

Reasoning about actions with CHRs and Finite Domain Constraints

Michael Thielscher

Department of Computer Science
Dresden University of Technology
01062 Dresden (Germany)
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. Systematic 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 cognitive capability of planning, finally, 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.

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, evaluating conditions in an agent program always necessitates to trace back the entire history of actions, hence requires ever increasing computational effort as the agent proceeds. Studies have shown that this concept fails to scale up to long-term agent control [14].

An explicit state representation being a fundamental concept in the Fluent Calculus [11], this established and versatile action representation formalism [12] 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 which implements the Fluent Calculus. Incomplete states are represented as *open* lists, that is, lists with a variable tail. This requires to encode both negative and disjunctive state knowledge as *constraints*. We present a set of declarative rules, 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. In an accompanying paper, 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. With its powerful constraint solver, the underlying FLUX kernel provides general reasoning facilities, so that the agent programmer can focus on designing complex high-level behavior. Moreover, studies have shown that our constraint solver exhibits excellent computational behavior and scales up well.

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.

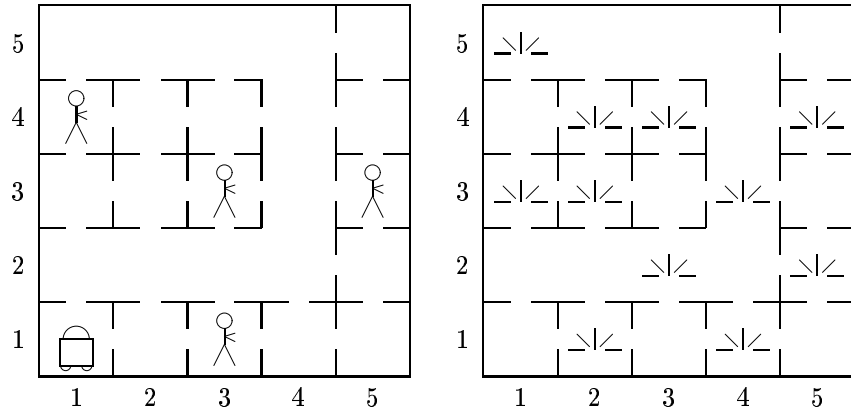


Figure 1: Layout of a sample office floor and a scenario in which four offices are occupied. In the right hand side are depicted the locations in which the cleaning robot senses light.

The constraint solver, the general FLUX system, the example agent program, and the accompanying papers all are available for download at our web site 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 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 must ensure that all their decisions are cautious, 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 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 $d \in \{1, \dots, 4\}$ (denoting, resp., north, east, south, and west); $Cleaned(x, y)$, rep-

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, denoted by $UNA[At, Faces, Cleaned, Occupied]$.¹

States are built up from fluents (as atomic states) and their conjunction, using the standard function $\circ : STATE \times STATE \mapsto STATE$ along with the standard 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 if z can be decomposed into two states one of which is the singleton f . Conversely, f does not hold in z if the latter cannot be decomposed in this way. For notational convenience, we introduce the macro $Holds(f, z)$ as an abbreviation for the corresponding equality formula:

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

This fundamental notion of fluents to hold in states requires a special theory of equality of state terms. The following foundational axioms of the Fluent Calculus serve this purpose.²

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:

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

$$\begin{aligned} (z_1 \circ z_2) \circ z_3 &= z_1 \circ (z_2 \circ z_3) & z \circ z &= z \\ z_1 \circ z_2 &= z_2 \circ z_1 & z \circ \emptyset &= z \end{aligned} \quad (2)$$

2. Empty state axiom,

$$\neg Holds(f, \emptyset) \quad (3)$$

3. Irreducibility and decomposition,

$$Holds(f_1, f) \supset f_1 = f \quad (4)$$

$$Holds(f, z_1 \circ z_2) \supset Holds(f, z_1) \vee Holds(f, z_2) \quad (5)$$

4. State equivalence and existence of states,

$$(\forall f) (Holds(f, z_1) \equiv Holds(f, z_2)) \supset z_1 = z_2 \quad (6)$$

$$(\forall \Phi)(\exists z)(\forall f) (Holds(f, z) \equiv \Phi(f)) \quad (7)$$

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

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

² 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 , respectively. The function “ \circ ” is written in infix notation.

