Planning with Action Languages: Perspectives using CLP(FD) and ASP

Agostino Dovier[1], Andrea Formisano[2], and Enrico Pontelli[3]

[1] Univ. di Udine, Dip. di Matematica e Informatica. dovier@dimi.uniud.it
[2] Univ. di L'Aquila, Dip. di Informatica. formisano@di.univaq.it
[3] New Mexico State University, Dept. Computer Science. epontell@cs.nmsu.edu

Abstract. Action description languages, such as $\mathcal{A}$ and $\mathcal{B}$ (Gelfond and Lifschitz, 1998), are expressive tools introduced for formalizing planning domains and problems. In this work, we investigate two alternative approaches to the problem of encoding action languages using logic programming. Starting from a slight variation of $\mathcal{B}$, we explore the use of *Answer Set Programming (ASP)* and *Constraint Logic Programming over Finite Domains (CLP(FD))* in processing action theory specifications. As regards ASP we present a Prolog translator from action theory into *lparse*'s syntax. Concerning CLP(FD), we describe a program that directly processes a specification in the $\mathcal{B}$ language.

1 Introduction

Building intelligent agents that can be effective in real-world environments has been a goal of researchers from the very first days of Artificial Intelligence (AI). It has long been recognized that such an agent must be able to *acquire*, *represent*, and *reason* with knowledge. As such, a *reasoning component* has been an inseparable part of any agent architecture in the literature.

Although the underlying representations and implementations may vary between agents, the reasoning component of an agent is often responsible for making decisions that are critical to its existence. As described in [10], such a component consists of a knowledge base, a plan database, and a search engine. The knowledge base stores domain models as well as heuristic knowledge, while the plan database consists of plan skeletons. The domain model is used to validate plans while the heuristic information, the planning database, and experts' knowledge help the search engine in quickly deriving plans. In addition, frameworks for planning and reasoning require the ability to deal with real-world knowledge, e.g., domains with actions with durations, actions with bounded resources, and temporally extended goals. Thus, the construction of intelligent agents requires a knowledge representation language capable of expressing various types of knowledge, such as domain and commonsense knowledge, and methodologies for reasoning with such knowledge.

Logic programming languages offer many attributes that make them suitable as knowledge representation languages. Their declarative nature allows for the modular development of provably correct reasoning modules [3]. Recursive definitions can be easily expressed and reasoned upon. Control knowledge and heuristic information can

be declaratively introduced in the reasoning process. Furthermore, many logic programming languages offer a natural support for nonmonotonic reasoning, which is considered essential for commonsense reasoning. These features, along with the presence of fast solvers [2, 15, 19, 9], make logic programming an attractive paradigm for knowledge representation and reasoning.

In the context of knowledge representation and reasoning, one of the most commonly explored application of logic programming has been in the domain of reasoning about actions and change [3]. Planning problems have been effectively encoded using *Answer Set Programming (ASP)* [3]—where distinct answer sets represent different trajectories leading to the desired goal. CLP(FD) has been used less frequently for planning problems (e.g., [13, 21]). Our encoding has some similarities to the one presented in [13] although we rely on CLP instead of CSP and our action language supports statical causal laws and non-determinism, while the work of Lopez and Bacchus is restricted to STRIPS.

A planning problem is based on the notions of *State*, *Action*, and *Fluent*. Recent works on representing and reasoning about actions and change have relied on the use of concise and high-level languages, commonly referred to as *action description languages*, e.g., the languages $\mathcal{A}$ and $\mathcal{B}$ [8]. Action languages allow us to write propositions that describe the effects of actions on states, and to create queries to infer properties of the underlying transition system.

In our experiments, we use a slight variation of $\mathcal{B}$, based on a syntax similar to the one used in [20]. An *action theory* is a specification of a planning problem using the action language.

The purpose of this paper is to investigate and compare two alternative approaches to the problem of encoding action languages using logic programming. In particular, we explore the use of two flavors of logic programming, *Answer Set Programming (ASP)* [14, 12, 17] and *Constraint Logic Programming over Finite Domains (CLP(FD))* [15, 2]. More specifically, regarding CLP(FD), we report a program that directly processes an action theory specification. Concerning the ASP solution, we developed a translator, written in Prolog, that, given an action theory specification, produces a suitable input file for *lparse*—the translation process follows the general guidelines highlighted, for example, in [20, 7].

The ultimate goal of this study is to investigate the relative strengths and weaknesses of the two approaches. While ASP has been widely used in the context of planning, the use of CLP(FD) has been more limited. Apart from the issue of performance, our interest is in exploring declarativeness and potential to handle extensions of traditional action languages.

2 An Action Description Language

Planning domains can be effectively described using high-level *action description languages*, e.g., the languages $\mathcal{A}$ and $\mathcal{B}$ [8]. In our experiments, we use a slight variation of $\mathcal{B}$, based on a syntax similar to the one used in [20]. With a slight abuse of notation, we simply refer to the language used in this paper as $\mathcal{B}$. The readers are referred to [12, 20, 8] for the theoretical foundations of action languages.

A planning problem can be described defining the notions of states, actions, and fluents and the relationships between these notions. In particular

- `Fluents`: they are *atomic formulae*, describing the state of the world, and whose truth value can change over time.
- `State`: a state is a possible configuration of the domain of interest; in particular, a state is described by an assignment of truth values to the fluents.
- `Actions`: they affect the state of the world, and thus allow the transition from a state to another.

Intuitively, starting from an initial state, a planner tries to successively apply actions, transitioning to other states, until a state (*final state*) meeting certain given conditions is reached.

