Inferring Implicit State Knowledge and Plans with Sensing Actions

Michael Thielscher

Dresden University of Technology

Abstract. An effective method is presented for deriving state knowledge in the presence of sensing actions. It is shown how conditional plans can be inferred with the help of a generalized concept of plan skeletons as search heuristics, which allow the planner to introduce conditional branching points by need.

1 Introduction

The problem of modeling sensing actions has gained much attention in the recent past as an important step for the development of extensive foundations for Cognitive Robotics. Several solutions to the technical Frame Problem have been generalized to reasoning about the knowledge of a robot and the effect of sensing, e.g., in the Situation Calculus [20] and the Fluent Calculus [23]. Based on general first-order logic, these approaches are sufficiently expressive to allow for modeling actions with knowledge preconditions, sensing of non-atomic properties, and deriving implicit knowledge. Moreover, to solve planning problems involving knowledge goals, the notion of conditional plans has been integrated [11, 23] since it may be necessary to plan ahead different action sequences for different outcomes of sensing [14].

The expressiveness of general theories for conditional planning, on the other hand, raises the challenge to evolve inference algorithms that efficiently deal with the modality of knowledge. Most existing planning methods are tailored to restricted classes of planning problems, e.g., [9, 7, 2, 16, 11]. In particular, none of these systems can solve planning problems where knowledge follows *implicitly*: A well-known example is to determine acidity of a chemical solution by sensing the color of a Litmus strip [18]. The only existing system with a general solution to the Frame Problem for knowledge is [19] based on GOLOG [15]. However, this Prolog implementation is not meant for planning with sensing as it does not allow to *search* for suitable sensing actions. Rather, the user is supposed to provide GOLOG programs where all necessary sensing actions have been correctly planned. This restriction to plan verification applies to other existing approaches as well, such as [10].

In this paper, we present the foundations for an effective, fully automatic reasoning system capable of solving planning problems which require conditional plans and implicit knowledge and which may involve incomplete states, nondeterministic actions, and knowledge preconditions as well as knowledge goals.

Based on the recent solution to the Frame Problem for knowledge in the Fluent Calculus [23], our main technical result is a proof that, under reasonable assumptions, knowledge can be identified with incomplete state specifications. This theorem is applied to FLUX (the Fluent Calculus Executor)—a recent logic programming methodology for Cognitive Robotics [22] with similar motivations as GOLOG [15] but where state update axioms [21] are used to solve the inferential Frame Problem [3] and where constraints are used for encoding incomplete states.

Since conditional planning is a highly complex search problem, we also adapt the heuristics of nondeterministic robot programs of GOLOG [15] and develop a generalization which allows to search for plans in the presence of sensing. Conditionals occurring in the plan skeleton are evaluated at planning time only if the state knowledge suffices to do so; otherwise, a branching point is introduced, leading to a conditional plan by need. Prior to presenting the results, we give a brief introduction to the basic Fluent Calculus and FLUX.

2 The Fluent Calculus for Knowledge and Sensing

2.1 State Update Axioms

The basic Fluent Calculus combines, in pure classical logic, the Situation Calculus with a STRIPS-like solution to the representational and inferential Frame Problem [21]. The standard sorts ACTION and SIT (i.e., situations) are inherited from the Situation Calculus [13] along with the standard functions $S_0 : \text{SIT}$ and $Do : \text{ACTION} \times \text{SIT} \mapsto \text{SIT}$ denoting, resp., the initial situation and the successor situation after performing an action; furthermore, the standard predicate $Poss : \text{ACTION} \times \text{SIT}$ denotes whether an action is possible in a situation. To this the Fluent Calculus adds the sort STATE with sub-sort FLUENT along with the pre-defined functions $\emptyset : \text{STATE}$; $\circ : \text{STATE} \times \text{STATE} \mapsto \text{STATE}$; and $State : \text{SIT} \mapsto \text{STATE}$; denoting, resp., the empty state, the union of two states, and the state of the world in a situation. Based on this signature, the Fluent Calculus provides a rigorously logical account of the concept of a state being characterized by the set of fluents that are true in the state. The following foundational axioms serve this purpose. They are a suitable subset of the Zermelo-Fraenkel axioms, stipulating that function $\circ$ behaves like set union with $\emptyset$ as the empty set:[1]

$$
\begin{array}{rl}
z_1 \circ (z_2 \circ z_3) = (z_1 \circ z_2) \circ z_3 & \neg Holds(f, \emptyset) \\
z_1 \circ z_2 = z_2 \circ z_1 & Holds(f_1, f) \supset f = f_1 \\
z \circ z = z & Holds(f, z_1 \circ z_2) \supset Holds(f, z_1) \vee Holds(f, z_2) \\
z \circ \emptyset = z & (\forall f)\,(Holds(f, z_1) \equiv Holds(f, z_2)) \supset z_1 = z_2 \\
\multicolumn{2}{c}{(\forall \Phi)(\exists z)(\forall f)\,(Holds(f, z) \equiv \Phi(f))}
\end{array}
$$

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

where Φ is a second-order predicate variable of sort FLUENT and the macro *Holds* means that a fluent is contained in a state:

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

The very last one of the foundational axioms above stipulates the existence of a state for all possible combinations of fluents. A second macro, which reduces to (1), is used for fluents holding in situations:

$$Holds(f, s) \stackrel{\text{def}}{=} Holds(f, State(s))$$

Consider, e.g., the FLUENT terms $OnTable(x)$, $Acidic(x)$, $Carries(x)$, and $Red(y)$, denoting, resp., whether a chemical solution x is on the table, x is acidic, the robot carries x, and Litmus strip y is red.[2] The following incomplete state specification says that initially there are three chemical solutions A, B, and C on the table, litmus paper P is not red, the robot carries nothing, and either B or C is not acidic:

$$\begin{array}{l} Holds(OnTable(A), S_0) \wedge Holds(OnTable(B), S_0) \wedge Holds(OnTable(C), S_0) \\ \wedge \neg Holds(Red(P), S_0) \wedge (\forall x)\, \neg Holds(Carries(x), S_0) \\ \wedge [\neg Holds(Acidic(B), S_0) \vee \neg Holds(Acidic(C), S_0)] \end{array} \tag{2}$$

Assuming uniqueness of names for all fluents, the macro definitions and the foundational axioms imply that (2) is equivalent to

$$\begin{array}{l} (\exists z)\, (\, State(S_0) = OnTable(A) \circ OnTable(B) \circ OnTable(C) \circ z \\ \quad \wedge \neg Holds(Red(P), z) \wedge (\forall x)\, \neg Holds(Carries(x), z) \\ \quad \wedge [\neg Holds(Acidic(B), z) \vee \neg Holds(Acidic(C), z)] \\ \quad \wedge \neg Holds(OnTable(A), z) \wedge \neg Holds(OnTable(B), z) \\ \quad \wedge \neg Holds(OnTable(C), z)\,) \end{array} \tag{3}$$

The reader may notice that the constraints on sub-STATE z not only reflect the negated *Holds* statements of (2) but also the fact that neither of $OnTable(A)$, $OnTable(B)$, or $OnTable(C)$ re-occurs. This will allow to quickly infer the result of removing any of these fluents from $State(S_0)$ as a negative effect.

The Frame Problem is solved in the Fluent Calculus using so-called state update axioms, which specify the difference between the states before and after an action. The axiomatic characterization of negative effects, i.e., facts that become false, is given by this inductive abbreviation, which generalizes STRIPS-style update [4] to incomplete states:

$$z' = z - f \;\stackrel{\text{def}}{=}\; [z' \circ f = z \vee z' = z] \wedge \neg Holds(f, z')$$

$$\begin{array}{l} z' = z - (f_1 \circ \ldots \circ f_n \circ f_{n+1}) \stackrel{\text{def}}{=} \\ \quad (\exists z'')\, (z'' = z - (f_1 \circ \ldots \circ f_n) \wedge z' = z'' - f_{n+1}) \end{array}$$

[2] This scenario, which will be used throughout the paper, is a variation of an example first used in [18].

On this basis, the following is the general form of a state update axiom for a (possibly nondeterministic) action $A(\vec{x})$ with a bounded number of (possibly conditional) effects:

$$\begin{aligned} Poss(A(\vec{x}), s) \supset\ & (\exists \vec{y}_1)\,(\Delta_1 \wedge State(Do(A(\vec{x}), s)) = (State(s) \circ \vartheta_1^+) - \vartheta_1^-) \\ & \vee \ldots \vee \\ & (\exists \vec{y}_n)\,(\Delta_n \wedge State(Do(A(\vec{x}), s)) = (State(s) \circ \vartheta_n^+) - \vartheta_n^-) \end{aligned}$$

where the sub-formulas $\Delta_i(\vec{x}, \vec{y}_i, State(s))$ specify the conditions on $State(s)$ under which $A(\vec{x})$ has the positive and negative effects ϑ_i^+ and ϑ_i^-, resp. Both ϑ_i^+ and ϑ_i^- are STATE terms composed of fluents with variables among $\vec{x}, \vec{y}_i$. If $n = 1$ and $\Delta_1 \equiv True$, then action $A(\vec{x})$ does not have conditional effects. If $n > 1$ and the conditions Δ_i are not mutually exclusive, then the action is nondeterministic.

Consider, e.g., the ACTION terms $Take(x)$ and $Test(x, y)$ denoting, resp., the robot taking x off the table and testing x by inserting Litmus paper y. The effects of these two actions can be defined by these state update axioms:

$$\begin{aligned} & Poss(Take(x), s) \supset \\ & \quad State(Do(Take(x), s)) = (State(s) \circ Carries(x)) - OnTable(x) \\ & \\ & Poss(Test(x, y), s) \supset \\ & \quad [Holds(Acidic(x), s) \wedge State(Do(Test(x, y), s)) = State(s) \circ Red(y)\,] \vee \\ & \quad [\neg Holds(Acidic(x), s) \wedge State(Do(Test(x, y), s)) = State(s)\,] \end{aligned} \tag{4}$$

Put in words, taking x has the effect that the robot carries x and x is no longer on the table; and testing x with the help of Litmus paper y causes y to turn red if the solution is acidic, otherwise nothing changes. The action preconditions shall be defined by:

$$\begin{aligned} Poss(Take(x), s) &\equiv Holds(OnTable(x), s) \\ Poss(Test(x, y), s) &\equiv True \end{aligned}$$

Recall formula (3). The state update axiom for $Take(x)$ and the foundational axioms imply

$$\begin{aligned} (\exists z)\,(\, & State(Do(Take(A), S_0)) = OnTable(B) \circ OnTable(C) \circ z \circ Carries(A) \\ & \wedge \neg Holds(OnTable(A), z)\,) \end{aligned}$$

Besides the positive effect $Carries(A)$, the right hand side of the equation includes all fluents which are not affected by the action. Moreover, facts given in (3) as to which fluents do not hold in z apply to the new state just as well as it includes z. Thus all unchanged knowledge continues to hold without the need to apply extra inference steps.