□

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. For example, consider the definition of what it means for our cleaning robot to sense light in a square (x, y) in some state z :

$$\begin{aligned} \text{LightPerception}(x, y, z) &\equiv \\ &\text{Holds}(\text{Occupied}(x + 1, y), z) \vee \text{Holds}(\text{Occupied}(x, y + 1), z) \\ &\vee \text{Holds}(\text{Occupied}(x - 1, y), z) \vee \text{Holds}(\text{Occupied}(x, y - 1), z) \end{aligned} \quad (8)$$

Suppose initially it is only known that neither the robot’s home $(1, 1)$ nor any square in the alley or outside the boundaries of the office floor can be occupied. Suppose further that the robot of Fig. 1 starts with cleaning $(1, 1)$, $(1, 2)$, and $(1, 3)$, sensing light in the last square only. Thus the current state, ζ , is known to be

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

for some z , along with the following axioms:

$$\neg \text{Holds}(\text{Occupied}(1, 1), z) \quad (10)$$

$$\neg \text{Holds}(\text{Occupied}(1, 5), z) \wedge \dots \wedge \neg \text{Holds}(\text{Occupied}(1, 2), z) \quad (11)$$

$$(\forall x) (\neg \text{Holds}(\text{Occupied}(x, 0), z) \wedge \neg \text{Holds}(\text{Occupied}(x, 6), z)) \quad (12)$$

$$(\forall y) (\neg \text{Holds}(\text{Occupied}(0, y), z) \wedge \neg \text{Holds}(\text{Occupied}(6, y), z)) \quad (13)$$

$$\neg \text{LightPerception}(1, 1, \zeta) \wedge \neg \text{LightPerception}(1, 2, \zeta) \quad (14)$$

$$\text{LightPerception}(1, 3, \zeta) \quad (15)$$

From (14) and (8), $\neg \text{Holds}(\text{Occupied}(2, 1), \zeta) \wedge \neg \text{Holds}(\text{Occupied}(2, 2), \zeta)$. The foundational axioms of decomposition, (5), and irreducibility, (4), along with uniqueness-of-names imply

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

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

$$\begin{aligned} &\text{Holds}(\text{Occupied}(2, 3), \zeta) \vee \text{Holds}(\text{Occupied}(1, 4), \zeta) \\ &\vee \text{Holds}(\text{Occupied}(0, 3), \zeta) \vee \text{Holds}(\text{Occupied}(1, 2), \zeta) \end{aligned}$$

As above, the foundational axioms of decomposition and irreducibility along with uniqueness-of-names imply

$$\begin{aligned} &\text{Holds}(\text{Occupied}(2, 3), z) \vee \text{Holds}(\text{Occupied}(1, 4), z) \\ &\vee \text{Holds}(\text{Occupied}(0, 3), z) \vee \text{Holds}(\text{Occupied}(1, 2), z) \end{aligned}$$

From (11) and (13) it follows that

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

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 west, 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 satisfy

$$\begin{aligned} \zeta' = & \text{At}(2,2) \circ \text{Facing}(2) \\ & \circ \text{Cleaned}(1,1) \circ \text{Cleaned}(1,2) \circ \text{Cleaned}(1,3) \circ \text{Cleaned}(2,2) \circ z \end{aligned}$$

for some z such that (10)–(13), (16), (17), plus $\neg \text{LightPerception}(2,2, \zeta')$. From (8), $\neg \text{Holds}(\text{Occupied}(2,3), \zeta')$; hence, decomposition and irreducibility along with uniqueness-of-names imply $\neg \text{Holds}(\text{Occupied}(2,3), z)$; hence, according to (17), $\text{Holds}(\text{Occupied}(1,4), z)$. That is to say, it is now known that the light in (1,3) comes from (1,4) while (2,3) is unoccupied.

3 Solving State Constraints

Based on the axioms of the Fluent Calculus, in the following we develop a provably correct CLP-approach to reasoning about incomplete state specifications. To begin with, incomplete states are encoded by open lists of the form

$$[\text{F1}, \dots, \text{Fk} \mid _]$$

along with constraints expressing negative and disjunctive state knowledge:

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

The auxiliary constraint `duplicate_free(Z)` is used to stipulate that a list contains no multiple occurrences, in order to reflect the foundational axiom of idempotence of “ \circ ” in the Fluent Calculus. For example, the following clause encodes the specification of state ζ of Section 2:

```
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,Zeta,false) ,
light_perception(1,2,Zeta,false) ,
light_perception(1,3,Zeta,true) ,
duplicate_free(Zeta) .

