# Proofs of the submitted paper
# and
# Concrete Syntax of $\mathcal{B}^{FD}_{MV}$

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 Perugia, Dip. di Matematica e Informatica. `formis@dipmat.unipg.it`
[3] New Mexico State University, Dept. Computer Science. `epontell@cs.nmsu.edu`

## 1 Proofs for the $\mathcal{B}$ language

### 1.1 Detailed Encoding

Let us start with a description of how action theories are mapped to finite domain constraints. In particular, we will provide a description of how constraints can be used to model the possible transitions from each individual state of the transition system. Let us indicate with $s_v$ and $s_u$ the starting and ending states of a transition. The approach is based on asserting constraints that relate the truth value of fluents in $s_v$ and $s_u$.

Let us introduce variables to describe the truth value of each fluent. All variables are boolean variables. The value of a fluent $f$ in $s_v$ (resp., $s_u$) is represented by the variable $\mathtt{IV}^v_f$ (resp., $\mathtt{EV}^u_f$). These variables can be used to build expressions $\mathtt{IV}^v_l$ ($\mathtt{EV}^u_l$) that represent the truth value of each fluent literal `l`. In paritcular, if `l` is a fluent `f`, then $\mathtt{IV}^v_l = \mathtt{IV}^v_f$; if `l` is the literal `neg(f)`, then $\mathtt{IV}^v_l = 1 - \mathtt{IV}^v_f$. Similar equations can be set for $\mathtt{EV}_l$. In a similar spirit, given a conjunction of literals $\alpha \equiv [\mathtt{l}_1, \ldots, \mathtt{l}_n]$ we will denote with $IV^v_\alpha$ the expression $IV^v_{l_1} \wedge \cdots \wedge IV^v_{l_n}$; similar definition is given for $EV^u_\alpha$. We will also introduce, for each action $a_i$, a boolean variable $A^v_i$ aimed at representing whether the action is executed or not in the transition from $s_v$ to $s_u$ under consideration.

Fixed a specific fluent $f$, we inted to develop constraints that determine when $EV^u_f$ is true. Let us consider the dynamic causal laws that have $f$ as a consequent:

$$\begin{array}{l}\mathtt{causes}(a_{t_1}, f, \alpha_1)\\ \cdots\\ \mathtt{causes}(a_{t_m}, f, \alpha_m)\end{array}$$

Analogously, we consider the static causal laws that assert `neg(f)`:

$$\begin{array}{l}\mathtt{causes}(a_{f_1}, neg(f), \beta_1)\\ \cdots\\ \mathtt{causes}(a_{f_n}, neg(f), \beta_n)\end{array}$$

Let us also consider the static causal laws related to `f`

$\texttt{caused}(\gamma_1, f)$
...
$\texttt{caused}(\gamma_h, f)$
$\texttt{caused}(\psi_1, neg(f))$
...
$\texttt{caused}(\psi_\ell, neg(f))$

Finally, for each action $a_i$ we will have an executability condition

$$\texttt{executable}(a_i, \delta_i)$$

Figure 1 describes the boolean constraints that can be used in encoding the relations that determine the truth value of the fluent `f`. We will denote with $C_f^{v,u}$ the conjunction of such constraints.

Given an action specification over the set of fluents $\mathcal{F}$, the system of constraints $C_{\mathcal{F}}^{v,v+1}$ include

– the constraint $C_f^{v,v+1}$ for each $f \in \mathcal{F}$ and for each $0 \leq v < N$ where $N$ is the chosen length of the plan;
– for each $f \in \mathcal{F}$ and $0 \leq v \leq N$, the constraints $IV_f^v = EV_f^v$
– for each $0 \leq v < N$, the constraint

$$\bigvee_{a_j \in \mathcal{A}} A_j^v$$

– for each $0 \leq v < N$ and for each action $a_i \in \mathcal{A}$, the constraints

$$A_i^v \leftrightarrow IV_{\delta_i}^v \wedge \bigwedge_{\substack{a_j \in \mathcal{A} \\ a_j \neq a_i}} \neg A_j^v$$