2.2 FLUX

The programming language FLUX is a recent implementation of the Fluent Calculus based on Constraint Logic Programming [22]. Its distinguishing feature is to support incomplete states, which are modeled by open lists of the form

```
Z0 = [F1,...,Fm | Z]
```

(encoding the state description $\texttt{Z0} = \texttt{F1} \circ \ldots \circ \texttt{Fm} \circ \texttt{Z}$), along with constraints

```
not_holds(F, Z)
not_holds_all([X1,...,Xk], F, Z)
```

encoding, resp., the negative statements $(\exists \vec{y})\, \neg \mathit{Holds}(\texttt{F}, \texttt{Z})$ (where $\vec{y}$ are the variables occurring in F) and $(\exists \vec{y})(\forall \texttt{X1}, \ldots, \texttt{Xk})\, \neg \mathit{Holds}(\texttt{F}, \texttt{Z})$ (where $\vec{y}$ are the variables occurring in F except $\texttt{X1}, \ldots, \texttt{Xk}$). These two constraints are used to bypass the problem of 'negation-as-failure' with incomplete states. In order to process these constraints, so-called declarative Constraint Handling Rules [5] have been defined and proved correct under the foundational axioms of the Fluent Calculus. In addition, the core of FLUX contains definitions for `holds(F,Z)`, by which is encoded macro (1), and `update(Z1,ThetaP,ThetaN,Z2)`, which encodes the state equation $\texttt{Z2} = (\texttt{Z1} \circ \texttt{ThetaP}) - \texttt{ThetaN}$. The following, for instance, is the FLUX encoding of our state update axioms (4) (ignoring preconditions) and the initial specification (2):

```
state_update(Z1, take(X), Z2) :-
  update(Z1, [carries(X)], [on_table(X)], Z2).

state_update(Z1, test(X,Y), Z2) :-
  holds(acidic(X), Z1), update(Z1, [red(Y)], [], Z2) ;
  not_holds(acidic(X), Z1), update(Z1, [], [], Z2).

init(Z0) :-
   holds(on_table(a), Z0),
   holds(on_table(b), Z0), holds(on_table(c), Z0),
   not_holds(red(p), Z0), not_holds_all([X], carries(X), Z0),
   (not_holds(acidic(b), Z0) ; not_holds(acidic(c), Z0)),
   duplicate_free(Z0).
```

where the constraint `duplicate_free(Z)` means that list `Z` does not contain multiple occurrences. Suppose, e.g., that Litmus paper P is red after testing solution B, then it follows that B must have been acidic but not C:

```
?- init(Z0), state_update(Z0, test(b,p), Z1), holds(red(p), Z1).

Z0 = [on_table(a),on_table(b),on_table(c),acidic(b) | _Z]
Constraints:
not_holds(acidic(a), _Z)
...
```

2.3 Knowledge Update Axioms

To represent knowledge in the Fluent Calculus and to reason about sensing actions, the predicate $KState : \text{SIT} \times \text{STATE}$ has been introduced in [23]. An instance $KState(s, z)$ means that according to the knowledge of the planning robot, z is a possible state in situation s. A fluent is then known to hold (resp. not hold) in a situation just in case it is true (resp. false) in all possible states; and it is known whether a fluent holds just in case it is known to hold or known not to hold:

$$\begin{aligned} Knows(f, s) &\stackrel{\text{def}}{=} (\forall z)\,(KState(s, z) \supset Holds(f, z)) \\ Knows(\neg f, s) &\stackrel{\text{def}}{=} (\forall z)\,(KState(s, z) \supset \neg Holds(f, z)) \\ Kwhether(f, s) &\stackrel{\text{def}}{=} Knows(f, s) \vee Knows(\neg f, s) \end{aligned} \tag{5}$$

These macros generalize to the knowledge of arbitrary non-atomic formulas in a natural way. A foundational axiom stipulates correctness of state knowledge:

$$KState(s, State(s))$$

The Frame Problem for knowledge is solved by axioms that determine the relation between the possible states before and after an action. More formally, the effect of an action $A(\vec{x})$, be it sensing or not, on the knowledge is specified by a *knowledge update axiom*,

$$\begin{aligned} &Poss(A(\vec{x}), s) \supset \\ &\quad (\forall z)\,(KState(Do(A(\vec{x}), s), z) \equiv (\exists z')(KState(s, z') \wedge \Psi(z, z', s))\,) \end{aligned} \tag{6}$$

In case of non-sensing actions, formula Ψ defines what the robot knows of the effects of the action. In case of sensing actions, formula Ψ restricts the possible states in such a way that the sensed property becomes known. In particular, let the generic ACTION term $Sense(f)$ denote sensing whether a fluent f holds, then:

$$\begin{aligned} &Poss(Sense(f), s) \supset \\ &\quad KState(Do(Sense(f), s), z) \equiv \\ &\qquad KState(s, z) \wedge [Holds(f, z) \equiv Holds(f, s)] \end{aligned} \tag{7}$$

That is to say, among the states possible in s only those are still possible after sensing which agree with the actual state of the world as far as the sensed fluent is concerned. A crucial immediate consequence is that sensing always causes the truth value of a property to be known [23].[3]

Based on knowledge update axioms, the inferential Frame Problem for knowledge is solved with the help of a simple inference schema. Suppose given an axiom which summarizes *all* that is known of a situation s, that is, $KState(s, z) \equiv \Phi(z)$. Suppose further, for the sake of argument, that $Poss(A(\vec{x}), s)$, then (6) entails

$$KState(Do(A(\vec{x}), s), z) \equiv (\exists z')\,(\Phi(z') \wedge \Psi(z, z', s)) \tag{8}$$

[3] While for the sake of abstraction axiom (7) specifies an ideal sensor for qualitative fluents, nondeterministic knowledge update axioms can be used to model sensor noise when sensing quantitative fluents.

which provides a specification of what is known in the successor situation.