```

where the meaning of perceiving light is given by this clause (c.f. (8)):

```

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 and ordering predicates $\#=, \#<, \#>$ along with the standard functions $+, -, *$), range constraints (written $X :: [a..b]$), and logical combinations using $\#/\wedge$ and $\#/\vee$ for conjunction and disjunction, respectively.

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

$$H_1, \dots, H_m \Leftarrow G_1, \dots, G_k \mid B_1, \dots, B_n.$$

where the *head* H_1, \dots, H_m are constraints ($m \geq 1$); the *guard* G_1, \dots, G_k are Prolog literals ($k \geq 0$); and the *body* B_1, \dots, B_n are constraints ($n \geq 0$). An empty guard is omitted; the empty body is denoted by **true**. The declarative interpretation of a CHR is given by the formula

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

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.

```

not_holds(_, [])    <=> true.
not_holds(F, [F1|Z]) <=> neq(F,F1), not_holds(F,Z).

not_holds_all(_, [])    <=> true.
not_holds_all(F, [F1|Z]) <=> neq_all(F,F1), not_holds_all(F,Z).

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

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

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) -> D=D1 ; D=((X#\=Y)#\ /D1) ).

```

Figure 2: CHRs for negation and multiple occurrences. The notation $H1 \setminus H2 \Leftrightarrow G \mid B$ is an abbreviation for $H1, H2 \Leftrightarrow G \mid H1, B$.

To begin with, consider the auxiliary clauses, which define a finite domain constraint that expresses the inequality of two fluent terms. Two cases are distinguished depending on whether the variables in the first term are existentially or universally quantified. For the latter case, 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 that the resulting finite domain constraint is correct.

Observation 2 *Consider a Fluent Calculus signature with a set \mathcal{F} of functions into sort FLUENT. If $f_1 = g(r_1, \dots, r_m)$ and $f = h(t_1, \dots, t_n)$ are two fluents and $f_2 = g(x_1, \dots, x_k, r_{k+1}, \dots, r_m)$ is a schematic fluent, 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 $UNA \models f_1 \neq f \equiv r_1 \neq t_1 \vee \dots \vee r_n \neq t_n$ and $UNA \models (\forall x) (f_2 \neq f \equiv r_{k+1} \neq t_{k+1} \vee \dots \vee r_n \neq t_n)$.

The CHRs 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,

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

Proof:

1. Follows by the empty state axiom.
2. We prove that $Holds(f, f_1 \circ z) \equiv f = f_1 \vee Holds(f, z)$:

“ \Rightarrow ”: Follows by foundational axioms (5) and (4).

“ \Leftarrow ”: 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 the second part is similar. ■

Correctness of the CHR which removes subsumed negative constraints is obvious since $(\forall \vec{x}) \neg Holds(f_1, z)$ implies $\neg Holds(f_2, z)$ for a schematic fluent f_1 and a fluents f_2 such that $f_1 \theta = f_2$ for some θ . Finally, the CHRs 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.

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 $Eq(\vec{r}, \vec{t})$ with \vec{r} and \vec{t} being of equal length. The semantics of a general disjunction or $([C1, \dots, Cn], Z)$ is

$$\bigvee_{i=1}^n \begin{cases} Holds(f, z) & \text{if } C_i \text{ is a fluent } f \\ \vec{x} = \vec{y} & \text{if } C_i \text{ is } Eq(\vec{x}, \vec{y}) \end{cases} \quad (18)$$

The first two CHRs in Fig. 3 simplify singleton disjunctions, whereby the topmost rule is justified by the following observation concerning the finite domain constraint generated by the auxiliary clauses: If $\vec{r} = r_1, \dots, r_n$ and $\vec{t} = t_1, \dots, t_n$, then $\vec{r} = \vec{t}$ iff $\bigwedge_{i=1}^n r_i = t_i$.

The third CHR simplifies a pure equational disjunction into a finite domain constraint. Its correctness follows directly from (18). The fourth CHR simplifies a disjunction applied to the empty state. It is justified by

$$\{(3)\} \models [Holds(f, \emptyset) \vee \Psi] \equiv \Psi$$

```

or([eq(X,Y)],Z) <=> and_eq(X,Y,D), call(D).
or([F],Z) <=> holds(F,Z).

or(V,Z) <=> \+(member(F,V), F\=eq(_,_)) | or_and_eq(V,D), call(D).

or(V,[]) <=> member(F,V,W), F\=eq(_,_) | or(W,[]).

not_holds(F,Z) \ or(V,Z) <=> member(G,V,W), F==G | or(W,Z).