As a concrete syntax, fluents and actions are atomic formulae $p(t_1, \dots, t_n)$ $(n \geq 0)$ from an underlying logic language $\mathcal{L}$. For the sake of simplicity, we assume that the Herbrand universe built using $\mathcal{L}$ is finite (e.g., either there are no function symbols, or the use of functions symbols is restricted to avoid the creation of arbitrary complex terms).

In the definition of an action theory, an assertion of the kind `fluent(f)` declares that `f` is a fluent. Fluent literals are constructed from fluents and their negations (denoted by `mneg(f)`). Declarations of the form `action(a)` are used to describe the possible actions (in this case, `a`). In these declarations, f and a are ground atomic formulae of $\mathcal{L}$.

Following the general principle of *domain predicates* of ASP [19], we assume that the admissible terms that can be used in fluents and actions are classified in different categories, described by facts (e.g., lines (1)–(3) of the example in Figure 1).

The language $\mathcal{B}$ allows one to specify an *action theory*, which relates actions, states, and fluents using predicates of the following forms:

- `executable(a, [list-of-preconditions])`
  asserting that the given preconditions have to be satisfied in the current state in order for the action `a` to be executable.
- `causes(a, f, [list-of-preconditions])`
  encodes a dynamic causal law, describing the effect (the fluent literal `f`) of the execution of action `a` in a state satisfying the given preconditions.
- `caused([list-of-preconditions], f)`
  describes a static causal law—i.e., the fact that the fluent literal `f` is true in a state satisfying the given preconditions.

(where `[list-of-preconditions]` denotes a list of fluent literals). As an example, see lines (7)–(26) of the example in Figure 1.

A specific instance of a planning problem contains also a description of the initial state and of the desired goal:

- `initially(f)`
  asserts that the fluent literal `f` is true in the initial state. In our examples, we assume that the initial state is *complete*—i.e., we have knowledge of the truth value of each fluent in the initial state.
- `goal(f)`
  asserts that the goal requires the fluent literal `f` to be true in the final state.

For an instance, see lines (27)–(34) in Figure 1.

A *trajectory* is a sequence $s_0 a_1 s_1 a_2 \ldots a_n s_n$, where $a_1, \ldots, a_n$ are actions and $s_0, \ldots, s_n$ are states. The actions $a_1, \ldots, a_n$ of a trajectory represent a *plan* if s_0 is the initial state and, for each `goal(f)` in the action theory we have that the fluent f is true in s_n. We assume that the length of the desired plan is given and we disallow parallel actions. Intuitively, we look for plans that go from the initial state to the final state crossing a fixed number of state. Each state transition involves exactly one action execution.

Let us discuss in much detail the above referred example, as reported in Figure 1. There are three authors and four places (lines (1)–(3)). The fluents describe whether the authors are alive, whether they are armed, and whether they are staying at a particular location (lines (2)–(6)). The actions are `shoot` and `move` (lines (7)–(8)).

```
(1)     author(andy). author(ago). author(rico).
(2)     place(udine). place(laquila).
(3)     place(paris). place(lascruces).

(4)     fluent(alive(A)):-author(A).
(5)     fluent(armed(A)):-author(A).
(6)     fluent(stay(A,P)):-author(A),place(P).

(7)     action(shoot(A,P)):-author(A),place(P).
(8)     action(move(A,P)):-author(A),place(P).

(9)     causes(shoot(A,P),mneg(alive(B)),[stay(B,P)]):-
(10)       author(A),author(B),place(P).
(11)    causes(shoot(A,P),mneg(armed(A)),[]) :-
(12)       author(A),place(P).
(13)    causes(move(A,P),stay(A,P),[] ):-
(14)       author(A),place(P).

(15)    caused([stay(A,P1)],mneg(stay(A,P2))):-
(16)       author(A),place(P1),place(P2),diff(P1,P2).
(17)    caused([stay(A,P)],mneg(stay(B,P)) ):-
(18)       author(A),author(B),place(P),diff(A,B).

(19)    executable(shoot(A,P),[armed(A),alive(A)]):-
(20)       author(A),place(P).
(21)    executable(move(A,P2),
(22)           [alive(A),stay(A,P1),mneg(stay(A,P2)),
(23)           mneg(stay(B,P2)),mneg(stay(C,P2))]):-
(24)       author(A),author(B),author(C),
(25)       place(P1),place(P2),
(26)       diff(P1,P2),diff(A,B,C).

(27)    initially(stay(andy,laquila)).
(28)    initially(stay(ago,udine)).
(29)    initially(stay(rico,lascruces)).
(30)    initially(armed(A)):-author(A).
(31)    initially(alive(A)):-author(A).

(32)    goal(mneg(armed(A))):-author(A).
(33)    goal(alive(rico)).
(34)    goal(stay(andy,paris)).
```

Fig. 1. Action Theory: The fighting authors

An author `A` can shoot in the direction of a place `P` if he is armed and alive. An author `A` can move to a place `P2` if he is alive and none of the other authors is currently

located in `P2`. These conditions represent the executability conditions of the actions, reported in lines (19)–(26).

The actions effects are described in lines (9)–(14). If an author `A` shoots in the direction of place `P`, and `B` is in the location `P`, then `B` is no longer alive (this also means that he cannot move away from `P`). If an author `A` shoots, then after this action he will no longer be armed. If, instead, the author `A` moves to the place `P`, then he will stay in `P`.

Lines (15)–(18) handle the static causal law of the action theory. If an author is in a place, then he cannot be located at any other place. If an author is in one place (alive or not) then no other author can stay in the same place.

Initially (lines (27)–(31)) the authors are armed, alive and located at their home universities). The final goal is that nobody is armed, that the author `rico` is alive, and that author `andy` is located in `paris`. A possible plan that meets the requirement is:

1. shoot(rico, paris)   2. shoot(andy, paris)
3. shoot(ago, udine)   4. move(andy, paris)

which includes the suicide of the oldest author. For other examples see `www.di.univaq.it/~formisano/CLPASP`.

