Handling Implication and Universal Quantification Constraints in FLUX

Michael Thielscher

Dresden University of Technology, 01062 Dresden, Germany,
mit@inf.tu-dresden.de

Abstract. FLUX is a CLP-approach for programming agents that reason about actions under incomplete state knowledge. FLUX is based on the solution to the fundamental frame problem in the fluent calculus. The core is a set of Constraint Handling Rules for the constraints that are used to encode state knowledge. In order to allow for efficient constraint solving, the original expressiveness of state representations in FLUX has been carefully restricted. In this paper, we enhance the expressiveness by adding both implication and universal quantification constraints. We do so without losing the computational merits of FLUX. We present a set of Constraint Handling Rules for these new constraints and prove their correctness against the fluent calculus.

1 Introduction

Reasoning about actions is one of the central issues in Artificial Intelligence [1]. The classical formalism for representing knowledge of actions and their effects is the situation calculus [2]. A fundamental challenge in this context is raised by the classical *frame problem*, which means to find an efficient way of inferring what changes and what does not change as a result of an action [3]. Simple solutions to this problem, such as STRIPS [4], apply only to the special case of complete knowledge. Solutions to the frame problem for the general case of incomplete states have recently evolved into logic programming approaches, e.g., [5, 6] based on the situation calculus and the event calculus, respectively. These allow to program agents who use an internal world model for decision making and who reason about their actions in order to keep this model up to date as they move along. However, both of the aforementioned approaches lack an explicit notion of states. Knowledge of the current state is represented indirectly via the initial conditions and the actions which the agent has performed up to now. As a consequence, the entire history of actions is needed when evaluating a condition in an agent program [7]. This problem has been overcome in the language FLUX [8], where an incomplete state is explicitly represented by a list (of atomic state components) along with constraints. Actions are specified in terms of how they affect a state, which allows agents to *progress* the state whenever they execute an action and, hence, to directly evaluate conditions in agent programs against the current world model. This is necessary if we aim at

programs that scale up to non-trivial domains in which agents need to perform long sequences of actions [8].

The core of FLUX is a set of Constraint Handling Rules [9] for the constraints that are used to describe (incomplete) states. The semantics of FLUX, and in particular of the constraint solver, is given by the fluent calculus—an action formalism which can be viewed as an extension of the classical situation calculus by the basic notion of a state [10]. In order to obtain an efficient constraint solver, the expressiveness of FLUX as presented in [11] has been carefully restricted: States are composed of finitely many atomic state components, so-called fluents, accompanied by constraints for (possibly universally quantified) negated single fluents and for disjunctions of fluents. These restrictions allow for efficient constraint solving based on *unit resolution*, so that evaluating a new constraint is linear in the size of the constraint store. This is necessary if we aim at a system which scales up gracefully to domains with a large state space [8].

However, the restrictions imposed on the state representation in FLUX are too weak to cover two important phenomena: Firstly, if an action has conditional effects and the condition is unknown at the time when the action is performed, then a complete encoding of what is known after the action requires *implication constraints*. Secondly, a state in which infinitely many instances of a fluent hold cannot be expressed in a finite list. In this paper, we will overcome these restrictions by extending FLUX to both implication constraints and constraints for universal quantification. We will develop a set of Constraint Handling Rules which again is carefully designed so as to retain the linear complexity of constraint solving in FLUX. Correctness of these rules is formally proved against the semantics of the fluent calculus.

2 Reasoning About Actions with FLUX

2.1 The Fluent Calculus

The fluent calculus [10] is a predicate logic language which extends the classical situation calculus [2]. The latter builds on the basic notions of actions (i.e., primitive behaviors of an agent), situations (i.e., sequences of actions), and fluents (i.e., atomic state components). To this, the fluent calculus adds the notion of states (i.e., collections of fluents).

Definition 1. *A* fluent calculus signature *consists of a finite set* $\mathcal{A}$ *of functions into sort* ACTION *and a finite set* $\mathcal{F}$ *of functions into sort* FLUENT*. Sort* FLUENT *is a sub-sort of* STATE*. Furthermore,*

$$S_0 : \text{SIT} \qquad \emptyset : \text{STATE}$$
$$Do : \text{ACTION} \times \text{SIT} \mapsto \text{SIT} \qquad \circ : \text{STATE} \times \text{STATE} \mapsto \text{STATE}$$
$$State : \text{SIT} \mapsto \text{STATE}$$

Inherited from the situation calculus, constant S_0 denotes the initial situation and $Do(a, s)$ denotes the situation after performing action a in situation s.

The term $State(s)$ denotes the state in situation s. Constant $\emptyset$ denotes the empty state. Finally, every fluent is a (singleton) state, and if z_1 and z_2 are states then so is $z_1 \circ z_2$.[1] A fluent f is defined to *hold* in a state z just in case z can be decomposed into two states one of which is the singleton f. For notational convenience, the following macro is used as an abbreviation for the corresponding equational formula:

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

The foundational axioms of the fluent calculus stipulate that function "$\circ$" shares essential properties with the union operation for sets:

Definition 2. *The* foundational axioms $\mathcal{F}_{state}$ of the fluent calculus *include:*

1. *Associativity and commutativity,*

$$(z_1 \circ z_2) \circ z_3 = z_1 \circ (z_2 \circ z_3) \qquad z_1 \circ z_2 = z_2 \circ z_1 \tag{2}$$

2. *Empty state axiom,*

$$\neg Holds(f, \emptyset) \tag{3}$$

3. *Irreducibility and decomposition,*

$$Holds(f_1, f) \supset f_1 = f \tag{4}$$

$$Holds(f, z_1 \circ z_2) \supset Holds(f, z_1) \vee Holds(f, z_2) \tag{5}$$

Associativity allows us to omit parentheses in nested applications of "$\circ$".[2]

Based on the notion of a state, the frame problem is solved in the fluent calculus by axioms which define the effects of an action $A(\overline{x})$ in situation s in terms of how $State(s)$ is updated to the successor $State(Do(A(\overline{x}), s))$. To this end, two functions "$-$" and "$+$" are used which denote, respectively, removal and addition of fluents to states. They have a purely axiomatic characterization: Let ϑ^-, ϑ^+ be finitely many FLUENT terms connected by "$\circ$", then

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

The crucial item is the second one, which inductively defines removal of fluents f using a case distinction: Either $z_1 - f$ equals z_1 (which applies in case $\neg Holds(f, z_1)$), or $(z_1 - f) \circ f$ equals z_1 (which applies in case $Holds(f, z_1)$).

