

Pushing the Envelope: Programming Reasoning Agents

Michael Thielscher

Department of Computer Science
Dresden University of Technology

Abstract

Formal theories of actions have evolved in recent years into high-level programming languages for robots. While these languages allow to implement complex strategies in a declarative, concise and modular fashion, they are often doubted to be sufficiently efficient for practical purposes. In this paper we push the envelope of reasoning agents, thereby making a case for state-based solutions to the frame problem. We analyze the computational behavior of the logic-based agent programming language FLUX and show that it scales up well to problems which require reasoning about the performance of several thousand actions.

Introduction

Problem solving by reasoning about actions is one of the classical fields of study in AI. For a long time, the major goal has been to build automated systems exhibiting intelligence through the capability of planning. This was also the motivation behind the first formal model for reasoning about actions, the Situation Calculus with its concept of situations as plans (McCarthy 1963).

In recent years, however, under the paradigm of Cognitive Robotics theories of actions are applied as a methodology for programming reasoning agents. Using an explicit, symbolic representation of the outside world of a robot, this paradigm allows to implement complex behaviors in a concise and modular fashion. The most prominent example is the high-level programming language GOLOG, which is grounded in the Situation Calculus (Levesque *et al.* 1997; Reiter 2001). Yet it is an open question whether logic-based approaches like this and others, e.g., (Shanahan & Witkowski 2000), scale up to problems of non-trivial size and real-time applications.

We argue that high-level programs based on action theories can be efficient and can scale up particularly well provided a slight paradigm shift is made: Instead of relying on situations as the fundamental concept for programming agent strategies, we propose to use the concept of a state, and to appeal to situations only if an agent shall solve planning problems on its way. Our argument is supported by an analysis of the computational behavior of programs written in FLUX (the Fluent Executor (Thielscher 2002)). This high-level language is grounded in the Fluent Calculus, which combines the Situation Calculus with a state-based solution to the fundamental frame problem in classical

logic (Thielscher 1999).

In the first part of the paper, we investigate a special variant of FLUX for complete states. We report on experiments with 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 inferring the effects of an action remains constant throughout the course of the program, which shows that FLUX scales up effortlessly to arbitrarily long sequences of actions. We compare this result to GOLOG, where the curve for the computation cost suggests a polynomial increase over time.

In the second part of the paper, we investigate the computational behavior of FLUX in the presence of incomplete states. Encoding partial state knowledge as constraints, FLUX employs an efficient constraint solver to interpret and combine sensor information acquired over time. We report on a series of experiments with a combinatorial problem that involves exploring a partially known environment and acting cautiously under incomplete information. Although incomplete states pose a much harder problem, FLUX proves to scale up impressively well again: During the first phase, where the agent builds its knowledge of the environment while acting, the curve shows but a linear increase of the average action computation cost. In the second phase, where the agent acts under the still incomplete knowledge, the average time for making decisions and inferring the effects of actions remains constant again, which shows that general FLUX, too, scales up gracefully to long sequences of actions.

All experiments were carried out on a lightweight IBM Thinkpad 600 using ECLiPSe Prolog, Release 4.2, under Red Hat Linux 7.1. All programs are available for download at our web page <http://fluxagent.org>.

Agent Programs in Special FLUX

Consider the following problem: A robot acting as a postboy has to pick up and deliver in-house mail exchanged between n offices that lie along a hallway. Up to $n - 1$ packages may be requested to be delivered from each office, whereby no two packages from one office can have the same addressee. The robot can carry at most k packages at a time. Fig. 1 depicts a sample initial state in an environment where $n = 6$ and $k = 3$.

This mail delivery problem illustrates nicely the underlying principles of Cognitive Robotics, where robots use an explicit model to keep track of their actions. To

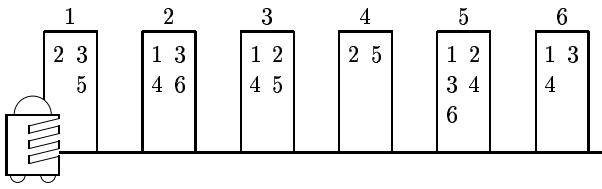


Figure 1: The initial state of a sample mail delivery problem, with a total of 21 requests.

program strategies for the mail robot, we use the language FLUX (Thielscher 2002) grounded in the action theory of the Fluent Calculus, which, based on earlier work (Fikes & Nilsson 1971; Hölldobler & Schneeberger 1990; Bibel 1986), provides a state-based solution to the Frame Problem in classical logic (Thielscher 1999).