In accordance with the classical notion of planning by deduction, conditional plans in the Fluent Calculus are first-order citizens, composed of the primitive actions of a domain and using the standard functions ϵ (empty action), $a_1; a_2$ (sequential composition), and $If(f, a_1, a_2)$ (conditional branching). Preconditions, state update, and knowledge update for these ACTION functions are defined by foundational axioms [23]. Here is an example of a situation representing a conditional plan:

$$\begin{aligned} S \;=\; Do(\, & If(Red(P), Take(C), Take(B)), \\ & Do(Sense(Red(P)), Do(Test(B, P), S_0))) \end{aligned} \tag{9}$$

Applied to our example initial specification, (2), this plan can be proved to achieve the goal of getting a chemical solution which is known not to be acidic:

$$(\exists x)\,(Knows(Carries(x), S) \wedge Knows(\neg Acidic(x), S))$$

3 Identifying Knowledge with Incomplete States

While the explicit notion of possible states leads to an extensive framework for reasoning about knowledge and sensing, automated deduction becomes considerably more intricate by the introduction of the modality-like *KState* predicate. In this section, we develop the foundations for an inference method which avoids separate update of knowledge and states. To this end, we show how knowledge updates are implicitly obtained by progressing an incomplete state through state update axioms.

Our approach rests on two assumptions. First, the planning robot needs to know the given initial specification $\Phi(State(S_0))$, and this is all it knows of S_0, that is, $KState(S_0, z) \equiv \Phi(z)$. Second, the robot must have accurate knowledge of its own actions:

Definition 1. A set of axioms Σ *represents accurate effect knowledge* if for each non-sensing ACTION function A, Σ contains a unique state update axiom

$$Poss(A(\vec{x}), s) \supset \Gamma_A\{z/State(Do(A(\vec{x}), s)), z'/State(s))\} \tag{10}$$

(where $\Gamma_A(\vec{x}, z, z')$ is a first-order formula with free variables among $\vec{x}, z, z'$ and without a sub-term of sort SIT) and a unique knowledge update axiom which is equivalent to

$$\begin{aligned} & Poss(A(\vec{x}), s) \supset \\ & \quad (\forall z)\,(KState(Do(A(\vec{x}), s), z) \equiv \\ & \qquad (\exists z')(KState(s, z') \wedge \Gamma_A(\vec{x}, z, z'))\,) \end{aligned} \tag{11}$$

Put in words, the possible states after a non-sensing action are those which would be the result of actually performing the action in one of the previously possible states.

Accurate knowledge of effects suffices to ensure that the possible states after a non-sensing action can be obtained by progressing a given state specification through the state update axiom for that action. The effect of sensing, on the other hand, cannot be obtained in the same fashion. To see why, let S be a situation and consider the knowledge specification

$$KState(S, z) \equiv [Holds(Red(P), z) \equiv Holds(Acidic(A), z)] \tag{12}$$

(which may have been inferred as the result of a $Test(A, P)$ action). Suppose that $Poss(Sense(Red(P)), S)$, then the knowledge update axiom (7) for $Sense(Red(P))$ yields two models for $KState(Do(Sense(Red(P)), S), z)$, the first of which satisfies

$$\begin{aligned} &KState(Do(Sense(Red(P)), S), z) \equiv \\ &\quad Holds(Red(P), z) \wedge Holds(Acidic(A), z) \end{aligned}$$

(for all z) whereas the other one satisfies

$$\begin{aligned} &KState(Do(Sense(Red(P)), S), z) \equiv \\ &\quad \neg Holds(Red(P), z) \wedge \neg Holds(Acidic(A), z) \end{aligned}$$

(again for all z). The first model represents the case where $Red(P)$ actually holds in S while the second model represents the case where $Red(P)$ actually does not hold in S. Due to the existence of these two models there can be no unique specification of the form $KState(Do(Sense(Red(P)), S), z) \equiv \Phi(z)$ entailed by (12) and (7). Hence, the effect of a sensing action cannot be obtained by straightforward progression.

In order to account for different models for $KState$ caused by sensing, we introduce the notion of a *sensing history* ς as a finite, possibly empty list of 0's and 1's. A history is meant to describe the outcome of each sensing action in a sequence of actions. For the sake of simplicity, we assume that the only sensing action is the generic $Sense(f)$ with knowledge update axiom (7) and state update axiom $State(Do(Sense(f), s)) = State(s)$.

