TLP **10** (2): 167–235, 2010. © Cambridge University Press 2010
doi:10.1017/S1471068410000013

Multivalued action languages with constraints in CLP(FD)[1]

AGOSTINO DOVIER

Università di Udine, Dipartimento di Matematica e Informatica, Via delle Scienze 206, 33100 UDINE (Italy)
(*e-mail:* `dovier@dimi.uniud.it`)

ANDREA FORMISANO

Università di Perugia, Dipartimento di Matematica e Informatica, Via Vanvitelli 1, 06123 Perugia (Italy)
(*e-mail:* `formis@dmi.unipg.it`)

ENRICO PONTELLI

New Mexico State University, Department of Computer Science, P.O. Box 30001, MSC CS, Las Cruces, NM 88003 (USA)
(*e-mail:* `epontell@cs.nmsu.edu`)

submitted 5 January 2009; revised 9 October 2009; accepted 10 December 2009

Abstract

Action description languages, such as $\mathscr{A}$ and $\mathscr{B}$ (Gelfond and Lifschitz, *Electronic Transactions on Artificial Intelligence*, 1998, vol. 2, pp. 193–210), are expressive instruments introduced for formalizing planning domains and planning problem instances. The paper starts by proposing a methodology to encode an action language (with conditional effects and static causal laws), a slight variation of $\mathscr{B}$, using *Constraint Logic Programming over Finite Domains*. The approach is then generalized to raise the use of constraints to the level of the action language itself. A prototype implementation has been developed, and the preliminary results are presented and discussed.

KEYWORDS: action description languages, knowledge representation, planning, constraint logic programming

1 Introduction

The construction of intelligent agents that can be effective in real-world environments has been a goal of researchers from the very first days of artificial intelligence. It has long been recognized that an intelligent agent must be able to *acquire*, *represent*, and *reason* with knowledge. As such, a *reasoning component* has been an inseparable part of most agent architectures in the literature.

[1] This manuscript is an extended version of the paper *"Multi-valued Action Languages with Constraints in CLP(FD)"* in the Proceedings of the International Conference on Logic Programming, pages 255–270, Springer Verlag, 2007.

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.

Logic programming languages offer many properties that make them very suitable as knowledge representation languages. Their declarative nature supports the modular development of provably correct reasoning modules (Baral 2003). Recursive definitions can be easily expressed and reasoned upon. Control knowledge and heuristic information can be declaratively and incrementally introduced in the reasoning process. Furthermore, many logic programming languages offer a natural support for nonmonotonic reasoning, which is considered essential for common-sense reasoning (Lifschitz 1999). These features, along with the presence of efficient inference engines (Marriott and Stuckey 1998; Simons 2000; Apt 2003; Giunchiglia *et al.* 2004b; Gebser *et al.* 2007), make logic programming an attractive paradigm for knowledge representation and reasoning.

In the context of knowledge representation and reasoning, a very important application of logic programming has been in the domain of *reasoning about actions and change* and, more specifically, *planning*. Planning problems have been effectively encoded using Answer Set Programming (ASP) (Baral 2003) – where distinct answer sets represent different trajectories leading to the desired goal. Other logic programming paradigms, e.g., *Constraint Logic Programming over Finite Domains (CLP(FD))* (Jaffar and Maher 1994; Apt 2003), have been used less frequently to handle problems in reasoning about actions (e.g., Reiter 2001; Thielscher 2002a). Comparably more emphasis has been placed in encoding planning problems as (nonlogic programming) constraint satisfaction problems (Lopez and Bacchus 2003).

Recent proposals 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*; some well-known examples include the languages $\mathscr{A}$ and $\mathscr{B}$ (Gelfond and Lifschitz 1998) and extensions like $\mathscr{K}$ (Eiter *et al.* 2004) and $\mathscr{ADC}$ (Baral *et al.* 2002). Action languages allow one to write propositions that describe the effects of actions on states, and to create queries to infer properties of the underlying transition system. An *action domain description* is a specification of a planning domain using an action language.

The goal of this work is to explore the relevance of constraint solving and constraint logic programming (Marriott and Stuckey 1998; Apt 2003) in dealing with action languages and planning. The push toward this exploratory study came from recent investigations (Dovier *et al.* 2005, 2009a) aimed at comparing the practicality and efficiency of ASP versus constraint logic programming in solving various combinatorial and optimization problems. The study indicated that CLP offers a valid alternative, especially in terms of efficiency, to ASP when dealing with planning problems. Furthermore, CLP offers the flexibility of programmer-developed search strategies and the ability to handle numerical constraints.

The first step, in this paper, is to illustrate a scheme that directly processes an action description specification, in a language similar to $\mathscr{B}$ (Gelfond and Lifschitz 1998), producing a CLP(FD) program that can be used to compute solutions to the planning problem. Our encoding has some similarities to the one presented

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

by Lopez and Bacchus (Lopez and Bacchus 2003), although we rely on constraint logic programming instead of plain constraint satisfaction (CSP), and our action language supports static causal laws and nondeterminism – while the work of Lopez and Bacchus is restricted to STRIPS-like specifications.

While the first step relies on using constraints to compute solutions to a planning problem, the second step brings the expressive power of constraints to the level of the action language, by allowing multivalued fluents and constraint-producing actions to be used in the domain specification. The extended action language (named $\mathscr{B}^{MV}$) can be as easily supported by the CLP(FD) framework, and it allows a declarative encoding of problems involving actions with resources, delayed effects, and maintenance goals. These ideas have been developed in a prototype, and some preliminary experiments are reported.

We believe that the use of CLP(FD) can greatly facilitate the transition of declarative extensions of action languages to concrete and effective implementations, overcoming some inherent limitations (e.g., efficiency and limited handling of numbers) of other logic-based systems (e.g., ASP).

The presentation is organized as follows. The first part of our paper (Sections 2 and 3) provides an overview of the action language $\mathscr{B}$ and illustrates our approach to modeling problem specifications in $\mathscr{B}$ using constraints and constraint logic programming. Section 4 provides motivations for the proposed multivalued extensions. Section 5 introduces the full syntax of the new language $\mathscr{B}^{MV}$. The action language $\mathscr{B}^{MV}$ expands the previous language to a language with constraints and multivalued fluents, which enables the use of dynamic and static causal laws (a.k.a. state constraints), executability conditions, and non-Markovian forms of reasoning with arbitrary relative or absolute references to past and future points in time. The semantics and the abstract implementation of $\mathscr{B}^{MV}$ is incrementally developed in Section 6, where we first consider a sublanguage not involving non-Markovian references, and later we extend it to the full $\mathscr{B}^{MV}$. A concrete implementation in CLP(FD) is described in Section 7, and an experimental evaluation is discussed in Section 8. Section 9 presents an overview of related efforts appeared in the literature, while Section 10 presents conclusions and the directions for future investigation.

2 The action language $\mathscr{B}$

"Action languages are formal models of parts of the natural language that are used for talking about the effects of actions" (Gelfond and Lifschitz 1998). Action languages are used to define *action descriptions* that embed knowledge to formalize planning problems. In this section, we use the same variant of the language $\mathscr{B}$ used in Son *et al.* (2001) – see also Section 9 for a comparison. With a slight abuse of notation, we simply refer to this language as $\mathscr{B}$.

2.1 Syntax of $\mathscr{B}$

An action signature consists of a set $\mathscr{F}$ of *fluent* names, a set $\mathscr{A}$ of *action* names, and a set $\mathscr{V}$ of values for fluents in $\mathscr{F}$. In this section, we consider Boolean fluents,

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

hence $\mathscr{V} = \{0,1\}^2$. A *fluent literal* is either a fluent f or its negation neg(f). Fluents and actions are concretely represented by *ground* atomic formulae $p(t_1, \ldots, t_m)$ from an underlying logic language $\mathscr{L}$. For simplicity, we assume that the set of terms is finite – e.g., either there are no function symbols in $\mathscr{L}$ or the use of functions symbols is restricted, for instance, by imposing a fixed maximal depth on the nesting of terms, to avoid the creation of arbitrary complex terms.

The language $\mathscr{B}$ allows us to specify an *(action) domain description* $\mathscr{D}$. The core components of a domain description are its *fluents* – properties used to describe the state of the world that may dynamically change in response to execution of actions – and *actions* – denoting how an agent can affect the state of the world. Fluents and actions are introduced by assertions of the forms `fluent(`f`)` and `action(`a`)`. An action description $\mathscr{D}$ relates actions, states, and fluents using axioms of the following types – where `[list-of-conditions]` denotes a list of fluent literals[3]:

- `causes(`a`, `ℓ`, [list-of-conditions])`: this axiom encodes a dynamic causal law, describing the effect (i.e., truth assignment to the fluent literal ℓ) of the execution of action a in a state satisfying the given conditions
- `caused([list-of-conditions], `ℓ`)`: this axiom describes a static causal law – i.e., the fact that the fluent literal ℓ is true in any state satisfying the given preconditions.

Moreover, preconditions can be imposed on the executability of actions by means of assertion of the forms:

- `executable(`a`, [list-of-conditions])`: this axiom asserts that, for the action a to be executable, the given conditions have to be satisfied in the current state.

A *domain description* is a set of static causal laws, dynamic laws, and executability conditions. A specific *planning problem* $\langle \mathscr{D}, \mathscr{O} \rangle$ contains a domain description $\mathscr{D}$ along with a set $\mathscr{O}$ of *observations* describing the *initial state* and the *desired goal*:

- `initially(`ℓ`)` asserts that the fluent literal ℓ is true in the initial state
- `goal(`ℓ`)` asserts that the goal requires the fluent literal ℓ to be true in the final state.

In the specification of an action theory, we can take advantage of a Prolog-like syntax to express in a more succinct manner the laws of the theory. For instance, to assert that in the initial state all fluents are true, we can simply write the following rule:

```
initially(F) :- fluent(F)
```

instead of writing a fact `initially(`f`)` for each possible fluent f. Remember that the notation $H \; :- B_1, \ldots, B_k$ is a syntactic sugar for the logical formula

$$\forall X_1 \cdots X_n (B_1 \wedge \cdots \wedge B_k \rightarrow H),$$

where $X_1, \ldots, X_n$ are all the variables present in $H, B_1, \ldots, B_k$.

[2] For simplicity, we use 0 to denote *false* and 1 to denote *true*. Consequently, we often say that a fluent is true (resp., false) if its value is 1 (resp., 0).

[3] We will sometimes write `true` as a synonymous for the empty list of conditions.

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

```
%% Some Type Information
barrel(5).
barrel(7).
barrel(12).
liter(0).
liter(1).
  :
  :
liter(12).

%% Identification of the fluents
fluent(cont(B,L)):- barrel(B), liter(L), L ≤ B.

%% Identification of the actions
action(fill(X,Y)):- barrel(X), barrel(Y), X ≠ Y.

%% Dynamic causal laws
causes(fill(X,Y), cont(X,0), [cont(X,LX), cont(Y,LY)]) :-
  action(fill(X,Y)), fluent(cont(X,LX)),
  fluent(cont(Y,LY)), Y-LY ≥ LX.
causes(fill(X,Y), cont(Y,LYnew), [cont(X,LX), cont(Y,LY)]) :-
  action(fill(X,Y)), fluent(cont(X,LX)),
  fluent(cont(Y,LY)), Y-LY ≥ LX, LYnew is LX+LY.
causes(fill(X,Y), cont(X,LXnew), [cont(X,LX), cont(Y,LY)]) :-
  action(fill(X,Y)), fluent(cont(X,LX)),
  fluent(cont(Y,LY)), LX ≥ Y-LY, LXnew is LX-Y+LY.
causes(fill(X,Y), cont(Y,Y), [cont(X,LX), cont(Y,LY)]) :-
  action(fill(X,Y)), fluent(cont(X,LX)),
  fluent(cont(Y,LY)), LX ≥ Y-LY.

%% Executability conditions
executable(fill(X,Y), [cont(X,LX), cont(Y,LY)]) :-
  action(fill(X,Y)), fluent(cont(X,LX)),
  fluent(cont(Y,LY)), LX > 0, LY < Y.

%% Static causal laws caused([cont(X,LX)], neg(cont(X,LY))) :-
  fluent(cont(X,LX)), fluent(cont(X,LY)),
  barrel(X), liter(LX), liter(LY), LX≠LY.

%% Description of the initial and goal state
initially(cont(12,12)).
initially(cont(7,0)).
initially(cont(5,0)).
goal(cont(12,6)).
goal(cont(7,6)).
goal(cont(5,0)).
```

Fig. 1. $\mathcal{B}$ description of the 12-7-5 barrels problem.

Example 1

Figure 1 presents an encoding of the three-barrel planning problem using the language $\mathcal{B}$. There are three barrels of capacity N (an even number), $N/2+1$, and $N/2-1$, respectively. At the beginning, the largest barrel is full of wine while the other two are empty. We wish to reach a state in which the two larger barrels contain the same amount of wine. The only permissible action is to pour wine from one barrel to another, until the latter is full or the former is empty. Figure 1 shows the encoding of the problem for $N = 12$. Notice that we also require that the smallest barrel is empty at the end. □

2.2 *Semantics of* $\mathcal{B}$

If $f \in \mathcal{F}$ is a fluent, and S is a set of fluent literals, we say that $S \models f$ if and only if $f \in S$ and $S \models \texttt{neg}(f)$ if and only if $\texttt{neg}(f) \in S$. A list of literals $L = [\ell_1, \ldots, \ell_m]$ denotes a conjunction of literals, hence $S \models L$ if and only if $S \models \ell_i$ for all

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

$i \in \{1,\ldots,m\}$. We denote with $\neg S$ the set $\{f \in \mathscr{F} : \texttt{neg}(f) \in S\} \cup \{\texttt{neg}(f) : f \in S \cap \mathscr{F}\}$. A set of fluent literals is *consistent* if there is no fluent f s.t. $S \models f$ and $S \models \texttt{neg}(f)$. If $S \cup \neg S \supseteq \mathscr{F}$ then S is *complete*. A set S of literals is *closed* w.r.t. a set of static laws $\mathscr{SL} = \{\texttt{caused}(C_1, \ell_1), \ldots, \texttt{caused}(C_m, \ell_m)\}$, if for all $i \in \{1,\ldots,m\}$ it holds that $S \models C_i$ implies $S \models \ell_i$. The set $\texttt{Clo}_{\mathscr{SL}}(S)$ is defined as the smallest set of literals containing S and closed w.r.t. $\mathscr{SL}$. $\texttt{Clo}_{\mathscr{SL}}(S)$ is uniquely determined and not necessarily consistent.

The semantics of an action language on the action signature $\langle \mathscr{V}, \mathscr{F}, \mathscr{A} \rangle$ is given in terms of a transition system $\langle \mathscr{S}, v, R \rangle$ (Gelfond and Lifschitz 1998), consisting of a set $\mathscr{S}$ of states, a total interpretation function $v : \mathscr{F} \times \mathscr{S} \rightarrow \mathscr{V}$ (in this section $\mathscr{V} = \{0,1\}$), and a transition relation $R \subseteq \mathscr{S} \times \mathscr{A} \times \mathscr{S}$.

Given a transition system $\langle \mathscr{S}, v, R \rangle$ and a state $s \in \mathscr{S}$, let

$$Lit(s) = \{f \in \mathscr{F} \; : \; v(f,s) = 1\} \cup \{\texttt{neg}(f) \; : \; f \in \mathscr{F}, \, v(f,s) = 0\}.$$

Observe that $Lit(s)$ is consistent and complete.

Given a set of dynamic laws $\{\texttt{causes}(a, \ell_1, C_1), \ldots, \texttt{causes}(a, \ell_m, C_m)\}$ for the action $a \in \mathscr{A}$ and a state $s \in \mathscr{S}$, we define the *(direct) effects of a in s* as follows:

$$E(a,s) = \{\ell_i : 1 \leqslant i \leqslant m, Lit(s) \models C_i\}.$$

The action a is said to be *executable* in a state s if it holds that

$$Lit(s) \models \bigvee_{i=1}^{h} C_i, \tag{1}$$

where $\texttt{executable}(a, C_1), \ldots, \texttt{executable}(a, C_h)$ for $h > 0$, are the executability axioms for the action a in $\mathscr{D}$. Observe that multiple executability axioms for the same action a are considered disjunctively. Hence, for each action a, at least one executable axiom must be present in the action description[4].

Let $\mathscr{D}$ be an action description defined on the action signature $\langle \mathscr{V}, \mathscr{F}, \mathscr{A} \rangle$, composed of dynamic laws $\mathscr{DL}$, executability conditions $\mathscr{EL}$, and static causal laws $\mathscr{SL}$.

The transition system $\langle \mathscr{S}, v, R \rangle$ *described by* $\mathscr{D}$ is a transition system such that

- $\mathscr{S}$ is the set of all states s such that $Lit(s)$ is closed w.r.t. $\mathscr{SL}$;
- R is the set of all triples $\langle s, a, s' \rangle$ such that a is executable in s and

$$Lit(s') = \texttt{Clo}_{\mathscr{SL}}(E(a,s) \cup (Lit(s) \cap Lit(s'))). \tag{2}$$

Let $\langle \mathscr{D}, \mathscr{O} \rangle$ be a planning problem instance, where $\{\ell \mid \texttt{initially}(\ell) \in \mathscr{O}\}$ is a consistent and complete set of fluent literals. A *trajectory* in $\langle \mathscr{S}, v, R \rangle$ is a sequence

$$\langle s_0, a_1, s_1, a_2, \cdots, a_{\mathbf{N}}, s_{\mathbf{N}} \rangle$$

such that $\langle s_i, a_{i+1}, s_{i+1} \rangle \in R$ for all $i \in \{0, \ldots, \mathbf{N}-1\}$.

[4] Observe that even if an action is "executable", its execution may lead to an inconsistent state (which effectively prevents the use of such action in that context). Even though "enabled" would be a better term to use for an action that can be executed in a state, we prefer to maintain the same terminology as used for $\mathscr{B}$ in Son *et al.* (2001) – see also Remark 2.

A sequence of actions $\langle a_1, \ldots, a_{\mathbf{N}} \rangle$ is a solution (a *plan*) to the planning problem $\langle \mathscr{D}, \mathscr{O} \rangle$ if there is a trajectory $\langle s_0, a_1, s_1, \ldots, a_{\mathbf{N}}, s_{\mathbf{N}} \rangle$ in $\langle \mathscr{S}, v, R \rangle$ such that

- $Lit(s_0) \models r$ for each `initially(`r`)` $\in \mathscr{O}$, and
- $Lit(s_{\mathbf{N}}) \models \ell$ for each `goal(`ℓ`)` $\in \mathscr{O}$.

The plans characterized in this definition are *sequential* – i.e., we disallow concurrent actions. Observe also that the desired plan length **N** is assumed to be given.

Remark 1

In this paper we focus on sequential plans only. Hence, we assume that only one action is executed in each state transition composing a given trajectory.

Note that the constraint-based encoding we will propose in the rest of this manuscript can be easily adapted to deal with concurrent actions. Nevertheless, we have opted to ignore this aspect in this manuscript, to avoid further complications of notation, and dealing with issues of concurrency goes beyond the scope of this paper. The interested reader is referred to Dovier *et al.* (2009b) for some further considerations on this matter.

Remark 2

Notice that the satisfaction of (1) is just a necessary requirement for the executability of an action and it might not represent a sufficient precondition. Indeed, as far as the definition of transition system is considered, it is easy to see that, even if (1) is satisfied for certain a and s, the execution of a in s might be inhibited because of the contradictory effects of the causal laws. A simple example is represented by the following action description $\mathscr{D}$:

```
executable(a,[]).
causes(a,f,[]).
causes(a,neg(f),[]).
```

The action a is always executable (according to its executability law), but the execution of a would yield an inconsistent situation. Indeed, the execution of a does not correspond to any state transition in the transition system described by $\mathscr{D}$.

The above example also suggests a possible extension of the action description language that involves laws of the form

```
nonexecutable(a, D).
```

The semantics for such an extended action language can be defined by replacing the condition (1), with the following one:

$$Lit(s) \models \bigvee_{i=1}^{h} C_i \wedge \neg \bigvee_{j=1}^{k} D_j,$$

where `executable(`a`, `C_1`)`$,\ldots,$ `executable(`a`, `C_h`)` and `nonexecutable(`a`, `D_1`)`$,\ldots,$ `nonexecutable(`a`, `D_k`)`, for $h > 0$ and $k \geqslant 0$, are defined for the action a. Thus, the action a is executable only if at least one of the C_is is satisfied and all D_js are unsatisfied in the state s.

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

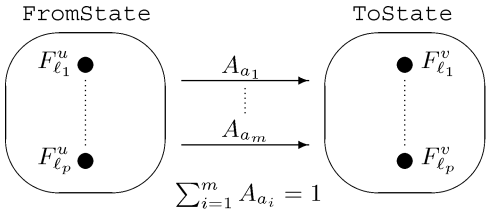


Fig. 2. Action constraints from state to state. (The states are described by p fluents, $\ell_1, \ldots, \ell_p$, and one among m possible actions is executed.)

An alternative interpretation of the `nonexecutable` axioms can be adopted. Namely, the law `nonexecutable(`a`, `D`)` can be considered simply as shorthand for the pair of dynamic causal laws `causes(`a`, `f`, `D`)` and `causes(`a`, neg(`f`), `D`)`. (Actually, this possibility also applies to the languages proposed in Gelfond and Lifschitz 1998).

This shows that (non)executability laws do not increase the expressive power of the action language. Nevertheless, the availability of both types of laws permits the direct and explicit formalization of preconditions for actions execution.

3 Modeling $\mathscr{B}$ and planning problems using constraints

Let us describe how action descriptions are mapped to finite domain constraints. We will focus on how constraints can be used to model the possible transitions from each individual state of the transition system.

3.1 Modeling an action theory as constraints

Let us consider a domain description $\mathscr{D}$ and the state transition system described by $\mathscr{D}$. Let us also denote with u and v the starting and ending states of an arbitrary transition of such a system. We assert constraints that relate the truth value of fluents in u and v. This is intuitively illustrated in Figure 2, where $u =$ `FromState` and $v =$ `ToState`[5].

A Boolean variable is introduced to describe the truth value of each fluent literal in a state. The value of a fluent literal ℓ in u is represented by the variable F^u_ℓ; analogously, its value in the destination state v is represented by the variable F^v_ℓ. For the sake of simplicity, we will freely refer to these variables as Boolean entities – and compose them with logical connectives to form Boolean expressions – as well as 0/1 variables – and compose them with arithmetic operators. Concrete CLP(FD) systems, e.g., SICStus, ECLiPSe, and BProlog[6], enable this type of alternative perspectives, providing basic primitive constraints (e.g., `#=` and `#>`) and Boolean compositions of constraints.

[5] For the sake of readability, the two variables named `FromState` and `ToState` are also used in the concrete implementation of $\mathscr{B}$ (cf., Section 3.3 and Fig. 6).

[6] Web sites for some CLP(FD) systems. SICStus: `www.sics.se/sicstus.html`, ECLiPSe: `http://87.230.22.228/`, BProlog: `http://www.probp.com/`

$$\texttt{Dyn}^u_\ell \leftrightarrow \bigvee_{j=1}^{m_\ell} (\alpha^u_{\ell,j} \wedge A^u_{a_{i_{\ell,j}}}), \quad (3)$$

$$\texttt{Stat}^v_\ell \leftrightarrow \bigvee_{j=1}^{h_\ell} \gamma^v_{\ell,j}, \quad (4)$$

$$\texttt{Fired}^{u,v}_\ell \leftrightarrow \texttt{Dyn}^u_\ell \vee \texttt{Stat}^v_\ell, \quad (5)$$

$$\neg\texttt{Fired}^{u,v}_\ell \vee \neg\texttt{Fired}^{u,v}_{\bar{\ell}}, \quad (6)$$

$$F^v_\ell \leftrightarrow \texttt{Fired}^{u,v}_\ell \vee (\neg\texttt{Fired}^{u,v}_{\bar{\ell}} \wedge F^u_\ell). \quad (7)$$

Fig. 3. The constraint $C^{u,v}_\ell$ for the fluent literal ℓ (cf., Section 3.1).

Given a conjunction of literals $\alpha = [\ell_1, \ldots, \ell_m]$ we will denote with α^u the expression $F^u_{\ell_1} \wedge \ldots \wedge F^u_{\ell_m}$. We will also introduce, for each action a, a Boolean variable A^u_a, representing whether the action is executed or not in the transition from u to v under consideration.

Given a specific fluent literal ℓ, we develop constraints that determine when F^v_ℓ is true and false. Let us consider the dynamic causal laws that have ℓ as a consequence:

$$\texttt{causes}(a_{i_{\ell,1}}, \ell, \alpha_{\ell,1}) \quad \cdots \quad \texttt{causes}(a_{i_{\ell,m_\ell}}, \ell, \alpha_{\ell,m_\ell}).$$

Let us also consider the static causal laws related to ℓ

$$\texttt{caused}(\gamma_{\ell,1}, \ell) \quad \cdots \quad \texttt{caused}(\gamma_{\ell,h_\ell}, \ell).$$

Finally, for each action a we will have its executability conditions:

$$\texttt{executable}(a, \delta_{a,1}) \quad \cdots \quad \texttt{executable}(a, \delta_{a,p_a}).$$

Figure 3 describes the Boolean constraints that can be used in encoding the relations that determine the truth value of the fluent literal ℓ. In the table, we denote with $\bar{\ell}$ the complement of literal ℓ, i.e., if ℓ is the fluent f, then $\bar{\ell}$ is $\texttt{neg}(f)$, while if ℓ is the literal $\texttt{neg}(f)$ then $\bar{\ell}$ is the fluent f. The intuitive meaning of the constraints is as follows:

(3) This constraint states that dynamic causal laws making ℓ true can fire if their conditions are satisfied and the corresponding actions are chosen for execution.
(4) This constraint captures the fact that at least one of the static causal laws that make f true is applicable.
(5) This constraint expresses the fact that a fluent literal ℓ can be made true during a transition form state u to state v, either by a dynamic causal law (determined by $\texttt{Dyn}^u_\ell$) or a static causal law (determined by $\texttt{Stat}^v_\ell$).
(6) This constraint is used to guarantee consistency of the action theory – in no situations a fluent and its complement are both made true.
(7) This constraint expresses the fact that a fluent literal ℓ is true in the destination state if and only if it is made true (by a static or a dynamic causal law) or if is true in the initial state and its truth value is not modified by the transition (i.e., inertia). Observe the similarity between this constraint and the successor state axiom commonly encountered in situation calculus (Levesque *et al.* 1998).

