A FLUX Agent for the Wumpus World

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

Abstract

FLUX is a programming method for the design of agents that reason logically about their actions and sensor information in the presence of incomplete knowledge. We show how FLUX can be used to program an agent for the Wumpus World.[1] Experimental results show how the agent performs in terms of the reward function and how well the FLUX program scales up.

1 Introduction

The paradigm of Cognitive Robotics [1] is to endow agents with the high-level cognitive capability of reasoning. Exploring their environment, agents need to reason when they interpret sensor information, memorize it, and draw inferences from combined sensor data. Acting under incomplete information, agents employ their reasoning facilities for selecting the right actions. To this end, intelligent agents form a mental model of their environment, which they constantly update to reflect the changes they have effected and the sensor information they have acquired. The Wumpus World [2] is a good example of an environment where an agent needs to choose its actions not only on the basis of the current status of its sensors but also on the basis of what it has previously observed or done. Moreover, some properties of the environment can only be observed indirectly and require the agent to combine observations made at different stages.

FLUX [4; 5] is a high-level programming method for the design of intelligent agents that reason about their actions on the basis of the fluent calculus [3]. A constraint logic program, FLUX comprises a method for encoding incomplete states along with a technique of updating these states according to a declarative specification of the elementary actions and sensing capabilities of an agent. Incomplete states are represented by lists (of fluents) with variable tail, and negative and disjunctive state knowledge is encoded by constraints.

FLUX programs consist of three parts: A *kernel* provides the general reasoning facilities by encoding the foundational

[1]Parts of an earlier version of the FLUX program were previously presented at the *International Conference on Principles of Knowledge Representation and Reasoning* 2002.

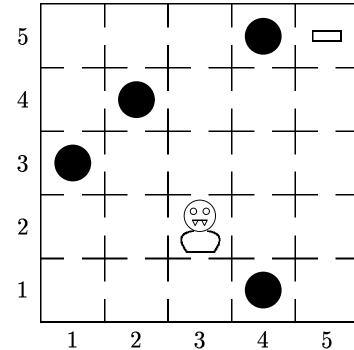

Figure 1: An example scenario in a Wumpus World where the 5×5-cave features four pits, the Wumpus in cell $(3, 2)$, and gold in cell $(5, 5)$.

axioms of the fluent calculus. The domain-specific *background theory* contains the formal specification of the underlying environment, including effect axioms for the actions of the agent. Finally, the *strategy* specifies the intended behavior of the agent. Space limitations do not permit us to fully recapitulate syntax and semantics of FLUX; we refer to [4; 5] for details. In the following section, we present a FLUX background theory for the Wumpus World, and in Section 3 we give a FLUX program that implements a particular strategy for an agent that systematically explores an unknown grid with the goal to find and bring home the gold. We conclude in Section 4 by reporting on some experiments and outlining possible ways of improving the basic strategy.

2 The Background Theory

A background theory describes the general properties of the environment and the actions of the agent. Following the specification laid out in [2], the Wumpus World agent moves in a rectangular grid of cells. An example scenario is depicted in Figure 1: There is a heap of gold somewhere in the grid, some of the cells contain bottomless pits, and one them houses the hostile Wumpus. The agent perceives 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 (through the arrow shot by the agent). Figure 2 shows

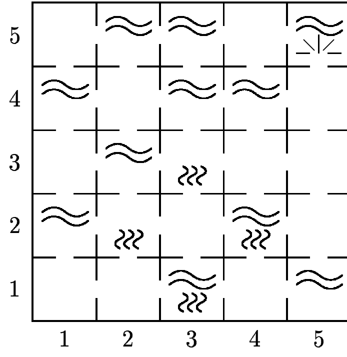

Figure 2: Perceptions corresponding to the scenario of Figure 1.

where the agent will sense a breeze, stench, and glitter, respectively, wrt. the scenario of Figure 1.

To axiomatize the Wumpus World, we use the following nine fluents. $At(x, y)$ and $Facing(d)$ represent, respectively, 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 $Wumpus(x, y)$ represent that square (x, y) houses, respectively, gold, a pit, or the Wumpus; *Dead* represents that the Wumpus is dead; $Has(x)$ represents that the agent has $x \in \{Gold, Arrow\}$; and $Ydim(n)$ and $Xdim(n)$, represent the (initially unknown) extent of the grid. The agent does not need a fluent to represent its own status of being alive or not, because we intend to write a cautious strategy by which the agent never takes the risk to fall into a pit or to enter the sqare with the Wumpus (unless the latter is known to be dead).