### 1.2 CLP(FD) Encoding

Let us proceed now with mapping the previous abstract encoding to concrete CLP(FD) constraints. 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 $i^{th}$ term in the $j^{th}$ list is assigned $1$ if and only if the $i^{th}$ fluent is true in the $j^{th}$ 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 $i^{th}$ term of the $j^{th}$ list is assigned $1$ if and only if the $i^{th}$ 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:

$$\texttt{EV}^u_f \leftrightarrow \texttt{Posfired}^{v,u}_f \vee (\neg\texttt{Negfired}^{v,u}_f \wedge \texttt{IV}^v_f) \tag{1}$$

$$\neg\texttt{Posfired}^{v,u}_f \vee \neg\texttt{Negfired}^{v,u}_f \tag{2}$$

$$\texttt{Posfired}^{v,u}_f \leftrightarrow \texttt{DynP}^v_f \vee \texttt{StatP}^u_f \tag{3}$$

$$\texttt{Negfired}^{v,u}_f \leftrightarrow \texttt{DynN}^v_f \vee \texttt{StatN}^u_f \tag{4}$$

$$\texttt{DynP}^v_f \leftrightarrow \bigvee_{i=1}^{m} (IV^v_{\alpha_i} \wedge A^v_{t_i}) \tag{5}$$

$$\texttt{StatP}^u_f \leftrightarrow \bigvee_{i=1}^{h} EV^u_{\gamma_i} \tag{6}$$

$$\texttt{DynN}^v_f \leftrightarrow \bigvee_{i=1}^{n} (IV^v_{\beta_i} \wedge A^v_{f_i}) \tag{7}$$

$$\texttt{StatN}^u_f \leftrightarrow \bigvee_{i=1}^{\ell} EV^u_{\psi_i} \tag{8}$$

**Fig. 1.** The constraint $C^{v,u}_f$ for the generic fluent $f$

```
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(a`$_i$`, VA`$_i$`)` contains a variable that is assigned the value 1).

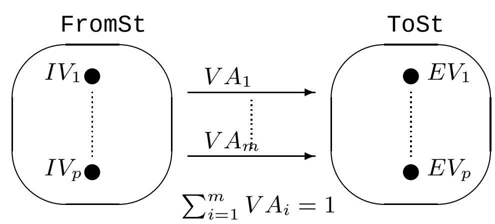


**Fig. 2.** Action constraints from state to state

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

The entry point of the planner is shown in Fig. 3. The main predicate is `main(N)` (line (1)) that computes a plan of length `N` for the action description present in the Prolog database. Lines (2) and (3) collect the lists of fluents (`Lf`) and actions (`La`). Lines (4)

```
(1)     main(N,Actionsocc,States)    setof(F,fluent(F),Lf),
(2)         setof(A,action(A),La),
(3)         make_states(N,Lf,States),
(4)         make_action_occurrences(N,La,Actionsocc),
(5)         setof(F,initially(F),Init),
(6)         setof(F,goal(F),Goal),
(7)         set_initial(Init,States),
(8)         set_goal(Goal,States),
(9)         set_transitions(Actionsocc,States),
(10)        set_executability(Actionsocc,States),
(11)        get_all_actions(Actionsocc,AllActions),
(12)        fd_labelingff(AllActions).
```

**Fig. 3.** Main predicate of the CLP(FD) planner

and (5) the predicates for defining the lists `States` and `ActionsOcc` are called. In particular, all the variables for fluents and actions are declared as Boolean variables; furthermore, a constraint is added to enforce that in every state transition, exactly one action can be fired (`fd_only_one` global constraint of GNU Prolog).

Lines (6) and (7) collect the description of the initial state (`Init`) and the required content of the final state (`Goal`). These information are then added to the Boolean variables related to the first and last state, respectively, by the predicates in lines (8) and (9).

Lines (10) and (11) impose the constraints on state transitions and action executability. We will give more details on this part below.

Line (12) gathers all variables denoting action occurrences, in preparation for the labeling phase (line (13)). 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 Fig. 3 we omit the parts concerning delivering the results to the user.

The main constraints are added by the predicate `set_transitions`. A recursion between fluents and consecutive states is made, then the predicate `set_one_fluent` is called (see Fig. 4). Its parameters are the fluent `F`, the starting state `FromSt`, the next state `ToST`, the list `Occ` of action variables, and finally the variables `IV` and `EV` related to the value of the fluent `F` (cf. also Fig. 2) in `FromSt`, and `ToST`, respectively.

For a given fluent `F`, the predicate `set_one_fluent` collects the list `DynPos` (resp. `DynNeg`) of pairs `[Act(ion),Prec(onditions)]` such that the dynamic action `Act` makes `F` true (resp. false) in the state transition (lines (15) and (16)). The variables involved are then constrained by the procedure `dynamic` (lines (17) and (18)).

Similarly, the static causal laws (`caused` assertions) are handled by collecting the lists of conditions that affect the truth value of a fluent `F` (cf., the variables `StatPos` and `StatNeg`, in lines (19)–(20)) and constraining them through the procedure `static` (lines (21) and (22)). The disjunctions of all the positive and negative conditions are collected in lines (23) and (24) and stored in `PosFired` and `NegFired`, respectively.

Finally, lines (25) and (26) take care of the relatioships between all these variables. Line (25) states that it is inconsistent that a fluent is made both true (`PosFired`) and false (`NegFired`) in the state `ToSt`. If `PosFired` and `NegFired` are both false, then $EV = IV$ (inertia). Precisely, a fluent is true in the next state (`EV`) if and only if there is an action or a static causal law making it true (`PosFired`) or it was true in the previous state (`IV`) and no causal law makes it false.

