Michael Thielscher

The Fluent Calculus

A Specification Language for Robots with Sensors in Nondeterministic, Concurrent, and Ramifying Environments

Technical Report CL-2000-01

Computational Logic Group
Artificial Intelligence Institute
Department of Computer Science

Dresden University of Technology

Contents

1 Introduction

The research area of Cognitive Robotics is concerned with the design of robots that are capable of task planning on a high level. Similar to the evolution of machine-oriented programming into modern languages, a central goal of Cognitive Robotics is to provide programming methodologies for robots which allow to abstract as far as possible from concrete physical platforms and the specifics of concrete environments [29]. High-level robotics thus enables the design, maintenance, and adaptability and portability of large, complex systems for robot control.

Yet unlike programming a stationary computer, whose interaction with the environment is restricted to a few, clear-cut input/output facilities, autonomous robots are embedded in and constantly interact with a complex, dynamic environment. This raises two major challenges for the programmer. First, it is generally difficult if not impossible to program suitable action sequences for all possible tasks and situations. On the high level, this requires an autonomous robot to be capable of searching on its own for plans tailored to the current situation.

The second major challenge for programming an autonomous robot is that one cannot assume the robot to have full control over the environment. In particular, autonomous robots hardly ever have complete information about the state of their environment. On a high level, imprecise and incomplete state knowledge requires the capability of reasoning when devising a plan in order to ensure that such a plan achieves the given task under any circumstances.

Putting together these two challenges, a high-level programming methodology for autonomous robots requires the programmer to provide a formal, suitably abstract specification of both a type of robot and a class of environments. The former consists in a description of the basic actions a robot can undertake while the latter contains a specification of the dynamics of the environment and in particular how a robot can manipulate it by its actions. The programming methodology then needs to provide means of computing with such specifications with the goal of generating and executing suitable plans for concrete tasks in concrete situations.

The development of an expressive methodology for Cognitive Robotics is made intricate by the fact that robots in real-world environments face a variety of complications besides having to cope with incomplete information.

1. **Nondeterminism and Uncertainty.** Even if a robot has precise knowledge of, e.g., its current position, the result of an action like moving forward for a certain amount of time may be predictable only with some uncertainty.

2. **Knowledge and Sensing Actions.** Robots may lack sufficient state knowledge to come up with a unique sequence of actions by which is guaranteed that a given task is achieved. A robot then needs to gather additional information by actively sensing, e.g., whether a particular door is open. To plan this ahead, a robot needs to reason about what it knows and how knowledge is gained by sensing actions. Moreover, the planning methodology must provide means to condition actions on the result of sensing.

3. **Ramifications.** A simple direct effect of an action, such as picking up an object and moving it, may cause a number of additional effects, e.g., the simultaneous relocation of all objects on top of, attached to, or inside of the primary one. All these indirect effects need to be respected in order not to jump to false conclusions as to the positions of seemingly unaffected objects.

4. **Concurrency.** Certain goals may be achievable only by performing actions concurrently, such as lifting a larger object from two sides. This requires to distinguish between the usual effect of actions and synergic effects. On the other hand, concurrent actions may interfere by canceling effects.

Research into Cognitive Robotics has progressed rapidly in the recent past: Launched in the early nineties by new, solid solutions[1] to the most fundamental modeling problem, the Frame Problem [43], a great number of theories have been developed for reasoning agents in complex environments. However, the existence of theoretical accounts for all of the abovementioned aspects does not imply that there be a unique model, let alone an executable specification language, which covers them all. Rather, these issues have mostly been investigated in isolation. As a consequence, combining co-existing models for different phenomena is often a problem as challenging as addressing further aspects.

