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:\footnote{Below, $\delta, \delta_1, \delta_2$ are Golog programs, $\phi$ is a formula with fluents as atoms, and $v$ is a variable.}

In this paper, we present an alternative semantics and implementation for Golog to overcome this disadvantage. The
In addition, the following macros are used:

$$\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2 \equiv (\phi ?; \delta_1) \backslash (\neg\phi ?; \delta_2)$$

$$\text{while } \phi \text{ do } \delta \equiv (\phi ?; \delta)^* \backslash \neg\phi ?$$

The intuitive meaning of the operator $\Sigma\delta$ is to search for a terminating run of sub-program $\delta$ 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 $\delta$ leads from situation $s$ to situation $s'$ and to the remaining program $\delta'$. The second predicate, $Final(\delta, s)$, means that program $\delta$ 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:

$$Trans(a, s, \delta', s') \equiv Poss(a, s) \land \delta' = \text{nil} \land s' = Do(a, s)$$

$$Trans(\phi?, s, \delta', s') \equiv \phi[s] \land \delta' = \text{nil} \land s' = s$$

(1)

Here, $Poss(a, s)$ denotes that action $a$ is possible in situation $s$, and $\phi[s]$ means that condition $\phi$ holds in situation $s$. This requires a background theory which contains action knowledge of the application domain (in form of preconditions 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:\(^3\)

$$\text{trans}(A, S, \text{nil}, [A|S]) :- \text{poss}(A, S).$$

$$\text{trans}(\text{nil}, P, S, [], S) :- \text{holds}(P, S).$$

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 $\text{holds}(P, S)$. 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:

$$\text{holds}(doorClosed, do(A, S)) :- A = close \text{ ; } \text{holds}(doorClosed, S), \text{ not } A = open.$$  

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:

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

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

$$\text{Holds}(f, s) \equiv \text{Holds}(f, State(s))$$

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:

$$\text{state_update}(Z1, close, Z2) :- \text{update}(Z1, [doorClosed], (, , Z2)).$$

where $\text{update}(z_1, \theta^+, \theta^-, z_2)$ means that $z_2$ is $z_1$ updated by, respectively, positive and negative effects $\theta^+$ and $\theta^-$. 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 \( \text{Trans} \) and \( \text{Final} \) the semantics of a Golog program can be defined as:

\[
\text{Do}(\delta, s, s') \overset{\text{def}}{=} (\exists \delta') \text{Trans}^* (\delta, s, \delta', s') \land \text{Final}(\delta', s')
\]

where \( \text{Trans}^* \) stands for the reflexive and transitive closure of \( \text{Trans} \). The predicate \( \text{Do}(\delta, s, s') \) means that the execution of a Golog program \( \delta \) 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 \( \text{State}(s) \) and thus denote transitions by \( \text{Trans}(\delta, z, \delta', z') \) and final configurations by \( \text{Final}(\delta, z) \) where \( z = \text{State}(s) \) and \( z' = \text{State}(s') \). 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 encode some abbreviations:

\[
\text{Final}(\delta, z) \iff \exists z': \text{Trans}(\delta, z, \delta', z') \land \text{Final}(\delta', z')
\]

where \( \text{Final}(\delta, z) \) contains the fluent predicate and \( \phi \) in the last equation is any variable that does not appear in \( \phi \).

\[
x[z] \text{ is an abbreviation for the function } \text{decode}(x, z), \text{ which maps } \text{Golog} \text{ programs to } \text{their real counterparts in state } z. \text{ This function is an adaptation of a similar function in } \text{Giacomo et al., 2000}:\
\]

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

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 \( f(\bar{x}) \) for a finite number of instances of \( \bar{x} \), otherwise the action would have an infinite number of effects.

Now we can define the relations \( \text{Trans} \) and \( \text{Final} \) inductively for all Golog programs:

- **Empty program:**
  \[
  \text{Trans}(\text{nil}, z, \delta', z', h') \equiv \text{True} \\
  \text{Final}(\text{nil}, z) \equiv \text{False}
  \]

- **Sequence:**
  \[
  \begin{align*}
  \text{Trans}(\delta_1; \delta_2, z, \delta', z', h') &\equiv \\
  (\exists \delta_1') \delta_1' &= (\delta_1; \delta_2) \land \text{Trans}(\delta_1, z, \delta_1', z', h') \\
  \lor & \\
  \text{Final}(\delta_1, z) \land \text{Trans}(\delta_2, z, \delta', z', h') \\
  \text{Final}(\delta_1; \delta_2, z) &\equiv \text{Final}(\delta_1, z) \land \text{Final}(\delta_2, z)
  \end{align*}
  \]
**Nondeterministic branch:**

\[
\text{Trans}(\delta_1 | \delta_2, z, \delta', z', h') \equiv \\
\text{Trans}(\delta_1, z, \delta', z', h') \lor \text{Trans}(\delta_2, z, \delta', z', h')
\]

\[
\text{Final}(\delta_1 | \delta_2, z) \equiv \text{Final}(\delta_1, z) \lor \text{Final}(\delta_2, z)
\]

