Multi-Agent FLUX for the Gold Mining Domain (System Description)

Stephan Schiffel and Michael Thielscher

Dresden University of Technology, 01062 Dresden, Germany
{stephan.schiffel,mit}@inf.tu-dresden.de

Abstract. FLUX is a declarative, CLP-based programming method for the design of agents that reason logically about their actions and sensor information in the presence of incomplete knowledge. The mathematical foundations of FLUX are given by the fluent calculus, which provides a solution to the fundamental frame problem in classical logic. We show how FLUX can be readily used as a platform for specifying and running a system of cooperating FLUX agents for solving the Gold Mining Problem.

1 Introduction

Research in Cognitive Robotics [1] addresses the problem of how to endow agents with the high-level cognitive capability of reasoning. Intelligent agents rely on this ability when they draw inferences from sensor data acquired over time, when they act under incomplete information, and when they exhibit plan-oriented behavior. For this purpose, a mental model of the environment is formed, which an agent constantly updates to reflect the changes it has effected and the sensor information it has acquired. The fluent calculus [2] is a knowledge representation language for actions that extends the classical situation calculus [3, 4] by the concept of a state, which provides an effective solution to the fundamental frame problem [5]. FLUX [6, 7] has recently been developed as a high-level programming method for the design of intelligent agents that reason about their actions on the basis of the fluent calculus. In this paper, we show how FLUX can be readily used as a platform for specifying and running a system of cooperating agents for solving the Gold Mining Problem.

Based on constraint logic programmin, FLUX comprises a method for encoding incomplete states along with a technique of updating these states according to a declarative specification of the elementary actions and sensing capabilities of an agent. Incomplete states are represented by lists (of fluents) with variable tail, and negative and disjunctive state knowledge is encoded by constraints. FLUX programs consist of three parts: a *kernel*, which provides the general reasoning facilities by means of an encoding of the foundational axioms in the fluent calculus; a domain-specific *background theory*, which contains the formal specification of the environment, including effect axioms for the actions of the agent; and a

Name	Type	Meaning
At	$\mathbb{N} \times \mathbb{N} \mapsto$ FLUENT	position of the agent
Depot	$\mathbb{N} \times \mathbb{N} \mapsto$ FLUENT	location of the depot
Obstacle	$\mathbb{N} \times \mathbb{N} \mapsto$ FLUENT	cells containing a static obstacle
CarriesGold	FLUENT	agent carries gold

Fig. 1. The fluents for the Gold Mining domain.

strategy, which specifies the intended behavior of an agent. We present the specification of a system of multiple FLUX agents which use identical background knowledge but act autonomously using individual strategies.

2 A Fluent Calculus Specification of the Gold Mining Domain

The fluent calculus is a method for representing knowledge of actions and change in classical logic. The calculus provides the formal foundation for agents to maintain an internal, symbolic model of their environment and to reason about the effects of their actions.

2.1 Fluents and states

In the fluent calculus, the various states of the environment of an agent are axiomatized on the basis of atomic components, called *fluents*. These are formally defined as functions into the pre-defined sort FLUENT. The table in Figure 1 lists the fluents used by each individual agent in the Gold Mining domain. We assume uniqueness-of-names for all fluents $\mathcal{F} = \{At, Depot, Obstacle, CarriesGold\}$, that is,

$$\bigwedge_{\substack{F,\,G \in \mathcal{F} \\ F \neq G}} F(\overline{x}) \neq G(\overline{y}) \;\wedge\; \bigwedge_{F \in \mathcal{F}} F(\overline{x}) = F(\overline{y}) \supset \overline{x} = \overline{y}$$

A distinctive feature of the fluent calculus is that it provides an explicit representation of *states*. Informally speaking, a state is a complete collection of fluents that are true. Formally, every term of sort FLUENT is also of the special sort STATE, representing a singleton state in which just this fluent holds. The special function $\circ$: STATE $\times$ STATE $\mapsto$ STATE is used to compose sub-states into a state in which each fluent from either of the sub-states holds. The special constant $\emptyset$: STATE denotes the empty state.

