Logic-Based Agents and the Frame Problem: A Case for Progression

Michael Thielscher
mit@inf.tu-dresden.de
Department of Computer Science, Dresden University of Technology

Intelligent agents that reason logically about their actions have to cope with the classical Frame Problem. We argue that a progression-based solution is necessary for agent programs to run efficiently over extended periods of time. We support this claim by comparing the computational behavior of two popular logic programming systems for reasoning agents: Regression-based GOLOG and progression-based FLUX.

1 Introduction

An intriguing application of logic as a formal model of rational thought is to endow artificial systems with the ability to reason. Software agents and autonomous robots exhibit rational behavior as a result of reasoning about the effects of their actions based on an abstract, symbolic model of their environment. This approach to Artificial Intelligence is inherently connected with the famous Frame Problem of how to axiomatize the effects of actions in a concise way so as to enable an automated agent to infer what has and what has not changed after a sequence of actions [6, 7].

Throughout its history, the Frame Problem has initiated many important developments—a prominent example is nonmonotonic logic [2]—but satisfactory solutions did not emerge until the past decade. These solutions have recently evolved into declarative, high-level programming languages and systems which can be used to create reasoning agents and robots. The core of

Hendricks et al. (eds.):
First-Order Logic Revisited
Logos Verlag Berlin (2004), 323–336

each such system is its underlying inference schema for solving the Frame Problem. These inference schemata come in two different flavors.

In a *regression-based* solution to the Frame Problem, the question whether a property φ holds after the agent has performed a sequence of actions, is reduced to the question whether another property $\mathcal{R}[\varphi]$ (the *regression* of φ) holds after the last but one action. This reduction is applied recursively through the whole sequence, so that in the end the fully regressed formula can be checked against what was initially true.

In a *progression-based* solution to the Frame Problem, a (possibly incomplete) initial world model is updated upon the performance of an action. In this way, the model is progressed through an action sequence executed by the agent, and the current model is used directly to decide whether a property φ holds in the current situation. We argue that this principle is mandatory for the efficient control of agents over extended periods of time. To support this claim, we analyze and compare the computational behavior of the regression-based logic programming system GOLOG [4] with progression-based FLUX [13]. Our analysis shows that when the former is used, the computational effort continually increases as a program proceeds, whereas the latter system scales up effortlessly to long-term control.

The remainder of this paper is organized as follows. In the next section, we compare the two principles of regression and progression in the context of logic-based agents. In Section 3 we present and analyze experimental results with GOLOG and FLUX applied to a mail delivery problem which requires to reason about action sequences of non-trivial length. We conclude in Section 4. We assume that the reader is familiar with basic notations of logic programming and Prolog (as can be found, e.g., in [1]). Lack of space does also not permit to give a full explanation of syntax and semantics of GOLOG and FLUX; we refer to [4, 9] and [13], respectively.

2 Progression vs. Regression

Consider a robot whose task is to pick up and deliver mail packages exchanged among a number of offices. The robot is equipped with several slots, a kind of mail bag, each of which can be filled with one such package. Figure 1 depicts a sample scenario in an environment consisting of six offices and a robot with three mail bags. A simple, general strategy for the robot is to deliver packages whenever it finds itself at some office for which it carries

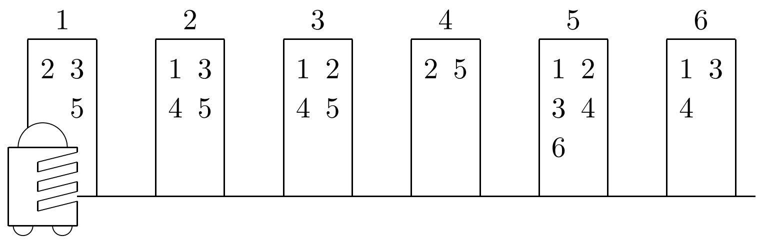

Figure 1: The initial state of a sample mail delivery problem, with a total of 21 delivery requests.

mail, then pick up packages whenever it happens to be at some place where items are still waiting to be collected, and finally move either up or down the hallway toward an office where a package can be picked up or delivered. This strategy is implemented by the following semi-formal algorithm:

loop
    if possible to deliver a package
        then do it
    else if possible to pick up a package
        then do it
    else if can pick up or deliver a package up (resp. down) the hallway
        then go up (resp. down)
    else stop
end loop