```
(14)    set_one_fluent(F,IV,EV, Occ,FromSt,ToSt)
           findall([X,L],causes(X,F,L),DynPos),
(15)       findall([Y,M],causes(Y,neg(F),M),DynNeg),
(16)       dynamic(DynPos, Occ, FromSt,DynP,EV),
(17)       dynamic(DynNeg, Occ, FromSt,DynN,EV),
(18)       findall(P,caused(P,F),StatPos),
(19)       findall(N,caused(N,neg(F)),StatNeg),
(20)       static(StatPos, ToSt, StatP,EV),
(21)       static(StatNeg, ToSt, StatN,EV),
(22)       bool_disj(DynP,StatP,PosFired),
(23)       bool_disj(DynN,StatN,NegFired),
(24)       PosFired * NegFired #= 0,
(25)       EV #<=> PosFired #\/ (#\ NegFired #/\ IV ).

(26)    dynamic([],_,_,[],_).
(27)    dynamic([[Act,Prec]|R],Occ,FromSt,[Flag|Flags],EV)
           member(action(Act,VA),Occ),
(28)       get_precondition_vars(Prec,FromSt,ListPV),
(29)       length(Prec,NPrec),
(30)       sum(ListPV, SumPrec),
(31)       (VA #/\ (SumPrec #= NPrec)) #<=> Flag,
(32)       dynamic(R,Occ,FromSt,Flags,EV).

(33)    static([],_,[],_).
(34)    static([Cond|Others],ToSt,[Flag|Flags],EV)
           get_precondition_vars(Cond,ToSt,ListPV),
(35)       length(ListPV, NPrec),
(36)       sum(ListPV, SumPV),
(37)       (SumPV #= NPrec) #<=> Flag,
(38)       static(Others,ToSt,Flags,EV).
```

**Fig. 4.** Transition from state to state

Let us consider the predicate `dynamic` (see line (27)). It recursively processes a list of pairs `[Act(ion),Prec(onditions)]`. The variable `VA` associated to the execution of action `Act` is retrieved in line (29). The variables associated to its preconditions are retrieved from state `FromSt` and collected in `ListPV` in line (30). A precondition holds if and only if all the variables in the list `ListPV` are assigned value 1. Namely, when their sum is equal to the length, `NPrec`, of the list `ListPV`. If (and only if) the action variable `VA` is true and the preconditions holds, then there is an action effect (line (33)).

Similarly, the predicate `static` recursively processes a list of preconditions `Cond`. The variables to such preconditions are retrieved from the state `ToSt` and collected in `ListPV` (line (37)). A precondition holds if and only if all the variables in the list `ListPV` have value 1. Namely, when their sum is equal to the length, `NPrec`, of `ListPV`. This happens if and only if there is a static action effect (cf., line (40)).

Executability conditions are handled as follows. For each state transition and for each action `Act`, the predicate `set_executability_sub` is called (see Fig. 5). The variable `VA`, encoding the application of an action `Act` is collected in line (43). A precondition hold if and only if the sum of the (Boolean) values of its fluent literals equals their number (lines (52)-(54)). The variable `Flags` stores the list of these conditions and the variable `F` their disjunction. If the action is executed (`VA = 1`, see line (47)), then at least one of the executability conditions must hold.