The Fluent Calculus is a many-sorted language with four standard sorts, namely, for actions and situations (as in the Situation Calculus) and for fluents and states, resp. The program for the mail delivery robot, for example, uses these four fluents: $At(r)$, representing that the robot is at room r ; $Empty(b)$, representing that the robot’s mail bag b is empty; $Carries(b, r)$, representing that the robot carries in bag b a package for room r ; and $Request(r, r')$, indicating a delivery request from room r to room r' . States are composed of fluents (as atomic states) using the standard function \circ of sort $STATE \times STATE \mapsto STATE$ and constant $\emptyset : STATE$ (denoting the empty state). As in the Situation Calculus, the constant S_0 denotes the initial situation and $Do(a, s)$ the situation after performing action a in situation s . Finally, the state of a situation s is denoted by the standard function $State(s)$. For example, the initial state in the mail delivery scenario of Fig. 1 may be axiomatized as

$$State(S_0) = At(1) \circ Empty(Bag1) \circ Empty(Bag2) \circ Empty(Bag3) \circ Request(1, 2) \circ \dots \circ Request(1, 5)$$

The foundational axioms of the Fluent Calculus ensure that the composition function “ \circ ” exhibits the properties of the union function for sets, so that a state can be identified with the fluents that hold. On this basis, the macros $Holds(f, z)$ and $Holds(f, s)$ are defined as follows:

$$\begin{aligned} Holds(f, z) &\stackrel{\text{def}}{=} (\exists z') z = f \circ z' \\ Holds(f, s) &\stackrel{\text{def}}{=} Holds(f, State(s)) \end{aligned}$$

The three elementary actions of our mail delivery robot are: $PickUp(b, r)$, picking up and putting into bag b a package for room r ; $Deliver(b)$, delivering the contents of bag b ; and $Go(d)$, moving $d = Up$ or $d = Down$ the hallway to the next room. The Fluent Calculus expression $Poss(a, z)$ denotes that action a is possible in state z . The fundamental frame problem is solved in the Fluent Calculus by a so-called state update axiom for each action $A(\vec{x})$, which describes the effects of the action as the difference between the states

before and after performing it. E.g., the $Go(d)$ action may be specified as

$$\begin{aligned} Poss(Go(d), s) &\supset (\exists r) (Holds(At(r), s) \wedge \\ &d = Up \wedge State(Do(Go(d), s)) = \\ &\quad (State(s) - At(r)) + At(r + 1) \\ \vee \\ &d = Down \wedge State(Do(Go(d), s)) = \\ &\quad (State(s) - At(r)) + At(r - 1) \end{aligned}$$

where $Poss(a, s) \stackrel{\text{def}}{=} Poss(a, State(s))$, and where “ $-$ ” and “ $+$ ” are macros for fluent removal and addition.

For the mail delivery problem, we use a special variant of FLUX suitable for agents with complete state information. Fig. 2 depicts the general architecture of FLUX programs. The agent program itself implements the behavior of the agent with the help of two basic commands which are defined in the kernel. The expression $Holds(f, z)$ is used to evaluate conditions against the current state of the environment, as in the Fluent Calculus. Secondly, agent programs use the statement $Execute(a, z_1, z_2)$ to trigger the agent to actually perform an action a in the outside world. This command has the side-effect that current state z_1 is updated to z_2 according to the changes effected by action a . This computation in turn relies on a specification of the elementary actions of the agent. As in the Fluent Calculus, the effects of actions are encoded as state update axioms. For this purpose, the FLUX kernel provides a definition of the auxiliary predicate $Update(z_1, \vartheta^+, \vartheta^-, z_2)$. Its intuitive meaning is that state z_2 is the result of positive and negative effects ϑ^+ and ϑ^- , resp., in state z_1 . In other words, the predicate encodes the state equation $z_2 = (z_1 - \vartheta^-) + \vartheta^+$. On this basis, the agent programmer can easily encode the update axioms by clauses which define the predicate $StateUpdate(z_1, A(\vec{x}), z_2)$.