This algorithm obviously requires the robot to evaluate conditions which depend on the current state of the environment. For in order to decide on its next action, the robot always needs to know the current contents of its mail bags, the requests that are still open, and its current location. Since these properties constantly change as the program proceeds, the robot has to keep track of what it does as it moves along. For this purpose, it needs an internal representation of the environment, which throughout the execution of the program conveys the necessary information about the current location of all packages that have not yet been delivered. Logical reasoning on the basis of this model allows the robot to decide which actions are possible and how the model needs to be updated after each action in accordance with the effects of the action. With regard to the scenario in Figure 1, for instance,

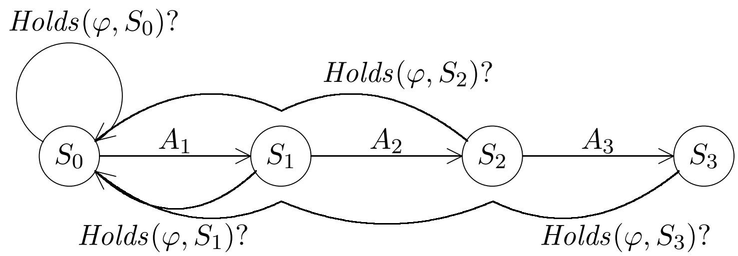

Figure 2: In regression-based solutions to the Frame Problem, the question whether a property φ holds in a situation S_i is decided by regressing φ through the actions that lead from the initial situation S_0 to S_i.

---

the robot needs to be able to conclude that it can start with putting one of the three available packages into one of its mail bags. Furthermore, the robot needs to infer that after this action, the package is in one of the mail bags while the other two bags are still empty. Hence, the robot has to cope with the Frame Problem [6].

In a *regression-based* inference schema for solving the Frame Problem [8], the question whether a property φ holds after a particular action, is reduced to the question whether another property $\mathcal{R}[\varphi]$ (the *regression* of φ) holds before the action. This reduction is applied recursively through all actions the agent has performed thus far, so that in the end the fully regressed formula can be checked against the initial world model. Figure 2 gives a schematic illustration of this principle. The graph shows that in general the effort of examining the validity of a property depends on the length of the history. As a consequence, the computational behavior of a regression-based agent program can be expected to worsen the longer the program runs.

The family of GOLOG dialects rooted in [4] is an example of regression-based implementations. The effects of actions are encoded by *successor state axioms* [8], which are of the form

$$Holds(f, Do(a, s)) \ \leftrightarrow \ \Phi_f(a, s) \tag{1}$$

Here, f is an atomic property, a so-called *fluent*, and $Do(a, s)$ denotes the *situation*, i.e., sequence of actions, reached by performing action a in situation s. Formula Φ_f describes the conditions on action a and situation s under which f can be concluded to hold in the successor situation $Do(a, s)$.

As an example, consider the following successor state axiom, given in Prolog notation, for the fluent $Empty(b)$, that is, the property of mail bag b to be empty:

```
holds(empty(B),do(A,S)) :- A=deliver(B)
                           ;
                           holds(empty(B),S),
                              not A=pickup(B,R).
```

This axiom says that mail bag b is empty after performing an action a in a situation s just in case the action was to deliver the contents of bag b, or mail bag b happened to be empty in situation s and the action was not to pick up into b a package for some room r. The atom $Holds(Empty(b), s)$ in the right hand side is solved recursively until the situation argument s is reduced to the initial situation S_0. In this way, the computational effort for deciding whether $Empty(b)$ holds depends on the number of actions performed thus far. As a consequence, the time it takes for a GOLOG agent to make a decision can be expected to increase with every action the agent takes.

In a *progression-based* inference schema for solving the Frame Problem [5, 12], a (possibly incomplete) initial world model is updated upon the performance of an action. In this way, the model is progressed through the action sequence performed by the agent, and the updated model is used directly to decide whether a property holds in the current situation. Figure 3 gives a schematic illustration of this principle. The graph shows that the effort of examining the validity of a property is independent of the length of the history. As a consequence, the computational behavior of a progression-based agent program should be expected to remain the same throughout the execution so that this principle has the potential to scale up to long-term control.