3 ASP

There are several ways to encode an action theory in ASP (e.g., [12, 3]). We implemented a Prolog translator from $\mathcal{B}$ to the *lparse*'s syntax [19]. The translation is relatively straightforward, and we do not enter here in its details. Most of the translation amounts to syntactical rewriting of the action theory specification to suitable ASP rules. The interested reader can obtain the Prolog program of the translator from [5], together with the action theory specifications we used.

The main predicate of such Prolog translator is `main(Input,Len)`. The user must provide the name of an input file containing an action theory specification (`Input`), and the exact length of the desired plan (`Len`). In Figure 2, we show an excerpt of the ASP specification produced by the translator for the action theory describing the problem of the fighting authors (Figure 1). In this case we used `Len=4`. Let us comment on some features of this code:

- line (1) is used to denote the legal time-steps of the plan (from 0 to 4);
- line (4) contains the description of the initial state;
- the constraint in line (5) is used to generate exactly one action occurrence for each time step; similarly, the constraint in line (6) imposes that the action chosen at each time step is executable at that time step;
- a fluent literal ℓ holds at time T if `hold(`ℓ`,T)` is true. The conditions that make `hold(`ℓ`,T)` true are described by the rules in lines (7)–(10); the initial state is defined in line (7), line (8) describes the direct effects of an action, line (9) describes the effect of static causal laws, while line (10) encodes the effect of inertia.
- lines (11)–(13) describe the executability conditions;
- lines (14)–(17) identify the direct effects of each action, while lines (18)–(21) encode the preconditions of the various actions;

◦ lines (22)–(24) encode the static causal laws;
◦ lines (25) and (26) represent the goal;
◦ lines (27)–(31) define some auxiliary rules.

```
(1)     time(0..4).
(2)     fluent(alive(ago)). ... fluent(armed(rico)).
(3)     action(move(ago,laquila)). ... action(shoot(rico,udine)).
(4)     initially(alive(ago)). ... initially(stay(rico,lascruces)).
(5)     1{occ(Act,Ti):action(Act)}1 :- time(Ti), Ti < 4.
(6)     :- occ(Act,Ti), action(Act), time(Ti), not exec(Act,Ti).
(7)     hold(Fl,0) :- initially(Fl).
(8)     hold(Fl,Ti+1) :- time(Ti), literal(Fl), occ(Act,Ti),
        causes(Act,Fl), ok(Act,Fl,Ti), exec(Act,Ti).
(9)     hold(Fl,Ti) :- time(Ti), literal(Fl), caused(Ti,Fl).
(10)    hold(Li,Ti+1) :- time(Ti), literal(Li), hold(Li,Ti),
        opposite(Li,Lu), not hold(Lu,Ti+1).
(11)    exec(move(ago,laquila),Ti) :-
        time(Ti),hold(alive(ago),Ti),hold(stay(ago,udine),Ti),
        hold(mneg(stay(ago,laquila)),Ti),hold(mneg(stay(andy,laquila)),Ti),
        hold(mneg(stay(rico,laquila)),Ti).
(12)    ...
(13)    exec(shoot(ago,laquila),Ti) :-
        time(Ti),hold(armed(ago),Ti),hold(alive(ago),Ti).
(14)    causes(shoot(andy,udine),mneg(alive(andy))).
(15)    causes(shoot(andy,udine),mneg(armed(andy))).
(16)    ...
(17)    causes(move(rico,udine),stay(rico,udine)).
(18)    ok(shoot(andy,udine),mneg(alive(andy)),Ti) :- time(Ti),
        hold(stay(andy,udine),Ti).
(19)    ok(shoot(andy,udine),mneg(armed(andy)),Ti) :- time(Ti).
(20)    ...
(21)    ok(move(rico,udine),stay(rico,udine),Ti) :- time(Ti).
(22)    caused(Ti,mneg(stay(andy,laquila))) :- time(Ti),
        hold(stay(andy,udine),Ti).
(23)    ...
(24)    caused(Ti,mneg(stay(ago,lascruces))) :- time(Ti),
        hold(stay(rico,lascruces),Ti).
(25)    :- not goal.
(26)    goal :- hold(alive(rico),4), hold(mneg(armed(ago)),4),
        hold(mneg(armed(andy)),4), hold(mneg(armed(rico)),4),
        hold(stay(andy,paris),4).
(27)    literal(Fl) :- fluent(Fl).
(28)    literal(mneg(Fl)) :- fluent(Fl).
(29)    opposite(Fl,mneg(Fl)) :- fluent(Fl).
(30)    opposite(mneg(Fl),Fl) :- fluent(Fl).
(31)    :- time(Ti), fluent(Fl), hold(Fl,Ti), hold(mneg(Fl),Ti).
```

Fig. 2. Translation in ASP of the fighting authors

4 CLP(FD)

The proposed CLP(FD) encoding of a planning problem uses the following representation. A plan with exactly N states, p fluents, and m actions is represented by:

◦ A list, called `States`, containing N lists, each composed of p terms of the type `fluent(fluent_name, Bool_var)`. The variable of the ith term in the jth list is

assigned 1 if and only if the ith fluent is true in the jth state of the trajectory. For example, if we have $N = 3$ and the fluents `f`, `g`, and `h`, we have:

```
States = [[fluent(f,X_f_1),fluent(g,X_g_1),fluent(h,X_h_1)],
          [fluent(f,X_f_2),fluent(g,X_g_2),fluent(h,X_h_2)],
          [fluent(f,X_f_3),fluent(g,X_g_3),fluent(h,X_h_3)]]
```

- A list `ActionsOcc`, containing $N - 1$ lists, each composed of m terms of the form `action(action_name,Bool_var)`. The variable of the ith term of the jth list is assigned 1 if and only if the ith action occurs during the transition from state j to state $j + 1$. For example, if we have $N = 3$ and the actions `a` and `b`, then:

```
ActionsOcc = [[action(a,X_a_1),action(b,X_b_1)],
              [action(a,X_a_2),action(b,X_b_2)]]
```

The planner will make use of this structure in the construction of the plan; appropriate constraints are set between the various boolean variables to capture their relationships (e.g., for each list in `ActionsOcc`, exactly one `action(action_name,Bool_var)` can contain a variable that is assigned the value 1).

