Reasoning about actions with CHRs and Finite Domain Constraints

Michael Thielscher*

Dresden University of Technology
mit@inf.tu-dresden.de

Abstract. We present a CLP-based approach to reasoning about actions in the presence of incomplete states. Constraints expressing negative and disjunctive state knowledge are processed by a set of special Constraint Handling Rules. In turn, these rules reduce to standard finite domain constraints when handling variable arguments of single state components. Correctness of the approach is proved against the general action theory of the Fluent Calculus. The constraint solver is used as the kernel of a high-level programming language for agents that reason and plan. Experiments have shown that the constraint solver exhibits excellent computational behavior and scales up well.

1 Introduction

One of the most challenging and promising goals of Artificial Intelligence research is the design of autonomous agents, including robots, that explore partially known environments and that are able to act sensibly under incomplete information. To attain this goal, the paradigm of Cognitive Robotics [5] is to endow agents with the high-level cognitive capabilities of reasoning and planning: Exploring their environment, agents need to reason when they interpret sensor information, memorize it, and draw inferences from combined sensor data. Acting under incomplete information, agents employ their reasoning facilities to ensure that they are acting cautiously, and they plan ahead some of their actions with a specific goal in mind. To this end, intelligent agents form a mental model of their environment, which they constantly update to reflect the changes they have effected and the sensor information they have acquired.

Having agents maintain an internal world model is necessary if we want them to choose their actions not only on the basis of the current status of their sensors but also by taking into account what they have previously observed or done. Moreover, the ability to reason about sensor information is necessary if properties of the environment can only indirectly be observed and require the agent to combine observations made at different stages. The ability to plan allows an agent to first calculate the effect of different action sequences in order to help it choosing one that is appropriate under the current circumstances.

* Parts of the work reported in this paper have been carried out while the author was a visiting researcher at the University of New South Wales in Sydney, Australia.

While standard programming languages such as Java do not provide general reasoning facilities for agents, logic programming constitutes the ideal paradigm for designing agents that are capable of reasoning about their actions [9]. Examples of existing LP-systems deriving from general action theories are GOLOG [6, 8], based on the Situation Calculus [7], or the robot control language developed in [10], based on the Event Calculus [4]. However, a disadvantage of both these systems is that knowledge of the current state is represented indirectly via the initial conditions and the actions which the agent has performed up to a point. As a consequence, each time a condition is evaluated in an agent program, the entire history of actions is involved in the computation. This requires ever increasing computational effort as the agent proceeds, so that this concept does not scale up well to long-term agent control [13].

An explicit state representation being a fundamental concept in the Fluent Calculus [11], this representation formalism offers an alternative theory as the formal underpinnings for a high-level agent programming method. In this paper, we present a CLP approach to reasoning about actions which implements the Fluent Calculus. Incomplete states are represented as *open* lists of state properties, that is, lists with a variable tail. Negative and disjunctive state knowledge is encoded by *constraints*. We present a set of so-called Constraint Handling Rules (CHRs) [2] for combining and simplifying these constraints. In turn, these rules reduce to standard finite domain constraints when handling variable arguments of single state components. Based on their declarative interpretation, our CHRs are verified against the foundational axioms of the Fluent Calculus. The constraint solver is used as the kernel of the high-level programming language FLUX (for: *Fluent Executor*) which allows the design of intelligent agents that reason and plan on the basis of the Fluent Calculus [12]. Studies have shown that FLUX and in particular the constraint solver scale up well [13].

The paper is organized as follows: In Section 2, we recapitulate the basic notions and notations of the Fluent Calculus as the underlying theory for our CLP-based approach to reasoning about actions. In Section 3, we present a set of CHRs for constraints expressing negative and disjunctive state knowledge, and we prove their correctness wrt. the foundational axioms of the Fluent Calculus. In Section 4, we embed the constraint solver into a logic program for reasoning about actions, which, too, is verified against the underlying semantics of the Fluent Calculus. In Section 5, we give a summary of studies showing the computational merits of our approach. We conclude in Section 6.

The constraint solver, the general FLUX system, the example agent program, and the accompanying papers all are available for download at our web site `http://fluxagent.org`.

2 Reasoning about states with the Fluent Calculus

Throughout the paper, we will use the following example of an agent in a dynamic environment: Consider a cleaning robot which, in the evening, has to empty the waste bins in the alley and rooms of the floor of an office building. The robot

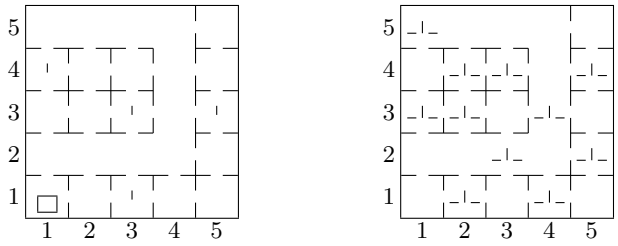

Fig. 1. Layout of a sample office floor and a scenario in which four offices are occupied. In the right hand side the locations are depicted in which the robot senses light.

shall not, however, disturb anyone working in late. The robot is equipped with a light sensor which is activated whenever the robot is adjacent to a room that is occupied, without being able to tell which direction the light comes from. An instance of this problem is depicted in Fig. 1. The task is to program the cleaning robot so as to empty as many bins as possible without risking to burst into an occupied office. This problem illustrates two challenges raised by incomplete state knowledge: Agents have to act cautiously, and they need to interpret and logically combine sensor information acquired over time.

