Programming of Reasoning and Planning Agents with FLUX

Michael Thielscher*
Department of Computer Science
Dresden University of Technology
01062 Dresden (Germany)
mit@inf.tu-dresden.de

Abstract

We present a high-level programming method FLUX which allows to design cognitive agents that reason about their actions and plan. Based on the established, general action representation formalism of the Fluent Calculus, FLUX agents maintain an explicit, partial world model by which they control their own behavior. Thanks to the extensive reasoning facilities provided by the underlying calculus, FLUX allows to implement complex strategies with concise and modular agent programs. Systematic experiments with problems that require to reason about the performance of thousands of actions, have shown that FLUX exhibits excellent computational behavior and scales up particularly well to long-term control.

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. Autonomy in solving complex tasks requires the high-level cognitive capabilities of reasoning and planning: Exploring their environment, agents reason when they interpret sensor information, memorize it, and draw inferences from combined sensor data. Acting under incomplete information, agents employ their reasoning facilities to ensure that they are acting cautiously, and they plan ahead some of their actions with a specific goal in mind.

Agents whose intelligence goes beyond simple reactions to stimuli, reason and plan on the basis of a mental model of the state of their environment. As they move along, these agents constantly update this model to reflect the changes they have effected and the sensor information they have acquired. Having agents maintain an internal world model is necessary if we want them to choose their actions not only on the basis of the current status of their sensors but also by taking into account what they have previously observed or done. Moreover, the ability to reason about sensor information is necessary if properties of the environment can only indirectly be observed and require the agent to combine observations made at different stages. The cognitive capability of planning, finally, allows an agent to first calculate the effect of different action sequences in order to help it choosing one that is appropriate under the current circumstances.

Standard programming languages for agents, such as Java, require programmers to write special-purpose modules if they intend to endow their agents with the cognitive capabilities of reasoning and planning for the domain at hand. Formal theories of reasoning about actions and change, on the other hand, have the expressive power to provide these capabilities. Examples of existing agent programming methods deriving from general action theories are GOLOG [Levesque *et al.*, 1997; Reiter, 2001], based on the Situation Calculus, or the robot control language developed in [Shanahan and Witkowski, 2000], based on the Event Calculus. Neither of these systems and underlying calculi, however, provides the crucial concept of an explicit state representation. During the execution of a program, state knowledge is only indirectly represented via the initial conditions and the actions which the agent has performed thus far. As a consequence, evaluating conditions in an agent program always necessitates to trace back the entire history of actions, and hence requires ever increasing computational effort as the agent pro-

*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.

gresses. Studies reported in an accompanying paper have shown that this concept fails to scale up to long-term agent control [Thielscher, 2002a].

An explicit state representation being a fundamental concept in the Fluent Calculus [Thielscher, 1999], this established and versatile action representation formalism [Thielscher, 2000a] offers an alternative theory as the formal underpinnings for a high-level agent programming method. Actions are specified in the Fluent Calculus by so-called state update axioms and knowledge update axioms, respectively, which can be readily used in agent programs for maintaining an internal world model in accordance with the performed actions and acquired sensor information. The Fluent Calculus is also equipped with the formal concept of action histories, so-called situations, which can be used by agents to solve planning tasks along their way.

In this paper, we present the high-level programming method FLUX (for: *Fluent Executor*) which allows the design of intelligent agents that reason and plan on the basis of the Fluent Calculus. Using the paradigm of constraint logic programming, 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. With its powerful constraint solver, the underlying FLUX kernel provides general reasoning facilities, so that the agent programmer can focus on 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. Moreover, systematic experiments have shown that FLUX exhibits excellent computational behavior and scales up particularly well to long-term control.

The rest of the paper is organized as follows. We begin with illustrating, in Section 2, the key features of our programming methodology by an example of a non-trivial agent program. In Section 3, the semantics of FLUX is given in terms of the Fluent Calculus. Section 4 contains a description of the FLUX kernel; a detailed account of the constraint solver and the proof of its correctness is given in an accompanying paper [Thielscher, 2002b]. In Section 5, we show how the Fluent Calculus allows to define and prove soundness of FLUX programs. In Section 6, we give an overview of studies showing the computational merits of FLUX. A brief outlook is given in Section 7.

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

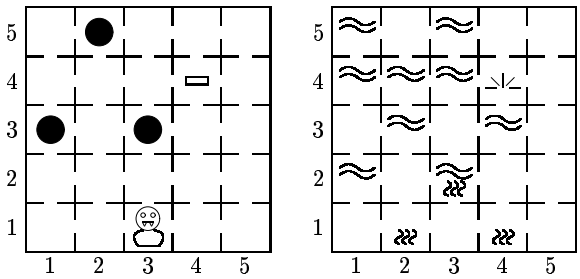

Figure 1: An example scenario in a Wumpus World where the 5×5-cave features three pits, the Wumpus in cell $(3,1)$, and gold in cell $(4,4)$. In the right hand side are depicted the corresponding perceptions (breeze, stench, glitter) for each location.

2 PROGRAMMING AGENTS IN FLUX

The example agent program in this paper is set in the artificial environment of the "Wumpus World," following the specification laid out in [Russell and Norvig, 1995] (see also Figure 1): An agent moves in a grid of cells, some of which contain, initially unknown to the agent, bottomless pits or gold, and there is one square which houses the hostile Wumpus. Its sensing capabilities allow the agent to perceive a breeze (a stench, respectively) if it is adjacent to a cell containing a pit (the Wumpus, respectively), and the agent notices a glitter in any cell containing gold and it hears a scream if the Wumpus gets killed. The elementary actions of the agent are to enter and exit the cave at cell $(1,1)$, turning clockwise by $90°$, going forward one square in the direction it faces, grabbing gold, and shooting an arrow in the direction it faces. The task of the agent is to find and bring home gold without falling into a pit and without encountering the Wumpus. A toy environment, the Wumpus World nonetheless requires crucial capabilities of intelligent agents, in particular acting cautiously under incomplete information, interpreting and logically combining sensor data, and planning. In the following, we develop a complete agent program for the Wumpus World, thereby illustrating the various features of our programming methodology.