[1] Throughout the paper, free variables in formulas are assumed to be universally quantified. Variables of sorts fluent, state, action, and situation shall be denoted by the letters f, z, a, and s, respectively. The function "$\circ$" is written in infix notation. A (possibly empty) sequence of variables is denoted by $\overline{x}$. We use the standard logical connectives $\neg$, $\wedge$, $\vee$, $\supset$ (implication), and $\equiv$ (equivalence).

[2] The full axiomatic foundation of the fluent calculus contains two further axioms [8], which, however, are not needed in the present paper. By the foundational axioms, states are essentially *flat* sets (of fluents), i.e., which do not contain sets as elements.

2.2 FLUX

The basic data structure in FLUX is that of an incomplete state, which represents what an agent knows of the state of its environment in a specific situation. An incomplete state is encoded as a fluent list which carries a tail variable; e.g.,

```
Z0 = [solution(a),solution(b),litmus(p1) | Z]
```

shall encode the knowledge that there are at least the two (chemical) solutions `a,b` and the litmus paper `p1`. In addition to knowledge of fluents that hold in a state, FLUX—as presented in [11]—allows to encode negative and disjunctive state information as *constraints* on the tail variable. For example,

```
not_holds(red(p1),Z), or_holds([acidic(a),acidic(b)],Z),
not_holds_all(solution(_),Z)
```

encodes the knowledge that the litmus paper is not red, that one of the two solutions is acidic, and that there are no other solutions available. The semantics of a FLUX state specification is given by an equational fluent calculus formula, here $(\exists \mathtt{Z})\,(\mathtt{Z0} = \mathtt{solution(a)} \circ \mathtt{solution(b)} \circ \mathtt{litmus(p1)} \circ \mathtt{Z})$, along with formulas corresponding to the semantics of the constraints:

constraint	semantics
`not_holds(F,Z)`	$\neg \mathit{Holds}(\mathtt{F},\mathtt{Z})$
`not_holds_all(F,Z)`	$(\forall \overline{x})\, \neg \mathit{Holds}(\mathtt{F},\mathtt{Z})$, $\overline{x}$ variables in `F`
`or_holds([F1,...,Fk],Z)`	$\bigvee_{i=1}^{k} \mathit{Holds}(\mathtt{F}_i,\mathtt{Z})$

(6)

In [11], a set of *Constraint Handling Rules* (CHRs) [9] has been developed for the FLUX constraints. CHRs are of the form

```
H1,...,Hm <=> G1,...,Gk | B1,...,Bn.
```

where the *head* $H_1, \ldots, H_m$ are constraints ($m \geq 1$); the *guard* $G_1, \ldots, G_k$ are Prolog literals ($k \geq 0$); and the *body* $B_1, \ldots, B_n$ are constraints ($n \geq 0$). An empty guard is omitted; the empty body is denoted by `true`. The declarative interpretation of a CHR is given by the formula

$$(\forall \overline{x})\,(\,G_1 \wedge \ldots \wedge G_k \;\supset\; [H_1 \wedge \ldots \wedge H_m \;\equiv\; (\exists \overline{y})\,(B_1 \wedge \ldots \wedge B_n)]\,) \qquad (7)$$

where $\overline{x}$ are the variables in both guard and head and $\overline{y}$ are the variables which additionally occur in the body. The procedural interpretation of a CHR is given by a transition in a constraint store: If the head can be matched against elements of the constraint store and the guard can be derived, then the constraints of the head are replaced by the constraints of the body.

The two main computation mechanisms for constraint solving in FLUX are propagation and unit resolution. Figure 1 depicts two out of the total of 18 rules, each of which has been verified against the foundational axioms of the fluent calculus. The first example CHR *propagates* a negation constraint through a list

```
not_holds(F,[F1|Z])              <=> neq(F,F1), not_holds(F,Z).

not_holds(F,Z) \ or_holds(V,Z) <=> member(G,V,W), F==G | or_holds(W,Z).
```

Fig. 1. Two example CHRs for the FLUX constraints. The auxiliary predicate `neq(F,G)` defines the inequality of `F` and `G` by a finite domain constraint [12] among the arguments of the two fluents. Predicate `member(X,Y,Z)` is true if `X` occurs in list `Y` and if `Z` is `Y` without `X`. Notation `H1 \ H2 <=> G | B` abbreviates `H1,H2 <=> G | H1,B`.

of fluents. Suppose, say, that tail variable `Z` of our example state were substituted by `[red(X)|Z1]`, then `not_holds(red(p1),[red(X)|Z1])` reduces to finite domain constraint `p1` $\neq$ `X` along with `not_holds(red(p1),Z1)`. The second CHR *resolves* a disjunction in the presence of a negation constraint. Suppose, for instance, we add the constraint `not_holds(acidic(b),Z)` to our example state, then `or_holds([acidic(a),acidic(b)],Z)` reduces to the singleton `or_holds([acidic(a)],Z)`.[3]