We will denote with $C_\ell^{u,v}$ the conjunction of such constraints.

Given an action domain specification over the signature $\langle \mathscr{V}, \mathscr{F}, \mathscr{A} \rangle$ and two states u and v, we introduce the system of constraints $C_{\mathscr{F}}^{u,v}$ which includes:

- for each fluent literal ℓ in the language of $\mathscr{F}$, the constraints $C_\ell^{u,v}$.
- the constraint

$$\sum_{a \in \mathscr{A}} A_a^u = 1, \tag{8}$$

- for each action $a \in \mathscr{A}$, the constraints

$$A_a^u \rightarrow \bigvee_{j=1}^{p_a} \delta_{a,j}^u. \tag{9}$$

Notice that the sequentiality of the plan if imposed through the constraint (8), while constraint (9) reflects actions' executability conditions.

3.2 Soundness and completeness results

Let us proceed with the soundness and completeness proofs of the constraint-based encoding. Consider a state transition from the state u to the state v and the corresponding constraint $C_f^{u,v}$ described earlier.

Let $S = Lit(u)$ and $S' = Lit(v)$ be the sets of fluent literals that hold in u and v, respectively. Note that, from any specific S (resp., S'), we can obtain a consistent assignment σ_S (resp., $\sigma_{S'}$) of truth values for all the variables F_f^u (resp., F_f^v) of u (resp., v). Conversely, each truth assignment σ_S (resp., $\sigma_{S'}$) for all variables F_f^u (resp., F_f^v) corresponds to a consistent and complete set of fluents S (resp., S').

Regarding the occurrence of actions, recall that in each state transition a single action a occurs and its occurrence is encoded by a specific Boolean variable, A_a^u. Let σ_a denote the assignment of truth values for such variables such that $\sigma_a(A_a^u) = 1$ if and only if a occurs in the state transition from u to v[7]. Note that the domains of σ_S, $\sigma_{S'}$, and σ_a are disjoint, so we can safely denote with $\sigma_S \circ \sigma_{S'} \circ \sigma_a$ the composition of the three assignments. With a slight abuse of notation, in what follows we will denote with E the direct effects $E(a, u)$ of an action a in u. Observe that $E \subseteq S'$.

Theorem 1 states the completeness of the system of constrains introduced in Section 3.1. It asserts that for any given $\mathscr{D} = \langle \mathscr{DL}, \mathscr{EL}, \mathscr{SL} \rangle$, if a triple $\langle u, a, v \rangle$ belongs to the transition system described by $\mathscr{D}$, then the assignment $\sigma = \sigma_S \circ \sigma_{S'} \circ \sigma_a$ satisfies the constraint $C_{\mathscr{F}}^{u,v}$.

Theorem 1 (*Completeness*)
Let $\mathscr{D} = \langle \mathscr{DL}, \mathscr{EL}, \mathscr{SL} \rangle$. If $\langle u, a, v \rangle$ belongs to the transition system described by $\mathscr{D}$, then $\sigma_S \circ \sigma_{S'} \circ \sigma_a$ is a solution of the constraint $C_{\mathscr{F}}^{u,v}$.

[7] We will use mapping applications either as $\sigma(X)$ or in postfix notation as $X\sigma$.

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

Proof

In constraints (3)–(7) of Figures 3 and (8)–(9) defined at the end of Subsection 3.1, a number of auxiliary constraint variables are defined, whose values are uniquely determined once the values of the fluents are assessed. In other words, when S, S', and a are fixed, the substitution $\sigma_S \circ \sigma_{S'} \circ \sigma_a$ uniquely determines the value of the right-hand sides of the constraints (3)–(5). To prove the theorem, we need to verify that if $S' = \texttt{Clo}_{\mathscr{SL}}(E \cup (S \cap S'))$, then the constraints (6) and (7) along with the constraints about the action variables A^u_a (i.e., constraints of the form (8) and (9)) are satisfied for every fluent f.

Let us observe that (8) is equivalent to say that if A_a is true ($A_a = 1$) then A_b is false for all $b \neq a$. Moreover, it also states that if all A_b for $b \neq a$ are false then A_a is true. Namely, (8) is equivalent to the conjunction, for $a \in \mathscr{A}$ of

$$A_a \leftrightarrow \bigwedge_{b \in \mathscr{A} \setminus \{b\}} \neg A_b.$$

Let us start by looking at the action occurrence. Let a be the action executed in state u, thus $\sigma_a = \{A^u_a/1\} \cup \{A^u_b/0 \mid b \neq a\}$. Hence, (8) is satisfied by σ_a.

Similarly, since the semantics require that actions are executed only if the executability conditions are satisfied, it holds that $S \models \delta_{a,h}$ (for at least one $h \in \{1, \ldots, p_a\}$, corresponding to a condition $\texttt{executable}(a, \delta_{a,h})$ in $\mathscr{SL}$). This quickly leads to $\bigvee_{j=1}^{p_a} \delta^u_{a,j}$ is true, and this allows us to conclude that (9) is satisfied by $\sigma_S \circ \sigma_a$.

Let us now consider the constraints dealing with fluents. We recall that S' is a set of fluent literals that is consistent, complete, and closed w.r.t. $\mathscr{SL}$. Let us consider a fluent f and let us prove that constraint (6) of Figure 3 is satisfied. Assume, by contradiction, that $\texttt{Fired}^{u,v}_f \sigma$ and $\texttt{Fired}^{u,v}_{\texttt{neg}(f)} \sigma$ are both true. Four cases must be considered:

(1) $\texttt{Dyn}^u_f \sigma$ and $\texttt{Dyn}^u_{\texttt{neg}(f)} \sigma$ are true. Since these values are determined by u, a, v, this means that both f and $\texttt{neg}(f)$ belong to $E(a, u)$. Since the closure under $\mathscr{SL}$ is monotonic this means that $Lit(v) = S'$ is inconsistent, representing a contradiction.
(2) $\texttt{Dyn}^u_f \sigma$ and $\texttt{Stat}^v_{\texttt{neg}(f)} \sigma$ are true. This means that f is in $E(a, u)$ and $\texttt{neg}(f)$ is added to S' by the closure operation. This implies that S' is inconsistent, which represents a contradiction.
(3) $\texttt{Stat}^v_f \sigma$ and $\texttt{Dyn}^u_{\texttt{neg}(f)} \sigma$ are true. This leads a contradiction as in the previous case.
(4) $\texttt{Stat}^v_f \sigma$ and $\texttt{Stat}^v_{\texttt{neg}(f)} \sigma$ are true. This means that f and $\texttt{neg}(f)$ are added to S' by the closure operation. Thus, S' is inconsistent, which is a contradiction.

It remains to prove that constraint (7) is satisfied by σ. Let us assume that $f \in S'$. Thus, $F^v_f \sigma_{S'}$ is true. Three cases must be considered.

(1) $f \in E(a, u)$. This means that there is a dynamic causal law $\texttt{causes}(a, f, \alpha_{f,i})$ where $S \models \alpha_{f,i}$. From the definition, this leads to $\alpha^u_{f,i} \sigma$ being true and $\sigma_a(A^u_a) = 1$. Thus, constraints (3) and (5) set $\texttt{Dyn}^u_f \sigma$ and $\texttt{Fired}^{u,v}_f \sigma$ both true. As a consequence, constraint (7) is satisfied.
(2) $f \notin E(a, u)$ and $f \in S$. This means that $f \in S \cap S'$. In this case $\texttt{Fired}^{u,v}_{\texttt{neg}(f)} \sigma$ must be false, otherwise S' would be inconsistent (by closure). Thus, $F^u_f \sigma_S$ should be

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

true, $F_f^v \sigma_{S'}$ is true and $\texttt{Fired}_{\texttt{neg}(f)}^{u,v}\sigma$ is false, which satisfy constraint (7) (regardless of the value of $\texttt{Fired}_f^{u,v}\sigma$).

(3) $f \notin E(a,u)$ and $f \notin S$. This means that f is inserted in S' by closure. Thus, there is a static causal law of the form $\texttt{caused}(\gamma_{f,j}, f)$ such that $S' \models \gamma_{f,j}$. In this case, by (4), $\texttt{Stat}_f^v \sigma$ is true and, by (5), so is $\texttt{Fired}_f^{u,v}\sigma$. Thus, constraint (7) 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 domain description where $\mathscr{F} = \{f, g, h\}$ and $\mathscr{A} = \{a\}$, with the following laws:

```
executable(a,[]).     caused([g],h).
causes(a,f,[]).       caused([h],g).
```

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

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

Proof
It is immediate to see that σ_S and $\sigma_{S'}$ uniquely determines two consistent and complete sets of fluent literals u and v. Let f be a positive fluent in $\texttt{Clo}_{\mathscr{SL}}(E(a,u) \cup (S \cap S'))$. We show now that $f \in S'$.

(1) If f is in $S \cap S'$ we are done.

(2) If $f \in E(a,u)$, there is a law $\texttt{causes}(a, f, \alpha_{f,i})$ such that $S \models \alpha_{f,i}$. Since S is determined by σ_S, by (3), we have that $\sigma_S \circ \sigma_a$ is a solution of $\alpha_{f,i}^u \wedge A_a^u$, which implies that $\texttt{Dyn}_f^u$ is true, and $\sigma_{S'}(F_f^v)$ is true in $\sigma_{S'}$. Therefore, $f \in S'$. Observe also that σ_a making true A_a^u will imply that $\delta_{a,h}^u$ is true (for some $h \in \{1, \dots, p_a\}$), which will imply satisfiability of the executability preconditions for a.

(3) We are left with the case of $f \notin E(a,u)$ and $f \notin S \cap S'$. Since S' is determined by $\sigma_{S'}$, and $f \in \texttt{Clo}_{\mathscr{SL}}(E(a,u) \cup (S \cap S'))$, there is a law $\texttt{caused}(\gamma_{f,j}, f)$ such that $S' \models \gamma_{f,j}$, and by construction $\sigma_{S'}$ makes $\gamma_{f,j}^v$ true. Thus, $\texttt{Stat}_f^v$ is true and therefore F_f^v is true. Hence, $f \in S'$.

The proof proceeds similarly in the case of a negative fluent $\texttt{neg}(f)$ in $\texttt{Clo}_{\mathscr{SL}}(E(a,u) \cup (S \cap S'))$. □

Let us consider the set of static causal laws $\mathscr{SL}$. We can introduce a notion of *positive dependence graph*, following the traditional principle of dependence analysis used in logic programming (e.g., Lin and Zhao 2004). The graph $\mathscr{G}(\mathscr{SL})$ is defined as follows:

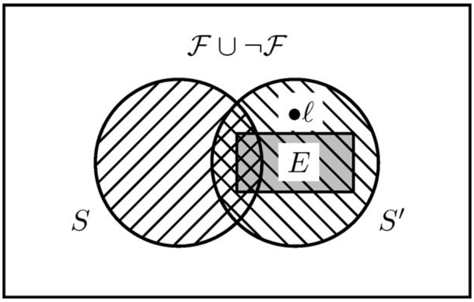


Fig. 4. Sets of fluents involved in a state transition and a literal ℓ introduced by closure.

- the set of the nodes in $\mathscr{G}(\mathscr{SL})$ corresponds to the set of fluent literals, i.e.,

$$Nodes(\mathscr{G}(\mathscr{SL})) = \{f \mid f \in \mathscr{F}\} \cup \{\texttt{neg}(f) \mid f \in \mathscr{F}\}$$

- edges are created to denote the dependence of a fluent literal on other literals due to a static causal law, i.e.,

$$Edges(\mathscr{G}(\mathscr{SL})) = \{(\ell_1, \ell_2) \mid \texttt{caused}(L, \ell_1) \in \mathscr{SL}, L = \texttt{[}\ldots, \ell_2, \ldots\texttt{]}\}$$

A set of fluent literals L is a *loop* if, for any $\ell_1, \ell_2 \in L$, we have that there is a path from ℓ_1 to ℓ_2 in $\mathscr{G}(\mathscr{SL})$ such that all nodes encountered in such path are in L. We say that a domain specification $\mathscr{D} = \langle \mathscr{DL}, \mathscr{EL}, \mathscr{SL} \rangle$ is *acyclic* if the graph $\mathscr{G}(\mathscr{SL})$ does not contain any loops.

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

Proof
Theorem 2 proves that $\texttt{Clo}_{\mathscr{SL}}(E(a,u) \cup (S \cap S')) \subseteq S'$. It remains to prove that for any (positive or negative) fluent ℓ, if $\ell \in S'$, then $\ell \in \texttt{Clo}_{\mathscr{SL}}(E(a,u) \cup (S \cap S'))$.

If $\ell \in E(a,u)$ or $\ell \in S$, then trivially $\ell \in \texttt{Clo}_{\mathscr{SL}}(E(a,u) \cup (S \cap S'))$.
Let us prove that (cf., Fig. 4)

$$(\ell \in S' \wedge \ell \notin E(a,u) \cup (S \cap S')) \rightarrow \ell \in \texttt{Clo}_{\mathscr{SL}}(E(a,u) \cup (S \cap S')).$$

To this aim, consider the dependence graph $\mathscr{G}(\mathscr{SL})$. Because of the acyclicity of $\mathscr{G}(\mathscr{SL})$, there are nodes in $\mathscr{G}(\mathscr{SL})$ without incoming edges – we will refer to them as *leaves*. For any node ℓ of $\mathscr{G}(\mathscr{SL})$, let $d(\ell)$ denote the length of the longest path from a leaf of $\mathscr{G}(\mathscr{SL})$ to ℓ. We prove the property for a positive fluent literal $\ell = f$, by induction on $d(\ell)$.

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

- There is no law of the form $\texttt{caused}(\_, f)$ in $\mathscr{SL}$. In this case, it cannot be that $f \in S'$ due to constraint (4).
- There is a law $\texttt{caused}(\texttt{[]}, f)$. In this case $f \in S'$ by closure.

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

Inductive step. Let $f \notin E(a,u) \cup (S \cap S')$ be a positive fluent such that there are laws $\texttt{caused}(\gamma_{f,1}, f), \ldots, \texttt{caused}(\gamma_{f,h}, f)$ in $\mathscr{SL}$. By the inductive hypothesis, let us assume that the thesis holds for each fluent literal ℓ such that $d(\ell) < d(f)$. Since $f \notin E(a,u)$ and $f \notin S \cap S'$, we have that F_f^u is false, F_f^v is true, and $\texttt{Dyn}_f^u$ is false under $\sigma_S \circ \sigma_{S'} \circ \sigma_a$. From the fact that constraint (7) is satisfied, it follows that $\texttt{Stat}_f^v$ is true. Moreover, $\texttt{Dyn}_f^u$ is false because $f \notin E(a,u)$. On the other hand, because of (6), we have that $\texttt{Dyn}_{\texttt{neg}(f)}^u$, $\texttt{Stat}_{\texttt{neg}(f)}^v$, and $\texttt{Fired}_{\texttt{neg}(f)}^{u,v}$ are all false. Consequently, constraint (7) can be rewritten as $F_f^v \leftrightarrow \bigvee_{j=1}^h \gamma_{f,j}^v$. Since $f \in S'$ (i.e., F_f^v is true), there must exists a $j \in \{1, \ldots, h\}$ such that $\gamma_{f,j}^v$ is verified by $\sigma_{S'}$. This implies that, for each fluent g required to be true (resp., false) in $\gamma_{f,j}$, F_g^v is set true (resp., false) by $\sigma_{S'}$. By inductive hypothesis, such fluent literals (either g or $\texttt{neg}(g)$) belong to $\texttt{Clo}_{\mathscr{SL}}(E(a,u) \cup (S \cap S'))$. Since $\texttt{Clo}_{\mathscr{SL}}(E(a,u) \cup (S \cap S'))$ is closed w.r.t. the static laws, it follows that $f \in \texttt{Clo}_{\mathscr{SL}}(E(a,u) \cup (S \cap S'))$.

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

In order to achieve soundness in cases where the graph $\mathscr{G}(\mathscr{SL})$ contains loops, it is necessary to introduce additional constraints in conjunction with $C_{\mathscr{F}}^{u,v}$. Intuitively, in the semantics of $\mathscr{B}$, cyclic dependencies created by the static causal laws are resolved by the closure operation $\texttt{Clo}_{\mathscr{SL}}(\cdot)$ by minimizing the number of fluent literals that are made true – this derives by the implicit minimality of the closure. Additional constraints can be added to enforce this behavior; these constraints can be derived by following a principle similar to that of *loop formulae* commonly used in the context of logic programming (Lin and Zhao 2004).

The notion of loop formulae can be developed in our context as follows. Let $L = \{\ell_1, \ldots, \ell_k\}$ be a loop in $\mathscr{G}(\mathscr{SL})$ and let us consider the transition from u to v as studied earlier. Let us define a *counter-support* for ℓ_i w.r.t. the loop L as a set of constraints cs with the following properties:

- for each $\texttt{causes}(a_j, \ell_i, \alpha)$ in $\mathscr{DL}$, cs contains either $A_{a_j}^u = 0$ or $F_{\bar{\ell}}^u = 1$ for some ℓ in α;
- for each $\texttt{caused}(\gamma, \ell_i)$ in $\mathscr{SL}$ such that none of $\ell_1, \ldots, \ell_k$ is in γ, for some ℓ in γ cs contains $F_{\bar{\ell}}^v = 1$;
- cs contains either $F_{\bar{\ell}_i}^u = 1$ or $F_{\bar{\ell}_i}^v = 1$.

(As usual, we might identify a set cs of constraint with their conjunction, depending on the need.) Let us denote with $Counters(\ell_i, L)^{u,v}$ the set of all such counter-supports. The loop formulae for L w.r.t. u, v is the set of constraints

$$Form(L)^{u,v} = \{c_1 \wedge \cdots \wedge c_k \rightarrow F_{\ell_1}^v = 0 \wedge \cdots \wedge F_{\ell_k}^v = 0 \mid c_i \in Counters(\ell_i, L)^{u,v}\}.$$

To take into account all different loops in $\mathscr{G}(\mathscr{SL})$, let $Form(\mathscr{D})^{u,v}$ be the constraint

$$Form(\mathscr{D})^{u,v} = \bigwedge_{L \text{ is a loop in } \mathscr{G}(\mathscr{SL})} Form(L)^{u,v}.$$

Following the analogous proofs relating answer sets and models of a program completion that satisfies loop formulae (e.g., Lin and Zhao 2004) one can show:

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

Theorem 4 (*Soundness*)
Let $\mathscr{D} = \langle \mathscr{DL}, \mathscr{EL}, \mathscr{SL} \rangle$ and let $\sigma_S \circ \sigma_{S'} \circ \sigma_a$ be a solution of the constraint $C^{u,v}_{\mathscr{F}} \wedge Form(\mathscr{D})^{u,v}$. Thus, $\mathtt{Clo}_{\mathscr{SL}}(E(a,u) \cup (S \cap S')) = S'$.

Let the action description $\mathscr{D}$ meet the conditions of Theorem 4 and let $\langle \mathscr{S}, v, R \rangle$ be its underlying transition system. The following can be proved.

Theorem 5
There is a trajectory $\langle s_0, a_1, s_1, a_2, \ldots, a_{\mathbf{N}}, s_{\mathbf{N}} \rangle$ in the transition system $\langle \mathscr{S}, v, R \rangle$ if and only if s_0 is closed w.r.t. $\mathscr{SL}$ and there is a solution for the constraint

$$\bigwedge_{j=0}^{\mathbf{N}-1} \left(C^{s_j,s_{j+1}}_{\mathscr{F}} \wedge Form(\mathscr{D})^{s_j,s_{j+1}} \right).$$

Proof
The result follows directly by application of Theorems 1 and 4 and by observing that for each transition $\langle s_j, a_{j+1}, s_{j+1} \rangle$, the satisfaction of constraint $C^{s_j,s_{j+1}}_{\mathscr{F}}$ implies that the state s_{j+1} is closed w.r.t. $\mathscr{SL}$. □

Let $\langle \mathscr{D}, \mathscr{O} \rangle$ be an instance of a planning problem where $\mathscr{D}$ is an action description and $\mathscr{O}$ contains any number of axioms of the form `initially`(C) and `goal`(C). We can state the following.

Corollary 1
There is a trajectory $\langle s_0, a_1, s_1, a_2, \ldots, a_{\mathbf{N}}, s_{\mathbf{N}} \rangle$ for the planning problem $\langle \mathscr{D}, \mathscr{O} \rangle$ if and only if s_0 is closed w.r.t. the static causal laws of $\mathscr{D}$ and there is a solution for the constraint

$$\bigwedge_{\mathtt{initially}(C) \in \mathscr{O}} C^{s_0} \wedge \bigwedge_{j=0}^{\mathbf{N}-1} \left(C^{s_j,s_{j+1}}_{\mathscr{F}} \wedge Form(\mathscr{D})^{s_j,s_{j+1}} \right) \wedge \bigwedge_{\mathtt{goal}(C) \in \mathscr{O}} C^{s_{\mathbf{N}}}.$$

3.3 Mapping the model to CLP(FD)

The modeling described in Section 3.1 has been translated into a concrete implementation using SICStus Prolog. In this translation, constrained CLP variables directly reflect the Boolean variables modeling fluents and action's occurrences. Consequently, causal laws and executability conditions are directly translated into CLP constraints (and inherit the corresponding completeness and soundness results). In this section we highlight the main aspects of the implementation – while the complete code can be found at `www.dimi.uniud.it/dovier/CLPASP`.

A plan with exactly $\mathbf{N}+1$ states, p fluents, and m actions is represented by

- A list, called `States`, containing $\mathbf{N}+1$ lists, each composed of p terms of the form `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 $\mathbf{N}=2$ and the fluents `f`, `g`, and `h`, we have

```
States = [[fluent(f,X_f_0),fluent(g,X_g_0),fluent(h,X_h_0)],
          [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)]].
```

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

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

Fig. 5. Main predicate of the CLP(FD) planner.

- A list `ActionsOcc`, containing **N** 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 $\mathbf{N} = 2$ and the actions are `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 makes use of these structures in the construction of the plan; appropriate constraints are set between the various Boolean variables to capture their relationships. For each list in `ActionsOcc`, exactly one `action(a`$_i$`,VA`$_i$`)` contains a variable that is assigned the value 1 (cf., constraint (8)).

The CLP implementation of the $\mathcal{B}$ language assumes that the action description is encoded as Prolog facts – observe that the syntax of $\mathcal{B}$ is compliant with Prolog's syntax, allowing us to directly store the domain description as rules and facts in the Prolog database. The entry point of the planner is shown in Figure 5.

The main predicate is `clpplan(N, ActionsOcc, States)` (line (1)) that computes a plan of length **N** for the action description present in the Prolog database. If such a plan exists, the variables in `ActionsOcc` and `States` will be instantiated so as to describe the found trajectory.

Lines (2) and (3) collect the lists of all fluents (`Lf`) and all actions (`La`). Lines (4) and (5) are used for the creation of the lists `States` and `ActionsOcc`. 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.

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, as described in Section 3.1. 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. Some details on the labeling strategy are

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

(27)  dynamic([], _, _, [], _).
(28)  dynamic([[Action,Precondition]|R], Occ, FromState, [Flag|Flags], EV) :-
(29)    member(action(Action,VA), Occ),
(30)    get_precondition_vars(Precondition, FromState, ListPV),
(31)    length(ListPV, NPrec),
(32)    sum(ListPV, SumPrec),
(33)    (VA #/\ (SumPrec #= NPrec)) #<=> Flag,
(34)    dynamic(R, Occ, FromState, Flags, EV).