We explain below the main parts of the CLP interpreter for the $\mathcal{B}$ language we developed. The interpreter assumes that the action theory is present in the Prolog database—observe that the syntax adopted (see, e.g., Figure 1) is compliant with Prolog's syntax, thus allowing us to directly store the action theory as rules and facts in the Prolog database.

The entry point of the planner is shown in Figure 3. The main predicate is `main(N)` (line (1)). It computes a plan of length `N` for the action theory present in the Prolog database. Lines (2) and (3) collect the lists of fluents (`Lf`) and actions (`La`), the description of the initial state (`Init`) and the required content of the final state (`Goal`).

Lines (4) and (5) calls the predicates for defining the lists `States` and `ActionsOcc`, as explained above. These predicates are defined in lines (12)–(26). Observe that all variables for fluents are declared as boolean variables in line (18); furthermore, in every state transition, exactly one action can be fired (line (23)).

The predicates in lines (6) and (7) handle the knowledge about the initial and the goal states, respectively. Lines (8) and (9) impose the constraints on state transitions and action executability. Line (10) gathers all variables denoting action occurrences, in preparation for the labeling phase (line (11)). Note that the labeling is focused on the selection of the action to be executed at each time step. Please observe that in the code of Figure 3 we omit the parts concerning delivering the results to the user.

In Figure 4, we report the definition of the predicates employed to assign the correct truth values to the fluents in the initial state (predicate `set_initial`) and to the desired fluents in the final state (predicate `set_goal`). The predicate `set_one_static_fluent` is used to perform a declarative closure operation on a state by using the static causal laws of the action theory (i.e., the rules of type `caused` in the action theory). In particular, this is realized by propagating information from the fluents whose truth value has already been established. It is a particular case of predicate `set_one_fluent` explained later, and therefore we do not list here its definition.

The core of the planning process is carried out by the predicates `set_transitions` and `set_executability` (lines (8)–(9)). The code for these predicates is reported in Figure 5. In particular, lines (43)–(50) recursively define the predicate `set_transitions`,

```
(1)     main(N,ACTIONSOCC,STATES):-
(2)        setof(F,fluent(F),Lf), setof(A,action(A),La),
(3)        setof(F,initially(F),Init), setof(F,goal(F),Goal),
(4)        make_states(N,Lf,STATES),
(5)        make_action_occurrences(N,La,ACTIONSOCC),
(6)        set_initial(Init,STATES),
(7)        set_goal(Goal,STATES),
(8)        set_transitions(ACTIONSOCC,STATES),
(9)        set_executability(ACTIONSOCC,STATES),
(10)       get_all_actions(ACTIONSOCC,AllActions),
(11)       labeling([ff],AllActions).

(12)    make_states(0,_,[]):-!.
(13)    make_states(N,List,[S|STATES]) :-
(14)       N1 is N-1, make_states(N1,List,STATES),
(15)       make_one_state(List,S).
(16)    make_one_state([],[]).
(17)    make_one_state([F|Fluents],[fluent(F,VarF)|VarFluents]) :-
(18)       make_one_state(Fluents,VarFluents), VarF in 0..1.

(19)    make_action_occurrences(1,_,[]):-!.
(20)    make_action_occurrences(N,List,[Act|ActionsOcc]) :-
(21)       N1 is N-1, make_action_occurrences(N1,List,ActionsOcc),
(22)       make_one_action_occs(List,Act),
(23)       get_action_list(Act,AList), sum(AList,#=,1).
(24)    make_one_action_occs([],[]).
(25)    make_one_action_occs([A|Acts],[action(A,OccA)|OccActs]) :-
(26)       make_one_action_occs(Acts,OccActs), OccA in 0..1.
```

Fig. 3. Entry Point of the CLP(FD) Planner

which ultimately relies on `set_one_fluent` to constrain all boolean variables related to each possible fluent in each possible state transition (i.e., action execution).

```
(27)    set_initial(List,[InitialState|_]):-
(28)       set_state(List,InitialState),
(29)       complete_state(InitialState,InitialState).
(30)    set_goal(List,States):-
(31)       last(States,FinalState), set_state(List,FinalState).
        set_state([],_).
(32)    set_state([Fluent|Rest],State :-
(33)       (Fluent=mneg(F),!,member(fluent(F,0),State);
(34)       member(fluent(Fluent,1),State)),
(35)       set_state(Rest,State).
(36)    complete_state([],_).
(37)    complete_state( [fluent(Fluent,EV)| Fluents],InitState):-
(38)       (integer(EV), ! ;
(39)       set_one_static_fluent(Fluent,EV,InitState) ),
(40)       complete_state( Fluents ,InitState).
(41)    set_one_static_fluent(Name,EV,State) :-
(42)          .........
```

