The Concurrent, Continuous FLUX

Yves Martin
Technische Universität Dresden
D-01062 Dresden, Germany
ym1@inf.tu-dresden.de

Abstract

FLUX belongs to the high-level programming languages for cognitive agents that have been developed in recent years. Based on the established, general action representation formalism of the Fluent Calculus, FLUX allows to implement complex strategies in a concise and modular fashion. In this paper, we extend the FLUX language to reason about domains involving continuous change and where actions occur concurrently. Using constraint logic programming, we show that this reasoning is performed in an efficient way.

1 Introduction

One of the most challenging and promising goals of Artificial Intelligence research is the design of autonomous agents, including robots, that solve complex tasks in a dynamic world. To reach autonomy in partially known, constantly changing environments requires the high-level cognitive capabilities of reasoning and planning. Using a mental model of the state of their environment allows for the agents to calculate the outcome of different action sequences in advance and then choose the best plan to execute for a specific goal in mind.

Formal theories of reasoning about actions and change have the expressive power to provide such high-level capabilities. The Fluent Calculus [Thielscher, 1999], as one of the established action representation formalisms, uses the concept of state update axioms to solve the representational and inferential aspect of the classical Frame Problem. Based on this formal theory, the high-level programming method FLUX has been developed in recent years [Thielscher, 2002]. Using the paradigm of constraint logic programming, the powerful FLUX kernel provides general reasoning facilities, so that the agent programmer can focus on the high-level design of intelligent agents.

Autonomous agents in real-world environments have to take into account that the execution of actions takes different amounts of time. Some actions can be modeled as discrete changes, others involve continuous change and should rather be seen as the initiation or termination of complex processes. Such processes may contain parameters whose values change continuously and which are formalized as functions over time. A car moving on a road with a constant velocity v, for instance, can be represented by a 'process fluent' $Movement(x_0, t_0, v)$, where the parameter x_0 denotes the location of the car at the time t_0 when the motion was initiated. The fluent $Movement$ itself, although it describes the particular continuous change $x = x_0 + v * (t - t_0)$ for the location x of the car at the time t, will stay unchanged until some other action will affect it. Continuous change is then modeled by fluents describing arbitrarily complex, continuous processes. These fluents remain stable in between the occurrence of two consecutive actions, and yet they internally represent continuous change.

In a world full of ongoing processes, however, an agent executing a plan is not the only source of change. Also the laws of physics frequently imply an evolution of the environment, like for example, the action of a falling ball bouncing when it reaches the floor. The fundamental property of such so-called *natural* actions [Reiter, 1996] is that they must occur at their predicted times, provided no earlier actions (natural or deliberative) prevent them from occurring. Because such actions may occur simultaneously, concurrency must be accommodated.

In this paper, we present a FLUX system which allows for the design of intelligent agents that reason and plan in domains involving continuous change and where actions occur concurrently. Using the paradigm of constraint logic programming, our extension to FLUX integrates both kinds of actions, deliberative and natural, into one method for the planning and execution of actions. Reasoning in terms of time intervals, our method allows for the efficient generation of plans in concurrent, continuous environments. Our work is based on the theoretical approach presented in [Herrmann and Thielscher, 1996; Thielscher, 2001a]. Other existing agent programming methods like the robot control language developed in [Shanahan and Witkowski, 2000] or the GOLOG programming language [Levesque *et al.*, 1997], do not handle, in domains involving concurrency and continuous change, the concept of a natural action at all or only have separate accounts for natural actions on the one hand [Reiter, 2001] and for deliberative actions on the other hand [Grosskreutz, 2002].