(35)  static([], _, [], _).
(36)  static([Cond|Others], ToState, [Flag|Flags], EV) :-
(37)    get_precondition_vars(Cond, ToState, ListPV),
(38)    length(ListPV, NPrec),
(39)    sum(ListPV, SumPV),
(40)    (SumPV #= NPrec) #<=> Flag,
(41)    static(Others, ToState, Flags, EV).
```

Fig. 6. Transition from state to state.

discussed in Section 8. Please observe that in the code of Figure 5 we omit the parts concerning delivering the results to the user.

The main constraints are added by the predicate `set_transitions`. The process is based on a recursion across fluents and consecutive states. The predicate `set_one_fluent` is called (see Fig. 6) at the core of the recursion. Its parameters are the fluent `F`, the starting state `FromState`, the next state `ToState`, the list `Occ` of action variables, and finally the variables `IV` and `EV`, related to the value of the fluent `F` in `FromState` and `ToState`, respectively (see also Fig. 2).

For a given fluent `F`, the predicate `set_one_fluent` collects the list `DynPos` (resp. `DynNeg`) of all the pairs `[Action,Preconditions]` such that the dynamic action `Action` 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 are handled by collecting the lists of conditions that affect the truth value of a fluent `F` (i.e., 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 relationships between all these variables. Line (25) implements the constraint (6) for the state `ToState` of Figure 3, stating that we do not want inconsistent action theories. 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.

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