Fig. 4. Initial and Final State

Let us consider a state transition from state `FromSt` to state `ToSt`, as depicted in Figure 6, where we assume that there are p fluents $1, \ldots, p$ involved. As mentioned earlier, two boolean variables IV_i and EV_i (for $i = 1, \ldots, p$) denote whether the fluent

```
(43)    set_transitions(_,[_]):-!.
(44)    set_transitions([Occ|Occurrences],[S1,S2|Rest]) :-
(45)        set_transition(O,S1,S2,S1,S2),
(46)        set_transitions(Occurrences,[S2|Rest]).
(47)    set_transition(_,[],[],_,_).
(48)    set_transition(Occ,[fluent(F,IV)|R1],[fluent(F,EV)|R2],FromSt,ToSt):-
(49)        set_one_fluent(F,IV,EV,Occ,FromSt,ToSt),
(50)        set_transition(Occ,R1,R2,FromSt,ToSt).
(51)    set_one_fluent(Fl,IV,EV,Occ,FromSt,ToSt) :-
(52)        findall([X,L],causes(X,Fl,L),Pos),
            findall([Y,M],causes(Y,mneg(Fl),M),Neg),
(53)        build_sum_prod(Pos,Occ,FromSt,PFormula,EV,p),
(54)        build_sum_prod(Neg,Occ,FromSt,NFormula,EV,n),
(55)        findall(P,caused(P,Fl),StatPos),
            findall(N,caused(N,mneg(Fl)),StatNeg),
(56)        build_sum_stat(StatPos,ToSt,PStatPos,EV,p),
(57)        build_sum_stat(StatNeg,ToSt,PStatNeg,EV,n),
(58)        append(PFormula,PStatPos,Pos_Fl),
(59)        append(NFormula,PStatNeg,Neg_Fl),
(60)        sum(Pos_Fl,#=,Psum),
(61)        sum(Neg_Fl,#=,Nsum),
(62)        EV #<=> ((Psum + IV - IV * Nsum) #> 0).
(63)    build_sum_prod([],_,_,[],_,_).
(64)    build_sum_prod([[Action,Prec]|Rest],Occ,State,[Flag|PF1],EV,Mode):-
(65)        get_precondition_vars(Prec,State,ListPV),
(66)        length(Prec,NPrec), sum(ListPV,#=,SumPrec),
(67)        member(action(Action,VA),Occ),
(68)        (VA #= 1 #/\ (SumPrec #= NPrec)) #<=> Flag,"
(69)        (Mode == p -> EV #>= Flag; Mode==n -> EV #=< 1-Flag),
(70)        build_sum_prod(Rest,Occ,State,PF1,EV,Mode).
(71)    build_sum_stat([],_,[],_,_).
(72)    build_sum_stat([Cond|Others],State,[Flag|Fo],EV,Mode):-
(73)        get_precondition_vars(Cond,State,List),
(74)        length(List,NL), sum(List,#=,Result),
(75)        Flag #<=> (Result #= NL),
(76)        (Mode==p -> EV #>= Flag; Mode==n -> EV #=< 1-Flag),
(77)        build_sum_stat(Others,State,Fo,EV,Mode).
```

Fig. 5. Establishing Constraints among Fluent and Action Variables

i holds in `FromSt` and in `ToSt`, respectively. Let the variables VA_j, (for $j = 1, \ldots, m$) denote whether the action j occurs in such a state transition. In this situation, let us consider a generic call (line (49)) of the predicate `set_one_fluent` (defined in line (51)). For a given fluent `Fl`, the predicate `set_one_fluent` collects the list `Pos` (resp. `Neg`) of pairs `[Action,Preconditions]` such that `Action` makes `Fl` true (resp. false) in the

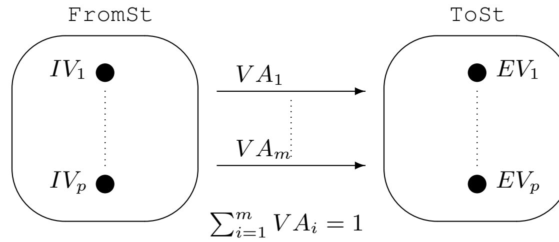

Fig. 6. Action constraints (see procedures `set_transitions`—`build_sum_prod`)

state transition (lines (52)). Similarly, it handles the static causal laws (`caused` assertions), by collecting the lists of conditions that affect the truth value of `Fl` (the variables `StatPos` and `StatNeg`, in lines (55)).

The CLP variables involved are then constrained by the procedures `build_sum_prod` (lines (53)–(54)) and `build_sum_stat` (lines (56)–(57)). Finally, all variables related to the introduction (respectively, removal) of fluents are collected in `Pos_Fl` (respectively `Neg_Fl`). Their sums are collected in variables `Psum` and `Nsum`, respectively. Further constraints in `build_sum_prod` and `build_sum_stat` ensure that these variables assume values that are greater than zero. Moreover, we take care of the inertia law in line (62): if none of the actions affect `Fl`, then $EV = IV$.

Let us focus on the predicate `build_sum_prod`. For the sake of simplicity, let us assume that it is called with `Mode == p` (line (53)). The predicate `build_sum_prod` recursively processes a list of pairs `[Action,Preconditions]`. For each action A_j, if A_j occurs and all of its preconditions (`Prec`) hold, then there is an action effect (`Flag = 1`, line (68)). Moreover, the constraint `Mode == p -> EV #>= Flag` in line (69), imposes that, if an action A_j occurs in a state transition (i.e., the corresponding boolean variable `VA_j` is 1), and such action makes a fluent true, then the boolean variable `EV` associated to it has to be 1. Analogous constraints are imposed in the case `Mode == n`, corresponding to the handling of actions that make a given fluent false (called in line (54)). A few remarks on line (62). A fluent is true in the next state (`EV`) if and only there is an action or a static causal law making it true (`Psum`) or it was true in the previous state (`IV`) and no causal laws make it false.