The Fluent Calculus is an axiomatic theory of actions that provides the formal underpinnings for agents to reason about their actions [11]. Formally, it is a many-sorted predicate logic language with four standard sorts for actions and situations (as in the Situation Calculus) and for fluents (i.e., atomic state properties) and states. For the cleaning robot domain, for example, we will use these four fluents (i.e., mappings into the sort FLUENT): $At(x, y)$, representing that the robot is at (x, y); $Facing(d)$, representing that the robot faces direction $d \in \{1, \ldots, 4\}$ (denoting, resp., north, east, south, and west); $Cleaned(x, y)$, representing that the waste bin at (x, y) has been emptied; and $Occupied(x, y)$, representing that (x, y) is occupied. We make the standard assumption of uniqueness-of-names, $UNA[At, Faces, Cleaned, Occupied]$.[1]

States are built up from fluents (as atomic states) and their conjunction, using the function $\circ$: STATE $\times$ STATE $\mapsto$ STATE along with the constant $\emptyset$: STATE denoting the empty state. For example, the term $At(1,1) \circ (Facing(1) \circ z)$ represents a state in which the robot is in square $(1, 1)$ facing north while other fluents may hold, too, summarized in the variable sub-state z.

A fundamental notion is that of a fluent to *hold* in a state. Fluent f is said to hold in state z just in case z can be decomposed into two states one of which is the singleton f. For notational convenience, we introduce the macro $Holds(f, z)$ as an abbreviation for the corresponding equational formula:

$$Holds(f, z) \stackrel{\text{def}}{=} (\exists z')\, z = f \circ z' \tag{1}$$

This definition is accompanied by the following foundational axioms of the Fluent

[1] $UNA[h_1, \ldots, h_n] \stackrel{\text{def}}{=} \bigwedge_{i \neq j} h_i(\vec{x}) \neq h_j(\vec{y}) \wedge \bigwedge_i [h_i(\vec{x}) = h_i(\vec{y}) \supset \vec{x} = \vec{y}]$.

Calculus, which constitute a special theory of equality of state terms.

Definition 1. Assume a signature which includes the sorts FLUENT and STATE such that FLUENT is a sub-sort of STATE, along with the functions $\circ, \emptyset$ of sorts as above. The *foundational axioms* Σ_{state} *of the Fluent Calculus* are:[2]

1. Associativity, commutativity, idempotence, and unit element,

$$\begin{array}{ll} (z_1 \circ z_2) \circ z_3 = z_1 \circ (z_2 \circ z_3) & z \circ z = z \\ z_1 \circ z_2 = z_2 \circ z_1 & z \circ \emptyset = z \end{array} \tag{2}$$

2. Empty state axiom,

$$\neg Holds(f, \emptyset) \tag{3}$$

3. Irreducibility and decomposition,

$$Holds(f_1, f) \supset f_1 = f \tag{4}$$

$$Holds(f, z_1 \circ z_2) \supset Holds(f, z_1) \vee Holds(f, z_2) \tag{5}$$

4. State equivalence and existence of states,

$$(\forall f)\,(Holds(f, z_1) \equiv Holds(f, z_2)) \supset z_1 = z_2 \tag{6}$$

$$(\forall P)(\exists z)(\forall f)\,(Holds(f, z) \equiv P(f)) \tag{7}$$

where P is a second-order predicate variable of sort FLUENT.

Axioms (2) essentially characterize "$\circ$" as the union operation with $\emptyset$ as the empty set of fluents. Associativity allows us to omit parentheses in nested applications of "$\circ$". Axiom (6) says that two states are equal if they contain the same fluents, and second-order axiom (7) guarantees the existence of a state for any combination of fluents.

The foundational axioms can be used to reason about incomplete state specifications and acquired sensor information. Consider, e.g., the definition of what it means for our cleaning robot to sense light in a square (x, y) in some state z:

$$\begin{array}{l} LightPerception(x, y, z) \equiv \\ \quad Holds(Occupied(x+1, y), z) \vee Holds(Occupied(x, y+1), z) \\ \quad \vee Holds(Occupied(x-1, y), z) \vee Holds(Occupied(x, y-1), z) \end{array} \tag{8}$$

Suppose that at the beginning the robot only knows that the following locations are not occupied: its home $(1, 1)$ (axiom (10) below), the squares in the alley (axiom (11) below), and any location outside the boundaries of the office floor (axioms (12),(13) below). Suppose further that the robot already went to clean $(1, 1)$, $(1, 2)$, and $(1, 3)$, sensing light in the last square only (c.f. Fig. 1). Thus the current state, ζ, is known to be

$$\zeta = At(1, 3) \circ Facing(1) \circ Cleaned(1, 1) \circ Cleaned(1, 2) \circ Cleaned(1, 3) \circ z \tag{9}$$

[2] Throughout the paper, free variables in formulas are assumed universally quantified. Variables of sorts FLUENT, STATE, ACTION, and SIT shall be denoted by the letters f, z, a, and s, resp. The function "$\circ$" is written in infix notation.

for some z, along with the following axioms:

$$\neg Holds(Occupied(1,1),z) \tag{10}$$
$$\neg Holds(Occupied(1,5),z) \wedge \ldots \wedge \neg Holds(Occupied(1,2),z) \tag{11}$$
$$(\forall x)\,(\neg Holds(Occupied(x,0),z) \wedge \neg Holds(Occupied(x,6),z)) \tag{12}$$
$$(\forall y)\,(\neg Holds(Occupied(0,y),z) \wedge \neg Holds(Occupied(6,y),z)) \tag{13}$$
$$\neg LightPerception(1,1,\zeta) \wedge \neg LightPerception(1,2,\zeta) \tag{14}$$
$$LightPerception(1,3,\zeta) \tag{15}$$