```
(42)    set_executability_sub([], _, _).
(43)    set_executability_sub([[Act,C]|CA], ActionsOcc, State) :-
(44)       member(action(Act,VA), ActionsOcc),
(45)       preconditions_flags(C, State, Flags),
(46)       bool_disj(Flags, F),
(47)       VA #==> F,
(48)       set_executability_sub(CA, ActionsOcc, State).
(49)    preconditions_flags([], _, []).
(50)    preconditions_flags([C|R], State, [Flag|Flags]) :-
(51)       get_precondition_vars(C, State, Cs),
(52)       length(Cs, NCs),
(53)       sum(Cs, SumCs),
(54)       (NCs #= SumCs) #<=> Flag,
(55)       preconditions_flags(R, State, Flags).
```

Fig. 7. Executability conditions.

Let us consider the predicate `dynamic` (see line (27) in Fig. 6). It recursively processes a list of pairs `[Action,Preconditions]`. The variable `VA` associated to the execution of action `Action` is retrieved in line (29). The variables associated to its preconditions are retrieved from state `FromState` 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, i.e., 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` (line (35) in Fig. 6) recursively processes a list of preconditions. The variables involved in each of such precondition `Cond` are retrieved from the state `ToState` and collected in `ListPV` (line (37)). A precondition holds if and only if all the variables in the list `ListPV` have value 1, i.e., when their sum is equal to the length, `NPrec`, of `ListPV`. This happens if and only if there is a static action effect (see 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. 7). The variable `VA`, encoding the application of an action `Act` is collected in line (44). 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.

4 The action language with constraints on multivalued fluents

As a matter of fact, constraints represent a very declarative notation to express relationships between unknowns. As such, the ability to use them directly in an action language greatly enhances the declarative and expressive power of the language, facilitating the encoding of complex action domains, such as those involving multivalued fluents. Furthermore, the encoding of an action theory using multivalued fluents leads to more concise and more efficient representations and better exposing nondeterminism (that could be exploited, e.g., by a parallel planner). Let us consider some representative examples.

Example 2 (*Maintenance goals*)
It is not uncommon to encounter planning problems where along with the type of goals described earlier (known as *achievement* goals), there are also *maintenance* goals, representing properties that must persist throughout the trajectory. Constraints are a natural way of encoding maintenance properties, and can be introduced along with simple temporal operators. For example, if the fluent *fuel* represents the amount of fuel available, then the maintenance goal which guarantees that we will not be left stranded could be encoded as: always($fuel > 0$). □

Example 3 (*Control knowledge*)
Domain-specific control knowledge can be formalized as constraints that we expect to be satisfied by all the trajectories. For example, we may know that if a certain action occurs at a given time step (e.g., *ingest_poison*) then at the next time step we will always perform the same action (e.g., *call_doctor*). This could be encoded as

$$\texttt{caused([occ(}\mathit{ingest\_poison}\texttt{)], occ(}\mathit{call\_doctor}\texttt{)}^{1}\texttt{)},$$

where occ(a) is a fluent describing the occurrence of the action a and f^1 indicates that the fluent f should hold at the next time step. □

Example 4 (*Delayed effect*)
Let us assume that the action *request_reimbursement* has a delayed effect (e.g., the increase by \$50 of *bank_account* after 30 time units). This could be expressed as a dynamic causal law:

$$\texttt{causes(}\mathit{request\_reimbursement}\texttt{,incr(}\mathit{bank\_account}\texttt{,}50\texttt{)}^{30}\texttt{,[])},$$

where incr is a constraint introduced to deal with additive computations – in a way closer to $\mathscr{B}$'s syntax we should write:

$$\texttt{causes(}\mathit{request\_reimbursement}\texttt{,}\mathit{bank\_account}^{30}\ \texttt{=}\ \mathit{bank\_account} + 50\texttt{,[])}.$$

This is a particular case of additive fluents (Lee and Lifschitz 2003). □

In what follows we introduce the action description language $\mathscr{B}^{MV}$ in which multivalued fluents are admitted and constraints are first-class components in the description of planning problems. The availability of multivalued constraints enables a number of immediate language extensions and improves the expressive power of the overall framework.

Action description languages such as $\mathscr{B}$ rely on the common assumption, traditionally referred to as *Markovian property* in the context of systems and control theory: the executability of an action and its effects depend exclusively on the current state of the world (McCarthy 1998; Gabaldon 2002). Nevertheless, it is not uncommon to encounter real world situations where such property is not satisfied, i.e., situations where the executability and/or the effects of an action depend not only on what holds in the current situation, but also on whether some conditions were satisfied at a previous point in time. For example, an agent controlling access to a database should forbid access if in the recent past three failed password submission attempts have been performed by the user.

Although non-Markovian preconditions and effects can be expressed in a Markovian theory through the introduction of additional fluents (and a correct handling of inertia), the resulting theory can become significantly larger and less intuitive. An alternative solution consists of admitting past references in modeling such kind of situations. In this frame of mind, $\mathscr{B}^{MV}$ allows timed references to past points in time within constraints, i.e., non-Markovian expressions that might involve fluents' values. Effects of dynamic laws that involves future references might also be specified. As a further feature the $\mathscr{B}^{MV}$ language admits the specification of global constraints (involving absolutely specified points in time) and costs for actions and plans.

The resulting description language supports all the kind of modeling and reasoning outlined in the above Examples 2–4.

In the next sections, we first introduce the syntax of the full-blown action description language $\mathscr{B}^{MV}$(Section 5). In Section 6 we will develop the semantics and the constraint-based abstract implementation of this new language. In doing this, for the sake of readability, we proceed incrementally in order to focus on the main points and features of the framework. We first consider the sublanguage $\mathscr{B}_0^{MV}$ obtained from $\mathscr{B}^{MV}$ by disallowing timed references (Section 6.1); in Section 6.2, we treat the general case dealing with past and future references. The abstract implementation is provided in Section 6.3. Finally, we give the semantics to the complete language involving cost and global constraints (Section 6.4).

5 The language $\mathscr{B}^{MV}$

As for $\mathscr{B}$, the action signature consists of a set $\mathscr{F}$ of fluent names, a set $\mathscr{A}$ of action names, and a set $\mathscr{V}$ of values for fluents in $\mathscr{F}$. In the following we assume that $\mathscr{V} \subseteq \mathbb{Z}$.

In an action domain description, an assertion (*domain declaration*) of the type

$$\texttt{fluent}(f, \{d_1, \dots, d_k\})$$

declares that f is a fluent and that its set of values is $\{d_1, \dots, d_k\}$; we refer to the set $\{d_1, \dots, d_k\}$ as the *domain* of f. We also admit the simplified notation $\texttt{fluent}(f, d_1, d_2)$ to specify all the integer values in the interval $[d_1, d_2]$ as admissible (with $d_1 \leqslant d_2$).

An *annotated fluent* (AF) is an expression f^t, where f is a fluent and $t \in \mathbb{Z}$. We will often denote f^0 simply by f. Intuitively speaking, if $t < 0$ then f^t denotes the value that the fluent f had t steps ago in the past; similarly, if $t > 0$, then f^t denotes the value f will have t steps in the future. We refer to annotated fluents with $t > 0$ as positively annotated fluents.

Annotated fluents can be used in *Fluent Expressions* (FE), which are defined inductively as follows:

$$\texttt{FE} ::= d \mid \texttt{AF} \mid \texttt{FE}_1 \oplus \texttt{FE}_2 \mid -(\texttt{FE}) \mid \texttt{abs(FE)} \mid \texttt{rei(FC)}$$

where $d \in \mathscr{V}$ and $\oplus \in \{+, -, *, /, \texttt{mod}\}$. FC is a fluent constraint (see below). We refer to the fluent expressions rei(FC) as the *reification* of the fluent constraint FC – its formal semantics is given in Section 6.1.

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

Fluent expressions can be used to build *primitive fluent constraints* (PC), i.e., formulae of the form $\texttt{FE}_1 \texttt{ op } \texttt{FE}_2$, where $\texttt{FE}_1$ and $\texttt{FE}_2$ are fluent expressions, and op is a relational operator, i.e., op $\in \{=, \neq, \geqslant, \leqslant, >, <\}$. *Fluent constraints* are propositional combinations of primitive fluent constraints:

$$\texttt{PC} ::= \texttt{FE}_1 \texttt{ op } \texttt{FE}_2,$$
$$\texttt{C} ::= \texttt{PC} \mid \neg\texttt{C} \mid \texttt{C}_1 \wedge \texttt{C}_2 \mid \texttt{C}_1 \vee \texttt{C}_2.$$

The constant symbols true and false can be used as a shorthand for true constraints (e.g., $d = d$, for some $d \in \mathscr{V}$) and unsatisfiable constraints (e.g., $d \neq d$).

The language $\mathscr{B}^{MV}$ allows one to specify an action domain description, which relates actions, states, and fluents using axioms of the following forms (PC denotes a primitive fluent constraint, while C is a fluent constraint).

- Axioms of the form $\texttt{executable}(a, C)$, stating that the fluent constraint C has to be satisfied by the current state for the action a to be executable.
- Axioms of the form $\texttt{causes}(a, PC, C)$ encode dynamic causal laws. When the action a is executed, if the constraint C is satisfied by the current state, then state produced by the execution of the action is required to satisfy the primitive fluent constraint PC.
- Axioms of the form $\texttt{caused}(C_1, C_2)$ describe static causal laws. If the fluent constraint C_1 is satisfied in a state, then the constraint C_2 must also hold in such state.

An *action domain description* of $\mathscr{B}^{MV}$ is a tuple $\langle \mathscr{DL}, \mathscr{EL}, \mathscr{SL} \rangle$, where $\mathscr{EL}$ is a set of executability conditions, $\mathscr{SL}$ is a set of static causal laws, and $\mathscr{DL}$ is a set of dynamic causal laws. In the following, we assume that positively annotated fluents can occur only in the effect part of dynamic causal laws.

A specific instance of a planning problem is a pair $\langle \mathscr{D}, \mathscr{O} \rangle$, where $\mathscr{D}$ is an action domain description and $\mathscr{O}$ contains any number of axioms of the form $\texttt{initially}(C)$ and $\texttt{goal}(C)$, where C is a fluent constraint.

Example 5

A sample action theory in $\mathscr{B}^{MV}$ is

$\texttt{fluent}(f,\ \{1,2,3,4,5\}).$
$\texttt{fluent}(g,\ \{1,2,3,4,5\}).$
$\texttt{fluent}(h,\ \{1,2,3,4,5\}).$
$\texttt{causes}(a,\ f = g + 2,\ g < 3).$
$\texttt{executable}(a,\ \texttt{true}).$
$\texttt{initially}(f = 1).$
$\texttt{initially}(g = 1).$
$\texttt{initially}(h = 1).$
$\texttt{goal}(f = 5).$ □

Notice that, for any given dynamic law $\texttt{causes}(a, PC, C)$, such that a is executed in a state u satisfying C, the constraint PC has to be evaluates/satisfied in the target state v. Hence, the (relative) timed references occurring in PC (respectively, in C) are resolved with respect to v (resp., u). On the other hand, for a static law

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

caused(C_1, C_2), relative timed references of both C_1 and C_2 have to be resolved with respect to the current state.

5.1 Absolute temporal references

The language $\mathscr{B}^{MV}$ allows the definition of *absolute temporal constraints*, i.e., constraints that refer to specific moments in time in the trajectory (by associating the time point 0 to the initial state). differently from the case of annotated fluents, where points in time are *relative* to the current state. A *timed fluent* is defined as an expression of the form

$$\texttt{FLUENT} @ \texttt{TIME}.$$

Timed fluents can be used to build *timed fluent expressions* (TE) and *timed primitive constraints* (TC), similarly to what done for normal fluents. For instance, the constraint

$$f@2 < g@4$$

states that the value the fluent f has at time 2 in the plan is less than the value that the fluent g has at time 4. Similarly, $h@2 = 3$ imposes that the fluent h must assume value 3 at time 2.

Timed constraints can be used in the following kind of assertion:

$$\texttt{time\_constraint}(TC).$$

The assertion requires the timed constraint TC to hold.

Some other accepted constraints are

- holds(FC, n): this constraint is a particular case of the previous one. It is satisfied if the primitive fluent constraint FC holds in the nth state. It is therefore a generalization of the initially axiom. Observe that assertions of this kind can be used to guide the search of a plan by adding some pointwise information about the states occurring along the computed trajectory (e.g., this is useful to implement the landmarks model as used in the FF planner (Hoffmann *et al.* 2004).
- always(FC): this constraint imposes the condition that the fluent constraint FC holds in all the states. Observe that FC has to be evaluated in all states, and its evaluation is strict – i.e., any reference to fluents outside the time limits leads to the satisfaction of the constraint; hence, annotated fluents should be avoided in FC.

In specifying a planning problem $\langle \mathscr{D}, \mathscr{O} \rangle$, we can consider such kinds of assertions as part of the observations $\mathscr{O}$.

Example 6

Let us consider the case of an agent that has a certain amount of money (e.g., $5,000) to invest; she is interested in purchasing as many stocks as possible. The stocks can be purchased from three trading agencies (1, 2, and 3); each agency has 1,000 stocks available at $2 each. The stocks have to be purchased in separate transactions, but each trading agency require the agent to have a balance of at least

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

\$2,000 at the start of the day before agreeing in the transaction. A purchase can be of at most 3,000 shares at a time.

We can model this problem with the following fluents:

$$\texttt{fluent}(money, 0, 5000). \quad \texttt{fluent}(have(stock1), 0, 1000).$$
$$\texttt{fluent}(have(stock2), 0, 1000). \quad \texttt{fluent}(have(stock3), 0, 1000).$$
$$\texttt{fluent}(available(stock1), 0, 1000). \quad \texttt{fluent}(available(stock2), 0, 1000).$$
$$\texttt{fluent}(available(stock3), 0, 1000).$$
$$\texttt{fluent}(price(stock1), 2, 2). \quad \texttt{fluent}(price(stock2), 2, 2).$$
$$\texttt{fluent}(price(stock3), 2, 2)).$$

The only action is

$$\texttt{action}(buy(StockType, N)) :- N > 0, N < 3000.$$

The executability condition for the action captures one property: the agent is accepted by the trading agency.

$$\texttt{executable}(buy(Type, N), money@0 > 2000 \wedge money > N * price(Type)).$$

The dynamic causal law for this action is

$$\texttt{causes}(buy(Type, N), money = money - N * price(Type), \texttt{true}).$$
$$\texttt{causes}(buy(Type, N), have(Type) = have(Type) + N, \texttt{true}).$$

The initial state can be described as

$$\texttt{initially}(price(stock1) = 2). \quad \texttt{initially}(price(stock2) = 2).$$
$$\texttt{initially}(price(stock3) = 2). \quad \texttt{initially}(have(stock1) = 0).$$
$$\texttt{initially}(have(stock2) = 0). \quad \texttt{initially}(have(stock3) = 0).$$
$$\texttt{initially}(money = 5000). \quad \texttt{initially}(available(stock1) = 1000).$$
$$\texttt{initially}(available(stock2) = 1000). \quad \texttt{initially}(available(stock3) = 1000). \quad \square$$

5.2 Cost constraints

In $\mathscr{B}^{MV}$ it is possible to specify information about the *cost* of each action and about the *global cost* of a plan (that is defined as the sum of the costs of all its actions). This type of information are useful to explore the use of constraints in determining *optimal* plans.

The cost of actions is expressed using assertions of the following forms (where FE is a fluent expression built using the fluents present in the state):

- action_cost(a, FE) specifies the cost of the execution of the action a as result of the expression FE.
- state_cost(FE) specifies the cost of a state as the result of the evaluation of FE.

Whenever, for an action or a state, no cost declaration is provided, a default cost of 1 is assumed. Once we have provided the costs for actions and states, we can impose constraints on the cumulative costs of specific states or complete trajectories. This can be done in $\mathscr{B}^{MV}$ using assertions of the following types (where k is a number and op a relational operator):

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

- `cost_constraint(plan op` k`)`; the assertion adds a constraint on the global cost of the plan.
- `cost_constraint(goal op` k`)`; the assertion imposes a constraint on the global cost of the final state.
- `cost_constraint(state(i) op` k`)`; the assertion imposes a constraint on the global cost of the `i`th state of the trajectory.

As an immediate generalization of the above constraints, we admit assertions of the form `cost_constraint(`C`)`, where C is a constraint, possibly involving fluents, where the atoms `plan`, `goal`, and `state(i)` might occur in any place where a fluent might – intuitively representing the cost of a plan, of the goal state, and of the `i`th state, respectively.

Some directives can be added to an action theory to select optimal solutions with respect to the specified costs:

`minimize_cost(`FE`)`,

where FE is an expression involving the atoms `plan`, `goal`, and `state(i)`, and possibly other fluents. This assertion constrains the search to determine a plan that minimizes the value of the expression FE. For instance, the two assertions `minimize_cost(plan)` and `minimize_cost(goal)` constrain the search of a plan with minimal global cost and with minimal cost of the goal state, respectively.

We provide a more precise semantics for all these assertions in Section 6.4. In specifying a planning problem $\langle \mathscr{D}, \mathscr{O} \rangle$, we consider cost constraints as part of the observations $\mathscr{O}$.

6 Semantics and abstract implementation of $\mathscr{B}^{MV}$

We will build the semantics of the language $\mathscr{B}^{MV}$ incrementally. We will start by building the semantics for the sublanguage of $\mathscr{B}^{MV}$ devoid of any form of time reference and cost constraints (Section 6.1). This core language is called $\mathscr{B}_0^{MV}$. The subsequent Sections 6.2–6.4 treat the full $\mathscr{B}^{MV}$.

6.1 Semantics for timeless constraints

Each fluent f is uniquely assigned to a domain $\text{dom}(f)$ in the following way:

- if `fluent(`f, Set`)` $\in \mathscr{D}$, then $\text{dom}(f) = Set$.

A function $v : \mathscr{F} \to \mathscr{V} \cup \{\bot\}$ is a *state* if $v(f) \in \text{dom}(f) \cup \{\bot\}$ for all $f \in \mathscr{F}$. The special symbol $\bot$ denotes that the value of the fluent is undefined. A state v is *complete* if for all $f \in \mathscr{F}$, $v(f) \neq \bot$. For a number $\mathbf{N} \geqslant 1$, we define a *state sequence* $\vec{v}$ as a tuple $\langle v_0, \ldots, v_{\mathbf{N}} \rangle$ where each v_i is a state.

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

Given a state v, and an expression φ, we define the *value* of φ in v (with abuse of notation, denoted by $v(\varphi)$) as follows[8]:

$$\left.\begin{array}{ll} \bullet & v(x) = x \quad \text{if } x \in \mathscr{V}, \\ \bullet & v(f) = v(f) \quad \text{if } f \in \mathscr{F} \text{ (abuse of notation here)}, \\ \bullet & v(-(\varphi)) = -(v(\varphi)), \\ \bullet & v(\texttt{abs}(\varphi)) = |(v(\varphi))|, \\ \bullet & v(\varphi_1 \oplus \varphi_2) = v(\varphi_1) \oplus v(\varphi_2), \\ \bullet & v(\texttt{rei}(C)) = 1 \ \ \text{if } v \models C, \\ \bullet & v(\texttt{rei}(C)) = 0 \ \ \text{if } v \not\models C. \end{array}\right\} \qquad (10)$$

We treat the interpretation of the various $\oplus$ operations and relations as strict with respect to $\bot$ (i.e., $\bot \oplus x = x \oplus \bot = \bot$, $\texttt{abs}(\bot) = \bot$, etc.).

The last two cases in (10) specify the semantics of reification. Reified constraints are useful to enable reasoning about the satisfaction state of other formulae. The intuitive semantics is that a fluent expression $\texttt{rei}(C)$, where C is a fluent constraint, assumes a Boolean value (0 or 1) depending on the truth of C. Note that the semantics of reified constrains relies on the notion of *satisfaction*, which in turn is defined by structural induction on constrains, as follows. Given a primitive fluent constraint $\varphi_1 \texttt{ op } \varphi_2$, a state v *satisfies* $\varphi_1 \texttt{ op } \varphi_2$, written $v \models \varphi_1 \texttt{ op } \varphi_2$, if and only if it holds that $v(\varphi_1) \texttt{ op } v(\varphi_2)$ where the semantics of the arithmetic relators/operators is the usual one on $\mathbb{Z}$. If either $v(\varphi_1)$ or $v(\varphi_2)$ is $\bot$, we assume that $v \not\models \varphi_1 \texttt{ op } \varphi_2$ (and $v \not\models \varphi_1 \texttt{ nop } \varphi_2$ where $\texttt{nop}$ is the negation of the operator $\texttt{op}$). Basically, undefined formulas are neither proved nor disproved. The satisfaction relation $\models$ can be generalized to the case of propositional combinations of fluent constraints in the usual manner.

Given a constraint C, let $\mathsf{fluents}(C)$ be the set of fluents occurring in it. A function $\sigma : \mathsf{fluents}(C) \longrightarrow \mathscr{V}$ is a *solution* of C if $\sigma \models C$. We denote the domain $\mathsf{fluents}(C)$ of the function σ as $\mathrm{dom}(\sigma)$. In other words, a solution σ of C can be seen as a partial state satisfying C. Observe that we require the solution to manipulate exclusively the fluents that appear in the constraint.

Example 7
Let us consider an action theory over the fluents f, g, h, where each fluent has domain $\{1, \ldots, 5\}$. If C is the constraint $f > g + 2$, then a solution of C is $\sigma = \{f/5, g/2\}$. Note that the substitution $\theta = \{f/5, g/2, h/1\}$ is not a solution of C, since $\mathrm{dom}(\theta) \neq \mathsf{fluents}(f > g + 2)$. □

Let σ be a solution of a constraint C and v a state, with $\mathsf{ine}(\sigma, v)$ we denote the state obtained completing σ in v *by inertia*, as follows:

$$\mathsf{ine}(\sigma, v)(f) = \begin{cases} \sigma(f) & \text{if } f \in \mathrm{dom}(\sigma) \\ v(f) & \text{otherwise} \end{cases}$$

[8] The expression $|n|$ denotes the (algebraic) absolute value of n.

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

Example 8
Let us continue with Example 7. If $\sigma = \{f/5, g/2\}$ and $v = \{f/1, g/1, h/1\}$, then $\mathsf{ine}(\sigma, v) = \{f/5, g/2, h/1\}$. □

An action a is *executable* in a state v if there is an axiom $\texttt{executable}(a, C)$ such that $v \models C$.

Remark 3
As for the case of the language $\mathscr{B}$, also in $\mathscr{B}^{MV}$ the executability laws express necessary but not sufficient preconditions for action execution (cf., Remark 2). Moreover, thanks to the generality of the constraint language – i.e., any propositional combination of primitive constraints can be used in $\mathscr{B}^{MV}$– the `executable` laws also allow direct formulation of nonexecutability conditions and the roles of the `executable` and `nonexecutable` axioms coincide.

Let us denote with $Dyn(a)$ the set of dynamic causal law axioms for action a. The *effect* of executing a in state v, denoted by $Eff(a, v)$, is a constraint defined as follows:

$$Eff(a, v) = \bigwedge \{C \mid \mathsf{causes}(a, C, C_1) \in Dyn(a), v \models C_1\}.$$

6.1.1 $\mathscr{B}_0^{MV}$ without static causal laws

Let us start by considering the simplified situation in which $\mathscr{SL} = \emptyset$, i.e., no static causal laws are specified in the domain description.

During the execution of an action a, a fluent has to be considered as inertial, provided that it does not appear among the effects of the dynamic laws for a. In other words, since these effects are expressed through constraints, a fluent is inertial if it does not occur in any of the constraints specified in the dynamic laws for a.

The description of the state transition system corresponding to a given action description theory $\langle \mathscr{DL}, \mathscr{EL}, \emptyset \rangle$ can be completed by defining the notion of transition.

A triplet $\langle v, a, v' \rangle$, where v, v' are complete states and a is an action, is a *valid state transition* if

- the action a is executable in v, and
- $v' = \mathsf{ine}(\sigma, v)$, where σ is a solution of the constraint $Eff(a, v)$.

Let $\langle \mathscr{D}, \mathscr{O} \rangle$ be an instance of a planning problem, $\vec{v} = \langle v_0, \ldots, v_\mathbf{N} \rangle$ be a sequence of complete states and $a_1, \ldots, a_\mathbf{N}$ be actions. We say that $\langle v_0, a_1, v_1, \ldots, a_\mathbf{N}, v_\mathbf{N} \rangle$ is a *valid trajectory* if

- for each axiom of the form $\texttt{initially}(C)$ in $\mathscr{O}$, we have that $v_0 \models C$,
- for each axiom of the form $\texttt{goal}(C)$ in $\mathscr{O}$, we have that $v_\mathbf{N} \models C$, and
- for all $i \in \{0, \ldots, \mathbf{N} - 1\}$, $\langle v_i, a_{i+1}, v_{i+1} \rangle$ is a valid state transition.

Example 9
Let us consider the Example 5. Observe that $\langle \{f/1, g/1, h/1\}, a, \{f/5, g/3, h/1\} \rangle$ is a valid trajectory. □

$$
\begin{array}{rcll}
F_f^v, F_f^u & \in & \mathrm{dom}(f), & (11) \\
A_a^u & \rightarrow & \bigvee_{j=1}^{p_a} \delta_{a,j}^u, & (12) \\
A_{a_{i_{f,j}}}^u \wedge \alpha_{f,j}^u & \leftrightarrow & Dyn_{f,j}^u, \quad \forall j \in \{1,\ldots,m_f\}, & (13) \\
Dyn_{f,j}^u & \rightarrow & C_{f,j}^v, \quad \forall j \in \{1,\ldots,m_f\}, & (14) \\
\neg \bigvee_{j=1}^{m_f} Dyn_{f,j}^u & \rightarrow & F_f^u = F_f^v. & (15)
\end{array}
$$

Fig. 8. The constraints $C_{f,a}^{u,v}$ for a state transition from u to v, for a fluent f.

Remark 4
Given a planning problem $\langle \mathscr{D}, \mathscr{O} \rangle$ in $\mathscr{B}_0^{MV}$, differently from what happens in the case of $\mathscr{B}$, a solution to a planning problem is described by a valid trajectory, not just by a sequence of actions. This is the case because actions might have nondeterministic effects. For instance, let us consider Example 5. If the action `a` is executed and the precondition `g<3` holds, then the dynamic causal law imposes the constraint `f=g+2` in the reached state. There are many different ways to satisfy this requirement. Hence, in general, a sequence of actions might not characterize a unique state sequence.

The same argument also applies to the action description language $\mathscr{B}^{MV}$, so in what follows we will consider the valid trajectories as the solutions of a planning problem.

6.1.2 *Abstract implementation in absence of static laws*

In this section we propose a constraint-based characterization of the state transition system defined in Section 6.1.1. Similarly to what we have done in the case of $\mathscr{B}$, for any specific state, each fluent f will be represented by an integer-valued constraint variable. Boolean variables will instead model the occurrences of actions.

Let u be a state; given a fluent f, we indicate with F_f^u the variable representing f in u. We generalize such a notation to any constraint C, i.e., we denote with C^u the constraint obtained from C by replacing each fluent $f \in \mathsf{fluents}(C)$ by F_f^u. For each action $a \in \mathscr{A}$, a Boolean variable A_a^u is introduced, representing whether the action is executed or not in the transition from u to the next state.

Given a specific fluent f, we develop a system of constraints to constrain the values of F_f^u. Let us consider the dynamic causal laws that have f within their consequences:

$$\mathscr{DL}_f = \{\texttt{causes}(a_{i_{f,1}}, C_{f,1}, \alpha_{f,1}),\ \cdots,\ \texttt{causes}(a_{i_{f,m_f}}, C_{f,m_f}, \alpha_{f,m_f})\}.$$

For each action a we will have its executability conditions:

$$\mathscr{EL}_a = \{\texttt{executable}(a, \delta_{a,1}),\ \cdots,\ \texttt{executable}(a, \delta_{a,p_a})\}.$$

Figure 8 describes the constraints $C_{f,a}^{u,v}$ that can be used in encoding the relations that determine the value of the fluent f in the state v (i.e., constrain the variable F_f^v)

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

w.r.t. the application of the action a in the state u. After the settings of the domains (by (11)), we impose through (12) that if action a is executed, then at least one of the preconditions for its executability must hold in u. For each $j \in \{1, \dots, m_f\}$ the constraint (13) defines a Boolean flag $Dyn^u_{f,j}$ that holds if and only if action $a_{i_{f,j}}$ is applicable in u and the preconditions of the jth dynamic causal law for f holds in u. The constraint (14) requires that if $Dyn^u_{f,j}$ is true, then the corresponding effects must hold in the new state v. Finally, inertia constraints are set by means of (15).

We will denote with $C^{u,v}_f$ the conjunction of these constraints for all actions $a \in \mathscr{A}$. Given an action domain specification over the signature $\langle \mathscr{V}, \mathscr{F}, \mathscr{A} \rangle$ and two states u, v, the system of constraints $C^{u,v}_{\mathscr{F}}$ includes:

- the constraint $C^{u,v}_f$ for each fluent literal f in the language of $\mathscr{F}$,
- the constraint $\sum_{a \in \mathscr{A}} A^u_a = 1$ (unique action execution in the state transition).

The next theorem states completeness and soundness of the encoding described so far. We need a further piece of notation. Given two states u, v, and an action a, let $C^{u,a}_{\mathscr{F}}$ be the constraint obtained from $C^{u,v}_{\mathscr{F}}$ by setting $A_a = 1$, $A_b = 0$ for all $b \neq a$, and $F^u_f = u(f)$ for each fluent literal f.

Theorem 6
Let $\mathscr{D} = \langle \mathscr{DL}, \mathscr{EL}, \emptyset \rangle$ and let u, v two states and a an action. Then $\langle u, a, v \rangle$ is a valid transition in the semantics of the language $\mathscr{B}^{MV}_0$ if and only if v represents a solution of the constraint $C^{u,a}_{\mathscr{F}}$.

Proof

(⇒) Let $\langle u, a, v \rangle$ be a valid transition. Then, a is executable in u. Hence $u \models \delta_{a,j}$ for some $j \in \{1, \dots, p_a\}$ and (12) is satisfied. By the definition of state we have that (11) is also satisfied. Let $v = \mathsf{ine}(\sigma, u)$ with σ solution of $\mathit{Eff}(a, u)$.
If f is a fluent not belonging to $\mathrm{dom}(\sigma)$ then f does not occur in $\mathit{Eff}(a, u)$ and it is not affected by any dynamic causal law involved in the state transition. By definition of $\mathsf{ine}(\cdot)$ we have that $v(f) = u(f)$ and this satisfies constraint (15). Satisfaction of constraints (13) and (14) is immediately verified by observing that for all dynamic causal laws $\mathtt{causes}(a_{i_{f,h}}, C_{f,h}, \alpha_{f,h})$ having f in $C_{f,h}$, the constraint $\alpha_{f,h}$ is false in u. Then, the corresponding flag $Dyn^u_{f,h}$ is set false by (13). Consequently, (14) is satisfied.
Assume now that f is a fluent in $\mathrm{dom}(\sigma)$. This means that there are dynamic causal laws $\mathtt{causes}(a_{i_{f,h}}, C_{f,h}, \alpha_{f,h})$ such that $\alpha_{f,h}$ is true in u, for $h \in X = \{j_1, \dots, j_r\} \subseteq \{1, \dots, m_f\}$. Consequently, the flag $Dyn^u_{f,h}$ is set true for $h \in X$ and false otherwise. Since σ is a solution of $\mathit{Eff}(a, u)$, v satisfies the constraint $C^v_{f,j}$ for all $j \in X$. This implies that (14) is satisfied for each $j \in \{1, \dots, m_f\}$. Since some flags $Dyn^u_{f,i}$ are true constraint (15) is satisfied too.

(⇐) Assume that v satisfies the constraint $C^{u,a}_{\mathscr{F}}$. By (12), because $A_a = 1$, some of the constraints $\delta^u_{a,h}$ is satisfied. Hence, action a is executable in u. By the satisfaction of (13) and (14), v satisfies all constraints $C^v_{f,j}$ for which the corresponding $\alpha^u_{f,j}$ is satisfied. Then, v is a solution for $\mathit{Eff}(a, u)$. Consequently, since $v = \mathsf{ine}(v, u)$ (by definition, since v is complete), $\langle u, a, v \rangle$ is a valid transition.

□

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

Let $\langle \mathscr{D}, \mathscr{O} \rangle$ be an instance of a planning problem where $\mathscr{D}$ is an action description and $\mathscr{O}$ contains any number of axioms of the form $\texttt{initially}(C)$ and $\texttt{goal}(C)$. We can state the following.

Theorem 7
There is a valid trajectory $\langle v_0, a_1, v_1, a_2, \ldots, a_{\mathbf{N}}, v_{\mathbf{N}} \rangle$ if and only if there is a solution for the constraint

$$\bigwedge_{\texttt{initially}(C) \in \mathscr{O}} C^{v_0} \wedge \bigwedge_{j=0}^{\mathbf{N}-1} \left(C_{\mathscr{F}}^{v_j, v_{j+1}} \right) \wedge \bigwedge_{\texttt{goal}(C) \in \mathscr{O}} C^{v_{\mathbf{N}}}.$$

Proof
The result follows from (repeated) applications of Theorem 6. □

6.1.3 Adding static causal laws

In this section we consider the case of action theories $\langle \mathscr{DL}, \mathscr{EL}, \mathscr{SL} \rangle$ of $\mathscr{B}_0^{MV}$, involving static causal laws (i.e., such that $\mathscr{SL} \neq \emptyset$).

The presence of static laws requires refining the semantics of the language, in order to ensure proper treatment of inertia in the construction of a valid trajectory.

We start by defining three operations $\cap, \cup$, and $\triangle$ on states, as follows:

$$v_1 \cup v_2(f) = \begin{cases} v_1(f) & \textit{if } \; v_1(f) = v_2(f), \\ v_1(f) & \textit{if } \; v_2(f) = \bot, \\ v_2(f) & \textit{if } \; v_1(f) = \bot, \\ \bot & \textit{otherwise}, \end{cases}$$

$$v_1 \cap v_2(f) = \begin{cases} v_1(f) & \textit{if } \; v_1(f) = v_2(f), \\ \bot & \textit{otherwise}, \end{cases}$$

$$\triangle(v_1, v_2, S)(f) = \begin{cases} v_1(f) & \textit{if } \; f \in S, \\ v_2(f) & \textit{otherwise}, \end{cases}$$

where the set S used in $\triangle$ is a set of fluents. Observe that $\mathsf{ine}(\sigma, v) = \triangle(\sigma, v, \text{dom}(\sigma))$.

A state v is *closed* w.r.t. a set of static causal laws

$$\mathscr{SL} = \{\texttt{caused}(C_1, D_1), \ldots, \texttt{caused}(C_k, D_k)\},$$

if $v \models (C_1 \to D_1) \wedge \cdots \wedge (C_k \to D_k)$. We denote this property as $v \models \mathscr{SL}$.

Given two states v, v', a set of fluents D, and a set $\mathscr{SL}$ of static causal laws, we say that v' is *minimally closed* w.r.t. v, D, and $\mathscr{SL}$ if

- $v' \models \mathscr{SL}$ (i.e., v' is closed) and
- for all $S \subseteq D$, if $\triangle(v, v', S) \neq v'$ then $\triangle(v, v', S) \not\models \mathscr{SL}$.

The notion of minimally closed state is intended to capture the law of inertia, w.r.t. a given set D of fluents. Notice, in fact, that $\triangle(v, v', \emptyset) = v'$. Intuitively speaking, v' is minimally closed when it is obtainable from v by applying a minimal set of (necessary) changes in the values of the "inertial" fluents (those in D). In other words, it is not possible to obtain from v a state different from v' and closed w.r.t.

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

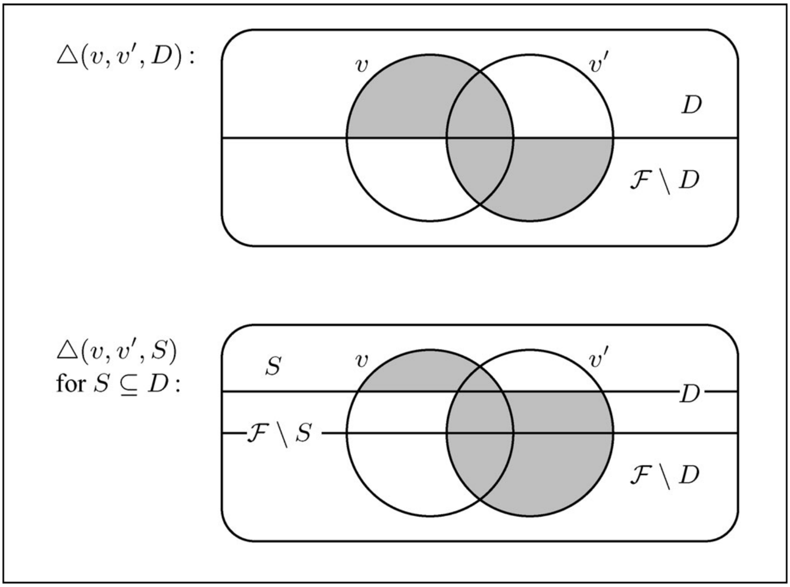


Fig. 9. The set $\triangle(v, v', X)$ is obtained by combining a portion of v and a portion of v', depending on the third argument X, which acts as a regulator in "mixing" portions of v and v'. The figure visualizes, in gray, the two sets $\triangle(v, v', D)$ (above) and $\triangle(v, v', S)$ (below) for $S \subseteq D \subseteq \mathscr{F}$ and illustrates the definition of minimal closure. A state v' is minimally closed if and only if $v' \models \mathscr{SL}$ and for all $S \subseteq D$, if $\triangle(v, v', D) \neq v'$ then $\triangle(v, v', S) \not\models \mathscr{SL}$. In both cases, the surrounding frame represents the set $\mathscr{F}$ of all fluents.

$\mathscr{SL}$, by applying "fewer changes" than those involved in obtaining v'. A pictorial representation of $\triangle(v, v', X)$ is shown in Figure 9.