In the next section, we introduce our example scenario and shortly recapitulate the fundamentals of the FLUX system. We then give briefly the theoretic solution in the Fluent Cal-

culus. Thereafter, we present our extension of FLUX to concurrent, continuous domains and discuss the employed constrained handling techniques. In the last section we conclude and show some possible future work. All programs are available on our web site: `http://www.fluxagent.org/`

2 FLUX

The example agent program in this paper is set in a *waterway scenario*. The agent has to steer a barge through a system of canals. The water levels in the canals are dependent on tides. At some places of the canals there are locks which a ship can only pass on a high tide. Due to the geographical facts the tide levels differ for different locks. The canals themselves are too small to turn around but sometimes there are intersections where the agent steering the barge can choose one direction (see also Fig. 1).

To develop an agent for this scenario, we use the high-level programming method FLUX which is grounded in the action theory of the Fluent Calculus. The Fluent Calculus is a many-sorted predicate logic language with four standard sorts: FLUENT, STATE, ACTION, and SIT(for situations) [Thielscher, 1999]. States are composed of fluents (as atomic states) using the standard function $\circ$: STATE $\times$ STATE $\mapsto$ STATE and constant $\emptyset$: STATE (denoting the empty state). The program for our agent, for example, uses these two fluents: $At(x, y, t)$, representing that the barge is at cell (x, y) at time t[1], and $Tide(l, w, t)$, denoting that at time t the water level w at lock l is at high/low tide. Similarly as in the Situation Calculus [Reiter, 2001], the constant S_0 denotes the initial situation and $Do(a, t, s)$ the situation after having performed action a in situation s at time t. The state of a situation s is denoted by the standard function $State(s)$. For example, the initial state in the waterway scenario of Fig. 1 may be axiomatized as[2]

$$(\exists z)\,(State(S_0) = At(1,2,0.0) \circ Tide(Lock1, High, 0.0) \circ Tide(Lock2, High, 0.0) \circ Tide(Lock3, High, 0.0) \circ z \wedge (\forall u, v, w)\, \neg Holds(At(u, v, w), z) \wedge (\forall u, v, w)\, \neg Holds(Tide(u, v, w), z))$$

The reader may notice that an incomplete state with additional negative information has been specified, i.e., the substate z may contain many more fluents, but no more *At* or *Tide* fluents. The foundational axioms of the Fluent Calculus ensure that the composition function "$\circ$" exhibits the properties of the union function for sets (with $\emptyset$ as the empty set), so that a state is identified with all the fluents that hold. On this basis, the macros $Holds(f, z)$ and $Holds(f, s)$ are defined as follows:

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

[1]In anticipation of the integration of continuous change, the argument t denotes the time when a fluent becomes true or an action takes place.

[2]Predicate and function symbols, including constants, start with a capital letter whereas variables are in lower case. Free variables in formulas are assumed universally quantified. Variables of sorts FLUENT, STATE, ACTION,and SIT shall be denoted by letters f, z, a, and s, respectively. The function $\circ$ is written in infix notation.

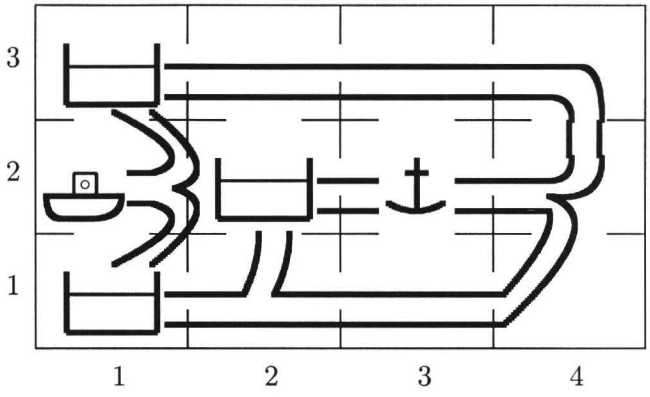

Figure 1: An example waterway scenario where the goal for the agent is to steer the barge from cell $(1, 2)$ to the harbor at cell $(3, 2)$. There are three locks at cells $(1, 3), (1, 1), (2, 2)$.