From (14) and (8), $\neg Holds(Occupied(2,1),\zeta) \wedge \neg Holds(Occupied(2,2),\zeta)$. With regard to (9), the foundational axioms of decomposition, (5), and irreducibility, (4), along with uniqueness-of-names imply

$$\neg Holds(Occupied(2,1),z) \wedge \neg Holds(Occupied(2,2),z) \tag{16}$$

On the other hand, (15) and (8) imply

$$Holds(Occupied(2,3),\zeta) \vee Holds(Occupied(1,4),\zeta)$$
$$\vee Holds(Occupied(0,3),\zeta) \vee Holds(Occupied(1,2),\zeta)$$

As above, with regard to (9), the foundational axioms of decomposition and irreducibility along with uniqueness-of-names imply

$$Holds(Occupied(2,3),z) \vee Holds(Occupied(1,4),z)$$
$$\vee Holds(Occupied(0,3),z) \vee Holds(Occupied(1,2),z)$$

From (13) and (11) it follows that

$$Holds(Occupied(2,3),z) \vee Holds(Occupied(1,4),z) \tag{17}$$

This disjunction cannot be reduced further, that is, at this stage the robot does not know whether the light in $(1,3)$ comes from office $(2,3)$ or $(1,4)$ (or both, for that matter). Suppose, therefore, the cautious robot goes back, turns east, and continues with cleaning $(2,2)$, which it knows to be unoccupied according to (16). Sensing no light there (c.f. Fig. 1), the new state ζ' is known to be $At(2,2) \circ Facing(2) \circ Cleaned(1,1) \circ Cleaned(1,2) \circ Cleaned(1,3) \circ Cleaned(2,2) \circ z$ for some z such that (10)–(13),(16),(17) along with $\neg LightPerception(2,2,\zeta')$. From (8), $\neg Holds(Occupied(2,3),\zeta')$; hence, decomposition and irreducibility along with uniqueness-of-names imply $\neg Holds(Occupied(2,3),z)$; hence by (17), $Holds(Occupied(1,4),z)$, that is, now the robot knows that $(1,4)$ is occupied.

3 Solving State Constraints

Based on the axiomatic foundation of the Fluent Calculus, in the following we develop a provably correct CLP-approach for reasoning about incomplete state specifications. To begin with, incomplete states are encoded by *open* lists of fluents (possibly containing variables):

```
Z = [F1,...,Fk | _]
```

It is assumed that the arguments of fluents are encoded by natural or rational numbers, which enables the use of a standard arithmetic solver for constraints on partially known arguments. Negative and disjunctive state knowledge is expressed by the following *state constraints*:

constraint	semantics
`not_holds(F,Z)`	$\neg Holds(f, z)$
`not_holds_all(F,Z)`	$(\forall \vec{x})\, \neg Holds(f, z)$, where $\vec{x}$ variables in f
`or([F1,...,Fn],Z)`	$\bigvee_{i=1}^{n} Holds(f_i, z)$

The state constraints have been carefully designed so as to be sufficiently expressive while allowing for efficient constraint solving. An auxiliary constraint `duplicate_free(Z)` is used to stipulate that a list of fluents contains no multiple occurrences, thus reflecting the foundational axiom of idempotence of "$\circ$" in the Fluent Calculus. As an example, the following clause encodes the specification of state ζ of Section 2 (c.f. (9) and (8), resp.):

```
zeta(Zeta) :-
  Zeta = [at(1,3),facing(1),cleaned(1,1),cleaned(1,2),cleaned(1,3) | Z],
  not_holds(occupied(1,1),Z),
  not_holds(occupied(1,5),Z), ..., not_holds(occupied(1,2),Z),
  not_holds_all(occupied(_,0),Z), not_holds_all(occupied(_,6),Z),
  not_holds_all(occupied(0,_),Z), not_holds_all(occupied(6,_),Z),
  light_perception(1,1,false,Zeta), light_perception(1,2,false,Zeta),
  light_perception(1,3,true,Zeta),
  duplicate_free(Zeta).

light_perception(X,Y,Percept,Z) :-
  XE#=X+1, XW#=X-1, YN#=Y+1, YS#=Y-1,
  ( Percept=false,
      not_holds(occupied(XE,Y),Z), not_holds(occupied(X,YN),Z),
      not_holds(occupied(XW,Y),Z), not_holds(occupied(X,YS),Z)
    ; Percept=true, or([occupied(XE,Y),occupied(X,YN),
                        occupied(XW,Y),occupied(X,YS)],Z) ).
```

Here and in the following we employ a standard constraint domain, namely, that of finite domains, which includes arithmetic constraints over rational numbers using the equality, inequality, and ordering predicates `#=`,`#<`,`#>` along with the standard functions `+`,`-`,`*`; range constraints (written `X::[`a`..`b`]`); and logical combinations using `#/\` and `#\/` for conjunction and disjunction, resp.