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

or(V,[F|Z]) <=> or(V,[],[F|Z]).

or(V,W,[F|Z]) <=> member(F1,V,V1), \+ F\F1
| F1=..[_|ArgX], F2=..[_|ArgY],
| or(V1,[eq(ArgX,ArgY),F1|W],[F|Z]).

or(V,W,[_|Z]) <=> append(V,W,V1), or(V1,Z).

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).

```

Figure 3: CHRs for the disjunctive constraint.

The next two CHRs constitute unit resolution steps. They are justified by

$$\neg Holds(f, z) \wedge [Holds(f, z) \vee \Psi] \equiv \neg Holds(f, z) \wedge \Psi$$

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

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

The last three CHRs in Fig. 3, finally, are used to propagate a disjunctive constraint through a non-variable state. The rules use the auxiliary constraint $or(C, D, [F|Z])$ with intended semantics $or(C, [F|Z]) \vee or(D, Z)$. As an example, consider the constraint

$$or([f(a, V), f(W, b)], [f(X, Y)|Z])$$

which, upon being processed, yields

$\text{or}([\text{f}(\text{a}, \text{V}), \text{f}(\text{W}, \text{b}), \text{eq}([\text{a}, \text{V}], [\text{X}, \text{Y}]), \text{eq}([\text{W}, \text{b}], [\text{X}, \text{Y}])], \text{Z})$

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. $[\text{Holds}(f(\vec{x}), f_1(\vec{y}) \circ z) \vee \Psi_1] \vee \Psi_2 \equiv \Psi_1 \vee [\vec{x} = \vec{y} \vee \text{Holds}(f(\vec{x}), z) \vee \Psi_2];$
3. *if $\bigwedge_{i=1}^n f_i \neq f$, then $[\bigvee_{i=1}^n \text{Holds}(f_i, f \circ z) \vee \Psi] \equiv \Psi$.*

Proof: Item 1 is obvious. Items 2 and 3 follow from the foundational axioms of decomposition and irreducibility. \blacksquare

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)
...
```

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

```

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) :- (\+holds(F,Z) -> Z1=Z; holds(F,Z,Z1)),
                        minus(Z1,Fs,Zp).

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

update(Z1,P,N,Z2) :- minus(Z1,N,Z), plus(Z,P,Z2).

```

Figure 4: The foundational clauses for reasoning about actions.

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^+ :

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

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 $\neg \text{Holds}(f, z_1)$), or $z_1 - f$ plus f equals z_1 (which applies in case $\text{Holds}(f, z_1)$).

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 $\text{Update}(z_1, \vartheta^+, \vartheta^-, z_2)$, by which an incomplete state z_1 is updated to z_2 according to positive and negative effects, resp., ϑ^+ and ϑ^- . 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 $\text{Holds}(f, z, z')$ encodes $\text{Holds}(f, z) \wedge z' = z - f$. The following proposition shows that the definition is correct wrt. the foundational axioms of the Fluent Calculus and the macro definition of fluent removal, under the assumption that lists are free of duplicates.

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

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

Proof: Suppose $f = f_1$, then $Holds(f, z)$ according to (1), and $z' = z - f \equiv z' = (f \circ z_1) - f$. Since $Holds(f, f \circ z_1)$, macro expansion implies $z' \circ f = f \circ z_1$. From $\neg Holds(f, z') \wedge \neg Holds(f, z_1)$ and the foundational axiom on state equivalence, (6), it follows that $z' = z_1$.

