causes}(a, f, \{p_1, \ldots, p_n\}) \quad (3) \\
\text{executable}(a, \{p_1, \ldots, p_n\}) \quad (4) \\
\text{initially } f \quad (5)
\]

where \( f \) and \( p_i \)'s are fluent literals (a fluent literal is either a fluent \( g \) or its negation \( \neg g \), written as \( \text{neg}(g) \)) 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 \( (D, \Gamma) \) where \( D \) consists of propositions of the form (2)-(4) and \( \Gamma \) 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 \( \{s, a, s'\} \in t \), then there exists a link with label \( a \) from state \( s \) to state \( s' \). A trajectory of the system is denoted by a sequence \( s_0, a_1, s_1 \ldots a_m, s_m \) where \( s_i \)'s are states and \( a_i \)'s are actions and \( \{s_i, a_{i+1}, s_{i+1}\} \in t \) for \( t \in \{0, \ldots, n - 1\} \).

### Answer Set Planning

A planning problem is specified by a triple \( \langle D, \Gamma, \Delta \rangle \) where \( (D, \Gamma) \) is an action theory and \( \Delta \) 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 \( \Delta \) if there exists a trajectory \( s_0, a_1, s_1 \ldots a_m, s_m \) such that \( s_0 \) and \( s_m \) satisfies \( \Gamma \) and \( \Delta \), 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 \( \{s, a, s'\} \in t \), then every possible plan for \( \Delta \) is also a plan for \( \Delta \).

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 \( \Pi \), for short) consisting of domain-dependent rules that describe \( D, \Gamma, \) and \( \Delta \) respectively, and domain-independent rules that generate action occurrences and represent the transitions between states.

### Action theory representation

We use two predicates \( \text{set} \) and \( \text{in} \) 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 \( \text{nil} \) denotes the set \( \{\} \). A set of literals \( \{p_1, \ldots, p_n\} \) will be replaced by the set of atoms \( Y = \{\text{set}(s), \text{in}(p_1, s), \ldots, \text{in}(p_n, s)\} \) 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 \( \Pi \). For example, a proposition \( \text{causes}(a, f, \{p_1, \ldots, p_n\}) \) with \( n > 0 \) is encoded as a set of atoms consisting of \( \text{causes}(a, f, s) \) and the set \( Y \) (\( s \) is the name assigned to \( \{p_1, \ldots, p_n\} \)).

### Goal representation

To encode \( \Delta \), 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 \( \text{formula}(\forall x_1, \ldots, x_n) \leftarrow \text{in}(X_1, s_1), \ldots, \text{in}(X_n, s_n) \) where \( f \) is the name assigned to it. As with literal, negation will be represented by the function symbol \( \text{neg} \). 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 \( \Pi \) 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:

- \( \text{holds}(L, T) \): \text{L} holds at time \( T \),
- \( \text{possibly}(A, T) \): action \( A \) is executable at time \( T \),
- \( \text{occ}(A, T) \): action \( A \) occurs at time \( T \),
- \( \text{holds}\_\text{formula}(\varphi, T) \): formula \( \varphi \) holds at time \( T \), and
- \( \text{holds}\_\text{set}(S, T) \): \( S \) - 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 \( \text{time} \), \( L, G, A \) are variables denoting fluent literals (written as \( F \) or \( \neg(F) \) for some fluent \( F \)), \( S \) is a variable set of the sort \( \text{set} \), and \( A, B \) are variables of the sort \( \text{action} \).

\[
\begin{align*}
\text{holds}(L, T+1):&= \\
&\text{occ}(A, T), \text{causes}(A, L, S), \\
&\text{holds}\_\text{set}(S, T).
\end{align*}
\]

\[
\begin{align*}
\text{holds}(L, T):&= \\
&\text{causes}(S, L), \text{holds}\_\text{set}(S, T).
\end{align*}
\]

\[
\begin{align*}
\text{holds}(L, T+1):&= \\
&\text{contrary}(L, G), \\
&\text{holds}(G, T), \neg\text{holds}(G, T+1).
\end{align*}
\]

\[
\begin{align*}
\text{possible}(A, T):&= \\
&\text{executable}(A, S), \\
&\text{holds}\_\text{set}(S, T).
\end{align*}
\]

\[
\begin{align*}
\text{holds}(L, 0):&= \\
&\text{literal}(L), \\
&\text{initially}(L).
\end{align*}
\]

\[
\begin{align*}
\text{occ}(A, T):=& \\
&A \neq B, \text{occ}(B, T), T < \text{length}.
\end{align*}
\]

\[
\begin{align*}
\text{occ}(A, T):=& \\
&T < \text{length}, \\
&\text{possible}(A, T), \neg\text{occ}(A, T).
\end{align*}
\]

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. \( \text{length} \) 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 \( \Pi_n(\Delta) \) (or \( \Pi_n \) when it is clear from the context what \( \Delta \), \( \Gamma \), and \( \Delta \) 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 \( \Gamma \), and
- the rule \( \langle \not f \rangle_n \) that encodes the requirement that \( \Delta \) 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 \( \Pi_n(\Delta, \Gamma) \).