FLUX [13] is an example of a progression-based implementation. World models, so-called *states*, are encoded as lists of fluent terms, possibly accompanied by constraints for negative and disjunctive state knowledge. The effects of actions are encoded by *state update axioms* [12], which are of the form

$$StateUpdate(z_1, a, z_2) \leftarrow \Phi_a(z_1, z_2)$$

Here, formula Φ_a describes the conditions under which z_2 is the state reached by performing action a in state z_1. As an example, consider the

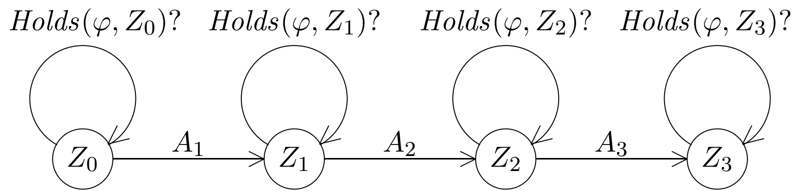

Figure 3: In progression-based solutions to the Frame Problem, the world model Z_i is progressed through the next action in every situation. A property φ can then be decided directly wrt. the current world model.

---

following state update axiom[1] for the action $Deliver(b)$ of delivering the contents of mail bag b:

```
state_update(Z1,deliver(B),Z2) :-
  holds(at(R),Z1), update(Z1,[empty(B)],[carries(B,R)],Z2).
```

This axiom says that state z_2 is the result of performing a $Deliver(b)$ action in state z_1 if the robot is at room r in z_1, and z_2 is the result of updating z_1 by the positive effect that bag b becomes empty and the negative effect that the robot no longer carries in bag b a package for room r. When executing a FLUX program, conditions of the form $Holds(\varphi, z)$ are always evaluated against the current world model. Since the computational effort for this evaluation is independent of the actions that have been performed thus far, the time it takes for a FLUX agent to make a decision is expected to remain the same as the program proceeds.

3 Progressive FLUX vs. Regressive GOLOG

In order to see how the theoretical differences between regression-based and progression-based implementations manifest in practice, we have applied both GOLOG and FLUX to mail delivery problems which require to reason about action sequences of non-trivial length. We use four fluents to describe a state in the mail delivery world: $At(r)$ to represent that the robot is at room r; $Empty(b)$ to represent that the robot's mail bag b is

[1]The standard FLUX predicate `update(Z1,P,N,Z2)` used below represents the update of state z_1 to state z_2 by positive effects p and negative effects n.

empty; $\mathit{Carries}(b,r)$ to represent that the robot carries in bag b a package for room r; and $\mathit{Request}(r,r')$ to indicate a delivery request from room r to room r'. The following logic programming clauses, for example, constitute a GOLOG specification of the initial situation depicted in Figure 1:

```
holds(at(1),s0).
holds(empty(bag1),s0).
holds(empty(bag2),s0).
holds(empty(bag3),s0).
holds(request(1,2),s0).
...
holds(request(6,4),s0).
```

The three elementary actions of the mail agent are: $\mathit{Pickup}(b,r)$ to pick up into bag b a package for room r; $\mathit{Deliver}(b)$ to deliver the contents of bag b at the current location; and $\mathit{Go}(d)$ to move $d = \mathit{Up}$ or $d = \mathit{Down}$ the hallway to the next room. Using GOLOG syntax, where $\mathit{Poss}(a,s)$ means that action a is possible in situation s, the following is a suitable definition of the action preconditions in the mail delivery world:

```
poss(pickup(B,R),S) :- holds(empty(B),S), holds(at(R1),S),
                       holds(request(R1,R),S).

poss(deliver(B),S)  :- holds(at(R),S), holds(carries(B,R),S).

poss(go(D),S)       :- holds(at(R),S),
                       ( D=up, R<6 ; D=down, R>1 ).
```

Verifying the executability of an action is a vital aspect of executing the agent program for the mail delivery robot. The effects of the actions are encoded by the successor state axioms given in Appendix A.

With the help of this background theory, our strategy for the mail delivery robot given at the beginning of Section 2 translates into the following recursive GOLOG procedure:[2]

```
proc(main_loop, [deliver(B),main_loop]  #
                [pickup(B,R),main_loop] #
                [continue,main_loop]    # []).
```

[2]For details regarding syntax and semantics of GOLOG, we refer to [4, 9].

```
proc(continue,
 [ [?(empty(B)),?(request(R1,R2))] # ?(carries(B,R1)),
   ?(at(R)), [?(less(R,R1)),go(up)] # go(down) ]).

holds(less(R1,R2),S) :- R1<R2.
```

The auxiliary procedure *Continue* succeeds if there is the possibility for the robot to pick up or deliver mail somewhere up or down the hallway. If neither a $Deliver(b)$ nor a $Pickup(b, r)$ action is possible, and if the robot needs not continue to another office, then the program terminates.