In this paper, we present the Fluent Calculus as a specification language and system for robots which meets the requirement to address all of the aspects listed above in a uniform way. The calculus roots in the logic programming formalism of [21], which in [18] has been proved equivalent to approaches to the Frame Problem that appeal to non-classical logics, namely, linearized versions of the connection method [3] and Gentzen's sequent calculus [37], resp. All three frameworks have been designed especially to address not only the representational but also the inferential aspect of the Frame Problem [4]. These approaches have thus been characterized as attempts to reconcile the expressive power of logical reasoning with the classical procedural solution to the Frame Problem of STRIPS [9]. The Fluent Calculus as will be presented in this paper comes closest to this goal since it provides a direct characterization of STRIPS-style state update in pure first-order logic. On the other hand, with the full expressive power of first-order logic, the Fluent Calculus can be viewed as a development of the Situation Calculus [40] and in particular the concept of successor state axioms [52, 65].

[1]An excellent overview of today's established action formalisms is provided by the set of reference articles published in [56].

Our major achievements in this paper are the following.

1. We provide the Fluent Calculus, which roots in the logic programming formalism of [21], with a new, simpler algebraic foundation. In so doing, we overcome a limitation of the existing axiomatization, which does not permit domain-specific equalities [65].

2. We present the programming language FLUX (the *Flu*ent *Calculus Ex*ecutor), which implements the Fluent Calculus using constraint logic programming [23, 11]. The core of this implementation is formally verified against the new algebraic theory of the Fluent Calculus. In relation to existing systems, the big achievement of FLUX is that incomplete knowledge of states is dealt with in a way that is both conceptually simple and computationally efficient.

3. We present a novel theory of knowledge and sensing and reconcile it with the Fluent Calculus. The approach is distinguished by its simple inference scheme for calculating the effects of actions on knowledge and a comparatively simple account of non-knowledge. Moreover, we show how sensing actions can be specified and computed in FLUX by exploiting the simple representation of incomplete states. As an outstanding feature, planning problems can be solved with FLUX where the goal is to acquire knowledge.

4. We reconcile isolated existing accounts of nondeterministic actions [67], ramifications [63], and concurrency [66] with the new Fluent Calculus. As a result we obtain the first theory which uniformly covers all of these aspects. Moreover, it is shown how nondeterminism, indirect effects, and the concurrent execution of actions can be programmed in FLUX.

The paper is organized as follows. After brief preliminaries on notational conventions, we introduce and formally discuss the new algebraic foundation of the Fluent Calculus in Section 2, show how actions and their effects are formalized so as to solve the Frame Problem (Section 3), and present basic FLUX in Section 4. The second part of the paper is devoted to successive extensions of the simple Fluent Calculus and FLUX, namely, nondeterminism (Section 5), sensing (Section 6), ramifications (Section 7), and concurrency (Section 8). Throughout the paper, we use the model of a delivery robot as example. The full FLUX program is shown in the appendix.

Preliminaries

The general Fluent Calculus is a second-order logic language with equality. The latter means to consider only interpretations in which the equality predicate "=" is interpreted as identity among the domain elements.[2] We also use the formal concept of sorts. This amounts to requiring the domain of an interpretation to contain at least one element of each sort and to assign to sorted variables, functions, and predicates, resp., only domain elements, functions among domain elements, and relations among domain elements, resp., of the right sort.[3] Sorts are generally disjoint, except for cases where one sort σ_1 is designated as sub-sort of another one, σ_2, written $\sigma_1 < \sigma_2$.

We will use the standard logical connectives "$\neg$" (negation), "$\wedge$" (conjunction), "$\vee$" (disjunction), "$\supset$" (implication), "$\equiv$" (equivalence), "$\forall$" (universal quantification), and "$\exists$" (existential quantification). Predicate and function symbols, including constants, start with a capital letter

[2]see, e.g., [7], Section 7.
[3]*ibid.*, Section 4.

whereas variables are in lower case, sometimes with sub- or superscripts. Free variables in formulas are assumed universally quantified. Sequences $x_1, \ldots, x_n$ of pairwise different variables are often written as $\vec{x}$. By $\vec{x} = \vec{y}$ we then mean $x_1 = y_1 \wedge \ldots \wedge x_n = y_n$.

