Interpreting Golog Programs in Flux

Stephan Schiffel and **Michael Thielscher**
Department of Computer Science
Dresden University of Technology
{stephan.schiffel,mit}@inf.tu-dresden.de

Abstract

A new semantics for the programming language Golog is presented based on Fluent Calculus. The semantics lays the foundation for interpreting Golog programs in Flux. This allows to employ the principle of progression to update a state specification and to evaluate fluent conditions in Golog programs directly against an updated state.

1 Introduction

Golog is a programming language for intelligent agents that combines elements from classical programming (conditionals, loops, etc.) with reasoning about actions. Primitive statements in Golog programs are *actions* to be performed by the agent. Conditional statements in Golog are composed of *fluents*, which describe the dynamic properties of the environment in which an agent lives. The execution of a Golog program requires to reason about the effects of the actions the agent performs, in order to determine the values of fluents when evaluating conditional statements in the program.

Existing semantics for Golog are based on Situation Calculus [McCarthy, 1963; Reiter, 2001b], and existing implementations [Levesque *et al.*, 1997; Giacomo *et al.*, 2000] use successor state axioms [Reiter, 1991] when evaluating conditional statements in a Golog program. A successor state axiom defines, for an individual fluent, the value of this fluent after an action in terms of what holds before. Accordingly, the implementations use pure regression to evaluate a fluent condition, which means that a given sequence of actions is rolled back through the successor state axioms. A consequence is that the evaluation of a condition in a Golog program in general depends on the length of the history and the number of fluents whose (past) values have an influence on the conditional statement. Alternatively, progression [Lin and Reiter, 1997] can be used in combination with successor state axioms, which means to employ regression to infer the values of all fluents after an action, and then to store these values for future evaluation. This avoids to store an ever increasing history, but the computational effort of a single progression step depends on the overall number of fluents of a domain.

In this paper, we present an alternative semantics and implementation for Golog to overcome this disadvantage. The semantics is based on Fluent Calculus, which extends Situation Calculus by the concept of a *state* and in which the effects of actions are specified by state update axioms [Thielscher, 1999]. Our new semantics lays the foundation for an implementation of Golog in Flux [Thielscher, 2005], where the inference principle of progression is employed to update a state specification upon the performance of an action. The advantage over regression is that fluent conditions can be evaluated directly against the updated world model. Moreover, progression via state update axioms requires to consider the affected fluents only.

The remainder of the paper is organized as follows. In Section 2, we repeat the basic definitions of Golog. We use a variant that extends the original version by a search operator, which allows to interleave planning and execution [Giacomo *et al.*, 2000]. We also briefly recapitulate both Fluent Calculus and Flux. In Section 3, we present a Fluent Calculus semantics for Golog programs. An implementation of Golog using Flux is described in Section 4. In Section 5, we compare the computational behavior of our implementation wrt. implementations that are based on successor state axioms. We conclude in Section 6.

2 Background

2.1 Golog

We consider the original Golog defined in [Levesque *et al.*, 1997] augmented by the search operator introduced in [Giacomo and Levesque, 1999].

Being a high-level language for agent control, Golog uses actions (of the agent) as primitive statements and fluents (describing the environment) for tests, i.e., conditional statements. These basic ingredients are embedded in a language that has standard elements of imperative programming. Specifically, a Golog program can be composed of these constructs:[1]

[1]Below, $\delta, \delta_1, \delta_2$ are Golog programs, ϕ is a formula with fluents as atoms, and v is a variable.

nil	empty program
a	action
$\phi?$	test
$\delta_1;\delta_2$	sequence
$\delta_1 \mid \delta_2$	nondeterministic choice (of programs)
$\pi v.\,\delta$	nondeterministic choice (of parameters)
δ^*	nondeterministic iteration
$\Sigma\delta$	search

In addition, the following macros are used:

$$\begin{aligned} \textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 &\stackrel{\text{def}}{=} (\phi? \,;\, \delta_1) \mid (\neg\phi? \,;\, \delta_2) \\ \textbf{while } \phi \textbf{ do } \delta &\stackrel{\text{def}}{=} (\phi? \,;\, \delta)^* \,;\, \neg\phi? \end{aligned}$$