```
set_executability_sub([],_,_).
set_executability_sub([[Act,C]|CA],ActionsOcc,State)
   member(action(Act,VA),ActionsOcc),
   preconditions_flags(C, State,Flags),
   bool_disj(Flags,F),
   VA #==> F,
   set_executability_sub(CA,ActionsOcc,State).
preconditions_flags([],_,[]).

preconditions_flags([C|R],State,[Flag|Flags])
   get_precondition_vars(C,State,Cs),
   length(Cs,NCs),
   sum(Cs, SumCs),
   (NCs #= SumCs) #<=> Flag,
   preconditions_flags(R,State,Flags).
```

**Fig. 5.** Executability conditions

### 1.3 Soundness and completeness

Let us proceed with the soundness and completeness proof. For each fluent $f$, the predicate `set_one_fluent` imposes the constraint $C_f^{v,v+1}$ described earlier. Moreover, such predicate constraints a number of auxiliary (Boolean) variables to the values of specific expressions, as shown in Table 1, and all the additional constraints described.

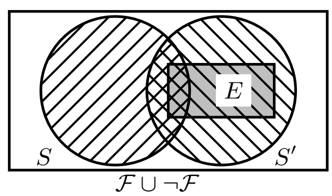


**Fig. 6.** Sets of fluents involved in a state transition

Let $S$ (resp., $S'$) be the set of fluent literals that holds in $s_v$ (resp., $s_{v+1}$). Note that, from any specific, known, $S$ (resp., $S'$), we can obtain a consistent assignment $\sigma_S$ (resp., $\sigma_{S'}$) of truth values for all the variables $\mathtt{IV}_f^v$ (resp., $\mathtt{EV}_f^{v+1}$) of $s_v$ (resp., $s_{v+1}$). Conversely, each truth assignment $\sigma_S$ (resp., $\sigma_{S'}$) for all variables $\mathtt{IV}_f^v$ (resp., $\mathtt{EV}_f^{v+1}$) corresponds to a consistent set of fluents $S$ (resp., $S'$). As regards the occurrence of actions, in each state transition a single action $a_i$ occurs and its occurrence is encoded through a specific Boolean variable, say $A_i^v$.

Let $\sigma_a$ be the assignment of truth values for such variables such that $\sigma_a(A_i^v) = 1$ if and only if $a_i$ occurs in the state transition from $s_v$ to $s_{v+1}$. Note that the domains of $\sigma_S$, $\sigma_{S'}$, and $\sigma_a$ are disjoint, so we can safely denote by $\sigma_S \circ \sigma_{S'} \circ \sigma_a$ the composition of the three assignments. With a slight abuse of notation, in what follows we will denote $E(a, s_v)$ with $E$. Clearly, $E \subseteq S'$.

Theorem 1 states the completeness of the planner of Fig. 3. It asserts that for any given $\mathcal{D} = \langle \mathcal{DL}, \mathcal{EL}, \mathcal{SL} \rangle$, if a triple $\langle s, a, s' \rangle$ belongs to the transition system described by $\mathcal{D}$, then the assignment $\sigma = \sigma_S \circ \sigma_{S'} \circ \sigma_a$ satisfies the condition $C_{\mathcal{F}}^{v,v+1}$.

**Theorem 1 (Completeness).** *Let $\mathcal{D} = \langle \mathcal{DL}, \mathcal{EL}, \mathcal{SL} \rangle$. If $S' = \texttt{Clo}(E(a_i, s_v) \cup (S \cap S'))$ then $\sigma_S \circ \sigma_{S'} \circ \sigma_a$ is a solution of the constraint $C^{v,v+1}_{\mathcal{F}}$.*

*Proof.* As mentioned, the planner introduces a number of auxiliary constrained variables whose values are univocally determined once the values of the fluents are assessed. In other words, when $S$, $S'$, and $a$ are fixed, the r.h.s. of the constraints $(5)$–$(8)$ are completely specified. To prove the theorem, we need to verify that if $S' = \texttt{Clo}(E \cup (S \cap S'))$, then the constraints $(1)$, $(2)$ along with the constraints about the action variables $A^v_i$ are satisfied for every fluent $f$.