A fluent is then defined to hold in a state just in case the latter contains it:[1]

$$\mathit{Holds}(f : \text{FLUENT}, z : \text{STATE}) \stackrel{\text{def}}{=} (\exists z')\, z = f \circ z' \tag{1}$$

[1] Throughout the paper, variables of sort FLUENT and STATE will be denoted by the letters f and z, respectively, possibly with sub- or superscript.

This definition is accompanied by the foundational axioms of the fluent calculus, by which, essentially, states are characterized as non-nested sets of fluents. In the following, free variables are assumed to be universally quantified.

1. Associativity and commutativity,

$$(z_1 \circ z_2) \circ z_3 = z_1 \circ (z_2 \circ z_3) \qquad z_1 \circ z_2 = z_2 \circ z_1$$

2. Empty state axiom,

$$\neg \mathit{Holds}(f, \emptyset)$$

3. Irreducibility and decomposition,

$$\begin{aligned} \mathit{Holds}(f_1, f) &\supset f_1 = f \\ \mathit{Holds}(f, z_1 \circ z_2) &\supset \mathit{Holds}(f, z_1) \vee \mathit{Holds}(f, z_2) \end{aligned}$$

4. State existence,

$$(\forall P)(\exists z)(\forall f)\,(\mathit{Holds}(f, z) \equiv P(f))$$

   where P is a second-order predicate variable of sort FLUENT.

This very last axiom guarantees the existence of a state term for all combinations of fluents. Note, however, that this does not imply that any combinations of fluents may correspond to a physically possible state. For example, in the Gold Mining domain a state containing $\mathit{At}(1,1) \circ \mathit{At}(1,2)$ as sub-state will be defined as impossible; see Section 2.2.

Semantically, a state always describes a complete state of affairs, since fluents not contained in a state are false by the definition of *Holds* (cf. (1)). On the other hand, the rigorous axiomatic definition allows for specifying incomplete state knowledge. As an example, let Z_0 denote the initial state, then the agent may know the following:

$$\begin{aligned} &(\exists a_x, a_y)\,(\mathit{Holds}(\mathit{At}(a_x, a_y), Z_0) \wedge \neg \mathit{Holds}(\mathit{Obstacle}(a_x, a_y), Z_0)) \\ &(\exists d_x, d_y)\,(\mathit{Holds}(\mathit{Depot}(d_x, d_y), Z_0) \wedge \neg \mathit{Holds}(\mathit{Obstacle}(d_x, d_y), Z_0)) \\ &\neg \mathit{Holds}(\mathit{CarriesGold}, Z_0) \end{aligned}$$

With the help of the foundation axioms, along with uniqueness-of-names, this specification can be rewritten to

$$\begin{aligned} (\exists z)(\exists a_x, a_y, d_x, d_y)\,(\, & Z_0 = \mathit{At}(a_x, a_y) \circ \mathit{Depot}(d_x, d_y) \circ z \wedge \\ & \neg \mathit{Holds}(\mathit{Obstacle}(a_x, a_y), z) \wedge \\ & \neg \mathit{Holds}(\mathit{Obstacle}(d_x, d_y), z) \wedge \\ & \neg \mathit{Holds}(\mathit{CarriesGold}, z)\,) \end{aligned} \tag{2}$$

2.2 Actions and situations

Adopted from the situation calculus [3, 4], actions are modeled in the fluent calculus by functions into the pre-defined sort ACTION. The actions of each individual agent in the Gold Mining domain are defined in Figure 2. Action

Name	Type	Meaning
Skip	ACTION	do nothing
Move	$\{East, \ldots, South\} \mapsto$ ACTION	move to the neighboring cell
Pick	ACTION	pick up gold
Drop	ACTION	drop gold
Mark	MARKING $\mapsto$ ACTION	mark a cell
Unmark	MARKING $\mapsto$ ACTION	unmark a cell

Fig. 2. The actions for the Gold Mining domain.

Mark allows to place a marker at the current location. Markers provide a simple form of communication because they can be read by other agents upon entering the cell. Our agents, however, do not use this functionality because markers can also be set, read, and erased (via action *Unmark*) by the opponents.

Sequences of actions are represented by so-called *situations*, which are axiomatized as terms of the special sort SIT. The constant $S_0 : \text{SIT}$ denotes the initial situation, and the function $Do : \text{ACTION} \times \text{SIT} \mapsto \text{SIT}$ denotes the successor situation reached by performing an action in a situation.