The intuitive meaning of the operator $\Sigma\delta$ is to search for a terminating run of sub-program δ and then to execute this run.[2]

Existing semantics for Golog are based on Situation Calculus [McCarthy, 1963; Reiter, 2001b], a sorted logical language with predefined sorts for fluents, actions, and situations. Situations are histories (i.e., sequences of actions), which are built using the situation constant S_0 and the function $Do(a, s)$, where a is an action and s a situation. In [Giacomo *et al.*, 2000], a transition semantics for Golog is given by an axiomatic definition of two predicates: $Trans(\delta, s, \delta', s')$, meaning that the execution of the next action or the next test in program δ leads from situation s to situation s' and to the remaining program δ'. The second predicate, $Final(\delta, s)$, means that program δ does not require to execute any more action or test in situation s. The core of this semantics are the definitions for executing an action and for evaluating a test:

$$\begin{aligned} Trans(a, s, \delta', s') &\equiv Poss(a, s) \wedge \\ &\quad\ \delta' = nil \wedge s' = Do(a, s) \\ Trans(\phi?, s, \delta', s') &\equiv \phi[s] \wedge \\ &\quad\ \delta' = nil \wedge s' = s \end{aligned} \tag{1}$$

Here, $Poss(a, s)$ denotes that action a is possible in situation s, and $\phi[s]$ means that condition ϕ holds in situation s. This requires a background theory which contains action knowledge of the application domain in form of precondition and effect axioms. Specifically, effects are described by successor state axioms [Reiter, 1991], which define the value of a particular fluent after an action (that is, in situation $Do(a, s)$), in terms of the values of fluents in situation s. Existing implementations of Golog [Levesque *et al.*, 1997; Giacomo and Levesque, 1999; Reiter, 2001b] use successor state axioms along with the principle of regression to evaluate test statements in programs. A straightforward encoding of (1), for example, is given by these two Prolog clauses:[3]

```
trans(A,S,nil,[A|S]) :- poss(A,S).
trans(?(P),S,[],S)   :- holds(P,S).
```

Put in words, the execution of an action is recorded in the situation (here encoded as list `[A|S]`). Situations are then used to evaluate test statements via `holds(P,S)`. This predicate is based on successor state axioms, by which situation `S` is "rolled back." An example is the following successor state axiom for a fluent called $doorClosed(s)$, indicating the status of the door of an elevator:

```
holds(doorClosed,do(A,S)) :-
  A=close
  ;
  holds(doorClosed,S), not A=open.
```

where the actions $close$ and $open$ bear the obvious meaning.

2.2 Fluent Calculus

The Fluent Calculus extends Situation Calculus by a predefined sort for *states*. The function $State(s)$ denotes the state (of the environment of an agent) in situation s. State terms are constructed from fluents (as singleton states) and the function $z_1 \circ z_2$, where z_1 and z_2 are states. The foundational axioms of Fluent Calculus stipulate that function "$\circ$" shares essential properties with the union operation for sets. A fluent f is then defined to hold in a state z if the former is a sub-state of the latter:

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

The addition of states to Situation Calculus allows to define fluents to hold in a situation, written $Holds(f, s)$, by referring to the state in the situation:

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

Based on the notion of a state, the frame problem [McCarthy and Hayes, 1969] is solved in Fluent Calculus by state update axioms, which define the effects of an action a as the difference between the state prior to the action, $State(s)$, and the successor $State(Do(a, s))$.

Flux

Fluent Calculus provides the formal underpinnings of the logic programming method Flux [Thielscher, 2005]. Flux is based on the encoding of (possibly incomplete) state knowledge with the help of constraints. The effects of actions are inferred on the basis of state update axioms, which effect an update of the set of constraints that encode a given state. An example is the following state update axiom for the action $close$ of closing the door of an elevator:

```
state_update(Z1,close,Z2) :-
  update(Z1,[doorClosed],[],Z2).
```

where $update(z_1, \vartheta^+, \vartheta^-, z_2)$ means that z_2 is z_1 updated by, respectively, positive and negative effects ϑ^+ and ϑ^-. Flux is thus amenable to the computation principle of progression: The current state description is updated upon the performance of an action, which allows to evaluate conditions on the successor situation directly against the new state.