FLUX allows to combine physical and sensing effects in single action specifications, which we have exploited here: Update axioms are encoded in FLUX as definitions of the predicate $StateUpdate(z_1, A(\vec{x}), z_2, y)$ describing the update of state z_1 to z_2 according to the physical effects of action $A(\vec{x})$ and the sensing result y. We assume that when executing any of its physical action, the agent perceives a vector with five truth-values:

[*stench,breeze,glitter,bump,scream*]

The actions of the Wumpus World agent are then encoded as follows:

```
state_update(Z1,go,Z2,[S,B,G,Bump,_]) :-
   holds(at(X,Y),Z1), holds(facing(D),Z1),
   adjacent(X,Y,D,X1,Y1),
   ( Bump=false ->
        update(Z1,[at(X1,Y1)],[at(X,Y)],Z2),
        stench_perception(X1,Y1,S,Z2),
        breeze_perception(X1,Y1,B,Z2),
        glitter_perception(X1,Y1,G,Z2)
     ; Bump=true,
        Z2=Z1, bump_perception(X,Y,D,Z2) ).

state_update(Z1,turn_left,Z2,[S,B,G,_,_]):-
   holds(facing(D),Z1),
   (D#>1 #/\ D1#=D-1) #\/ (D#=1 #/\ D1#=4),
   update(Z1,[facing(D1)],[facing(D)],Z2),
   holds(at(X,Y),Z2),
   stench_perception(X,Y,S,Z2),
   breeze_perception(X,Y,B,Z2),
   glitter_perception(X,Y,G,Z2).

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

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

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

state_update(Z,exit,Z,_).
```

In the effect axiom for *Go*, for instance, the first four components of the sensory input are evaluated: If the agent does not perceive a *bump*, then the physical effect is to reach the adjacent location, and the *stench*, *breeze*, and *glitter* percepts are then evaluated against the updated (incomplete) state. The auxiliary predicates used in this and the other update axioms employ the two constraints *NotHolds* and *OrHolds*, for which the FLUX kernel contains a constraint solver:

```
stench_perception(X,Y,Percept,Z) :-
   XE#=X+1, XW#=X-1, YN#=Y+1, YS#=Y-1,
   ( Percept=false ->
        not_holds(wumpus(XE,Y),Z),
        not_holds(wumpus(XW,Y),Z),
        not_holds(wumpus(X,YN),Z),
        not_holds(wumpus(X,YS),Z)
     ; Percept=true,
        or_holds([wumpus(XE,Y),wumpus(X,YN),
                  wumpus(XW,Y),wumpus(X,YS)]
                 ,Z) ).

 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_holds([pit(XE,Y),pit(X,YN),
                  pit(XW,Y),pit(X,YS)],Z) ).

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

bump_perception(X,Y,D,Z) :-
   D#=1 -> holds(ydim(Y),Z)
         ; holds(xdim(X),Z).

adjacent(X,Y,D,X1,Y1) :-
   D :: 1..4, X1#>0, Y1#>0,
      (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
```

The update axioms for the two *Turn* actions use standard predicates for FD-constraints (*finite domains*), preceded by

the symbol "#".[2] The update clauses are direct encodings of the corresponding knowledge update axioms for the Wumpus World [4].

For the sake of simplicity, our FLUX program for the Wumpus World agent does not include precondition axioms, since going and turning is always possible while the preconditions for *Grab*, *Shoot*, and *Exit* are implicitly verified as part of the strategy (see Section 3). In addition to the specifications of the actions, a FLUX background theory consists of domain constraints and an initial state specification. Initially, the agent is at $(1,1)$, faces west (that is, 2) and possesses an arrow. Moreover, the agent knows that the Wumpus is still alive and that the home square is safe. The agent does not know the extension of the grid, nor the locations of the gold, the Wumpus, or any of the pits. The domain constraints are summarized in a clause defining consistency of states, which adds range and other constraints to the initial knowledge of the agent:

```
init(Z0) :-
   Z0 = [at(1,1),facing(2),has(arrow) | Z],
   not_holds(dead,Z),
   not_holds(wumpus(1,1),Z0),
   not_holds(pit(1,1),Z),
   consistent(Z).

consistent(Z) :-
   % uniqueness constraints
   holds(xdim(X),Z,Z1),
      not_holds_all(xdim(_),Z1),
   holds(ydim(Y),Z,Z2),
      not_holds_all(ydim(_),Z2),
   holds(at(AX,AY),Z,Z3),
      not_holds_all(at(_,_),Z3),
   holds(facing(D),Z,Z4),
      not_holds_all(facing(_),Z4),
   holds(gold(GX,GY),Z,Z5),
      not_holds_all(gold(_,_),Z5),
   holds(wumpus(WX,WY),Z,Z6),
      not_holds_all(wumpus(_,_),Z6),

   % range constraints
   [X,Y] :: [1..100], D :: [1..4],
   AX #>= 1, AX #<= X, AY #>= 1, AY #<= Y,
   GX #>= 1, GX #<= X, GY #>= 1, GY #<= Y,
   WX #>= 1, WX #<= X, WY #>= 1, WY #<= Y,

   % constraints for pits (boundary etc.)
   not_holds_all(pit(_,0),Z),
   not_holds_all(pit(0,_),Z),
   Y1 #= Y+1, not_holds_all(pit(_,Y1),Z),
   X1 #= X+1, not_holds_all(pit(X1,_),Z),
   not_holds(pit(GX,GY),Z),
   not_holds(pit(WX,WY),Z),

   duplicate_free(Z).
```