In FLUX, the initial state of Figure 1 is encoded by this clause:

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

The specification of the precondition axioms is the same as in GOLOG while the effects of the three actions are encoded by the state update axioms given in Appendix B.

The following FLUX program implements the same algorithm as the GOLOG procedure for the mail robot:

```
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(Z,Z1)
                   -> main_loop(Z1)
              ; true.

continue(Z,Z1) :- ( holds(empty(B),Z), holds(request(R1,R2),Z)
                    ; holds(carries(B,R1),Z) ),
                  holds(at(R),Z),
                  ( R<R1 -> execute(go(up),Z,Z1)
                          ; execute(go(down),Z,Z1) ).
```

Both the GOLOG and the FLUX program are available for download from our web page `www.fluxagent.org`. We ran a series of experiments

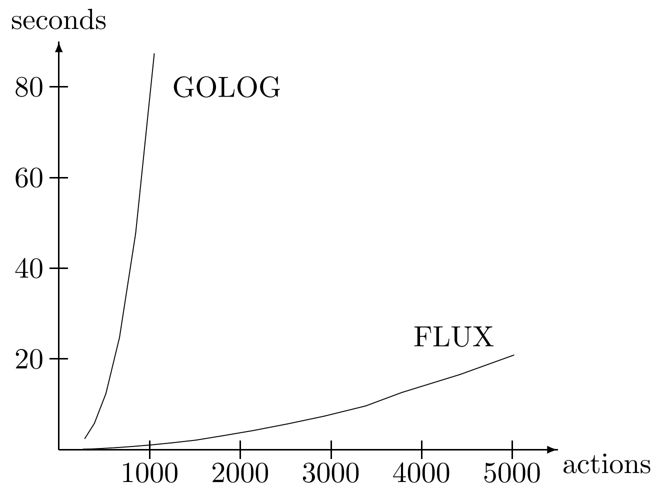

Figure 4: Overall runtime of the mail delivery program in GOLOG and FLUX (vertical axis) depending on the solution length (horizontal axis).

with maximal delivery problems, that is, with initial requests from every office to every other. The following table shows the resulting lengths of the action sequences for all problem sizes from $n = 10$ offices up to $n = 30$ and with a robot with three mail bags:[3]

n	# act	n	# act	n	# act
10	492	17	2144	24	5658
11	640	18	2516	25	6352
12	814	19	2928	26	7100
13	1016	20	3382	27	7904
14	1248	21	3880	28	8766
15	1512	22	4424	29	9688
16	1810	23	5016	30	10672

Figure 4 shows the runtime of the two programs in relation to the length of the solution. The experiments were carried out on a standard PC with

[3]We 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 effect computation.

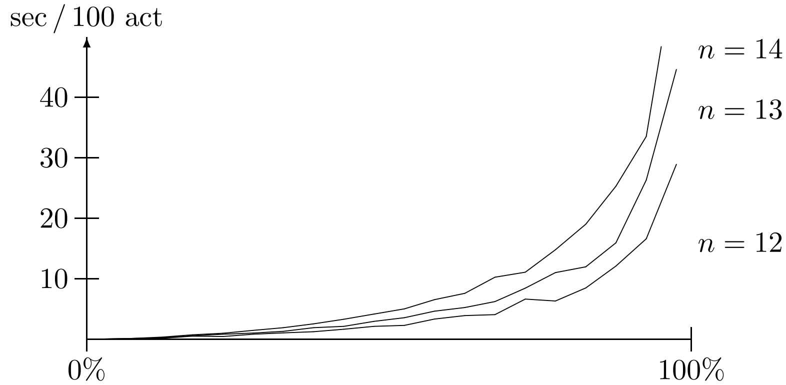

Figure 5: The computational behavior of the GOLOG program for the mail delivery problem in the course of its execution. The horizontal axis depicts the degree to which the run is completed while the vertical scale is in seconds per 100 actions.

a 500 MHz processor. A detailed analysis of the computational behavior as the two programs proceed shows that the superiority of FLUX is mainly due to its progressive solution to the Frame Problem: Figure 5 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 runs, which is a consequence of the regression-based solution to the Frame Problem. Figure 6 depicts the average time for action selection and state update computation in the course of the execution of the FLUX program, again for three selected problem sizes. The curves show that the computational effort remains essentially constant throughout, thanks to the progression-based solution to the Frame Problem. The slight general descent can be explained by the decreasing state size due to fewer remaining requests.