FLUX being a logic programming-based language, complete states are encoded as *ground* lists of fluents. For example, the initial state depicted in Fig. 1 may be specified by this clause:

```
init(Z0) :-
  Z0=[at(1),empty(bag1),empty(bag2),empty(bag3),
    request(1,2),...,request(6,4)].
```

Our program for the mail delivery problem uses the following axioms for preconditions and state update:

```
poss(pickup(B,R),Z) :-
  holds(empty(B),Z), holds(at(R1),Z),
  holds(request(R1,R),Z).
poss(deliver(B),Z) :-
  holds(at(R),Z), holds(carries(B,R),Z).
poss(go(D),Z) :-
  holds(at(R),Z), ( D=up, R<6 ; D=down, R>1 ).
state_update(Z1,pickup(B,R),Z2) :-
  holds(at(R1),Z1),
  update(Z1,[carries(B,R)],
    [empty(B),request(R1,R)],Z2).
state_update(Z1,deliver(B),Z2) :-
  holds(at(R),Z1),
  update(Z1,[empty(B)], [carries(B,R)],Z2).
state_update(Z1,go(D),Z2) :-
```

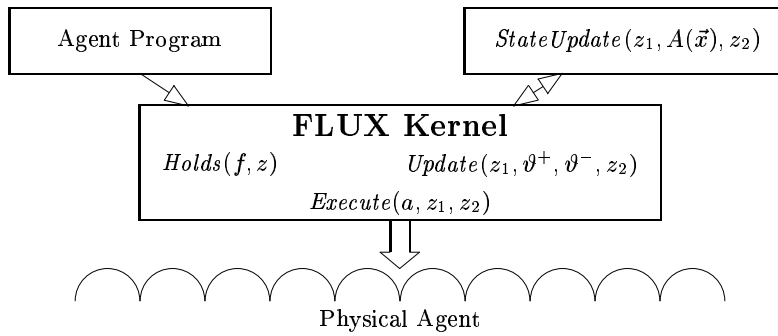


Figure 2: The architecture of a FLUX program with complete state knowledge.

```
holds(at(R), Z1),
( D=up -> R1 is R+1 ; R1 is R-1 ),
update(Z1, [at(R1)], [at(R)], Z2).
```

See (Thielscher 2002) for the formal relation between Fluent Calculus axioms and their encoding in FLUX.¹

The following program for the automatic postboy realizes a simple strategy: Whenever possible, a bag is emptied or a package is picked up; otherwise, as long as it carries a package or there are still open requests, the robot goes to the next office in the corresponding direction.

```
main :- init(Z), main_loop(Z).
main_loop(Z) :-
  poss(deliver(B), Z) -> execute(deliver(B), Z, Z1),
    main_loop(Z1) ;
  poss(pickup(B, R), Z) -> execute(pickup(B, R), Z, Z1),
    main_loop(Z1) ;
  continue(D, Z) -> execute(go(D), Z, Z1),
    main_loop(Z1) ;
  true.
continue(D, Z) :-
  ( holds(empty(_), Z), holds(request(R1, _), Z)
  ; holds(carries(_, R1), Z) ),
  holds(at(R), Z), ( R<R1 -> D=up ; D=down ).
```

The reader may notice how the cut, hidden in the “->” operator, ensures that no backtracking occurs over executed actions.

We ran a series of experiments with this FLUX program applied to maximal $\langle n, k = 3 \rangle$ problems, i.e., where packages from each office to each other office need to be delivered.² The resulting lengths of the action sequences and the measured run-time for all problem sizes from $n = 9$ up to $n = 30$ are listed in Table 1. The results show that up to a thousand actions per second are selected and their effects computed. Even for the largest problem, where the initial state contains 874 flu-

¹For efficiency reasons, executability is not checked in the state update axioms in FLUX, as it is assumed that the agent executes actions only if they are known to be possible.

²We have kept the value for k constant because while it influences the overall number of actions needed to carry out all requests, this parameter turned out to have negligible influence on the computational effort needed for action selection and state update.

n	# act	time	n	# act	time
9	368	0.19	20	3382	9.66
10	492	0.31	21	3880	12.62
11	640	0.48	22	4424	16.60
12	814	0.72	23	5016	20.87
13	1016	1.05	24	5658	26.24
14	1248	1.53	25	6352	33.02
15	1512	2.12	26	7100	40.87
16	1810	3.07	27	7904	50.09
17	2144	4.24	28	8766	61.20
18	2516	5.66	29	9688	73.55
19	2928	7.40	30	10672	88.39

Table 1: Solution length and overall run-time in seconds CPU time for the FLUX program for mail delivery problems with n offices, $k = 3$ mail bags, and maximal number of requests.

ents, the average processing time for an action is less than the hundredth of a second.