For notational convenience, we adopt from [1] the following notation for sets of equational axioms expressing uniqueness of names:

$$UNA[h_1, \ldots, h_n] \stackrel{\text{def}}{=} \bigwedge_{i<j} h_i(\vec{x}) \neq h_j(\vec{y}) \wedge \bigwedge_{i} [h_i(\vec{x}) = h_i(\vec{y}) \supset \vec{x} = \vec{y}]$$

For example, $UNA[\mathit{Alley}, \mathit{R401}, \mathit{R402}]$ is $\mathit{Alley} \neq \mathit{R401} \wedge \mathit{Alley} \neq \mathit{R402} \wedge \mathit{R401} \neq \mathit{R402}$, and $UNA[\mathit{Closed}]$ is $\mathit{Closed}(x) = \mathit{Closed}(y) \supset x = y$.

2 State Terms

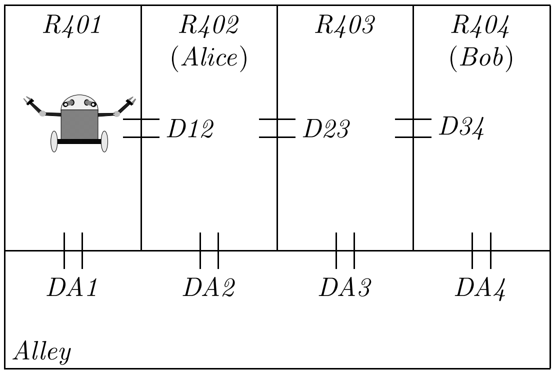
The name Fluent Calculus derives from the theory's most fundamental entity, namely, the fluent. A fluent represents a single atomic property of the physical world which may change in the course of time, in particular through manipulation by the robot. Examples of such properties can be the location of a movable object, the status of a door (i.e., whether open or closed), or the position of the robot. Being a logic with sorts, the Fluent Calculus contains the reserved sort FLUENT for these entities. Thus, formally speaking, fluents are terms in the language. This technique of representing properties as terms, generally known as reification [50], introduces a great deal of flexibility as regards reasoning about the manipulation of these properties, which is essentially what a robot has to do. An example for fluent definitions as part of a domain signature is depicted in Figure 1. Throughout the paper, variables of sort FLUENT are denoted by the letter f, possibly with sub- or superscripts.

Based on the notion of fluents, a so-called state is a snapshot of the environment at a certain moment. The reserved sort STATE is used for terms denoting states. State terms are often mere abstract denotations like *InitialState*. On the other hand, given a definition of the fluents of a domain, each fluent constitutes a particular state, namely, the one in which just this fluent holds. Thus, formally speaking, FLUENT is a sub-sort of STATE.

State terms, in particular fluents, can be composed to new states with the reserved function "$\circ$" of type STATE $\times$ STATE $\mapsto$ STATE. Written in infix notation, this function maps two states into a state in which the fluents of both arguments hold. For example, the term $\mathit{InitialState} \circ \mathit{Carries}(\mathit{Projector})$ denotes the state which is exactly like the initial one but the robot has picked up the projector. Another example is the state term $(\mathit{InRoom}(\mathit{Office}(\mathit{Alice})) \circ \mathit{Carries}(x)) \circ z$ with variables x and z being of sort OBJECT and STATE, resp.; this term describes a state in which the robot is in Alice's office carrying something and in which arbitrary other fluents, summarized in z, hold. For technical reasons, the Fluent Calculus includes the pre-defined STATE constant $\emptyset$ denoting the empty state, i.e., in which—intuitively—no fluent is true. Throughout the paper, variables of sort STATE are denoted by the letter z, possibly with sub- or superscripts.