In our waterway scenario there are two elementary actions: $GoByShip(d)$, the deliberative action of the agent to steer the barge to the next cell which lies in the direction d, and $TurnTide(l, w)$, the natural action indicating the turn to the tide w at lock l. The fundamental Frame Problem is solved in the Fluent Calculus by a so-called state update axiom for each action, which describes the effects of the action in terms of the difference between the states before and after the execution of it. For example, the action $TurnTide(l, w)$ can be specified as

$$\begin{aligned} &Poss(TurnTide(l, w), t, s) \supset \\ &\quad (\exists w_0, t_0)(Holds(Tide(l, w_0, t_0), s) \wedge \\ &\quad\quad w = High \wedge State(Do(TurnTide(l, w), t, s)) = \\ &\quad\quad\quad (State(s) - Tide(l, w_0, t_0)) + Tide(l, High, t) \;\vee \\ &\quad\quad w = Low \wedge State(Do(TurnTide(l, w), t, s)) = \\ &\quad\quad\quad (State(s) - Tide(l, w_0, t_0)) + Tide(l, Low, t)) \end{aligned}$$

where "$-$" and "$+$" are macros for fluent removal and addition; and the macro $Poss(a, t, s) \stackrel{\text{def}}{=} Poss(a, t, State(s))$ denoting in the Fluent Calculus that at time t action a is possible in state $State(s)$.

To reflect the incomplete knowledge of an agent about its environment, incomplete states are encoded in FLUX as open lists, that is, lists with a variable tail, of fluents [Thielscher, 2002]. These lists are accompanied by constraints for negated state knowledge as well as for variable range restrictions. The constraints are of the form $NotHolds(f, z)$, indicating that fluent f does not hold in state z; and $NotHoldsAll(f, z)$, indicating that no instance of f holds in z. In order to process these constraints, so-called declarative Constraint Handling Rules [Frühwirth, 1998] have been defined and proved correct under the foundational axioms of the Fluent Calculus (for details see [Thielscher, 2002]).

For example, the initial state depicted in Fig. 1 may be specified by this clause,

```
init(Z0) :-
 Z0=[at(1,2,0.0),tide(lock1,high,0.0),tide(
     lock2,high,0.0),tide(lock3,high,0.0)|Z],
 not_holds_all(at(_,_,_),Z),
 not_holds_all(tide(_,_,_),Z).
```

which also reflects the negative information that no At fluent occurs in sub-state z (the location of our agent is unique), and that there are no more $Tide$ fluents other than specified in state z_0.

The predicate $Poss(a, t, z)$ realizes the precondition axioms for actions, that is, it defines under which conditions an action is possible at time t in state z. There is one such predicate clause for each action. Conditioning in FLUX is based on the foundational predicates $Knows(f, z)$, $KnowsNot(f, z)$, and $KnowsVal(\vec{x}, f, z)$, representing that the agent knows that fluent f holds (respectively, does not hold) in state z, and that there exist ground instances of the variables in $\vec{x}$ such that fluent f is known to be true in state z. Take, for example, the precondition axiom for the action $TurnTide(l, w)$ which is implemented as follows,

```
poss(turntide(L,W),T,Z) :-
 (W=high, knows_val([L,TR],tide(L,low,TR),Z),
  duration(L,low,D);
  W=low, knows_val([L,TR],tide(L,high,TR),Z),
  duration(L,high,D)),
 {T =:= TR + D}.
```

where the auxiliary predicate $Duration(l, w, d)$ denotes the duration d of the corresponding tide w at lock l. The execution time t of the action will be constrained, using the syntax of the constraint handling library CLP(R) of the Eclipse-Prolog system, to be the sum of the starting time of the old tide plus the duration of the old tide.