Situations and states are related by the pre-defined function $State : \text{SIT} \mapsto \text{STATE}$. This allows to extend the *Holds*-expression to situation arguments:[2]

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

This allows to define so-called *domain constraints*, which restrict the set of all states to those that can actually occur. The Gold Mining domain is characterized by the following domain constraints:

$$\begin{array}{l}
(\forall s)\,(\exists a_x, a_y)\,(Holds(At(a_x, a_y), s) \wedge \neg Holds(Obstacle(a_x, a_y), s)) \\
(\forall s)\,(\forall a_x, a_y, a'_x, a'_y)\,(Holds(At(a_x, a_y), s) \wedge Holds(At(a'_x, a'_y), s) \supset \\
\qquad\qquad a_x = a'_x \wedge a_y = a'_y) \\
(\forall s)\,(\exists d_x, d_y)\,(Holds(Depot(d_x, d_y), s) \wedge \neg Holds(Obstacle(d_x, d_y), s)) \\
(\forall s)\,(\forall d_x, d_y, d'_x, d'_y)\,(Holds(Depot(d_x, d_y), s) \wedge Holds(Depot(d'_x, d'_y), s) \supset \\
\qquad\qquad d_x = d'_x \wedge d_y = d'_y)
\end{array} \tag{3}$$

Put in words, the agent is always at a unique location, which cannot contain a static obstacle; and there is a unique location for the team's depot, which too is free of an obstacle.

2.3 Preconditions and state update axioms

Similar to the classical situation calculus, the executabilty of actions is axiomatized with the help of the predicate $Poss : \text{ACTION} \times \text{STATE}$. In the Gold Mining

[2] Throughout the paper, variables of sort ACTION and SIT will be denoted by the letters a and s, respectively, possibly with sub- or superscript.

domain, we have the following precondition axioms, where the auxiliary predicate $Adjacent(a_x, a_y, d, a'_x, a'_y)$ defines (a'_x, a'_y) to be adjacent to cell (a_x, a_y) in direction d.

$$\begin{aligned}
Poss(Skip, z) &\equiv \top \\
Poss(Move(d), z) &\equiv (\exists a_x, a_y, a'_x, a'_y)\,(\,Holds(At(a_x, a_y), z) \wedge \\
&\qquad\qquad Adjacent(a_x, a_y, d, a'_x, a'_y) \wedge \\
&\qquad\qquad \neg Holds(Obstacle(a'_x, a'_y), z)\,) \\
Poss(Pick, z) &\equiv \top \\
Poss(Drop, z) &\equiv Holds(CarriesGold, z) \\
Poss(Mark, z) &\equiv \top \\
Poss(Unmark, z) &\equiv \top
\end{aligned} \tag{4}$$

Note that picking up gold is always possible by definition; if the respective cell does not contain gold, the action will have no effect (see below).

Effects of actions are specified in the fluent calculus on the basis of two macros defining the removal and addition of fluents:

$$\begin{aligned}
z_1 - f = z_2 &\stackrel{\text{def}}{=} (z_2 = z_1 \vee z_2 \circ f = z_1) \wedge \neg Holds(f, z_2) \\
z_1 + f = z_2 &\stackrel{\text{def}}{=} z_2 = z_1 \circ f
\end{aligned}$$

These macros generalize to removal and addition of finitely many fluents in a straightforward way. The *hauptsatz* of the fluent calculus says that these two functions provide an effective solution to the frame problem (see, e.g., [7]):

Theorem 1. *The foundational axioms of the fluent calculus entail*

$$\begin{aligned}
&z_2 = z_1 - f_1 - \ldots - f_m + g_1 + \ldots + g_n \supset \\
&\quad [\,Holds(f, z_2) \;\equiv\; \textstyle\bigvee_i (f = g_i) \vee [Holds(f, z_1) \wedge \bigwedge_j (f \neq f_j)]\,]
\end{aligned}$$