2.1 FLUX States

FLUX agents use *states* as their internal model of the world. According to a convention in action theories, the atomic components of states are called *fluents*. Our program for the Wumpus World agent, for example, uses seven fluents: $At(x,y)$ and $Facing(d)$, representing that the agent is in cell (x,y) and faces

direction $d \in \{1,2,3,4\}$ (north, east, south, or west); $Gold(x,y)$, $Pit(x,y)$, and $Wumpus(x,y)$, representing that square (x,y) houses, respectively, gold, a pit, or the Wumpus; $Dead$, representing that the Wumpus is dead; and $Has(x)$, representing that the agent has $x \in \{Gold, Arrow\}$.

While a state is identified with all fluents that are true, agents hardly ever have complete information about their environment. To reflect this, incomplete states are encoded in FLUX as *open* lists, that is, lists with a variable tail, of fluents (possibly containing further variables). These lists are accompanied by *constraints* both for negated and disjunctive state knowledge as well as for variable range restrictions. The constraints are of the form $NotHolds(f,z)$, indicating that fluent f does not holds in state z; $NotHoldsAll(f,z)$, indicating that no instance of f holds in z; and $Or([f_1,\ldots,f_n],z)$, indicating that at least one of the fluents $f_1,\ldots,f_n$ holds in state z. Furthermore, FLUX employs a standard constraint solver for finite domains, which includes arithmetic constraints over rational numbers (using the equality and ordering predicates `#=`,`#<`,`#>` along with the standard functions `+`,`-`,`*`), range constraints (written `X::[`a`..`b`]`), and logical combinations using `#/\` and `#\/` for conjunction and disjunction, respectively. Consider, for example, the initial state of the Wumpus World agent, who has one arrow and knows that the Wumpus is not dead and could be in any square of the cave but $(1,1)$, that there are no pits in $(1,1)$ or outside the boundaries of the cave, and that initially the agent is nowhere inside of the cave and therefore not facing any direction:[1]

```
init(Z0) :- Z0 = [has(arrow),wumpus(WX,WY)|Z],
             [WX,WY] :: [1..5],
             not_holds(wumpus(1,1),Z0),
             not_holds_all(wumpus(_,_),Z),
             not_holds(dead,Z),
             not_holds(pit(1,1),Z),
             not_holds_all(pit(_,0),Z), %boundary
             not_holds_all(pit(_,6),Z),
             not_holds_all(pit(0,_),Z),
             not_holds_all(pit(6,_),Z),
             not_holds_all(at(_,_),Z),  %agent
             not_holds_all(facing(_),Z),
             duplicate_free(Z0).
```

The reader may notice the difference in specifying the location of the Wumpus and the pits: While there is a unique but unknown cell housing the former, there can be many pits or none at all. Stipulating that pits may not lie outside the boundaries of the cave will simplify the specification of what it means to sense a breeze.

[1] The auxiliary constraint $DuplicateFree(z)$ stipulates that list z does not contain multiple occurrences.

2.2 Update Specifications

As agents move along, they need to update their internal world model whenever they perform an action, in order to reflect the changes that have been effected and the sensor information that has been acquired. This maintenance of the state is based on a specification of the elementary actions of the agent. Following the solution to the fundamental frame problem in the Fluent Calculus [Thielscher, 1999], FLUX uses so-called state update axioms, one for each action, defining the positive and negative effects and the meaning of sensing results. To this end, the FLUX kernel provides the predicate $Update(z_1,\vartheta^+,\vartheta^-,z_2)$, encoding that state z_2 is the result of updating state z_1 by the positive and negative effects ϑ^+ and ϑ^-, respectively. Both ϑ^+ and ϑ^- are finite, possibly empty lists of fluents. The auxiliary predicate $Holds(f,z)$, indicating that fluent f holds in state z, is also provided by the kernel to be used in state update axioms. On this basis, update axioms are encoded by defining the predicate $StateUpdate(z_1,a,z_2,\pi)$, defining state z_2 as the result of performing action a in state z_1 and perceiving sensor information π.

The update axioms used in our program for the Wumpus World agent, for example, are as follows. Entering the cave has the effect that the agent is in cell $(1,1)$ facing north. Furthermore, its sensors tell it whether it perceives a breeze, a stench, or a glitter. (The auxiliary predicates are defined below.):

```
state_update(Z1,enter,Z2,[B,S,G]) :-
  update(Z1,[at(1,1),facing(1)],[],Z2),
  breeze_perception(1,1,B,Z2),
  stench_perception(1,1,S,Z2),
  glitter_perception(1,1,G,Z2).
```

Exiting the cave has the reverse effect:

```
state_update(Z1,exit,Z2,[]) :-
  holds(facing(D),Z1),
  update(Z1,[],[at(1,1),facing(D)],Z2).
```

Turning has the effect of facing the next direction (clockwise) on the compass:

```
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).
```

Going forward has the effect of being in the adjacent cell and acquiring new sensor inputs:

```
state_update(Z1,go,Z2,[B,S,G]) :-
  holds(at(X,Y),Z1), holds(facing(D),Z1),
  adjacent(X,Y,D,X1,Y1),
```

```
  update(Z1,[at(X1,Y1)],[at(X,Y)],Z2),
  breeze_perception(X1,Y1,B,Z2),
  stench_perception(X1,Y1,S,Z2),
  glitter_perception(X1,Y1,G,Z2).
```

Grabbing gold has the positive effect of having gold and the negative effect of clearing the cell:

```
state_update(Z1,grab,Z2,[]) :-
  holds(at(X,Y),Z1),
  update(Z1,[has(gold)],[gold(X,Y)],Z2).
```

Finally, shooting has a conditional effect, depending on whether the Wumpus is hit by the arrow, which is indicated by the perception of a scream. In either case the agent looses its arrow:

```
state_update(Z1,shoot,Z2,[S]) :-
  ( S=true, update(Z1,[dead],[has(arrow)],Z2)
    ; S=false, update(Z1,[],[has(arrow)],Z2) ).
```

Both the *Enter* action and the *Go* action use auxiliary predicates defining the meaning of perceiving a breeze, a stench, or a glitter at the new location:

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

The clause for sensing a stench is identical but with *Pit* being replaced by *Wumpus*, whereas perceiving glitter indicates the presence of gold in the very cell:

```
glitter_perception(X,Y,Percept,Z) :-
  Percept=false, not_holds(gold(X,Y),Z) ;
  Percept=true,  holds(gold(X,Y),Z).
```

The update axiom for *Go* uses a further auxiliary predicate defining the notion of adjacent cells wrt. the different directions:

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

2.3 Agent Programs

Agent programs written in FLUX use the fundamental command $Execute(a, z_1, z_2)$. Resolving this predicate triggers the actual performance of action a. Furthermore, the update of the current state z_1 to state z_2 is inferred on the basis of the state update axiom for a. The expressive power of high-level agent programming becomes apparent when using the internal world model to control the continuation of a program. Conditioning in FLUX is based on the foundational predicates $Knows(f, z)$, $KnowsNot(f, z)$, and $KnowsVal(\vec{x}, f, z)$, representing that the agent knows that fluent f holds (respectively, does not hold) in state z, and that there exist ground instances of the variables in $\vec{x}$ such that fluent f is known to be true in state z. In the following we implement a simple strategy for a Wumpus World agent, allowing it to systematically and cautiously explore the cave. The program maintains three parameters: a list of choicepoints; a list of cells already visited; and the current path the agent has taken, which is used for backtracking.