Actions are specified in FLUX by *state update axioms*. Two examples are,

```
state_update(Z1,get_litmus_paper(P),Z2,[]) :-
   update(Z1,[litmus(P)],[],Z2).

state_update(Z1,sense_paper(P),Z2,[Red]) :-
   ( Red=true,  holds(red(P),Z1) ;
     Red=false, not_holds(red(P),Z1) ), Z2 = Z1.
```

The semantics of the auxiliary predicate `update(Z1,P,N,Z2)` is given by the fluent calculus update equation `Z2` $=$ `(Z1` $-$ `N)` $+$ `P`. The last argument of `state_update` being empty indicates that action `get_litmus_paper` does not involve sensing. The action of sensing the status of a litmus paper, on the other hand, does not cause any physical effect, hence the state equality `Z2=Z1`. Recall, for instance, the initial state from above, then

```
?- state_update(Z0,get_litmus_paper(p2),Z1,[]),
   state_update(Z1,sense_paper(p2),Z2,[true]).

Z2 = [litmus(p2),solution(a),solution(b),litmus(p1),red(p2)|Z]
Constraints:
not_holds(red(p1),Z)
or_holds([acidic(a),acidic(b)],Z)
not_holds_all(solution(_),Z)
...
```

[3] It is worth mentioning that the guard in the second CHR of Figure 1 cannot be simplified to `member(F,V,W)` because constraints may contain variables. For example, $(\exists$`X`$)\neg$*Holds*(`acidic(X)`, `Z`) and *Holds*(`acidic(a)`, `Z`) $\vee$ *Holds*(`red(p)`, `Z`) do *not* imply *Holds*(`red(p)`, `Z`).

The constraint solver and the definition of state update provide the basis for agent programs in which an internal model of the environment is used for decision making and where this model is updated through the execution of actions.

3 Handling Implication Constraints

3.1 Why Implication Constraints?

The constraints and CHRs for FLUX presented in [11] have been carefully designed to allow for efficient constraint solving. This has been achieved by restricting disjunctions to positive atoms only, which allows to apply unit resolution. As a consequence, the computational effort of evaluating a new constraint is *linear* in the size of the constraint store. A disadvantage, however, is that the restricted expressiveness is too weak for solving problems which involve actions with conditional effects and where the condition is unknown at execution time. As an example, consider the action `dip(P,X)` of dipping litmus paper `P` into solution `X`, of which it is not known whether it is acidic or not. With the given restricted expressiveness, FLUX requires to specify this action by the following state update axiom:

```
state_update(Z1,dip(P,X),Z2,[]) :-
   \+ not_holds(acidic(X),Z1) -> update(Z1,[red(P)],[],Z2) ;
   \+ holds(acidic(X),Z1)     -> update(Z1,[],[red(P)],Z2) ;
   cancel(red(P),Z1,Z2).
```

That is, if state `Z1` contains sufficient information to conclude that `acidic(X)` cannot be false, then the agent can update its state knowledge by $+$`red(P)`. Conversely, if the agent knows that `acidic(X)` cannot hold, then it updates its state by $-$`red(P)`. If, however, the status of `acidic(X)` is not known in `Z1`, then `cancel(F,Z1,Z2)` means that state `Z2` is as state `Z1` except that any constraint on fluent `F` is cancelled. With this, the essence of the Litmus Test cannot be expressed, because testing a solution and afterwards checking the color of the paper does not enable the agent to infer the status of the solution. Recall, for example, the initial state `Z0` in Section 2, then

```
?- state_update(Z0,dip(p1,a),Z1,[]),
   state_update(Z1,sense_paper(p1),Z2,[true]).

Z2 = [solution(a),solution(b),litmus(p1),red(p1) | _]
```

Although fluent `red(p1)` is known to be true now, it does not follow that solution `a` is acidic. The reason is that initially the status of the solution is unknown, and hence the only inferred effect of applying the state update axiom for `dip(p1,a)` is that paper `p1` is no longer known not to be red. The restricted expressiveness of FLUX does not allow to encode the effect of this action in such a way that the logical dependency between redness of the paper and acidity of the solution is captured in the successor state. This is a general limitation of the existing FLUX when it comes to actions with conditional effects.

```
if_then_holds(F,G1,Z) <=> if_then_or_holds(F,[G1],Z).                    %1

if_then_or_holds(F,[],Z) <=> not_holds(F,Z).                             %2
if_then_or_holds(_,_,[]) <=> true.                                       %3
if_then_or_holds(_,V,Z)  <=> member(eq(X,Y),V),                          %4
                             or_neq(exists,X,Y,D), \+ call(D) | true.
if_then_or_holds(F,V,Z)  <=> member(eq(X,Y),V,W),                        %5
                             \+ (and_eq(X,Y,D), call(D))
                             | if_then_or_holds(F,W,Z).
```

Fig. 2. Simplification CHRs for implication constraints. The auxiliary predicates `or_neq(exists,`$\overline{\mathtt{X}}$`,`$\overline{\mathtt{Y}}$`,D)` and `and_eq(`$\overline{\mathtt{X}}$`,`$\overline{\mathtt{Y}}$`,D)` define `D` to be a finite domain constraint that encodes, respectively, inequality $\overline{\mathtt{X}} \neq \overline{\mathtt{Y}}$ and equality $\overline{\mathtt{X}} = \overline{\mathtt{Y}}$ (see [11] for details).