On this basis, so-called state update axioms define the effects of an action a in a situation s as the difference between the current $State(s)$ and its successor $State(Do(a, s))$. The state update axioms in the Gold Mining domain are as follows:

$$\begin{aligned}
&Poss(Skip, s) \supset State(Do(Skip, s)) = State(s) \\
&\\
&Poss(Move(d), s) \supset \\
&\quad (\exists a_x, a_y, a'_x, a'_y)\,(\,Holds(At(a_x, a_y), s) \wedge Adjacent(a_x, a_y, d, a'_x, a'_y) \wedge \\
&\qquad\qquad State(Do(Move(d), s)) = State(s) - At(a_x, a_y) + At(a'_x, a'_y)\,) \\
&\\
&Poss(Pick, s) \supset State(Do(Pick, s)) = State(s) + CarriesGold \\
&\qquad\qquad \vee State(Do(Pick, s)) = State(s) \\
&\\
&Poss(Drop, s) \supset State(Do(Drop, s)) = State(s) - CarriesGold \\
&\\
&Poss(Mark, s) \supset State(Do(Mark, s)) = State(s) \\
&\\
&Poss(Unmark, s) \supset State(Do(Unmark, s)) = State(s)
\end{aligned} \tag{5}$$

Note that the state update axioms for action *Pick* is *nondeterministic*, that is, it does not entail whether the agent will actually be successful in trying to pick up gold. By its control strategy, however, an agent will only attempt this action in a situation where it has sensed the presence of gold in its current cell. Furthermore, the sensor information received immediately after such an action will inform the agent about its success. Since our agents do not make use of markers, these are not represented in their world model and, hence, the two actions *Mark* and *Unmark* do not change the state.

Sensor information provides additional knowledge of the environment. We represent the result of sensing in the Gold Mining domain with the help of situation-dependent predicates like $ObstacleSensed : \mathbb{N} \times \mathrm{SIT}$ such that an instance $ObstacleSensed(d, s)$ is true if an obstacle has been sensed in direction d (where d encodes the eight directions north, north-east, ...). For the sake of simplicity, and in accordance with the specification of the Gold Mining challenge, we assume that knowledge of the sensing predicates is given to the agent after any of its actions.[3] Thus, if the agent is informed about an obstacle in its vicinity, then this provides additional knowledge of the state at the current situation:

$$\begin{aligned} ObstacleSensed(s) \equiv (\exists x, y, x', y', d)\,(\, & Holds(At(x, y), s) \wedge \\ & Adjacent(x, y, d, x', y') \wedge \\ & Holds(Obstacle(x', y'), s)\,) \end{aligned}$$

3 A Multi-Agent FLUX System

FLUX [6, 7] is a high-level programming method for the design of intelligent agents that reason about their actions on the basis of the fluent calculus. Using the paradigm of constraint logic programming, FLUX comprises a method for encoding incomplete states along with a technique of updating these states via a declarative specification of the elementary actions and sensing capabilities of an agent.

Figure 3 depicts the general architecture of a FLUX control program. The kernel P_{kernel}, which endows agents with general reasoning facilities, is the same for all programs. It has been formally verified against the foundational axioms of the fluent calculus. The second part, P_{domain}, of a FLUX agent program contains encodings of the axiomatization of a particular application domain. In our system of agents for the Gold Mining domain, all agents use the same background theory. On top of this, the intended behavior of an individual agent is defined via a control program $P_{strategy}$.

[3] This is conceptually simpler than the use of an extension of the fluent calculus which is based on an explicit model of the knowledge of an agent and which allows to explicitly reason about the effects of sensing actions [8].

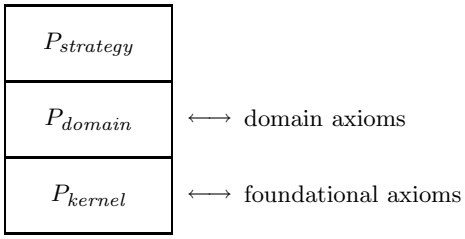

Fig. 3. The three components of a FLUX program.