Of particular interest is the computational behavior of the program as it proceeds. Fig. 3 depicts for four selected problem sizes the average time for action selection and state update computation in the course of the execution of the program. The curves show that the computational effort remains essentially constant throughout. The slight general descent can be explained by the decreasing state size due to fewer remaining requests.

To compare these results with the original GOLOG as presented in (Levesque *et al.* 1997), we have encoded the mail delivery problem by the following successor state axioms:³

```
holds(at(R), do(A, S)) :-
  A=go(up), holds(at(R1), S), R is R1+1 ;
  A=go(down), holds(at(R1), S), R is R1-1 ;
  \+ A=go(_), holds(at(R), S).
holds(empty(B), do(A, S)) :-
  A=deliver(B) ;
  holds(empty(B), S), \+ A=pickup(B, _).
holds(carries(B, R), do(A, S)) :-
  A=pickup(B, R) ;
```

³For details regarding syntax and semantics of GOLOG, we refer to (Levesque *et al.* 1997; Reiter 2001).

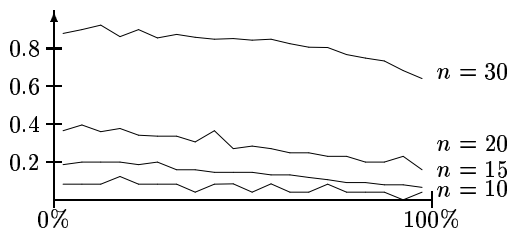


Figure 3: The computational behavior of the simple FLUX program for the mail delivery robot in the course of its execution. The horizontal axis depicts the degree to which the run is completed while the vertical scale is in seconds per 100 actions.

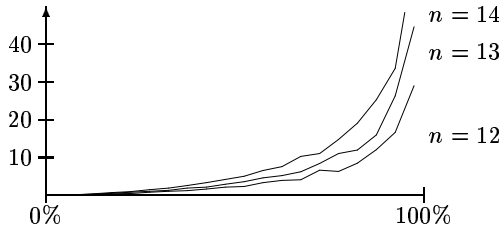


Figure 4: The computational behavior of the GOLOG program for the mail delivery problem in the course of its execution. The horizontal axis depicts the degree to which the run is completed while the vertical scale is in seconds per 100 actions.

```

holds(carries(B,R),S), \+ A=deliver(B).
holds(request(R,R1),do(A,S)) :-
holds(request(R,R1),S),
(A=pickup(_,R1) -> holds(at(R2),S), R2\R
; true ).

```

The following GOLOG program implements the same strategy as the FLUX program:

```

proc(main,[deliver(_),main] # [pickup(_,_),main]
# [continue,main] # []).
proc(continue,
[[?(empty(_)),?(request(R,_))]??(carries(_,R)),
?(at(R1)),[?(less(R1,R)),go(up)] # go(down)]).
holds(less(R1,R2),_) :- R1<R2.

```

For a fair comparison, we have tuned the GOLOG kernel so as to avoid the creation of unnecessary backtracking points. For the details, we refer to our web page containing the programs as mentioned in the introduction. Fig. 4 depicts for three selected problem sizes the average action selection time in the course of the execution of the GOLOG program. The curves show that the computational effort increases polynomially as the program proceeds. This can be explained by the increasing size of the situation term, which carries the entire history of actions and which needs to be completely regressed to the initial situation for every evaluation of a $Hold(s, f, s)$ statement.