The modus operandi for imposing the executability conditions for actions is similar to what we have just described—and the corresponding code is listed in Figure 7. By using the predicate `set_executability` (defined in line (78)), all the preconditions of each action are considered. If the action is executed (`VA = 1`, see line (91)), then at least one of the executability conditions of that action holds (see predicates `get_all_preconditions` and `formula`—lines (98)–(105)). Lines (106)–(120) define some auxiliary predicates.

5 Experimentation

We have conducted some experiments to compare the performance of the two approaches to planning. In particular, the CLP(FD) planner has been implemented using SICStus Prolog, and using the `clpfd` library for constraint solving over finite domains [24]. The ASP-based planning approach has been developed using the `lparse` language, and the resulting encodings have been tested using two different systems: SMODELS [19] and CMODELS [11]. The first system is a native ASP solver, based on a variation of the Davis-Putnam procedure. The second system relies on a mapping of the problem of computing models to the problem of testing satisfaction of propositional theories (SAT solving). The advantage of this second approach is the ability to reuse efficient SAT solving technology; our experiments have been conducted using two different underlying SAT solvers—mChaff [16] and Simo [9].

As an example of the planning problems we experimented with, we describe a planning problem in the block world domain with N blocks (blocks $1, \dots, N$). In the initial

```
(78)    set_executability(ActionsOcc,States) :-
(79)        findall([Act,C],executable(Act,C),Conds),
(80)        group_cond(Conds,GroupedConds),
(81)        set_executability1(ActionsOcc,States,GroupedConds).
(82)    set_executability1([],[_],_).
(83)    set_executability1([AStep|ARest],[State|States],Conds):-
(84)        set_executability_sub(AStep,State,Conds),
(85)        set_executability1(ARest,States,Conds).
(86)    set_executability_sub(_,_,[]).
(87)    set_executability_sub(Step,State,[[Act,C]|CA]) :-
(88)        member(action(Act,VA),Step),
(89)        get_all_preconditions(C,State,NCs,Temps),
(90)        formula(NCs,Temps,Phi),
(91)        (VA #=1) #=> Phi #= 0,
(92)        set_executability_sub(Step,State,CA).
(93)    group_cond([],[]).
(94)    group_cond([[Action,C]|R],[[Action,[C|Cs]]|S]) :-
(95)        findall(L,(member([Action,L],R)),Cs),
(96)        filter(Action,R,Others),
(97)        group_cond(Others,S).
(98)    get_all_preconditions([],_,[],[]).
(99)    get_all_preconditions([C|R],State,[NC|NCs],[T|Temps]) :-
(100)       get_precondition_vars(C,State,Cs),
(101)       length(Cs,NC), sum(Cs,#=,T),
(102)       get_all_preconditions(R,State,NCs,Temps).
(103)   formula([],[],1).
(104)   formula([N|Rest],[V|Val],Phi) :-
(105)       Phi #= Phi1 #/\ (N #> V), formula(Rest,Val,Phi1).
(106)   get_precondition_vars([],_,[]).
(107)   get_precondition_vars([P1|Rest],State,[F|LR]) :-
(108)   get_precondition_vars(Rest,State,LR),
(109)       (P1 = mneg(FN),!,member(fluent(FN,A),State),F #= 1-A;
(110)       member(fluent(P1,F),State)).
(111)   get_all_actions([],[]).
(112)   get_all_actions([A|B],List) :-
(113)       get_action_list(A,List1),get_all_actions(B,List2),
(114)       append(List1,List2,List).
(115)   get_action_list([],[]).
(116)   get_action_list([action(_,V)|Rest],[V|MRest]) :-
(117)       get_action_list(Rest,MRest).
(118)   filter(_,[],[]).
(119)   filter(A,[[A,_]|R],S) :- !, filter(A,R,S).
(120)   filter(A,[C|R],[C|S]) :- !, filter(A,R,S).
```

Fig. 7. Executability Conditions

state, the blocks are arranged in a single stack, in increasing order, i.e., block 1 is on the table, block 2 is on top of block 1, etc. Block N is on top of the stack. In the goal state, there must be two stacks, composed of the blocks with odd and even numbers, respectively. In both stacks the blocks are arranged in increasing order, i.e., blocks 1 and 2 are on the table and blocks $N-1$ and N are on top of the respective stacks. The planning problem consists of finding a plan made of T actions. An additional restriction must be met: in each state at most three blocks can lie on the table.

Figure 8 displays the specification of an instance of the planning problem with $N = 5$ using the above described action language. In such specification, the blocks are defined in line (1). Lines (2)–(5) define the fluents of the problem. A block may lie on the table or on top of another block (fluent `on(X,Y)`). A block may be clear (if no other block is on top of it) or not. There may be space on the table for other blocks (fluent `space_on_table`). There are two possible moves (lines (6)–(7)). The `causes` rules (lines (8)–(22)) as well as the `caused` assertions (lines (23)–(27)) are of easy reading. Lines (28)–(31) specify the executability conditions for the two possible actions: one can move a block `X` on top of another block `Y` only if both are clear, and a block `X` can be moved to the table only if there is free space left (i.e., there are at most two blocks lying on the table). The initial state and the goal state are described in lines (32)–(40) and (41)–(43), respectively.

Table 1 reports the execution times from the three systems, for different number of blocks and plan lengths (i.e., the number of moves). All the timing results, expressed in seconds, have been obtained by measuring only the CPU usage time needed for computing the first solution, if any. In the case of the ASP solvers, we separately report the time needed by *lparse* to ground the programs. As regards CLP(FD), we used the `runtime` option to measure the execution time. Hence we do not account for the time spent in garbage collection and system calls. All tests have been performed on a PC (P4 processor 2.8 GHz and 512 MB RAM memory) running Linux.