To begin with, the agent enters the cave and sets the choicepoints for cell $(1, 1)$, i.e., north and east; the lists of visited cells and the backtrack path are initialized accordingly:

```
main :- init(Z0), execute(enter,Z0,Z1),
        Cpts=[1,1,[1,2]], Vis=[[1,1]], Btr=[],
        main_loop(Cpts,Vis,Btr,Z1).
```

In the main loop, the agent systematically selects a direction to explore from its current location. If this step is successful, then the agent tries to hunt the Wumpus and checks if gold is known to be in the current square; if so, it grabs the gold and goes home, otherwise choicepoints are created for the new location and both the list of visited nodes and the backtrack path are extended. If, on the other hand, the selected choicepoint cannot safely be explored, then it is removed; and if there are no choicepoints left for the current location, then the agent backtracks:

```
main_loop([X,Y,Choices|Cpts],Vis,Btr,Z) :-
  Choices=[Dir|Dirs] ->
    % choicepoint exists
    (explore(X,Y,Dir,Vis,Z,Z1) ->
       % successful choicepoint
       knows_val([X1,Y1],at(X1,Y1),Z1),
       hunt_wumpus(X1,Y1,Z1,Z2),
       (knows(gold(X1,Y1),Z2) ->
          execute(grab,Z2,Z3), go_home(Z3)
        ; Cpts1=[X1,Y1,[1,2,3,4],X,Y,Dirs|Cpts],
          Vis1=[[X1,Y1]|Vis], Btr1=[X,Y|Btr],
          main_loop(Cpts1,Vis1,Btr1,Z2) )
     % unsuccessful choicepoint
     ; main_loop([X,Y,Dirs|Cpts],Vis,Btr,Z) )
  % no choicepoints left
  ; backtrack(Cpts,Vis,Btr,Z).
```

Let us first consider the procedure for exploring a certain direction, which the agent does only if it has not

yet visited the adjacent square and if the new location is safe. Acting cautiously, the agent shall enter a cell only if the latter is known to be free of a pit and if either the Wumpus is known to be elsewhere or known to be dead:

```
explore(X,Y,D,V,Z1,Z2) :-
  adjacent(X,Y,D,X1,Y1), \+ member([X1,Y1],V),
  knows_not(pit(X1,Y1),Z1),
  (knows_not(wumpus(X1,Y1),Z1);knows(dead,Z1)),
  turn_to(D,Z1,Z), execute(go,Z,Z2).
turn_to(D,Z1,Z2) :-
  knows(facing(D),Z1) -> Z2=Z1
  ; execute(turn,Z1,Z), turn_to(D,Z,Z2).
```

Next, consider the procedure for backtracking, which simply means to go back to the first square in the list of backtracking points. If the list happens to be empty, then the agent has reached the home square and therefore exits the cave:

```
backtrack(_,_,[],Z) :- execute(exit,Z,_).
backtrack(Cpts,Vis,[X,Y|Btr],Z) :-
  go_back(X,Y,Z,Z1), main_loop(Cpts,Vis,Btr,Z1).
go_back(X,Y,Z1,Z2) :-
  holds(at(X1,Y1),Z1), adjacent(X1,Y1,D,X,Y),
  turn_to(D,Z1,Z), execute(go,Z,Z2).
```

A simple strategy for hunting the Wumpus, if it is still alive, is to check if the cell is known where it hides and if the agent happens to be in the same row or column. If so, then our agent turns into the direction of the Wumpus and shoots; otherwise, no action is performed and, hence, the state does not change:

```
hunt_wumpus(X,Y,Z1,Z2) :-
  \+ knows(dead,Z1),
  knows_val([WX,WY],wumpus(WX,WY),Z1),
  in_direction(X,Y,D,WX,WY)
  -> turn_to(D,Z1,Z), execute(shoot,Z,Z2)
   ; Z2=Z1.
in_direction(X,Y,D,X1,Y1) :-
  [X,Y,X1,Y1]::1..5, D::1..4,
      (D#=1) #/\ (X1#=X) #/\ (Y1#>Y)  % north
  #\/ (D#=3) #/\ (X1#=X) #/\ (Y1#<Y)  % south
  #\/ (D#=2) #/\ (X1#>X) #/\ (Y1#=Y)  % east
  #\/ (D#=4) #/\ (X1#<X) #/\ (Y1#=Y). % west
```

With just the procedure *GoHome* remaining to be defined, the FLUX program illustrates how high-level strategies for intelligent agents can be encoded in a concise and modular fashion. In particular, the reader may appreciate that there is no need to program any inference capabilities—these are fully provided by the underlying FLUX kernel. This allows the agent programmer to focus on specifying complex behaviors on the basis of the elementary actions of an agent.

Applied to the scenario depicted in Figure 1, the program runs as follows: After entering the cave the agent first goes north to $(1,2)$. Since it senses a breeze and cannot decide whether there is a pit in $(1,3)$ or $(2,2)$, or both for that matter, the agent backtracks and then goes to $(2,1)$. Sensing no breeze there, the agent concludes that $(2,2)$ does not house a pit and that the breeze in $(1,2)$ must have come from $(1,3)$. Moreover, sensing a stench in $(2,1)$ without having experienced one in $(1,2)$, the agent infers that the Wumpus is in $(3,1)$. The agent shoots its arrow in that direction, making it possible to step over the dead Wumpus later on, and continues to explore the cave. Eventually, the agent arrives at $(4,4)$, where it senses a glitter and grabs the gold. At this stage, the current backtracking path is as depicted in Figure 2. Furthermore, the agent has acquired knowledge about the contents of all but four cells.

