FLUX: A Logic Programming Method for Reasoning Agents

MICHAEL THIELSCHER

Dresden University of Technology, 01062 Dresden, Germany
(*e-mail:* mit@inf.tu-dresden.de)

submitted 31 August 2002; revised 7 November 2003, 23 April 2004; accepted 9 August 2004

Abstract

FLUX is a programming method for the design of agents that reason logically about their actions and sensor information in the presence of incomplete knowledge. The core of FLUX is a system of Constraint Handling Rules, which enables agents to maintain an internal model of their environment by which they control their own behavior. The general action representation formalism of the fluent calculus provides the formal semantics for the constraint solver. FLUX exhibits excellent computational behavior due to both a carefully restricted expressiveness and the inference paradigm of progression.

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 (Lespérance et al. 1994) is to endow agents with the high-level cognitive capability of reasoning. 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 for selecting the right actions. 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 on the basis of what they have previously observed or done. Moreover, the ability to reason about sensor information is necessary if properties of the environment can only be observed indirectly and require the agent to combine observations made at different stages.

While standard programming languages such as Java do not provide general reasoning facilities for agents, logic programming (LP) constitutes the ideal paradigm for designing agents that are capable of reasoning about their actions (Shanahan 1997). Examples of existing LP-systems that have been developed from general action theories are GOLOG (Levesque et al. 1997; Reiter 2001a), based on the situation cal-

culus (McCarthy 1963), or the robot control language developed in (Shanahan and Witkowski 2000), based on event calculus (Kowalski and Sergot 1986). 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 now. 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.

Having an explicit state representation as a fundamental concept, the fluent calculus (Thielscher 1999) offers an alternative theory as the formal underpinnings for a high-level agent programming method. In this paper, we present the logic programming method FLUX (for: *Fluent Executor*) for the design of intelligent agents that reason about their actions using the fluent calculus. A constraint logic program, FLUX comprises a method for encoding incomplete states along with a technique of updating these states according to a declarative specification of the elementary actions and sensing capabilities of an agent. Atomic state knowledge is encoded in a list with a tail variable, which signifies the incompleteness of the state. Negative and disjunctive state knowledge is encoded by *constraints*. We present a set of Constraint Handling Rules (CHRs) (Frühwirth 1998) for combining and simplifying these constraints. In turn, these rules reduce to standard finite domain constraints when handling variable arguments of individual state components. Appealing to their declarative interpretation, our CHRs are verified against the foundational axioms 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 specifying the application domain and designing the high-level behavior. Allowing for concise programs and supporting modularity, our method promises to be eminently suitable for programming complex strategies for artificial agents. Thanks to a restricted expressiveness and a sound but incomplete inference engine, reasoning in FLUX is linear in the size of the internal state representation. FLUX therefore exhibits excellent computational behavior. Thanks to the progression principle, FLUX scales up particularly well to long-term control.

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 an LP-based approach to reasoning about actions. In Section 3, we present a set of CHRs for constraints expressing negative and disjunctive state knowledge. We prove their correctness wrt. the foundational axioms of the fluent calculus. In Section 4, the constraint solver is embedded into a logic program for reasoning about actions, which, too, is verified against the underlying semantics of the fluent calculus. In Section 5, we integrate state knowledge and sensing into FLUX. An example of a FLUX agent program is given in Section 6, in which we also present the results of experiments showing the computational merits of our approach. We conclude in Section 7.

The constraint solver, the general FLUX system, and the example agent program are available for download at our web site `www.fluxagent.org`.

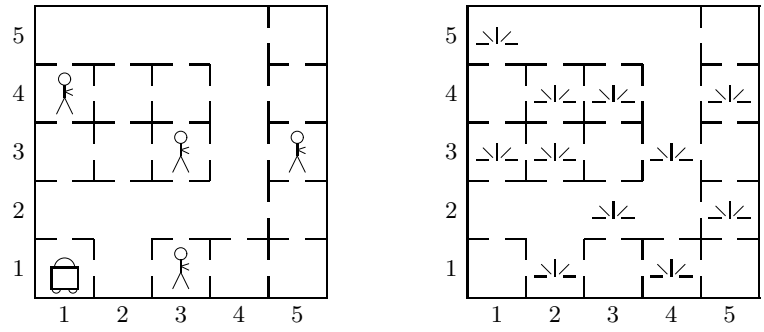

Fig. 1. Layout of a sample office floor and a scenario in which four offices are occupied. In the right hand side, the locations are depicted in which the robot senses light.

2 Reasoning about states and sensor input 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 hallway and rooms of the floor of an office building. The robot shall not, however, disturb anyone working in late. It is equipped with a light sensor which is activated whenever the robot is adjacent to a room that is occupied, without indicating which direction the light comes from. An instance of this problem is depicted in Figure 1. The robot can perform three basic actions, namely, cleaning the current location, turning clockwise by 90 degrees, and moving forward in the current direction to the adjacent cell. Our task is to program the "cleanbot" 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 (Thielscher 1999). Formally, it is a many-sorted predicate logic language which includes the two standard sorts of a FLUENT (i.e., an atomic state property) and a STATE. 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, respectively, 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, $UNA[At, Facing, Cleaned, Occupied]$. [1]

States are built up from fluents (as atomic states) and their conjunction, using

[1] Following (Baker 1989), $UNA[h_1, \ldots, h_n] \stackrel{\text{def}}{=} \bigwedge_{i<j} h_i(\vec{x}) \neq h_j(\vec{y}) \wedge \bigwedge_i [h_i(\vec{x}) = h_i(\vec{y}) \supset \vec{x} = \vec{y}]$.

the binary function $\circ : \textsc{state} \times \textsc{state} \mapsto \textsc{state}$ along with the constant $\emptyset : \textsc{state}$ 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.[2]

A fundamental notion is that of a fluent f to *hold* in a state z. For notational convenience, the macro $Holds(f, z)$ serves as an abbreviation for an equational formula which says that z can be decomposed into f and some state z':

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

This definition is accompanied by the following foundational axioms of the fluent calculus, which ensure that a state can be identified with the fluents that hold in it.

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 and commutativity,

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

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)–(5) 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

[2] A word on the notation: Predicate and function symbols start with a capital letter while variables are denoted by lowercase letters, possibly with sub- or superscripts. Function "$\circ$" is written in infix notation. Throughout the paper, free variables in formulas are assumed universally quantified. Variables of sorts FLUENT and STATE shall be denoted, respectively, by the letters f and z.

fluents, and second-order axiom (7) guarantees the existence of a state for any combination of fluents. [3]

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

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

Suppose that at the beginning the only given unoccupied locations are: the home square of the robot (axiom (10) below), the squares in the hallway (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 (cf. Figure 1). Thus the current state, ζ, satisfies

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

for some z, along with the following axioms:

$$\neg \mathit{Holds}(\mathit{Occupied}(1,1), z) \tag{10}$$

$$\neg \mathit{Holds}(\mathit{Occupied}(1,2), z) \wedge \ldots \wedge \neg \mathit{Holds}(\mathit{Occupied}(4,5), z) \tag{11}$$

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

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

$$\neg \mathit{Light}(1, 2, \zeta) \tag{14}$$

$$\mathit{Light}(1, 3, \zeta) \tag{15}$$

From (14) and (8) it follows $\neg \mathit{Holds}(\mathit{Occupied}(1,3), \zeta)$. With regard to (9), the foundational axioms of decomposition (5) and irreducibility (4) along with the axiom of uniqueness-of-names imply

$$\neg \mathit{Holds}(\mathit{Occupied}(1,3), z)$$

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

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

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

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

[3] A remark for readers who are familiar with early papers on the fluent calculus: The original solution to the frame problem in this calculus required function "$\circ$" to be non-idempotent (Hölldobler and Schneeberger 1990), so that, e.g., $\mathit{Occupied}(2,3) \neq \mathit{Occupied}(2,3) \circ \mathit{Occupied}(2,3)$. Since this is against the intuition of "$\circ$" as a reified logical conjunction, the new axiomatization, first used in (Thielscher 2001), is no longer based on non-idempotence. In fact, foundational axiom (6) along with (5) and associativity implies that $z \circ z = z$ for any z.

From (13) and (11) it follows that

$$Holds(Occupied(2,3), z) \vee Holds(Occupied(1,4), z) \tag{16}$$

This disjunction cannot be reduced further, that is, at this stage the robot cannot decide whether the light in $(1,3)$ comes from office $(2,3)$ or $(1,4)$ (or both, for that matter). Suppose, therefore, the cautious cleanbot goes back, turns east, and continues with cleaning $(2,2)$, which is a hallway location and therefore cannot be occupied according to (11). Sensing no light there (cf. Figure 1), the new state is

$$\zeta' = At(2,2) \circ Facing(2) \circ \\ Cleaned(1,1) \circ Cleaned(1,2) \circ Cleaned(1,3) \circ Cleaned(2,2) \circ z$$

for some z that satisfies (10)–(13) and (16). We also know that $\neg Light(2,2,\zeta')$. From (8), $\neg Holds(Occupied(2,3), \zeta')$; hence, decomposition and irreducibility along with the axiom of uniqueness-of-names imply $\neg Holds(Occupied(2,3), z)$; hence, from (16) it follows $Holds(Occupied(1,4), z)$, that is, now the robot can conclude that $(1,4)$ is occupied.