Our approach is based on so-called Constraint Handling Rules, which support the declarative programming of constraint solvers [2]. CHRs are of the form

```
H1,...,Hm <=> G1,...,Gk | B1,...,Bn.
```

where the *head* $H_1, \ldots, H_m$ are constraints ($m \geq 1$); the *guard* $G_1, \ldots, G_k$ are Prolog literals ($k \geq 0$); and the *body* $B_1, \ldots, B_n$ are constraints ($n \geq 0$). An

```
not_holds(_,[])         <=> true.                                    %1
not_holds(F,[F1|Z])     <=> neq(F,F1), not_holds(F,Z).               %2
not_holds_all(_,[])     <=> true.                                    %3
not_holds_all(F,[F1|Z]) <=> neq_all(F,F1), not_holds_all(F,Z).       %4

not_holds_all(F,Z) \ not_holds(G,Z)     <=> instance(G,F) | true. %5
not_holds_all(F,Z) \ not_holds_all(G,Z) <=> instance(G,F) | true. %6

duplicate_free([])    <=> true.                                      %7
duplicate_free([F|Z]) <=> not_holds(F,Z), duplicate_free(Z).        %8

neq(F,F1)     :- or_neq(exists,F,F1).
neq_all(F,F1) :- or_neq(forall,F,F1).

or_neq(Q,Fx,Fy) :- Fx =.. [F|ArgX], Fy =.. [G|ArgY],
                   ( F=G -> or_neq(Q,ArgX,ArgY,D), call(D) ; true ).

or_neq(_,[],[],(0#\=0)).
or_neq(Q,[X|X1],[Y|Y1],D) :-
  or_neq(Q,X1,Y1,D1),
  ( Q=forall, var(X) ->
       ( binding(X,X1,Y1,YE) -> D=((Y#\=YE)#\/D1) ; D=D1 ) ;
    D=((X#\=Y)#\/D1) ).

binding(X,[X1|ArgX],[Y1|ArgY],Y) :- X==X1 -> Y=Y1
                                           ; binding(X,ArgX,ArgY,Y).
```

Fig. 2. CHRs for negation and multiple occurrences. The notation `H1 \ H2 <=> G | B` is an abbreviation for `H1,H2 <=> G | H1,B`.

empty guard is omitted; the empty body is denoted by *True*. The declarative interpretation of a CHR is given by the formula

$$(\forall \vec{x})\,(\,G_1 \wedge \ldots \wedge G_k \;\supset\; [H_1 \wedge \ldots \wedge H_m \;\equiv\; (\exists \vec{y})\,(B_1 \wedge \ldots \wedge B_n)]\,)$$

where $\vec{x}$ are the variables in both guard and head and $\vec{y}$ are the variables which additionally occur in the body. The procedural interpretation of a CHR is given by a transition in a constraint store: If the head can be matched against elements of the constraint store and the guard can be derived, then the constraints of the head are replaced by the constraints of the body.

3.1 Handling Negation

Fig. 2 depicts the first part of the constraint solver, which contains the CHRs and auxiliary clauses for the two negation constraints and the auxiliary constraint

on multiple occurrences. In the following, these rules are proved correct wrt. the foundational axioms of the Fluent Calculus.

To begin with, consider the auxiliary clauses, which define a finite domain constraint that expresses the inequality of two fluent terms. By `or_neq` inequality of two fluents with arguments $\texttt{ArgX} = [\texttt{X1}, \ldots, \texttt{Xn}]$ and $\texttt{ArgY} = [\texttt{Y1}, \ldots, \texttt{Yn}]$ is decomposed into the arithmetic constraint $\texttt{X1} \neq \texttt{Y1} \vee \ldots \vee \texttt{Xn} \neq \texttt{Yn}$. Two cases are distinguished depending on whether the variables in the first term are existentially or universally quantified. In the latter case, a simplified disjunction is generated, where the variables of the first fluent are discarded while possibly giving rise to dependencies among the arguments of the second fluent. E.g., $\texttt{neq\_all}(\texttt{f}(\_, \texttt{a}, \_), \texttt{f}(\texttt{U}, \texttt{V}, \texttt{W}))$ reduces to $\texttt{a} \neq \texttt{V}$ and $\texttt{neq\_all}(\texttt{f}(\texttt{X}, \texttt{X}, \texttt{X}), \texttt{f}(\texttt{U}, \texttt{V}, \texttt{W}))$ reduces to $\texttt{U} \neq \texttt{V} \vee \texttt{V} \neq \texttt{W}$. To formally capture the universal quantification, we define the notion of a *schematic* fluent $f = h(\vec{x}, \vec{r})$ where $\vec{x}$ denotes the variable arguments in f. The following observation implies the correctness of the constraints generated by the auxiliary clauses.

Observation 2 *Consider a set $\mathcal{F}$ of functions into sort* FLUENT. *Consider a fluent $f_1 = g(r_1, \ldots, r_m)$, a schematic fluent $f_2 = g(x_1, \ldots, x_k, r_{k+1}, \ldots, r_m)$, and a fluent $f = h(t_1, \ldots, t_n)$. Then*

1. *if $g \neq h$, then $UNA[\mathcal{F}] \models f_1 \neq f$ and $UNA[\mathcal{F}] \models (\forall \vec{x})\, f_2 \neq f$;*
2. *if $g = h$, then $m = n$ and $UNA[\mathcal{F}] \models f_1 \neq f \equiv r_1 \neq t_1 \vee \ldots \vee r_m \neq t_n$ and $UNA[\mathcal{F}] \models (\forall x)\,(f_2 \neq f \equiv r_{k+1} \neq t_{k+1} \vee \ldots \vee r_m \neq t_n \vee \bigvee_{x_i \equiv x_j}^{i \neq j} t_i \neq t_j)$.*

CHRs 1–4 for negation constraints can then be justified by the foundational axioms of the Fluent Calculus, as the following proposition shows.