Observe that if $\mathscr{SL} = \emptyset$ then v' is minimally closed w.r.t. v, D, and $\mathscr{SL}$ if and only if $v = v'$.

Example 10
Let f, g, h be fluents with $\text{dom}(f) = \text{dom}(g) = \text{dom}(h) = \{0, 1\}$ and

$$\mathscr{SL} = \{\texttt{caused}(f = 1, g = 1), \texttt{caused}(f = 0, g = 0)\}.$$

Consider the states $v = \{f/0, g/0, h/0\}$, $v' = \{f/1, g/1, h/1\}$, $v'' = \{f/0, g/0, h/1\}$ and let $D = \{f, g\}$. Then, v' and v'' are both closed w.r.t. $\mathscr{SL}$.

However, v'' is minimally closed w.r.t. v, D, and $\mathscr{SL}$, while v' is not minimally closed since $\triangle(v, v', D) = \{h/1, f/0, g/0\}$ is different from v' and closed. □

A triplet $\langle v, a, v' \rangle$, where v and v' are complete states and a is an action, is a *valid transition* if

(1) the action a is executable in v and
(2) we have that $v' = \text{ine}(\sigma, v')$ where,
  - σ is a solution of the constraint $\mathit{Eff}(a, v)$, and
  - v' is minimally closed w.r.t. v, $\mathscr{F} \setminus \text{dom}(\sigma)$, and $\mathscr{SL}$.

Intuitively, the conditions that define a transition are designed to guarantee that

- a solution σ for the constraints describing the effects of the action is determined;

- such solution is part of the new state v' constructed (thanks to $v' = \mathsf{ine}(\sigma, v')$); and
- the new state is minimally closed with respect to all the fluents not affected by the execution of the action.

Let us observe that, since all fluents in the domain of any solution σ of $\mathit{Eff}(a, v)$ maintain the same value in v', it holds that $v' \models \mathit{Eff}(a, v)$.

Notice that the notion of a valid transition given in presence of static laws properly extends the one given in Section 6.1.1. In fact, the following property holds:

Lemma 1
If $\mathscr{SL} = \emptyset$ then $\mathsf{ine}(\sigma, v) = \mathsf{ine}(\sigma, v')$.

Proof
It is sufficient to note that, if $\mathscr{SL} = \emptyset$ then v' is minimally closed w.r.t. $\mathscr{F} \setminus \mathit{dom}(\sigma)$ if and only if $\mathsf{ine}(\sigma, v) = v'$. □

Example 11
Let us extend the action description of Example 10. We consider the following domain description:

$$
\begin{array}{ll}
\texttt{fluent}(f, \{0,1\}). & \texttt{fluent}(g, \{0,1\}). \\
\texttt{fluent}(h, \{0,1\}). & \\
\texttt{action}(a). & \texttt{executable}(a, h = 0). \\
\texttt{causes}(a, h = 1) & \\
\texttt{caused}(f = 1, g = 1). & \texttt{caused}(f = 0, g = 0).
\end{array}
$$

Let us consider the three states $v = \{f/0, g/0, h/0\}$, $v' = \{f/1, g/1, h/1\}$, and $v'' = \{f/0, g/0, h/1\}$. Then $\langle v, a, v'' \rangle$ is a valid transition, while $\langle v, a, v' \rangle$ is not. □

Let $\langle \mathscr{D}, \mathscr{O} \rangle$ be a planning problem instance. Let $\vec{v} = \langle v_0, \ldots, v_{\mathbf{N}} \rangle$ be a sequence of complete states and let $a_1, \ldots, a_{\mathbf{N}}$ be actions. Then $\langle v_0, a_1, v_1, \ldots, a_{\mathbf{N}}, v_{\mathbf{N}} \rangle$ is a *valid trajectory* if the following conditions hold:

- $v_0 \models \mathscr{SL}$, and for each axiom $\texttt{initially}(C)$ in $\mathscr{O}$, we have that $v_0 \models C$;
- for each axiom of the form $\texttt{goal}(C)$ in $\mathscr{O}$, we have that $v_{\mathbf{N}} \models C$;
- $\langle v_i, a_{i+1}, v_{i+1} \rangle$ is a valid transition, for each $i \in \{0, \ldots, \mathbf{N} - 1\}$.

6.1.4 Abstract implementation in presence of static laws

Let us consider a fluent f and a transition from state u to state w, due to an action a, and let us adopt the same notation (F^u_f, C^u, A^u_a, etc.) introduced in Section 6.1.2. The state transition from u to w can be seen as the composition of two steps involving an intermediate state v. The first of these steps reflects the effects of the dynamic laws, whereas the second step realizes the closure w.r.t. the static causal laws. Hence we proceed by introducing a set of variables corresponding to the intermediate state $v = \mathsf{ine}(\sigma, u)$, where σ is a solution of $\mathit{Eff}(a, u)$. The constraint-based description of the first step is essentially the same we described in Section 6.1.2 – thus, we only need to extend the constraint system defined in Figure 8 to reflect the second part of the transition.

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

$$F_f^w \in \mathrm{dom}(f), \tag{16}$$

$$Dyn_{f,j}^u \rightarrow F_f^w = F_f^v, \quad \forall j \in \{1, \ldots, m_f\}, \tag{17}$$

$$G_{f,j}^w \rightarrow D_{f,j}^w, \quad \forall j \in \{1, \ldots, h_f\}, \tag{18}$$

$$Stat_f^v \leftrightarrow \bigwedge_{g \in L_f} \left(\left(\bigvee_{j=1}^{m_f} \left(Dyn_{f,j}^u \wedge g \in \mathsf{fluents}(C_{f,j}) \right) \right) \rightarrow F_g^v = F_g^u \right), \tag{19}$$

$$Stat_f^v \rightarrow \bigwedge_{g \in L_f} F_g^w = F_g^v. \tag{20}$$

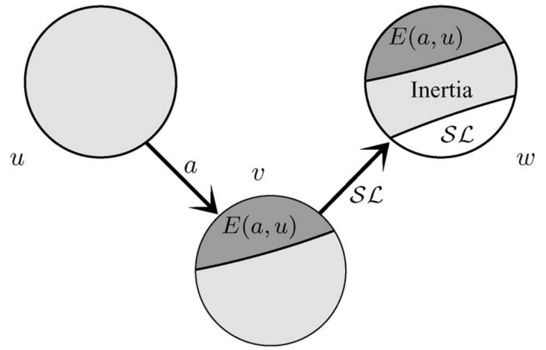


Fig. 10. The constraints for a state transition from u to w (with intermediate state v), for a fluent f.

Given a set $L \subseteq \mathscr{F}$ of fluents, let $\mathscr{SL}_L \subseteq \mathscr{SL}$ be the collection of all static causal laws in which at least one fluent of L occurs. Moreover, for simplicity, let $\mathscr{SL}_f$ denote $\mathscr{SL}_{\{f\}}$, i.e., the set of all static causal laws that involve the fluent f.

Let us define a relation $R \subseteq \mathscr{F} \times \mathscr{F}$ so that $f_1 R f_2$ if and only if $\mathscr{SL}_{f_1} \cap \mathscr{SL}_{f_2} \neq \emptyset$. R is an equivalence relation and it partitions $\mathscr{F}$. Each element (i.e., equivalence class) of the quotient $\mathscr{F}/R$ is said to be a *cluster* (w.r.t. $\mathscr{SL}$). Notice that a cluster can be a singleton $\{f\}$. Let f be a fluent, we denote with L_f its cluster w.r.t. $\mathscr{SL}$.

Example 12
Assume that $\mathscr{SL}$ consists of the rules

$$\texttt{caused}(\texttt{true}, f = 1), \quad \texttt{caused}(g = 2, h = 3), \quad \texttt{caused}(h < 5, r = 2).$$

Then the two clusters are $\{f\}$ and $\{g, h, r\}$. $\square$

Given a fluent f, let us consider the sets of dynamic and executability laws $\mathscr{DL}_f$ and $\mathscr{EL}_a$, as defined in Section 6.1.2. Moreover, let us consider the cluster containing f, let it be $L_f = \{f_1, \ldots, f_k\}$, and the corresponding set of static causal laws $\mathscr{SL}_{L_f}$:

$$\mathscr{SL}_{L_f} = \{\texttt{caused}(G_{f,1}, D_{f,1}), \ldots, \ \texttt{caused}(G_{f,h_f}, D_{f,h_f})\}.$$

Figure 10 describes the constraints (to be added to those in Fig. 8) that are used in encoding the relations that determine the value of the fluent f in state w (represented

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

through the variable F_f^w) after the execution of action a in the state u (we recall that v is to be considered as an intermediate state $v = \mathsf{ine}(\sigma, u)$).

The constraint (16) sets the domains for the variables F_f^w. The constraint (17) propagates to w the effects of the dynamic laws. Constraint (18) imposes closure w.r.t. the static causal laws. Finally, constraints (19)–(20) require that if all the fluents in dom(σ) that belong to the cluster L_f are left unchanged in the transition, then all the fluents of L_f should not change their values. More precisely, as far as (19) is concerned, $Stat_f^v$ is set to true if, for all fluents g in L_f, either g is not affected by the dynamic laws (i.e., $F_g^v = F_g^u$), or for each activated dynamic law $\mathtt{causes}(a_{i_{f,j}}, C_{f,j}, \alpha_{f,j})$ (i.e., such that its precondition $\alpha_{f,j}^u$ is true), g does not occur in its effects (i.e., in $C_{f,j}$). Notice that, with respect to a specific state transition, we are not considering subject to inertia all those fluents that occur in the effects of (at least) one activated dynamic law.

The enforcement of the constraint (20) constitutes a necessary, but not sufficient, condition for the target state to be minimally closed. We will discuss later on this point.

Let us denote with $C_f^{u,w}$ the conjunction of the constraints (11)–(18) for all actions $a \in \mathscr{A}$. Given an action domain specification over the signature $\langle \mathscr{V}, \mathscr{F}, \mathscr{A} \rangle$, the system of constraints $C_{\mathscr{F}}^{u,w}$ includes:

- the constraint $C_f^{u,w}$ for each fluent literal f in the language of $\mathscr{F}$;
- the constraint $\sum_{a \in \mathscr{A}} A_a^u = 1$.

Similarly, let $Stat_{\mathscr{F}}^{u,w}$ denote the conjunction of all the constraints of the forms (19) and (20).

The next theorem states completeness of the encoding described so far. Again, given two states u, w and an action a, let $C_{\mathscr{F}}^{u,a}$ and $Stat_{\mathscr{F}}^{u,a}$ denote the constraints obtained from $C_{\mathscr{F}}^{u,w}$ and $Stat_{\mathscr{F}}^{u,w}$, respectively, by setting $A_a = 1$, $A_b = 0$ for all $b \neq a$, and $F_f^u = u(f)$ for each fluent literal f.

Theorem 8

Let $\mathscr{D} = \langle \mathscr{DL}, \mathscr{EL}, \mathscr{SL} \rangle$ and let u, w two states and a an action. Then, if $\langle u, a, w \rangle$ is a valid transition in the semantics of the language $\mathscr{B}_0^{MV}$, then w represents a solution of the constraint $C_{\mathscr{F}}^{u,a} \wedge Stat_{\mathscr{F}}^{u,a}$.

Proof

For the constraints (11)–(16), considering the transition from u to v, the proof proceeds analogously to the first part of the proof of Theorem 6.

Let us sketch the part of the proof regarding the effect of the static causal laws. Since $\langle u, a, w \rangle$ is a valid transition, $w = \mathsf{ine}(\sigma, w)$, w agrees with $v = \mathsf{ine}(\sigma, u)$ on all fluents in dom(σ), hence (17) hold. Moreover, w is closed w.r.t. $\mathscr{SL}$, hence it satisfies (18). From the fact that w is minimally closed w.r.t. $\mathsf{ine}(\sigma, u), \mathscr{F} \setminus \mathrm{dom}(\sigma)$, and $\mathscr{SL}$, it follows that w satisfies (19)–(20). □

The above encoding does not guarantee soundness. This is because the constraints (17)–(18) in Figures 8 and 10 might admit solutions not corresponding to minimally closed states.

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

We introduced the notion of cluster to partially recover the soundness of the encoding. Intuitively speaking, a cluster generalizes, to the multivalued case, the notion of loop seen in Section 3.2: a cluster is a set of fluents whose values have been declared to be mutually dependent through a set of static causal laws. In a state transition, similarly to the case of loops, changes to the fluents of a cluster might occur because of their mutual influence, not being (indirectly) caused by dynamic laws.

Constraints (19) and (20) impose inertia on all the fluents of a cluster whenever none of them is influenced by dynamic laws. However, note that imposing (19)–(20) does not completely circumvent the problem because state transitions violating the inertia are still admitted. In fact, (19)–(20) do not impose inertia on the fluents of a cluster when at least one of them is changed by the dynamic laws. This might lead to invalid transitions, in which a change in the value of a fluent of a cluster happens even if this is not necessary in order to satisfy all the static causal laws.

Nevertheless, we introduced the constraints (19) and (20) because they constitute a good compromise w.r.t. the efficiency of a concrete implementation (as discussed later).

To completely enforce soundness, we need to apply a filter on the solutions that are admitted by the encoding described so far. To this aim, let us introduce a condition on the values of the fluent, which is intended to mimic, in the multivalued setting, the effect of loop formulae.

Let us assume that the action a is executed in the state u, and that σ, v, and w have been determined so that to satisfy the constraint $C^{u,w}_{\mathscr{F}}$. In this situation the following constraint characterizes an hypothetical state x, different from w:

$$Form(\mathscr{D})^{u,a} = \Big(C^{u,x}_{\mathscr{F}} \ \wedge \tag{21}$$

$$\bigwedge_{f \in \mathscr{F}} \Big(\bigvee_{j=1}^{m_f} Dyn^u_{f,j} \rightarrow F^x_f = F^w_f \Big) \ \wedge \tag{22}$$

$$\bigvee_{f \in \mathscr{F}} F^x_f \neq F^w_f \ \wedge \ \bigwedge_{f \in \mathscr{F}} \Big(F^x_f \neq F^w_f \rightarrow F^x_f = F^u_f \Big) \Big). \tag{23}$$

Intuitively, the satisfaction of such a formula witnesses the existence of a counterexample for the minimal closure of w. Notice that, being σ, v, and w already determined, the only fluents/variables to be determined are those describing the state x, if any. The conjunct in line (21) states that x is a target state alternative to w; in particular, it enforces the closure of x w.r.t. $\mathscr{SL}$. The conjunction (22) states that x and w agree on the fluents in dom(σ). Finally, (23) states that x must differ from w and it must agree with u in at least one fluent – that, because of (22), it is in $\mathscr{F} \setminus$ dom(σ).

We can prove the following result that generalizes Theorem 6 to the case of $\mathscr{SL} \neq \emptyset$.

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

Theorem 9

Let $\mathscr{D} = \langle \mathscr{DL}, \mathscr{EL}, \mathscr{SL} \rangle$ and u, w two states, with u closed w.r.t. $\mathscr{SL}$. Let a an action such that w represents a solution of the constraint $C^{u,a}_{\mathscr{F}}$. Then $\langle u, a, w \rangle$ is a valid transition in the semantics of the language $\mathscr{B}^{MV}_0$, if $Form(\mathscr{D})^{u,a}$ is unsatisfiable.

Proof

By proceeding as in the proof of Theorem 6, we can show that all needed conditions for $\langle u, a, w \rangle$ to be a valid transition are satisfied, except for the minimal closure of w.

Let us assume, by contradiction, that w is not minimally closed w.r.t. u, $\mathscr{F} \setminus \mathrm{dom}(\sigma)$, and $\mathscr{SL}$. Then, there exists $S \subseteq \mathscr{F} \setminus \mathrm{dom}(\sigma)$ such that $x = \triangle(u, w, S) \neq w$ and $x \models \mathscr{SL}$ For each fluent $f \notin S$ it holds that $F^x_f = F^w_f$. Moreover, $F^v_f = F^w_f$ holds too, because w satisfies $C^{u,a}_{\mathscr{F}}$. Hence, $Dyn^u_{f,j} \rightarrow F^x_f = F^v_f$ holds for all j.

For each fluent f, since x is closed w.r.t. $\mathscr{SL}$, we have that $G^x_{f,j} \rightarrow D^x_{f,j}$ (for all $j \in \{1, \ldots, h_f\}$). Observe that the conditions of the forms (11)–(16) in the conjunct at line (21) (i.e., in $C^{u,x}_{\mathscr{F}}$) do not depend on the specific x. Then, the conjunct (21) is satisfied.

Let us also observe that condition (22) holds too. This is so because, for all $f \in \mathrm{dom}(\sigma)$ we have that $F^x_f = F^w_f = F^v_f$. From the fact that $x \neq w$ it follows that $\bigvee_{f \in \mathscr{F}} F^x_f \neq F^w_f$ holds. Finally, the condition (23) is satisfied because, whenever $F^x_f \neq F^w_f$ holds, by the definition of $\triangle$, it must be the case that $F^x_f = F^u_f$. It follows that $Form(\mathscr{D})^{u,a}$ is satisfiable (by x).

This is a contradiction and proves that w is minimally closed w.r.t. u, $\mathscr{F} \setminus \mathrm{dom}(\sigma)$, and $\mathscr{SL}$, and that $\langle u, a, w \rangle$ is a valid transition. □

Let $\langle \mathscr{D}, \mathscr{O} \rangle$ be an instance of a planning problem, where $\mathscr{D}$ is a domain description and $\mathscr{O}$ contains any number of axioms of the form `initially`(C) and `goal`(C). We conclude this section by stating a generalization of Theorem 7 to the case of $\mathscr{SL} \neq \emptyset$.

Theorem 10

There is a valid trajectory $\langle v_0, a_1, v_1, a_2, \ldots, a_{\mathbf{N}}, v_{\mathbf{N}} \rangle$ if and only if

- $v_0 \models \mathscr{SL}$,
- there is a solution for the constraint

$$\bigwedge_{\texttt{initially}(C) \in \mathscr{O}} C^{v_0} \wedge \bigwedge_{j=0}^{\mathbf{N}-1} C^{v_j, v_{j+1}}_{\mathscr{F}} \wedge \bigwedge_{\texttt{goal}(C) \in \mathscr{O}} C^{v_{\mathbf{N}}},$$

- For each $j \in \{0, \ldots, \mathbf{N}-1\}$ the formula $Form(\mathscr{D})^{v_j, a_{j+1}}$ is unsatisfiable.

Proof

The result follows from Theorems 8 and 9. □

Remark 5 (*Embedding of $\mathscr{B}$ into $\mathscr{B}^{MV}_0$*)

We conclude this section by showing that $\mathscr{B}^{MV}_0$ is at least as expressive as $\mathscr{B}$. To this aim it suffices to describe how to translate a domain description $\mathscr{D}$ of $\mathscr{B}$ to a $\mathscr{B}^{MV}_0$domain description $\mathscr{D}'$, in such a way that the semantics of the domain is preserved. Let us outline the main points of such a translation.

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

Each Boolean fluent f in $\mathscr{D}$ can be modeled in $\mathscr{B}_0^{MV}$ by a multivalued fluent f' whose domain is $\mathscr{V} = \{0,1\} \subseteq \mathbb{Z}$.
Each action in $\mathscr{D}$ uniquely corresponds to an action in $\mathscr{D}'$.
Let us consider a dynamic causal law of $\mathscr{D}$, e.g.,

$$\texttt{causes}(a, f, [f_1, \ldots, f_k, \texttt{neg}(g_1), \ldots, \texttt{neg}(g_h)]).$$

This law is translated in $\mathscr{D}'$ as

$$\texttt{causes}(a, f' = 1, [f'_1 = 1, \ldots, f'_k = 1, g'_1 = 0, \ldots, g'_h = 0]).$$

In a similar manner, static laws and executability conditions of $\mathscr{D}$ are mapped into $\mathscr{B}_0^{MV}$. Consequently, the two domain descriptions $\mathscr{D}$ and $\mathscr{D}'$ describe two isomorphic transition systems.

6.2 Adding annotated fluents and non-Markovian references

In this section, we generalize the treatment described in Section 6.1 in order to provide a state-transition semantics for $\mathscr{B}^{MV}$ suitable to cope with temporal references. The first form of temporal references involves annotated fluents and concerns relative access to their past values, w.r.t. the current state. There is no restriction on the occurrences of this kind of annotated fluents: they might be used in all laws of a domain description. In this case, the extension of the semantics described in Section 6.1 comes rather naturally. Since references may relate different points in time along the plan, the approach consists of considering sequences of states instead of pairs of states, to define the transition constraints.

Regarding references to future points in time (i.e., positively annotated fluents), we recall that they are admitted in the consequences of dynamic causal laws only. This restriction allows the treatment of future and past references by exploiting the very same mechanisms. The semantics is further enriched in Section 6.4 to encompass state constraints specified by using absolute time references, as well as costs.

Let $\vec{v} = \langle v_0, \ldots, v_{\mathbf{N}} \rangle$ be a state sequence. Given $\vec{v}$, and $i \in \{0, \ldots, \mathbf{N}\}$, we define the concept of *value* of φ in $\vec{v}$ at time i (with abuse of notation, denoted by $\vec{v}(i, \varphi)$) as follows[9]:

$$\begin{array}{l}
\vec{v}(i,x) = x \quad \text{if } x \in \mathscr{V}, \\
\vec{v}(i,f^j) = v_{i+j}(f) \quad \text{if } f \in \mathscr{F}, \text{ and } 0 \leqslant i+j \leqslant \mathbf{N}, \\
\vec{v}(i,f^j) = v_0(f) \quad \text{if } f \in \mathscr{F} \text{ and } i+j < 0, \\
\vec{v}(i,f^j) = v_{\mathbf{N}}(f) \quad \text{if } f \in \mathscr{F} \text{ and } i+j > \mathbf{N}, \\
\vec{v}(i, abs(\varphi)) = |\vec{v}(i,\varphi)|, \\
\vec{v}(i, -(\varphi)) = -(\vec{v}(i,\varphi)), \\
\vec{v}(i, \varphi_1 \oplus \varphi_2) = \vec{v}(i,\varphi_1) \oplus \vec{v}(i,\varphi_2), \\
\vec{v}(i, \texttt{rei}(C)) = 1 \quad \text{if } \vec{v} \models_i C, \\
\vec{v}(i, \texttt{rei}(C)) = 0 \quad \text{if } \vec{v} \not\models_i C,
\end{array}$$

where $n \in \mathscr{V}$, $\oplus \in \{+, -, *, /, \texttt{mod}\}$.

[9] A slightly simplified treatment could be described if only past references are admitted. In this case, we consider i to be the current point in time and j to be negative. The notation could then be simplified by considering just a prefix $\vec{v} = \langle v_0, \ldots, v_i \rangle$ of the state sequence.

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

As for (10) of Section 6.1, the semantics of reified constraints relies on the notion of satisfaction, which in turn has to be contextualized to a specific point in time i. More formally, given a fluent constraint $\varphi_1 \ \mathsf{op} \ \varphi_2$ and a state sequence $\vec{v}$, the notion of satisfaction at time i is defined as $\vec{v} \models_i \varphi_1 \ \mathsf{op} \ \varphi_2 \Leftrightarrow \vec{v}(i, \varphi_1) \ \mathsf{op} \ \vec{v}(i, \varphi_2)$. The notion $\models_i$ is generalized to the case of propositional combinations of fluent constraints in the usual manner.

Given a constraint C, let $\geqslant$-$\mathsf{fluents}(C)$ be the set of annotated fluents f^i, for $i \geqslant 0$, occurring in C. Given a state sequence $\vec{v} = \langle v_0, \dots, v_i \rangle$, with $0 \leqslant i < \mathbf{N}$, a function $\sigma : \geqslant$-$\mathsf{fluents}(C) \longrightarrow \mathscr{V}$ is an *i-solution* of C w.r.t. $\vec{v}$, if it holds that

$$\langle v_0, \dots, v_i, \mathsf{ine}(\sigma|_0, v_i), \overline{(\sigma|_1)}, \dots, \overline{(\sigma|_{\mathbf{N}-(i+1)})} \rangle \models_{i+1} C,$$

where each $\sigma|_k$ (for $k \geqslant 0$) is the restriction of the assignment σ to the fluent annotated with k, and $\overline{\mu}$ denotes the substitution obtained by completing μ, with assignment to $\perp$ for all fluents not in $\mathrm{dom}(\mu)$. Note that we treat the interpretation of the various operations as strict w.r.t. $\perp$ and we assume satisfied all constraints that refer to undefined expressions. Hence, for instance, if C is constraint and there is a subexpression ψ of C evaluated as $\perp$, then we assume $\vec{v} \models_i C$.

Example 13
Let $\mathbf{N} = 3$ and $i = 1$. Consider the constraint $C \equiv (g^0 = f^{-1} + f^{-2})$ and let $\vec{v} = \langle v_0, v_1 \rangle = \langle \{f/2, g/1\}, \{f/1, g/2\} \rangle$.

Then $\sigma = \{g/3\} = \sigma|_0$ is a 1-solution of the constraint C, since

- $\mathsf{ine}(\sigma|_0, \{f/1, g/2\}) = \mathsf{ine}(\{g/3\}, \{f/1, g/2\}) = \{f/1, g/3\}$, and
- $\langle \{f/2, g/1\}, \{f/1, g/2\}, \{f/1, g/3\}, \{f/\perp, g/\perp\}, \rangle \models_2 g^0 = f^{-1} + f^{-2}$, in fact, we have that $\vec{v}(2, C)$ is $\vec{v}(2, g^0) = \vec{v}(2, f^{-1} + f^{-2})$, which is equivalent to $v_2(g^0) = v_1(f) + v_0(f)$. □

A state sequence $\vec{v} = \langle v_0, \dots, v_h \rangle$ is *closed* w.r.t. a set of static causal laws

$$\mathscr{SL} = \{\mathtt{caused}(C_1, D_1), \dots, \mathtt{caused}(C_k, D_k)\}$$

if for all $i \in \{0, \dots, h\}$ it holds that $\vec{v} \models_i (C_1 \rightarrow D_1) \wedge \cdots \wedge (C_k \rightarrow D_k)$.

We also generalize the notion of minimal closure as follows: given a state sequence $\vec{v} = \langle v_0, \dots, v_i \rangle$ and a state v' we say that v' is minimally closed w.r.t. $\vec{v}$, D, and $\mathscr{SL}$ if

- $\langle v_0, \dots, v_i, v' \rangle$ is closed w.r.t. $\mathscr{SL}$,
- for all sets of fluents $S \subseteq D$, if the state $\Delta(v_i, v', S)$ is different from v', then $\langle v_0, \dots, v_i, \Delta(v_i, v', S) \rangle$ is not closed w.r.t. $\mathscr{SL}$.

The action a is *executable* in $\vec{v}$ at time i if there is an axiom $\mathtt{executable}(a, C)$ such that $\vec{v} \models_i C$.

Let us denote with $Dyn(a)$ the set of dynamic causal laws for an action a. The *effects* of executing a in $\vec{v}$ at time i, denoted by $\mathit{Eff}(a, \vec{v}, i)$, is

$$\mathit{Eff}(a, \vec{v}, i) = \bigwedge \{PC \mid \mathtt{causes}(a, PC, C) \in Dyn(a), \vec{v} \models_i C\}.$$

Given a constraint C, we denote by $\mathit{shift}^t(C)$ the constraint obtained from C by replacing each fluent f^x with f^{x-t}.

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

Let us assume that $\vec{v} = \langle v_0, \ldots, v_i \rangle$ is a sequence of complete states and that $\vec{a}$ is a sequence of actions $\langle a_1, \ldots, a_{i+1} \rangle$. The effects of the sequence of actions in $\vec{v}$ is represented by the formula

$$E(i, \vec{a}, \vec{v}) = \bigwedge_{j=0}^{i} shift^{j-i}(\mathit{Eff}(a_{j+1}, \vec{v}, j)) \wedge \bigwedge_{j=0}^{i} \bigwedge_{f \in \mathscr{F}} f^{j-i} = v_j(f).$$

Let us observe that this constraint might involve all fluents of the states $v_0, \ldots, v_i$, as well as fluents of future states. The values of fluents in states $v_0, \ldots, v_i$ are fixed by $\vec{v}$.

Let $\langle \mathscr{D}, \mathscr{O} \rangle$ be a planning problem instance, $\vec{v} = \langle v_0, \ldots, v_{\mathbf{N}} \rangle$ be a sequence of complete states and $a_1, \ldots, a_{\mathbf{N}}$ be actions. Then, $\langle v_0, a_1, v_1, \ldots, a_{\mathbf{N}}, v_{\mathbf{N}} \rangle$ is a *valid trajectory* if the following conditions hold:

- $\langle v_0, \ldots, v_{\mathbf{N}} \rangle$ is closed w.r.t. $\mathscr{SL}$,
- for each axiom of the form `initial`(C) in $\mathscr{O}$, we have that $\vec{v} \models_0 C$,
- for each axiom of the form `goal`(C) in $\mathscr{O}$, we have that $\vec{v} \models_{\mathbf{N}} C$,
- for each $i \in \{0, \ldots, \mathbf{N} - 1\}$ the following conditions hold
  - — action a_{i+1} is executable in $\vec{v}$ at time i and
  - — we have that $v_{i+1} = \mathsf{ine}(\sigma|_0, v_{i+1})$ where
    - – σ is a i-solution of the constraint $E(i, \langle a_1, \ldots, a_{\mathbf{N}} \rangle, \langle v_0, \ldots, v_{\mathbf{N}-1} \rangle)$ w.r.t. $\langle v_0, \ldots, v_i \rangle$,
    - – v_{i+1} is minimally closed w.r.t. $\langle v_0, \ldots, v_i \rangle$, $\mathscr{F} \setminus \mathrm{dom}(\sigma)$, and $\mathscr{SL}$.

Example 14

Let us consider the following domain specification and planning problem instance (for $\mathbf{N} = 2$):

`fluent`$(f, 1, 5)$.
`fluent`$(g, 1, 5)$.
`fluent`$(h, 1, 5)$.
`action`(a).
`action`(b).
`executable`$(a,$ `true`$)$.
`executable`$(b,$ `true`$)$.
`causes`$(a, g^0 = g^{-1} + 2,$ `true`$)$.
`causes`$(b, f^0 = g^{-1} + h^{-2},$ `true`$)$.
`initially`$(f = 1)$.
`initially`$(g = 1)$.
`initially`$(h < 3)$.
`goal`$(f > 4)$.

Observe that the only valid trajectory is

$$\langle \{f/1, g/1, h/2\}, a, \{f/1, g/3, h/2\}, b, \{f/5, g/3, h/2\} \rangle.$$

The validity can be verified by observing that

- $\{f/1, g/1, h/2\}$ satisfies all the constraints provided in the `initial` declarations;

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

- $\{f/5, g/3, h/2\}$ satisfies the goal constraint $f > 4$;
- the action a is executable in $\langle v_0 \rangle = \langle \{f/1, g/1, h/2\} \rangle$ and action b is executable in

$$\langle v_0, v_1 \rangle = \langle \{f/1, g/1, h/2\}, \{f/1, g/3, h/2\} \rangle$$

(since both their executability laws and the action conditions are trivially true).
- Consider the first state transition and $i = 0$ and note that $\geqslant$-fluents$(g^0 = g^{-1} + 2) = \{g\}$. Then, $\sigma' = \{g/3\}$ is a 0-solution of $g^0 = g^{-1} + 2$ w.r.t. $\langle \{f/1, g/1, h/2\} \rangle$. In fact, $\sigma'|_0 = \sigma'$, $\sigma'|_1 = \{\}$, and
  — $v_1 = \mathsf{ine}(\sigma'|_0, v_0) = \mathsf{ine}(\{g/3\}, \{f/1, g/1, h/2\}) = \{f/1, g/3, h/2\}$,
  — $\langle v_0, v_1, \overline{\sigma'|_1} \rangle = \langle v_0, v_1, \{f/\bot, g/\bot, h/\bot\} \rangle \models_1 g^0 = g^{-1} + 2$,
  — v_1 is minimally closed w.r.t. $\langle \{f/1, g/1, h/2\} \rangle$, $\{f, h\}$ and $\emptyset$.
- Consider the second state transition and $i = 1$ and note that $\geqslant$-fluents$(f^0 = g^{-1} + h^{-2}) = \{f\}$. Then, $\sigma'' = \{f/5\}$ is a 1-solution of $f^0 = g^{-1} + h^{-2}$ w.r.t. $\langle v_0, v_1 \rangle$. In fact, $\sigma''|_0 = \sigma''$, and
  — $v_2 = \mathsf{ine}(\sigma''|_0, v_1) = \mathsf{ine}(\{f/5\}, \{f/1, g/3, h/2\}) = \{f/5, g/3, h/2\}$,
  — $\langle v_0, v_1, v_2 \rangle \models_2 f^0 = g^{-1} + h^{-2}$,
  — v_2 is minimally closed w.r.t. $\langle v_0, v_1 \rangle$, $\{g, h\}$ and $\emptyset$. □

6.3 Abstract implementation of $\mathscr{B}^{MV}$

The constraint encoding for $\mathscr{B}^{MV}$ is similar to the one developed earlier for the case of $\mathscr{B}_0^{MV}$ (cf., Figs. 8 and 10). In the encoding of a trajectory $\langle v_0, a_1, v_1, \dots, a_{\mathbf{N}}, v_{\mathbf{N}} \rangle$ in $\mathscr{B}_0^{MV}$, we introduced a variable $F_f^{v_i}$ to represent the value of the fluent f in the ith state v_i. In each state transition, say from v_i to v_{i+1}, the implementation of $\mathscr{B}_0^{MV}$ imposes only constraints involving the variables/fluents of the current state. In the language encompassing timed references, each constraint occurring in the action description can address the values that fluents assume in any of the states of the sequence $\vec{v} = \langle v_0, \dots, v_{\mathbf{N}} \rangle$. Since all the variables representing these values are present in the encoding, only the following change is needed to adapt to $\mathscr{B}^{MV}$ the implementation designed for $\mathscr{B}_0^{MV}$: to obtain from a constraint C (involving fluents), a constraint $C^{\vec{v},i}$ (involving the corresponding variables), at time i, we replace each f^j with the variable $F_f^{v_{i+j}}$.

By adopting this refined construction for $C^{\vec{v},i}$, we can inherit all the results of Section 6.1.4. In particular, for an action description $\mathscr{D}$, similarly to what done in Section 6.1.4, we denote by $C_{\mathscr{F}}^{\vec{v},a_i}$ and by $Form(\mathscr{D})^{\vec{v},a_i}$ the constraints homologous to $C_{\mathscr{F}}^{v_{i-1},v_i}$ and $Form(\mathscr{D})^{v_{i-1},a_i}$, respectively.

The completeness result for $\mathscr{B}^{MV}$ directly generalizes that obtained for $\mathscr{B}_0^{MV}$. With regards to soundness, the observation made w.r.t. $\mathscr{B}_0^{MV}$ in Section 6.1.4 still applies. In fact, let $\langle \mathscr{D}, \mathscr{O} \rangle$ be an instance of a planning problem where $\mathscr{D}$ is a domain description and $\mathscr{O}$ contains axioms of the form `initially`(C) and `goal`(C). We state the following:

Theorem 11
There is a valid trajectory $\vec{v} = \langle v_0, a_1, v_1, \dots, v_{\mathbf{N}}, v_{\mathbf{N}} \rangle$ if and only if

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

- $\vec{v}$ is closed w.r.t. $\mathscr{SL}$,
- There is a solution for the constraint

$$\bigwedge_{\texttt{initially}(C)\in\mathscr{C}} C^{\vec{v},0} \wedge \bigwedge_{j=0}^{\mathbf{N}-1} C_{\mathscr{F}}^{\vec{v},a_{j+1}} \wedge \bigwedge_{\texttt{goal}(C)\in\mathscr{C}} C^{\vec{v},\mathbf{N}},$$

- For each $j \in \{0,\ldots,\mathbf{N}-1\}$ the formula $\mathit{Form}(\mathscr{D})^{\vec{v},a_{j+1}}$ is unsatisfiable.

6.4 Adding costs and global constraints

Cost and time constraints can be introduced by filtering the solutions characterized by Theorem 11, in order to rule out the unsatisfactory solutions. More precisely, given a trajectory $\langle v_0, a_1, v_1, \ldots, a_{\mathbf{N}}, v_{\mathbf{N}}\rangle$ satisfying the requirements of Theorem 11, we say that the trajectory satisfies a set of global constraints as described in Sections 5.1 and 5.2 if all the constraints described next hold.

Let us start by investigating the cost constraints. Let

$$\texttt{action\_cost}(a_1, FE_1), \ldots, \texttt{action\_cost}(a_{\mathbf{N}}, FE_{\mathbf{N}})$$

and $\texttt{state\_cost}(FE')$ be specified in the action description[10].

Let us recall that the general form of cost constraints is $\texttt{cost\_constraint}(C)$, where C is a constraint defined as in Section 5, with the added ability to refer to the atoms `plan`, `goal`, and $\texttt{state}(i)$ wherever fluents can be used. Consequently, we extend our definition of value of an expression φ in $\vec{v} = \langle v_0, \ldots, v_{\mathbf{N}}\rangle$ at time i (for all j):

$$\begin{aligned}
\vec{v}(j, \texttt{plan}) &= v_0(FE_1) + \cdots + v_{\mathbf{N}-1}(FE_{\mathbf{N}}),\\
\vec{v}(j, \texttt{goal}) &= v_{\mathbf{N}}(FE'),\\
\vec{v}(j, \texttt{state}(i)) &= v_i(FE') \quad \text{if } 0 \leqslant i \leqslant \mathbf{N}
\end{aligned}$$

(assigning cost constraints to to states outside the plan is senseless. However, for completeness, for $i < 0$ or $i > \mathbf{N}$ we set $\vec{v}(j, \texttt{state}(i)) = 0$ but any other choice – e.g., $\perp$, or the values on states 0 or $\mathbf{N}$ – is reasonable). This modification allows us to derive the notion of satisfaction of a cost constraint C from the notion of satisfaction defined in Section 6.3. As particular cases, we obtain that

- for each assertion $\texttt{cost\_constraint}(\texttt{plan}\ \texttt{op}\ k)$ the plan cost $(v_0(FE_1) + \cdots + v_{\mathbf{N}-1}(FE_{\mathbf{N}}))$ has to satisfy the stated constraint, i.e., it must hold that $(v_0(FE_1) + \cdots + v_{\mathbf{N}-1}(FE_{\mathbf{N}}))\ \texttt{op}\ k$;
- for each assertion $\texttt{cost\_constraint}(\texttt{goal}\ \texttt{op}\ k)$, the cost $v_{\mathbf{N}}(FE')$ of the goal state must satisfy the constraint: $v_{\mathbf{N}}(FE')\ \texttt{op}\ k$;
- for each assertion $\texttt{cost\_constraint}(\texttt{state}(i)\ \texttt{op}\ k)$, the cost $v_i(FE')$ assigned to the i^{th} state has to satisfy the constraint $v_i(FE')\ \texttt{op}\ k$.

The handling of time constraints requires the following modifications:

- for each assertion $\texttt{time\_constraint}(C)$, it holds that $\langle v_0, \ldots, v_{\mathbf{N}}\rangle \models_0 C$, where each timed fluent $f\texttt{@}i$ is evaluated as $v_i(f)$;

[10] As mentioned, if some of these assertion is missing a default cost 1 is assumed.

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

- for each assertion of the form holds(C,i) it holds that $\langle v_0,\ldots,v_{\mathbf{N}}\rangle \models_i C$;
- for each assertion of the form always(C), it holds that $\langle v_0,\ldots,v_{\mathbf{N}}\rangle \models_i C$ for all $i \in \{0,\ldots,\mathbf{N}\}$.

Moreover, if minimize_cost(FE'') is specified, then there exists no other trajectory $\vec{v}'$ such that $\vec{v}'(\mathbf{N}, FE'') < \vec{v}(\mathbf{N}, FE'')$. As particular cases, we have that

- if minimize_cost(plan) is specified, then there exists no other trajectory having a smaller plan cost;
- if minimize_cost(goal) is specified in the action description, then there is no trajectory $\langle v_0', a_1', v_1', \ldots, a_{\mathbf{N}}', v_{\mathbf{N}}'\rangle$, fulfilling all constraints, and such that $v_{\mathbf{N}}'(FE') < v_{\mathbf{N}}(FE')$.

In this manner, we characterize the solutions of a given planning problem to be exactly those solutions described by Theorem 11 that additionally satisfy all the global constraints, the requirements on costs, and the time constraints expressed in the action description. Soundness and completeness properties directly carry over.

7 Concrete implementation of $\mathscr{B}^{MV}$

The overall structure of the concrete implementation of the language $\mathscr{B}^{MV}$ follows that used for implementing the $\mathscr{B}$ language. We focus here on the main differences.

To start, let us briefly describe the code depicted in Figure 11 and show that this concrete implementation reflects the abstract one defined in Figure 8[11]. Hence, we preliminarily ignore lines (65)–(66) of Figure 11.

The first difference w.r.t. the implementation of $\mathscr{B}$ (cf., Section 3) is that each fluent variable is assigned to a finite set domain, drawn from the fluent declaration – instead of being treated as a Boolean variable.

The predicate set_one_fluent (lines (56)–(68)) has a similar role as in the implementation of $\mathscr{B}$. Given the fluent FluentName, the relevant parts of the dynamic causal laws are collected in lines (57)–(59). The predicate zero_subterm is an auxiliary predicate that detects if a constraint involves a fluent – i.e., it looks for an occurrence of FluentName in the constraint imposed by the dynamic causal laws. All the fluents explicitly involved in the consequence of a dynamic law are collected. In line (63), the variable EV identifying the fluent FluentName in the following state ToState is retrieved.

The predicate dynamic (line (64)) collects the list of Boolean flags DynFormula. If one of the variables in Dyn is true then the variable EV is involved in a constraint imposed by a dynamic causal law. In line (67) the disjunction of these flag variables is computed in Formula (let us ignore, for the time being, the variable StatFormula). In line (68) the inertia constraint is added: if Formula is false then the value of the fluent is left unchanged by the transition (i.e., IV = EV). This corresponds to the $\mathsf{ine}(\cdot)$ operator.

[11] Observe that the concrete implementation uses the functors eq, neq, etc. to denote the primitive constraints $=$, $\neq$, etc.

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