3 Fluent Calculus Semantics for Golog Programs

Our starting point is the transition semantics for Golog given in [Giacomo *et al.*, 2000] with the exception of the axioms for

[2] The Golog variant of [Giacomo *et al.*, 2000] contains additional constructs, e.g., for interrupts and procedures, which we will not deal with in this paper for the sake of simplicity.

[3] The following is taken from [Giacomo and Levesque, 1999], with slight simplifications.

concurrency. With the help of the predicates *Trans* and *Final* the semantics of a Golog program can be defined as:

$$
\begin{aligned}
&Do(\delta, s, s') \stackrel{\text{def}}{=} \\
&\quad (\exists \delta') \mathit{Trans}^*(\delta, s, \delta', s') \wedge \mathit{Final}(\delta', s')
\end{aligned}
$$

where Trans^* stands for the reflexive and transitive closure of *Trans*. The predicate $Do(\delta, s, s')$ means that the execution of a Golog program δ in situation s leads to a final situation s' with a finite number of transitions.

In Fluent Calculus it is now possible to replace the situation s of a configuration by its associated state $\mathit{State}(s)$ and thus denote transitions by $\mathit{Trans}(\delta, z, \delta, z')$ and final configurations by $\mathit{Final}(\delta, z)$ where $z = \mathit{State}(s)$ and $z' = \mathit{State}(s')$. By doing this we do no longer have to calculate values of fluents by regression over the situation as in Situation Calculus. Instead we progress the state on every execution of an action. Then a simple lookup in the Fluent Calculus state is sufficient to acquire a fluents value, as, by the Completeness Assumption, all fluents that hold in a situation are part of the associated state. By replacing situations by states, however, we lose the information about the performed actions. This information is necessary in order to be able to actually execute the actions in the physical environment after reasoning about their outcomes. This is particularly important for offline execution, where a sequence of actions is executed only after the termination of the program (or a part of it) is guaranteed. Thus we introduce a list of actions not executed so far as additional result of a transition and will therefore denote a transition between two configurations by $\mathit{Trans}(\delta, z, \delta', z', h')$ where h' is the history of the actions to be executed in z in order to reach z'. In fact, the action history h' is mostly either empty or a singleton list because the transition semantics performs at most one primitive action per transition. The action history can be empty as the result of a test action.

Now we define the semantics of a Golog program in Fluent Calculus by:

$$
\begin{aligned}
&Do(\delta, z, z', h') \stackrel{\text{def}}{=} \\
&\quad (\exists \delta', h') \mathit{Trans}^*(\delta, z, \delta', z', h') \wedge \mathit{Final}(\delta', z')
\end{aligned}
$$

where $Do(\delta, z, z', h')$ means that the execution of a Golog program δ in state z results in state z' and h' contains the actions executed in between.

Before defining the predicates *Trans* and *Final* we introduce some abbreviations:

δ^v_x is an abbreviation for the substitution function $\mathit{sub}(v, x, \delta)$, which replaces all occurrences of v in a Golog program δ by x, or to be more precise, by a term $nameOf(x)$ where $nameOf$ is a mapping of Fluent Calculus objects and actions to their corresponding program terms. The full definition of *sub* can be directly adopted from [Giacomo *et al.*, 2000].

$\phi[z]$ is an abbreviation for a predicate $\mathit{HoldsCond}(\phi, z)$ which maps Golog condition expressions to Fluent Calculus state formulas.

$$
\begin{aligned}
&\mathit{HoldsCond}(p(x_1, \ldots, x_n), z) \equiv p(x_1, \ldots, x_n)[z] \\
&\mathit{HoldsCond}(\phi_1 \wedge \phi_2, z) \equiv \\
&\qquad \mathit{HoldsCond}(\phi_1, z) \wedge \mathit{HoldsCond}(\phi_2, z) \\
&\mathit{HoldsCond}(\neg \phi, z) \equiv \neg \mathit{HoldsCond}(\phi, z) \\
&\mathit{HoldsCond}((\exists v)\phi, z) \equiv \\
&\qquad (\exists y) \mathit{HoldsCond}(\phi^v_y, z)
\end{aligned}
$$

where p in the first equation is any fluent or non-fluent predicate and y in the last equation is any variable that does not appear in ϕ.