As in the Fluent Calculus, the effects of actions are encoded as state update axioms. For this purpose, the FLUX kernel provides a definition of the auxiliary predicate $Update(z_1, \vartheta^+, \vartheta^-, z_2)$. Its intuitive meaning is that state z_2 is the result of positive and negative effects ϑ^+ and ϑ^-, respectively, wrt. state z_1. In other words, the predicate encodes the state equation $z_2 = (z_1 - \vartheta^-) + \vartheta^+$. On this basis, the agent programmer can easily implement the update axioms by clauses which define the predicate $StateUpdate(z_1, a, t, z_2)$, as for example in the following encoding for the action $TurnTide(l, w)$:

```
state_update(Z1,turntide(L,W),T,Z2) :-
 holds(tide(L,W0,T0),Z1),
 W=high, update(Z1,[tide(L,high,T)],
                   [tide(L,W0,T0)],Z2);
 W=low, update(Z1,[tide(L,low,T)],
                  [tide(L,W0,T0)],Z2).
```

3 The concurrent, continuous Fluent Calculus

The Fluent Calculus for concurrent actions is based on the additional pre-defined sort CONCURRENT, of which ACTION is a sub-sort [Thielscher, 2001a]. Single actions which are performed simultaneously are composed to terms of sort CONCURRENT by a new binary function. The latter is denoted by "·" and written in infix notation. This function shares with the function combining fluents to states the properties of associativity, commutativity, idempotency and existence of a unit element. The constant "ϵ" (read: *no-op*) of sort CONCURRENT acts as the unit element wrt. function "·". Similar to the *Holds* macro the abbreviation $In(c_1, c)$ is used to denote that concurrent action c_1 is included in concurrent action c[3]:

$$In(c_1, c) \stackrel{\text{def}}{=} (\exists c')c = c_1 \cdot c'$$

State update axioms for concurrent actions are recursive. They specify the effect of an action relative to the effect of arbitrary other, concurrent actions:

$$\begin{aligned} &Poss(a \cdot c, t, s) \wedge \Delta(c, t, s) \supset \\ &\quad State(Do(a \cdot c, t, s)) \circ \vartheta^- = State(Do(c, t, s)) \circ \vartheta^+ \end{aligned}$$

I.e., ϑ^- and ϑ^+ are the additional negative and positive, respectively, effects which occur if action a is performed besides c. Here, a can be a single action or a compound action which produces synergic effects, that is, effects which no single action would have if performed alone. Using recursive state update axioms, the effect of, say, two simultaneous but independent actions can be inferred by first inferring the effect of one of them and, then, inferring the effect of the other action on the result of the first inference. The recursions stops with the base case of the empty action, which is defined as:

$$State(Do(\epsilon, t, s)) = State(s)$$

Two or more actions may interfere when executed concurrently, which is why the condition Δ in the above state update axiom may restrict the applicability of the implication in view of concurrent action c.

Integrating continuous change in the Fluent Calculus requires the introduction of process fluents which can represent arbitrarily complex, continuous processes. Because such processes may be modeled by equations of motions, continuous time must be represented. To this end, the new sort REAL is added, which is to be interpreted as the real numbers [Thielscher, 2001a]. The sort is accompanied by the usual arithmetic operations along with their standard interpretation. The continuous Fluent Calculus includes the pre-defined fluent $StartTime(t)$, where t is of sort REAL, determining the time $Start(z)$ at which a state arises, provided that $StartTime$ is unique:

$$\begin{aligned} &(\exists! t) Holds(StartTime(t), z) \supset \\ &\quad (\forall t)(Holds(StartTime(t), z) \supset Start(z) = t) \end{aligned}$$

As already indicated throughout the paper, a parameter t of sort REAL is also used to denote the time at which a fluent arises as in $Tide(l, w, t)$ or to represent the occurrence of an action as in $Do(a, t, s)$. A standard requirement for the possibility to perform a concurrent action c at time t in state z may then be expressed as follows:[4]

$$Poss(c, t, z) \equiv c \neq \epsilon \wedge (\forall a)(In(a, c) \supset Poss(a, t, z))$$

The Fluent Calculus for continuous change includes the distinction between deliberative and natural actions [Thielscher, 2001a]. The latter are not subject to the free will of a planning agent. Rather they happen automatically under specific circumstances. In our example domain, the

[3] Variables of the new sort CONCURRENT are denoted by the letter c.