3 A constraint solver for the fluent calculus

The axiomatic fluent calculus provides the formal underpinnings for an LP-based approach to reasoning about incomplete state specifications. To begin with, incomplete states are encoded by open-ended lists of fluents (possibly containing variables):

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

It is assumed that the arguments of fluents are encoded by integers or symbolic constants, 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)`	$(\forall \vec{x})\, \neg Holds(f, z)$, $\vec{x}$ variables in f
`or_holds([F1,...,Fn],Z)`	$\bigvee_{i=1}^{n} Holds(f_i, z)$

These state constraints have been carefully designed so as to be sufficiently expressive while allowing for efficient constraint solving. An auxiliary constraint, written `duplicate_free(Z)`, is used to stipulate that a list of fluents contains no multiple occurrences. As an example, the following clause encodes the specification of state ζ of Section 2 (cf. axioms (9)–(15)):

```
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(1,2), Z), ..., not_holds(occupied(4,5), 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),
```

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

The auxiliary predicate $Light(x, y, p, z)$ defines under what circumstances there is light ($p = True$) or no light ($p = False$) in state z at square (x, y) (cf. axiom (8)).

```
light(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_holds([occupied(XE,Y),occupied(X,YN),
                                occupied(XW,Y),occupied(X,YS)], Z) ).
```

Here and in the following, we use the standard constraint language of finite domains (see, e.g., (Hentenryck 1989)), which includes arithmetic constraints over the integers and symbolic constants, using the equality, inequality, and ordering predicates `#=`, `#\=`, `#<`, `#>` along with the arithmetic functions `+`, `-`, `*`; range constraints (written `X::`a`..`b); and logical combinations using `#/\` and `#\/` for conjunction and disjunction, respectively.

The state constraints are processed using Constraint Handling Rules (Frühwirth 1998). The general form of these rules is

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

where the *head* $H_1, \ldots, H_m$ is a sequence of constraints ($m \geq 1$); the *guard* $G_1, \ldots, G_k$ is a sequence of Prolog literals ($k \geq 0$); and the *body* $B_1, \ldots, B_n$ is a sequence of constraints or Prolog literals ($n \geq 0$). An empty guard is omitted; the empty body is denoted by $True$. The declarative interpretation of such a rule 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 which match the head are replaced by the body.

3.1 Handling negation

Figure 2 depicts the first part of the constraint solver, which contains the CHRs and auxiliary clauses for the two negation constraints and the 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 *OrNeq*, 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

```
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. Constraint Handling Rules for the negation constraints and multiple occurrences of fluents. The notation `H1 \ H2 <=> G | B` is an abbreviation for `H1,H2 <=> G | H1,B`.

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. Thus $\texttt{neq\_all}(\texttt{f}(\_,\texttt{a},\_),\texttt{f}(\texttt{U},\texttt{V},\texttt{W}))$ 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 and $\vec{r}$ the non-variable arguments. The following observation implies the correctness of the constraints generated by the auxiliary clauses.

Observation 1

Consider a set $\mathcal{F}$ of functions into sort FLUENT, 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 $f = h(t_1,\ldots,t_n)$. Let $\mathit{Neq}(f_1,f) \stackrel{\text{def}}{=} f_1 \neq f$ and $\mathit{NeqAll}(f_2,f) \stackrel{\text{def}}{=} (\forall x_1,\ldots,x_k)\, f_2 \neq f$, then

1. if $g \neq h$, then $\mathit{UNA}[\mathcal{F}] \models \mathit{Neq}(f_1,f)$ and $\mathit{UNA}[\mathcal{F}] \models \mathit{NeqAll}(f_2,f)$;

2. if $g = h$, then $m = n$, and $UNA[\mathcal{F}]$ entails

$$Neq(f_1, f) \equiv r_1 \neq t_1 \vee \ldots \vee r_m \neq t_n \vee 0 \neq 0$$
$$NeqAll(f_2, f) \equiv [\bigvee_{\substack{i \neq j \\ x_i = x_j}} t_i \neq t_j] \vee r_{k+1} \neq t_{k+1} \vee \ldots \vee r_m \neq t_n \vee 0 \neq 0$$

CHRs 1–4 in Figure 2, by which negation constraints are propagated, are then justified—on the basis of their declarative interpretation—by the foundational axioms of the fluent calculus.

Proposition 1
Foundational axioms Σ_{state} entail

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. For claim 2 we prove that $Holds(f, f_1 \circ z)$ iff $f = f_1 \vee Holds(f, z)$. The "$\Rightarrow$" direction follows by the foundational axioms of decomposition and irreducibility. For the "$\Leftarrow$" direction, suppose $f = f_1$, then $f_1 \circ z = f \circ z$, hence $Holds(f, f_1 \circ z)$. Likewise, suppose $Holds(f, z)$, 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. □

CHRs 5 and 6, by which subsumed negative constraints are removed, are correct since $(\forall \vec{x})\, \neg Holds(f_1, z)$ implies both $\neg Holds(f_2, z)$ and $(\forall \vec{y})\, \neg Holds(f_2, z)$, where f_1 is a schematic fluent and f_2 is a fluent 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.

3.2 Handling disjunction

Figure 3 depicts the second part of the constraint solver, which contains the CHRs and auxiliary clauses for disjunctive state knowledge. The solver employs an extended notion of a disjunctive clause, where each disjunction may include atoms of the form $Eq(\vec{x}, \vec{y})$ in addition to fluents. The meaning of such a general disjunctive constraint $OrHolds([\delta_1, \ldots, \delta_k], z)$ is

$$\bigvee_{i=1}^{k} \begin{cases} Holds(f, z) & \text{if } \delta_i \text{ is fluent } f \\ \vec{x} = \vec{y} & \text{if } \delta_i \text{ is } Eq(\vec{x}, \vec{y}) \end{cases} \tag{17}$$

This generalization is needed for propagating disjunctions with variables through compound states. Consider, as an example, $OrHolds([F(x), F(1)], [F(y)|z])$. This

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

or_holds(V,[])  <=> member(F,V,W), F\=eq(_,_) | or_holds(W,[]).       %11

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

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

or_holds(V,[F|Z])        <=> or_holds(V,[],[F|Z]).                    %16
or_holds([F1|V],W,[F|Z]) <=> F1==F -> true ;                          %17
                             F1\=F -> or_holds(V,[F1|W],[F|Z]) ;
                             F1=..[_|ArgX], F=..[_|ArgY],
                              or_holds(V,[eq(ArgX,ArgY),F1|W],[F|Z]).
or_holds([],W,[_|Z])     <=> or_holds(W,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. Constraint Handling Rules for the disjunctive constraint.

constraint will be rewritten to $OrHolds([Eq([1],[y]),F(1),Eq([x],[y]),F(x)],z)$, in accordance with the fact that $\Sigma_{state} \cup UNA[F]$ entails

$$\begin{gathered} Holds(F(x),F(y)\circ z) \vee Holds(F(1),F(y)\circ z) \\ \equiv \\ x = y \vee Holds(F(x),z) \vee 1 = y \vee Holds(F(1),z) \end{gathered}$$

which follows by the foundational axioms of irreducibility and decomposition.

CHR 9 in Figure 3 simplifies a singleton disjunction according to (17). CHR 10 reduces a pure equational disjunction to a finite domain constraint. Its correctness follows directly from (17), too. CHR 11 simplifies a disjunction applied to the empty state. It is justified by the empty state axiom, which entails

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

for any formula Ψ. CHRs 12 and 13 deal with disjunctions which include an equality which is either true under any variable assignment, or false. If the former, then the entire disjunction is true. If, on the other hand, the equality is necessarily false, then it is removed from the disjunction. Correctness follows from

$$\vec{x} = \vec{y} \supset [(\vec{x} = \vec{y} \vee \Psi) \equiv \top] \quad \text{and} \quad \vec{x} \neq \vec{y} \supset [(\vec{x} = \vec{y} \vee \Psi) \equiv \Psi]$$

The next two CHRs are unit resolution steps: Rule 14 says that if a fluent f does not hold, then any disjunction that contains an equal fluent g can be reduced by g. Rule 15 generalizes this to universally quantified negation constraints. The two CHRs are justified, respectively, by

$$\begin{aligned} \neg \mathit{Holds}(f, z) &\supset [(\mathit{Holds}(f, z) \vee \Psi) \equiv \Psi] \\ (\forall \vec{x})\, \neg \mathit{Holds}(f, z) &\supset [(\mathit{Holds}(g, z) \vee \Psi) \equiv \Psi] \quad \text{if } f\theta = g \text{ for some } \theta \end{aligned}$$

where $\vec{x}$ are the variables in f.

The last group of CHRs, 16–18, encode the propagation of a disjunction through a compound state. Informally speaking, each element in the disjunct is compared to the head of the state and, if the two are unifiable, the respective equational constraint is introduced into the disjunction. Specifically, with the help of the auxiliary ternary constraint $\mathit{OrHolds}(v, w, [f|z])$, a disjunction is divided into two parts. List v contains the fluents that have not yet been evaluated against the head f of the state. List w contains those fluents that have been evaluated. Thus the meaning of a ternary expression $\mathit{OrHolds}(\Delta_1, \Delta_2, [f|z])$ is

$$\mathit{OrHolds}(\Delta_1, [f|z]) \vee \mathit{OrHolds}(\Delta_2, z) \tag{18}$$

In the special case that disjunction Δ_1 contains a fluent f_1 which is identical to the head f of the state, disjunction (18) is necessarily true and, hence, is resolved to *True* by CHR 17. Otherwise, any fluent f_1 in Δ_1 which does not unify with f is propagated without inducing an equality. Any fluent f_1 which does unify with f extends the disjunction by the equality of the arguments of f_1 and f. Recall, for example, the constraint $\mathit{OrHolds}([F(x), F(1)], [F(y)|z])$ mentioned earlier, which is propagated thus:

$$\begin{aligned} & \mathit{OrHolds}([F(x), F(1)], [F(y)|z]) \\ \xrightarrow{\text{\%16}}\ & \mathit{OrHolds}([F(x), F(1)], [\,], [F(y)|z]) \\ \xrightarrow{\text{\%17}}\ & \mathit{OrHolds}([F(1)], [Eq([x], [y]), F(x)], [F(y)|z]) \\ \xrightarrow{\text{\%17}}\ & \mathit{OrHolds}([\,], [Eq([1], [y]), F(1), Eq([x], [y]), F(x)], [F(y)|z]) \\ \xrightarrow{\text{\%18}}\ & \mathit{OrHolds}([Eq([1], [y]), F(1), Eq([x], [y]), F(x)], z) \end{aligned}$$

The three rules for propagating a disjunction are justified by the following proposition, where item 1 is for CHR 16, items 2–4 are for the three cases considered in CHR 17, and item 5 is for CHR 18.

Proposition 2
Consider a fluent calculus signature with a set $\mathcal{F}$ of functions into sort FLUENT. Foundational axioms Σ_{state} and uniqueness-of-names $\mathit{UNA}[\mathcal{F}]$ entail each of the following, where Ψ_1 is of the form $\mathit{OrHolds}(\Delta, [f|z])$ and Ψ_2 is of the form $\mathit{OrHolds}(\Delta, z)$:

1. $\Psi_1 \equiv [\Psi_1 \vee \mathit{OrHolds}([\,], z)]$;
2. $[\mathit{Holds}(f, f \circ z) \vee \Psi_1] \vee \Psi_2 \equiv \top$;
3. $f_1 \neq f \supset ([\mathit{Holds}(f_1, f \circ z) \vee \Psi_1] \vee \Psi_2 \equiv \Psi_1 \vee [\mathit{Holds}(f_1, z) \vee \Psi_2])$;
4. $[\mathit{Holds}(F(\vec{x}), F(\vec{y}) \circ z) \vee \Psi_1] \vee \Psi_2 \equiv \Psi_1 \vee [\vec{x} = \vec{y} \vee \mathit{Holds}(F(\vec{x}), z) \vee \Psi_2]$;
5. $[\mathit{OrHolds}([\,], [f|z]) \vee \Psi_2] \equiv \Psi_2$.

Proof
Claims 1 and 5 are obvious. Claim 2 follows by the definition of *Holds*. Claims 3 and 4 follow from the foundational axioms of decomposition and irreducibility along with $\mathit{UNA}[\mathcal{F}]$. □

3.3 Using the constraint solver

The constraint solver constitutes a system for automated reasoning about incomplete states and sensor information. As an example, evaluating the specification from the beginning of Section 3 results in