3.2 Handling Implication Constraints

In this section, we will extend FLUX by a constraint for disjunctions containing a negative literal. A special case of this will be an auxiliary constraint for implicational dependencies between two fluents:

constraint	semantics
`if_then_or_holds(F,[`$\mathtt{G}_1$`,...,`$\mathtt{G}_k$`],Z)`	$\mathit{Holds}(\mathtt{F},\mathtt{Z}) \supset \bigvee_{i=1}^{k} \mathit{Holds}(\mathtt{G}_i,\mathtt{Z})$
`if_then_holds(F,G,Z)`	$\mathit{Holds}(\mathtt{F},\mathtt{Z}) \supset \mathit{Holds}(\mathtt{G},\mathtt{Z})$

(8)

We incorporate the new constraint into FLUX by adding a set of Constraint Handling Rules. Each of the new CHRs, too, constitutes either a simplification, propagation, or unit resolution step, so that evaluating a new constraint is still linear in the size of the constraint store. The first part of the new set of CHRs is depicted in Figure 2. The solver employs an extended notion of an implication constraint where the disjunctive part may include atoms of the form `eq(X,Y)` and `neq(X,Y)` with `X` and `Y` being lists of equal length. The meaning of this general constraint `if_then_or_holds(F,[`$\mathtt{G}_1$`,...,`$\mathtt{G}_k$`],Z)` is

$$\mathit{Holds}(\mathtt{F},\mathtt{Z}) \supset \bigvee_{i=1}^{k} \begin{cases} \mathit{Holds}(\mathtt{G}_i,\mathtt{Z}) & \text{if } \mathtt{G}_i \text{ is a fluent} \\ \mathtt{X} = \mathtt{Y} & \text{if } \mathtt{G}_i \text{ is } \mathtt{eq(X,Y)} \\ \mathtt{X} \neq \mathtt{Y} & \text{if } \mathtt{G}_i \text{ is } \mathtt{neq(X,Y)} \end{cases} \tag{9}$$

This generalization is needed for propagating an implication constraint containing fluents with variable arguments, as will be shown below.

To begin with, CHR 1 defines `if_then_holds` in terms of the general implication constraint. CHRs 2–5 are simplification rules. Consider, say, the implication constraint `if_then_or_holds(acidic(a),[eq([p1],[p2])],Z)`, which has a singleton, equational disjunction. To this, CHR 5 applies since `p1` = `p2` fails. The application of the rule yields `if_then_or_holds(acidic(a),[],Z)`, which by CHR 2 gets reduced to `not_holds(acidic(a),Z)`.

The four CHRs in Figure 3 encode unit resolution steps. Specifically, CHRs 6 and 7 solve an implication whose antecedent is implied by a negation constraint.

```
not_holds(F,Z)     \ if_then_or_holds(G,_,Z) <=> F==G | true.          %6
not_holds_all(F,Z) \ if_then_or_holds(G,_,Z) <=> inst(G,F) | true.     %7

not_holds(F,Z)     \ if_then_or_holds(C,V,Z) <=>                       %8
                   member(G,V,W), F==G      | if_then_or_holds(C,W,Z).
not_holds_all(F,Z) \ if_then_or_holds(C,V,Z) <=>                       %9
                   member(G,V,W), inst(G,F) | if_then_or_holds(C,W,Z).
```

Fig. 3. Unit resolution CHRs for implication constraints. Predicate `inst(G,F)` means that fluent term `G` is an instance of `F`.

CHRs 8 and 9 resolve an implication containing a disjunct that unifies with a negation constraint. For example, in the presence of `not_holds(red(p1),Z)`, implication `if_then_or_holds(acidic(a),[red(p1)],Z)` reduces, by CHR 8, to `if_then_or_holds(acidic(a),[],Z)`.

Crucial for constraint solving in FLUX is the propagation through a list of fluents. It is needed whenever the variable state argument is substituted by an (incomplete) list. This happens when an agent performs actions or acquires new information about the world. In general, propagating a constraint through a list of fluents requires us to evaluate these fluents against those that occur in the constraint. CHRs 10–12 in Figure 4 model the propagation of our new implication constraint. Specifically, the first case in CHR 10 applies if the antecedent of the implication is true in the given state. The other two cases employ the 4-ary constraint `if_then_or_holds(C,[`$\mathtt{G}_1$`,...,`$\mathtt{G}_k$`],[`$\mathtt{H}_1$`,...,`$\mathtt{H}_l$`],[F|Z])`, whose intended semantics is

$$\begin{aligned} \mathit{Holds}(\mathtt{C},\mathtt{Z}) \supset \quad & \bigvee_{i=1}^{k} \begin{cases} \mathit{Holds}(\mathtt{G}_i, \mathtt{F} \circ \mathtt{Z}) & \text{if } \mathtt{G}_i \text{ is a fluent} \\ \mathtt{X} = \mathtt{Y} & \text{if } \mathtt{G}_i \text{ is } \mathtt{eq(X,Y)} \\ \mathtt{X} \neq \mathtt{Y} & \text{if } \mathtt{G}_i \text{ is } \mathtt{neq(X,Y)} \end{cases} \\ \vee \bigvee_{j=1}^{l} & \begin{cases} \mathit{Holds}(\mathtt{H}_j, \mathtt{Z}) & \text{if } \mathtt{H}_j \text{ is a fluent} \\ \mathtt{X} = \mathtt{Y} & \text{if } \mathtt{H}_j \text{ is } \mathtt{eq(X,Y)} \\ \mathtt{X} \neq \mathtt{Y} & \text{if } \mathtt{H}_j \text{ is } \mathtt{neq(X,Y)} \end{cases} \end{aligned} \tag{10}$$

Hence, the $\mathtt{G}_i$'s are the fluents that have not yet been evaluated against the head `F` of the state list, while the $\mathtt{H}_j$'s are those fluents that have been evaluated. For example, $\mathit{Holds}(G, F(x) \circ z) \supset \mathit{Holds}(F(a), F(x) \circ z) \vee \mathit{Holds}(F(b), F(x) \circ z)$ is equivalent to $\mathit{Holds}(G, z) \supset x = a \vee x = b \vee \mathit{Holds}(F(a), z) \vee \mathit{Holds}(F(b), z)$ according to the foundational axioms of decomposition and irreducibility and given the unique-name axiom $F(x) = F(y) \supset x = y$. Correspondingly,

```
    if(g,[f(a),f(b)],[f(X)|Z])
<=> if(g,[f(a),f(b)],[],[f(X)|Z])
<=> if(g,[f(b)],[eq([a],[X]),f(a)],[f(X)|Z])
<=> if(g,[],[eq([b],[X]),f(b),eq([a],[X]),f(a)],[f(X)|Z])
<=> if(g,[eq([b],[X]),f(b),eq([a],[X]),f(a)],Z)
```