$x[z]$ is an abbreviation for the function $\mathit{decode}(x, z)$, which maps Golog program terms x (objects or actions) to their real counterparts in state z. This function is an adaptation of a similar function in [Giacomo *et al.*, 2000]:

$$
\begin{aligned}
&\mathit{decode}(nameOf(x), z) = x \\
&\mathit{decode}(g(x_1, \ldots, x_n), z) = g(x_1[z], \ldots, x_n[z]) \\
&\qquad \text{(for each non-fluent } g) \\
&\mathit{decode}(f(x_1, \ldots, x_n), z) = true \quad \equiv \\
&\qquad \mathit{Holds}(f(x_1[z], \ldots, x_n[z]), z) \\
&\qquad \text{(for each relational fluent } f) \\
&\mathit{decode}(f(x_1, \ldots, x_n), z) = v \quad \equiv \\
&\qquad \mathit{Holds}(f(x_1[z], \ldots, x_n[z], v), z) \\
&\qquad \text{(for each functional fluent } f)
\end{aligned}
$$

Functional fluents $f(x_1, \ldots, x_n)$, which have a particular value v in every situation, cannot be expressed directly in Fluent Calculus. Therefore, they are mapped onto fluents $f(x_1, \ldots, x_n, v)$, in which the value is added as argument. This requires that in all situations and for all $x_1, \ldots, x_n$ there is a unique v such that $f(x_1, \ldots, x_n, v)$ holds. The consequence of this is that an action can only change the value of $f(\vec{x})$ for a finite number of instances of $\vec{x}$, otherwise the action would have an infinite number of effects.

Now we can define the relations *Trans* and *Final* inductively for all Golog programs:

- **Empty program:**

$$
\begin{aligned}
\mathit{Trans}(nil, z, \delta', z', h') &\equiv \mathit{False} \\
\mathit{Final}(nil, z) &\equiv \mathit{True}
\end{aligned}
$$

- **Sequence:**

$$
\begin{aligned}
&\mathit{Trans}(\delta_1; \delta_2, z, \delta', z', h') \equiv \\
&\qquad (\exists \delta_1') \delta' = (\delta_1'; \delta_2) \wedge \mathit{Trans}(\delta_1, z, \delta_1', z', h') \\
&\qquad \vee \\
&\qquad \mathit{Final}(\delta_1, z) \wedge \mathit{Trans}(\delta_2, z, \delta', z', h') \\
&\mathit{Final}(\delta_1; \delta_2, z) \equiv \mathit{Final}(\delta_1, z) \wedge \mathit{Final}(\delta_2, z)
\end{aligned}
$$

- **Nondeterministic branch:**

$$Trans(\delta_1|\delta_2, z, \delta', z', h') \equiv$$
$$Trans(\delta_1, z, \delta', z', h') \vee Trans(\delta_2, z, \delta', z', h')$$
$$Final(\delta_1|\delta_2, z) \equiv Final(\delta_1, z) \vee Final(\delta_2, z)$$

- **Nondeterministic choice of argument:**

$$Trans(\pi v.\delta, z, \delta', z', h') \equiv$$
$$(\exists x) Trans(\delta^v_x, z, \delta', z', h')$$
$$Final(\pi v.\delta, z) \equiv (\exists x) Final(\delta^v_x, z)$$

- **Iteration:**

$$Trans(\delta^*, z, \delta', z', h') \equiv$$
$$(\exists \gamma)\delta' = \gamma; \delta^* \wedge Trans(\delta, z, \gamma, z', h')$$
$$Final(\delta^*, z) \equiv True$$

- **Search operator:**

$$Trans(\Sigma\delta, z, \delta', z', h') \equiv$$
$$(\exists \gamma')\delta' = \Sigma\gamma' \wedge Trans(\delta, z, \gamma', z', h') \wedge$$
$$(\exists \gamma'', z'', h'') Trans^*(\gamma', z', \gamma'', z'', h'') \wedge$$
$$Final(\gamma'', z'')$$
$$Final(\Sigma\delta, z) \equiv Final(\delta, z)$$