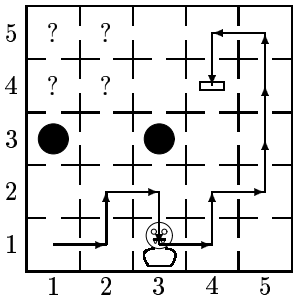

Figure 2: Exploring the cave depicted in Figure 1, our agent eventually reaches the cell with the gold, having shot the Wumpus along its way and inferred the locations of two pits. The northwest corner of the cave is still unknown territory.

2.4 Planning

The ability to devise plans adds a second dimension to high-level agent programming. A useful application of planning in our example domain is to have the agent find a direct route home after claiming gold: While the agent could just follow its backtracking path, this usually involves a considerable detour, as the situation in Figure 2 illustrates.

Since unrestricted planning with incomplete states is a notoriously hard problem, FLUX allows to define compound actions and to specify domain-dependent search trees for planning problems. For instance, to plan a short but safe route home, our Wumpus World agent uses the compound action of going to any adjacent cell, thereby postponing to execution time the task of finding the right number of *Turn* actions. The virtual action of going to square (x,y) has the effect of changing the agent's location:

```
state_update(Z1,go_to(X,Y),Z2,[]) :-
  holds(at(X1,Y1),Z1),
  update(Z1,[at(X,Y)],[at(X1,Y1)],Z2).
```

Search trees for planning problems are specified in FLUX by defining the predicate $PlanProc(g,p)$, where g denotes a planning task and p describes the search space. Adopting key notions and notations from GOLOG [Levesque *et al.*, 1997], the concept of a search space in FLUX is defined as follows. An elementary action, a compound action, and a test $?(\varphi)$ are search spaces, and if $e_1, e_2, \ldots, e_n$ are search spaces, then so are $e_1 \# e_2$ (nondeterministic choice) and $[e_1, \ldots, e_n]$ (sequencing; $n \geq 0$). The search space for our example planning problem can thus be specified as follows. (The auxiliary tests are defined below.):

```
plan_proc(find_path(Vis),P) :-
  P = (?(home) #
       [?(poss_go(X,Y,Vis,Vis1)), go_to(X,Y),
        find_path(Vis1)]).
```

Put in words, a successful plan is either to be home or to go to an adjacent cell, if possible, followed by a plan to go home from there. The planning procedure employs a list of visited nodes to avoid running into a loop.

The tests $?(\varphi)$ occurring in the specification of a search space need to be accompanied by definitions for φ to hold in the current situation, that is, after performing the current sequence of actions s in the initial state z_0 of the planning problem. To this end, a clause with head $P(\vec{x}, z, s_0)$ needs to be defined for every test $?(P(\vec{x}))$. The FLUX kernel provides definitions of the standard predicates $Knows(f,s,z_0)$, $KnowsNot(f,s,z_0)$, and $KnowsVal(\vec{x},f,s,z_0)$, which carry the additional situation argument, too. On this basis, the tests needed for our example planning problem can be defined as follows:

```
home(S,Z) :-
  knows(at(1,1),S,Z).
poss_go(X1,Y1,Vis,Vis1,S,Z) :-
  knows_val([X,Y],at(X,Y),S,Z),
  ( D=1 ; D=2 ; D=3 ; D=4 ),
  adjacent(X,Y,D,X1,Y1),
  \+ member([X1,Y1],Vis),
  knows_not(pit(X1,Y1),Z),
  ( \+ knows(dead,Z)->knows_not(wumpus(X1,Y1),Z)
    ; true ),
  Vis1=[[X,Y]|Vis].
```

The reader may notice how the agent plans it safe, that is, only those locations are searched which are known to be free of a pit and of the alive wumpus. For the sake of efficiency, these tests refer to the initial state of the planning problem as they are not affected by the planned actions.

What remains to be done is to define the cost c of a plan p for a planning problem g, using the standard FLUX predicate $PlanCost(g,p,c)$. Furthermore, the execution of a virtual action a leading from state z_1 to state z_2 is to be defined using the standard FLUX predicate $ExecuteCompoundAction(a,z_1,z_2)$. The cost of a plan to go home is simply its length, while its execution requires to find the right number of turns prior to going to the adjacent cell:

```
plan_cost(find_path(_),P,C) :-
  length(P,C).
execute_compound_action(go_to(X,Y),Z1,Z2) :-
  holds(at(X1,Y1),Z1), adjacent(X1,Y1,D,X,Y),
  turn_to(D,Z1,Z), execute(go,Z,Z2).
```

Employing the FLUX predicates $Plan(g,p,z_1)$ and $Execute(p,z_1,z_2)$, agents can be programmed so as to find a plan p for problem g in state z_1 and to actually execute this plan, thereby updating state z_1 to z_2. In our example,

```
go_home(Z) :-
  plan(find_path([]),Plan,Z),
  execute(Plan,Z,Z1), execute(exit,Z1,_).
```

Applied to the state depicted in Figure 2, the agent finds and executes the plan to go straight down to $(4,1)$, to turn, and to walk straight to $(1,1)$.

3 A FLUENT CALCULUS SEMANTICS FOR FLUX

3.1 Fluents and States

The Fluent Calculus is a many-sorted predicate logic language with four standard sorts: FLUENT, STATE, ACTION, and SIT (for situations). States are composed of fluents (as atomic states) using the standard function $\circ$: STATE $\times$ STATE $\mapsto$ STATE and constant $\emptyset$: STATE (denoting the empty state). In order to capture the intuition of identifying a state with the fluents that hold, the special connection function of the Fluent Calculus should obey certain properties which resemble the union operation for sets:[2]

Definition 1 The foundational axioms Σ_{state} of the Fluent Calculus are,

1. Associativity, commutativity, idempotence, and

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

unit element:

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

2. Empty state axiom:

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

3. Irreducibility and decomposition:

$$\begin{aligned}Holds(f_1, f_2) &\supset f_1 = f_2\\ Holds(f, z_1 \circ z_2) &\supset Holds(f, z_1) \vee Holds(f, z_2)\end{aligned}$$

4. State equality and state existence:

$$\begin{aligned}&[Holds(f, z_1) \equiv Holds(f, z_2)] \supset z_1 = z_2\\ &(\forall \Phi)(\exists z)(\forall f)\,(Holds(f, z) \equiv \Phi(f))\end{aligned}$$

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

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

□

The very last, second-order axiom above stipulates the existence of a state for all possible combinations of fluents.