```
?- zeta(Zeta).

Zeta=[at(1,3),facing(1),cleaned(1,1),cleaned(1,2),cleaned(1,3) | Z]

Constraints:
or_holds([occupied(1,4),occupied(2,3)], Z)
...
```

Light at $(1,3)$ thus implies that $(1,4)$ or $(2,3)$ is occupied, but it does not follow which of the two. Adding the information that there is no light in $(2,2)$, the system is able to infer that $(1,4)$ must be occupied:

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

Zeta=[at(1,3),facing(1),cleaned(1,1),cleaned(1,2),cleaned(1,3),
      occupied(1,4) | Z]

Constraints:
not_holds(occupied(2,3), Z)
...
```

Although the CHRs in the FLUX constraint system are correct, they may not enable agents to draw all conclusions that follow logically from a state specification if the underlying arithmetic solver trades full inference capabilities for efficiency. In standard implementations this is indeed the case, because a conjunction or a disjunction is simplified only if one of its equations or disequations is either necessarily true or necessarily false. As a crucial advantage of these concessions we have designed an efficient inference system: The computational effort of evaluating a new constraint is *linear* in the size of the constraint store.

```
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. Auxiliary predicate *Cancel* is defined in Figure 5.

4 Inferring state update in FLUX

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 (Hölldobler and Schneeberger 1990; Bibel 1986), the fluent calculus provides a solution to the fundamental frame problem in the presence of incomplete states (Thielscher 1999). The key is a rigorously axiomatic characterization of addition and removal of (finitely many) fluents from incompletely specified states. The following inductive definition introduces the macro equation $z_1 - \vartheta^- = z_2$ with the intended meaning that state z_2 is state z_1 minus the fluents in the finite state ϑ^-:

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

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 $\mathit{Holds}(f, z_1)$).

A further 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^+:

$$z_2 = (z_1 - \vartheta^-) + \vartheta^+ \stackrel{\text{def}}{=} (\exists z)\,(z_1 - \vartheta^- = z \wedge z_2 = z \circ \vartheta^+) \tag{19}$$

where both ϑ^+, ϑ^- are finitely many FLUENT terms connected by "$\circ$".

Figure 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 culmi-

nates in the predicate $Update(z_1, \vartheta^+, \vartheta^-, z_2)$, by which an incomplete state z_1 is updated to z_2 by positive and negative effects ϑ^+ and ϑ^-, respectively, according to macro (19). The first two clauses in Figure 4 encode macro (1). Correctness of this definition follows from the foundational axioms of decomposition and irreducibility. The ternary $Holds(f, z, z')$ means $Holds(f, z) \wedge z' = z - f$. The following proposition implies that the corresponding clauses are correct wrt. the macro definition of fluent removal, under the assumption that lists of fluents are free of duplicates.

Proposition 3
Axioms $\Sigma_{state} \cup \{z = f_1 \circ z_1 \wedge \neg Holds(f_1, z_1)\}$ entail

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

Proof
We distinguish two cases.

Suppose $f = f_1$, then $Holds(f, z)$ since $z = f_1 \circ z_1$. If $z' = z - f$, 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 $z' = (f_1 \circ z_1) - f_1 = z - f$.

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 $Holds(f, f_1 \circ z_1)$ 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 in Figure 4. 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 $z - f$ equals z. If, however, the status of the fluent is not entailed by the state specification at hand for z, then partial information 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

$$Holds(F(y), z) \wedge [\, Holds(F(A), z) \vee Holds(F(B), z) \,] \tag{20}$$

This formula does not entail $Holds(F(A), z)$ nor $\neg Holds(F(A), z)$. So what can be inferred about the state $z - F(A)$? Macro expansion of "$-$" implies that Σ_{state} and $\{(20)\} \cup \{z_1 = z - F(A)\}$ entail $\neg Holds(F(A), z_1)$. But it does not follow whether $F(y)$ holds in z_1, nor whether $F(B)$ does, because

$$\begin{aligned} &[\, y = A \supset \neg Holds(F(y), z_1) \,] \wedge \\ &[\, y \neq A \supset Holds(F(y), z_1) \,] \wedge \\ &[\, \neg Holds(F(B), z) \supset \neg Holds(F(B), z_1) \,] \wedge \\ &[\, Holds(F(B), z) \supset Holds(F(B), z_1) \,] \end{aligned}$$

For this reason, all partial information concerning f in the current state z is cancelled in the clause for *Minus* prior to asserting that f does not hold in the resulting state. The definition of cancellation of a fluent f is given in Figure 5 as an

```
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_holds(V,Z)      <=> member(G,V), \+ F\=G | true.

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

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

extension of our system of CHRs. In the base case, all negative and disjunctive state information affected by f is resolved via the constraint $\mathit{Cancel}(f, z)$. The latter in turn is resolved by the auxiliary constraint $\mathit{Cancelled}(f, z)$, indicating that z contains no (more) state knowledge which is affected by f. In the recursive clause for $\mathit{Cancel}(f, z_1, z_2)$, each atomic, positive state information that unifies with f is cancelled.

In a similar fashion, the recursive clause for *Plus* in Figure 4 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. 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, respectively, in the resulting state. Moreover, 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 state resulting from the update. The correctness of this encoding of update follows from the macros for "$-$" and "$+$", which imply that 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.

5 Reasoning about actions

In this section, we extend our basic programming system so as to enable agents to reason about what they know and to infer the results of actions involving sensor information. Reasoning about knowledge is necessary for agents with incomplete information, as they need to select actions according to what they know of the state of the environment. The formal concept of state knowledge also allows to specify the effects of sensing actions, which, rather than affecting the state itself, provide the agent with more information about it.

5.1 Knowledge and sensing in the fluent calculus

Adopted from the situation calculus (McCarthy 1963), the two standard sorts ACTION and SIT (i.e., situations) are used in the fluent calculus to represent, respectively, actions and sequences of actions. Action sequences are rooted in an initial situation, usually denoted by the constant $S_0 : \text{SIT}$. The pre-defined function $Do : \text{ACTION} \times \text{SIT} \mapsto \text{SIT}$ maps an action and a situation into the situation after the action. The function symbol $State : \text{SIT} \mapsto \text{STATE}$ is unique to the fluent calculus and links the two key notions of a state and a situation: $State(s)$ denotes the state in situation s.