**Nondeterministic choice of argument:**

\[
\text{Trans}(\pi \vartheta, \delta, z, \delta', z', h') \equiv \\
(\exists x)\text{Trans}(\delta^x, z, \delta', z', h')
\]

\[
\text{Final}(\pi \vartheta, \delta, z) \equiv (\exists x)\text{Final}(\delta^x, z)
\]

**Iteration:**

\[
\text{Trans}(\delta^\gamma, z, \delta', z', h') \equiv \\
(\exists \gamma)\delta^\gamma = \gamma; \delta' \land \text{Trans}(\delta, z, \gamma, z', h')
\]

\[
\text{Final}(\delta^\gamma, z) \equiv \text{True}
\]

**Search operator:**

\[
\text{Trans}(\Sigma \delta, z, \delta', z', h') \equiv \\
(\exists \gamma')\delta' = \Sigma \gamma' \land \text{Trans}(\delta, z, \gamma', z', h') \lor \\
(\exists \gamma'', z'', h''\rangle)\text{Trans}(\gamma', z', \gamma'', z'', h'') \land \\
\text{Final}(\gamma'', z'')
\]

\[
\text{Final}(\Sigma \delta, z) \equiv \text{Final}(\delta, z)
\]

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 \( \text{Trans} \) predicate. The major differences arise in the definitions for primitive actions and test actions. These \( \text{Trans} \) predicates in Fluent Calculus are:

\[
\text{Trans}(a, z, \delta', z', h') \equiv \\
\text{Poss}(a[z], s) \land \delta' = \text{nil} \land (\exists s)\text{State}(s) = z \land \\
z' = \text{State}(\text{Do}(a[z], s)) \land h' = a[z]
\]

\[
\text{Trans}(\phi?\, z, \delta', z', h') \equiv \\
\phi[z] \land \delta' = \text{nil} \land z' = z \land h' = []
\]

**Iteration:**

\[
\text{Trans}(a, s, \delta', s') \equiv \\
\text{Poss}(a[s], s) \land \delta' = \text{nil} \land s' = \text{Do}(a[s], s)
\]

\[
\text{Trans}(\phi?\, a, \delta', s') \equiv \\
\phi[s] \land \delta' = \text{nil} \land s' = s
\]

The major difference is that in Situation Calculus \( s' = \text{Do}(a[s], s) \) is just a variable assignment but in Fluent Calculus \((\exists s)\text{State}(s) = z \land z' = \text{State}(\text{Do}(a[z], s)) \) 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 \( \phi[a] \) in Situation Calculus means to do a regression over the situation for each fluent \( f \) in \( \phi \) and for each fluent on which the value of \( f \) depends. In contrast, in Fluent Calculus \( \phi[a] \) can be evaluated 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 \( \text{Final} \) predicates for primitive actions and test actions are again just syntactical transformations of the original ones:

- **Primitive action:**
  \[
  \text{Final}(a, z) \equiv \text{False}
  \]

- **Test action:**
  \[
  \text{Final}(\phi?, z) \equiv \text{False}
  \]

### 4 Flux-Interpreter 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 \( \text{Trans}, \text{Final}, \text{and HoldsCond} \) predicates and the additional functions \text{sub} and \text{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 \text{state_update}(Z, A, Zp) as described in Section 2. Furthermore, the predicate \text{prim_fluent}(F) defines all fluents \( F \), and the predicate \text{prim_action}(A) serves the same purpose for actions. The precondition axioms for actions are encoded by a predicate \text{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 \( \text{Trans} \) for primitive actions is similar to the definition of \( \text{trans} \) 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:

\[
\text{trans}(A, Z, [], Zr, [Ap]) :- \\
\text{decode}(A, Ap, Z), \text{prim_action}(Ap), \\
\text{poss}(Ap, F), \text{holdsCond}(F, Z), \\
\]

The clause \text{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 \text{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/
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, f1, f2, s) ∈ {true, false}**. In situation s there is a request with id id of a person to go from floor f1 to f2.
- **carries(f, s) = n**. In situation s the elevator carries n people destined for floor f.
- **doorClosed(s) ∈ {true, false}**. In situation s the elevator door is closed.
- **lastDirection(s) ∈ {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:

```golog
proc serve
  π direction.(  
    decide_direction(direction);  
    ( invite_someone(direction) |  
      goto_next_floor(direction) )  
  ) endProc;

proc park
  if currentFloor ≠ 0 then  
    close; down(0); open  
  else nil  
endProc;

proc control
  (while(exists:request ∨ 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.\(^6\)
As one can see in table 1 executing a long sequence of actions without progression is not feasible. But Go Flux 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