Here, the FLUX kernel predicate $Holds(f,z,z_1)$ means that fluent f holds in state z, and z_1 is z without f. The kernel constraint $DuplicateFree(z)$ is used to ensure that fluents do not occur twice in state list z.

[2] The update specification of *TurnLeft* includes the evaluation of some of the percepts, as this will be the very first action of the agent to acquire knowledge of the two cells surrounding its home square (1,1).

3 The Strategy

Strategy programs are based on the following pre-defined FLUX predicates:

- $Knows(f,z)$ (respectively, $KnowsNot(f,z)$), meaning that fluent f is known to hold (respectively, known not to hold) in incomplete state z.
- $KnowsVal(\vec{x},f,z)$, meaning that fluent f is known to hold for arguments $\vec{x}$ in incomplete state z.
- $Execute(a,z_1,z_2)$, meaning that the actual execution of action a in state z_1 leads to state z_2.

These predicates are used in the followin sample control program for a cautious Wumpus World agent:

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

main_loop([X,Y,Choices|Cpts],Vis,Btr,Z) :-
   Choices=[Dir|Dirs] ->
     (explore(X,Y,Dir,Vis,Z,Z1) ->
        (knows(at(X,Y),Z1) ->
            main_loop([X,Y,Dirs|Cpts],
                      Vis,Btr,Z1)
         ; knows_val([X1,Y1],at(X1,Y1),Z1),
           (knows(gold(X1,Y1),Z1) ->
              execute(grab,Z1,Z2),
              go_home(Z2)
            ; hunt_wumpus(X1,Y1,Z1,Z2,
                          Vis,Vis2,_,_),
              Cpts1=[X1,Y1,[1,2,3,4],
                     X,Y,Dirs|Cpts],
              Vis1=[[X1,Y1]|Vis2],
              Btr1=[Dir|Btr],
              main_loop(Cpts1,Vis1,Btr1,Z2)
           )
        )
      ; main_loop([X,Y,Dirs|Cpts],Vis,Btr,Z)
     )
   ; backtrack(Cpts,Vis,Btr,Z).

explore(X,Y,D,V,Z1,Z2) :-
   adjacent(X,Y,D,X1,Y1),
   \+ member([X1,Y1],V),
   (D#=1 -> \+ knows(ydim(Y),Z1)
    ;
    D#=2 -> \+ knows(xdim(X),Z1)
    ; true),
   knows_not(pit(X1,Y1),Z1),
   ( knows_not(wumpus(X1,Y1),Z1)
     ; knows(dead,Z1) ),
   turn_to(D,Z1,Z), execute(go,Z,Z2).

backtrack(_,_,[],_) :- execute(exit,_,_).

backtrack(Cpts,Vis,[D|Btr],Z) :-
   R is (D+1) mod 4 + 1,
   turn_to(R,Z,Z1), execute(go,Z1,Z2),
   knows_val([X,Y],at(X,Y),Z2),
   hunt_wumpus(X,Y,Z2,Z3,
               Vis,Vis1,Cpts,Cpts1),
   main_loop(Cpts1,Vis1,Btr,Z3).