For the formal definition of progression we also need the notion of an *action sequence* σ as a finite, possibly empty list of ground ACTION terms. An action sequence corresponds naturally to a situation, which we denote by S_σ:

$$S_{[\,]} \stackrel{\text{def}}{=} S_0 \quad \text{and} \quad S_{[A(\vec{t}) \,|\, \sigma]} \stackrel{\text{def}}{=} Do(A(\vec{t}), S_\sigma)$$

We are now in a position to define, inductively, a *progression operator* $\mathcal{P}(\sigma, \varsigma, z)$, by which an initial state specification $\Phi(State(S_0))$ is progressed through an action sequence σ wrt. a sensing history ς, resulting in a formula specifying z:

$$\mathcal{P}([\,], \varsigma, z) \stackrel{\text{def}}{=} \Phi(z) \quad \text{if } \varsigma = [\,] \tag{13}$$

$$\begin{aligned} \mathcal{P}([A(\vec{t}) \,|\, \sigma], \varsigma, z) \stackrel{\text{def}}{=} (\exists z')\,(\mathcal{P}(\sigma, \varsigma, z') \wedge \Gamma_A(\vec{t}, z, z')) \\ \text{if } A \text{ non-sensing with state update (10)} \end{aligned} \tag{14}$$

$$\begin{aligned} \mathcal{P}([Sense(f) \,|\, \sigma], \varsigma, z) \stackrel{\text{def}}{=} \begin{array}{ll} \mathcal{P}(\sigma, \varsigma', z) \wedge \neg Holds(f, S_\sigma) & \text{if } \varsigma = [0 \,|\, \varsigma'] \\ \mathcal{P}(\sigma, \varsigma', z) \wedge Holds(f, S_\sigma) & \text{if } \varsigma = [1 \,|\, \varsigma'] \end{array} \end{aligned} \tag{15}$$

In case the length of the history ς does not equal the number of sensing actions in σ, we define $\mathcal{P}(\sigma, \varsigma, z)$ as *False*. As the main result, progression provides a provably correct inference method for knowledge update.[4]

Theorem 2. *Consider the initial state and knowledge* $\Sigma_0 = \{\Phi(State(S_0)), KState(S_0, z) \equiv \Phi(z)\}$ *and let* Σ *be the foundational axioms plus a set of domain axioms representing accurate effect knowledge. Let* σ *be an action sequence such that* $\Sigma \cup \Sigma_0 \models POSS(\sigma)$. *Then for any model* $\mathcal{M}$ *of* $\Sigma_0 \cup \Sigma$ *and any valuation* ν,

$$\mathcal{M}, \nu \models KState(S_\sigma, z) \quad \textit{iff} \quad \mathcal{M}, \nu \models \mathcal{P}(\sigma, \varsigma, z) \ \textit{for some}\ \varsigma$$

Proof (sketch). The proof is by induction on σ. The base case $\sigma = [\,]$ follows by (13) and Σ_0. The induction step for $\sigma = [A(\vec{t}) \,|\, \sigma']$ with $A(\vec{t})$ being a non-sensing action follows by (14) and knowledge update axiom (11). The induction step for $\sigma = [Sense(f) \,|\, \sigma']$ follows by (15) and knowledge update axiom (7).

This theorem serves as the formal justification for the FLUX encoding of knowledge and sensing. The generic sensing action $Sense(f)$ is encoded by a state update axiom which carries as additional argument the result of sensing, where the sensing value is either 0 or 1:

```
state_update(Z, sense(F), Z, SV) :-
   not_holds(F, Z), SV=0 ; holds(F, Z), SV=1.
```

The definition of progression is a direct encoding of (13)–(15):

```
p([], [], Z) :- init(Z).
p([A|S], H2, Z2) :- p(S, H1, Z1),
                    ( state_update(Z1, A, Z2), H2=H1 ;
                      state_update(Z1, A, Z2, SV), H2=[SV|H1] ).
```

The FLUX definitions for $Knows(f, s)$, $Knows(\neg f, s)$, and $Kwhether(f, s)$ then follow from Theorem 2.

Corollary 3. *Let* ϕ *be a* FLUENT *term. Under the assumptions of Theorem 2,*

1. $\Sigma_0 \cup \Sigma \models Knows(\phi, S_\sigma)$ *iff there is no model* $\mathcal{M}$ *of* $\Sigma_0 \cup \Sigma$, *no valuation* ν, *and no history* ς *such that* $\mathcal{M}, \nu \models \mathcal{P}(\sigma, \varsigma, z) \wedge \neg Holds(\phi, z)$.
2. $\Sigma_0 \cup \Sigma \models Knows(\neg\phi, S_\sigma)$ *iff there is no model* $\mathcal{M}$ *of* $\Sigma_0 \cup \Sigma$, *no valuation* ν, *and no history* ς *such that* $\mathcal{M}, \nu \models \mathcal{P}(\sigma, \varsigma, z) \wedge Holds(\phi, z)$.
3. $\Sigma_0 \cup \Sigma \models Kwhether(\phi, S_\sigma)$ *iff there is no model* $\mathcal{M}$ *of* $\Sigma_0 \cup \Sigma$, *no valuation* ν, *and no history* ς *such that* $\mathcal{M}, \nu \models \mathcal{P}(\sigma, \varsigma, z_1) \wedge Holds(\phi, z_1) \wedge \mathcal{P}(\sigma, \varsigma, z_2) \wedge \neg Holds(\phi, z_2)$.

Proof (sketch). Follows from Theorem 2 and macro (5).

[4] Below, $POSS(\sigma)$ means that σ is possible in S_0, that is, $POSS([\,]) \stackrel{\text{def}}{=} True$ and $POSS([A(\vec{t}) \,|\, \sigma]) \stackrel{\text{def}}{=} POSS(\sigma) \wedge Poss(A(\vec{t}), S_\sigma)$.