[4] In our example domain we have no actions that are in mutual conflict. Therefore, we do not need to specify additional constraints in this precondition axiom.

action of the turn of the tide is a natural one. The standard predicate $Natural(a)$ [adopted from [Reiter, 1996]] declares the action a to be natural. To facilitate the formalization of the automatic evolution of natural actions, the continuous Fluent Calculus introduces two macros. The expression $ExpectedNatActions(c, t, z)$ shall indicate that in state z actions c are all the natural actions that are expected to happen at time t:

$$ExpectedNatActions(c, t, z) \stackrel{\text{def}}{=} c \neq \epsilon \wedge (\forall a)(In(a, c) \equiv Natural(a) \wedge Poss(a, t, z))$$

Given the above notion, the macro $NextNatActions(c, t, z)$ stands for the concurrent action c being all natural actions that happen in state z at time t with t being the earliest possible time point at which natural actions are expected:

$$NextNatActions(c, t, z) \stackrel{\text{def}}{=} ExpectedNatActions(c, t, z) \wedge \neg(\exists t', c')(ExpectedNatActions(c', t', z) \wedge t' < t)$$

The Fluent Calculus for continuous change uses the notion of a situation tree with trajectories [Thielscher, 2001a] where a trajectory is associated with a situation and denotes the further evolution of the state determined by the natural actions that are expected to happen. We do not follow this approach in this paper, as the original motivation for employing trajectories has been domains with uncertainty about the occurrence of natural actions, and we do not consider such domains here. The incorporation of domains with uncertain natural actions is left for further work. Instead, we include the natural actions in the situation terms, as with deliberative actions.

4 Integrating Concurrency and Continuous Change into FLUX

Similar to the binary function "$\circ$", which denotes the composition of states from single fluents and is represented in FLUX by a list of fluents, we represent the binary function "$\cdot$" as a list of actions in FLUX. In this way, we introduce concurrency into FLUX, i.e., all the actions in the list are performed concurrently. On this basis, the unit element of the function "$\cdot$", the constant "ϵ", is encoded as the empty list `[]`.

Given the notion of a list of concurrent actions, the state update axioms for the actions are defined recursively in FLUX. The predicate $Res(z_1, s_1, li, t, z_2, Do(li, t, s_1))$ specifies the effect of performing at time t the list of concurrent actions li in state z_1 and situation s_1, and leading to the new state z_2 and the new situation $Do(li, t, s_1)$ after the execution of the concurrent actions. It represents one plan step and is encoded as follows

```
res(Z1,S1,[],T,Z2,S1) :-
 holds(starttime(T0), Z1),
 update(Z1,[starttime(T)],[starttime(T0)],Z2).

res(Z1,S1,[A|L],T,Z2,do([A|L],T,S1)) :-
 update([A|L],[],[A],L1),
 state_update(Z1, A, T, Z3),
 res(Z3,_,L1,T,Z2,_).
```

where the clause for the base case of the recursion extends the effect of the constant "ϵ", which by itself has none, to update the pre-defined fluent $StartTime(t)$ needed for FLUX with continuous change to the new value t. Similar to the encoding of the macros for fluent removal and addition, the predicate *Update* is used in the clause for the recursive case to implement the elimination of one single action from a term of arbitrary other, concurrent actions.