3.1 State encoding

Incomplete states are represented by lists of (possibly non-ground) fluents and a variable tail, that is, `Z = [F1,...,Fk | _ ]`. It is assumed that the arguments of fluents are encoded by integers or symbolic constants, which enables the use of a standard arithmetic solver for constraints on partially known arguments. Further state knowledge is expressed by the following *state constraints* [6, 9]:

constraint	semantics
`not_holds(F,Z)`	$\neg Holds(f, z)$
`not_holds_all(F,Z)`	$(\forall \overline{x})\, \neg Holds(f, z)$, $\overline{x}$ variables in f
`or_holds([F1,...,Fn],Z)`	$\bigvee_{i=1}^{n} Holds(f_i, z)$
`if_then_holds(F,G,Z)`	$Holds(f, z) \supset Holds(g, z)$
`all_holds(F,Z)`	$(\forall \overline{x})\, Holds(f, z)$, $\overline{x}$ variables in f

These state constraints have been carefully designed so as to be sufficiently expressive while allowing for efficient constraint solving. As an example, the following clause encodes the specification of state Z_0 of Section 2.1 (cf. (2)) along with additional knowledge that follows from domain constraints (cf. (3)):

```
init(Z0) :- Z0 = [at(AX,AY),depot(DX,DY) | Z],
            not_holds_all(at(_,_), Z),
            not_holds_all(depot(_,_), Z),
            not_holds(obstacle(AX,AY), Z),
            not_holds(obstacle(DX,DY), Z),
            not_holds(carries_gold, Z).
```

The FLUX kernel includes an efficient solver for the various state constraints using Constraint Handling Rules [10], which have been formally verified against the fluent calculus [6, 9].

3.2 State update in FLUX

The second part, P_{domain}, of a FLUX agent program contains encodings of the domain axioms. These include action precondition axioms, which are straightforward encodings of the corresponding fluent calculus axioms (cf. (4)), and state

update axioms. For encoding the latter, the FLUX kernel provides the definition of a predicate `update(Z1,[G1,...,Gn],[F1,...,Fm],Z2)`, which encodes the equation $z_2 = z_1 - f_1 - \ldots - f_m + g_1 + \ldots + g_n$ of the fluent calculus.

Interaction with the environment

The physical effects of actions are straightforwardly specified in FLUX in accordance with the state update axioms. In addition, the update of the world model requires to account for the interaction of an agent with its environment, possibly including communication with other agents. To this end, effects of actions are encoded in FLUX by clauses which define the predicate `state_update(Z1,A,Z2,Y)` in such a way that performing action a in state z_1 and receiving sensor information $\overline{y}$ yields updated state z_2. The following encoding shows how the physical effects (cf. axioms (5) in Section 2.3) as well as the interaction with the environment can be specified for the Gold Mining agents:

```
state_update(Z, skip, Z, []).

state_update(Z1, move(D), Z2, [Obst]) :-
   holds(at(AX,AY), Z1), adjacent(AX, AY, D, AX1, AY1),
   update(Z1, [at(AX1,AY1)], [at(AX,AY)], Z2),
   obstacles_sensed(Obst, Z2).

state_update(Z1, pick, Z2, [Gold]) :-
     Gold = true,  update(Z1, [carriesGold], [], Z2)
   ; Gold = false, Z1 = Z2.

state_update(Z1, drop, Z2, []) :-
   update(Z1, [], [carriesGold], Z2).

state_update(Z, mark, Z, []).

state_update(Z, unmark, Z, []).
```

According to this definition, the execution of the action $Move(d)$ is assumed to return a sensing vector `Obst` of binary values indicating which of the surrounding cells house a static obstacle. Auxiliary predicate `obstacles_sensed(Obst,Z2)` is then defined in such a way as to extend the incomplete state z_2 with this knowledge. The execution of action *Pick* is assumed to return a sensing value indicating whether picking up the gold was actually successfully. This allows to resolve the nondeterminism in the corresponding state update axiom.[4] In the same fashion, information received through communication can be added to the specification of state updates in FLUX [11].

[4] The specification of the Gold Mining challenge says that a $Move(d)$ action, too, may fail with a small likelihood. This can be incorporated into the corresponding state update axiom in a similar way.

3.3 FLUX Strategies

While the FLUX agents for the Gold Mining problem share the same background theory, each one of them acts according to its individual strategy, depending on the role it plays. The agents communicate acquired knowledge of the obstacles in the environment. As a high-level programming language, FLUX allows to define complex strategies by which the agents systematically explore the environment and cooperate in transporting collected gold items to their depot.