On this basis, the semantics of a state specification in FLUX of the form

$$z = [f_1, \ldots, f_k \mid z'],\ DuplicateFree(z)$$

is given by the equational axiom $z = f_1 \circ \ldots \circ f_k \circ z'$, where z, z' are of sort STATE and the f_i's are of sort FLUENT. The meaning of the FLUX atom $Holds(f, z)$ is as in macro (1), while the semantics for the constraints $NotHolds(f, z)$, $NotHoldsAll(f, z)$, and $Or([f_1, \ldots, f_n], z)$ is given by the following axioms:

$$\neg Holds(f, z);\quad (\forall \vec{x})\, \neg Holds(f, z);\quad \bigvee_{i=1}^{n} Holds(f_i, z)$$

where $\vec{x}$ are the free variables in f. For example, the initial state in the Wumpus World is suitably described by the following Fluent Calculus axiom (c.f. Section 2.1):

$$\begin{aligned}&(\exists x, y, z)\\ &\quad (\ State(S_0) = Has(Arrow) \circ Wumpus(x, y) \circ z \wedge\\ &\quad\ 1 \le x \le 5 \wedge 1 \le y \le 5 \wedge\\ &\quad\ \neg Holds(\,Wumpus(1,1), z_0) \wedge\\ &\quad\ (\forall x', y')\, \neg Holds(\,Wumpus(x', y'), z) \wedge\\ &\quad\ \neg Holds(Dead, z) \wedge \neg Holds(Pit(1,1), z) \wedge\\ &\quad\ (\forall x')\, (\neg Holds(Pit(x', 0), z) \wedge\\ &\qquad\ \neg Holds(Pit(x', 6), z)\,) \wedge\\ &\quad\ (\forall y')\, (\neg Holds(Pit(0, y'), z) \wedge\\ &\qquad\ \neg Holds(Pit(6, y'), z)\,) \wedge\\ &\quad\ (\forall x', y')\, \neg Holds(At(x', y'), z) \wedge\\ &\quad\ (\forall d)\, \neg Holds(Facing(d), z)\,)\end{aligned} \qquad (2)$$

The axiomatization of states in the Fluent Calculus via the foundational axioms paves the way for an extensional definition of addition and removal of (finitely many) fluents from states, which in turn lays the foundation for an effective solution to the fundamental frame problem in the presence of incomplete states. The following definition introduces the macro equation $z_1 - \tau = z_2$ with the intended meaning that state z_2 is state z_1 minus the fluents in the finite state τ. The compound macro $z_2 = (z_1 - \vartheta^-) + \vartheta^+$ means that state z_2 is state z_1 minus the fluents in ϑ^- plus the fluents in ϑ^+:

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

where both ϑ^+, ϑ^- are finitely many FLUENT terms connected by "$\circ$". The crucial item is the second one, which defines removal of a single fluent f using a case distinction: Either $z_1 - f$ equals z_1 (which applies in case $\neg Holds(f, z_1)$), or $z_1 - f$ plus f equals z_1 (which applies in case $Holds(f, z_1)$). On this basis, the semantics of the FLUX predicate $Update(z_1, \vartheta^+, \vartheta^-, z_2)$ is given by the equation $z_2 = (z_1 - \vartheta^-) + \vartheta^+$.

3.2 Actions and Situations

Adopted from the Situation Calculus, the two standard sorts ACTION and SIT are used to axiomatize sequences of actions. The standard function Do : ACTION × SIT ↦ SIT denotes the situation reached by performing an action in a situation, and the constant S_0 : SIT denotes the initial situation. The standard function $State$: SIT ↦ STATE, which features uniquely in the Fluent Calculus, serves as denotation for the state of the environment in a situation.

Generalizing previous approaches [Bibel, 1986; Hölldobler and Schneeberger, 1990], the fundamental frame problem is solved in the Fluent Calculus by axioms which specify the difference between the states before and after an action [Thielscher, 1999]. Let $Poss$: ACTION × STATE denote that an action is possible in a state, then a *precondition axiom* for an action $A(\vec{x})$ is of the form

$$Poss(A(\vec{x}), z) \equiv \Pi(z)$$

where $\Pi(z)$ is a first-order formula with free variable z. The following is the general form of a *state update axiom* for a (possibly nondeterministic) action $A(\vec{x})$ with possibly conditional effects:[3]

$$\begin{array}{l} Poss(A(\vec{x}), s) \supset \\ \quad (\exists)\,(\,\Delta_1(s) \wedge \\ \qquad State(Do(A(\vec{x}), s)) = (State(s) - \vartheta_1^-) + \vartheta_1^+\,) \\ \quad \vee \ldots \vee \\ \quad (\exists)\,(\,\Delta_n(s) \wedge \\ \qquad State(Do(A(\vec{x}), s)) = (State(s) - \vartheta_n^-) + \vartheta_n^+\,) \end{array}$$

First-order formulas $\Delta_i(s)$ specify the conditions on $State(s)$ under which $A(\vec{x})$ has the positive and negative effects ϑ_i^+ and ϑ_i^-, respectively. Both ϑ_i^+ and ϑ_i^- are STATE terms consisting of FLUENTS only $(1 \leq i \leq n;\ n \geq 1)$.

Regarding our Wumpus World agent, consider the following precondition axioms:

$$\begin{array}{rcl} Poss(Enter, z) & \equiv & (\forall x, y)\, \neg Holds(At(x, y), z) \\ Poss(Exit, z) & \equiv & Holds(At(1, 1), z) \\ Poss(Turn, z) & \equiv & (\exists x, y)\, Holds(At(x, y), z) \\ Poss(Go, z) & \equiv & (\exists d, x, y, x', y') \\ & & (Holds(At(x, y), z) \wedge Holds(Facing(d), z) \wedge \\ & & Adjacent(x, y, d, x', y') \wedge \neg Holds(Pit(x, y), z) \wedge \\ & & [\neg Holds(Wumpus(x, y), z) \vee Holds(Dead, z)]\,) \\ Poss(Grab, z) & \equiv & (\exists x, y)\,(Holds(At(x, y), z) \wedge \\ & & \qquad Holds(Gold(x, y), z)) \\ Poss(Shoot, z) & \equiv & Holds(Has(Arrow), z) \end{array}$$

where $Adjacent(x, y, d, x', y')$ shall be defined as in Section 2.2.