4 Discussion

We have argued that progression-based solutions to the Frame Problem are necessary for logic-based agents that need to reason about action sequences of non-trivial length: By continually updating their internal model of the environment, agents can evaluate properties directly at every stage. In contrast, regression-based solutions to the Frame Problem give rise to a computational

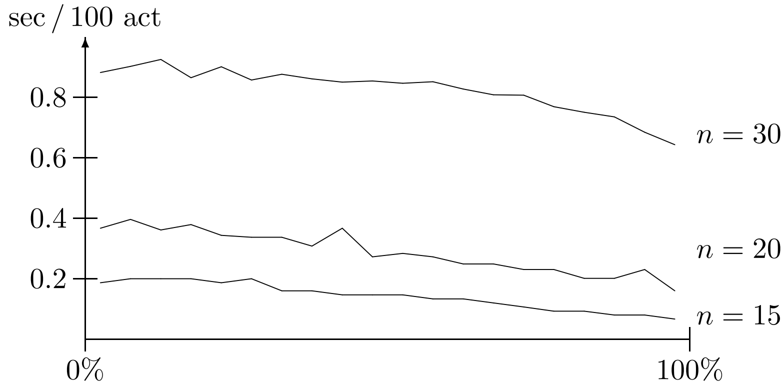

Figure 6: The computational behavior of the FLUX program for the mail delivery problem in the course of its execution. The horizontal axis depicts the degree to which the run is completed while the vertical scale is in seconds per 100 actions.

---

effort for evaluating properties which increases with every action taken by the agent. In the long run, the polynomial effort for regression worsens the complexity of any polynomial algorithm for agent control. We have shown how this difference manifests in practice by comparing regression-based GOLOG with progression-based FLUX on a problem which requires to reason about several hundreds or thousands of actions.

A prominent alternative to GOLOG, the implementation [11] of the event calculus [10] is essentially regression-based just as well: In order to verify that a property holds at some time t, it must be proved that this property was initiated by some previous event and that no event in between terminated this property. This, too, requires to take into account the history of events (i.e., actions) when examining the validity of a property, so that again the computational behavior of a control program can be expected to worsen with every action taken by the agent.

In FLUX, the notion of a history of actions serves different purposes: It is used to give semantics to program execution and to endow agents with the ability of planning. As argued in [3], since planning is a computationally demanding problem, it should be restrictively employed in agent programs and interleaved with action execution. By combining progression with much of GOLOG's powerful concept for plan search control, FLUX combines the best of both worlds.

5 Bibliography

[1] Krzysztof Apt. *From Logic Programming to Prolog*. Prentice-Hall, 1997.

[2] Daniel G. Bobrow, editor. *Artificial Intelligence 13: Special Issue on Non-Monotonic Reasoning*. Elsevier, 1980.

[3] Giuseppe De Giacomo and Hector J. 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.

[4] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1–3):59–83, 1997.

[5] Fangzhen Lin and Ray Reiter. How to progress a database. *Artificial Intelligence*, 92:131–167, 1997.

[6] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[7] Zenon W. Pylyshyn, editor. *The Robot's Dilemma: The Frame Problem in Artificial Intelligence*. Ablex, Norwood, New Jersey, 1987.

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

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

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

[11] Murray Shanahan and Mark Witkowski. High-level robot control through logic. In C. Castelfranchi and Y. Lespérance, editors, *Proceedings of the International Workshop on Agent Theories Architectures and Languages (ATAL)*, volume 1986 of *LNCS*, pages 104–121, Boston, MA, July 2000. Springer.

[12] 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.

[13] Michael Thielscher. FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 2004.

A Successor State Axioms in GOLOG

```
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
                      ; not A=go(D), holds(at(R),S).

holds(empty(B),do(A,S)) :- A=deliver(B)
                           ;
                           holds(empty(B),S),
                              not A=pickup(B,R).

holds(carries(B,R),do(A,S)) :- A=pickup(B,R)
                               ;
                               holds(carries(B,R),S),
                                  not A=deliver(B).

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

B State Update Axioms in FLUX

For the sake of simplicity and because our example domain does not involve any sensing actions, we have omitted the argument for sensory input, which is required for general update axioms [13].

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