Notably, the definitions above are essentially just syntactical transformations of the original ones from [Giacomo *et al.*, 2000]: Situations are replaced by states, and the action history h' is added to the *Trans* predicate. The major differences arise in the definitions for primitive actions and test actions. These *Trans* predicates in Fluent Calculus are:

$$Trans(a, z, \delta', z', h') \equiv$$
$$Poss(a[z], z) \wedge \delta' = nil \wedge (\exists s) State(s) = z \wedge$$
$$z' = State(Do(a[z], s)) \wedge h' = a[z]$$
$$Trans(\phi?, z, \delta', z', h') \equiv$$
$$\phi[z] \wedge \delta' = nil \wedge z' = z \wedge h' = []$$

compared to the definition of [Giacomo *et al.*, 2000],

$$Trans(a, s, \delta', s') \equiv$$
$$Poss(a[s], s) \wedge \delta' = nil \wedge s' = Do(a[s], s)$$
$$Trans(\phi?, s, \delta', s') \equiv$$
$$\phi[s] \wedge \delta' = nil \wedge s' = s$$

The major difference is that in Situation Calculus $s' = Do(a[s], s)$ is just a variable assignment but in Fluent Calculus $(\exists s) State(s) = z \wedge z' = State(Do(a[z], s))$ results in a state update which calculates the new state z' from z and the effects of executing the action $a[z]$ in z.

This first of all means that computing a transition for a primitive action with our semantics is more expensive in terms of calculation time than with the original semantics. The reward for this is that the evaluation of $\phi[s]$ in Situation Calculus means to do a regression over the situation for each fluent f in ϕ and for each fluent on which the value of f depends. In contrast, in Fluent Calculus $\phi[z]$ can be evaluted by looking up the values of the fluents in the state z that was computed by the last transition.

Thus calculating a transition for test actions in Fluent Calculus does not depend on the length of the action history and is therefore less expensive in cases with situations of a certain length.

The *Final* predicates for primitive actions and test actions are again just syntactical transformations of the original ones:

- **Primitive action:**

$$Final(a, z) \quad \equiv \quad False$$

- **Test action:**

$$Final(\phi?, z) \quad \equiv \quad False$$

4 Flux-Interpeter for Golog

Based on the semantics defined in the previous section we have developed a Prolog implementation of a Golog interpreter in (special[4]) Flux, called "GoFlux".

The interpreter borrows elements from the LeGolog interpreter [Levesque and Pagnucco, 2000] taken from the LeGolog web page[5]. It consists mainly of the direct encoding of the *Trans*, *Final* and *HoldsCond* predicates and the additional functions *sub* and *decode*. This is combined with the kernel program of Flux for updating states on execution of actions and evaluating fluents. It requires to encode state update axioms by Prolog predicates `state_update(Z, A, Zp)` as described in Section 2. Furthermore, the predicate `prim_fluent(F)` defines all fluents `F`, and the predicate `prim_action(A)` serves the same purpose for actions. The precondition axioms for actions are encoded by a predicate `poss(A, P)` assigning to action `A` a Golog conditional expression `P` representing the precondition axiom for action `A`.

In the original LeGolog implementation, the (crucial) definition of *Trans* for primitive actions is similar to the definition of `trans` in Section 2. The precondition of the action is tested and the action is recorded in the resulting situation. In GoFlux, the definition is now as follows:

```
trans(A, Z, [], Zr, [Ap]) :-
  decode(A, Ap, Z), prim_action(Ap),
  poss(Ap, P), holdsCond(P, Z),
  state_update(Z, Ap, Zr).
```

The clause `state_update(Z,Ap,Zr)` infers a new state `Zr` resulting from the execution of the action `Ap` in state `Z`. In that way this clause for `trans` differs from the original

[4]Special Flux is an implementation of a subset of Fluent Calculus which deals only with ground states and therefore does not allow to describe uncertainty or knowledge or deal with indeterministic actions.