The state update axioms are straightforward, following the description in Section 2.2. For example, the effect of Go on the state can be axiomatized as follows:

$$\begin{array}{l} Poss(Go, s) \supset \\ \quad (\exists d, x, y, x', y')\,(\,Holds(At(x, y), State(s)) \wedge \\ Holds(Facing(d), State(s)) \wedge Adjacent(x, y, d, x', y') \wedge \\ State(Do(Go, s)) = (State(s) - At(x, y)) + At(x', y')) \end{array}$$

The reader may notice that sensor information is not incorporated in state update axioms as sensing does not affect the state itself.

[3]Below, $Poss(a, s) \stackrel{\text{def}}{=} Poss(a, State(s))$.

3.3 Knowledge and Sensing

The basic Fluent Calculus has been extended in [Thielscher, 2000b] by the foundational predicate $KState$: SIT × STATE to allow for both representing state knowledge and reasoning about actions which involve sensing. An instance $KState(s, z)$ means that, according to the knowledge of the agent, z is a possible state in situation s. For example, the initial knowledge of our Wumpus World agent can be specified by this axiom:

$$(\forall z_0)\,(\,KState(S_0, z_0) \equiv \Psi(z_0)\,) \tag{3}$$

where $\Psi(z_0)$ is formula (2) with $State(S_0)$ replaced by variable z_0. That is to say, all states which satisfy the initial specification are to be considered possible by the agent. In particular, the agent has no further prior knowledge of the cave.

Based on the notion of a knowledge state, a fluent is known to hold in a situation (not to hold, respectively) just in case it is true (false, respectively) in all possible states:

$$\begin{array}{rcl} Knows(f, s) & \stackrel{\text{def}}{=} & (\forall z)\,(KState(s, z) \supset Holds(f, z)) \\ Knows(\neg f, s) & \stackrel{\text{def}}{=} & (\forall z)\,(KState(s, z) \supset \neg Holds(f, z)) \end{array}$$

Moreover, a value of a fluent is known just in case a particular instance holds in all possible states:

$$\begin{array}{l} KnowsVal(\vec{x}, f, s) \stackrel{\text{def}}{=} \\ \quad (\exists \vec{x})(\forall z)(KState(s, z) \supset Holds(f, z)) \end{array}$$

For example, the axiomatization of the initial knowledge entails that the agent knows that the Wumpus must be somewhere,

$$\begin{array}{l} \Sigma_{state} \cup \{(3)\} \models \\ \quad KState(S_0, z_0) \supset (\exists x, y)\, Holds(Wumpus(x, y), z_0) \end{array}$$

but the agent does not know where,

$$\begin{array}{l} \Sigma_{state} \cup \{(3)\} \models \\ \quad \neg KnowsVal(x, y, Wumpus(x, y), S_0) \end{array}$$

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 of the agent is specified by a so-called knowledge update axiom,[4]

$$\begin{array}{l} Knows(Poss(A(\vec{x})), s) \supset \\ \quad (KState(Do(A(\vec{x}), s), z) \equiv \\ \qquad (\exists z_1)(KState(s, z_1) \wedge \Psi(z, z_1) \wedge \Pi(z, s))\,) \end{array} \tag{4}$$

[4]Below, macro $Knows(Poss(a), s)$ stands for the formula $(\forall z)\,(KState(s, z) \supset Poss(a, z))$.

where Ψ specifies the physical state update while Π restricts the possible states so as to agree with the actual state $State(s)$ on the sensed properties. For example, this is the knowledge update axiom for action Go of our Wumpus World agent:

$$\begin{array}{l} Knows(Poss(Go), s) \supset (KState(Do(Go, s), z) \equiv \\ \quad (\exists d, x, y, x', y', z_1)\,(KState(s, z_1) \wedge \\ \qquad Holds(At(x, y), z_1) \wedge Holds(Facing(d), z_1) \wedge \\ \qquad Adjacent(x, y, d, x', y') \wedge \\ \qquad z = (z_1 - At(x, y)) + At(x', y') \wedge \\ \qquad [Breeze(x', y', z) \equiv Breeze(x', y', State(s))] \wedge \\ \qquad [Stench(x', y', z) \equiv Stench(x', y', State(s))] \wedge \\ \qquad [Glitter(x', y', z) \equiv Glitter(x', y', State(s))]\,)\,) \end{array}$$

where

$$\begin{array}{l} Breeze(x, y, z) \equiv \\ \quad Holds(Pit(x + 1, y), z) \vee Holds(Pit(x, y + 1), z) \vee \\ \quad Holds(Pit(x - 1, y), z) \vee Holds(Pit(x, y - 1), z) \end{array}$$

Likewise for $Stench(x, y, z)$, while

$$Glitter(x, y, z) \equiv Holds(Gold(x, y), z)$$

The other knowledge update axioms are straightforward and correspond to the encoding in Section 2.2.

4 THE FLUX KERNEL

Figure 3 gives an overview of the architecture of FLUX programs: The basic statements in agent programs are testing knowledge (predicate *Knows* etc.) and performing actions (predicate *Execute*), both of which are defined in the FLUX kernel. Maintaining the state when performing an action relies on the specification of update axioms, which in turn use the basic FLUX predicate *Update*. FLUX itself appeals to the paradigm of constraint logic programming, which enhances logic programs by mechanisms for solving constraints. In particular, so-called Constraint Handling Rules [Frühwirth, 1998] (CHRs) support declarative specifications of rules for processing the FLUX constraints which express negative and disjunctive state knowledge. In turn, these rules use finite domain constraints for handling variable arguments of fluents, which can be natural or rational numbers or of any user-defined finite domain.

4.1 States and Update

The basic definitions for states and update in the FLUX kernel are as follows:

```
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) :-
  (knows_not(F,Z)->Z1=Z; holds(F,Z,Z1)),
  minus(Z1,Fs,Zp).
plus(Z,[],Z).
plus(Z,[F|Fs],Zp) :-
  (knows_not(F,Z)->Z1=[F|Z]; holds(F,Z), Z1=Z),
  plus(Z1,Fs,Zp).
update(Z1,P,N,Z2) :- minus(Z1,N,Z),plus(Z,P,Z2).
```

The accompanying paper [Thielscher, 2002b] contains a proof of correctness of these clauses wrt. the foundational axioms Σ_{state} of the Fluent Calculus and the axiomatic characterization of fluent removal and addition.

Knowledge in FLUX is identified with logical entailment wrt. incomplete state specifications, employing the principle of negation-as-failure:

```
knows(F,Z)       :- \+ not_holds(F,Z).
knows_not(F,Z)   :- \+ holds(F,Z).
knows_val(X,F,Z) :- holds(F,Z), \+ nonground(X).
```

Again we refer to [Thielscher, 2002b] for the formal correctness of this definition wrt. the theory of knowledge in the Fluent Calculus as outlined in Section 3.3.