Inspired by a model of knowledge in the situation calculus (Moore 1985; Scherl and Levesque 2003), the predicate $KState : \text{SIT} \times \text{STATE}$ has been introduced in (Thielscher 2000). An instance $KState(s, z)$ means that, according to the knowledge of the agent, z is a possible state in situation s. As an example, recall the initial state of our cleaning robot as depicted in Figure 1. For the sake of argument, suppose that the robot is told it would perceive light in $(1, 3)$. The initial knowledge of the cleanbot can then be specified by the following axiom, which defines the *knowledge state* in situation S_0:

$$\begin{array}{l}(\forall z_0)\,(\,KState(S_0, z_0) \equiv \\ \quad (\exists z)\,(\,z_0 = At(1,1) \circ Facing(1) \circ z \wedge \\ \qquad (\forall x, y)\,\neg Holds(At(x, y), z) \wedge (\forall d)\,\neg Holds(Facing(d), z) \wedge \\ \qquad \neg Holds(Occupied(1, 1), z) \wedge \\ \qquad \neg Holds(Occupied(1, 2), z) \wedge \ldots \wedge \neg Holds(Occupied(4, 5), z) \wedge \\ \qquad (\forall x)\,(\neg Holds(Occupied(x, 0), z) \wedge \neg Holds(Occupied(x, 6), z)) \wedge \\ \qquad (\forall y)\,(\neg Holds(Occupied(0, y), z) \wedge \neg Holds(Occupied(6, y), z)) \wedge \\ \qquad Light(1, 3, z_0)\,)\,)\end{array} \tag{21}$$

That is to say, initially possible are all states in which the robot is at a unique position, viz. $(1, 1)$, facing a unique direction, viz. north (1), and neither $(1, 1)$ nor any square in the hallway or outside the boundaries can be occupied. The possible states are further constrained by the knowledge that there is light at $(1, 3)$. On the other hand, the agent has no further prior knowledge as to which offices are occupied or if any location is cleaned.

A universal property of knowledge is that it is correct. To this end, a simple foundational axiom stipulates that the actual state is always among the possible ones:

Definition 2

The *foundational axioms of the fluent calculus for knowledge* are Σ_{state} as in Definition 1 (cf. Section 2) augmented by

$$(\forall s)\,KState(s, State(s))$$

Based on the notion of possible states, a fluent is known to hold in a situation (or

not to hold) just in case it is true (false, respectively) in all possible states in that situation: [4]

$$\begin{aligned} \mathit{Knows}(f, s) &\stackrel{\text{def}}{=} (\forall z)\,(\mathit{KState}(s, z) \supset \mathit{Holds}(f, z)) \\ \mathit{Knows}(\neg f, s) &\stackrel{\text{def}}{=} (\forall z)\,(\mathit{KState}(s, z) \supset \neg\mathit{Holds}(f, z)) \end{aligned} \tag{22}$$

For example, the axiomatization of the initial knowledge, (21), entails that the cleanbot knows it is at $(1, 1)$ not facing east, that is,

$$\Sigma_{state} \cup \{(21)\} \models \mathit{Knows}(\mathit{At}(1, 1), S_0) \wedge \mathit{Knows}(\neg\mathit{Facing}(2), S_0)$$

On the other hand, the cleanbot does not know that office $(1, 4)$ is occupied:

$$\Sigma_{state} \cup \{(21)\} \models \neg\mathit{Knows}(\mathit{Occupied}(1, 4), S_0)$$

This is so because there is a possible state z_0 which satisfies the right hand side of the equivalence in (21) and in which $\mathit{Occupied}(1, 4)$ does not hold.

A supplementary macro defines knowledge of a value of a fluent. An agent has this knowledge just in case a particular instance of the fluent in question is known:

$$\mathit{KnowsVal}(\vec{x}, f, s) \stackrel{\text{def}}{=} (\exists \vec{x})\,\mathit{Knows}((\exists \vec{x}_1)\,f, s) \tag{23}$$

where $\vec{x}_1$ are the variables in f besides $\vec{x}$, and $\mathit{Knows}((\exists \vec{x}_1)\,f, s)$ stands for the formula $(\forall z)\,(\mathit{KState}(s, z) \supset (\exists \vec{x}_1)\,\mathit{Holds}(f, z))$. For example, the axiomatization of the initial knowledge entails that the cleanbot knows which direction it faces,

$$\Sigma_{state} \cup \{(21)\} \models \mathit{KnowsVal}(d, \mathit{Facing}(d), S_0)$$

On the other hand, although it knows that some office must be occupied, i.e.,

$$\Sigma_{state} \cup \{(21)\} \models \mathit{Knows}((\exists x, y)\,\mathit{Occupied}(x, y), S_0)$$

the cleanbot does not know which one,

$$\Sigma_{state} \cup \{(21)\} \models \neg\mathit{KnowsVal}((x, y), \mathit{Occupied}(x, y), S_0)$$

This is so because there exists a possible state z_0 which satisfies the right hand side of the equivalence in (21) and in which $\mathit{Occupied}(1, 4)$ is the only positive instance of this fluent; and there also exists a possible state in which a different one, viz. $\mathit{Occupied}(2, 3)$, is the only positive instance of this fluent.

While the definitions of knowledge by macros (22) and (23) are similar to the approach in the situation calculus (Scherl and Levesque 2003), a crucial difference is that the latter defines knowledge in terms of possible *situations*. To this end, the binary relation $K(s, s')$ is used with the intuitive meaning that as far as the agent knows in situation s, it could as well be in situation s'. This allows for a nested definition of *Knows*, which provides a form of introspection that is not supported in the fluent calculus. On the other hand, the full expressiveness of modal logic is computationally demanding. The notion of possible states allows for a straightforward and—based on the results of the previous sections—tractable implementation of knowledge, which is crucial for practical purposes. We refer to (Thielscher 2000) for a more detailed comparison between the two approaches.

[4] For the sake of simplicity, we only consider knowledge of fluent literals in this paper; see (Thielscher 2000) for the generic extension to knowledge of formulas.

```
knows(F, Z) :- \+ not_holds(F, Z).

knows_not(F, Z) :- \+ holds(F, Z).

knows_val(X, F, Z) :- k_holds(F, Z), \+ nonground(X).

k_holds(F, Z) :- nonvar(Z), Z = [F1|Z1],
                 ( instance(F1, F), F = F1 ; k_holds(F, Z1) ).
```

Fig. 6. Knowledge in FLUX.

5.2 Inferring knowledge in FLUX

The concept of knowing properties of the state is essential for the evaluation of conditions in agent programs under incomplete information. By definition, a property is known just in case it is true in all possible states. From a computational perspective, it is of course impractical to evaluate a condition by literally checking every possible state, since there is usually quite a number, often even infinitely many of them. Fortunately, our constraint solver provides a feasible alternative. Instead of verifying that all states satisfy a property, we can just as well prove that the *negation* of the property is *unsatisfiable* under a given knowledge state. This suggests an elegant way of encoding knowledge in FLUX using the principle of negation-as-failure. To begin with, a knowledge state $KState(\sigma, z) \equiv \Phi(z)$ is identified with the (incomplete) state specification $\Phi(z)$. Then a fluent f is known in situation σ iff the axiom set $\{\Phi(z), \neg Holds(f, z)\}$ is unsatisfiable. Likewise, f is known to be false in situation σ iff $\{\Phi(z), Holds(f, z)\}$ is unsatisfiable.

Theorem 1
Let $KState(\sigma, z) \equiv \Phi(z)$ be a knowledge state and f a fluent, then

$$\{KState(\sigma, z) \equiv \Phi(z)\} \models Knows(f, \sigma) \text{ iff } \{\Phi(z), \neg Holds(f, z)\} \models \bot$$

and

$$\{KState(\sigma, z) \equiv \Phi(z)\} \models Knows(\neg f, \sigma) \text{ iff } \{\Phi(z), Holds(f, z)\} \models \bot$$

Proof

$$\begin{aligned}
& \{KState(\sigma, z) \equiv \Phi(z)\} \models Knows(f, \sigma) \\
\text{iff} \quad & \{KState(\sigma, z) \equiv \Phi(z)\} \models (\forall z)\,(KState(\sigma, z) \supset Holds(f, z)) \\
\text{iff} \quad & \models (\forall z)\,(\Phi(z) \supset Holds(f, z)) \\
\text{iff} \quad & \models \neg(\exists z)\,(\Phi(z) \wedge \neg Holds(f, z)) \\
\text{iff} \quad & \{\Phi(z), \neg Holds(f, z)\} \models \bot
\end{aligned}$$

The proof of the second part is similar. □

This result is a formal justification of concluding knowledge of f if the constraint solver derives an inconsistency upon asserting state constraint $\neg Holds(f, z)$ under

state specification $\Phi(z)$. Figure 6 shows how this is realized in FLUX by clauses for $\mathit{Knows}(f, s)$ and $\mathit{Knows}(\neg f, s)$ as well as for knowing a value of a fluent. More complex knowledge expressions, such as disjunctive knowledge, can be defined and encoded in a similar fashion. The clausal definition of $\mathit{KnowsVal}(\vec{x}, f, z)$ uses the auxiliary predicate $\mathit{KHolds}(f, z)$, which *matches* the fluent expression f against all fluents that positively occur in state z. If so doing grounds all variables in $\vec{x}$, then a value for these variables is known.