turn_to(D,Z1,Z2) :-
   knows(facing(D),Z1) -> Z2=Z1
```

```
    ;
    knows_val([D1],facing(D1),Z1),
    ( (D-D1#=1 ; D1-D#=3) ->
        execute(turn_right,Z1,Z)
      ; execute(turn_left,Z1,Z) ),
    turn_to(D,Z,Z2).
```

After the initialization of the world model and the execution of a *TurnLeft* action at the home square (to acquire the first sensory input), the main loop is entered by which the agent systematically explores the environment. To this end, the program employs three parameters containing, respectively, choice points yet to be explored, the squares that have been visited, and the current path. The latter is used to backtrack from a location once all choices have been considered. A choice point is a list of directions, which are encoded by 1 (for north) to 4 (for west) as usual. The path is represented by the sequence, in reverse order, of the directions the agent took in each step.

In the main loop, the agent selects the first element of the current choices. If this direction could indeed be explored (predicate *Explore*) and the agent did not end in the same square (indicating that it actually bumped into a wall), then it checks whether it has found the gold. If so, it takes the quickest rout home (see below), else it sees if the Wumpus can be hunted down, and then a new choice point is created while augmenting the backtrack path by the direction into which the agent just went. If, on the other hand, the chosen direction cannot be taken, then the main loop is called with a reduced list of current choices. In case no more choices are left, the agent backtracks (predicate *Backtrack*).

The auxiliary predicate *Explore*(x, y, d, v, z_1, z_2) succeeds if the agent can safely go into direction d from its current location (x, y) in state z_1, ending up in state z_2. A direction is only explored if the adjacent square both does not occur in the list v of visited cells and is not known to lie outside the boundaries. Moreover, and most importantly, the adjacent location must be *known* to be safe. Thus the strategy implements a cautious agent, who never runs the risk to fall into a pit or to be eaten by the Wumpus. By the auxiliary predicate *Backtrack*, the agent takes back one step on its current path by reversing the direction. The program terminates once this path is empty, which implies that the agent has returned to its home after it has visited and cleaned as many locations as possible.

Here are the definitions of the missing parts of the strategy, that is, how to hunt the Wumpus and how to get home once gold has been found:

```
hunt_wumpus(X,Y,Z1,Z2,
            Vis1,Vis2,Cpts1,Cpts2) :-
    \+ 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),
       path(X,Y,WX,WY,Vis),
       subtract(Vis1,Vis,Vis2),
       ( Cpts1=[X,Y,Dirs1|Cpts] ->
            union(Dirs1,[D],Dirs2),
            Cpts2=[X,Y,Dirs2|Cpts]
         ; Cpts2=Cpts1 )
     ; Z2=Z1, Vis2=Vis1, Cpts2=Cpts1.
```

```
in_direction(X,Y,D,X1,Y1) :-
   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