Finally, action execution is defined as follows, whereby the predicate $Perform(a, \pi)$ is assumed to trigger the actual performance of elementary action a with sensing values π returned:

```
execute(E,Z1,Z2) :-
  E=[]    -> Z2=Z1 ;
  E=[A|P] -> execute(P,Z1,Z), execute(A,Z,Z2) ;
  elementary_action(E) ->
    perform(E,SV), state_update(Z1,E,Z2,SV) ;
  execute_compound_action(E,Z1,Z2).
```

The FLUX constraint solver consists in a small set of progression and evaluation CHRs dealing with the constraints for negative and disjunctive state knowledge. By these rules, constraints are constantly simplified and combined in the course of a program in order to draw new inferences and to detect inconsistencies. See [Thielscher, 2002b] for the complete set of CHRs. Thanks to their declarative nature, correctness of these rules is easily verified against the foundational axioms of the Fluent Calculus.

4.2 Planning

Planning in FLUX is based on searching for a situation, represented by a list of actions, which matches

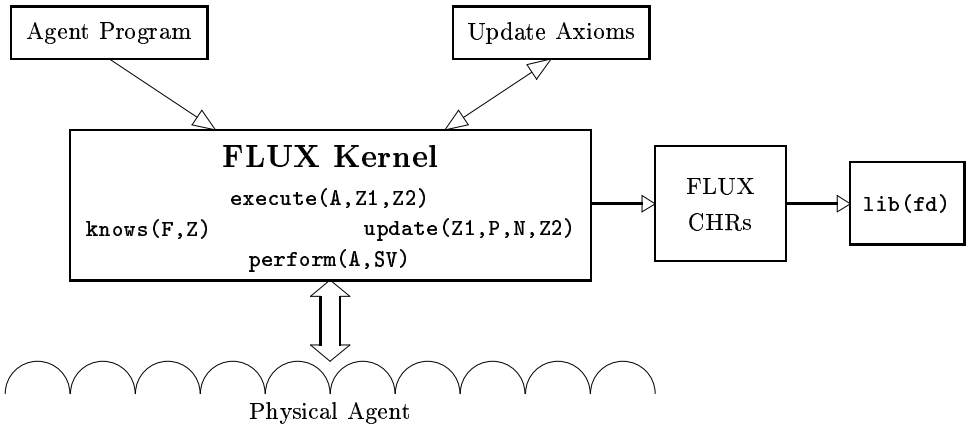

Figure 3: The architecture of a FLUX program. The actual performance of an action, triggered by *Perform*, connects the program to the effectors and sensors of the physical agent.

the specification of a search space (c.f. Section 2.4). To this end, the predicate $DO(g, s_0, s, z_0)$ defines situation s_0 plus action sequence s as a solution to planning problem g in state z_0. The definition follows closely the corresponding clauses in GOLOG [Levesque *et al.*, 1997]:

```
do(E,S0,S,Z0) :-
  E=[] -> S=S0 ;
  E=[E1|L] -> do(E1,S0,S1,Z0), do(L,S1,S,Z0) ;
  E=(E1#E2)-> (do(E1,S0,S,Z0); do(E2,S0,S,Z0)) ;
  plan_proc(E,E1) -> do(E1,S0,S,Z0) ;
  E=?(P) -> P =.. [Pred|Args],
            append(Args,[S0,Z0],ExtArgs),
            P1 =.. [Pred|ExtArgs], call(P1),
            S=S0 ;
  elementary_action(E) -> S=[E|S0] ;
  compound_action(E)   -> S=[E|S0].
```

Since at planning time the outcome of sensing actions cannot be predicted, all possibilities need to be taken into account. To this end, FLUX uses the predicate $Res(s, z_0, z)$, denoting that incomplete state z is a possible result of performing, in state z_0, the actions of situation s. In this way, a property of a future situation is known just in case there is no possible result in which the property could be false:

```
res([],Z,Z).
res([A|S],Z0,Z) :-
  res(S,Z0,Z1), state_update(Z1,A,Z,_).
knows(F,S,Z0) :-
  \+ ( res(S,Z0,Z), not_holds(F,Z) ).
knows_not(F,S,Z0) :-
  \+ ( res(S,Z0,Z), holds(F,Z) ).
```

Finally, given a specification of the cost of a plan, the search for an optimal plan is done by a standard search procedure that uses backtracking until no further solution can be found:

```
plan(Proc,Plan,Z0) :-
  assert(plan_search_best(void,0)),
  plan_search(Proc,Z0),
  plan_search_best(Plan,_),
  retract(plan_search_best(Plan,_)),
  Plan \= void.

plan_search(Proc,Z0) :-
  do(Proc,[],Plan,Z0),
  plan_cost(Proc,Plan,Cost),
  plan_search_best(BestPlan,BestCost),
  ( BestPlan \= void -> Cost < BestCost
                      ; true ),
  retract(plan_search_best(BestPlan,BestCost)),
  assert(plan_search_best(Plan,Cost)), fail
  ; true.
```

5 SOUNDNESS OF FLUX PROGRAMS

In this section, we give a brief overview of how properties of FLUX programs can be formally established with the help of the underlying Fluent Calculus semantics. We assume the reader to be familiar with basic notions and notations of logic programming with negation and constraints. In particular, we consider the standard left-to-right selection rule and so-called SLDNF-derivation trees, which consist of a main tree along with a number of subsidiary trees for negated literals [Apt and Bol, 1994].

The notion of a situation provides the semantics for the execution of a FLUX agent program.

Definition 2 Let T be the derivation tree for a FLUX program and query $P \cup \{Q\}$. An *execution node* in T is a node whose selected atom is $Init(\tau)$ or $Execute(\alpha, \tau_1, \tau_2)$. In the latter case,

term α is called the *executed action*. If N is a node in T, then the *situation associated* with N is $Do(\alpha_n, \ldots, Do(\alpha_1, S_0)\ldots)$ iff the path leading to N, but without N itself, satisfies the following:

1. the first execution node is of the form $Init(\tau)$;
2. no other execution node is of the form $Init(\tau)$; and
3. $\alpha_1, \ldots, \alpha_n$ is the ordered sequence of executed actions.