Recall, for example, the FLUX state specification at the beginning of Section 3, encoding state specification (9)–(15). We can use FLUX to show that the robot knows that room $(1, 3)$ is not occupied, while it does not know that office $(1, 4)$ is free, nor that it is not so:

```
?- zeta(Zeta),
   knows_not(occupied(1,3), Zeta),
   \+ knows(occupied(1,4), Zeta),
   \+ knows_not(occupied(1,4), Zeta).

yes.
```

As an example for the FLUX definition of knowing a value, consider this incomplete state specification:

```
init(Z0) :-
   Z0=[at(X,2),facing(2)|Z], X#=1 #\/ X#=2, duplicate_free(Z0).
```

The corresponding axiom in fluent calculus is

$$\mathit{KState}(S_0, z_0) \equiv (\exists x, z)\,(z_0 = \mathit{At}(x, 2) \circ \mathit{Facing}(2) \circ z \wedge [x = 1 \vee x = 2])$$

It follows that $\mathit{KnowsVal}(d, \mathit{Facing}(d), S_0)$ while $\neg\mathit{KnowsVal}((x, y), \mathit{At}(x, y), S_0)$ but $\mathit{KnowsVal}(y, \mathit{At}(x, y), S_0)$:

```
?- init(Z0),
   knows_val([D], facing(D), Z0),
   \+ knows_val([X,Y], at(X,Y), Z0),
   knows_val([Y], at(_,Y), Z0).

D = 2
Y = 2
```

In theory, agents using the fluent calculus are logically omniscient. Therefore, the general problem of inferring knowledge under incomplete states is computationally demanding, if not undecidable in the first-order case. This is so because full theorem proving is required to this end. The careful design of the state constraints supported in FLUX and the incomplete constraint solver, however, make the task computationally feasible. Since deciding unsatisfiability of a set of constraints is linear in the size of the constraint store, inferring knowledge in FLUX is linear in the size of the state description.

5.3 Knowledge update

The frame problem for knowledge is solved in the fluent calculus by axiomatizing the relation between the possible states before and after an action (Thielscher 2000). The effect of $A(\vec{x})$, be it a sensing action or not, on the knowledge of the agent is specified by a so-called *knowledge update axiom*, [5]

$$\begin{array}{l} \mathit{Knows}(\mathit{Poss}(A(\vec{x})), s) \supset \\ \quad (\exists \vec{y})(\forall z')\,[\mathit{KState}(\mathit{Do}(A(\vec{x}), s), z') \equiv \\ \qquad\qquad (\exists z)(\mathit{KState}(s, z) \wedge \Psi(z', z) \wedge \Pi(\vec{y}, z', \mathit{Do}(A(\vec{x}), s)))\,] \end{array} \tag{24}$$

where Ψ specifies the physical state update while Π restricts the possible states so as to agree with the actual state $\mathit{State}(\mathit{Do}(A(\vec{x}), s))$ on the sensed properties and values $\vec{y}$.

As an example, let the three actions of the cleaning robot be denoted by

$$\begin{array}{rll} \mathit{Clean}: & \text{ACTION} & \text{empty waste bin at current location} \\ \mathit{Turn}: & \text{ACTION} & \text{turn clockwise by } 90^\circ \\ \mathit{Go}: & \text{ACTION} & \text{move forward to adjacent square} \end{array}$$

The action preconditions can be axiomatized as

$$\begin{array}{rcl} \mathit{Poss}(\mathit{Clean}, z) & \equiv & \top \\ \mathit{Poss}(\mathit{Turn}, z) & \equiv & \top \\ \mathit{Poss}(\mathit{Go}, z) & \equiv & (\forall d, x, y)\,(\,\mathit{Holds}(\mathit{At}(x, y), z) \wedge \mathit{Holds}(\mathit{Facing}(d), z) \\ & & \qquad \supset (\exists x', y')\,\mathit{Adjacent}(x, y, d, x', y')) \end{array} \tag{25}$$

in conjunction with the auxiliary axiom

$$\begin{array}{rcl} \mathit{Adjacent}(x, y, d, x', y') & \equiv & 1 \le d \le 4 \wedge 1 \le x, x', y, y' \le 5 \wedge \\ & & [\,d = 1 \wedge x' = x \wedge y' = y + 1 \vee \\ & & \;\; d = 2 \wedge x' = x + 1 \wedge y' = y \vee \\ & & \;\; d = 3 \wedge x' = x \wedge y' = y - 1 \vee \\ & & \;\; d = 4 \wedge x' = x - 1 \wedge y' = y\,] \end{array} \tag{26}$$

That is to say, going forward requires the robot not to face the wall of the building while emptying a waste bin and making a quarter turn clockwise is always possible.

The actions *Clean* and *Turn* of our cleanbot involve no sensing. The physical effects of these actions are specified by the following knowledge update axioms:

$$\begin{array}{l} \mathit{Knows}(\mathit{Poss}(\mathit{Clean}), s) \supset \\ \quad [\,\mathit{KState}(\mathit{Do}(\mathit{Clean}, s), z') \equiv \\ \qquad (\exists z)\,(\,\mathit{KState}(s, z) \wedge \\ \qquad\quad (\exists x, y)\,(\mathit{Holds}(\mathit{At}(x, y), z) \wedge z' = z + \mathit{Cleaned}(x, y))\,)\,] \\ \\ \mathit{Knows}(\mathit{Poss}(\mathit{Turn}), s) \supset \\ \quad [\,\mathit{KState}(\mathit{Do}(\mathit{Turn}, s), z') \equiv \\ \qquad (\exists z)\,(\,\mathit{KState}(s, z) \wedge \\ \qquad\quad (\exists d)\,(\mathit{Holds}(\mathit{Facing}(d), z) \wedge \\ \qquad\qquad z' = z - \mathit{Facing}(d) + \mathit{Facing}(d \bmod 4 + 1))\,)\,] \end{array} \tag{27}$$

[5] Below, the standard predicate Poss : ACTION × STATE denotes that an action is possible in a state. Macro $\mathit{Knows}(\mathit{Poss}(a), s)$ stands for the formula $(\forall z)\,(\mathit{KState}(s, z) \supset \mathit{Poss}(a, z))$.

Thus z' is a possible state after cleaning or turning, respectively, just in case z' is the result of cleaning or turning in one of the previously possible states z.

The following knowledge update axiom for Go combines the physical effect of going forward with information about whether light is sensed at the new location:

$$\begin{array}{l} Knows(Poss(Go), s) \supset \\ \quad [\, KState(Do(Go, s), z') \equiv \\ \qquad (\exists z)\,(\, KState(s, z) \wedge \\ \qquad\quad (\exists d, x, y, x', y')\,(\, Holds(At(x, y), z) \wedge \\ \qquad\qquad\qquad\qquad Holds(Facing(d), z) \wedge \\ \qquad\qquad\qquad\qquad Adjacent(x, y, d, x', y') \wedge \\ \qquad\qquad\qquad\qquad z' = z - At(x, y) + At(x', y'))\,) \wedge \\ \qquad [\, \Pi_{Light}(z') \equiv \Pi_{Light}(State(Do(Go, s)))\,]\,] \end{array} \tag{28}$$

where the sensed property indicates whether or not the robot perceives a light at its current location:

$$\Pi_{Light}(z) \stackrel{\text{def}}{=} (\exists x, y)\,(Holds(At(x, y), z) \wedge Light(x, y, z)) \tag{29}$$

Thus axiom (28) says that z' is a possible state after going forward if z' is the result of doing this action in some previously possible state and there is light at the current location in z' just in case it is so in the actual state $State(Do(Go, s))$.

As an example of sensing a fluent value rather than a proposition, consider the specification of a location sensor. As a pure sensing action, self-location has no physical effect. In general, this is indicated in a knowledge update axiom by the sub-formula $(\exists z)\,(KState(s, z) \wedge z' = z)$ describing the (empty) physical effect. For the sake of compactness, this sub-formula has been simplified to $KState(s, z')$ in the following axiom:

$$\begin{array}{l} Knows(Poss(SenseLoc), s) \supset \\ \quad (\exists x, y)(\forall z')\,(\, KState(Do(SenseLoc, s), z') \equiv \\ \qquad\qquad KState(s, z') \wedge Holds(At(x, y), z') \end{array} \tag{30}$$

Put in words, there exist coordinates x, y such that the robot is at (x, y) in all possible states of the successor situation. (The foundational axiom for knowledge of Definition 2 (Section 5.1) then implies that (x, y) must also be the actual location of the robot.)

5.4 Inferring knowledge update in FLUX

Updating the knowledge state of a FLUX agent involves two steps, the physical effect and the sensing result of an action. Since knowledge states are identified with (incomplete) FLUX states as discussed in Section 5.2, knowledge update according to the physical effect amounts to updating a FLUX state specification in the way discussed in Section 4. Having inferred the physical effect of an action, agents need to evaluate possible sensing results as part of the update. To this end, the sensing outcome of an action is encoded by a (possibly empty) list of individual *sensing results*. The result of sensing a proposition is either of the constants $True$ or

```
poss(clean, _).
poss(turn, _).
poss(go, Z) :-
   knows_val([X,Y], at(X,Y), Z),
   knows_val([D], facing(D), Z),
   adjacent(X, Y, D, _, _),

state_update(Z1, clean, Z2, []) :-
   holds(at(X,Y), Z1),
   update(Z1, [cleaned(X,Y)], [], Z2).

state_update(Z1, turn, Z2, []) :-
   holds(facing(D), Z1),
   (D#<4 #/\ D1#=D+1) #\/ (D#=4 #/\ D1#=1),
   update(Z1, [facing(D1)], [facing(D)], Z2).

state_update(Z1, go, Z2, [Light]) :-
   holds(at(X,Y), Z1),
   holds(facing(D), Z1),
   adjacent(X, Y, D, X1, Y1),
   update(Z1, [at(X1,Y1)], [at(X,Y)], Z2),
   light(X1, Y1, Light, Z2).

adjacent(X, Y, D, X1, Y1) :-
   [X,Y,X1,Y1] :: 1..5, D :: 1..4,
   (D#=1) #/\ (X1#=X) #/\ (Y1#=Y+1)      % north
   #\/
   (D#=2) #/\ (X1#=X+1) #/\ (Y1#=Y)      % east
   #\/
   (D#=3) #/\ (X1#=X) #/\ (Y1#=Y-1)      % south
   #\/
   (D#=4) #/\ (X1#=X-1) #/\ (Y1#=Y).     % west
```

Fig. 7. FLUX encoding of the precondition and update axioms for the cleanbot.