Consider, for example, the state update axiom of the single action $GoByShip(d)$ which is implemented as:

```
state_update(Z1,gobyship(D),T,Z2) :-
 knows_val([X,Y],at(X,Y,T0),Z1),
 holds(starttime(ST),Z1), {T >= ST + 1.0},
 adjacent(X,Y,D,X1,Y1),
 update(Z1,[at(X1,Y1,T)],[at(X,Y,T0)],Z2).
```

That is, the location of the barge together with the agent will be updated from the old position (x, y) to the new cell (x_1, y_1), where the auxiliary predicate $Adjacent(x, y, d, x_1, y_1)$ computes the adjacent cell (x_1, y_1) lying in direction d of cell (x, y). The travel from one cell to another is assumed to take one hour in our example scenario. Therefore, although the execution time t of this action is not fixed by the clause in any way, the effects of the action manifest at least one hour later as the formation of the old state. Now take, e.g., the FLUX query

```
?-init(Z0), res(Z0,s0,[gobyship(3),
            turntide(lock3,low)],2.0,Z1,S1).
```

together with the above definition of the state update axiom and the definitions for the predicate $Init(z)$ (with the additional inclusion of fluent `starttime(0.0)`) and the state update axiom of action *TurnTide* given in Section 2. Together with an appropriate encoding of the fact that a high tide lasts two hours at the third lock, `duration(lock3,high,2.0)`, and the knowledge that going south is represented by direction number `3`, our extension of FLUX can infer the effects of this concurrent action by first inferring the effects of action `gobyship(3)` and on the result of this inference infer the effects of action `turntide(lock3,low)`. FLUX yields the correct substitution:

```
Z1=[at(1,1,2.0),tide(lock1,high,0.0),
    tide(lock2,high,0.0),tide(lock3,low,2.0),
    starttime(2.0)|Z]
S1=do([gobyship(3),
       turntide(lock3,low)],2.0,s0)
```

Continuous time is, as already shown above and in Section 2, easily integrated into FLUX. The Eclipse-Prolog system Version 5.4, which we use, includes the constraint handling library CLP(R). This library allows for solving linear constraints with real numbers. Its syntax requires for constraints to be included in braces.

The precondition axioms and state update axioms for natural actions are encoded in our extension of FLUX in the same fashion as for deliberative actions (see Section 2). Additionally, we include an implementation for the predicate $Natural(a)$ which is as follows for the example domain:

```
natural(A) :- A=turntide(L,W),
              (L=lock1; L=lock2; L=lock3).
```

Given this predicate, we model the macro $ExpectedNatActions(c, t, z)$ by the built-in second order predicate *SetOf* as follows:

```
setof(A, (natural(A),poss(A,T,Z)), C)
```

On this basis, the macro $NextNatActions(c, t, z)$ is defined in FLUX as:

```
NextNatTime(T,Z) :-
 natural(A), poss(A,T,Z),
 not (natural(A1), poss(A1,T1,Z), T1<T), !.

NextNatActions(C,T,Z) :- NextNatTime(T,Z),
 setof(A, (natural(A),poss(A,T,Z)), C).
```

Having defined natural actions in the same way as deliberative ones leads to the question: How to combine deliberative and natural actions into one common approach for planning? Natural actions must occur at their predicted times. The times for the execution of deliberative actions are not fixed in advance. How to determine these execution times? The examination of every possible time would lead to a combinatorial explosion and is, in general, not possible for time of sort REAL. One general solution, which reduces the search space to a minimum and still yields answers for all possible domains, is to use qualitative instead of quantitative information. We only consider periods of time in our approach. There we discriminate three time intervals for a deliberative action wrt. the next expected natural action(s): Firstly, the deliberative action can be postponed to the next plan step and the (possible set of) natural action(s) is executed. Secondly, the deliberative action is performed before all natural actions which are expected next. Finally, the deliberative action and the natural actions are joined together to a new concurrent action. The predicate $Exec(z_1, s_1, z_2, s_2, depth)$ encodes the recursive planner for a plan with *depth* steps integrating deliberative and natural actions into one method. The computed plan leads from state z_1 and situation s_1 to the new state z_2 and the new situation s_2. The predicate is implemented as

```
exec(Z1,S1,Z1,S1,0).

exec(Z1,S1,Z2,S2,Depth) :-
 Depth>0, NextNatActions(C,T,Z),
 ({T1=T}, C1=C;
  action(A), poss(A,TA,Z1),
  holds(starttime(T0),Z1), {TA>=T0},
  ({TA<T, T1=TA}, C1=[A];
   {TA=T, T1=T}, append([A],C,C1))),
 res(Z1,S1,C1,T1,Z3,S3), Depth1 is Depth-1,
 exec(Z3,S3,Z2,S2,Depth1).
```