Proposition 3. *Σ_{state} entails,*

1. *$\neg Holds(f, \emptyset)$; and*
2. *$\neg Holds(f, f_1 \circ z) \equiv f \neq f_1 \wedge \neg Holds(f, z)$.*

Likewise, if $f = g(\vec{x}, \vec{r})$ is a schematic fluent, then Σ_{state} entails,

3. *$(\forall \vec{x})\, \neg Holds(f, \emptyset)$; and*
4. *$(\forall \vec{x})\, \neg Holds(f, f_1 \circ z) \equiv (\forall \vec{x})\, f \neq f_1 \wedge (\forall \vec{x})\, \neg Holds(f, z)$.*

Proof. Claim 1 follows by the empty state axiom. Regarding claim 2 we prove that $Holds(f, f_1 \circ z) \equiv f = f_1 \vee Holds(f, z)$. The "$\supset$" direction follows by foundational axioms (5) and (4). For the "$\subset$" direction, if $f = f_1$, then $f_1 \circ z = f \circ z$, hence $Holds(f, f_1 \circ z)$. Likewise, if $Holds(f, z)$, then $z = f \circ z'$ for some z', hence $f_1 \circ z = f_1 \circ f \circ z'$, hence $Holds(f, f_1 \circ z)$. The proof of 3 and 4 is similar.

Correctness of CHRs 5 and 6, which remove subsumed negative constraints, is obvious as $(\forall \vec{x})\, \neg Holds(f_1, z)$ implies $\neg Holds(f_2, z)$ and $(\forall \vec{y})\, \neg Holds(f_2, z)$, resp., for a schematic fluent f_1 and a fluent f_2 such that $f_1\theta = f_2$ for some θ. Finally, CHRs 7 and 8 for the auxiliary constraint on multiple occurrences are correct since the empty list contains no duplicate elements and a non-empty list contains no duplicates iff the head does not occur in the tail and the tail itself is free of duplicates.

```
or([F],Z) <=> F\=eq(_,_) | holds(F,Z).                                  %9
or(V,Z)   <=> \+ ( member(F,V),F\=eq(_,_) )                            %10
              | or_and_eq(V,D), call(D).
or(V,[])  <=> member(F,V,W), F\=eq(_,_) | or(W,[]).                    %11

or(V,Z) <=> member(eq(X,Y),V),                                         %12
            or_neq(exists,X,Y,D), \+ call(D) | true.
or(V,Z) <=> member(eq(X,Y),V,W),                                       %13
            \+ (and_eq(X,Y,D), call(D)) | or(W,Z).

not_holds(F,Z) \ or(V,Z)     <=> member(G,V,W), F==G | or(W,Z).         %14
not_holds_all(F,Z) \ or(V,Z) <=> member(G,V,W),                        %15
                                    instance(G,F) | or(W,Z).

or(V,[F|Z])   <=> or(V,[],[F|Z]).                                      %16
or(V,W,[F|Z]) <=> member(F1,V,V1), \+ F\=F1                            %17
                  | ( F1==F -> true ; F1=..[_|ArgX], F=..[_|ArgY],
                    or(V1, [eq(ArgX,ArgY),F1|W], [F|Z]) ).
or(V,W,[_|Z]) <=> append(V,W,V1), or(V1,Z).                            %18

and_eq([],[],(0#=0)).
and_eq([X|X1],[Y|Y1],D) :- and_eq(X1,Y1,D1), D=((X#=Y)#/\D1).

or_and_eq([],(0#\=0)).
or_and_eq([eq(X,Y)|Eq],(D1#\/D2)) :- or_and_eq(Eq,D1), and_eq(X,Y,D2).

member(X,[X|T],T).
member(X,[H|T],[H|T1]) :- member(X,T,T1).
```

Fig. 3. CHRs for the disjunctive constraint.

3.2 Handling Disjunction

Fig. 3 depicts the second part of the constraint solver, which contains the CHRs and auxiliary clauses for the disjunctive constraint. Internally, a disjunctive constraint may contain, besides fluents, atoms of the form $\mathtt{eq}(\vec{r},\vec{t})$ where $\vec{r}$ and $\vec{t}$ are lists of equal length. Such a general disjunction $\mathtt{or}([\delta_1,\ldots,\delta_k],\mathtt{Z})$ means

$$\bigvee_{i=1}^{k} \begin{cases} \mathit{Holds}(\mathtt{F},\mathtt{Z}) & \text{if } \delta_i \text{ is fluent } \mathtt{F} \\ \vec{x}=\vec{y} & \text{if } \delta_i \text{ is } \mathtt{eq}(\vec{x},\vec{y}) \end{cases} \tag{18}$$

CHR 9 in Fig. 3 simplifies singleton disjunctions according to (18). CHR 10 reduces a pure equational disjunction to a finite domain constraint. Its correctness follows directly from (18), too. CHR 11 simplifies a disjunction applied to the empty state. It is justified by the empty state axiom, (3), which entails

$$[\mathit{Holds}(f,\emptyset) \vee \Psi] \equiv \Psi$$

for any formula Ψ. CHRs 12 and 13 apply to disjunctions which include a decided equality. If the equality is true, then the entire disjunction is true, else if the equality is false, then the disjunction gets simplified. Correctness follows from

$$\vec{x} = \vec{y} \supset [\vec{x} = \vec{y} \vee \Psi \equiv \mathit{True}] \quad \text{and} \quad \vec{x} \neq \vec{y} \supset [\vec{x} = \vec{y} \vee \Psi \equiv \Psi]$$