A fundamental notion is that of a fluent to *hold* in a state. Fluent f is said to hold in state z if z can be decomposed into two states one of which is the singleton f. Conversely, f does not hold in z if the latter cannot be decomposed in this way. For notational convenience, we introduce the macro $\mathit{Holds}(f, z)$ as an abbreviation for the corresponding equality formula:

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



domain sorts	
ROOM	*Alley*, *R401*, *Office*(*Alice*), ...
DOOR	*DA1*, *DA2*, *D12*, ...
OBJECT	*Projector*, *DocumentFolder*, ...
PERSON	*Alice*, *Bob*, ...

function	type	meaning
Office	PERSON $\mapsto$ ROOM	office of person x
InRoom	ROOM $\mapsto$ FLUENT	the robot is in room x
AtDoor	DOOR $\mapsto$ FLUENT	the robot is at door x
Closed	DOOR $\mapsto$ FLUENT	door x is closed
HasKeyCode	DOOR $\mapsto$ FLUENT	robot has the key code for door x
Carries	OBJECT $\mapsto$ FLUENT	robot carries object x
Request	ROOM $\times$ OBJECT $\times$ ROOM $\mapsto$ FLUENT	there is a request to deliver object x_2 from room x_1 to room x_3

Figure 1: A delivery scenario. The signature consists of four domain-specific sorts, for which some example terms are shown. States are described on the basis of six functions whose range is the reserved sort FLUENT. Examples of fluent terms are, $\mathit{Closed}(\mathit{DA2})$, $\mathit{InRoom}(\mathit{Office}(\mathit{Alice}))$, or $\mathit{Request}(\mathit{R401}, x, \mathit{R404})$ with variable x being of sort OBJECT.

This fundamental notion of truth and falsity of fluents in states requires a special theory of equality of state terms. The following new fundamental axioms of the Fluent Calculus serve this purpose.

Definition 1 Assume a signature which includes the sorts FLUENT < STATE and the functions $\circ, \emptyset$ of sorts as above. The set $\mathcal{F}_{state}$ comprises these equational axioms:

1. Axioms ACI1 (associativity, commutativity, idempotency, unit element),

$$\begin{aligned}(z_1 \circ z_2) \circ z_3 &= z_1 \circ (z_2 \circ z_3) \\ z_1 \circ z_2 &= z_2 \circ z_1 \\ z \circ z &= z \\ z \circ \emptyset &= z\end{aligned} \tag{2}$$

2. Decomposition axiom

$$\mathit{Holds}(f, f_1 \circ z) \supset f = f_1 \vee \mathit{Holds}(f, z) \tag{3}$$

□

Axioms ACI1 essentially characterize "$\circ$" as the union operation with $\emptyset$ as the empty set of fluents. (Associativity allows to omit parentheses in nested applications of "$\circ$".) The decomposition axiom relates equality of states to equality of fluents; note that (3) is just a representation of an equality formula according to (1).

Although the explicit, fundamental notion of a state is the characteristic feature of the Fluent Calculus, we do not assume that a robot has complete knowledge of the state of its environment. Rather, (partial) knowledge about states is represented by formulas talking about abstract state denotations, like *InitialState*. The following, for example, may be a suitable description of what is known about a state in our delivery scenario (cf. Figure 1):

$$\begin{array}{l} \mathit{Holds}(\mathit{InRoom}(\mathit{R401}), \mathit{InitialState}) \wedge \mathit{Holds}(\mathit{AtDoor}(\mathit{D12}), \mathit{InitialState}) \wedge \\ \mathit{Holds}(\mathit{Closed}(\mathit{D12}), \mathit{InitialState}) \wedge \neg \mathit{Holds}(\mathit{Closed}(\mathit{DA1}), \mathit{InitialState}) \wedge \\ (\forall y)\, \neg \mathit{Holds}(\mathit{Carries}(y), \mathit{InitialState}) \wedge \\ (\forall d)\, (\mathit{Holds}(\mathit{HasKeyCode}(d), \mathit{InitialState}) \equiv d = \mathit{D12} \vee d = \mathit{DA4}) \wedge \\ (\exists x)(\forall r_1, r_2, y)\, (\mathit{Holds}(\mathit{Request}(r_1, y, r_2), \mathit{InitialState}) \equiv \\ \qquad\qquad r_1 = \mathit{Office}(\mathit{Alice}) \wedge y = x \wedge r_2 = \mathit{Office}(\mathit{Bob})) \end{array} \tag{4}$$

That is to say, given are the robot's location, the status of doors $\mathit{D12}$ and $\mathit{DA1}$, the fact that the tray of the robot is clear, that it possesses two and no more key codes, and that there is a single request. Notice that in particular nothing is known about the states of other doors besides the two leading out of room $\mathit{R401}$.