If the path does not contain an execution node, then the situation associated with N is S_0; in any other case the associated situation is undefined. □

The following notion of soundness describes a generally desirable property of FLUX agent programs. Informally speaking, all executed actions should be possible and there should be no backtracking over executed actions.

Definition 3 A FLUX program P with query Q is *sound* wrt. a Fluent Calculus axiomatization Σ iff each execution node $N = Execute(\alpha, \tau_1, \tau_2)$ in the computation tree for $P \cup \{Q\}$ satisfies the following:

1. α is a ground ACTION term;
2. the situation σ associated with N is defined and satisfies $\Sigma \models Poss(\alpha, \sigma)$;
3. N is in the main tree;
4. N lies on a branch that does not fail. □

Our agent program of Section 2 for the Wumpus World can be proved sound in this sense wrt. the Fluent Calculus axiomatization of Section 3.

Theorem 4 *The Wumpus World agent program with query* $\{Main\}$ *is sound.*

Proof (sketch):

1. The only occurrence of an execution node of the form *Init* is as the first atom in the body of the clause for *Main*.
2. All executed actions are constants of sort ACTION.
3. Action *Enter* is executed only as the first action, which is possible according to (2) and the precondition axioms.
4. Action *Exit* is executed only if the agent has either returned to square $(1,1)$ by unrolling the backtrack path or has planned and executed a route to this square. In both cases, the action is possible.
5. Action *Turn* is executed only after *Enter* and prior to *Exit*.
6. Action *Go* is executed only if there is an adjacent cell which is known to be free of a pit and if the Wumpus is known to be elsewhere or dead.
7. Action *Grab* is executed only if the agent is in a square that is known to contain gold.
8. Action *Shoot* is executed at most once, where the agent still has the arrow.
9. No negated literal depends on the atoms *Init* or *Execute* in the program; hence, all execution nodes are in the main tree.
10. The program always ends with success and there is no backtracking over execution nodes. ■

6 COMPUTATIONAL BEHAVIOR

Studies have shown that FLUX exhibits excellent computational behavior beyond problems of toy size. In the accompanying paper [Thielscher, 2002a], we report on experiments with a special variant of FLUX for complete states applied to a robot control program for a combinatorial mail delivery problem. The results show that FLUX can compute the effects of hundreds of actions per second. Most notably, the average time for selecting an action and inferring the effects remains essentially constant as the program progresses, which shows that FLUX scales up effortlessly to arbitrarily long sequences of actions. This result has been compared to GOLOG [Levesque *et al.*, 1997], where the curve for the computation cost suggests a polynomial increase over time. The analysis shows that the paradigm of a state-based representation is necessary for programs to scale up well to the control of agents and robots over extended periods of time: By maintaining an explicit state term throughout the execution of the program, fluents can be directly evaluated in FLUX programs, whereas an implicit state representation as in [Levesque *et al.*, 1997] or [Shanahan and Witkowski, 2000] leads to ever increasing computational effort as the program proceeds.

The computational behavior of FLUX in the presence of incomplete states has been analyzed in [Thielscher,

2002a] with a combinatorial problem that involves exploring a partially known environment and acting cautiously under incomplete information, much like in the Wumpus World. Although incomplete states pose a much harder problem, FLUX proves to scale up impressively well again: During a first phase, where the agent enhances its knowledge of the environment wandering around, there is a mere linear increase of the computation cost as the knowledge base grows. This result is particularly remarkable since the agent needs to constantly perform theorem proving tasks when conditioning its behavior on what it knows about the environment. Linear performance has been achieved due to a careful design of the state constraints supported in FLUX; the restricted expressiveness makes theorem proving computationally feasible. During the second phase of the aforementioned program, where the agent acts under the still incomplete knowledge, the average time for making decisions and inferring the effects of actions remains constant again. This shows that general FLUX, too, scales up effortlessly to long sequences of actions.

Planning with incomplete states, on the other hand, is a notoriously hard problem in FLUX, too. If the domain-dependent search space contains just a linear number of nondeterministic choices, there are exponentially many plans to be searched, and hence planning cannot scale up. Following an argument already put forward in [Giacomo and Levesque, 1999], the consequence for the agent programmer is that the planning facilities should be used restrictively and for bounded sub-problems only, in order not to spoil the computational merits of FLUX.

7 FUTURE WORK

Several crucial aspects of acting in real-world environments have not been tackled in this paper, such as the fact that actions may fail unexpectedly. An approach to this problem has been presented in [Thielscher, 2001b], where default assumptions regarding executability have been added to the theory of the Fluent Calculus. Incorporating this technique into agent programs is an important aspect of future work, as is modeling dynamic environments that constantly evolve around agents.

References

[Apt and Bol, 1994] K. Apt and R. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19/20:9–71, 1994.

[Bibel, 1986] W. Bibel. A deductive solution for plan generation. *New Gener. Comp.*, 4:115–132, 1986.

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

[Giacomo and Levesque, 1999] G. De Giacomo and H. Levesque. An incremental interpreter for high-level programs with sensing. In H. Levesque and F. Pirri, editors, *Logical Foundations for Cognitive Agents*, pages 86–102. Springer, 1999.

[Hölldobler and Schneeberger, 1990] S. Hölldobler and J. Schneeberger. A new deductive approach to planning. *New Gener. Comp.*, 8:225–244, 1990.

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

[Reiter, 2001] R. Reiter. *Logic in Action.* MIT Press, 2001.

[Russell and Norvig, 1995] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, 1995.

[Shanahan and Witkowski, 2000] M. Shanahan and M. Witkowski. High-level robot control through logic. In *Proceedings of ATAL*, volume 1986 of *LNCS*, pages 104–121, 2000. Springer.

[Thielscher, 1999] M. Thielscher. From Situation Calculus to Fluent Calculus: State update axioms as a solution to the inferential frame problem. *Artificial Intelligence*, 111(1–2):277–299, 1999.

[Thielscher, 2000a] M. Thielscher. Modeling actions with ramifications in nondeterministic, concurrent, and continuous domains—and a case study. In *Proceedings of AAAI*, pages 497–502, 2000. MIT Press.

[Thielscher, 2000b] M. Thielscher. Representing the knowledge of a robot. In *Proceedings of KR*, pages 109–120, 2000. Morgan Kaufmann.

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

[Thielscher, 2002a] M. Thielscher. Pushing the envelope: Programming reasoning agents. 2002. (Submitted).

[Thielscher, 2002b] M. Thielscher. Reasoning about actions with CHRs and finite domain constraints. 2002. (Submitted).