Hence:

```
knows(F, S) :- is_fluent(F), \+ ( p(S, _, Z), not_holds(F, Z) ).
knows(-(F), S) :- is_fluent(F), \+ ( p(S, _, Z), holds(F, Z) ).
kwhether(F, S) :- is_fluent(F), \+ ( p(S, H, Z1), holds(F, Z1),
                  p(S, H, Z2), not_holds(F, Z2) ).
```

where `is_fluent` shall be true if the argument constitutes a FLUENT term of the language. Recall, for instance, the example initial state of Section 2. Whether solution A is acidic is still unknown after testing it but will be known after further sensing the color of the Litmus strip—though it cannot be predicted that it is acidic (nor, of course, that it is not):

```
?- \+ kwhether(acidic(a), [test(a,p)]),
   kwhether(acidic(a), [sense(red(p)),test(a,p)]),
   \+ knows(acidic(a), [sense(red(p)),test(a,p)]).
yes
```

The range of Theorem 2 includes nondeterministic actions. The latter may in particular cause loss of knowledge [17]; e.g.,

```
state_update(Z1, dilute(X), Z2) :-
  Z2 = Z1 ; update(Z1, [], [acidic(X)], Z2).

?- kwhether(acidic(a), [dilute(a),sense(red(p)),test(a,p)]).
no
```

4 Conditional Plans and Plan Skeletons

The reified conditional plans of the Fluent Calculus are encoded in FLUX as possibly nested lists of actions, in the order of execution; e.g.,

```
[test(b,p), sense(red(p)), if(red(p),[take(c)],[take(b)])]
```

represents conditional plan (9) from above. A planning problem with incomplete states and sensing actions is the problem of finding a conditional plan which can be proved to be executable and to achieve the goal under any circumstances. Therefore, if a conditional action is inserted into a plan, then each branch must be searched individually. To this end, we introduce the auxiliary actions $Commit(f)$ and $Commit(\neg f)$, which, formally, do not affect the world state but the knowledge:

$$KState(Do(Commit(f), s), z) \equiv KState(s, z) \wedge Holds(f, z)$$
$$KState(Do(Commit(\neg f), s), z) \equiv KState(s, z) \wedge \neg Holds(f, z)$$

In terms of FLUX:

```
state_update(Z, commit(F), Z) :- \+ F = -(_), holds(F, Z).
state_update(Z, commit(-(F)), Z) :- not_holds(F, Z).
```

In principle, the FLUX clauses we arrived at can readily be used by a simple forward-chaining search algorithm. Enumerating the set of plans, including all possible sensing actions, a solution will eventually be found if only the problem is solvable. However, planning with incomplete states usually involves a considerable search space, and the possibility to generate conditional plans only enlarges it. The concept of nondeterministic robot programs has been introduced in GOLOG as a powerful heuristics for planning, where only those plans are searched which match a given skeleton [15]. This avoids considering obviously useless actions such as ineffectual sensing. In the following, we generalize this concept to incomplete states and state knowledge. Our major extension concerns conditionals, which we resolve at planning time only if the state knowledge suffices to do so; otherwise, a branching point is introduced, leading to a conditional plan by need.

Similar to GOLOG we use a macro $do(\delta, \sigma, p)$ where δ is a robot program (represented as sequence of commands), σ a sequence of actions (possibly including the auxiliary *Commit*), and p is a (possibly conditional) plan. The intended reading is that executing δ in situation S_σ may result in the executable plan p. The crucial extension to GOLOG is this new definition of a conditional:

$$\begin{aligned} do(\texttt{if}\ f\ \texttt{then}\ \delta_1\ \texttt{else}\ \delta_2, \sigma, p) \stackrel{\text{def}}{=} (\exists p_1, p_2)\,(\,& Kwhether(f, S_\sigma) \wedge \\ & do(\delta_1, [Commit(f) \mid \sigma], p_1) \wedge \\ & do(\delta_2, [Commit(\neg f) \mid \sigma], p_2) \wedge \\ & p = [If(f, p_1, p_2)] \end{aligned} \tag{16}$$

The other standard macros of GOLOG are straightforwardly adapted to the Fluent Calculus. We just mention those which will be used in our example below, namely, primitive actions, testing, and nondeterministic choice of sub-programs (denoted by $\delta_1 \# \delta_2$) and of arguments (denoted by $(\pi x)\delta$).

$$do([\,], \sigma, p) \stackrel{\text{def}}{=} p = [\,]$$

$$\begin{aligned} do([a \mid \delta], \sigma, p) \stackrel{\text{def}}{=} (\exists p')\,(\,& Poss(a, S_\sigma) \wedge \\ & do(\delta, [a \mid \sigma], p')\ \wedge\ p = [a \mid p']\,) \end{aligned}$$

$$do([\{\neg\}Knows(f)? \mid \delta], \sigma, p) \stackrel{\text{def}}{=} Knows(\{\neg\}f, \sigma) \wedge do(\delta, \sigma, p)$$

$$do([\{\neg\}Kwhether(f)? \mid \delta], \sigma, p) \stackrel{\text{def}}{=} \{\neg\}Kwhether(f, \sigma) \wedge do(\delta, \sigma, p)$$

$$do([\delta_1 \# \delta_2 \mid \delta], \sigma, p) \stackrel{\text{def}}{=} do(\delta_1 + \delta, \sigma, p)\ \vee\ do(\delta_1 + \delta, \sigma, p)$$

$$do((\pi x)\delta, \sigma, p) \stackrel{\text{def}}{=} (\exists x)\, do(\delta, \sigma, p)$$

where $\delta + \delta'$ denotes concatenation of two programs. The encoding in FLUX is straightforward; we just mention the clause which encodes the conditional:

```
do([if(F,E1,E2)|L], S, P) :-
  is_fluent(F),
  append(E1, L, L1), append(E2, L, L2),
```

```
( knows(F, S), !, do(L1, S, P) ;
  knows(-(F), S), !, do(L2, S, P) ;
  kwhether(F, S), do(L1, [commit(F)|S], P1),
                  do(L2, [commit(-(F))|S], P2),
                  P = [if(F,P1,P2)] ).
```