Formulas about states may stipulate an unbounded or even infinite number of fluents to hold in a state, as in $(\forall x)\, \mathit{Holds}(\mathit{Closed}(x), z)$ or $(\forall n)\, (n \geq 2001 \supset \mathit{Holds}(\mathit{FutureYear}(n), z))$ where $\mathit{FutureYear}$ is of type NAT (natural numbers) $\mapsto$ FLUENT. The possibility of formulas which require states to comprise infinitely many fluents raises the issue of consistency of such formulas wrt. our foundational axioms. Theorem 3 below ensures that $\mathcal{F}_{\mathit{state}}$ is indeed consistent with any formula about a state provided the formula is not self-contradictory. Prior to formalizing and proving this, we need to make precise the notion of a formula about a state:

Definition 2 A *pure state formula in* z is a first-order formula $\Phi(z)$ with just one free state variable z and which is composed of atomic formulas of the form

1. $\mathit{Holds}(\phi, z)$, where ϕ is of sort fluent;[4]
2. atoms which do not use any of the reserved predicates or sorts of the Fluent Calculus.

□

The idea behind the proof that our axiomatization of states is consistent, is quite a simple one: If a pure state formula admits a model $\mathcal{M}$, then a concrete state can always be obtained by letting it contain precisely those fluents that hold in it according to $\mathcal{M}$.

Theorem 3 *Let ζ be a constant of sort* STATE *and let $\Phi(z)$ be a pure state formula in z. If $\Phi(\zeta)$ is satisfiable then $\mathcal{F}_{\mathit{state}} \cup \{\Phi(\zeta)\}$ is satisfiable.*

Proof: Let $\mathcal{M}$ be a model of $\Phi(\zeta)$. We construct a model $\mathcal{M}^*$ of $\mathcal{F}_{\mathit{state}} \cup \{\Phi(\zeta)\}$ as follows. Let $\mathcal{M}^*$ be as $\mathcal{M}$ except that

1. the domain for sort FLUENT consists of all singleton sets $\{F\}$ where F is a domain element for sort FLUENT in $\mathcal{M}$;
2. the domain for sort STATE consists of all sets over the domain for sort FLUENT in $\mathcal{M}$;
3. constant $\emptyset$ is interpreted by the empty set;

[4]Notice that by (1) these atoms are actually equality sentences.

function	type	meaning
Go	DOOR $\mapsto$ ACTION	go to door x
Open	DOOR $\mapsto$ ACTION	open door x
Enter	ROOM $\mapsto$ ACTION	enter room x
Pickup	OBJECT $\mapsto$ ACTION	pick up object x
Drop	OBJECT $\mapsto$ ACTION	drop object x
Connects	DOOR $\times$ ROOM $\times$ ROOM	door x_1 connects rooms x_2 and x_3

$$\begin{array}{l}
\mathit{Office}(\mathit{Alice}) = \mathit{R402} \;\wedge\; \mathit{Office}(\mathit{Bob}) = \mathit{R404} \\
\mathit{Connects}(d,x,y) \;\equiv\; d = \mathit{D12} \;\wedge\; x = \mathit{R401} \;\wedge\; y = \mathit{R402} \;\vee \\
\qquad\qquad\qquad\qquad\quad d = \mathit{D12} \;\wedge\; x = \mathit{R402} \;\wedge\; y = \mathit{R401} \;\vee \\
\qquad\qquad\qquad\qquad\quad d = \mathit{DA1} \;\wedge\; x = \mathit{Alley} \;\wedge\; y = \mathit{R401} \;\vee \\
\qquad\qquad\qquad\qquad\quad \ldots \\
r = \mathit{Alley} \vee r = \mathit{R401} \vee \ldots \vee r = \mathit{R404} \\
d = \mathit{DA1} \vee d = \mathit{D12} \vee \ldots \vee d = \mathit{DA4} \\
\mathit{UNA}[\mathit{DA1}, \ldots, \mathit{D34}] \;\wedge\; \mathit{UNA}[\mathit{Alley}, \mathit{R401}, \ldots, \mathit{R404}] \\
\mathit{UNA}[\mathit{InRoom}, \mathit{AtDoor}, \mathit{Closed}, \mathit{HasKeyCode}, \mathit{Carries}, \mathit{Request}] \\
\mathit{UNA}[\mathit{Go}, \mathit{Open}, \mathit{Enter}, \mathit{Pickup}, \mathit{Drop}]
\end{array}$$