The next two CHRs, 14 and 15, constitute unit resolution steps. They are justified by

$$\neg \mathit{Holds}(f, z) \supset [\mathit{Holds}(f, z) \vee \Psi \equiv \Psi]$$
$$(\forall \vec{x})\, \neg \mathit{Holds}(f_1, z) \supset [\mathit{Holds}(f_2, z) \vee \Psi \equiv \Psi]$$

given that $f_1\theta = f_2$ for some θ,

Finally, CHRs 16-18 in Fig. 3 are used to propagate a disjunction through a non-variable state. Given the constraint $\mathtt{or}(\delta, [\mathtt{F} \,|\, \mathtt{Z}])$, the basic idea is to infer all possible bindings of `F` with fluents in δ. The rules use the auxiliary constraint $\mathtt{or}(\delta, \gamma, [\mathtt{F} \,|\, \mathtt{Z}])$ with the intended semantics $\mathtt{or}(\delta, [\mathtt{F} \,|\, \mathtt{Z}]) \vee \mathtt{or}(\gamma, \mathtt{Z})$, that is, δ contains the fluents that have not yet been evaluated against the head `F` of the state list, while γ contains those fluents that have been evaluated. As an example, consider the constraint `or([f(a,V),f(W,b)],[f(X,Y)|Z])` which, upon being processed, yields

```
or([f(a,V),f(W,b),eq([a,V],[X,Y]),eq([W,b],[X,Y])], Z)
```

The rules are justified by the following proposition.

Proposition 4. *Consider a Fluent Calculus signature with a set $\mathcal{F}$ of functions into sort* FLUENT. *Foundational axioms Σ_{state} and uniqueness-of-names UNA$[\mathcal{F}]$ entail each of the following:*

1. $\Psi \equiv [\Psi \vee \bigvee_{i=1}^{0} \Psi_i]$;
2. $[\mathit{Holds}(f(\vec{x}), f(\vec{y}) \circ z) \vee \Psi_1] \vee \Psi_2 \equiv \Psi_1 \vee [\vec{x} = \vec{y} \vee \mathit{Holds}(f(\vec{x}), z) \vee \Psi_2]$;
3. *if* $\bigwedge_{i=1}^{n} f_i \neq f$, *then* $[\bigvee_{i=1}^{n} \mathit{Holds}(f_i, f \circ z) \vee \Psi] \equiv [\bigvee_{i=1}^{n} \mathit{Holds}(f_i, z) \vee \Psi]$.

Proof. Claim 1 is obvious. Claims 2 and 3 follow from the foundational axioms of decomposition and irreducibility.

This completes the constraint solver. As an example, running the specification from the beginning of this section results in

```
?- zeta(Zeta).

Zeta=[at(1,3),facing(1),cleaned(1,1),cleaned(1,2),cleaned(1,3) | Z]
Constraints:
or([occupied(1,4),occupied(2,3)], Z)
...
```

Adding the information that there is no light in $(2, 2)$, the system is able to infer that $(4, 1)$ must be occupied:

```
?- zeta(Zeta), light_perception(2,2,Zeta,false).

Zeta=[at(1,3),facing(1),cleaned(1,1),cleaned(1,2),cleaned(1,3),
      occupied(4,1) | Z]
Constraints:
not_holds(occupied(2,3), Z)
...
```

While the FLUX constraint system is sound, it may not enable agents to draw all conclusions that follow logically from a state specification because the underlying standard arithmetic solver trades completeness for efficiency. This is due to the fact that a conjunction or a disjunction is evaluated only if one of its atoms has been decided. The advantage of so doing is that the computational effort of evaluating a new constraint is *linear* in the size of the constraint store while a complete solver would require exponential time for this task.

4 Reasoning About Actions

In this section, we embed our constraint solver into a logic program for reasoning about the effects of actions based on the Fluent Calculus. Generalizing previous approaches [3, 1], the Fluent Calculus provides a solution to the fundamental frame problem in the presence of incomplete states [11]. The solution is based on a rigorously axiomatic characterizations of addition and removal of (finitely many) fluents from incompletely specified states. The following definition introduces the macro equation $z_1 - \tau = z_2$ with the intended meaning that state z_2 is state z_1 minus the fluents in the finite state τ. The compound macro $z_2 = (z_1 - \vartheta^-) + \vartheta^+$ means that state z_2 is state z_1 minus the fluents in ϑ^- plus the fluents in ϑ^+:

$$
\begin{aligned}
&z_1 - \emptyset = z_2 \stackrel{\text{def}}{=} z_2 = z_1\\
&z_1 - f = z_2 \stackrel{\text{def}}{=} (z_2 = z_1 \vee z_2 \circ f = z_1) \wedge \neg \mathit{Holds}(f, z_2)\\
&z_1 - (f_1 \circ f_2 \circ \ldots \circ f_n) = z_2 \stackrel{\text{def}}{=} (\exists z)\,(z = z_1 - f \wedge z_2 = z - (f_2 \circ \ldots \circ f_n))\\
&(z_1 - \vartheta^-) + \vartheta^+ = z_2 \stackrel{\text{def}}{=} (\exists z)\,(z = z_1 - \vartheta^- \wedge z_2 = z \circ \vartheta^+)
\end{aligned}
$$

where both ϑ^+, ϑ^- are finitely many FLUENT terms connected by "$\circ$". The crucial item is the second one, which defines removal of a single fluent f using a case distinction: Either $z_1 - f$ equals z_1 (which applies in case $\neg \mathit{Holds}(f, z_1)$), or $z_1 - f$ plus f equals z_1 (which applies in case $\mathit{Holds}(f, z_1)$).