That is to say, if the condition can be decided in advance, then the corresponding branch is chosen; otherwise, a conditional plan is generated and both branches are searched. Regarding the latter case, notice that it is checked (using `kwhether`) that it will be possible to evaluate the condition at execution time (c.f. (16)); if not, then the clause fails as the resulting plan would not be executable.

The empty robot program terminates successfully with the empty plan if the planning goal is satisfied:

```
do([], S, []) :- goal(S).
```

As an example, consider the following recursive robot program, which can be used to find among any selection of chemical solutions a non-acidic one with a sufficient supply of Litmus paper:

```
proc(find_non_acidic, [pi(x,[(knows(on_table(x)))?,
                             (not knows(acidic(x)))?,
                             []#[test_acidity(x)],
                             if(acidic(x), [find_non_acidic],
                                           [take(x)])])]).

proc(test_acidity(X), [(not kwhether(acidic(X)))?,
                       pi(y,[test(X,y),sense(red(y))])]).
```

Put in words, to find a non-acidic solution, pick one which is not known to be acidic. It may be necessary to test the solution. (The first item in the body of the auxiliary procedure `test_acidity` avoids redundant testing.) If the selected solution is acidic, try to find another one, else grab it.

Consider, now, the goal to get a non-acidic solution,

```
goal(S) :- knows(carries(X), S), knows(-(acidic(X)), S).
```

With suitable domain clauses defining `is_fluent` and the action preconditions, the program will generate the following plan given the example initial specification of Section 2:

```
?- do([find_non_acidic], [], P)

P = [test(b,p), sense(red(p)), if(red(p),[take(c)],[take(b)])]
```

The reader may notice that it suffices to test solution B; if it turns out to be acidic, then C must be non-acidic.[5] It is worth stressing that even with the

[5] A second solution to the planning problem is of course to test solution C and to branch upon the result accordingly.

given plan skeleton, it is necessary to find the right sensing action. In particular, the system has to backtrack over the attempt to test solution A (which renders unusable the only available Litmus paper)!

5 Related Work

A distinguishing feature of our system is its expressiveness in comparison to most existing systems for planning with knowledge and sensing. In [9] an implementation is described for which a semantics is given based on the general Situation Calculus solution to the Frame Problem for knowledge of [20]. However, the implementation is based on the notion of an incomplete state as a triple of true, false, and unknown propositional fluents. The same representation is used in the logic programming systems [2, 16], which are both given semantics by a three-valued variant [2] of the Action Description Language [6]. This restricted notion of incomplete states does not allow for handling any kind of *disjunctive* information. As a consequence, none of the aforementioned systems can solve planning problems that require to derive implicit knowledge (as in the Litmus scenario) or reasoning by cases. The latter is necessary whenever an action has conditional effects depending on whether some unknown fluent is true or false, but where both conditional effects suffice to achieve the goal [2]. Similar restrictions apply to the approach of [7], based on Description Logic.

The only existing systems with a general solution to the Frame Problem for knowledge is [19]. However, this Prolog implementation cannot be used for planning with sensing as it does not allow to *search* for suitable sensing actions. Rather, the user is supposed to provide GOLOG programs where all necessary sensing actions have been correctly planned. Likewise restricted to plan verification is the approach [10], which is based on a special epistemic propositional logic. In contrast, our system is designed for solving planning problems as it allows to backtrack over sensing actions that lead to a dead end.

The semantics of our logic program is given by previous work on integrating a solution to the Frame Problem for knowledge into the Fluent Calculus [23]. This axiomatization technique is related to the Situation Calculus-based formalization of [20]. The basic idea there is to represent state knowledge by a binary situation-situation relation $K(s, s')$, meaning that as far as the robot knows in situation s it could as well be in situation s'. Hence, every given fact about any such s' is considered possible by the robot. Having readily available the explicit notion of a state in the Fluent Calculus, our formalization avoids this indirect encoding of state knowledge, which is intuitively less appealing because it seems that a robot should always know exactly which *situation* it is in—after all, situations in the Situation Calculus are merely sequences of actions that have been or will be taken by the robot [13]. In view of the computational challenge raised by the Frame Problem for knowledge, a crucial advantage of our approach is also the simple inference scheme (8) provided by the concept of knowledge update axioms.

A conceptually different semantical approach has been proposed in [17] as an

extension of the Action Description Language which is more powerful than the abovementioned [2]. Incomplete knowledge is formalized by a so-called epistemic state, which is a set of possible sets of possible states. Intuitively, an epistemic state corresponds to the set of models for our *KState* predicate. We therefore suspect that our logic program can be shown to provide a sound and complete proof procedure for this semantics, too, but the formal details have yet to be worked out.