Let us start by looking at the action occurrence. Let $a_i$ be the action executed in state $s_v$, thus $\sigma_a = \{A^v_i/1\} \cup \{A^b_j/0 | j \neq i\}$. Thus, it is easy to see that $(\bigvee_{a_j \in \mathcal{A}} A^v_j)\sigma_a$ is true. Similarly, since the semantics requires that actions are executed only if the executability conditions are satisfied, then this means that $S \models \delta_i$, which quickly leads to $(IV^v_{\delta_i})\sigma_s$ is true, and this allows us to conclude that

$$(A^v_r \leftrightarrow IV^v_{\delta_r} \wedge \bigwedge_{\substack{a_j \in \mathcal{A} \\ a_j \neq a_r}} \neg A^v_j)\sigma_s \circ \sigma_a$$

is true for each $a_r \in \mathcal{A}$.

Let us now consider the constraints dealing with fluents. First of we recall that $S'$ is a consistent, complete, and closed w.r.t. $\mathcal{SL}$, set of fluent literals. Let us consider a fluent $f$. We prove that constraint $(2)$ is satisfied. Assume, by contradiction, that $\texttt{Posfired}^{v,v+1}_f\sigma$ and $\texttt{Negfired}^{v,v+1}_f\sigma$ are both true. Four cases must be considered:

1. $\texttt{DynP}^v_f\sigma$ and $\texttt{DynN}^v_f\sigma$ are true. Since these values are determined by $s_v, a_i, s_{v+1}$, this means that both $f$ and $\texttt{neg}(f)$ belong to $E(a_i, s_v)$. Since the closure is monotonic this means that $Lit(s_{v+1} = S'$ is inconsistent, representing a contradiction.
2. $\texttt{DynP}^v_f\sigma$ and $\texttt{StatN}^{v+1}_f\sigma$ are true. This means that $f$ is in $E(a_i, s_v)$ and $\texttt{neg}(f)$ is added to $S'$ by the closure operation. This implies that $S'$ is inconsistent, which represents a contradiction.
3. $\texttt{StatP}^{v+1}_f\sigma$ and $\texttt{DynN}^v_f\sigma$ are true. This leads a contraddiction as in the previous case.
4. $\texttt{StatP}^{v+1}_f\sigma$ and $\texttt{StatN}^{v+1}_f\sigma$ are true. This means that $f$ and $\texttt{neg}(f)$ are added to $S'$ by the closure operation. This means that $S'$ is inconsistent, which is a contradiction.

It remains to prove that constraint $(1)$ is satisfied by $\sigma$. Let us assume that $f \in S'$. Thus, $EV^{v+1}_f\sigma_{S'}$ is true. Three cases must be considered.

1. $f \in E(a_i, s_v)$. This means that there is a dynamic causal law $\texttt{causes}(a_i, f, \alpha_i)$ where $S \models \alpha_i$. From the definition, this leads to $IV^v_{\alpha_i}\sigma$ being true and $\sigma_a(A^v_i) = 1$. Thus, constraint $(5)$ sets $\texttt{DynP}^v_f\sigma$ and $\texttt{Posfired}^{v,v+1}_f\sigma$ are both true. As a consequence, constraint $(1)$ is satisfied.
2. $f \notin E(a_i, s_v)$ and $f \in S$. This means that $f \in S \cap S'$. In this case $\texttt{Negfired}^{v,v+1}_f\sigma$ must be false, otherwise $S'$ would be inconsistent (by closure). Then, $\texttt{IV}^v_f\sigma_S$ should be true, $\texttt{EV}^{v+1}_f\sigma_{S'}$ is true and $\texttt{Negfired}^{v,v+1}_f\sigma$ is false, which satisfy constraint $(1)$ (regardless of the value of $\texttt{Posfired}^{v,v+1}_f\sigma$).

3. $f \notin E(a_i, s_v)$ and $f \notin S$. This means that $f$ is inserted in $S'$ by closure. Thus, there is a static causal law of the form $\texttt{caused}(\gamma_j, f)$ such that $S' \models \gamma_j$. In this case, by $(6)$, $\texttt{StatP}_f^{v+1}\sigma$ is true and, by $(3)$, so is $\texttt{Posfired}_f^{v,v+1}\sigma$. Thus, constraint $(1)$ is satisfied.

If $f \notin S'$, then $\texttt{neg}(f) \in S'$ and the proof is similar with positive and negative roles interchanged. □

Let us observe that the converse of the above theorem does not necessarily hold. The problem arises from the fact that the implicit minimality in the closure operation is not reflected in the computation of solutions to the constraint. Consider the action description where $\mathcal{F} = \{\mathtt{f}, \mathtt{g}, \mathtt{h}\}$ and $\mathcal{A} = \{\mathtt{a}\}$, with predicates:

```
(1)     executable(a,[]).
(2)     causes(a,f,[]).
(3)     caused([g],h).
(4)     caused([h],g).
```

Let us consider $S = \{\mathtt{neg(f)}, \mathtt{neg(g)}, \mathtt{neg(h)}\}$ and $S' = \{\mathtt{f}, \mathtt{g}, \mathtt{h}\}$ determines a solution of the constraint $C_{\mathcal{F}}^{v,v+1}$ with the execution of action $a$, but $\texttt{Clo}(E \cup (S \cap S')) = \{\mathtt{f}\} \subset S'$. However, the following holds:

**Theorem 2 (Weak Soundness).** *Let $\mathcal{D} = \langle \mathcal{DL}, \mathcal{EL}, \mathcal{SL} \rangle$. Let $\sigma_S \circ \sigma_{S'} \circ \sigma_a$ identify a solution of the constraint $C_{\mathcal{F}}^{v,v+1}$. Then $\texttt{Clo}(E(a_i, s_v) \cup (S \cap S')) \subseteq S'$.*

*Proof.* It is immediate to see that $\sigma_S$ and $\sigma_{S'}$ uniquely determines two consistent and complete sets of fluent literals. Moreover, they are closed under $\mathcal{SL}$ (thanks to constraints $(6)$ and $(8)$ in Figure 1). Let $f$ be a positive fluent in $\texttt{Clo}(E(a_i, s_v) \cup (S \cap S'))$. We show now that $f \in S'$.

1. If $f$ is in $S \cap S'$ we are done.
2. If $f \in E(a_i, s_v)$, there is a law $\texttt{causes}(a_i, f, \alpha_i)$ such that $S \models \alpha_i$. Since $S$ is determined by $\sigma_S$, by $(5)$, we have that $\sigma_S \circ \sigma_a$ is a solution of $IV_{\alpha_i}^v \wedge A_i^v$, which implies that $\texttt{DynP}_f^v$ is true, and $\sigma_{S'}(EV_f)^{v+1}$ is true in $\sigma_{S'}$. Therefore, $f \in S'$. Observe also that $\sigma_a$ making true $A_i^v$ will imply that $IV_{\delta_i}^v$, which will imply the executability of $a_i$.
3. We are left with the case of $f \notin E(a_i, s_v)$ and $f \notin S \cap S'$. Since $S'$ is determined by $\sigma_{S'}$, and $f \in \texttt{Clo}(E(a_i, s_v) \cup (S \cap S'))$, there is a law $\texttt{caused}(\gamma_j, f)$ such that $S' \models \gamma_j$, and by construction $\sigma_{S'}$ makes $EV_{\gamma_j}^{v+1}$ true. Thus, $\texttt{StatP}_f^{v+1}$ is true and therefore $EV_f^{v+1}$ is true. Hence, $f \in S'$.

If $\texttt{neg}(f)$ is a negative fluent in $\texttt{Clo}(E(a_i, s_v) \cup (S \cap S'))$, the proof proceeds similarly. □

Let us consider the set of static causal laws $\mathcal{SL}$. $\mathcal{SL}$ identifies a *definite propositional program* $P$ as follows. For each positive fluent literal $\mathtt{p}$, let $\varphi(\ \mathtt{p}\ )$ be the (fresh) predicate symbol $p$, and for each negative fluent literal $\mathtt{neg(p)}$ let $\varphi(\mathtt{neg(p)})$ be the (fresh) predicate symbol $\tilde{p}$. The program $P$ is the set of clauses of the form $\varphi(\mathtt{p}) \leftarrow \varphi(\mathtt{l1}), \ldots, \varphi(\mathtt{lm})$, for each static causal law $\mathtt{caused([l1,\ldots,lm],p)}$. Notice that $p$ and $\tilde{p}$ are independent predicate symbols in $P$. From $P$ one can extract the dependency graph $\mathcal{G}(P)$ in the usual way, and the following result can be stated.

**Theorem 3 (Correctness).** *Let $\mathcal{D} = \langle \mathcal{DL}, \mathcal{EL}, \mathcal{SL} \rangle$. Let $\sigma_S, \sigma_{S'}, \sigma_a$ be a solution of the constraint $C^{v,v+1}_{\mathcal{F}}$. If the dependency graph of $P$ is acyclic, then* $\texttt{Clo}(E(a_i, s_v) \cup (S \cap S')) = S'$.

*Proof.* Theorem 2 proves that $\texttt{Clo}(E(a_i, s_v) \cup (S \cap S')) \subseteq S'$. It remains to prove that for any (positive or negative) fluent $\ell$, if $\ell \in S'$, then $\ell \in \texttt{Clo}(E(a_i, s_v) \cup (S \cap S'))$.

If $\ell \in E(a_i, s_v)$ or $\ell \in S$, then trivially $\ell \in \texttt{Clo}(E(a_i, s_v) \cup (S \cap S'))$. On the other hand, let us prove that whenever $\ell \in S'$ and $\ell \notin E(a_i, s_v) \cap (S \cap S')$ then $\ell \in \texttt{Clo}(E(a_i, s_v) \cap (S \cap S'))$. To this aim, consider the program $P$ and its dependency graph $\mathcal{G}(P)$. With a slight abuse of notation, let us identify a fluent $f$ with both the corresponding atom $\varphi(f)$ and the associated node in $\mathcal{G}(P)$. Because of the acyclicity of $\mathcal{G}(P)$, there are graph nodes without incoming edges—we will refer to them as *leaves*. Let us now prove our claim, by induction on the length of the shortest path from a leaf to the positive fluent literal $f$.