The basic strategy of each agent is defined by the following clause, which defines the next action of an agent based on the current state z:

```
getNextAction(Action, Z) :-
  holds(at(X, Y), Z), holds(depot(DepotX, DepotY), Z),
  ( holds(carries_gold, Z) ->
     ( (X = DepotX, Y = DepotY) -> Action = drop
       ;
       direction(X, Y, DepotX, DepotY, Z, Action) )
    ;
     ( holds(gold(X,Y),Z) -> Action = pick
       ;
       gotoNextGold(X, Y, Z, Action) ) ).
```

First the agent determines its own and the depot's position. Which action to take depends on whether the agent carries gold or not. If the agent carries a gold item and is currently at the depot then the obvious action to take is dropping the gold. In case the agent is not at the depot it determines the direction it has to take to reach the depot. This might involve path planning to avoid obstacles and other agents and also to explore unknown territory on the way to the destination in order to find new gold. For the exploration of the environment, the agents also maintain a list of choicepoints along with the current path in order to be able to backtrack if necessary.

In case the agent does not carry gold at the moment it goes to a location that contains a gold item and picks up gold there. This involves deciding which known gold location to go to or which unknown territory to explore to find new gold. Our agents usually head for the nearest gold item unless another agent, which is also aiming at this cell, is closer. This condition makes sure that teammates do not compete but cooperate in collecting gold items. If there is no known gold item nearby, or all known gold items will be taken by the other agents, the agent heads to the nearest unexplored location.

For efficient calculation of the agents' actions, any plan that an agents devises is stored in the state and followed unless new knowledge is obtained which makes replanning necessary or favorable. Depending on the plan this can include failure of actions; previously unknown obstacles, or other agents, blocking the way; discovery of new gold; or teammates taking the gold the agent is heading for.

4 Discussion

The purpose of this study was to demonstrate that FLUX can be readily used for the high-level control of multiple cooperating agents in a competitive environment. Unfortunately, the team of FLUX agents did not perform very well at the actual CLIMA-06 competition. The problem, however, was entirely due to a suboptimal strategy and did not show up in any of our preceding experiments with smaller environments.

The closest work related to FLUX is the programming language GOLOG [1] for dynamic domains, which is based on successor state axioms as a solution to the frame problem. The main differences are, first, that with its underlying constraint solver, FLUX provides a natural way of representing and reasoning with incomplete states as well as nondeterministic actions. Second, the logic programs for GOLOG described in the literature all apply the principle of *regression* to evaluate conditions in agent programs, whereas FLUX is based on the *progression* of states. While the former is efficient for short action sequences, the computational effort increases with the number of performed actions. With the progression principle, FLUX programs scale up well to the control of agents over extended periods.

For future work, we intend to develop a formal model of cooperation and competition in multiagent settings based on the fluent calculus and FLUX.

References

1. Lespérance, Y., Levesque, H., etal: A logical approach to high-level robot programming—a progress report. In Kuipers, B., ed.: Control of the Physical World by Intelligent Agents, Papers from the AAAI Fall Symposium, New Orleans, LA (1994) 109–119
2. Thielscher, M.: From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. Artificial Intelligence **111** (1999) 277–299
3. McCarthy, J.: Situations and Actions and Causal Laws. Stanford Artificial Intelligence Project, Memo 2, Stanford University, CA (1963)
4. Reiter, R.: Knowledge in Action. MIT Press (2001)
5. McCarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. Machine Intelligence **4** (1969) 463–502
6. Thielscher, M.: FLUX: A logic programming method for reasoning agents. Theory and Practice of Logic Programming **5** (2005) 533–565
7. Thielscher, M.: Reasoning Robots: The Art and Science of Programming Robotic Agents. Volume 33 of Applied Logic Series. Kluwer (2005)
8. Thielscher, M.: Representing the knowledge of a robot. In: Proceedings of KR (2000) 109–120
9. Thielscher, M.: Handling implicational and universal quantification constraints in flux. In: Proceedings of CP. Volume 3709 of LNCS, Springer (2005) 667–681
10. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming **37** (1998) 95–138
11. Narasamdya, I., Martin, Y., Thielscher, M.: Knowledge of Other Agents and Communicative Actions in the Fluent Calculus. In: Proceedings of KR (2004) 623–633