Figure 2: Additions to the signature for the delivery scenario. The robot can perform five high-level actions. The floor plan is specified with the help of the domain predicate *Connects*. The domain axiomatization also contains domain closure axioms and axioms on uniqueness of names as depicted.

4. function $\circ$ is interpreted by the union operation;

5. constant ζ is interpreted as the set of all F such that $\mathcal{M} \models \mathit{Holds}^{\mathcal{M}}(F, \zeta^{\mathcal{M}})$.

To prove that $\mathcal{M}^*$, too, is a model of $\Phi(\zeta)$, it suffices to show that $\mathcal{M}^* \models \mathit{Holds}^{\mathcal{M}^*}(\{F\}, \zeta^{\mathcal{M}^*})$ iff $\mathcal{M} \models \mathit{Holds}^{\mathcal{M}}(F, \zeta^{\mathcal{M}})$: Following (1), $\mathcal{M}^* \models \mathit{Holds}^{\mathcal{M}^*}(\{F\}, \zeta^{\mathcal{M}^*})$ iff there exists a set $\mathcal{Z}'$ such that $\zeta^{\mathcal{M}^*} = \{F\} \cup \mathcal{Z}'$, hence iff F is an element of $\zeta^{\mathcal{M}^*}$, hence iff $\mathcal{M} \models \mathit{Holds}^{\mathcal{M}}(F, \zeta^{\mathcal{M}})$ according to item 5 above.

To prove that $\mathcal{M}^*$ is a model of ACI1, it suffices to note that set union is an associative-commutative and idempotent operation with the empty set as its unit element.

To prove that decomposition holds, suppose $\mathit{Holds}^{\mathcal{M}^*}(\{F_1\}, \{F_2\} \cup \mathcal{Z})$, that is, $\{F_2\} \cup \mathcal{Z} = \{F_1\} \cup \mathcal{Z}'$ for some $\mathcal{Z}'$. Then $F_1 = F_2$ or there exists some $\mathcal{Z}''$ such that $\mathcal{Z} = \{F_1\} \cup \mathcal{Z}''$, that is, $\mathit{Holds}^{\mathcal{M}^*}(\{F_1\}, \mathcal{Z})$. ■

3 Simple State Update Axioms

Actions are the second fundamental entity, besides fluents, in the Fluent Calculus. Actions are denoted by terms of the reserved sort ACTION. To our delivery robot of Figure 1, for example, we ascribe the ability to perform five kinds of high-level actions, namely, finding its way to a door, opening a door by sending out they key code, entering a room through an open door, picking up a requested object, and dropping an object which it carries on its tray. The formal definition of these actions as an extension of the domain signature, along with some foundational domain axioms, is shown in Figure 2.

For the formalization of action preconditions, the Fluent Calculus provides the pre-defined predicate $Poss : \textsc{action} \times \textsc{state}$. The intended reading is that its first argument is an action which is possible in the state denoted by the second argument.