Fig. 4 depicts a set of clauses which encode the solution to the frame problem on the basis of the constraint solver for the Fluent Calculus. The program culminates in the predicate $\mathit{Update}(z_1, \vartheta^+, \vartheta^-, z_2)$, by which an incomplete state z_1 is updated to z_2 according to positive and negative effects ϑ^+ and ϑ^-, resp. The first two clauses in Fig. 4 encode macro (1). Correctness of this definition follows from the foundational axioms of decomposition and irreducibility. The ternary $\mathit{Holds}(f, z, z')$ encodes $\mathit{Holds}(f, z) \wedge z' = z - f$. The following proposition implies that the definition is correct wrt. the macro definition of fluent removal, under the assumption that lists are free of duplicates.

```
holds(F,[F|_]).
holds(F,Z) :- nonvar(Z), Z=[F1|Z1], \+ F==F1, holds(F,Z1).

holds(F,[F|Z],Z).
holds(F,Z,[F1|Zp]) :- nonvar(Z), Z=[F1|Z1], \+ F==F1, holds(F,Z1,Zp).

minus(Z,[],Z).
minus(Z,[F|Fs],Zp) :- ( \+ not_holds(F,Z) -> holds(F,Z,Z1) ;
                        \+ holds(F,Z)     -> Z1 = Z ;
                        cancel(F,Z,Z1), not_holds(F,Z1) ),
                      minus(Z1,Fs,Zp).
plus(Z,[],Z).
plus(Z,[F|Fs],Zp) :- ( \+ holds(F,Z)     -> Z1=[F|Z] ;
                       \+ not_holds(F,Z) -> Z1=Z ;
                       cancel(F,Z,Z2), Z1=[F|Z2], not_holds(F,Z2) ),
                     plus(Z1,Fs,Zp).

update(Z1,ThetaP,ThetaN,Z2) :- minus(Z1,ThetaN,Z), plus(Z,ThetaP,Z2).
```

Fig. 4. The foundational clauses for reasoning about actions.

Proposition 5. *Axioms* $\Sigma_{state} \cup \{z = f_1 \circ z_1 \wedge \neg Holds(f_1, z_1)\}$ *entails*

$$\begin{aligned} &Holds(f, z) \wedge z' = z - f \equiv \\ &\quad f = f_1 \wedge z' = z_1 \\ &\quad \vee (\exists z'')\,(f \neq f_1 \wedge Holds(f, z_1) \wedge z'' = z_1 - f \wedge z' = f_1 \circ z'') \end{aligned}$$

Proof. Suppose $f = f_1$. If $Holds(f, z) \wedge z' = z - f$, then $z' = (f_1 \circ z_1) - f_1$ since $z = f_1 \circ z_1$; hence, $z' = z_1$ since $\neg Holds(f_1, z_1)$. Conversely, if $z' = z_1$, then $z' = (f_1 \circ z_1) - f_1 = z - f$, and $Holds(f, z)$ since $z = f_1 \circ z_1$ and $f_1 = f$.

Suppose $f \neq f_1$. If $Holds(f, z)$ and $z' = z - f$, then $Holds(f, z_1)$ and $z' = (f_1 \circ z_1) - f$; hence, there is some z'' such that $z'' = z_1 - f$ and $z' = f_1 \circ z''$. Conversely, if $Holds(f, z_1) \wedge z'' = z_1 - f \wedge z' = f_1 \circ z''$, then $Holds(f, z)$ and $z' = (f_1 \circ z_1) - f$; hence, $Holds(f, z) \wedge z' = z - f$.

Removal and addition of finitely many fluents is defined recursively. The recursive clause for `minus` says that if $\neg Holds(f, z)$ is unsatisfiable (that is, f is known to hold in z), then subtraction of f is given by the definition of the ternary *Holds* predicate. Otherwise, if $Holds(f, z)$ is unsatisfiable (that is, f is known to be false in z), then hence $z - f$ equals z. If, however, the status of the fluent is not entailed by the state specification at hand for z, then partial knowledge of f in $\Phi(z)$ may not transfer to the resulting state $z - f$ and, hence, needs to be cancelled. Consider, for example, the partial state specification

$$Holds(F(y), z) \wedge [\, Holds(F(A), z) \vee Holds(F(B), z) \,] \tag{19}$$

```
cancel(F,Z1,Z2) :- var(Z1) -> cancel(F,Z1), cancelled(F,Z1), Z2=Z1 ;
                   Z1=[G|Z], ( F\=G -> cancel(F,Z,Z3), Z2=[G|Z3]
                                     ; cancel(F,Z,Z2) ).

cancel(F,Z) \ not_holds(G, Z)     <=>              \+ F\=G | true.
cancel(F,Z) \ not_holds_all(G, Z) <=>              \+ F\=G | true.
cancel(F,Z) \ or(V,Z)             <=> member(G,V), \+ F\=G | true.

cancel(F,Z), cancelled(F,Z) <=> true.
```

Fig. 5. Auxiliary clauses and CHRs for canceling partial information about a fluent.

This formula does not entail $\mathit{Holds}(F(A), z)$ nor $\neg\mathit{Holds}(F(A), z)$. So what can be inferred about the state $z - F(A)$? Macro expansion of "$-$" implies that $\Sigma_{state} \cup \{(19)\} \cup \{z_1 = z - F(A)\}$ entails $\neg\mathit{Holds}(F(A), z_1)$. But it does not follow whether $F(y)$ holds in z_1 or whether $F(B)$ does, since

$$\begin{aligned}\Sigma_{state} \cup \{(19)\} \cup \{z_1 = z - F(A)\} \models\\ [\, y = A \supset \neg\mathit{Holds}(F(y), z_1)\,] \wedge\\ [\, y \neq A \supset \mathit{Holds}(F(y), z_1)\,] \wedge\\ [\, \neg\mathit{Holds}(F(B), z) \supset \neg\mathit{Holds}(F(B), z_1)\,] \wedge\\ [\, \mathit{Holds}(F(B), z) \supset \mathit{Holds}(F(B), z_1)\,]\end{aligned}$$

Therefore, in the clause for `miunus` all partial information concerning f in the current state z is cancelled prior to asserting that f does not hold in the resulting state. The definition of cancellation of a fluent f is given in Fig. 5. In the base case, all negative and disjunctive state information affected by f is cancelled via the constraint $\texttt{cancel}(f, z)$. The latter is resolved itself by the auxiliary constraint $\texttt{cancelled}(f, z)$, indicating the termination of the cancellation procedure. In the recursive clause for $\texttt{cancel}(f, z_1, z_2)$, each atomic, positive state information that unifies with f is cancelled.