(where `if` abbreviates `if_then_or_holds`). The third case in CHR 10 applies when the antecedent of an implication constraint unifies with the head of a state.

```
if_then_or_holds(C,V,[F|Z]) <=>                                    %10
   C==F -> or_holds(V,[F|Z]) ;
   C\=F -> if_then_or_holds(C,V,[],[F|Z]) ;
   C=..[_|ArgX], F=..[_|ArgY], or_holds([neq(ArgX,ArgY)|V],[F|Z]),
                               if_then_or_holds(C,V,[],[F|Z]).

if_then_or_holds(C,[G|V],W,[F|Z]) <=>                              %11
   G==F -> true ;
   G\=F -> if_then_or_holds(C,V,[G|W],[F|Z]) ;
   G=..[_|ArgX], F=..[_|ArgY],
      if_then_or_holds(C,V,[eq(ArgX,ArgY),G|W],[F|Z]).

if_then_or_holds(C,[],W,[_|Z]) <=> if_then_or_holds(C,W,Z).        %12
```

Fig. 4. Propagation CHRs for implication constraints.

For example, $\mathit{Holds}(F(a), F(x) \circ z) \supset \mathit{Holds}(G, F(x) \circ z)$ is equivalent to

$$[a \neq x \vee \mathit{Holds}(G, z)] \ \wedge \ [\mathit{Holds}(F(a), z) \supset \mathit{Holds}(G, z)]$$

Correspondingly, `if_then_or_holds(f(a),[g],[f(X)|Z])` is reduced to

```
or_holds([neq([a],[X]),g],Z)
if_then_or_holds(f(a),[g],Z)
```

3.3 Correctness

In the following we prove the formal correctness of the new Constraint Handling Rules against the underlying theory of the fluent calculus. The proof is based on the declarative interpretation of CHRs (see (7)) and the semantics of the constraints in terms of the fluent calculus, given by (6) and (8)–(10).

Theorem 1. *CHRs 1–12 are correct under the foundational axioms $\mathcal{F}_{state}$ and the assumption of uniqueness-of-names (UNA) for all fluents.*

Proof. The logical reading of the rules are given by these formulas:

1. $[\mathit{Holds}(f, z) \supset G_1] \equiv [\mathit{Holds}(f, z) \supset \bigvee_{i=1}^{1} G_i]$;
2. $[\mathit{Holds}(f, z) \supset \bigvee_{i=1}^{0} G_i] \equiv \neg \mathit{Holds}(f, z)$;
3. $[\mathit{Holds}(f, \emptyset) \supset \bigvee_{i=1}^{k} G_i] \equiv \top$;
4. If $\neg \overline{x} \neq \overline{y}$ then $[\mathit{Holds}(f, z) \supset \overline{x} = \overline{y} \vee \bigvee_{i=1}^{k} G_i] \equiv \top$;
5. If $\neg \overline{x} = \overline{y}$ then $[\mathit{Holds}(f, z) \supset \overline{x} = \overline{y} \vee \bigvee_{i=1}^{k} G_i] \equiv [\mathit{Holds}(f, z) \supset \bigvee_{i=1}^{k} G_i]$;
6. If $\neg \mathit{Holds}(f, z)$ then $[(\mathit{Holds}(f, z) \supset \bigvee_{i=1}^{k} G_i] \equiv \top$;
7. If $(\forall \overline{x}) \neg \mathit{Holds}(f, z)$ and g is an instance of f then $[(\mathit{Holds}(g, z) \supset \bigvee_{i=1}^{k} G_i] \equiv \top$;

8. If $\neg Holds(f_1, z)$ then
   $[Holds(f, z) \supset Holds(f_1, z) \vee \bigvee_{i=1}^{k} G_i] \equiv [Holds(f, z) \supset \bigvee_{i=1}^{k} G_i]$;
9. If $(\forall \overline{x})\, \neg Holds(f_1, z)$ and g is an instance of f_1 then
   $[Holds(f, z) \supset Holds(g, z) \vee \bigvee_{i=1}^{k} G_i] \equiv [Holds(f, z) \supset \bigvee_{i=1}^{k} G_i]$;
10. The following corresponds to the three cases in CHR 10:
    (a) $[Holds(f, f \circ z) \supset \bigvee_{i=1}^{k} G_i] \equiv \bigvee_{i=1}^{k} G_i$;
    (b) If $f \neq f_1$ then $[Holds(f, f_1 \circ z) \supset \bigvee_{i=1}^{k} G_i] \equiv [Holds(f, z) \supset \bigvee_{i=1}^{k} G_i]$;
    (c) $[Holds(F(\overline{x}), F(\overline{y}) \circ z) \supset \bigvee_{i=1}^{k} G_i] \equiv$
    $[(\overline{x} \neq \overline{y} \vee \bigvee_{i=1}^{k} G_i) \wedge (Holds(F(\overline{x}), z) \supset \bigvee_{i=1}^{k} G_i)]$;
11. The following corresponds to the three cases in CHR 11:
    (a) $[Holds(f, z) \supset Holds(f_1, f_1 \circ z) \vee \bigvee_{i=1}^{k} G_i \vee \bigvee_{j=1}^{l} H_j] \equiv \top$;
    (b) If $g \neq f_1$ then $[Holds(f, z) \supset (Holds(g, f_1 \circ z) \vee \bigvee_{i=1}^{k} G_i) \vee \bigvee_{j=1}^{l} H_j] \equiv$
    $[Holds(f, z) \supset \bigvee_{i=1}^{k} G_i \vee (Holds(g, z) \vee \bigvee_{j=1}^{l} H_j)]$;
    (c) $[Holds(f, z) \supset (Holds(F(\overline{x}), F(\overline{y}) \circ z) \vee \bigvee_{i=1}^{k} G_i) \vee \bigvee_{j=1}^{l} H_j] \equiv$
    $[Holds(f, z) \supset \bigvee_{i=1}^{k} G_i \vee (\overline{x} = \overline{y} \vee Holds(F(\overline{x}), z) \vee \bigvee_{j=1}^{l} H_j)]$;
12. $[Holds(c, z) \supset \bigvee_{i=1}^{0} G_i \vee \bigvee_{j=1}^{l} H_j] \equiv [Holds(c, z) \supset \bigvee_{j=1}^{l} H_j]$.