**Theorem 1** Given a planning problem, \( \langle D, \Gamma, \Delta \rangle \). Let \( S \) be a stable model of \( \Pi_n(\Delta, \Gamma) \), and define \( s(i) = \{ f \mid \text{holds}(f, i) \in S \} \) and \( A[i, j] = \{ a_1, \ldots, a_l \} \) where \( i, j \) are integers, \( f \) is a fluent, \( a_i \) is an action, and for every \( t, i \leq t \leq j, \text{occ}(a_i, t) \in S \), 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 \( \Pi_n \) 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 \( \Pi_n \) 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 \( \langle D, \Gamma \rangle \) we define

- an action \( a \) is a program,
- a formula \( \phi \) is a program,
- if \( p_1 \)’s are programs then \( p_1; \ldots; p_n \) is a program,
- if \( p_1 \)’s are programs then \( \langle p_1; \ldots; p_n \rangle \) is a program,
- if \( p_1 \) and \( p_2 \) are programs and \( \phi \) is a formula then \( \text{if } \phi \text{ then } p_1 \text{ else } p_2 \) is a program,
- if \( \phi \) is a program and \( \psi \) is a formula then “while \( \phi \) do \( \psi \)” is a program, and

\(^1\)http://www.ksl.stanford.edu/people/son/macros.html
The rules for the *trans* predicate are listed below.

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

\[ \text{trans}(P,Tb,Te) :\]  
\[ \text{time}(Tb), \text{time}(Te), \text{time}(Te1), \]  
\[ Tb \leq Te, \text{time}(P1), \text{time}(P2), \]  
\[ \text{time}(P1,Tb,Te1), \text{time}(P2,Te1,Te). \]  

\[ \text{trans}(A,Tb,Tb+1) :\]  
\[ \text{time}(Tb), \text{action}(A), \text{neg}(A, \text{null}), \text{occ}(A,Tb). \]  

\[ \text{trans}(null,Tb,Tb) :\]  
\[ \text{time}(Tb). \]  

\[ \text{trans}(N,Tb,Te) :\]  
\[ \text{time}(Tb), \text{time}(Te), \]  
\[ \text{time}(P1,Tb,Te). \]  

\[ \text{trans}(F,Tb,Tb) :\]  
\[ \text{time}(Tb), \text{formula}(F), \text{holds_formula}(F,Tb). \]  

\[ \text{trans}(I,Tb,Te) :\]  
\[ \text{time}(Tb), \text{time}(Te), \]  
\[ \text{time}(P1,Tb,Te). \]  

\[ \text{trans}(W,Tb,Te) :\]  
\[ \text{time}(Tb), \text{time}(Te), \]  
\[ \text{time}(W,F,P), \text{time}(Tel), \text{time}(P1,Tb,Te). \]  

\[ \text{trans}(W,Tb,Tb) :\]  
\[ \text{time}(Tb), \text{time}(W,F,P), \text{time}(Tel), \text{time}(P1,Tb,Te). \]  

\[ \text{trans}(S,Tb,Te) :\]  
\[ \text{time}(Tb), \text{time}(Te), \]  
\[ \text{time}(W,F,P), \text{time}(Tel), \text{time}(P1,Tb,Te). \]  

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).

% fluent 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(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).
head(t1((N), turnoff(N)):- floor(N).
tail(t1((N), t2):- floor(N).
head(t2, open). tail(t2, close).

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
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.

Our implementation can be used only with finite action theories without static causal laws, the logic programming implementation and Golog, we prove that for action theories without static causal laws, the logic programming implementation and Golog, we prove that for answer set planning and PROLOG query answering procedure.

As a final point in our comparison between our answer set planning and PROLOG query answering procedure.

3. Our implementation is 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).

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.

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:

1. 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.

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 action theories, provided that the successor state axioms of fluents can be encoded in such a way that loops can be avoided.

3. Our implementation is 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).

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 planning and PROLOG query answering procedure.

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:

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

---

4E.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 \(-G(S_0), \forall n [H(n, S_0)]\), 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 \(D(o([A^*; G]^?)\), 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 $\Gamma$, 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 $\text{trans}(P,0,n) \in S$ then $D^T \models \text{Do}(P, S_0, \text{do}([A[0,n], S_0]))$.

**Theorem 3** For every program $P$ and action sequence $a_0,\ldots,a_n$ such that $D^T \models \text{Do}(P, S_0, \text{do}([a_0,\ldots,a_n], S_0))$, there exists a stable model $S$ of $G_n$ such that $\text{trans}(P,0,n) \in S$ and $\text{occ}(a_1,i) \in S$.

### 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 planning 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 planning 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.

\*The intuitive meaning of $D^T \models \text{Do}(P, S_0, \text{do}([a_1,\ldots,a_n], s))$ is that the program $P$, starting execution in situation $S_0$ will legally terminate in situation $\text{do}([a_1,\ldots,a_n], s)$ which is a shorthand of the situation $\text{do}(a_n, \text{do}([\ldots, \text{do}(a_1, S_0)]))$. More on GOLOG axioms and definitions can be found in (Levesque et al. 1997; Reiter 1998) etc.

### References


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.