[5]`http://www.cs.toronto.edu/cogrobo/Legolog/`

one, whose only effect is to record the executed action `A` in the resulting situation `[A|H]`.

The second major difference arises in the implementation of *decode* and *HoldsCond*. We use a predicate `has_val(F, V, Z)` which is true iff the (functional) fluent `F` has the value `V` in state `Z`.

```
has_val(F, V, Z) :-
  func_fluent_to_rel(F, V, Frel),
  holds(Frel, Z).
```

First we convert the functional fluent `F` to a relational fluent `Frel` by adding the fluents value `V` as last argument. Then our implementation uses the Flux predicate `holds(Frel, Z)` to determine the truth-value of the fluent `Frel` in state `Z`. This predicate does just a look-up for fluent `Frel` in state `Z`. Thus the complexity depends only on the size of the state, where the state is either the initial state or computed by `state_update` on execution of an action. As opposed to the original implementation, where `has_val(F, V, S)` determines the value `V` of fluent `F` in situation `S` in that way:

- if $\mathtt{S} = [](= \mathtt{S}_0)$ then see if the fluent held initially
- if $\mathtt{S} = [\mathtt{A}|\mathtt{Sp}]$ then see (with the help of the successor state axiom for `F`) if either `A` changes the value of `F` to `V` or if `A` did not change `F` and `has_val(F, V, Sp)`

That means `has_val(F, V, S)` does a regression over the situation, i.e. the list of actions executed so far. This has potentially exponential complexity because the successor state axioms can contain conditions on other fluents and these have to be regressed as well.

Our interpreter is proved to be sound wrt. the semantics of Golog programs in Fluent Calculus [Schiffel, 2004]. The implementation is not complete in a sense that there are Golog programs δ, states z, z' and action histories h such that $Do(\delta, z, z', h)$ (or $\neg Do(\delta, z, z', h)$) is entailed by the background theory but the goal $\mathtt{Do}(\delta, \mathtt{z}, \mathtt{z'}, \mathtt{h})$ does not terminate. This is due to the nature of Golog programs, which may contain loops that will not finish with a finite number of transitions.

5 Computational Behavior

For comparing the computational behavior of our Golog interpreter *GoFlux* and a situation calculus based Golog implementation we used the IndiGolog interpreter contained in the LeGolog distribution[5]. As an example domain we took the elevator domain of [Levesque *et al.*, 1997; Reiter, 2001b] and adapted it to a more sophisticated version. In the domain we have an elevator in a building with 20 floors. For making a more intelligent elevator controller that can bring the people faster to their destinations, there is a panel at each floor where people can enter their desired destination floor while obtaining a request id. (For the sake of simplicity we assume that all the requests are already entered at the beginning, such that we don't have to consider exogenous actions.) The elevator now goes up and down bringing the people to their destination. On opening the door at a floor the elevator shows the request ids of the people which are admitted to the elevator at this stage. The elevator has only a limited capacity; we assume that only five people may be in the elevator at the same time.

The elevator controller has five primitive actions:

- $up(n)$. Move the elevator up to floor n.
- $down(n)$. Move the elevator down to floor n.
- $open$. Open the door of the elevator.
- $close$. Close the door of the elevator.
- $invite(id)$. Show request id of the person who may enter the elevator.

These actions affect the following fluents:

- $currentFloor(s) = f$. In situation s the elevator is at floor f.
- $capacity(s) = n$. In situation s there is room for n people left in the elevator.
- $request(id, f_1, f_2, s) \in \{true, false\}$. In situation s there is a request with id id of a person to go from floor f_1 to f_2.
- $carries(f, s) = n$. In situation s the elevator carries n people destined for floor f.
- $doorClosed(s) \in \{true, false\}$. In situation s the elevator door is closed.
- $lastDirection(s) \in \{up, down, none\}$. The direction the elevator last moved in before reaching situation s.

The strategy of the elevator controller is to either go up or down and stop at every floor where either he carries people to or where there is a request in its current moving direction, if there is space left in elevator. The elevator goes in the other direction if there is neither request heading from or destined in the current direction.