Suppose, on the other hand, $f \neq f_1$, then decomposition and irreducibility imply $Holds(f, z) \equiv Holds(f, z_1)$. Moreover, if $Holds(f, z_1)$ then $z' = (f_1 \circ z_1) - f$ iff $z' \circ f = f_1 \circ z_1 \wedge \neg Holds(f, z')$. Likewise, $z'' = z_1 - f \wedge z' = f_1 \circ z''$ iff $z' = (z_1 - f) + f_1$, iff $z' \circ f = f_1 \circ z_1 \wedge \neg Holds(f, z')$. ■

The clauses for removal and addition of finitely many fluents in Fig. 4 are straightforward: If $Holds(f, z)$ is inconsistent with the specification of z , then $z - f$ equals z ; otherwise, subtraction of f is given by the definition of the ternary $Holds$ predicate. Likewise, if $Holds(f, z)$ is inconsistent with the specification of z , then a list with head f and tail z represents $z \circ f$; otherwise, $z \circ f = z$ if $Holds(f, z)$ according to the foundational axiom of idempotence.

As an example, recall the specification of state ζ at the beginning of Section 3, and consider the following update:

```
?- zeta(Z0), update(Z0, [at(1,2), cleaned(1,2)],
                    [at(1,3), occupied(1,4)], Z1).

Z1=[at(1,2), facing(1), cleaned(1,1), cleaned(1,2), cleaned(1,3) | Z]
Constraints:
not_holds(occupied(1,4), Z)
...
```

In the accompanying paper [13], 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 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. Maintaining the internal state is based on the definition of so-called state update axioms for each action, which in turn appeals to the definition of update as developed in this paper. Thanks to the extensive reasoning facilities provided by the constraint solver, FLUX allows to implement complex strategies with concise and modular agent programs.

5 Computational Behavior

Experiments have shown that FLUX and the underlying constraint solver exhibit excellent computational behavior beyond problems of toy size. In the accompanying paper [14], 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 showed that FLUX can compute

the effects of hundreds of actions per second. Most notably, the average time for inferring the effects of an action remains constant throughout the course of the program, which shows that 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.

The computational behavior of FLUX 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. Although incomplete states pose a much harder problem, FLUX proved to scale up impressively well again: During the first phase, where the agent builds its knowledge of the environment while acting, the curve shows but a linear increase of the average action computation cost. In the second phase, where the agent acts under the still incomplete knowledge, the average time for making decisions and inferring the effects of actions remains constant again, which shows that general FLUX, too, scales up effortlessly to long sequences of actions.

The results and the analysis of the behavior of GOLOG have shown that the paradigm of a state-based representation is necessary for programs to scale up well to the control of agents and robots over extended periods of time: By maintaining an explicit state term throughout the execution of the program, fluents can be directly evaluated at any stage. In contrast, the implicit representation via a situation term as in GOLOG leads to ever increasing computational effort as the program proceeds.

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. Systematic experiments have shown that the constraint solver exhibits excellent computational behavior and scales up well. This result 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 losing the computational merits of the approach.

References

- [1] W. Bibel. A deductive solution for plan generation. *New Generation Computing*, 4:115–132, 1986.
- [2] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, 1998.
- [3] S. Hölldobler and J. Schneeberger. A new deductive approach to planning. *New Generation Computing*, 8:225–244, 1990.
- [4] R. Kowalski and M. Sergot. A logic based calculus of events. *New Generation Computing*, 4:67–95, 1986.
- [5] Y. Lespérance, H. Levesque, F. Lin, D. Marcu, Ray Reiter, and R. Scherl. A logical approach to high-level robot programming—a progress report. In: *Control of the Physical World by Intelligent Agents, Papers from the AAAI Fall Symposium*, pages 109–119, 1994.
- [6] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1–3):59–83, 1997.
- [7] J. McCarthy. *Situations and Actions and Causal Laws*. Stanford Artificial Intelligence Project, Memo 2, Stanford University, CA, 1963.
- [8] R. Reiter. *Logic in Action*. MIT Press, 2001.
- [9] M. Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press, 1997.
- [10] M. Shanahan and M. Witkowski. High-level robot control through logic. In: *Proceedings of the International Workshop on Agent Theories Architectures and Languages (ATAL)*, volume 1986 of *LNCS*, pages 104–121, 2000. Springer.
- [11] M. 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] M. Thielscher. Modeling actions with ramifications in nondeterministic, concurrent, and continuous domains—and a case study. In: *Proceedings of AAAI*, pages 497–502, Austin, TX, July 2000. MIT Press.
- [13] M. Thielscher. Programming of reasoning and planning agents with FLUX. In: *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, Toulouse, France, April 2002. Morgan Kaufmann.
- [14] M. Thielscher. Pushing the envelope: Programming reasoning agents. 2002. (Submitted).