Let $\mathcal{F}$ be the underlying fluent functions, then $\mathcal{F}_{state} \cup UNA[\mathcal{F}]$ [4] entails each of the formulas above: Claims 1, 2, 4–9, and 12 are tautologies. Claim 3 follows by the foundational axiom on the empty state, (3). Claims 10(a) and 11(a) follow by the definition of *Holds*, (1). Claim 10(b) follows by the foundational axioms of decomposition, (5), and irreducibility, (4). Regarding 10(c), by decomposition and irreducibility $Holds(F(\overline{x}), F(\overline{y}) \circ z) \supset \bigvee_{i=1}^{k} G_i$ is equivalent to $[\overline{x} \neq \overline{y} \wedge \neg Holds(F(\overline{x}), z)] \vee \bigvee_{i=1}^{k} G_i$. This, in turn, is equivalent to the conjunction $[\overline{x} \neq \overline{y} \vee \bigvee_{i=1}^{k} G_i] \wedge [\neg Holds(F(\overline{x}), z) \vee \bigvee_{i=1}^{k} G_i]$, which implies the claim. Regarding 11(b), if $g \neq f_1$ then by decomposition and irreducibility $Holds(g, f_1 \circ z)$ is equivalent to $Holds(g, z)$, which implies the claim. Regarding 11(c), finally, by decomposition, $UNA[\mathcal{F}]$, and irreducibility $Holds(F(\overline{x}), F(\overline{y}) \circ z)$ is equivalent to $\overline{x} = \overline{y} \vee Holds(F(\overline{x}), z)$, which implies the claim.

While the extended set of CHRs is provably correct, a limitation is inherited from the original solver: Agents are not able to draw all conclusions that follow logically from a state specification if the underlying arithmetic solver trades full inference capabilities for efficiency; we refer to [11] for details.

3.4 Using the Implication Constraint

The new constraint allows to encode the logical dependencies that result from applying an action with conditional effects in situations where it is unknown whether the condition holds. The crucial action in the Litmus Test example can thus be encoded in such a way as to retain the dependency of cause and effect in case the status of the chemical solution in question is unknown. By the following

[4] $UNA[f_1, \ldots, f_n] \stackrel{\text{def}}{=} \bigwedge_{i<j} f_i(\overline{x}) \neq f_j(\overline{y}) \wedge \bigwedge_i [f_i(\overline{x}) = f_i(\overline{y}) \supset \overline{x} = \overline{y}]$

update specification for action `dip(P,X)`, first any knowledge of fluent `red(P)` is cancelled in state `Z1`, since this fluent may be affected by the action, and then the effect of the action is that `red(P)` is true in case `acidic(X)` holds and false in case `acidic(X)` does not hold:

```
state_update(Z1,dip(P,X),Z2,[]) :-
   cancel(red(P),Z1,Z2),
   if_then_holds(acidic(X),red(P),Z2),
   if_then_holds(red(P),acidic(X),Z2).
```

Recall, e.g., the scenario discussed in Section 3.1, where it is given that either of two solutions is acidic and where the litmus paper is dipped into one of them:[5]

```
init(Z0) :- Z0 = [solution(a),solution(b),litmus(p1) | Z],
            not_holds(red(p1),Z),
            or_holds([acidic(a),acidic(b)],Z),
            not_holds_all(solution(_),Z),
            duplicate_free(Z0).

?- init(Z0), state_update(Z0,dip(p1,a),Z1,[]).

Z1 = [solution(a),solution(b),litmus(p1) | Z]
Constraints:
or_holds([acidic(a),acidic(b)],Z),
if_then_or_holds(acidic(a),[red(p1)],Z)
if_then_or_holds(red(p1),[acidic(a)],Z)
...
```

Suppose, now, that the subsequent test of the litmus paper reveals that it turned red. The update axiom for action `sense_paper` (cf. Section 2) effects a substitution of state variable `Z` by `[red(p1)|_]`. The extended FLUX constraint solver is then able to reduce the two implication constraints from above:

```
?- init(Z0),
   state_update(Z0,dip(p1,a),Z1,[]),
   state_update(Z1,sense_paper(p1),Z2,[true]).

Z2 = [solution(a),solution(b),litmus(p1),red(p1),acidic(a) | _]
```

Thus it follows that solution `a` must be acidic. Conversely, suppose that the test of the litmus paper reveals that it did not turn red. In this case the update axiom for action `sense_paper` adds the constraint `not_holds(red(p1),Z)`. Again the extended FLUX constraint solver is able to reduce the two implication constraints from above. Furthermore, the disjunction is solved:

```
?- init(Z0),
   state_update(Z0,dip(p1,a),Z1,[]),
```

[5] Auxiliary constraint `duplicate_free(Z)` ensures that no fluent occurs twice in `Z`.

```
state_update(Z1,sense_paper(p1),Z2,[true]).

Z2 = [solution(a),solution(b),litmus(p1),acidic(b) | _]
```

Thus it follows that solution `b` must be acidic.