The core of the strategy is encoded in the following Golog procedures:

proc $serve$
  $\pi\ direction.($
    $decide\_direction(direction);$
    $(\ invite\_someone(direction)\ |$
      $goto\_next\_floor(direction)\ )$
  $)$ **endProc**;
proc $park$
  if $currentFloor \neq 0$ **then**
    $close;\ down(0);\ open$
  else nil
endProc;
proc $control$
  $($**while**$(exists\_request \vee carries\_something)$
    do $serve\ )$;
  $park$
endProc;

The GoFlux interpreter as well as the full implementation of the elevator controller is published on our web page[6].

[6] `http://www.fluxagent.org`

initial requests	Golog without	Golog with progression	GoFlux
10	19.8s	2.7s	0.9s
20	168.3s	8.7s	3.3s
100	8612.8s	146.8s	35.0s

Table 1: *Experimental results for online execution of the elevator control program by IndiGolog and GoFlux. The times were measured on a desktop computer system (1.7GHz AMD AthlonXP).*

As one can see in table 1 executing a long sequence of actions without progression is not feasible. But GoFlux is even faster than Golog with progression. This is due to the fact that progression in Golog is done by calculating the value of all fluents in the current situation from time to time and setting this as new "initial" situation by clearing the action history. In contrast to this progression in Flux is done by updating the state on every execution of an action. Thereby not every fluent's value is calculated in the new state, but only those fluents are considered, which are effects of the executed action, i.e. which change by executing the action. Since in most domains an action only changes a small set of fluents updating the state on every execution is still less expensive than calculating all fluents by regression only after some actions.

6 Conclusion

We have presented a new semantics for Golog based on the Fluent Calculus, by which the standard model for Golog is enriched with the notion of a state. The essential difference to previous semantics is that states, and not situations (i.e. histories of actions) are propagated when describing the execution of a Golog program. We have given a Fluent Calculus interpretation for all language elements of the original Golog (as defined by [Levesque *et al.*, 1997]) augmented by the search operator introduced in [Giacomo *et al.*, 2000]. All further constructs from the latter, extended dialect, e.g. those for interrupts and procedures, can be interpreted in our semantics in a straightforward way.

Our new semantics lays the foundation for interpreting Golog programs in Flux. This allows to employ states and the inference principle of progression for state update. The motivation for the alternative semantics and implementation is that

- progression of states allows to evaluate conditions in Golog programs directly against the updated state;
- progression via state update axioms only requires to consider those fluents that are affected by an action.

In an extended version of a standard scenario for Golog [Levesque *et al.*, 1997], our new implementation proved to be a more efficient interpreter for Golog compared to existing implementations. For future work, we intend to conduct systematic experiments and to analyze which computation principle is better suited for different classes of Golog programs. We are currently developing automatic translations of domain descriptions from Situation Calculus to Fluent Calculus and vice versa. Finally, we intend to extend our interpreter to knowledge-based Golog programs [Reiter, 2001a], thereby using the full expressiveness of Flux for incomplete states.

References

[Giacomo and Levesque, 1999] Giuseppe De Giacomo and Hector 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.

[Giacomo *et al.*, 2000] Giuseppe De Giacomo, Yves Lespérance, and Hector Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1–2):109–169, 2000.

[Levesque and Pagnucco, 2000] Hector Levesque and Maurice Pagnucco. Legolog: Inexpensive experiments in cognitive robotics. In *Cognitive Robotics Workshop at ECAI*, pages 104–109, Berlin, Germany, August 2000.

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

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

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

[McCarthy, 1963] John McCarthy. *Situations and Actions and Causal Laws*. Stanford Artificial Intelligence Project, Memo 2, Stanford University, CA, 1963.

[Reiter, 1991] 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.

[Reiter, 2001a] Ray Reiter. On knowledge-based programming with sensing in the situation calculus. *ACM Transactions on Computational Logic*, 2(4):433–457, 2001.

[Reiter, 2001b] Raymond Reiter. *Knowledge in Action*. MIT Press, 2001.

[Schiffel, 2004] Stephan Schiffel. Development of a fluent calculus semantics for golog programs. Großer Beleg, Dresden University of Technology, 2004.

[Thielscher, 1999] 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.

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