High-level programming languages like FLUX make it easy to implement complex strategies for agents. In the following program, our mail delivery robot always

n	# act	time	n	# act	time
9	264	0.14	20	2020	5.06
10	344	0.22	21	2274	6.43
11	430	0.33	22	2600	8.02
12	534	0.47	23	2930	10.03
13	656	0.68	24	3254	12.39
14	798	0.93	25	3676	15.51
15	956	1.29	26	4098	18.80
16	1100	1.66	27	4470	22.76
17	1286	2.26	28	4996	27.78
18	1516	2.99	29	5478	33.27
19	1760	3.98	30	5980	39.18

Table 2: Solution length and overall run-time in seconds CPU time for the improved FLUX program.

chooses the closest destination when picking up a package, and it also chooses to go in the direction of the closest room in which it can either drop or pick up a package:

```

main :- init(Z), main_loop(Z).
main_loop(Z) :-
poss(deliver(B),Z) -> execute(deliver(B),Z,Z1),
main_loop(Z1) ;
fill_a_bag(B,R,Z) -> execute(pickup(B,R),Z,Z1),
main_loop(Z1) ;
continue(D,Z) -> execute(go(D),Z,Z1),
main_loop(Z1) ;
true.
fill_a_bag(B,R1,Z) :-
poss(pickup(B,_),Z) ->
holds(at(R),Z), holds(request(R,R1),Z),
\+(holds(request(R,R2),Z), closer(R2,R1,R)).
continue(D,Z) :-
holds(at(R),Z),
( holds(empty(_),Z), holds(request(_,_),Z)
->(holds(request(R1,_),Z),
\+(holds(request(R2,_),Z), closer(R2,R1,R)))
; holds(carries(_,R1),Z),
\+(holds(carries(_,R2),Z), closer(R2,R1,R)))
), ( R<R1 -> D=up ; D=down ).
closer(R1,R2,R) :- abs(R1-R)<abs(R2-R).

```

While the computation time for action selection is slightly increased under the refined strategy, all problem instances are solved much faster due to considerably fewer actions needed, as the results listed in Table 2 show. The detailed analysis reveals that the average computation time does not change drastically throughout the execution of this program, too; see Fig. 5. The slightly steeper descent can be explained by the faster decrease of the state size due to generally shorter delivery times. Due to space restrictions we refrain from discussing the corresponding GOLOG program, which behaves computationally similar to the one analyzed above.

Agent Programs in General FLUX

While the restriction to complete states enables the particularly fast execution of agent programs, the expressive power of logic-based action theories and FLUX

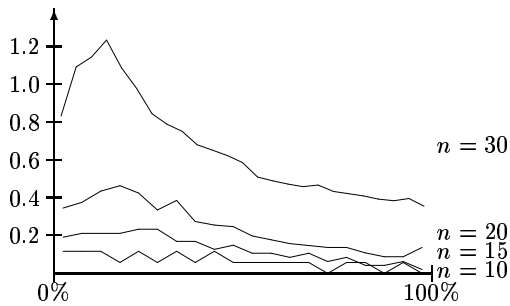


Figure 5: The computational behavior of the improved FLUX program for the mail delivery robot in the course of its execution. The horizontal axis depicts the degree to which the run is completed while the vertical scale is in seconds per 100 actions.

agents is displayed in problems where agents have to act under partial information.

Consider the following problem: After hours, a cleaning robot has to empty the waste bins in the alley and rooms of the floor of an office building. The robot shall not, however, disturb anyone working in late. The robot is equipped with a light sensor which is activated whenever the robot is adjacent to a room that is occupied, without being able to tell which direction the light comes from. An instance of this problem is depicted in Fig. 6. The task is to write a program that allows the cleaning robot to empty as many bins as possible without risking bursting into an occupied office. This problem illustrates two challenges raised by incomplete state knowledge: Agents must ensure that all their decisions are cautious, and they need to interpret and logically combine sensor information acquired over time. For example, suppose our robot starts with cleaning (1,1), (1,2), and (1,3), where it senses light. Since it cannot know whether the light comes from office (1,4) or (2,3), the cautious robot must not continue with either of these rooms. Going back and continuing with (2,2) without detecting light from an adjacent office at this point, it follows that (2,3) cannot be occupied and, hence, that the light in (1,3) comes from office (1,4). In the end, the robot should be able to clean all bins except for the ones in the occupied offices and the one in room (5,1), for the robot cannot deduce that this room is empty.