We also experimented with a variant of the problem described above, where a further constraint is imposed: no block x can be placed on top of another block y if $y \geq x$. This constraint can be modeled in the action language, by replacing line `(6)` in Figure 8 with the following one:

```
(6') action(move(X,Y)):- blk(X), blk(Y), X>Y.
```

The results reported in Table 1 show that the ASP solvers can solve the instances (save for the easiest ones) more quickly than CLP(FD). Notice that, the CLP(FD) solution is general and applicable to every action theory described in the proposed action language. We have also tested the two approaches on other classical planning problems, such as for instance, the man-wolf-goat-cabbage problem, the missionaries and cannibals problem, the problem of finding best movements of elevators, Hanoi towers, obtaining similar results. Observe also that, in CLP(FD), a price is paid in making use of a declarative translation from a generic action theory to constraints; for example, a manual and ad-hoc encoding of the same planning problem presented here (using rule `(6')`) for 5 blocks and 13 actions founds the solution in 2ms (see [5] for this code and related running times). We expect similar improvements also from an ad-hoc encoding of planning problems using ASP.

```
(1)     blk(1). blk(2). blk(3). blk(4). blk(5).
(2)     fluent(on_table(X)):- blk(X).
(3)     fluent(clear(X)):- blk(X).
(4)     fluent(on(X,Y)):- blk(X), blk(Y), diff(X,Y).
(5)     fluent(space_on_table).
(6)     action(move(X,Y)):- blk(X), blk(Y), diff(X,Y).
(7)     action(to_table(X)):- blk(X).
(8)     causes(move(X,Y),clear(Z),[on(X,Z)]):- blk(X), blk(Y),
(9)        blk(Z), action(move(X,Y)),diff(X,Z),diff(Y,Z).
(10)    causes(move(X,Y),on(X,Y),[]):- blk(X), blk(Y),
(11)       action(move(X,Y)).
(12)    causes(move(X,Y),mneg(on(X,Z)),[on(X,Z)]):- blk(X),
(13)       blk(Y), blk(Z), action(move(X,Y)), diff(Y,Z).
(14)    causes(move(X,Y),space_on_table,[on_table(X)]):- blk(X),
(15)       blk(Y), action(move(X,Y)).
(16)    causes(to_table(X),on_table(X),[]):-
(17)       blk(X), action(to_table(X)).
(18)    causes(to_table(X),clear(Y),[on(X,Y)]):- blk(X), blk(Y),
(19)       diff(X,Y), action(to_table(X)).
(20)    causes(to_table(X),mneg(space_on_table),[on_table(Y),on_table(Z)]):-
(21)       blk(X), blk(Y), blk(Z), diff(X,Y,Z),
(22)       action(to_table(X)).
(23)    caused([on(X,Y)],mneg(clear(Y))) :- blk(X), blk(Y), diff(X,Y).
(24)    caused([clear(Y)],mneg(on(X,Y))) :- blk(X), blk(Y), diff(X,Y).
(25)    caused([on(X,Y)],mneg(on_table(X))) :- blk(X), blk(Y), diff(X,Y).
(26)    caused([on_table(X)],mneg(on(X,Y))) :- blk(X), blk(Y), diff(X,Y).
(27)    caused([on(X,Y)],mneg(on(Y,X))) :- blk(X), blk(Y), diff(X,Y).
(28)    executable(move(X,Y),[clear(X),clear(Y)]) :-
(29)       blk(X), blk(Y), action(move(X,Y)).
(30)    executable(to_table(X),[clear(X),mneg(on_table(X)),space_on_table]):-
(31)       blk(X), action(to_table(X)).
(32)    initially(clear(5)).
(33)    initially(mneg(clear(X))) :- blk(X), X<5.
(34)    initially(on_table(1)).
(35)    initially(mneg(on_table(X))) :- blk(X), X>1.
(36)    initially(space_on_table).
(37)    initially(on(X,Y)) :- blk(X), blk(Y), Y<5, X is Y+1.
(38)    initially(mneg(on(X,Y))):- blk(X), blk(Y), X<Y.
(39)    initially(mneg(on(X,Y))) :- blk(X), blk(Y), Y<5,
(40)       diff(Y,X), P is Y+1, diff(X,P).
(41)    goal(on(X,Y)) :- blk(X), blk(Y), Y<4, X is Y+2.
(42)    goal(on_table(1)).   goal(on_table(2)).
(43)    goal(space_on_table).
```

Fig. 8. Planning in blocks world

Instance (using (6)) Blocks	Length	Plan exists	*lparse*	SMODELS	CMODELS mChaff	Simo	SICStus
5	5	N	2.31	0.14	0.02	0.02	0.20
5	6	N	2.29	0.17	0.11	0.06	0.11
5	7	Y	2.34	0.21	0.12	0.10	0.08
6	7	N	7.64	0.32	0.16	0.13	0.31
6	8	N	7.65	0.37	0.19	0.15	1.70
6	9	Y	7.69	0.55	0.27	0.43	0.99
7	9	N	22.96	0.64	0.32	0.27	6.23
7	10	N	23.06	0.75	0.39	0.32	38.24
7	11	Y	23.10	2.15	0.57	1.35	17.40
8	11	N	36.71	1.18	0.63	0.53	154.96
8	12	N	36.81	1.92	0.74	0.62	948.31
8	13	Y	37.10	7.98	2.14	10.36	422.51
9	13	N	98.69	2.25	1.09	0.93	–
9	14	N	98.45	5.99	1.46	1.13	–
9	15	Y	100.01	433.28	4.16	23.07	–
Instance (using (6′)) Blocks	**Length**	**Plan exists**	***lparse***	**SMODELS**	**CMODELS mChaff**	**Simo**	**SICStus**
4	4	N	0.40	0.05	0.00	0.00	0.08
4	5	N	0.41	0.06	0.06	0.02	0.01
4	6	Y	0.42	0.06	0.06	0.02	0.01
5	11	N	1.67	0.35	0.18	0.33	2.17
5	12	N	1.68	0.59	0.26	0.64	5.92
5	13	Y	1.68	0.75	0.28	0.50	8.07
6	25	N	3.38	–	685.43	–	–
6	26	N	3.38	–	1173.55	–	–
6	27	Y	3.41	–	1181.99	–	–