3.5 Why not General Disjunctions?

Read as disjunction, the new implication constraint allows to encode disjunctive clauses which include one negative literal (namely, the antecedent). This immediately raises the question of introducing general disjunctions with two or more negative literals. This, however, can only be done either by defining a highly incomplete constraint solver, or by worsening the complexity to exponential. To see why, consider a disjunctive statement with two negations, like

$$\neg \mathit{Holds}(F(a), F(x_1) \circ \ldots \circ F(x_n) \circ z) \vee \neg \mathit{Holds}(F(b), F(x_1) \circ \ldots \circ F(x_n) \circ z)$$

Using propagation through the first element of the state, $F(x_1)$, this formula is equivalent to the conjunction of these four constraints:

$$\begin{array}{l} a \neq x_1 \vee b \neq x_1 \\ a \neq x_1 \vee \neg \mathit{Holds}(F(b), F(x_2) \circ \ldots \circ F(x_n) \circ z) \\ b \neq x_1 \vee \neg \mathit{Holds}(F(a), F(x_2) \circ \ldots \circ F(x_n) \circ z) \\ \neg \mathit{Holds}(F(a), F(x_2) \circ \ldots \circ F(x_n) \circ z) \vee \neg \mathit{Holds}(F(b), F(x_2) \circ \ldots \circ F(x_n) \circ z) \end{array}$$

Assuming uniqueness-of-names, the first disjunction is trivially true, but the other three constraints need to be propagated further. Propagation through the remaining $n-1$ fluents will result in $2^n - 1$ constraints. Hence, either constraint solving becomes exponential, or the solver avoids propagating a general disjunction. The latter, however, would render the solver powerless.

4 Handling Universal Quantification Constraints

A second limitation of the constraint solver presented in [11] is that it lacks universally quantified positive information. In this section, we will extend FLUX further by a constraint `all_holds(F,C,Z)` with the intended semantics

$$(\forall \overline{x})\,(\mathtt{C}[\overline{x}] \supset \mathit{Holds}(\mathtt{F}, \mathtt{Z})\,) \tag{11}$$

where $\overline{x}$ are the variables occurring in fluent `F` and where $\mathtt{C}[\overline{x}]$ is a finite domain constraint over $\overline{x}$. The special case of unconstrained universal quantification shall be encoded by `all_holds(F,Z)`, meaning $(\forall \overline{x})\,\mathit{Holds}(\mathtt{F}, \mathtt{Z})$. In a similar way, we generalize universally quantified negation to `all_not_holds(F,C,Z)` with the intended semantics

$$(\forall \overline{x})\,(\mathtt{C}[\overline{x}] \supset \neg \mathit{Holds}(\mathtt{F}, \mathtt{Z})\,) \tag{12}$$

where $\overline{x}$ are the variables occurring in fluent `F` and where $\mathtt{C}[\overline{x}]$ is a finite domain constraint over $\overline{x}$. In the following, we present a set of CHRs for the new universal quantification constraints. Lack of space, however, will prevent us from discussing this extension as thoroughly as in the preceding section.

```
all_holds(F,Z) <=> all_holds(F,(0#=0),Z).

all_holds(F,C,Z), not_holds(G,Z) ==>                                  %1
   copy_fluent(F,C,F1,C1) | F1=G, call(#\+C1).
all_holds(F,C,Z), all_not_holds(G,D,Z) <=>                            %2
   copy_fluent(F,C,F1,C1), copy_fluent(G,D,G1,D1),
   F1=G1, call(C1#/\D1) | false.
all_holds(F,C,Z) \ or_holds(V,Z) <=>                                  %3
   member(G,V), copy_fluent(F,C,F1,C1), F1=G, \+ call(#\+C1) | true.
all_holds(F,C,Z) \ if_then_or_holds(G,V,Z) <=>                        %4
   copy_fluent(F,C,F1,C1), F1=G, \+ call(#\+C1) | or_holds(V,Z).
all_holds(F,C,Z) \ if_then_or_holds(_,V,Z) <=>                        %5
   member(G,V), copy_fluent(F,C,F1,C1), F1=G, \+ call(#\+C1) | true.
all_holds(F,C,[G|Z]) <=>                                              %6
   \+ (F=G, call(C)) -> all_holds(F,C,Z) ;
   F=..[_|ArgX], G=..[_|ArgY],
   or_neq(exists,ArgX,ArgY,C1), all_holds(F,(C#/\C1),Z).
```

Fig. 5. The CHRs for universal quantification constraints. Following the syntax of Eclipse Prolog, the symbol # is used to identify the operators for composing finite domain constraints. The auxiliary predicate `copy_fluent(F,C,F1,C1)` defines `F1` and `C1` to be variable-disjoint variants of, respectively, fluent `F` and constraint `C`. CHR 1 is a so-called propagation rule, where the constraints in the head are not removed from the constraint store.

4.1 Handling the Universal Constraint

Figure 5 depicts unit resolution and propagation rules for constraint `all_holds`. Specifically, CHRs 1 and 2 model unit resolution wrt. negation constraints. For example, the constraints `all_holds(f(X),X#>5,Z)` and `not_holds(f(8),Z)` together imply, by CHR 1, the finite domain constraint `#\+8#>5`, which fails (since $8 > 5$). CHR 3 models the subsumption of a disjunctive constraint by a universal one. For example, given `all_holds(f(X),X#>5,Z)` the disjunctive constraint `or_holds([f(3),f(7)],Z)` is true (since $7 > 5$). CHR 4 models unit resolution wrt. an implication constraint whose antecedent is implied by a universal constraint, whereas CHR 5 solves an implication constraint whose succedent is subsumed by a universal constraint. Finally, CHR 6 defines the propagation of a universal constraint through a state list: In case the head fluent g of the state is within the range of the universal constraint, this range is restricted so as to be unequal to g. For example, propagation applied to `all_holds(f(X),X#>5,[f(6)|Z])` yields `all_holds(f(X),X#>6,Z)`.