In a similar fashion, the recursive clause for `plus` says that if $\mathit{Holds}(f, z)$ is unsatisfiable (that is, f is known to be false in z), then f is added to z; otherwise, if $\neg\mathit{Holds}(f, z)$ is unsatisfiable (that is, f is known to hold in z), then $z + f$ equals z. But if the status of the fluent is not entailed by the state specification at hand for z, then all partial information about f in z is cancelled prior to adding f to the state and asserting that f does not hold in the tail.

The definitions for `minus` and `plus` imply that a fluent to be removed or added does not hold or hold, resp., in the resulting state. Moreover, by definition cancellation does not affect the parts of the state specification which do not unify with the fluent in question. Hence, these parts continue to hold in the resulting state after the update. The correctness of this encoding of update follows from the main theorem of the Fluent Calculus, which says that the axiomatization of state update by the macros for "$-$" and "$+$" solves the frame problem [11]: A

fluent holds in the updated state just in case it either holds in the original state and is not subtracted, or it is added.

In the accompanying paper [12], it is shown how this CLP-based approach to reasoning about actions can be used as the kernel for a high-level programming method which allows to design cognitive agents that reason about their actions and plan. Thereby, agents use the concept of a state as their mental model of the world when conditioning their own behavior or when planning ahead some of their actions with a specific goal in mind. As they move along, agents constantly update their world model in order to reflect the changes they have effected. This maintaining the internal state is based on the definition of so-called state update axioms for each action, which in turn appeal to the definition of update as developed in Section 4. Thanks to the extensive reasoning facilities provided by the kernel of FLUX and in particular the constraint solver, the language allows to implement complex strategies with concise and modular agent programs.

5 Computational Behavior

Experiments have shown that FLUX scales up well. In the accompanying paper [13], we report on results with a special variant of FLUX for complete states applied to a robot control program for a combinatorial mail delivery problem. The experiments show that FLUX can compute the effects of hundreds of actions per second. The computational behavior of FLUX and the constraint solver in the presence of incomplete states has been analyzed with an agent program for the office cleaning domain, by which the robot systematically explores its partially known environment and acts cautiously under incomplete information. The results show that there is but a linear increase in the action computation cost as the knowledge of the environment grows. Notably, due to the state-based paradigm, action selection and update computation never depends on the history of actions. Therefore, FLUX scales up effortlessly to arbitrarily long sequences of actions. This result has been compared to GOLOG [6], where the curve for the computation cost suggests a polynomial increase over time [13].

6 Summary

We have presented a CLP-based approach to reasoning about actions in the presence of incomplete states based on Constraint Handling Rules and finite domain constraints. Both the constraint solver and the logic program for state update have been verified against the action theory of the Fluent Calculus. The experiments reported in [13] have shown that the constraint solver scales up well. This is particularly remarkable since the agent needs to constantly perform theorem proving tasks when conditioning its behavior on what it knows about the environment. Linear performance has been achieved due to a careful design of the state constraints supported in our approach; the restricted expressiveness makes

theorem proving computationally feasible. Future work will be to gradually extend the language, e.g., by constraints expressing exclusive disjunction, without loosing the computational merits of the approach.

References

1. Wolfgang Bibel. A deductive solution for plan generation. *New Generation Computing*, 4:115–132, 1986.
2. Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, 1998.
3. Steffen Hölldobler and Josef Schneeberger. A new deductive approach to planning. *New Generation Computing*, 8:225–244, 1990.
4. Robert Kowalski and M. Sergot. A logic based calculus of events. *New Generation Computing*, 4:67–95, 1986.
5. Yves Lespérance, Hector J. Levesque, Fangzhen Lin, D. Marcu, Ray Reiter, and Richard B. Scherl. A logical approach to high-level robot programming—a progress report. In B. Kuipers, editor, *Control of the Physical World by Intelligent Agents, Papers from the AAAI Fall Symposium*, pages 109–119, New Orleans, LA, November 1994.
6. Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1–3):59–83, 1997.
7. John McCarthy. *Situations and Actions and Causal Laws.* Stanford Artificial Intelligence Project, Memo 2, Stanford University, CA, 1963.
8. Raymond Reiter. *Logic in Action.* MIT Press, 2001.
9. Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia.* MIT Press, 1997.
10. Murray Shanahan and Mark Witkowski. High-level robot control through logic. In C. Castelfranchi and Y. Lespérance, editors, *Proceedings of the International Workshop on Agent Theories Architectures and Languages (ATAL)*, volume 1986 of *LNCS*, pages 104–121, Boston, MA, July 2000. Springer.
11. Michael Thielscher. From Situation Calculus to Fluent Calculus: State update axioms as a solution to the inferential frame problem. *Artificial Intelligence*, 111(1–2):277–299, 1999.
12. Michael Thielscher. Programming of reasoning and planning agents with FLUX. In D. Fensel, D. McGuinness, and M.-A. Williams, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 435–446, Toulouse, France, April 2002. Morgan Kaufmann.
13. Michael Thielscher. Pushing the envelope: Programming reasoning agents. In Chitta Baral and Sheila McIlraith, editors, *AAAI Workshop on Cognitive Robotics*, Edmonton, Canada, July 2002. AAAI Press.