*Base case.* If $f \notin E(a_i, s_v) \cup (S \cap S')$ is a positive fluent which is a leaf (the proof is similar for the case of negative literals), then two cases are possible.

- There is no law of the form $\texttt{caused}(\gamma_j, f)$ in $\mathcal{SL}$. Thus, it cannot be that $f \in S'$. The claim holds.
- There is a law $\texttt{caused}([\,], f)$. In this case $f \in S'$ by closure.

*Inductive step.* Let $f \notin E(a_i, s_v) \cup (S \cap S')$ be a positive fluent such that there are laws $\texttt{caused}(\gamma_1, f), \ldots, \texttt{caused}(\gamma_h, f)$ in $\mathcal{SL}$. Since $f \notin E(a_i, s_v)$ and $f \notin S \cap S'$, we have that $IV^v_f$ is false, $EV^{v+1}_f$ is true, and $\texttt{DynP}^v_f$ is false under $\sigma_S \circ \sigma_{S'} \circ \sigma_a$. From the fact that constraint $(1)$ is satisfied, it follows that $\texttt{StatP}^{v+1}_f$ is true. Moreover, $\texttt{DynP}^v_f$ is true because $f \notin E(a_i, s_v)$. On the other hand, because of $(2)$, we have that $\texttt{DynN}^v_f$, $\texttt{StatN}^{v+1}_f$, and $\texttt{NegFired}^{v,v+1}_f$ are all false. Consequently, constraint $(1)$ can be simplified to $EV^v_f \leftrightarrow \bigvee_{i=1}^h EV^{v+1}_{\gamma_i}$. If $f \in S'$ (i.e., $EV^{v+1}_f$ is true), than one of $EV^{v+1}_{\gamma_j}$ is verified by $\sigma_{S'}$. This implies that, for each fluent $g$ required to be true (resp., false) in $\gamma_j$, $g$ is set true (resp., false) by $\sigma_{S'}$. By inductive hypothesis, such fluent literals (either $g$ or $neg(g)$) belong to $\texttt{Clo}(E(a_i, s_v) \cup (S \cap S'))$. Since $\texttt{Clo}(E(a_i, s_v) \cup (S \cap S'))$ is closed under the static laws, it follows that $f \in S'$.

The proof in case of a negative fluent $\texttt{neg}(f)$ is similar. □

Let the program $P$ meet the conditions of the previous theorem; we can prove the following.

**Theorem 4.** *There is a trajectory $\langle s_0, a_1, s_1, a_2, \ldots, a_n, s_n \rangle$ in the transition system if and only if there is a solution for the constraints*

$$C^{0,1}_{\mathcal{F}} \wedge C^{1,2}_{\mathcal{F}} \wedge \cdots \wedge C^{n-1,n}_{\mathcal{F}}$$

*Proof.* The result is a simple inductive (on $n$) application of the previous theorem.

## 2 Concrete Syntax of $\mathcal{B}^{FD}_{MV}$

An action signature consists of a set $\mathcal{F}$ of fluent names, a set $\mathcal{A}$ of action names, and a set $\mathcal{V}$ of values for fluents in $\mathcal{F}$.

As a concrete syntax, fluents and actions are ground atomic formulae $p(t_1,\ldots,t_n)$ from an underlying logic language $\mathcal{L}$. We assume that the set of admissible terms is finite (e.g., either there are no function symbols in $\mathcal{L}$, or the use of functions symbols is restricted to avoid the creation of arbitrary complex terms).

In the definition of an action description, an assertion of the kind

$$\texttt{fluent}(f,v_1,v_2) \;\; \textit{or} \;\; \texttt{fluent}(f,\{v_1,\ldots,v_k\})$$

declares that $f$ is a fluent and that its set of values $\mathcal{V}$ is the interval $[v_1,v_2]$ or the set $\{v_1,\ldots,v_k\}$.

An *annotated fluent (AF)* is of the form $f^{-i}$ where $f$ is a fluent and $i \in \mathbb{N}$. $f^0$ is said a *current fluent* and should be represented simply by $f$.[1]

Annotated fluents can be used inside *fluent expressions (*`FE`*)* that can be defined inductively as follows:

$$\texttt{FE} ::= n|\langle\texttt{AF}\rangle|\texttt{abs}(\texttt{FE})|\texttt{FE}_1 \oplus \texttt{FE}_2|\texttt{rei(FC)}$$