```
set_one_fluent(fluent(FluentName,IV), ActionOccs, Now, States) :-
  findall([Act,OP,FE1,FE2,L],
       (causes(Act,FC,L), zero_subterm(FluentName,FC),
       FC =.. [OP,FE1,FE2]), Dyn),
  state_select(Now, States, FromState),
  Next is Now+1,
  state_select(Next, States, ToState),
  member(fluent(FluentName,EV), ToState),
  dynamic(Dyn, ActionOccs, FromState, DynFormula, Next, States),
  cluster_rules(FluentName, Stat), %%% These 2 lines can be dropped in
  static(Stat, States, Next, StatFormula),%%% absence of static laws
  bool_disj(DynFormula, StatFormula, Formula),
  #\ Formula #=> EV #= IV.

dynamic([], _, _, [], _, _).
dynamic([[Act,OP,FE1,FE2,Prec]|Rest],AOccs,State,[Flag|PF1],Now,States) :-
  member(action(Act,VA), AOccs),
  Last is Now-1, %%% Looks for preconditions in FromState and before
  get_precondition_vars(Last, Prec, States, ListPV),
  length(Prec, NPrec),
  sum(ListPV, SumPrec),
  %%% The effect is in the next state (Now=Last+1)
  rel_parsing(FE1, Val1, Now, States),
  rel_parsing(FE2, Val2, Now, States),
  exp_constraint(Val1, OP, Val2, C),
  (VA #/\ (SumPrec #= NPrec)) #<=> Flag,
  Flag #=> C,
  dynamic(Rest, ActionOccs, State, PF1, Now, States).

rel_parsing(Num, Num, _, _) :-
  integer(Num), !.
rel_parsing(rei(RC), Val, Time, States) :-
  RC =.. [OP,E1,E2],
  rel_parsing(E1, Val1, Time, States),
  rel_parsing(E2, Val2, Time, States),
  exp_constraint(Val1, OP, Val2, Val), !.
rel_parsing(abs(FE), Val, Time, States) :- %%% similar for -(FE)
  rel_parsing(FE, Val1, Time, States),
  Val #= abs(Val1), !.
rel_parsing(FE, Val, Time, States) :-
  FE =.. [OP,FE1,FE2],
  member(OP, [+,-,mod,/,*]),
  rel_parsing(FE1, Val1, Time, States),
  rel_parsing(FE2, Val2, Time, States),
  ( OP = + -> Val #= Val1 + Val2;
  OP = - -> Val #= Val1 - Val2;
  OP = * -> Val #= Val1 * Val2;
  OP = / -> Val #= Val1 / Val2;
  OP = mod -> Val #= Val1 mod Val2 ), !.
rel_parsing(Fluent^Delta, Val, Time, States) :-
  H is Time+Delta,
  length(States,N),
  in_interval(H,N,E),
  state_select(E, States, State),
  member(fluent(Fluent,Val),State),!.
rel_parsing(Fluent @ Time, Val, _, States) :-
  state_select(Time,States,State),
  member(fluent(Fluent,Val),State), !.
rel_parsing(Fluent, Val, Time, States) :-
  state_select(Time, States, State),
  member(fluent(Fluent,Val), State).

parsing(Fluent, Val, State) :-
  rel_parsing(Fluent, Val, 0, [State]).

exp_constraint(L, OP, R, C) :-
  (OP == eq -> C #<=> L #= R;
  OP == neq -> C #<=> L #\= R;
  OP == geq -> C #<=> L #>= R;
  OP == leq -> C #<=> L #=< R;
  OP == gt -> C #<=> L #> R;
  OP == lt -> C #<=> L #< R).
```

Fig. 11. Relevant parts of the $\mathcal{B}^{MV}$ implementation.

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

For each action `Act` affecting the value `EV`, the predicate `dynamic` (lines (69)–(82)) retrieves its preconditions and builds the constraint `C` involving `EV` that must be imposed if the preconditions are satisfied. The flag variable `Flag` in line (80) is introduced to keep track of the fact that the action has occurred (i.e., `VA` is true) and the corresponding precondition holds. If `Flag` is true then the constrain `C` is asserted (line (81)). All flags are stored in a list (cf., the variable `DynFormula` in line (64)).

Lines (83)–(114) provide an excerpt of the definition of the predicate `rel_parsing`. This predicate is used to transform fluent expressions to internal expressions involving fluent variables. `States` is a list of states (each of them, in turn is a list of all the fluent variables). The first argument is the fluent expression and the second one is the output internal expression. The argument `Time` represents the specific point in time in which a fluent is referred to (cf., the variable `Now` used in lines (69)–(82) and (124)–(134) to specify the precise point in time in which a fluent expression/constraint has to be evaluated). The predicate `in_interval` called in line (106) sets $\mathtt{E} = \mathtt{H}$ if $0 \leqslant \mathtt{H} \leqslant \mathbf{N}$, $\mathtt{E} = 0$ (resp., $\mathtt{E} = \mathbf{N}$) if $\mathtt{H} < 0$ (resp., $\mathtt{H} > \mathbf{N}$). Similarly, predicate `exp_constraint` (lines (117)–(123)) transforms fluent constraints into the corresponding constraints on the fluent variables.

The above described fragment of implementation is completed with the code needed to handle initial and goal state specifications. Namely, for a specific instance of a planning problem $\langle \mathscr{D}, \mathscr{O} \rangle$, as done for $\mathscr{B}$, all constraint on the initial state (resp., those on the goal state) are reflected by constraining the variables F_f in the representation of the initial (resp., final) state.

We proceed by splitting the correctness proof into steps. We can now state the following result[12].

Theorem 12

The concrete implementation (partially depicted in Fig. 11) is correct and complete w.r.t. the system of constraints of Figure 8.

Proof

This result immediately follows from the above argument. In fact, the constraint (11) of Figure 8 is implicitly rendered by domain assignment for CLP variables. Constraints (13) and (14) are dealt with in lines (57)–(64). Line (68) imposes constraint (15). Concerning the sequentiality of the plan and the executability conditions (i.e., constraint (12)), we can observe that the implementation does not differ from that of $\mathscr{B}$ (in Fig. 11 we omitted the corresponding code, see Fig. 7). □

Let us now consider the presence of static causal laws. In Figure 12, we list the predicate used to add constraints for the static causal laws. Notice that the concrete implementation of Figure 12 contains a discrepancy with respect to the abstract one of Figure 10. In particular, the concrete implementation does not deal with an intermediate state (named v in the abstract implementation). The fluents of the

[12] When establishing completeness an soundness results for the concrete implementation, we assume the same properties hold for the real implementation of the CLP(FD) solver at hand (in our case, SICStus Prolog).

```
(124)    static([], _, _, []).
(125)    static([[OP,FE1,FE2,Cond]|Others], States, Now, [Flag|Flags]) :-
(126)      get_precondition_vars(Now, Cond, States, List),
(127)      length(List, NL),
(128)      sum(List, Result),
(129)      rel_parsing(FE1, Val1, Now, States),
(130)      rel_parsing(FE2, Val2, Now, States),
(131)      exp_constraint(Val1, OP, Val2, C),
(132)      (Result #= NL) #<=> Flag,
(133)      Flag #=> C,
(134)      static(Others, States, Now, Flags).
```

Fig. 12. Static causal laws treatment.

target state are computed by exploiting direct relationships with the starting state of the transition. This allows us to introduce fewer CLP variables.

In line (65) of Figure 11 the predicate `cluster_rules` collects all the (static) conditions imposed on the fluents of the cluster of `FluentName`. The call to the predicates `static` (line (66)) collects the list of Boolean flags `StatFormula` which are used to model the constraints (19) and (20) of Figure 10. In line (67), the disjunction of these flag variables, together with those originating from the dynamic causal laws (i.e., `DynFormula`), is computed in `Formula`, as explained above.

For each condition implied by a static causal law, the predicate `static` (lines (124)–(134)) builds the constraint `C` that must be imposed to ensure closure. The flag variable `Flag` in line (132) is introduced to reflect the satisfaction of the constraint. If `Flag` is true then the constrain `C` is asserted (line (133)). All such flags are stored in the list `Flags` (cf., the variable `StatFormula`).

We have the following result:

Theorem 13

The concrete implementation (partially depicted in Figs. 11 and 12) is complete w.r.t. the system of constraints of Figures 8 and 10.

Proof

The result directly follows from the above argument. Constraint (16) of Figure 10 is implicitly rendered by the domain assignment for the CLP variables (let us remember that the intermediate state v is not explicit in the concrete implementation). Constraints (11)–(15) are dealt with as done in Theorem 12. The conditions originating from the static causal laws are dealt with through the predicates `cluster_rules` and `static`. □

Let us observe that there is a second difference between the concrete implementation of Figures 11 and 12 and the abstract one of Figure 10: no requirements for the unsatisfiability of $Form(\mathscr{D})^{\vec{v},a_i}$ are imposed in correspondence of the state transition from v_{i-1} to v_i (for any i). This allows the generation of state transitions where the target state is potentially not minimally closed. This means that the concrete implementation may produce solutions (i.e., plans) that the abstract semantics would forbid because of the nonminimal effects of (clusters of) static causal laws. On the other hand, we reflect constraints (19) and (20) as described earlier, through the predicates `static` (listed in Fig. 12) and `cluster_rules` (whose obvious code is omitted).

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

```
(135)   lm_labeling(Actionsocc, States) :-
(136)     lm_labeling(Actionsocc, States, 1).
(137)   lm_labeling([], _, _) :- !.
(138)   lm_labeling([CurrAct|Actions], States, I) :-
(139)     lm_labeling_aux(CurrAct),
(140)     no_loop(States, I),
(141)     I1 is I+1,
(142)     lm_labeling(Actions, States, I1).
(143)   lm_labeling_aux([]).
(144)   lm_labeling_aux([action(_,A)|R]) :-
(145)     indomain(A),
(146)     lm_labeling_aux(R).

(147)   no_loop(States, A) :-
(148)     state_select(A, States, StateA),
(149)     no_loop(A, States, StateA).
(150)   no_loop(0, _, _) :- !.
(151)   no_loop(B, States, StateA) :-
(152)     B1 is B-1,
(153)     state_select(B1, States, StateB),
(154)     StateA \== StateB,
(155)     no_loop(B1, States, StateA).
```

Fig. 13. Implementation of a leftmost labeling strategy.

The final step in the design of the concrete implementation is the introduction of suitable restrictions on the labeling phase of the CLP solver. Notice that, if at step i in a trajectory, a consequence of a dynamic law involves a fluent f^j, for $j > i$, then such a constraint has to be evaluated considering as already assessed all the states v_h preceding v_i. Hence, the labeling has to proceed "left-to-right" w.r.t. the CLP variables that model the states $v_1, \ldots, v_i$. In other words, when searching for a solution, the variables representing the state v_h have to be labeled before those representing the state v_{h+1}, for each v_h in the trajectory. The implementation of this labeling strategy is depicted in Figure 13. Moreover, observe that we impose further restrictions (through the predicate `no_loop` in lines (147)–(155)) to avoid loops in plans, i.e., to forbid those trajectories where the same state appears twice.

To complete the implementation of $\mathscr{B}^{MV}$ we need to take care of the cost-based constraints, whose behavior relies on the optimization features offered by SICStus' labeling predicate: the labeling phase is guided by an objective function to be optimized.

Constraints on costs, as well as absolute temporal constraints, are handled by asserting suitable CLP constraints on the variables that model fluent values. This is realized through the predicates listed in Figure 14. In particular, `set_cost_constraints` deals with constraints on actions/plans and states. For instance, `set_statecosts` (line 167) retrieves all the assertions of the form `cost_constraint(state(I) OP Num)` and imposes the corresponding constraints. A similar predicate `set_goal` (not reported in the figure) accomplishes the same for the final state only. The predicate `set_plancost` acts similarly, using the predicate `make_one_action_occurrences` (lines (192)–(202)) where the cost for each single action is considered.

All the absolute temporal constraints defined in the action description are handled by the predicate `set_time_constraint` (cf., lines (194)–(202)). Also in this case, direct references to CLP variables implement the references to fluent expressions in any absolute point in time.

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

```
set_cost_constraints(States, PlanCost, GOALCOST) :-
  set_goalcost(States, GOALCOST),
  set_plancost(PlanCost),
  set_statecosts(States).

set_plancost(PC) :-
  findall([OP,Num],(cost_constraint(C), C=..[OP,plan,Num]), PlanCosts),
  set_plancost_aux(PlanCosts,PC).
set_plancost_aux([],_).
set_plancost_aux([[OP,Num]|PlanCosts],PC) :-
  add_constraint(PC,OP,Num),
  set_plancost_aux(PlanCosts,PC).

set_statecosts(States) :-
  findall([I,OP,N],(cost_constraint(C), C=..[OP,state(I),N]), Costs),
  set_statecost_aux(Costs,States).

set_statecost_aux([],_).
set_statecost_aux([[I,OP,Num]|StateCosts],States) :-
  (state_cost(FE),!; FE = 1),
  rel_parsing(FE,Val,I,States),
  add_constraint(Val,OP,Num),
  set_statecost_aux(StateCosts,States).

 make_action_occs(N, ActionsOcc, PlanCost, Na) :-
   setof(A, action(A), La),
   length(La, Na),
   make_action_occurrences(N, La, ActionsOcc, PlanCost).
 make_action_occurrences(1, _, [], 0).
 make_action_occurrences(N, List, [Act|ActionsOcc], Cost) :-
   N1 is N-1,
   make_action_occurrences(N1, List, ActionsOcc, Cost1),
   make_one_action_occurrences(List, Act, Cost2),
   get_action_list(Act, AList),
   fd_only_one(AList),
   Cost #= Cost1+Cost2.
 make_one_action_occurrences([], [], 0).
 make_one_action_occurrences([A|Actions], [action(A,OccA)|OccActs], Cost) :-
   make_one_action_occurrences(Actions, OccActs, Cost1),
   fd_domain_bool(OccA),
   (action_cost(A,CA),!; CA = 1), %%%Default action cost = 1
   Cost #= OccA*CA+Cost1.

 set_time_constraints(States) :-
   findall([FE1,OP,FE2], (time_constraint(C),C=..[OP,FE1,FE2]), TimeCs),
   set_time_constraints(TimeCs, States).

 set_time_constraints([], _).
 set_time_constraints([[FE1,OP,FE2]|Rest], States) :-
   rel_parsing(FE1, Val1, _, States),
   rel_parsing(FE2, Val2, _, States),
   add_constraint(Val1, OP, Val2),
   set_time_constraints(Rest, States).

 add_constraint(L, OP, R) :-
   exp_constraint(L, OP, R, 1).
```

Fig. 14. Handling of global constraints and costs.

As mentioned, all these constraints can be seen as filters used to validate each trajectory found by the labeling phase. The planner described in Figures 11–13 is completed by adding the code in Figure 14. Completeness of the implementation of the full $\mathscr{B}^{MV}$ immediately follows from the above discussion.

8 Experimental analysis

We implemented CLP-based prototypes of $\mathscr{B}$ and $\mathscr{B}^{MV}$. These have been realized in SICStus Prolog 4, and they have been developed on an AMD Opteron 2.2GHz

Linux machine. Extensive testing has been performed to validate our CLP-based approach. Here we concentrate on a few representative examples. The source code of the implementations and the examples can be found at `www.dimi.uniud.it/dovier/CLPASP`. No particular built-in predicates of SICStus have been used and therefore porting to other CLP-based Prolog systems is straightforward. A porting to B-Prolog has been realized and used to participate in the 2009 ASP Competition[13].

In the rest of this section, we analyze the performance of the implementation on a diverse set of benchmarks. For each benchmark, we compare a natural encoding using the traditional $\mathscr{B}$ language with an encoding using $\mathscr{B}^{MV}$.

The problems encoded in $\mathscr{B}$ have been solved using both the CLP(FD) implementation and implementations obtained by mapping the problem to ASP and using different ASP solvers (Smodels, Clasp, and Cmodels with different SAT-solvers).

In order to solve a $\mathscr{B}$-planning problem $\langle \mathscr{D}, \mathscr{O} \rangle$ using an ASP solver, we have developed a Prolog translator that takes as input $\langle \mathscr{D}, \mathscr{O} \rangle$ and the plan length `n`, and it generates an ASP program, whose stable models are in one-to-one correspondence with the plans of length `n` for $\langle \mathscr{D}, \mathscr{O} \rangle$. This encoding follows the general ideas outlined in Lifschitz (1999). In particular, the definitions of `fluent`, `action`, and `initially` are already in ASP syntax. The length of the plan `n` is used to define the predicate `time(0..n)`. The ASP-based planner makes use of a choice rule to ensure that exactly one action is applied at each time step:

```
1{occ(Act,Ti):action(Act)}1 :- time(Ti), Ti < n.
```

The predicate `hold(Fluent,Time)` defines the truth value of a fluent `Fluent` at a given time step (`Time`). The truth value of the fluents at time 0 are given as facts describing the initial state; we require the initial state to be complete. The executability rules, the dynamic causal laws and the static causal laws are instantiated for each admissible time step. Finally, the goal conditions are added to define the predicate `goal`; the requirement that the goal has to be satisfied at the end of the plan is imposed using an ASP constraint of the form

```
:- not goal.
```

As far as the CLP-based implementations are concerned, we use a leftmost variable selection strategy. Moreover, we included a loop control feature to avoid the repetition of the same state in a trajectory (cf., the predicate `no_loop` in Fig. 13).

Tables 1–5, discussed in detail in the next subsections, illustrate an excerpt of the experimental results. In order to simplify the comparison among the solvers, in each table we introduce an extra column, denoted by "Best ASP", which indicates the performance of an hypothetical ASP-solver that always acts as the best between all the ASP-solvers considered.

The specific meaning of the various columns is as follows:

- *Instance*: the name of the specific instance of the problem;
- *Length*: the plan length used in searching for a solution;

[13] See the web site `http://www.cs.kuleuven.be/~dtai/events/ASP-competition/Teams/Bpsolver-CLPFD.shtml`

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

- *Answer*: indication of whether an answer exists or not for the given plan length;
- *lparse*: the time required to ground the ASP encoding of the problem (using lparse 1.1.1);
- *Smodels*: the execution time using the Smodels system (using Smodels 2.32);
- *Cmodels*: the execution time using the Cmodels system (using Cmodels 3.70 with different SAT solvers);
- *Clasp*: the execution time using the Clasp system (using Clasp 1.0.2);
- *Best ASP*: a summary of the best execution time across all the different ASP solvers;
- *CLP(FD)*: the execution time using the CLP(FD)-based implementation of $\mathcal{B}$. Execution times have the form $t_1 + t_2$, where t_1 is the time needed for posting constraints and t_2 the time for solving the constraints (i.e., finding a plan);
- $\mathcal{B}^{MV}$: the execution time using the $\mathcal{B}^{MV}$ encoding of the problem. The first column is related to computations where no constraints for the plan cost are imposed. Instead, the computations of the second column have a constraint that limits the plan cost to the number in parenthesis. The format is $t_1 + t_2$ as explained in the previous point.

In the remaining subsections we briefly describe the benchmarks tested and the obtained results. The actual encoding in $\mathcal{B}$ and $\mathcal{B}^{MV}$ have been placed in the Appendix A for the sake of readability. A summary and a discussion of all the experiments is presented in Section 8.6.

8.1 Three-barrel problem

We experimented with different encodings of the three-barrel problem. Our formulation is as described in Example 1. Figure 1 and Section A.1 show the encoding of the problem (for $N = 12$) in $\mathcal{B}$ and in $\mathcal{B}^{MV}$, respectively. Notice that, in order to represent each multivalued fluent f of the $\mathcal{B}^{MV}$ formulation, a number of Boolean fluents have to be introduced in the $\mathcal{B}$ encoding, one for each admissible value of f.

Table 1 provides the execution times (in seconds) for different values of N and different plan lengths. The results show that the constraint-based encoding of $\mathcal{B}$ outperforms the ASP encodings (if we consider both grounding and execution). In turn, the $\mathcal{B}^{MV}$ encoding outperforms all other encodings. This can be explained by considering that the CLP encoding of this problem benefits from numerical fluents (in reduced number, w.r.t. the $\mathcal{B}$ formulation) and from arithmetic constraints (efficiently handled by CLP(FD)).

8.2 Two-dimensional protein folding problem

The problem we have encoded is a simplification of the protein structure folding problem. The input is a chain $\alpha_1\alpha_2\cdots\alpha_n$ with $\alpha_i \in \{0, 1\}$, initially placed in a vertical position, as in Figure 15-left. We will refer to each α_i as an *amino acid*. The permissible actions are the counter-clockwise/clockwise *pivot moves*. Once one point i of the chain is selected, the points $\alpha_1, \alpha_2, \ldots, \alpha_i$ will remain fixed, while the points $\alpha_{i+1}, \ldots, \alpha_n$

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

Table 1. *Experimental results with various instances of the three-barrel problem (timeout 24,000 sec)*

			$\mathscr{B}$								$\mathscr{B}^{MV}$		
Barrels' capacities	Length	Answer	*lparse*	Smodels	zchaff	Cmodels relsat	Cmodels minisat	Clasp	Best ASP	CLP(FD)	Unconstrained plan cost	constrained plan cost (in parentheses)	
8-5-3	6	*N*	8.74	0.10	0.34	0.63	0.30	0.27	0.10	0.14+0.29	0.03+0.03	(70)	0.02+0.03
8-5-3	7	*Y*	8.92	0.20	1.87	2.39	0.55	0.23	0.20	0.22+0.28	0.03+0.02	(70)	0.02+0.02
8-5-3	8	*Y*	8.87	0.20	7.34	3.63	0.62	0.53	0.20	0.26+1.04	0.05+0.07	(70)	0.01+0.06
8-5-3	9	*Y*	9.03	0.17	17.60	5.02	0.60	2.34	0.17	0.24+1.03	0.02+0.05	(70)	0.02+0.06
12-7-5	10	*N*	34.47	1.98	153.36	14.56	41.34	29.13	1.98	0.58+4.85	0.04+0.13	(120)	0.04+0.13
12-7-5	11	*Y*	34.54	2.28	98.72	15.78	11.71	52.15	2.28	0.64+2.61	0.02+0.07	(120)	0.03+0.07
12-7-5	12	*Y*	35.42	1.60	125.84	20.45	83.06	35.81	1.60	0.73+8.11	0.07+0.18	(120)	0.05+0.19
12-7-5	13	*Y*	35.69	0.68	342.40	42.36	97.99	111.36	0.68	0.79+6.23	0.07+0.14	(120)	0.07+0.14
16-9-7	14	*N*	115.47	11.15	1508.43	613.42	75.67	1838.39	11.15	1.30+27.16	0.03+0.31	(200)	0.07+0.31
16-9-7	15	*Y*	114.03	12.30	586.43	58.45	65.19	1133.21	12.30	1.53+13.35	0.06+0.13	(200)	0.07+0.14
16-9-7	16	*Y*	115.60	6.06	793.00	151.56	157.38	744.60	6.06	1.62+37.69	0.07+0.37	(200)	0.07+0.36
16-9-7	17	*Y*	114.60	1.75	2963.37	128.91	145.11	14106.98	1.75	1.67+26.98	0.07+0.27	(200)	0.07+0.27
20-11-9	18	*N*	185.38	43.71	2949.10	2312.09	493.98	–	43.71	2.76+102.14	0.09+0.58	(300)	0.08+0.57
20-11-9	19	*Y*	186.76	40.08	3053.53	1187.10	1152.27	11292.40	40.08	2.94+45.43	0.09+0.24	(300)	0.10+0.24
20-11-9	20	*Y*	186.31	21.67	1866.28	2265.05	1378.93	12286.98	21.67	3.05+120.90	0.09+0.68	(300)	0.09+0.65
20-11-9	21	*Y*	189.28	4.39	5482.78	586.18	1746.81	–	4.39	3.17+80.54	0.10+0.46	(300)	0.10+0.43

Table 2. *The HP-protein folding problem: some results for different sequences, and plan lengths (timeout 12,000sec)*

Instance	Length	Answer	$\mathcal{B}^{FD}_{MV}$
1^7-2	3	*Y*	0.07+0.01
1^7-2	4	*Y*	0.09+0.01
1^{13}-6	3	*N*	0.42+19.91
1^{13}-6	4	*Y*	0.57+35.16
$1(001)^2$-2	3	*N*	0.06+0.09
$1(001)^2$-2	4	*Y*	0.07+0.01
$1(001)^3$-4	7	*N*	0.47+7521.13
$1(001)^3$-4	8	*Y*	0.49+50.46
$1(001)^3$-4	9	?	–
$1(001)^3$-4	10	*Y*	0.63+603.37

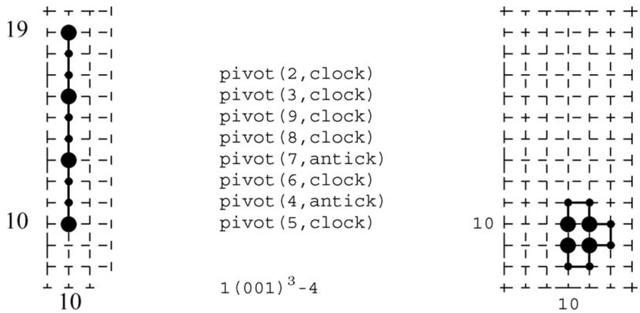


Fig. 15. An instance of the HP-protein folding problem: initial configuration, a plan, and final configuration with 4 contacts between 1-amino acids.

will perform a rigid counter-clockwise/clockwise rotation. Each conformation must be a *self-avoiding-walk*, i.e., no two amino acids are in the same position. Moreover, the chain cannot be broken – i.e., two consecutive amino acids are always at points at distance 1 (i.e., in contact). The goal is to perform a sequence of pivot moves leading to a configuration where at least k nonconsecutive amino acids of value 1 are in contact. Figure 15 shows a possible plan to reach a configuration with 4 contacts. Table 2 reports some execution times. Section reports the $\mathcal{B}^{MV}$ action description encoding this problem. Since the goal is based on the notion of cost of a given state, for which reified constraints are used extensively, a direct encoding in $\mathcal{B}$ does not seem to be feasible.

Let us consider the resolution of the instance depicted in Figure 15, i.e., the folding of the input chain 1001001001 of $n = 10$ amino acids. Asking for a plan of 8 (resp. 10) moves and for a solution with cost $\geqslant 4$, our planner finds the 8-moves plan shown in Figure 15-center in 50.46s (a 10-moves plan in found in 603.37s). By removing the two constraints that keep fixed α_2:

```
always(x(2) eq 10).
always(y(2) eq 11).
```

the solutions are found in 52.72s and 617.68s, respectively. On the other hand, by keeping fixed α_2 and adding the two constraints

```
holds(x(3) eq 11,1).
holds(y(3) eq 11,1).
```

the execution time is reduced to 4.06s and 52.97s. Adding the additional constraints

```
holds(x(4) eq 11,2).
holds(y(4) eq 10,2).
```

the plans are found in only 0.37s and 4.62s. This shows that the use of multivalued fluents and the ability to exploit domain-specific knowledge, in the form of symmetry-breaking constraints, allows $\mathcal{B}^{MV}$ to effectively converge to a solution.

8.3 The community problem

The *Community* problem is formulated as follows. There are M individuals, identified by the numbers $1, 2, \ldots, M$. At each time step, one of them, say j, gives exactly j dollars to someone else, provided she/he owns more than j dollars. Nobody can give away all of her/his money. The goal consists of reaching a state in which all the participants have the same amount of money.

Table 3 lists some results for four variants of the problem: the person i initially owns $2 * i$ dollars (instances A_M), $i + 1$ dollars (instances B_M), $i * i$ dollars (instances C_M), or $i * (1 + i)$ dollars (instances D_M).

The representations of this problem are reported in Sections A.3.1 and A.3.2.

Notice that the large number of Boolean fluents that have to be introduced in the $\mathcal{B}$ description causes failures due to lack of memory during the grounding phase (these instances are marked "mem" in Table 3). For all these experiments, the bound on memory usage was 4 GB (for the grounder, the ASP-solvers, and the CLP(FD) engine). Observe that, in some cases, also the CLP(FD)-based solver for $\mathcal{B}$ runs out of memory, while the failures of the CLP(FD) solver for $\mathcal{B}^{MV}$ have been caused by expiration of the time limit. In summary, the constraint-based encodings provides better performance in most of the instances, especially considering their better scalability w.r.t. the size of the instances. This originates from the smaller number of numerical fluents and from the efficiency of the underlying constraint solver.

8.4 The gas-diffusion problem

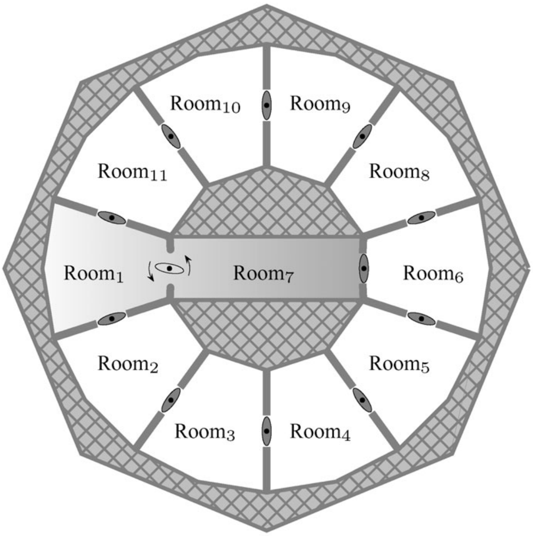
The Gas-diffusion problem can be formulated as follows. A building contains a number of rooms. Each room is connected to (some) other rooms via gates. Initially, all gates are closed and some of the rooms contain a quantity of gas – while the other rooms are empty. Each gate can be opened or closed – `open(x,y)` and `close(x,y)` are the only possible actions, provided that there is a gate between room `x` and room `y`. When a gate between two rooms is open, the gas contained in these rooms flows through the gate. The gas diffusion continues until the pressure reaches an

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

Table 3. *Experimental results for instances of the Community problem. "mem" denotes out-of-memory failures. Some results are missing for the ASP solvers, for those instances that are unable to complete grounding*

			$\mathscr{B}$								$\mathscr{B}^{MV}$
					Cmodels				Best		
Instance	Length	Answer	*lparse*	Smodels	zchaff	relsat	minisat	Clasp	ASP	CLP(FD)	CLP(FD)
A_4	5	*N*	34.34	11.12	1.78	11.68	0.67	0.45	0.45	0.71+14.14	0.01+3.31
A_4	6	*Y*	34.90	1.43	0.26	7.38	0.57	0.09	0.09	0.82+0.10	0.03+0.00
A_4	7	*Y*	35.44	15.72	0.39	47.74	0.80	0.10	0.10	0.94+0.12	0.03+0.01
A_5	5	*N*	201.88	100.58	5.22	125.63	2.30	1.19	1.19	2.64+157.48	0.02+41.15
A_5	6	*Y*	202.64	11.43	1.85	442.22	1.63	0.28	0.28	3.17+0.21	0.01+0.04
A_5	7	*Y*	202.12	34.02	2.81	114.74	2.31	0.27	0.27	3.71+447.87	0.04+142.27
B_5	5	*N*	51.87	30.04	4.24	44.49	1.49	0.69	0.69	1.03+77.06	0.03+23.13
B_5	6	*Y*	52.04	2.07	1.32	37.96	0.99	0.14	0.14	1.31+0.11	0.04+0.02
B_5	7	*Y*	52.94	13.49	0.80	41.86	1.27	0.42	0.42	1.40+0.17	0.05+0.04
B_7	5	*N*	mem							7.67+3345.56	0.05+1421.54
C_5	5	*N*	mem							16.98+85.71	0.02+49.83
C_5	6	*N*	mem							20.44+1926.97	0.04+888.30
C_7	5	*N*	mem							mem	0.05+3186.34
D_4	5	*N*	138.91	7.08	1.28	13.48	0.76	0.43	0.43	3.70+21.19	0.01+6.83
D_4	6	*N*	139.88	90.32	11.56	87.11	3.62	3.72	3.72	4.32+0.50	0.02+0.74
D_4	7	*N*	139.82	1015.44	104.36	788.94	33.70	22.86	22.86	5.17+5.55	0.04+7.64
D_5	5	*N*	mem							24.64+24.12	0.05+93.88
D_5	6	*N*	mem							29.60+1490.48	0.02+1801.78

(a)



(b)

Instance	Length	Answer	$\mathcal{B}^{MV}$
A_1	6	N	0.07 + 13.48
A_1	7	Y	0.10 + 5.35
B_1	10	N	0.17 + 3846.20
B_1	11	Y	0.14 + 1802.76
B_1	12	Y	0.15 + 933.35
B_1	13	Y	0.16 + 302.34
B_1	14	Y	0.14 + 4.60
B_2	10	N	0.13 + 11134.82
B_2	11	Y	0.15 + 4191.20
B_2	12	Y	0.16 + 2156.52
B_2	13	Y	0.18 + 710.53
B_2	14	Y	0.17 + 6.36
B_3	10	N	0.12 + 18763.27
B_3	11	Y	0.16 + 6124.91
B_3	12	Y	0.15 + 3148.43
B_3	13	Y	0.20 + 1145.04
B_3	14	Y	0.18 + 9.97
B_4	10	N	0.11 + 17109.05
B_4	11	Y	0.15 + 6173.49
B_4	12	Y	0.14 + 3180.53
B_4	13	Y	0.16 + 1159.27
B_4	14	Y	0.18 + 10.34

Fig. 16. (a) A simple schema of the 11 rooms for the Gas-diffusion problem. The locked gates are in red color. The gas (in pink) is flowing through the open gate (in green), from Room_7 to Room_1. (b) Some results for different instances (i.e., different goals and initial allocations of amounts of gas – see Section 8.4).

equilibrium. The only condition to be always satisfied is that a gate in a room can be opened only if all the other gates are closed. The goal is to move a desired quantity of gas to one specified room.

We experimented with instances of the problem where the building has a specific topology: there are 11 rooms, all having the same physical volume. Each room is connected to the other rooms via gates as depicted in Figure 16. Since all rooms have the same volume, when equilibrium is reached between two rooms sharing an open gate, they will both contain the same amount of gas.

A $\mathcal{B}^{MV}$ specification of this planning problem is given in Section A.6. We experimented with different instances of the Gas-diffusion problem obtained by considering different goal states and by requiring that some of the rooms have to be kept empty. Moreover, we seek plans of different length. Figure 16(b) summarizes the results obtained. In particular, all instances share the same initial state: rooms 10 and 3 contain 128 moles of gas. All the other rooms are empty. Moreover,

- in the instance A_1 the goal state is: room 1 contains at least 32 moles of gas;
- in all the instances B_i the goal is: room 1 contains at least 50 moles of gas. The B_i instances differ in the constraints imposed on the desired plan:

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

— in the instance B_1, rooms 7, 9, and 4 must remain empty. This condition can be imposed by including in the action description the constraints

```
always(contains(7) eq 0).
always(contains(9) eq 0).
always(contains(4) eq 0).
```

— in the instance B_2, rooms 7, 8, and 5 must be kept empty;
— in the instance B_3, only room 6 must be kept empty;
— in the instance B_4, no constraint is imposed.

Observe that it is quite natural to design a $\mathscr{B}^{MV}$ encoding of this problem, by exploiting the multivalued fluents. On the other hand, adopting the naive approach used for the three-barrel problem would force the introduction of (at least) 128 distinct Boolean fluents for each multivalued fluent. Such a large number of Boolean fluents generates a large state space, making the task of any solver for $\mathscr{B}$ considerably harder.

8.5 Other puzzles

We report results from two other planning problems. The first – 3x3-puzzle – is an encoding of the 8-tile puzzle problem, where the goal is to find a sequence of moves to re-order the 8 tiles, starting from a random initial position. The performance results for this puzzle are reported in Table 4. The second problem is the well-known *Wolf-goat-cabbage* problem. The performance results are reported in Table 5.

Notice that these planning problems are predominantly Boolean. The constraint-based encodings perform well in solving the instances of the Wolf-goat-cabbage problem. In contrast, for the 8-tile puzzle problem, the use of numerical fluents allows us to achieve a compact encoding, but it does not necessarily lead to a better performance w.r.t. ASP.

8.6 A summary of the experiments

Table 6 pictorially summarizes some of the results relating the performance of the different approaches. For each problem instance, we compare the execution times obtained by the best ASP-solver and the CLP(FD) solvers for $\mathscr{B}$ and $\mathscr{B}^{MV}$ action description languages. We considered only those instances for which at least one of the solvers gave an answer. A score of 1 (0, −1) is assigned to the fastest (second fastest, slowest) solver. The scores of all instances of a problem have been summed together, and this provides the radius of the circles in the figure. Instances have been separated between *"Yes"* instances (they admit a solution) and *"No"* instances (they have no solutions).

The success of the constraint-based approach is evident. However, it is interesting to observe that the planning problems that do not make significant use of non-Boolean fluents tend to perform better in the ASP-based implementations – possibly due to the greater efficiency of ASP solvers in propagating Boolean knowledge during search for a solution. Conversely, when numerical quantities are relevant in

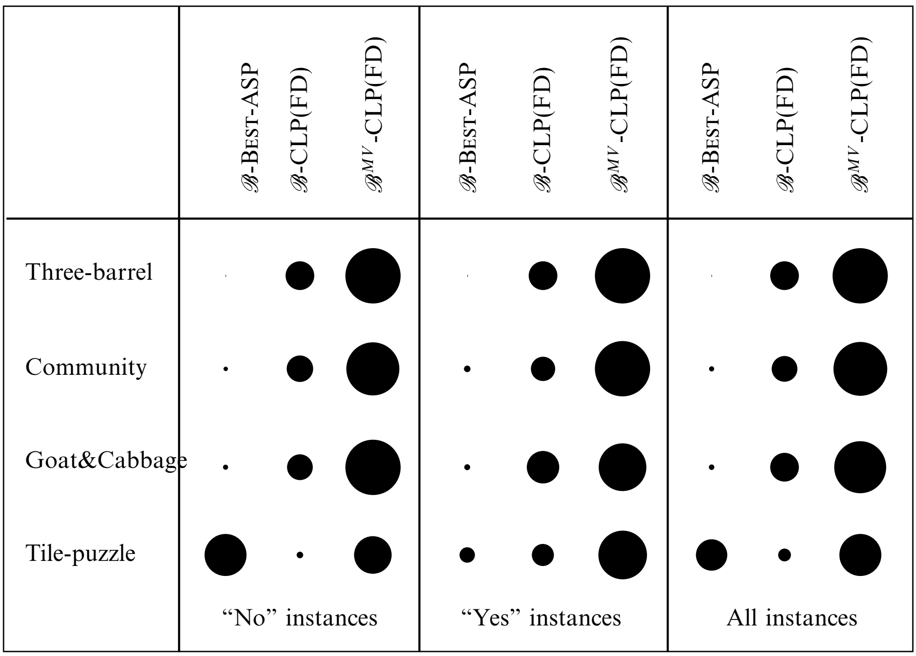
Table 4. *Experimental results for instances of the 8-tile puzzle problem (timeout 36,000 sec)*

			$\mathscr{B}$								$\mathscr{B}^{MV}$
Instance	Length	Answer	*lparse*	Smodels	zchaff	Cmodels relsat	minisat	Clasp	Best ASP	CLP(FD)	CLP(FD)
I_1	9	*N*	41.49	0.94	2.06	3.36	1.54	0.52	0.52	0.64+4.42	0.25+2.64
I_1	10	*Y*	41.80	2.02	2.52	7.36	2.06	0.70	0.70	0.73+5.43	0.29+3.64
I_2	14	*N*	42.68	27.10	34.46	90.07	7.15	7.42	7.15	1.03+57.54	0.40+38.67
I_2	15	*Y*	43.14	50.73	49.50	131.38	8.90	1.98	1.98	1.06+7.08	0.43+4.60
I_3	19	*N*	43.76	739.39	1255.46	911.82	91.75	268.69	91.75	1.39+967.26	0.54+673.66
I_3	20	*Y*	44.52	368.28	1090.66	1445.78	58.89	268.59	58.89	1.46+597.92	0.52+435.96
I_4	24	*N*	51.59	10247.47	–	5613.98	7862.10	4185.42	4185.42	1.70+13887.17	0.71+10109.58
I_4	25	*Y*	55.54	1430.43	954.68	1023.22	437.11	875.16	437.11	1.84+79.20	0.73+57.00
I_5	24	*N*	49.64	6936.39	–	6041.87	1239.72	4901.13	1239.72	1.69+11092.48	0.73+9155.79
I_5	25	*N*	51.07	14079.78	3747.96	8583.44	11745.93	8557.94	3747.96	1.84+18301.15	0.73+14195.54

Table 5. *Experimental results for instances of the Wolf-goat-cabbage problem*

		$\mathcal{B}$								$\mathcal{B}^{MV}$
				Cmodels				Best		
Length	Answer	*lparse*	Smodels	zchaff	relsat	minisat	Clasp	ASP	CLP(FD)	CLP(FD)
21	*N*	0.10	0.19	1.38	1.89	0.67	0.19	0.19	0.10+0.20	0.09+0.15
22	*N*	0.10	0.25	1.46	3.32	0.77	0.56	0.25	0.09+0.21	0.11+0.17
23	*Y*	0.10	0.26	2.30	4.34	0.58	0.13	0.13	0.12+0.17	0.07+0.15
24	*N*	0.11	0.43	3.10	4.75	0.67	1.09	0.43	0.07+0.32	0.06+0.25
25	*Y*	0.12	0.27	1.15	4.92	0.74	0.42	0.27	0.12+0.06	0.08+0.08
26	*N*	0.12	0.68	7.23	11.52	1.18	0.69	0.68	0.10+0.49	0.10+0.40
27	*Y*	0.13	0.43	1.93	6.68	0.93	0.84	0.43	0.10+0.03	0.06+0.03
28	*N*	0.14	1.24	9.44	18.72	1.59	2.15	1.24	0.10+0.80	0.08+0.69
29	*Y*	0.14	0.41	1.75	15.55	1.10	0.60	0.41	0.11+0.01	0.07+0.03
30	*N*	0.15	2.97	16.17	43.53	2.31	1.78	1.78	0.11+1.08	0.08+1.05
31	*Y*	0.15	0.49	8.40	7.10	0.89	4.60	0.49	0.12+0.01	0.11+0.04
32	*N*	0.16	2.78	23.76	38.58	2.20	5.37	2.20	0.13+1.35	0.09+1.32
33	*Y*	0.16	1.06	31.92	26.67	1.23	0.57	0.57	0.10+0.07	0.14+0.06
34	*N*	0.17	3.61	38.62	51.22	3.11	5.86	3.11	0.13+1.75	0.10+1.60
35	*Y*	0.18	1.39	31.10	30.25	3.20	4.21	1.39	0.15+0.54	0.08+0.32
36	*N*	0.18	4.55	43.97	57.21	4.24	12.68	4.24	0.13+1.87	0.11+1.79

Table 6. *Relative performance of the solvers for each set of instances (the radii of the circles are proportional to the performance of the specific solver)*



https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

modeling a planning problem, the use of multivalued fluents and constraints not only reduces the modeling effort, yielding more concise formalizations, but also requires a smaller number of fluents (compared with the analogous Boolean encoding). This, combined with the use of constraints, often translates into a smaller state space to be explored in finding a solution. These seem to be the main reasons for the better behavior provided by the $\mathscr{B}^{MV}$ approach.

The distinction between *"Yes"* and *"No"* instances is also very relevant. The CLP-based solvers tend to perform better on the *"Yes"* instances, especially for large instances. It is interesting to observe that a similar behavior has been observed in recent studies comparing performance of ASP and CLP solutions to combinatorial problems (Dovier *et al* 2005, 2007, 2009a).

9 Related work

The literature on planning and planning domain description languages is extensive, and it would be impossible to summarize it all in this context. We focus our discussion and comparison to the papers that present languages and techniques similar to ours.

The language investigated in this work is a variant of the language $\mathscr{B}$ originally introduced in Gelfond and Lifschitz (1998), as presented in Son *et al.* (2001, Section 2). Apart from minor syntactical differences, any action description $\mathscr{D}$ from the language of Son *et al.* (2001) can be embedded in our $\mathscr{B}$. The semantics for $\mathscr{B}$ presented here reproduces the one of Gelfond and Lifschitz (1998).

The language $\mathscr{ADC}$ has been introduced in Baral *et al.* (2002) to model planning problems in presence of actions with duration and delayed effects. The language relies on multivalued fluents, akin to those used in our language. $\mathscr{ADC}$ actions have two types of effects:

(1) Direct modification of fluent values, described by dynamic causal laws of the forms

$$a \textbf{ causes } f = g(f, f_1, \ldots, f_n, t) \textbf{ from } t_1 \textbf{ to } t_2, \tag{24}$$

$$a \textbf{ contributes } g(f, f_1, \ldots, f_n, t) \textbf{ to } f \textbf{ from } t_1 \textbf{ to } t_2. \tag{25}$$

The first axiom describes the value of the fluent f as a function, which modifies its value over the period of time from t_1 to t_2 – these represent time units relative to the current point in time. The second axiom is similar, except that it denotes the quantity that should be added to the value of f over the period of time. These axioms are important when describing actions whose effect has a known duration over time (i.e., the interval of length $t_2 - t_1$).

(2) Indirect modifications through the initiation and termination of *processes* that can modify fluents until explicitly stopped; the axioms involved are axioms for the creation and termination of processes:

$$a_1 \textbf{ initiates } p \textbf{ from } t_1, \tag{26}$$

$$a_2 \textbf{ terminates } p \textbf{ at } t_2, \tag{27}$$

and axioms that describe how processes modify fluents

$$p \textbf{ is\_associated\_with } f = g(f, f_1, \ldots, f_n, t), \quad (28)$$

$$p \textbf{ is\_associated\_with } f \leftarrow g(f, f_1, \ldots, f_n, t). \quad (29)$$

The first axiom describes how the value of the fluent f will change as a function of time once a process is started; the second axiom determines how the value of f changes while the process p is active.

$\mathscr{ADC}$ has some similarities to $\mathscr{B}^{MV}$; they both allow multivalued fluents and some forms of temporal references. $\mathscr{B}^{MV}$ has the flexibility of allowing non-Markovian behavior and it allows references to values of fluents at different time points, features that are missing in $\mathscr{ADC}$. On the other hand, $\mathscr{ADC}$ allows the representation of continuous time and the ability to describe continuous changes to the value of fluents.

Several features of $\mathscr{ADC}$ can be reasonably simulated in $\mathscr{B}^{MV}$; we will focus on the axioms of type (26)–(29), since these subsume the capabilities of axioms (24) and (25):

- we can represent each process p using a corresponding fluent;
- the axioms (26) and (27) can be simulated by

$$\texttt{causes}(a_1, p^{t_1-1} = 1, \texttt{true}), \qquad \texttt{causes}(a_2, p^{t_2-1} = 0, \texttt{true});$$

- the axiom (28) can be simulated by introducing the static causal law

$$\texttt{caused}(p > 0, f = g(f^{-1}, f_1^{-1}, \ldots, f_n^{-1}, p^{-1}) \wedge p = p^{-1} + 1).$$

Note that, due to the inability of $\mathscr{B}^{MV}$ to handle continuous time, we are considering only discrete time measures.

The language $\mathscr{C}^+$ proposed in Giunchiglia *et al.* (2004a) also has some similarities to the language $\mathscr{B}^{MV}$. $\mathscr{C}^+$ does not offer capabilities for non-Markovian and temporal references, but supports multivalued fluents. The syntax of $\mathscr{C}^+$ builds on a language of fluent constants (each with an associated domain) and action names (viewed as Boolean variables):

- Static causal laws

$$\textbf{caused } F \textbf{ if } G,$$

  where F and G are fluent formulae (i.e., propositional combinations of atoms of the form $f = v$ for f fluent and $v \in \text{dom}(f)$). The language introduces syntactic restrictions that are effectively equivalent to preventing cyclic dependencies among fluents. Static causal laws describe dependencies between fluents within a state of the world.
- Fluent dynamic laws

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H,$$

  where F and G are fluent formulae and H is a formula that may also contain action variables. The semantics of dynamic laws can be summarized as follows: if H holds in a state, then the implication $G \rightarrow F$ should hold in the successive state.

- Actions that can be freely generated are declared to be exogenous

$$\textbf{exogenous } a.$$

- Fluents can be declared to be inertial (i.e., they satisfy the frame axiom)

$$\textbf{inertial } f.$$

The relationships between the two languages can be summarized as follows:

- $\mathscr{C}^+$ is restricted to noncyclic dependencies among fluents, while $\mathscr{B}^{MV}$ lifts this restriction.
- $\mathscr{C}^+$ is capable of identifying fluents as inertial or noninertial, while $\mathscr{B}^{MV}$ focuses only on inertial fluents (though it is relatively simple to introduce an additional type of constraint to create noninertial fluents).
- $\mathscr{C}^+$ can describe domains where concurrent actions are allowed – by allowing occurrences of different action variables in the H component of the fluent dynamic laws; although $\mathscr{B}^{MV}$ does not currently supports this feature, a similar extension has been investigated in a recent paper (Dovier *et al.* 2009b).

Subsets of $\mathscr{B}^{MV}$ and $\mathscr{C}^+$ can be shown to have the same expressive power; in particular, let us consider the subset of $\mathscr{C}^+$ that contains only domains that meet the following requirements:

- there are no concurrent actions – i.e., each H contains exactly one occurrence of an action variable; thus

$$\textbf{caused } F \textbf{ if } G \textbf{ after } a \wedge H,$$

where H is a fluent formula;
- for each action a, there is a declaration

$$\textbf{exogenous } a.$$

Under these restrictions, it is possible to map a $\mathscr{C}^+$ domain D to an equivalent domain in $\mathscr{B}^{MV}$. In particular,

- for each noninertial fluent f, with default value v, we introduce the static law

$$\texttt{caused}(f^{-1} \neq v, f^0 = v)$$

- for each static causal law **caused** F **if** G we introduce a causal law $\texttt{caused}(G, F)$
- for each fluent dynamic law r of the form **caused** F **if** G **after** $a \wedge H$, we introduce the following axioms (where $exec\_r$ is a fresh fluent):

$$\texttt{causes}(a, exec\_r = 1, H),$$
$$\texttt{causes}(a, exec\_r^1 = 0, H),$$
$$\texttt{caused}(exec\_r = 1 \wedge G, F).$$

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

Logic programming, and more specifically Prolog, has been also used to implement the first prototype of GOLOG (as discussed in Levesque *et al.* 1997). GOLOG is a programming language for describing agents and their capabilities of changing the state of the world. The language builds on the foundations of situation calculus. It provides high-level constructs for the definition of complex actions and for the introduction of control knowledge in the agent specification. Prolog is employed to create an interpreter, which enables, for example, to answer projection queries (i.e., determine the properties that hold in a situation after the execution of a sequence of actions). The goals of GOLOG and the use of logic programming in that work are radically different from the focus of our work.

The work by Thielscher (2002a) takes a different perspective in using constraint programming to handle problems in reasoning about actions and change. Thielscher's work builds on the use of Fluent Calculus (Thielscher 1999) for the representation of actions and their effects. Fluent calculus views states as sets of fluents, constructed using an operator $\circ$, and with the ability to encode partially specified sets (e.g., $f_1 \circ f_2 \circ Z$ where Z represents the "rest" of the state). In Thielscher (2002a), an encoding of the fluent calculus axioms using Constraint Handling Rules (CHRs) is presented; the encoding uses *lists* to represent states, and it employs CHRs to explicitly implement the operations on lists required to operate on states – e.g., truth or falsity of a fluent, validation of disjunctions of fluents. The ability to code open lists enables reasoning with incomplete knowledge. Experimental results (reported in Thielscher 2002b) denote a good performance with respect to GOLOG. The framework is very suitable for dealing with incomplete knowledge and sensing actions. Differently from our framework, it does not support non-Markovian reasoning, multivalued reasoning, and it does not bring the expressiveness of constraint programming to the level of the action specification language. The use of constraints in the two approaches is radically different – Thielscher's work develops new constraint solvers to implement reasoning about states, while we use existing solvers as black boxes.

A strong piece of work regarding the use of constraint programming in planning is (Vidal and Geffner 2006). The authors use constraint programming, based on the CLAIRE language (Caseau *et al.* 2002), to encode temporal planning problems and to search for minimal plans. They also use a series of interesting heuristics for solving that problem. This line of research is more accurate than ours from the implementation point of view – although their heuristic strategies can be implemented in our system and it would be interesting to exploit them during the labeling phase. On the other hand, the proposal by Vidal and Geffner only deals with Boolean fluents and without explicitly defined static causal laws.

Similar considerations can be done with respect to the cited proposal by Lopez and Bacchus (Lopez and Bacchus 2003). The authors start from Graphplan and exploit constraints to encode k-plan problems. Fluents are in this case only Boolean (not multivalued) and the process is deterministic once an action is chosen (instead, we deal also with nondeterminism, e.g., when we have consequences such as $f > 5$). The proposal of Lopez and Bacchus does not address the encoding of static causal laws.

10 Conclusions and future work

In this paper, we investigated the application of constraint logic programming technology to the problem of reasoning about actions and change and planning. In particular, we presented a modeling of the action language $\mathscr{B}$ using constraints, developed an implementation using CLP(FD), and reported on its performance. We also presented the action language $\mathscr{B}^{MV}$, which allows the use of multivalued fluents and the use of constraints as conditions and consequences of actions. Once again, the use of constraints is instrumental in making these extensions possible. We illustrated the application of both $\mathscr{B}$ and $\mathscr{B}^{MV}$ to several planning problems. Both languages have been implemented using SICStus Prolog.

We consider the research and the results discussed in this paper as a preliminary step in a very promising direction. The experimental results, as well as the elegance of the encodings of complex problems, shows the promise of constraint-based technology to address the needs of complex planning domains. A number of research directions are currently being pursued:

- we have introduced the use of global constraints to encode different forms of preferences (e.g., action costs) and control knowledge. Global constraints have been widely used in constraint programming to enhance efficiency, by providing more effective constraint propagations between sets of variables; we believe a similar use of global constraints can be introduced in the context of planning – e.g., the use of techniques used to efficiently handle the `alldifferent` global constraint to enforce nonrepetition of states in a trajectory.
- We also believe that significant improvements in efficiency can be achieved by delegating parts of the constraint solving process to an efficient dedicated solver (e.g., encoded using a constraint platform such as GECODE, possibly enhanced with local search moves).
- The encoding in CLP(FD) allow us to think of extensions in several directions, such as the encoding of qualitative and quantitative preferences (a preliminary study has been presented in Tu *et al.* 2007), and the use of constraints to represent incomplete states – e.g., to determine most general conditions for the existence of a plan and to support conformant planning (Son *et al.* 2007).
- An interesting line of research is represented by the application of the approach discussed here to multiagent systems. In that case, besides admitting the execution of more that one action in each state transition (cf., Remark 1), other important issues have to be addressed, since different agents may compete or collaborate in order to reach the desired results. For instance, concurrency of actions may be subject to constraints to model incompatibilities or interdependencies among the occurrences/effects of different actions executed by different agents (even in different points in time). Hence, the action description language, as well as its CLP encoding, has to be suitably enriched in order to deal with these aspects. A first step in this direction has been presented in (Dovier *et al.* 2009b).

Acknowledgements

The authors would like to thank the following researchers for their help, comments, and suggestions: Son Cao Tran, Michael Gelfond, and the anonymous reviewers of ICLP 2007 and TPLP.

The research has been partially supported by NSF Grants IIS-0812267, HRD-0420407, and CNS-0220590, by the FIRB grant RBNE03B8KK, and by GNCS – *Gruppo Nazionale per il Calcolo Scientifico* (project *Tecniche innovative per la programmazione con vincoli in applicazioni strategiche*), and by MUR-PRIN 2008.

Appendix A Some of the codes of the experimental section

A.1 The three-barrel problem: $\mathscr{B}^{MV}$ description of the 12-7-5 barrels problem

The $\mathscr{B}^{MV}$ encoding of the three barrels planning problem for $N = 12$. (Fig. 1 presents an encoding using the language $\mathscr{B}$.)