Table 1. Planning in blocks world (20 minutes time limit)

6 Conclusions and Future Work

In this paper, we described two alternative approaches to solve planning problems using logic programming technology. The first approach, frequently adopted in the literature, relies on the mapping of an action language specification to an answer set program, and on the use of an answer set solver to compute the plans. The second approach relies instead on the use of constraint logic programming (over finite domains), encoding the problem as a collection of boolean variables and finite domain constraints. We described the two encodings in detail and discuss some preliminary performance results. ASP implementations run faster. However, no grounding is needed for the CLP(FD) approach.

The study presented in this paper is in the same spirit of the recent investigation in benchmarking logic programming systems (e.g., [1, 22, 6, 4]); our hope is that this line of investigation will shed some lights on the relative strengths and weaknesses of CLP and ASP, and suggest ways to integrate their use in the context of planning.

Acknowledgments This work is partially supported by PRIN2005 project 2005015491 on constraints.

References

[1] C. Anger, T. Schaub, and M. Truszczyński. ASPARAGUS – the Dagstuhl Initiative. *ALP Newsletter*, 17(3), 2004.

[2] K. Apt. *Principles of Constraint Programming.* Cambridge Unviversity Press, 2003.

[3] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.

[4] A. Dovier, A. Formisano, and E. Pontelli. A Comparison of CLP(FD) and ASP Solutions to NP-Complete Problems. In Proc. of *ICLP05*, LNCS 3668, pp. 67–82, 2005.

[5] A. Dovier, A. Formisano, and E. Pontelli. A comparison of CLP(FD) and ASP in tackling hard combinatorial problems. Web: `www.di.univaq.it/~formisano/CLPASP`.

[6] A. J. Fernandez and P. M. Hill. A Comparative Study of 8 Constraint Programming Languages Over the Boolean and Finite Domains. *Constraints*, 5(3):275–301, 2000.

[7] M. Gelfond. Representing Knowledge in A-Prolog. In *Computational Logic: Logic Programming and Beyond*, Springer Verlag, pp. 413–451, 2002.

[8] M. Gelfond and V. Lifschitz. Action Languages. *Electron. Trans. Artif. Intell.* 2:193–210, 1998.

[9] E. Giunchiglia, Y. Lierler, and M. Maratea. SAT-Based Answer Set Programming. In Proc. of *AAAI'04*, pp. 61–66, AAAI/Mit Press, 2004.

[10] A.K. Jonsson et al. Planning in Interplanetary Space: Theory and Practice. In *AIPS*, 2002.

[11] Y. Lierler and M. Maratea. CMODELS-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In Proc. of *LPNMR04*, pp. 346–350. Springer Verlag, 2004.

[12] V. Lifschitz. Answer Set Planning. In Proc. of *ICLP99*, MIT Press, pp. 23–37, 1999.

[13] A. Lopez and F. Bacchus. Generalizing GraphPlann by Formulating Planning as a CSP. In Proc. of IJCAI, 2003.

[14] V. W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. *The Logic Programming Paradigm*, pp. 375–398. Springer Verlag, 1999.

[15] K. Marriott and P. J. Stuckey. *Programming with Constraints*. MIT Press, 1998.

[16] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: Engineering an Efficient SAT Solver. In Proc. of *Design Automation Conference*, ACM Press, pp. 530–535, 2001.

[17] I. Niemela. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. In *Annals of Math and AI*, 25(3–4):241–273, 1999.

[18] R. Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems.* MIT Press, 2001.

[19] P. Simons. Extending and Implementing the Stable Model Semantics. Doctoral dissertation. Report 58, Helsinki University of Technology, 2000.

[20] T. C. Son, C. Baral, and S. McIlraith. Planning with different forms of domain-dependent control knowledge — an answer set programming approach. In Proc. of *LPNMR01*, Springer Verlag, pp. 226–239, 2001.

[21] M. Thielscher. Reasoning about Actions with CHRs and Finite Domain Constraints. In Proc. of *ICLP02*, LNCS 2401, pp. 70–84, 2002.

[22] M. Wallace, J. Schimpf, K. Shen, and W. Harvey. On Benchmarking Constraint Logic Programming Platforms. *Constraints*, 9(1):5–34, 2004.

[23] Web references for some ASP solvers. ASSAT: `assat.cs.ust.hk`. CCalc: `www.cs.utexas.edu/users/tag/cc`. Cmodels: `www.cs.utexas.edu/users/tag/cmodels`. DeReS and aspps: `www.cs.uky.edu/ai`. DLV: `www.dbai.tuwien.ac.at/proj/dlv`. SMODELS: `www.tcs.hut.fi/Software/smodels`.

[24] Web references for some CLP(FD) implementations. SICStus Prolog: `www.sics.se/isl/sicstuswww/site/index.html`. B-Prolog: `www.probp.com`. ECLiPSe: `eclipse.crosscoreoptimization.com`. GNU-Prolog: `pauillac.inria.fr/~diaz/gnu-prolog`.