A similar set of CHRs for negated universal quantification is given in Figure 6. Altogether the rules can be shown to be correct wrt. the foundational axioms of the fluent calculus.

Theorem 2. *CHRs 1–11 in Figures 5 and 6 are correct under the foundational axioms $\mathcal{F}_{state}$ and the assumption of uniqueness-of-names for all fluents.*

```
not_holds_all(F,Z) <=> all_not_holds(F,(0#=0),Z).

all_not_holds(F,C,Z) \ not_holds(G,Z) <=>                          %7
   copy_fluent(F,C,F1,C1), F1=G, \+ call(#\+C1) | true.
all_not_holds(F,C,Z) \ or_holds(V,Z) <=>                           %8
   member(G,V,W), copy_fluent(F,C,F1,C1), F1=G, \+ call(#\+C1)
   | or_holds(W,Z).
all_not_holds(F,C,Z) \ if_then_or_holds(G,_,Z) <=>                 %9
   copy_fluent(F,C,F1,C1), F1=G, \+ call(#\+C1) | true.
all_not_holds(F,C,Z) \ if_then_or_holds(F2,V,Z) <=>                %10
   member(G,V,W), copy_fluent(F,C,F1,C1), F1=G, \+ call(#\+C1)
   | if_then_or_holds(F2,W,Z).
all_not_holds(F,C,[G|Z]) <=>                                       %11
   (\+ (F=G, call(C)) -> true ;
   copy_fluent(F,C,F1,C1), F1=G, call(#\+C1)), all_not_holds(F,C,Z).
```

Fig. 6. CHRs for universally quantified negation.

4.2 Using the Universal Constraints

The new universal constraints can be used for domains with states in which infinitely many instances of a fluent are true and, at the same time, infinitely many of its instances are false. We conclude this section with a small example of a fluent F which, initially, is known to be true for all integers greater than 2 and false for all negative integers. The agent then performs several actions which affect specific instances of the fluent:

```
init(Z0) :- all_holds(f(X),X#>2,Z0),
            all_not_holds(f(X),X#<0,Z0), duplicate_free(Z0).

state_update(Z1,set(X),  Z2,[]) :- update(Z1,[f(X)],[],Z2).
state_update(Z1,reset(X),Z2,[]) :- update(Z1,[],[f(X)],Z2).

?- init(Z0),
   state_update(Z0,set(-2), Z1,[]),
   state_update(Z1,reset(5),Z2,[]),
   state_update(Z2,set(0),  Z3,[]).

Z3 = [f(0),f(-2) | Z]
Constraints:
all_holds(f(X),X#>2 #/\ X#\=5,Z)
all_not_holds(f(X),X#<0,Z)
not_holds(f(5),Z)
not_holds(f(0),Z)
duplicate_free(Z)
```

5 Summary

We have enriched significantly the expressiveness of state representations in FLUX by introducing both implication and universal quantification constraints. We have presented a set of Constraint Handling Rules based solely on simplification, propagation, and unit resolution, so that the efficiency of constraint solving in FLUX is retained. The rules have been formally verified against the fluent calculus.

The closest work is the logic program presented in [5] for GOLOG with incomplete states. The main difference is that states are only indirectly represented in GOLOG and that regression through the history of actions is used to evaluate conditions in agent programs. In contrast, FLUX uses explicit representations of incomplete states along with the computation mechanism of progression, which allows to evaluate conditions directly against the updated state. A further difference is that the GOLOG variant of [5] employs a general theorem prover for evaluating a regressed condition (against the initial state knowledge). In contrast, the motivation for FLUX is to retain a restricted expressiveness in order to be able to employ efficient inference techniques. Benchmark problems have shown that FLUX scales much better to domains with large state space and in which agents perform long sequences of actions [8, 7].

References

1. McCarthy, J.: Programs with common sense. In: Proc. of the Symposium on the Mechanization of Thought Processes. Volume 1, London (1958) 77–84.
2. McCarthy, J.: Situations and Actions and Causal Laws. Stanford Artif. Intell. Project, Memo 2, Stanford University, CA (1963)
3. McCarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. Machine Intell. **4** (1969) 463–502
4. Fikes, R.E., Nilsson, N.J.: STRIPS: A new approach to the application of theorem proving to problem solving. Artif. Intell. **2** (1971) 189–208
5. Reiter, R.: On knowledge-based programming with sensing in the situation calculus. ACM Transactions on Computational Logic **2** (2001) 433–457
6. Shanahan, M., Witkowski, M.: High-level robot control through logic. In Castelfranchi, C., Lespérance, Y., eds.: Proc. of the Internat. Workshop on Agent Theories Architectures and Languages. Vol. 1986 of LNCS, Springer (2000) 104–121
7. Thielscher, M.: Pushing the envelope: programming reasoning agents. In Baral, C., ed.: *AAAI Workshop on Cognitive Robotics*. AAAI Press (2002) 110–117
8. Thielscher, M.: FLUX: A logic programming method for reasoning agents. Theory and Practice of Logic Programming (2005). Available at: `www.fluxagent.org`
9. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming **37** (1998) 95–138
10. Thielscher, M.: From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. Artif. Intell. **111** (1999) 277–299
11. Thielscher, M.: Reasoning about actions with CHRs and finite domain constraints. In Stuckey, P., ed.: Proc. of ICLP. Vol. 2401 of LNCS, Springer (2002) 70–84
12. Van Hentenryck, P.: Constraint Satisfaction in Logic Programming. MIT Press (1989)