False. The result of sensing a value is a ground term of the respective sort. For example, the sensing result for knowledge update axiom (28) is encoded by $[\pi]$ where $\pi \in \{\mathit{True}, \mathit{False}\}$, depending on whether light is actually sensed at the new location. The sensing result for knowledge update axiom (30), on the other hand, should be encoded by $[x, y]$ where $x, y : \mathbb{N}$.

Based on the notion of sensing results, knowledge update axioms are encoded in FLUX as definitions of the predicate $\mathit{StateUpdate}(z_1, A(\vec{x}), z_2, y)$ describing the update of state z_1 to z_2 according to the physical effects of action $A(\vec{x})$ and the sensing result y. As an example, Figure 7 depicts a FLUX encoding of the action precondition and knowledge update axioms for the cleaning robot domain. Neither *Clean* nor *Turn* provides any sensor data. The sensing result for action *Go* is evaluated with the help of the auxiliary predicate *Light* as defined in Section 3.

Consider, for example, the initial FLUX state for the cleaning robot shown in

```
init(Z0) :-
   Z0 = [at(1,1),facing(1) | Z],
   not_holds(occupied(1,1), Z),
   not_holds(occupied(2,1), Z),         % hallway
   ..., not_holds(occupied(4,5), Z),    %
   consistent(Z0).

consistent(Z) :-
   holds(at(X,Y), Z, Z1), [X,Y] :: 1..5, not_holds_all(at(_,_), Z1),
   holds(facing(D), Z, Z2), [D] :: 1..4, not_holds_all(facing(_), Z2),
   not_holds_all(occupied(_,0), Z),
   not_holds_all(occupied(_,6), Z),
   not_holds_all(occupied(0,_), Z),
   not_holds_all(occupied(6,_), Z),
   duplicate_free(Z).
```

Fig. 8. Initial state specification for the cleanbot domain. The clause for *Consistent*(z) specifies general domain constraints, such as uniqueness of the robot's position and orientation.

Figure 8. Suppose that when going north twice, the robot senses no light after the first action but after the second one. With the following query the cleanbot computes the knowledge update for this sequence of actions and the given sensing results:

```
?- init(Z0), state_update(Z0, go, Z1, [false]),
              state_update(Z1, go, Z2, [true]).

Z0 = [at(1,1),facing(1) | Z]
Z1 = [at(1,2),facing(1) | Z]
Z2 = [at(1,3),facing(1) | Z]

Constraints:
not_holds(occupied(1,3), Z)
or_holds([occupied(2,3),occupied(1,4)], Z)
...
```

Thus the agent has evaluated the acquired sensor data and inferred its actual position according to the physical effects of *Go*.

As an example for inferring the update when sensing a value of a fluent, consider the following FLUX clause, which encodes knowledge update axiom (30) for action *SenseLoc*:

```
state_update(Z, sense_loc, Z, [X,Y]) :- holds(at(X,Y), Z).
```

That is, no physical effect affects the state but the sensed value is incorporated into the specification. Suppose, for instance, the agent is uncertain as to whether it

moved north or east from its initial location $(1,1)$, while the subsequent position tracking reveals that it is at $(1,2)$:

```
init(Z0) :- Z0 = [at(1,1),facing(D) | _], D#=1 #\/ D#=2,
            consistent(Z0).

?- init(Z0), state_update(Z0, go, Z1, [false]),
             state_update(Z1, sense_loc, Z2, [1,2]).

Z0 = [at(1,1),facing(1) | Z]
Z1 = [at(1,2),facing(1) | Z]
Z2 = [at(1,2),facing(1) | Z]

Constraints:
not_holds(occupied(1,3), Z)
...
```

Thus the agent has inferred its actual position and, hence, concluded that it is actually facing north. Incidentally, knowing the location also allows to infer that office $(1,3)$ is not occupied, which follows from the observation that no light is sensed after the *Go* action.

5.5 Defining knowledge update for actions with conditional effects

FLUX agents rely on knowledge update axioms in order to maintain their internal model of the environment. As this model is usually incomplete, the update axioms need to be carefully encoded in FLUX so as to always lead to a correct resulting knowledge state. In particular, when specifying an action with conditional effects the programmer needs to define the correct update for any possible knowledge the agent may have concerning the fluents affected by the action. Consider, for example, the action $Alter(x)$ to alter the position of a toggle switch. If x happens to be open (fluent $Open(x)$), then it will be closed afterwards (i.e., not $Open$); otherwise, i.e., if it is closed beforehand, then it will be open after the action. Tacitly assuming that the action is always possible, its conditional effect is specified in the fluent calculus by the following knowledge update axiom:

$$\begin{aligned} &KState(Do(Alter(x),s),z') \equiv \\ &\quad (\exists z)\,(\,KState(s,z) \wedge [\,Holds(Open(x),z) \wedge z' = z - Open(x) \\ &\qquad\qquad\qquad\qquad \vee \\ &\qquad\qquad\qquad\qquad \neg Holds(Open(x),z) \wedge z' = z + Open(x)\,]\,) \end{aligned} \tag{31}$$

The FLUX encoding of this update axiom requires to distinguish three kinds of knowledge states. In case the current knowledge entails that switch x is open, the resulting knowledge state is obtained through updating by negative effect $-Open(x)$. Conversely, in case the current knowledge entails that switch x is not open, the resulting knowledge state is obtained through updating by positive effect $+Open(x)$. Finally, if the current knowledge state does not entail the status of the

switch, then this uncertainty transfers to the updated knowledge state. Moreover, possible partial (e.g., disjunctive) information regarding the position of the affected switch is no longer valid and, hence, needs to be cancelled.

```
state_update(Z1, alter(X), Z2, []) :-
   knows(open(X), Z1)     -> update(Z1, [], [open(X)], Z2) ;
   knows_not(open(X), Z1) -> update(Z1, [open(X)], [], Z2) ;
   cancel(open(X), Z1, Z2).
```

For example,

```
?- not_holds(open(t1), Z0),
   or_holds([open(t2),open(t3)], Z0),
   state_update(Z0, alter(t1), Z1, []),
   state_update(Z1, alter(t2), Z2, []).

Z2 = [open(t1) | Z0]

Constraints:
not_holds(open(t1), Z0)
```

That is to say, while switch T_1 is known to be open after altering its position, it no longer follows, after altering T_2, that T_2 or T_3 is open. [6]

6 A FLUX control program for the cleaning robot

In this section, we show how our LP-based approach to reasoning about actions can be used as the kernel for a high-level programming method for the design of agents that reason about their actions. These agents use the concept of a state as their mental model of the world when controlling their own behavior. As they move along, agents constantly update their world model in order to reflect the changes they have effected and the sensor information they have acquired. 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 programs.

The general architecture of FLUX agent programs is depicted in Figure 9. Every agent program contains the kernel P_{kernel}, which consists of

[6] Actually, the inferred knowledge state in this example is slightly weaker than what is implied by knowledge update axiom (31). Suppose that initially T_2 or T_3 is open. Then it follows that after altering the position of T_2, if T_2 is open then so is T_3! This is so because if T_2 is open after changing its position, it must have been closed initially, and hence T_3 was (and still is) open. The corresponding implication, i.e., $\mathit{Holds}(\mathit{Open}(T_2), z_2) \supset \mathit{Holds}(\mathit{Open}(T_3), z_2)$, is not entailed by the updated FLUX state. Fortunately, obtaining a weaker update specification—just like an incomplete inference engine—is not an obstacle towards sound agent programs. Since FLUX agents are controlled by what they know of the environment, a sound but incomplete knowledge state suffices to ensure that the agent draws correct conclusions. This is a consequence of the simple fact that everything that is known under a weaker knowledge state is also known under the stronger one.

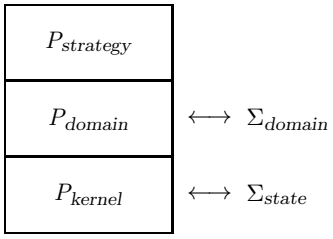

Fig. 9. The three components of FLUX agent programs.

- the FLUX constraint system of Figure 2 and 3 plus a constraint solver for finite domains;
- the definition of update of Figure 4 and 5;
- the definition of knowledge of Figure 6; and
- the following definition of execution, by which action a is performed and, simultaneously, the current state z_1 is updated to state z_2 according to the effects and sensing result of performing a:

```
execute(A, Z1, Z2) :-
   perform(A, Y), state_update(Z1, A, Z2, Y).
```

The second part, P_{domain}, of a FLUX agent program contains encodings of the domain axioms. These include

- action precondition axioms,
- update axioms,
- domain constraints, and
- initial knowledge state.