Definition 4 Let A be a function symbol with range $\textsc{action}$. A *simple action precondition axiom for* A is of the form

$$Poss(A(\vec{x}), z) \equiv \Pi_A(\vec{x}, z)$$

where $\Pi_A(\vec{x}, z)$ is a pure state formula in z. □

For example, the precondition axioms for the actions of our robot shall be as follows:

$$\begin{aligned}
Poss(Go(d), z) &\equiv (\exists r, r') \, (Holds(InRoom(r), z) \wedge Connects(d, r, r')) \\
Poss(Open(d), z) &\equiv Holds(AtDoor(d), z) \\
&\qquad\qquad \wedge [Holds(HasKeyCode(d), z) \vee \neg Holds(Closed(d), z)] \\
Poss(Enter(r), z) &\equiv (\exists d, r') \, (Holds(AtDoor(d), z) \wedge Holds(InRoom(r'), z) \\
&\qquad\qquad \wedge Connects(d, r', r) \wedge \neg Holds(Closed(d), z)) \qquad (5) \\
Poss(Pickup(x), z) &\equiv (\exists r_1, r_2) \, (Holds(Request(r_1, x, r_2), z) \wedge Holds(InRoom(r_1), z) \\
&\qquad\qquad \wedge \neg Holds(Carries(x), z)) \\
Poss(Drop(x), z) &\equiv Holds(Carries(x), z) \\
&\qquad \wedge (\exists r_1, r_2) \, (Holds(Request(r_1, x, r_2), z) \wedge Holds(InRoom(r_2), z))
\end{aligned}$$

Based on the notion of actions, a so-called situation is a history of action performances [40, 28]. Situations are represented by terms of the reserved sort $\textsc{sit}$. The standard function $Do : \textsc{action} \times \textsc{sit} \mapsto \textsc{sit}$ maps into a new situation a pair consisting of an action and a situation, which may be a constant like, e.g., S_0 denoting a particular, initial situation. For example, the situation term $Do(Pickup(x), Do(Enter(R402), Do(Open(D12), S_0)))$ represents the beginning of a potential plan for our delivery robot starting in situation S_0.

The reserved function $State : \textsc{sit} \mapsto \textsc{state}$ maps each situation to the state of the environment in that situation. However, since complete descriptions of states are not assumed, the expression $State(s)$ is a mere abstract denotation of a world state. Knowledge about situations is then formalized by referring to the associated state term. An example are the following two macros, which denote, resp., that a fluent holds in a situation and that an action is possible:

$$\begin{aligned}
Holds(f, s) &\stackrel{\text{def}}{=} Holds(f, State(s)) \\
Poss(a, s) &\stackrel{\text{def}}{=} Poss(a, State(s))
\end{aligned} \qquad (6)$$

Suppose, for example, we define $State(S_0) = InitialState$ as specified with formula (4), then $(\exists r, r') \, (Holds(InRoom(r), S_0) \wedge Connects(DA1, r, r'))$. Hence, $Poss(Go(DA1), S_0)$ according to (5).

A further example of talking about situations in terms of the corresponding state are the so-called state constraints, which formalize properties a $\textsc{state}$ term must satisfy in order to represent a state that can actually occur in the world.

Definition 5 Let $\Gamma(z)$ be a pure state formula in z, then $\Gamma(State(s))$ is a *state constraint*. □

Our example domain of the delivery robot calls for these constraints, which should be self-explanatory:

$$
\begin{array}{l}
(\exists r)\, \mathit{Holds}(\mathit{InRoom}(r), s) \\
\mathit{Holds}(\mathit{InRoom}(r), s) \wedge \mathit{Holds}(\mathit{InRoom}(r'), s) \supset r = r' \\
\mathit{Holds}(\mathit{AtDoor}(d), s) \wedge \mathit{Holds}(\mathit{AtDoor}(d'), s) \supset d = d' \\
\mathit{Holds}(\mathit{InRoom}(r), s) \wedge \mathit{Holds}(\mathit{AtDoor}(d), s) \supset (\exists r')\, \mathit{Connects}(d, r, r') \\
\mathit{Holds}(\mathit{Request}(r_1, x, r_2), s) \wedge \mathit{Holds}(\mathit{Request}(r_1', x, r_2'), s) \supset r_1 = r_1' \wedge r_2 = r_2'
\end{array}
\tag{7}
$$

(With the last constraint we reject contradicting requests concerning the same object.)