go_home(Z) :-
   a_star_plan(Z,S),
   execute(S,Z,Z1), execute(exit,Z1,_).
```

The agent hunts the Wumpus only in case the location is known and, for the sake of simplicity, only when the agent happens to be in the same row or column (predicate *InDirection*). When the Wumpus has been killed, the agent can explore areas which it may have rejected earlier. Therefore, all cells that lie on the path $\vec{v}$ between the agent and the Wumpus (predicate *Path*$(x, y, w_x, w_y, \vec{v})$) may be re-vistited. To this end, it is ensured that the current list of choice points includes the direction in which the agent has shot the arrow.

After it has found the gold, the auxiliary predicate *GoHome* directs the agent to the exit square on a shortest safe path. Predicate *AStarPlan*(z, s) is defined in such a way that the agent employs A*-search to find a plan, i.e., sequence of actions s, that from the current location in state z to square $(1, 1)$. As the heuristic function, we use the Manhattan distance. We omit the details here; the complete FLUX program is available for download at `www.fluxagent.org`.

The following table illustrates what happens in the first nine calls to the main loop when running the program with the scenario depicted in Figure 1:

At	Cpts	Btr	Actions
$(1, 1)$	`[[1,2,3,4]]`	`[ ]`	*G*
$(1, 2)$	`[[1,2,3,4],[2,3,4]]`	`[1]`	–
$(1, 2)$	`[[2,3,4],[2,3,4]]`	`[1]`	–
$(1, 2)$	`[[3,4],[2,3,4]]`	`[1]`	–
$(1, 2)$	`[[4],[2,3,4]]`	`[1]`	–
$(1, 2)$	`[[ ],[2,3,4]]`	`[1]`	*LLG*
$(1, 1)$	`[[2,3,4]]`	`[ ]`	*LG*
$(2, 1)$	`[[1,2,3,4],[3,4]]`	`[ ]`	*LG*
$(2, 2)$	`[[1,2,3,4],[2,3,4],[3,4]]`	`[ ]`	–

The letters G, L are abbreviations for the actions *Go* and *TurnLeft*, respectively. After going north to $(1, 2)$, the agent cannot continue in direction 1 or 2 because both $(1, 3)$ and $(2, 2)$ may be occupied by a pit according to the agent's current knowledge. Direction 3 is not explored since location $(1, 1)$ has already been visited, and direction 4 is ruled out as $(0, 2)$ is outside of the boundaries. Hence, the agent backtracks to $(1, 1)$ and continues with the next choice there, direction 2, which brings it to location $(2, 1)$. From there it goes north, and so on. Eventually, the agent arrives at $(5, 5)$, where it senses a glitter and grabs the gold. The backtracking path at this stage is depicted in Figure 3. Along its way, the agent has determined the square with the Wumpus and shot its arrow in order to safely pass through this square.

4 Experimental Results

In order to see how the agent performs wrt. the reward function as specified in [2] and how the program scales to environments of increasing size, we ran series of experiments

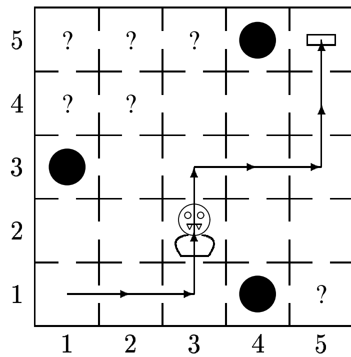

Figure 3: Exploring the cave depicted in Figure 1, our agent eventually reaches the cell with the gold, having shot the Wumpus along its way and inferred the locations of three pits. Parts of the grid are still unknown territory.

with square grids of different size. The scenarios were randomly chosen by adding a pit to each cell with probability 0.2 and then randomly placing the Wumpus and the gold in one of the free cells. The following table shows the cumulated reward[3] over 100 runs each for various grid sizes. The third row shows the number of successful runs, that is, where the agent found the gold:

Size	Total Reward	Successful Runs
5×5	43,028	45
6×6	32,596	35
7×7	20,276	23
8×8	15,471	18
9×9	12,616	15
10×10	10,012	13

As can be seen from the table, with increasing size it gets more and more difficult for our cautious agent to find the gold.

The following table shows the average time (seconds CPU time of a 1733 MHz processor) it takes for the agent to select an action and to infer the update. The times were obtained by averaging over 100 runs with different scenarios and by dividing the total time for solving a problem by the number of physical actions executed by the agent:

Size	Time per Action
5×5	0.0282 sec
6×6	0.0287 sec
7×7	0.0342 sec
8×8	0.0463 sec
9×9	0.0503 sec
10×10	0.0579 sec

The figures show that the program scales well. The slight increase is due to the increasing size of the state when the agent has acquired knowledge of large portions of the (increasingly large) environment. Indeed, the program scales gracefully to environments of much larger size; we also ran experiments with a 50×50-grid, placing the gold in the upper right corner and with a sparser distribution of pits (using a probability 0.05 for each square, so that in most cases the agent had to explore large portions of the grid). These scenarios were completed, on the average over 20 runs, within 340 seconds, and the average time for each action was 0.7695 sec.

[3]Following the specification in [2], the agent receives a reward of +1000 when it makes an exit with the gold. A "reward" of -1000 is given when the agent dies, but this never happens to our cautious agent. Every action counts -1, and using the arrow counts an additional -10.

5 Improvements

The FLUX program presented in this paper encodes a particular, quite simple strategy for a Wumpus World agent. Due to its declarative nature, it should be easy to improve the program in various ways:

1. Our agent tends to make more turns than necessary, because it systematically goes through the possible choices of directions for the next exploration step. The agent should rather check whether the direction it currently faces is among the possible choices, and then simply continue on its way.
2. Since the use of the arrow gives a negative reward, the agent should shoot the Wumpus only if this allows to enter areas that are otherwise inaccessible.
3. With increasing size, it gets increasingly more difficult for a cautious agent to find the gold; the average reward may be increased if the agent ventures to take the occasional step into a square that is not known to be safe.
4. The computational behavior may be tuned by a one-dimensional representation of the grid.

References

[1] Yves Lespérance *etal.* A logical approach to high-level robot programming—a progress report. In B. Kuipers, editor, *Control of the Physical World by Intelligent Agents, Papers from the AAAI Fall Symposium*, pages 109–119, 1994.

[2] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (Second Edition)*. Prentice-Hall, 2003.

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

[4] Michael Thielscher. FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 2005. Available at: `www.fluxagent.org`.

[5] Michael Thielscher. *Reasoning Robots*. Applied Logic Series 33, Kluwer Academic 2005.