The domain program for the cleanbot, for example, consists of the precondition and update axioms of Figure 7 along with the initial knowledge state and domain constraints of Figure 8.

On top of this, the programmer defines the intended behavior of the agent via a control program $P_{strategy}$. This program uses the basic predicate $Execute(z_1, a, z_2)$ for the execution of an action. To this end, the interaction of the agent with the outside world needs to be defined by the predicate $Perform(a, y)$, which causes the physical agent to carry out action a in the environment such that y returns the sensing information acquired by performing this action. Control programs $P_{strategy}$ use the predicate $Knows(f, z)$ (and its derivatives $KnowsNot$ and $KnowsVal$) to evaluate conditions against the internal world model.

Figure 10 depicts a sample control program for our cleaning robot. After the initialization of the world model and the execution of a $Clean$ action at the home square, the main loop is entered by which the robot systematically explores and cleans the office floor. To this end, the program employs two parameters containing, respectively, choice points yet to be explored and the current path of the robot. The

```
main :-
   init(Z0),
   execute(clean, Z0, Z1),
   Choicepoints = [[1,2,3,4]], Backtrack = [],
   main_loop(Choicepoints, Backtrack, Z1).

main_loop([Choices|Choicepoints], Backtrack, Z) :-
   Choices = [Direction|Directions] ->
   ( go_in_direction(Direction, Z, Z1)
     -> execute(clean, Z1, Z2),
        Choicepoints1 = [[1,2,3,4], Directions | Choicepoints],
        Backtrack1 = [Direction | Backtrack],
        main_loop(Choicepoints1, Backtrack1, Z2)
     ;
     main_loop([Directions|Choicepoints], Backtrack, Z) )
   ;
   backtrack(Choicepoints, Backtrack, Z).

go_in_direction(D, Z1, Z2) :-
   knows_val([X,Y], at(X,Y), Z1),
   adjacent(X, Y, D, X1, Y1),
   \+ knows(cleaned(X1,Y1), Z1),
   knows_not(occupied(X1,Y1), Z1),
   turn_to_go(D, Z1, Z2).

backtrack(_, [], _).
backtrack(Choicepoints, [Direction|Backtrack], Z) :-
   Reverse is (Direction+1) mod 4 + 1,
   turn_to_go(Reverse, Z, Z1),
   main_loop(Choicepoints, Backtrack, Z1).

turn_to_go(D, Z1, Z2) :-
   knows(facing(D), Z1) -> execute(go, Z1, Z2)
   ;
   execute(turn, Z1, Z), turn_to_go(D, Z, Z2).
```

Fig. 10. A cleanbot agent in FLUX.

latter is used to backtrack from a location once all choices have been considered. A choice point is a list of directions, which are encoded by 1 (for north) to 4 (for west) as usual. The path is represented by the sequence, in reverse order, of the directions the robot took in each step.

In the main loop, the cleanbot selects the first element of the current choices. If the attempt to go into this direction is successful (predicate *GoInDirection*), then the robot empties the waste bin at the new location. A new choice point is created, and the backtrack path is augmented by the direction into which the robot just went. If, on the other hand, the chosen direction cannot be taken, then the main

loop is called with a reduced list of current choices. In case no more choices are left, the cleanbot backtracks (predicate *Backtrack*).

The auxiliary predicate *GoInDirection*(d, z_1, z_2) succeeds if the cleanbot can safely go into direction d from its current location in state z_1, ending up in state z_2. A direction is only explored if the adjacent square is inside of the boundaries. Furthermore, this location must not have been visited already (that is, it is not known to be cleaned), and—most importantly—the adjacent location must *known* not to be occupied. By the auxiliary predicate *Backtrack*, the robot takes back one step on its current path by reversing the direction. The program terminates once this path is empty, which implies that the robot has returned to its home after it has visited and cleaned as many locations as possible. The two auxiliary predicates *GoInDirection* and *Backtrack* in turn call the predicate *TurnToGo*, by which the robot makes turns until it faces the intended direction, and then moves forward.

The following table illustrates what happens in the first nine calls to the main loop when running the program with the initial state of Figure 8 and the scenario depicted in Figure 1.

At	Choicepoints	Backtrack	Actions
$(1,1)$	`[[1,2,3,4]]`	`[]`	GC
$(1,2)$	`[[1,2,3,4],[2,3,4]]`	`[1]`	GC
$(1,3)$	`[[1,2,3,4],[2,3,4],[2,3,4]]`	`[1,1]`	–
$(1,3)$	`[[2,3,4],[2,3,4],[2,3,4]]`	`[1,1]`	–
$(1,3)$	`[[3,4],[2,3,4],[2,3,4]]`	`[1,1]`	–
$(1,3)$	`[[4],[2,3,4],[2,3,4]]`	`[1,1]`	–
$(1,3)$	`[[],[2,3,4],[2,3,4]]`	`[1,1]`	TTG
$(1,2)$	`[[2,3,4],[2,3,4]]`	`[1]`	$TTTGC$
$(2,2)$	`[[1,2,3,4],[3,4],[2,3,4]]`	`[2,1]`	$TTTGC$

The letters G, C, T are abbreviations for the actions *Go*, *Clean*, and *Turn*, respectively. After going north twice to office $(1,3)$, the cleanbot cannot continue in direction 1 or 2 because both office $(1,4)$ and office $(2,3)$ may be occupied according to the robot's current knowledge. Direction 3 is not explored since location $(1,2)$ has already been cleaned, and direction 4 is ruled out as $(0,3)$ is outside of the boundaries. Hence, the cleanbot backtracks to $(1,2)$ and continues with the next choice there, direction 2, which brings it to location $(2,2)$. From there it goes north, and so on. Figure 11 depicts the knowledge state at the time the program terminates. Back home, the cleanbot has acquired knowledge of all four occupied offices. Moreover, it has emptied all waste bins but the ones in these four offices and the bin in office $(5,1)$. This office has not been visited because the robot cannot know that it is not occupied—the light sensors have been activated at both surrounding locations, $(4,1)$ and $(5,2)$!

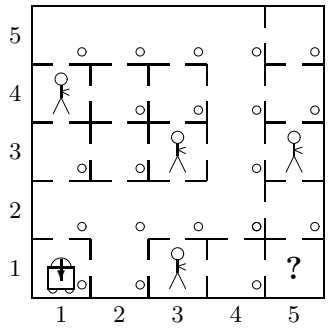

Fig. 11. The final knowledge state in the cleaning robot scenario. The small circles indicate the cleaned locations.

6.1 Semantics of FLUX programs

The semantics of a FLUX agent program is given as a combination of the fluent calculus and the standard semantics of logic programming. We assume the reader to be familiar with the basic notion of a computation tree for constraint logic programs (see, e.g., (Jaffar and Maher 1994)).

Let T be the computation tree for an agent program $P_{strategy} \cup P_{domain} \cup P_{kernel}$ along with a query $\{\leftarrow Q\}$. Tree T determines a particular action sequence as follows. Let an *execution node* be any node in T which starts with the atom *Execute*. Let $Execute(\alpha_1, \_, \_), Execute(\alpha_2, \_, \_), \ldots$ be the ordered sequence of all execution nodes occurring in T, then this tree is said to *generate* the action sequence $\alpha_1, \alpha_2, \ldots$ This sequence is to be used when proving formal properties of the agent program with the help of the fluent calculus and the axiomatization Σ_{domain} of the application domain. For example, a program can be called *sound* if the domain axiomatization entails that all actions are possible in the situation in which they are executed. Formally,

$$\Sigma_{state} \cup \Sigma_{domain} \models Poss(\alpha_1, S_0) \wedge Poss(\alpha_2, Do(\alpha_1, S_0)) \wedge \ldots$$

Domain-dependent requirements are proved in a similar fashion. The program for the cleanbot, for example, can be shown to admit a finite computation tree; hence to terminate. Other properties are that the cleanbot will always end up in its home $(1, 1)$, it will never enter an office which is occupied (provided its light sensor functions correctly), and it always cleans all locations in the hallway. The formal proofs of these properties are not deep but tedious, which is why we refrain from giving them here.

6.2 Computational Behavior

To illustrate the computational merits of FLUX, we have compared it to GOLOG (Levesque et al. 1997), an agent programming language with similar purposes. The

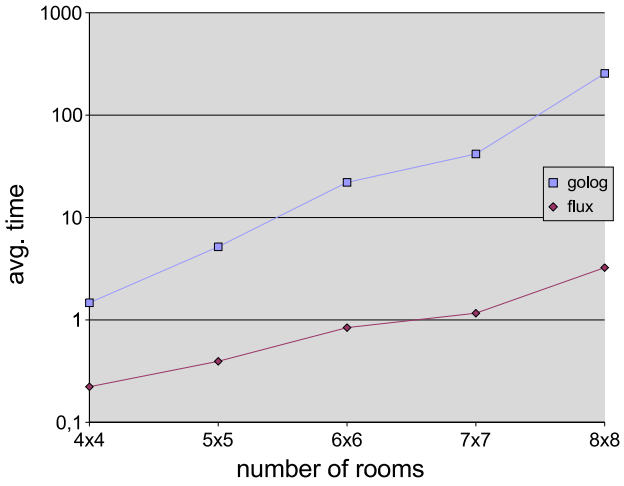

Fig. 12. Experimental results with the cleanbot control program in FLUX and GOLOG. (Notice the exponential scale on the vertical axis.)

cleanbot domain requires a variant of GOLOG which supports incomplete states and sensing (Reiter 2001b). In this system, incompletely specified initial situations are encoded by sets of (propositional) prime implicates. To decide whether a property is known to hold after a sequence of actions, the property is *regressed* to the initial situation. If the resulting formula is entailed by the initial prime implicates, then the original property is known to hold in the respective situation. Acquired sensor information is regressed, too, and the result is added to the initial set of prime implicates.