Fig. 7 gives an overview of the architecture of general FLUX programs: Facing incomplete states, evaluating conditions amounts to testing knowledge via the predicate *Knows* defined in the FLUX kernel. Furthermore, the update axioms now define a predicate *StateUpdate*($z_1, A(\vec{x}), z_2, v$), which includes the argument v for sensed values and which define *knowledge update* in the sense of (Thielscher 2000). The FLUX kernel appeals to the paradigm of constraint logic programming, which enhances logic programs by mechanisms for solving constraints. In particular, so-called Constraint Handling Rules support declarative speci-

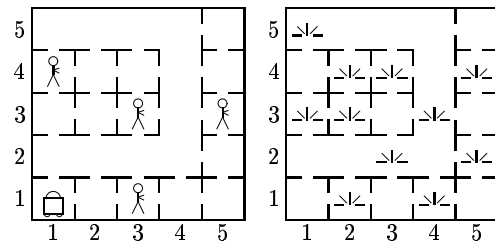


Figure 6: Layout of the office floor and sample scenario, where four offices are occupied. In the right hand side are depicted the locations in which the cleaning robot senses light.

fications 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.

Incomplete states are encoded in FLUX as *open* lists (i.e., which have a tail variable) of fluents (possibly containing further variables) along with constraints representing both negated and disjunctive state knowledge. The constraints are of the form $\neg Holds(f, z)$, $(\forall \vec{x}) \neg Holds(f, z)$ (where \vec{x} are the variables in f), and $Holds(f_1, z) \vee \dots \vee Holds(f_n, z)$ (where $n \geq 1$), denoted by `not_holds(F, Z)`, `not_holds_all(F, Z)`, and `or([F1, ..., Fn], Z)`, resp.

In our FLUX program for the cleaning robot, we use these four fluents: *At*(x, y), representing that the robot is at (x, y); *Facing*(d), representing that the robot faces direction $d \in \{1, \dots, 4\}$ (denoting, resp., north, east, south, and west); *Cleaned*(x, y), representing that the waste bin at (x, y) has been emptied; and *Occupied*(x, y), representing that (x, y) is occupied. For example, the initial knowledge of our cleaning robot includes its unique location, its facing north, the fact that neither the cleaning room nor any location in the alley or any point beyond the surrounding walls can be occupied, plus the fact that no light is perceived in (1,1):⁴

```
init(Z0) :-
    Z0 = [at(1,1),facing(1) | Z],
    not_holds_all(at(_,_),Z),
    not_holds_all(facing(_,)Z),
    not_holds(occupied(1,1),Z),
    not_holds(occupied(1,5),Z),      % alley
    not_holds(occupied(2,5),Z),...,
    not_holds_all(occupied(_,0),Z), % boundary
    not_holds_all(occupied(_,6),Z),
    not_holds_all(occupied(0,_),Z),
    not_holds_all(occupied(6,_),Z),
    light_perception(1,1,false,Z0),
    duplicate_free(Z0).
light_perception(X,Y,Percept,Z) :-
    XE#=X+1, XW#=X-1, YN#=Y+1, YS#=Y-1,
    ( Percept=false, not_holds(occupied(XE,Y),Z),
```

⁴The auxiliary constraint `duplicate_free(Z)` stipulates that list Z does not contain multiple occurrences.

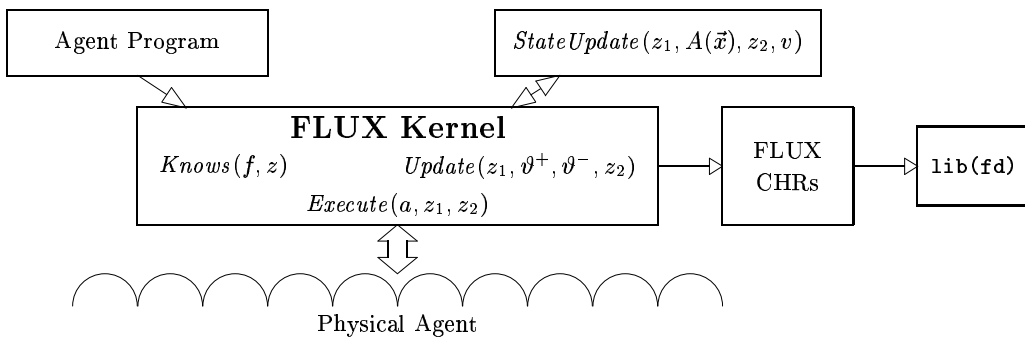


Figure 7: The architecture of a FLUX program for incomplete states.

```

not_holds(occupied(XW,Y),Z),
not_holds(occupied(X,YN),Z),
not_holds(occupied(X,YS),Z)
; Percept=true,
or([occupied(XE,Y),occupied(X,YN),
occupied(XW,Y),occupied(X,YS)],Z) ).