6 Discussion

We have developed the formal foundations for an effective inference method for state knowledge in the presence of incomplete states, nondeterministic actions, and sensing. Conditional plans are computed by reasoning about knowledge based on progression and with the help of a generalized concept of nondeterministic robot programs as search heuristics. The resulting extension of the high-level programming language FLUX exhibits a clear distinction between nondeterministic actions and nondeterminism in the heuristics. The latter needs to be resolved at planning time, possibly by introducing a branching point into a plan. Nondeterminism in state update axioms, on the other hand, is respected when verifying knowledge preconditions or proving that the plan is correct under any outcome.

We have successfully applied FLUX to the high-level control of a simple Lego robot [12] as well as a Pioneer-2, both of which perform delivery tasks and need to generate conditional plans which include sensing whether doors are closed.

Future work will be to extend the progression operator to actions with ramifications and to concurrency, in order to provide the formal justification for inferring knowledge in more complex domains. Furthermore, off-line planning in FLUX should be interleaved with on-line execution of sensing actions, following the argument of [8] that pure off-line planning can often be inefficient in the presence of sensing. Finally, while sensor noise has been ignored in all our applications thus far, the concept of knowledge update axioms can be readily applied to model nondeterministic outcomes, and hence to account for noise [1]. We currently pursue the axiomatization and implementation of sensor noise within our approach along this line.

References

1. F. Bacchus, J. Halpern, and H. Levesque. Reasoning about noisy sensors and effectors in the situation calculus. *Artif. Intell.*, 111(1–2):171–208, 1999.
2. C. Baral and T. C. Son. Approximate reasoning about actions in presence of sensing and incomplete information. In J. Maluszynski, ed., *Proc. of ILPS*, p. 387–401, Port Jefferson, 1997.
3. W. Bibel. Let's plan it deductively! *Artif. Intell.*, 103(1–2):183–208, 1998.
4. R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2:189–208, 1971.
5. T. Frühwirth. Theory and practice of constraint handling rules. *J. of Logic Programming*, 37(1–3):95–138, 1998.

6. M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *J. of Logic Programming*, 17:301–321, 1993.
7. G. De Giacomo, L. Iocchi, D. Nardi, and R. Rosati. Planning with sensing for a mobile robot. In *Proc. of the European Conf. on Planning*, vol. 1348 of *LNAI*, p. 158–170. Springer, 1997.
8. G. De Giacomo and H. Levesque. An incremental interpreter for high-level programs with sensing. In H. Levesque and F. Pirri, ed.'s, *Logical Foundations for Cognitive Agents*, p. 86–102. Springer, 1999.
9. K. Golden and D. Weld. Representing sensing actions: The middle ground revisited. In L. C. Aiello, J. Doyle, and S. Shapiro, ed.'s, *Proc. of KR*, p. 174–185, Cambridge, 1996.
10. A. Herzig, J. Lang, D. Longin, and T. Polascek. A logic for planning under partial observability. In H. Kautz and B. Porter, ed.'s, *Proc. of AAAI*, p. 768–773, 2000.
11. G. Lakemeyer. On sensing and off-line interpreting GOLOG. In H. Levesque and F. Pirri, ed.'s, *Logical Foundations for Cognitive Agents*, p. 173–189. Springer, 1999.
12. H. Levesque and M. Pagnucco. Legolog: Inexpensive experiments in cognitive robotics. In *Cognitive Robotics Workshop at ECAI*, p. 104–109, Berlin, 2000.
13. H. Levesque, F. Pirri, and R. Reiter. Foundations for a calculus of situations. *Electronic Transactions on Artif. Intell.*, 3((1–2)):159–178, 1998. `http://www.ep.liu.se/ea/cis/1998/018/`.
14. H. Levesque. What is planning in the presence of sensing? In B. Clancey and D. Weld, ed.'s, *Proc. of AAAI*, p. 1139–1146, Portland, 1996.
15. H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *J. of Logic Programming*, 31(1–3):59–83, 1997.
16. J. Lobo. COPLAS: A conditional planner with sensing actions. In *Cognitive Robotics*, vol. FS–98–02 of *AAAI Fall Symposia*, p. 109–116. AAAI Press 1998.
17. J. Lobo, G. Mendez, and S. Taylor. Adding knowledge to the action description language $\mathcal{A}$. In B. Kuipers and B. Webber, ed.'s, *Proc. of AAAI*, p. 454–459, Providence, 1997.
18. R. Moore. A formal theory of knowledge and action. In J. R. Hobbs and R. C. Moore, ed.'s, *Formal Theories of the Commonsense World*, p. 319–358. Ablex, 1985.
19. R. Reiter. On knowledge-based programming with sensing in the situation calculus. In *Cognitive Robotics Workshop at ECAI*, p. 55–61, Berlin, 2000.
20. R. Scherl and H. Levesque. The frame problem and knowledge-producing actions. In *Proc. of AAAI*, p. 689–695, Washington, 1993.
21. M. Thielscher. From Situation Calculus to Fluent Calculus: State update axioms as a solution to the inferential frame problem. *Artif. Intell.*, 111(1–2):277–299, 1999.
22. M. Thielscher. The fluent calculus: A specification language for robots with sensors in nondeterministic, concurrent, and ramifying environments. Technical Report CL-2000-01, Dresden University of Technology, 2000. `http://www.cl.inf.tu-dresden.de/~mit/publications/reports/CL-2000-01.pdf`
23. M. Thielscher. Representing the knowledge of a robot. In A. Cohn, F. Giunchiglia, and B. Selman, ed.'s, *Proc. of KR*, p. 109–120, Breckenridge, 2000.