where $n \in \mathbb{Z}$, $\oplus \in \{+,-,*,/,\texttt{mod}\}$. `rei(FC)` is the reified constraint, where `FC` is a fluent constraint defined below.

Fluent expressions can be used to build *fluent constraints (*`FC`*))*. A primitive fluent constraint is a formula $\texttt{FE}_1 \texttt{op}\, \texttt{FE}_2$ where $\texttt{FE}_1$ and $\texttt{FE}_2$ are fluent expressions —without reification— and $\texttt{op} \in \{\texttt{eq},\texttt{neq},\texttt{geq},\texttt{leq},\texttt{lt},\texttt{gt}\}$. A *fluent constraint* is a conjunction of primitive fluent constraints. Concretely, $C_1 \wedge \cdots \wedge C_n$ is represented by $[C_1,\ldots,C_n]$. The empty list stands for `true`.

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

- Declarations of the form `action(a)` are used to describe the possible actions (in this case, `a`).
- `executable(a,`$C$`)`
  where $C$ is a fluent constraint. asserting that the constraint $C$ has to be satisfied for the action `a` to be executable.
- `causes(a,`$FC$`,`$C$`)`
  where $C$ is a fluent constraint, and $FC$ is a primitive fluent constraint containing at least one current fluent, encodes a dynamic causal law. If action `a` happens and the fluent constraint $C$ is satisfied then the value of the primitive constraint $FC$ must be satisfied.
- `caused`$(C,FC)$
  where $C$ is a fluent constraint, and $FC$ is a primitive fluent constraint containing at least one current fluent, describes a static causal law. If the fluent constraint $C$ holds then the primitive fluent constraint $FC$ must hold. [2]

An *action description* is a set of executability conditions, static and dynamic laws.

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

[1] We suggest to use this short notation.
[2] We suggest to use static causal laws only with current fluents

- `initially(FE1 op FE2)`,
  asserts that the fluent constraint $FE1$`op` $FE2$ holds in the initial state.
- `goal(FE1 op FE2)`
  asserts that the fluent constraint $FE1$`op` $FE2$ holds in the final state.

It is possible to add information about the *cost* of each action and about the global cost of a plan. This can be done by writing rules of the form:

- `action_cost(action,VAL)` (if no information is given, the default cost is 1).
- `plan_cost(plan OP NUM)` where NUM is a number, adds the information about the global cost admitted

A similar requirement can be done on fluents and states.

- `state_cost(FE)` (if no information is given, the default cost is 1) is the cost of a state, where $FE$ is a fluent expression built on current fluents.
- `goal_cost(goal op NUM)` adds a contraint about the global cost admitted

Further constraints can be added among fluents. We define a timed fluent a pair `FLUENT @ TIME`. Timed fluent can be used to build timed fluent expressions (TE) and timed primitive constraints (TC). For instance `contains(5) @ 2 leq contains(5) @ 4` states that at time 2 the barrel number 5 contains at most the same amount of water as at time 4. `contains(12) @ 2 eq 3` states that at time 3 the barrel 12 contains exactly 3 liters of water.

- `cross_constraint(TC)` allows to impose a timed primitive constraint. It allows to impose constraints between fluent expressions of different states, as well as to force values of fluents in some predetermined times. the execution.
- `holds(FC,StateNumber)` It is a simplification of the above constraint. states that the primitive fluent constraint $FC$ holds at the desired State Number (0 is the number of the initial state).[3] It is therefore a generalization of the `initially` primitive. It allows to drive the plan search with some point information.
- `always(FC)` states that the fluent constraints $FC$ holds in all the states. It applies `holds(FC,i)` for all states $i$. Current fluents must be used in order to avoid negative references.

[3] Annotated fluents can be used here, if needed.