where the predicate $Action(a)$ defines the action a to be deliberative and the auxiliary built-in predicate *Append* appends two list. For our example domain, the predicate *Action* is encoded by the fact `action(gobyship(_))`. To ensure that the time never goes backward, the execution time TA of the deliberative action is constrained in an appropriate way.

Reasoning with time constraints instead of real time renders planning efficient. Using this plan method we are left with only three choices regarding the execution time of a deliberative action. Furthermore, the order of these choices, which represents a kind of heuristic, can be adjusted to the concrete domain at hand. Only after a plan, where the constraint solution lies in the appropriate time intervals, has been computed, a concrete time for the execution of the actions is fixed and the actions are executed.

To complete our planning method, we include the following definition of the predicate $Goal(z)$ denoting the goal state z,

```
goal(Z) :- knows(at(3,2,_),Z).
```

where the goal in our example scenario is that the barge is situated at the harbor in cell $(3, 2)$ (see also Fig. 1). Given the predicate *Goal*, we define a recursive predicate $Ida(z_0, s_0, z, s, n)$ representing the sequence of actions s which leads from the initial state z_0 in the initial situation s_0 to the goal state z in n steps. This predicate implements the iterative deepening algorithm, which is optimal and complete [Russell and Norvig, 1995]. It is encoded as follows:

```
ida(Z0,S0,Z,S,N) :-
 exec(Z0,S0,Z,S,N), goal(Z);
 N1 is N+1, findplan(Z0,S0,Z,S,N1).
```

The precondition axiom for the action *GoByShip* is specified in the following way:

```
poss(gobyship(D),T,Z) :-
 knows_val([X,Y],at(X,Y,_),Z),
 directions(X,Y,DL), member(D,DL)
 adjacent(X,Y,D,X1,Y1),
 (lockplace(L,X1,Y1),
  knows_val([L,W,T0],tide(L,W,T0),Z),
  (W=high; W=low,
   duration(L,low,TD), {T>=T0+TD}));
  not (lockplace(L,X1,Y1)).
```

That is, after having determined the current location, the auxiliary predicate $Directions(x, y, dl)$ delivers a list of possible directions for the cell (x, y) and the standard predicate *Member* selects one direction d. Afterwards, with the help of the auxiliary predicate $LockPlace(l, x, y)$, denoting the occurrence of a lock l at cell (x, y), the adjacent cell (x_1, y_1) is searched for a lock. If there is none, the action is possible without further constraints for the execution time t. In the other case, the water level at the lock must be high or the action has to be executed after the disappearance of the low tide. Specifying the preconditions for the actions *GoByShip* and *TurnTide* as given above and in Section 2, respectively, fulfills the general condition for the possibility to perform a concurrent action as given in Section 3.

Consider now, for example, all specified FLUX clauses together with suitably specified facts for the example domain and the following query:

```
?-init(Z0), ida(Z0,s0,Z,S,1).
```

Our extended FLUX system then generates a plan with four steps and yields the following substitutions and linear constraints:

```
Z = [start(TA_2),at(3,2,TA_2),tide(lock3,
     high,4.0),tide(lock1,low,4.0),
     tide(lock2,high,0.0)|_]

S = do([gobyship(2)], do([gobyship(1),
    turntide(lock1,low),turntide(lock3,high)],
    do([gobyship(2),turntide(lock3,low)],
    do([gobyship(3)],s0,TA_1),2.0),4.0),TA_2)

Linear constraints: TA_1>=1.0, TA_1<2.0
                    TA_2>=5.0, TA_2<6.0
```

The above sequence of actions S constitutes a solution to our planning problem given in Fig. 1. It is not yet completely specified. Rather, the execution times of some deliberative actions are given as time intervals. The reader may also notice that some deliberative actions are planned simultaneously together with natural actions. Finally, we can apply the built-in predicate $Minimize(t)$, which tries to find a minimal solution for a constraint variable t, to the above linear constraints and get the following:

```
TA_1 = 1.0          TA_2 = 5.0
```

5 Discussion

We have presented an extension to FLUX for domains involving continuous change and where actions occur concurrently. Our method is based on the theoretic solution in the Fluent Calculus [Thielscher, 2001a].