```

The kernel of general FLUX includes a high-speed constraint solver, which combines special constraint handling rules (based on the standard chr-library) with the standard fd-library for finite domains.⁵

The theoretical underpinnings of FLUX are given by an extension of the Fluent Calculus for reasoning about knowledge and sensing (Thielscher 2000). In particular, the update of incomplete states in FLUX is grounded in the concept of knowledge update axioms, which allow to incorporate sensor information. The three elementary actions of our cleaning robot are: *Clean*, emptying the waste bin at the current location; *Turn*, turning clockwise by 90°; and *Go*, going to the adjacent point in the faced direction. Knowledge update axioms are specified in FLUX just like ordinary state update axioms but with an additional argument encoding sensor information. Our program for the cleaning robot uses the following update axioms, whereby the execution of a *Go* action is assumed to inform the program about whether light is sensed at the new location:

```

state_update(Z1,clean,Z2,[]) :-
holds(at(X,Y),Z1),
update(Z1,[cleaned(X,Y)],[],Z2).
state_update(Z1,turn,Z2,[]) :-
holds(facing(D),Z1),
(D#<4 #/\ D1#=#D+1) #/\ (D#=#4 #/\ D1#=#1),
update(Z1,[facing(D1)], [facing(D)],Z2).
state_update(Z1,go,Z2,[Light]) :-
holds(at(X,Y),Z1), holds(facing(D),Z1),
adjacent_point(X,Y,D,X1,Y1),
update(Z1,[at(X1,Y1)], [at(X,Y)],Z2),
light_perception(X1,Y1,Light,Z2).
adjacent_point(X,Y,D,X1,Y1) :-
[X,Y,X1,Y1] :: 1..5, D :: 1..4,
(D#=#1) #/\ (X1#=#X) #/\ (Y1#=#Y+1) #/\

```

⁵The latter 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, resp.

```

(D#=#2) #/\ (X1#=#X+1) #/\ (Y1#=#Y) #\/
(D#=#3) #/\ (X1#=#X) #/\ (Y1#=#Y-1) #\/
(D#=#4) #/\ (X1#=#X-1) #/\ (Y1#=#Y) .

```

See (Thielscher 2002) for the formal relation between knowledge update axioms and their encoding in FLUX.

To control the behavior of agents, the FLUX kernel provides definitions for the expression $Knows(f, z)$ and $Knows(\neg f, z)$, denoting that in the (incomplete) knowledge state z fluent f is known to hold or not to hold, resp. On this basis, it is straightforward to write a program by which our robot cleans the floor systematically: For each newly visited location, choice points for all four directions are created. If the robot cannot safely continue on any of the remaining choice points, then it backtracks along the path it came. The program terminates with the robot ending up in its home square if the backtrack path is empty.

```

main :- init(Z0), execute(clean,Z0,Z1),
Choicepoints=[1,1,[1,2]], Backtrack=[],
main_loop(Choicepoints,Backtrack,Z1).
main_loop([X,Y,Choices|Chpts],Btr,Z) :-
Choices=[Dir|Dirs] ->
( continue_cleaning(X,Y,Dir,Z,Z1)->
execute(clean,Z1,Z2), holds(at(X1,Y1),Z1),
Chpts1=[X1,Y1,[1,2,3,4],X,Y,Dirs|Chpts],
Btr1=[X,Y|Btr], main_loop(Chpts1,Btr1,Z2)
; main_loop([X,Y,Dirs|Chpts],Btr,Z) )
; backtrack(Chpts,Btr,Z).
backtrack(_,[],Z).
backtrack(Chpts,[X,Y|Btr],Z) :-
go_back(X,Y,Z,Z1), main_loop(Chpts,Btr,Z1).
continue_cleaning(X,Y,D,Z1,Z2) :-
adjacent_point(X,Y,D,X1,Y1),
\+ knows(cleaned(X1,Y1),Z1),
knows_not(occupied(X1,Y1),Z1),
turn_to(D,Z1,Z), execute(go,Z,Z2).
go_back(X,Y,Z1,Z2) :-
holds(at(X1,Y1),Z1),
adjacent_point(X1,Y1,D,X,Y),
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).