```
barrel(5).
barrel(7).
barrel(12).

fluent(cont(B),0,B) :- barrel(B).

action(fill(X,Y)) :- barrel(X), barrel(Y), neq(X,Y).

causes(fill(X,Y), cont(X) eq 0, [Y-cont(Y) geq cont(X)]) :-
   action(fill(X,Y)).
causes(fill(X,Y), cont(Y) eq cont(Y)^(-1)+cont(X)^(-1),
      [Y-cont(Y) geq cont(X)]) :-
   action(fill(X,Y)).
causes(fill(X,Y), cont(Y) eq Y, [Y-cont(Y) lt cont(X)]) :-
   action(fill(X,Y)).
causes(fill(X,Y), cont(X) eq cont(X)^(-1)-Y+cont(Y)^(-1),
      [Y-cont(Y) lt cont(X)]) :-
   action(fill(X,Y)).

executable(fill(X,Y), [cont(X) gt 0, cont(Y) lt Y]) :-
   action(fill(X,Y)).

caused([], cont(12) eq 12-cont(5)-cont(7)).

initially(cont(12) eq 12).

goal(cont(12) eq cont(7)).
```

A.2 The HP protein folding problem

$\mathscr{B}^{MV}$ encoding of the HP-protein folding problem with pivot moves on input of the form 1001001001... starting from a vertical straight line.

```
length(10).
amino(A) :- length(N), interval(A,1,N).
direction(clock).
direction(antick).
```

```
fluent(x(A),1,M) :-
   length(N), M is 2*N, amino(A).
fluent(y(A),1,M) :-
   length(N), M is 2*N, amino(A).
fluent(type(A),0,1) :-
   amino(A).
fluent(saw,0,1).

action(pivot(A,D)) :-
   length(N), amino(A),
   1<A, A<N, direction(D).

executable(pivot(A,D), []) :- action(pivot(A,D)).

causes(pivot(A,clock), x(B) eq x(A)^(-1)+y(B)^(-1)-y(A)^(-1), []) :-
   action(pivot(A,clock)), amino(B), B > A.
causes(pivot(A,clock), y(B) eq y(A)^(-1)+x(A)^(-1)-x(B)^(-1), []) :-
   action(pivot(A,clock)), amino(B), B > A.
causes(pivot(A,antick), x(B) eq x(A)^(-1)-y(B)^(-1)+y(A)^(-1), []) :-
   action(pivot(A,antick)), amino(B), B > A.
causes(pivot(A,antick), y(B) eq y(A)^(-1)-x(A)^(-1)+x(B)^(-1), []) :-
   action(pivot(A,antick)), amino(B), B > A.

caused([x(A) eq x(B), y(A) eq y(B)], saw eq 0) :-
   amino(A), amino(B), A < B.

initially(saw eq 1).
initially(x(A) eq N) :- length(N), amino(A).
initially(y(A) eq Y) :- length(N), amino(A), Y is N+A-1.
initially(type(X) eq 1) :- amino(X), X mod 3 =:= 1.
initially(type(X) eq 0) :- amino(X), X mod 3 =\= 1.

goal(saw gt 0).

state_cost(FE) :- length(N), auxc(1,4,N,FE).
auxc(I,J,N,0) :- I > N-3,!.
auxc(I,J,N,FE) :- J > N, !, I1 is I+1,
   J1 is I1+3, auxc(I1,J1,N,FE).
auxc(I,J,N,FE1+type(I)*type(J)*rei(abs(x(I)-x(J))+abs(y(I)-y(J)) eq 1)) :-
   J1 is J+2, auxc(I,J1,N,FE1).

always(x(1) eq 10). always(y(1) eq 10).
always(x(2) eq 10). always(y(2) eq 11).

cost_constraint(goal geq 4).
```

A.3 The community problem

A.3.1 $\mathscr{B}$ description of the instance A_4

```
max_people(4).
person(X) :- max_people(N), interval(X,1,N).
money(X) :- max_people(N), M is N*(N+1), interval(X,1,M).

fluent(owns(B,M)) :- person(B), money(M).

action(gives(X,Y)) :-
   person(X), person(Y), neq(X,Y).

executable(gives(X,Y), [owns(X,Mx)]) :-
   action(gives(X,Y)),
   fluent(owns(X,Mx)), Mx > X.

causes(gives(X,Y), owns(X,NewMx), [owns(X,Mx)]) :-
   action(gives(X,Y)), money(Mx),
   fluent(owns(X,NewMx)), fluent(owns(X,Mx)),
   NewMx is Mx-X.
causes(gives(X,Y), owns(Y,NewMy), [owns(Y,My)]) :-
   action(gives(X,Y)), money(My),
```

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

```
      fluent(owns(Y,NewMy)), fluent(owns(Y,My)),
      NewMy is My+X.

   caused([owns(X,Mx)], neg(owns(X,Other))) :-
      fluent(owns(X,Mx)), fluent(owns(X,Other)),
      person(X), money(Mx), money(Other), neq(Mx,Other).

   initially(owns(X,M)) :-
      person(X), M is 2*X.

   goal(owns(X,Mid)) :-
      person(X), max_people(N), Mid is (N*(N+1))//N.
```

A.3.2 $\mathscr{B}^{MV}$ description of the instance A_4

```
max_people(4).
person(X) :- max_people(N), interval(X,1,N).

fluent(owns(B),1,M) :-
   person(B), max_people(N), M is N*(N+1).

action(gives(X,Y)) :-
   person(X), person(Y), neq(X,Y).

executable(gives(X,Y), [owns(X) gt X]) :-
   action(gives(X,Y)).

causes(gives(X,Y), owns(X) eq owns(X)^(-1)-X, []) :-
   action(gives(X,Y)).
causes(gives(X,Y), owns(Y) eq owns(Y)^(-1)+X, []) :-
   action(gives(X,Y)).

initially(owns(X) eq M) :-
   person(X), M is 2*X.

goal(owns(X) eq Mid) :-
   person(X), max_people(N), Mid is (N*(N+1))//N.
```

A.4 The 8-tile puzzle problem

A.4.1 $\mathscr{B}$ description of the instance I_1

```
cell(X) :- interval(X,1,9).
val(X) :- interval(X,1,9), neq(X,3).
near(1,2).  near(1,4).
near(2,1).  near(2,3).  near(2,5).
near(3,2).  near(3,6).
near(4,1).  near(4,5).  near(4,7).
near(5,2).  near(5,4).  near(5,6).  near(5,8).
near(6,3).  near(6,5).  near(6,9).
near(7,4).  near(7,8).
near(8,5).  near(8,7).  near(8,9).
near(9,6).  near(9,8).

fluent(at(X,Y)) :- val(X), cell(Y).
fluent(free(Y)) :- cell(Y).

action(move(X,Y)) :- val(X), cell(Y).

executable(move(X,Y), [at(X,Z), free(Y)]) :-
   val(X), cell(Y), cell(Z), near(Z,Y).

causes(move(X,Y), at(X,Y), []) :-
   val(X), cell(Y).
causes(move(X,Y), free(Z), [at(X,Z)]) :-
   val(X), cell(Y), cell(Z).
```

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

```
caused([at(X,Y)], neg(free(Y))) :-
   val(X), cell(Y).

caused([at(X,Y)], neg(at(X,Z))) :-
   val(X), cell(Y), cell(Z), neq(Y,Z).
caused([at(X,Y)], neg(at(W,Y))) :-
   val(X), val(W), cell(Y), neq(X,W).

initially(at(1,1)). initially(at(2,3)). initially(at(4,8)).
initially(at(5,2)). initially(at(6,9)). initially(at(7,4)).
initially(at(8,6)). initially(at(9,7)). initially(free(5)).
initially(neg(at(1,X))) :- cell(X), neq(X,1).
initially(neg(at(2,X))) :- cell(X), neq(X,3).
initially(neg(at(4,X))) :- cell(X), neq(X,8).
initially(neg(at(5,X))) :- cell(X), neq(X,2).
initially(neg(at(6,X))) :- cell(X), neq(X,9).
initially(neg(at(7,X))) :- cell(X), neq(X,4).
initially(neg(at(8,X))) :- cell(X), neq(X,6).
initially(neg(at(9,X))) :- cell(X), neq(X,7).
initially(neg(free(X))) :- cell(X), neq(X,5).

goal(at(X,X)) :- val(X).
goal(free(3)).
```

A.4.2 $\mathscr{B}^{MV}$ *description of the instance* I_1

```
cell(X) :- interval(X,1,9).
tile(X) :- interval(X,1,9), neq(X,3).
near(1,2).  near(1,4).
   ...%as for 𝒝...
near(9,6).  near(9,8).
fluent(at(X),1,9) :- tile(X).
fluent(free,1,9).

action(move(X,Y)) :- cell(Y), tile(X).

executable(move(X,Y), [at(X) eq Z, free eq Y]) :-
   tile(X), cell(Y), near(Z,Y).
causes(move(X,Y), at(X) eq Y, []) :-
   tile(X), cell(Y).
causes(move(X,Y), free eq at(X)^(-1), []) :-
   tile(X), cell(Y).

initially(at(1) eq 1). initially(at(2) eq 3).
initially(at(4) eq 8). initially(at(5) eq 2).
initially(at(6) eq 9). initially(at(7) eq 4).
initially(at(8) eq 6). initially(at(9) eq 7).
initially(free eq 5).
goal(at(X) eq X) :- tile(X).
goal(free eq 3).
```

A.5 The Wolf-goat-cabbage problem

A.5.1 $\mathscr{B}$ *description of the Wolf-goat-cabbage problem*

```
obj(goat).
obj(cabbage).
obj(wolf).
obj(man).
side(left). side(right).
pos(X) :- side(X).
pos(boat).

fluent(is_in(X,Y)) :- obj(X), pos(Y).
fluent(boat_at(Y)) :- side(Y).
fluent(alive).
```

```
action(sail(A,B)) :- side(A), side(B), neq(A,B).
action(go_aboard(A)) :- obj(A).
action(get_off(A)) :- obj(A).

executable(sail(A,B), [boat_at(A), is_in(man,boat)]) :-
   side(A), side(B), neq(A,B).
executable(go_aboard(A), [boat_at(L), is_in(A,L)]) :-
   obj(A), side(L).
executable(get_off(A), [is_in(A,boat)]) :-
   obj(A).

causes(sail(A,B), boat_at(B), []) :-
   side(A), side(B), neq(A,B).
causes(go_aboard(A), is_in(A,boat), []) :-
   obj(A).
causes(get_off(A), is_in(A,L), [boat_at(L)]) :-
   obj(A), side(L).

caused([is_in(Ogg,L1)], neg(is_in(Ogg,L2))) :-
   obj(Ogg), pos(L1), pos(L2), neq(L1,L2).
caused([boat_at(L1)], neg(boat_at(L2))) :-
   side(L1), side(L2), neq(L1,L2).
caused([is_in(A,boat), is_in(B,boat)], neg(alive)) :-
   obj(A), obj(B), diff(A,B,man).
caused([is_in(wolf,L), is_in(goat,L), neg(is_in(man,L))], neg(alive)) :-
   pos(L).
caused([is_in(cabbage,L), is_in(goat,L), neg(is_in(man,L))], neg(alive)) :-
   pos(L).

initially(is_in(A,left)) :- obj(A).
initially(alive).
initially(boat_at(left)).

goal(is_in(A,right)) :- obj(A).
goal(alive).
```

A.5.2 $\mathscr{B}^{MV}$ description of the Wolf-goat-cabbage problem

```
obj(goat).
obj(cabbage).
obj(wolf).
obj(man).
% 0=boat, 1=on-the-left, 2=on-the-right:

fluent(is_in(X),0,2) :- obj(X).
fluent(boat_at,1,2).
fluent(alive,0,1).

action(sail).
action(go_aboard(A)) :- obj(A).
action(get_off(A)) :- obj(A).

executable(sail, [is_in(man) eq 0]).
executable(go_aboard(A), [boat_at eq is_in(A)]) :-
   obj(A).
executable(get_off(A), [is_in(A) eq 0]) :-
   obj(A).

causes(sail, boat_at eq 1, [boat_at eq 2]).
causes(sail, boat_at eq 2, [boat_at eq 1]).
causes(go_aboard(A), is_in(A) eq 0, []) :-
   obj(A).
causes(get_off(A), is_in(A) eq boat_at^(-1), []) :-
   obj(A).

caused([is_in(A) eq 0, is_in(B) eq 0], alive eq 0) :-
   obj(A), obj(B), diff(A,B,man).
caused([is_in(wolf) eq is_in(goat),
      is_in(man) neq is_in(wolf)], alive eq 0).
```

https://doi.org/10.1017/S1471068410000013 Published online by Cambridge University Press

```
caused([is_in(cabbage) eq is_in(goat),
      is_in(man) neq is_in(cabbage)], alive eq 0).

initially(is_in(A) eq 1) :- obj(A).
initially(boat_at eq 1).
initially(alive eq 1).

goal(is_in(A) eq 2) :- obj(A).
goal(alive eq 1).
```

A.6 The gas-diffusion problem: $\mathscr{B}^{MV}$ description of the instance A_4

```
room(N) :- interval(N,1,11).
gate(1,2).
gate(1,7).
gate(1,11).
gate(2,3).
gate(3,4).
gate(4,5).
gate(5,6).
gate(6,7).
gate(6,8).
gate(8,9).
gate(9,10).
gate(10,11).

fluent(contains(N),0,255) :- room(N).
fluent(is_open(X,Y),0,1) :- gate(X,Y).

action(open(X,Y)) :- gate(X,Y).
action(close(X,Y)) :- gate(X,Y).

executable(open(X,Y),L) :-
   action(open(X,Y)),
   findall((is_open(X,Z) eq 0), gate(X,Z),L1),
   findall((is_open(Z,X) eq 0), gate(Z,X),L2),
   findall((is_open(Y,Z) eq 0), (gate(Y,Z),neq(Z,X)),L3),
   findall((is_open(Z,Y) eq 0), (gate(Z,Y),neq(Z,X)),L4),
   append(L1,L2,La),append(L3,L4,Lb),append(La,Lb,L).
executable(close(X,Y), [is_open(X,Y) eq 1]) :-
   action(close(X,Y)).

causes(open(X,Y),
      contains(Y) eq (contains(X)^(-1)+contains(Y)^(-1))/2,
      []) :-
   action(open(X,Y)).
causes(open(X,Y),
      contains(X) eq (contains(X)^(-1)+contains(Y)^(-1))/2,
      []) :-
   action(open(X,Y)).
causes(open(X,Y), is_open(X,Y) eq 1, []) :-
   action(open(X,Y)).
causes(close(X,Y), is_open(X,Y) eq 0, []) :-
   action(close(X,Y)).

initially(is_open(X,Y) eq 0) :- gate(X,Y).
initially(contains(10) eq 128).
initially(contains(3) eq 128).
initially(contains(A) eq 0) :- room(A), diff(A,3,10).

goal(contains(1) gt 50).
```

References

Apt, K. R. 2003. *Principles of Constraint Programming*. Cambridge University Press.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

BARAL, C., SON, T. AND TUAN, L.-C. 2002. A transition function based characterization of actions with delayed and continuous effects. In *Principles and Practice of Knowledge Representation and Reasoning*, Morgan Kaufmann, 291–302.

CASEAU, Y., JOSSET, F.-X. AND LABURTHE, F. 2002. CLAIRE: Combining sets, search and rules to better express algorithms. *Theory and Practice of Logic Programming 2* (6), 769–805.

DOVIER, A., FORMISANO, A. AND PONTELLI, E. 2005. A comparison of CLP(FD) and ASP solutions to NP-complete problems. In *Proc. of ICLP 2008*, M. Gabbrielli and G. Gupta, Eds. Lecture Notes in Computer Science, vol. 3668. Springer Verlag, 67–82.

DOVIER, A., FORMISANO, A. AND PONTELLI, E. 2007. An experimental comparison of constraint logic programming and answer set programming. In *Proc. of the Twenty-Second AAAI Conference on Artificial Intelligence*. AAAI Press, Vancouver, British Columbia, Canada, 1622–1625.

DOVIER, A., FORMISANO, A. AND PONTELLI, E. 2009a. An empirical study of CLP and ASP solutions of combinatorial problems. *Journal of Experimental & Theoretical Artificial Intelligence 21* (2) (June), 79–121.

DOVIER, A., FORMISANO, A. AND PONTELLI, E. 2009b. Representing multi-agent planning in CLP. In *Logic Programming and Non-monotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14–18, 2009, Proceedings*, E. Erdem, F. Lin, and T. Schaub, Eds. Lecture Notes in Computer Science, vol. 5753. Springer, 423–429.

EITER, T., FABER, W., LEONE, N., PFEIFER, G. AND POLLERES, A. 2004. A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM Transactions on Computational Logic 5* (2) (April), 206–263.

GABALDON, A. 2002. Non-Markovian control in the situation calculus. In *Proc. of the Eighteenth National Conference on Artificial Intelligence*, R. Dechter, M. Kearns, and R. S. Sutton, Eds. American Association for Artificial Intelligence, AAAI Press, Menlo Park, California, 519–524.

GEBSER, M., KAUFMANN, B., NEUMANN, A. AND SCHAUB, T. 2007. *Clasp:* A conflict-driven answer set solver. In *Logic Programming and Non-Monotonic Reasoning*, C. Baral, G. Brewka, and J. S. Schlipf, Eds. Springer Verlag, 260–265.

GELFOND, M. AND LIFSCHITZ, V. 1998. Action languages. *Electronic Transactions on Artificial Intelligence 2*, 193–210.

GIUNCHIGLIA, E., LEE, J., LIFSCHITZ, V., MCCAIN, N. AND TURNER, H. 2004a. Non-monotonic causal theories. *Artificial Intelligence 153* (1–2), 49–104.

GIUNCHIGLIA, E., LIERLER, Y. AND MARATEA, M. 2004b. SAT-based answer set programming. In *AAAI-04*, D. L. McGuinness and G. Ferguson, Eds. The MIT Press, 61–66.

HOFFMANN, M. J., PORTEOUS, J. AND SEBASTIA, L. 2004. Ordered landmarks in planning. *Journal of Artificial Intelligence Research 22*, 215–278.

JAFFAR, J. and MAHER, M. 1994. Constraint logic programming: A survey. *Journal of Logic Programming 19/20*, 503–581.

LEE, J. AND LIFSCHITZ, V. 2003. Describing additive fluents in action language C+. In *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9–15, 2003*, G. Gottlob and T. Walsh, Eds. Morgan Kaufmann, 1079–1084.

LEVESQUE, H. J., PIRRI, F. AND REITER, R. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming 31* (1–3), 59–83.

LEVESQUE, H. J., PIRRI, F. AND REITER, R. 1998. Foundations for the situation calculus. *Electronic Transactions on Artificial Intelligence 2*, 159–178.

LIFSCHITZ, V. 1999. Answer set planning. In *Proc. of the 16th Intl. Conference on Logic Programming*, D. de Schreye, Ed. MIT Press, 23–37.

Lin, F. and Zhao, Y. 2004. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence 157* (1–2), 115–137.

Lopez, A. and Bacchus, F. 2003. Generalizing Graphplan by formulating planning as a CSP. In *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9–15, 2003*, G. Gottlob and T. Walsh, Eds. Morgan Kaufmann, 954–960.

Marriott, K. and Stuckey, P. J. 1998. *Programming with Constraints*. The MIT Press.

McCarthy, J. 1998. Elephant 2000 – A programming language based on speech acts February 1st 2010. [Online] URL: `www.formal.stanford.edu/jmc`

Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press, Bradford Books, Cambridge, MA.

Simons, P. 2000. *Extending and Implementing the Stable Model Semantics*. Ph.D. thesis, Helsinki University of Technology.

Son, T. C., Baral, C. and McIlraith, S. A. 2001. Planning with different forms of domain-dependent control knowledge – An answer set programming approach. In *Logic Programming and Non-monotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17–19, 2001, Proceedings*, T. Eiter, W. Faber, and M. Truszczyński, Eds. Lecture Notes in Computer Science, vol. 2173. Springer, 226–239.

Son, T. C., Tu, P. H. and Baral, C. 2007. Reasoning and planning with sensing actions, incomplete information, and static causal laws using answer set programming. *Theory and Practice of Logic Programming 7* (4), 377–450.

Thielscher, M. 1999. From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. In *Artificial Intelligence 111* (1–2), 277–299.

Thielscher, M. 2002a. Reasoning about actions with CHRs and finite domain constraints. *Lecture Notes in Computer Science 2401*, 70–84.

Thielscher, M. 2002b. Pushing the envelope: Programming reasoning agents. In *AAAI Workshop on Cognitive Robotics*, C. Baral and S. McIlraith, Eds. AAAI Press.

Tu, P. H., Son, T. C. and Pontelli, E. 2007. CPP: A constraint logic programming based planner with preferences. In *Logic Programming and Non-monotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, C. Baral, G. Brewka, and J. S. Schlipf, Eds. Lecture Notes in Computer Science, vol. 4483. Springer, 290–296.

Vidal, V. and Geffner, G. 2006. Branching and pruning: An optimal temporal POCL planner based in constraint programming. *Artificial Intelligence 170*, 298–335.