Our extension allows for the generation of plans including both, deliberative and natural actions. If necessary, the system generates and executes concurrent actions, i.e., where two or more single actions are performed simultaneously. In order to plan efficiently, our FLUX program computes with time intervals instead of single time points using the paradigm of constraint logic programming. We have illustrated how this method can be successfully applied to example domains like the waterway scenario. Additionally, our approach can easily be applied, with only minor modifications, to more complex domains involving, for example, compound concurrent actions which produce synergic effects.

Other high-level programming languages for reasoning about action and change, like GOLOG or the robot control language, have not yet an approach to integrate both, deliberative and natural actions in a common system to generate plans. The robot control language [Shanahan and Witkowski, 2000] does not have the notion of a natural action. The systems based on GOLOG either accommodate only natural actions [Reiter, 2001] or handle only deliberative actions [Grosskreutz and Lakemeyer, 2000] in domains involving concurrency and continuous change.

The extension to domains involving uncertainty about the occurrence of a natural action has not been tackled in this paper. An approach to this problem could be the use of conditional planning. Conditional plans based on a generalized concept of plan skeletons as search heuristics have been incorporated into FLUX [Thielscher, 2001b]. To use conditional plans as a method to accommodate such domains is an important aspect of future work.

Acknowledgments

I want to thank my supervisor Michael Thielscher, Olaf Perner and all members of the FLUX Group at Technische Universität Dresden for many fruitful discussions about this work. Three anonymous reviewers provided helpful comments on this article for which I am grateful.

References

[Frühwirth, 1998] Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, 1998.

[Grosskreutz and Lakemeyer, 2000] Henrik Grosskreutz and Gerhard Lakemeyer. cc-Golog: Towards more realistic logic-based robot control. In H. Kautz and B. Porter, editors, *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 476–482, Austin, TX, July 2000.

[Grosskreutz, 2002] Henrik Grosskreutz. *Towards more realistic logic-based robot controllers in the Golog framework*. PhD thesis, RWTH Aachen, Germany, February 2002.

[Herrmann and Thielscher, 1996] Christoph S. Herrmann and Michael Thielscher. Reasoning about continuous processes. In B. Clancey and D. Weld, editors, *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 639–644, Portland, OR, August 1996. MIT Press.

[Levesque *et al.*, 1997] 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.

[Reiter, 1996] Ray Reiter. Natural actions, concurrency and continuous time in the situation calculus. In L. C. Aiello, J. Doyle, and S. Shapiro, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 2–13, Cambridge, MA, November 1996. Morgan Kaufmann.

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

[Russell and Norvig, 1995] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.

[Shanahan and Witkowski, 2000] 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.

[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, 2001a] Michael Thielscher. The concurrent, continuous Fluent Calculus. *Studia Logica*, 67(1), 2001.

[Thielscher, 2001b] Michael Thielscher. Inferring implicit state knowledge and plans with sensing actions. In F. Baader, G. Brewka, and T. Eiter, editors, *Proceedings of the German Annual Conference on Artificial Intelligence (KI)*, volume 2174 of *LNAI*, pages 366–380, Vienna, Austria, September 2001. Springer.

[Thielscher, 2002] Michael Thielscher. Programming of reasoning and planning agents with FLUX. In D. Fensel, D. McGuinness, and M.-A. Williams, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, Toulouse, France, April 2002. Morgan Kaufmann.