```

Notice that sub-procedure `continue_cleaning` succeeds only if the new location is not known to be already cleaned and if it is known to be unoccupied, which is

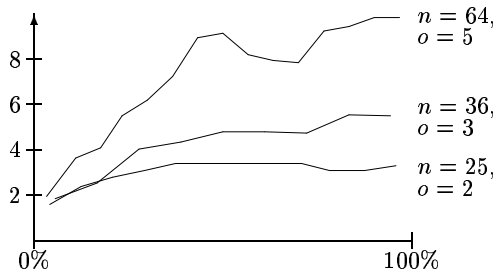


Figure 8: The average computational behavior of the FLUX program for the office cleaning robot in the course of its execution. The horizontal axis depicts the degree to which the run is completed while the vertical scale is in seconds per 100 actions.

in accordance with the requirement for cautious behavior. The reader may appreciate that there is no need to program any inference capabilities—these are fully provided by the underlying FLUX kernel. If the program is applied to the sample scenario depicted in Fig. 6, then in the end the robot has automatically derived which locations are (un-)occupied, with the exception of the uncertain status of office (1,5).

To analyze the computational behavior of the program, we ran a series of experiments with square office floors of different size and using a simulator in which several offices are randomly chosen to be occupied. For simplicity, no initial information about unoccupied cells besides (1,1) and the two adjacent ones were given to the robot. Fig. 8 depicts the results of three sets of experiments with 25 rooms (and two being occupied), 36 rooms (three), and 64 rooms (five). Each curve depicts the average of 10 runs with randomly chosen occupied locations. The results indicate two execution phases: In the first phase, the agent acquires increasing knowledge of the environment while acting. The curves show that the program scales up gracefully with just a linear increase of the average computation cost for action selection, update computation, and evaluation of sensor information. 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. In the second phase, where the agent acts under the acquired but still incomplete knowledge, the average time remains basically constant throughout, which shows that general FLUX, too, scales up particularly well to long sequences of actions. Moreover, although incomplete states pose a much harder problem, the average computation time for each action still lies well below the tenth of a second even for the largest of the three problem sizes.

Discussion

Challenging a widespread belief, we have shown that declarative, logic-based agent programs can exhibit excellent computational behavior beyond problems of toy size. In particular, we have argued for a state-based representation as a way to obtain programs which scale up well to the control of agents and robots over extended periods of time: By maintaining an explicit state term throughout the execution of the program, fluents can be directly evaluated at any stage. In contrast, the implicit representation via a situation term leads to ever increasing computational effort as the program proceeds. While this has been illustrated with a single example, the polynomial effort will dominate, in the long run, any algorithm for agent control whose inherent complexity is better than polynomial. In FLUX, the notion of a situation serves different purposes (Thielscher 2002): It is used to give semantics to program execution and, most importantly, to add the cognitive capability of planning to agent programs in accordance with the major motivation behind both the Situation Calculus and GOLOG. Inheriting much of GOLOG’s powerful concept of nondeterministic programs to guide the search for plans, and interleaving planning with state-based program execution, FLUX seems to combine the best of both worlds.

References

- Bibel, W. 1986. A deductive solution for plan generation. *New Generation Computing* 4:115–132.
- Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.
- Hölldobler, S., and Schneeberger, J. 1990. A new deductive approach to planning. *New Generation Computing* 8:225–244.
- Levesque, H. J.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming* 31(1–3):59–83.
- McCarthy, J. 1963. *Situations and Actions and Causal Laws*. Stanford University, CA: Stanford Artificial Intelligence Project, Memo 2.
- Reiter, R. 2001. *Logic in Action*. MIT Press.
- Shanahan, M., and Witkowski, M. 2000. High-level robot control through logic. In *Proc. of ATAL*, vol. 1986 of *LNCS*, 104–121. Springer.
- Thielscher, M. 1999. From Situation Calculus to Fluent Calculus. *Artificial Intelligence* 111(1–2):277–299.
- Thielscher, M. 2000. Representing the knowledge of a robot. In *Proc. of KR*, 109–120. Morgan Kaufmann.
- Thielscher, M. 2002. Programming of reasoning and planning agents with FLUX. In *Proc. of KR*. Morgan Kaufmann. (To appear.)