We have re-implemented the strategy of Figure 10 for the cleanbot as a GOLOG program and ran a series of experiments with square office floors of different size. For simplicity, no initial information about unoccupied cells besides (1,1) and the two adjacent ones were given to the robot. Figure 12 depicts the results of five sets of experiments. The given runtimes (seconds CPU time of a 1733 MHz processor) are the average of 10 runs with randomly chosen occupied cells.

The dominance of FLUX has two main reasons:

1. Since prime implicates can be used to encode arbitrary propositional formulas, the complexity of inferring knowledge in the GOLOG system of (Reiter 2001b) is exponential. In contrast, the restricted first-order state representation and the incomplete inference engine of FLUX allows for inferring knowledge in linear time.
2. In FLUX, the world model is *progressed* whenever an action is performed, and the new model is directly used to decide whether a property is currently known. The GOLOG system of (Reiter 2001b), on the other hand, is

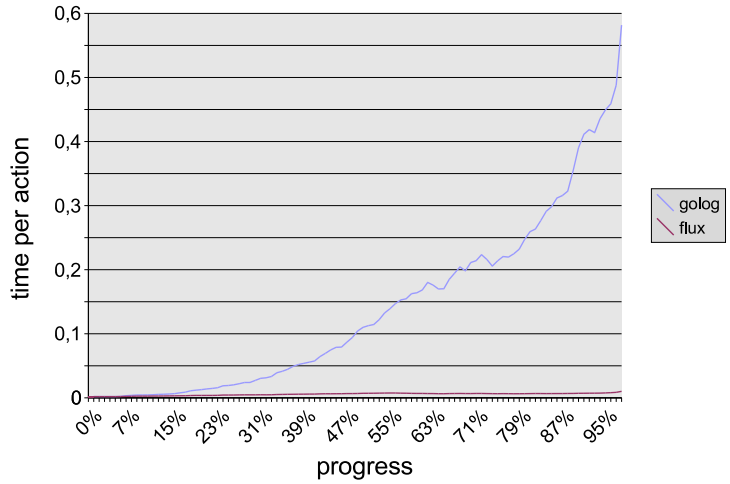

Fig. 13. Growth of the action selection time as the execution of the cleanbot program proceeds (averaged over 10 runs with 6×6 rooms).

regression-based, so that deciding whether a property is known in a situation requires to regress the property through the previously performed actions. Consequently, the computational behavior of the GOLOG program worsens the longer the program runs. This can be clearly seen from the graphs in Figure 13, which depict the average time for action selection at different stages of the execution of the cleanbot program.

3. To solve the frame problem, FLUX uses state update axioms, which specify the effects of an action on an entire state. When progressing a state through an update axiom, the large body of unaffected knowledge simply remains in the constraint store. This is what makes up an efficient solution to the frame problem even in the presence of incomplete states.

7 Discussion

We have presented the logic programming method FLUX for the design of logically reasoning agents. The agents use a system of Constraint Handling Rules and finite domain constraints to reason about actions in the presence of incomplete states. Both the constraint solver and the logic program for state update have been formally verified against the action theory of the fluent calculus. Thanks to a carefully chosen expressiveness, the FLUX kernel exhibits excellent computational behavior.

The closest related work is the programming language GOLOG (Levesque et al. 1997) for dynamic domains, which is based on the situation calculus and successor state axioms as a solution to the frame problem (Reiter 1991). The main differences are:

1. GOLOG defines a special programming language for strategies, while FLUX strategies are standard logic programs.
2. With the exception of (Reiter 2001b), existing implementations of GOLOG apply the principle of negation-as-failure to state specifications and, hence, are restricted to complete state knowledge and deterministic actions. With its underlying constraint solver, FLUX provides a natural way of representing and reasoning with incomplete states as well as nondeterministic actions.
3. The logic programs for GOLOG described in the literature all apply the principle of regression to evaluate conditions in agent programs. While this is efficient for short action sequences, the computational effort increases with the number of performed actions. With the progression principle, FLUX programs scale up well to the control of agents over extended periods. [7] Moreover, progression through state update axioms in FLUX provides an efficient solution to the frame problem, because unaffected state knowledge simply remains in the constraint store.
4. GOLOG includes the concept of nondeterministic programs as a means to define a search space for a planning problem. To find a plan, such a program is executed "off-line" with the aim to find a run by which the planning goal is attained. A similar concept can be added to FLUX, allowing agents to interleave planning with program execution (Thielscher 2002).

We are conducting experiments where FLUX is applied to the high-level control of a real robot, whose task is to collect and deliver in-house mail in an office floor (Fichtner et al. 2003). To this end, the logic programming system has been extended by a solution to the qualification problem (McCarthy 1977) in the fluent calculus which accounts for unexpected failure of actions (Thielscher 2001; Martin and Thielscher 2001). Future work will include the gradual extension of the expressiveness of FLUX, e.g., by constraints for exclusive disjunction, without loosing the computational merits of the approach.

Acknowledgments

The author wants to thank Stephan Schiffel for his help with the experiments and Matthias Fichtner, Axel Großmann, Yves Martin, and the anonymous reviewers for valuable comments on an earlier version. 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.

References

Baker, A. B. 1989. A simple solution to the Yale Shooting problem. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*

[7] In order to achieve a similar behavior, GOLOG would have to be reimplemented by appealing to the definition of progression in the situation calculus of (Lin and Reiter 1997), which, however, is not first-order definable in general.

(KR), R. Brachman, H. Levesque, and R. Reiter, Eds. Morgan Kaufmann, Toronto, Canada, 11–20.

Bibel, W. 1986. A deductive solution for plan generation. *New Generation Computing 4*, 115–132.

Fichtner, M., Grossmann, A., and Thielscher, M. 2003. Intelligent execution monitoring in dynamic environments. *Fundamenta Informaticae 57,* 2–4, 371–392.

Frühwirth, T. 1998. Theory and practice of constraint handling rules. *Journal of Logic Programming 37,* 1–3, 95–138.

Hentenryck, P. V. 1989. *Constraint Satisfaction in Logic Programming.* MIT Press.

Hölldobler, S. and Schneeberger, J. 1990. A new deductive approach to planning. *New Generation Computing 8*, 225–244.

Jaffar, J. and Maher, M. J. 1994. Constraint Logic Programming: A Survey. *Journal of Logic Programming 19/20*, 503–581.

Kowalski, R. and Sergot, M. 1986. A logic based calculus of events. *New Generation Computing 4*, 67–95.

Lespérance, Y., Levesque, H., Lin, F., Marcu, D., Reiter, R., and Scherl, R. 1994. 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*, B. Kuipers, Ed. New Orleans, LA, 109–119.

Levesque, H., Reiter, R., Lespérance, Y., Lin, F., and Scherl, R. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming 31,* 1–3, 59–83.

Lin, F. and Reiter, R. 1997. How to progress a database. *Artificial Intelligence 92*, 131–167.

Martin, Y. and Thielscher, M. 2001. Addressing the qualification problem in FLUX. In *Proceedings of the German Annual Conference on Artificial Intelligence (KI)*, F. Baader, G. Brewka, and T. Eiter, Eds. LNAI, vol. 2174. Springer, Vienna, Austria, 290–304.

McCarthy, J. 1963. *Situations and Actions and Causal Laws.* Stanford Artificial Intelligence Project, Memo 2, Stanford University, CA.

McCarthy, J. 1977. Epistemological problems of artificial intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, R. Reddy, Ed. MIT Press, Cambridge, MA, 1038–1044.

Moore, R. 1985. A formal theory of knowledge and action. In *Formal Theories of the Commonsense World*, J. R. Hobbs and R. C. Moore, Eds. Ablex, 319–358.

Reiter, R. 1991. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In *Artificial Intelligence and Mathematical Theory of Computation*, V. Lifschitz, Ed. Academic Press, 359–380.

Reiter, R. 2001a. *Knowledge in Action.* MIT Press.

Reiter, R. 2001b. On knowledge-based programming with sensing in the situation calculus. *ACM Transactions on Computational Logic 2,* 4, 433–457.

Scherl, R. and Levesque, H. 2003. Knowledge, action, and the frame problem. *Artificial Intelligence 144,* 1, 1–39.

Shanahan, M. 1997. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia.* MIT Press.

Shanahan, M. and Witkowski, M. 2000. High-level robot control through logic. In *Proceedings of the International Workshop on Agent Theories Architectures and Languages (ATAL)*, C. Castelfranchi and Y. Lespérance, Eds. LNCS, vol. 1986. Springer, Boston, MA, 104–121.

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

Thielscher, M. 2000. Representing the knowledge of a robot. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, A. Cohn, F. Giunchiglia, and B. Selman, Eds. Morgan Kaufmann, Breckenridge, CO, 109–120.

Thielscher, M. 2001. The qualification problem: A solution to the problem of anomalous models. *Artificial Intelligence 131,* 1–2, 1–37.

Thielscher, M. 2002. Programming of reasoning and planning agents with FLUX. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, D. Fensel, D. McGuinness, and M.-A. Williams, Eds. Morgan Kaufmann, Toulouse, France, 435–446.