Reasoning about actions essentially means to reason about the effects of performing them. A crucial advantage when specifying effects lies in the fact that actions almost always affect only very few fluents and thus leave most of a state unchanged.[5] Hence, actions should be conveniently describable by saying which fluents under which circumstances are changed by each action, while the vast majority of fluents is not explicitly mentioned and assumed unchanged. On the other hand, it is precisely this desire for 'focused' action specifications that brings about the famous Frame Problem, which has been uncovered as early as in [39] and is recognized as one of the most important modeling problems in computer science [60].

The Frame Problem is to find a representation formalism which allows for specifying actions solely in terms of effects in a way that all unchanged knowledge about a state still follows about the successor state. This representational aspect of the Frame Problem is linked with an inferential perspective concerned with the efficient computation of non-changes, which essentially means to not apply separate inference steps for each unaffected piece of knowledge.

The Fluent Calculus approach to the Frame Problem exploits the explicit notion of states. Change is modeled by specifying the difference between two states. The simple Fluent Calculus is restricted to deterministic actions which are performed in isolation and which have a bounded number of direct and no indirect effects. Positive effects are modeled by adding them to a state, negative effects are modeled by removing them. We denote removal of a fluent by $z - f$; the axiomatic characterization of this operation is as follows:

$$
z - f = z' \stackrel{\text{def}}{=} \neg \mathit{Holds}(f, z') \wedge [z' \circ f = z \vee z' = z] \tag{8}
$$

Put in words, removing f from z results in z' just in case $\neg \mathit{Holds}(f, z')$ and either z' plus f equals z (in case $\mathit{Holds}(f, z)$) or z' equals z (in case $\neg \mathit{Holds}(f, z)$). It is easy to see that the macro can be generalized to removal of finite collections of fluents:

$$
\begin{array}{rcl}
z' = z - \emptyset & \stackrel{\text{def}}{=} & z' = z \\
z' = z - f_1 \circ \ldots \circ f_n \circ f_{n+1} & \stackrel{\text{def}}{=} & (\exists z'')\, (z'' = z - f_1 \circ \ldots \circ f_n \wedge z' = z'' - f_{n+1})
\end{array}
\tag{9}
$$

On this basis, effects of actions in the simple Fluent Calculus are specified as follows:

Definition 6 A *fluent collection* is either the empty STATE $\emptyset$ or a STATE term of the form $F_1(\vec{\tau}_1) \circ \ldots \circ F_n(\vec{\tau}_n)$ where each F_i is a function symbol with range FLUENT $(1 \leq i \leq n;\ n \geq 1)$.

Let A be a function symbol with range ACTION. A *simple state update axiom* for A is of the form

$$
\mathit{Poss}(A(\vec{x}), s) \wedge \Delta(\vec{x}, \mathit{State}(s)) \supset \mathit{State}(\mathit{Do}(A(\vec{x}), s)) = \mathit{State}(s) \circ \vartheta^{+} - \vartheta^{-}
$$

[5] In the words of [46], actions are "local surgeries."

where $\Delta(\vec{x}, z)$ is a pure state formula in z and ϑ^+ (the *positive* effects) and ϑ^- (the *negative* effects) are fluent collections. □

Besides the presupposition that the action is possible, sub-formula Δ may represent additional conditions under which the equation in the consequent of a state update axiom defines the relation between a state and its successor. Thus actions can be specified by more than one update axiom, each one of which specifies the action's effect in different circumstances.

Under the assumption that positive and negative effects are disjoint, state update axioms are a provably correct solution to the Frame Problem. More specifically, the following theorem establishes that all positive and negative effects materialize, provided they do not cancel out, and that all other fluents hold in the resulting state just in case they hold in the original state.

Theorem 7 *Consider the fluent collections* $\vartheta^+ = f_1^+ \circ \ldots \circ f_m^+$ *and* $\vartheta^- = f_1^- \circ \ldots \circ f_n^-$ $(m \geq 0, n \geq 0)$*. Then* $\mathcal{F}_{state} \cup \{\bigwedge_i \bigwedge_j f_i^+ \neq f_j^-\} \cup \{z_{new} = z_{old} \circ \vartheta^+ - \vartheta^-\}$ *entails,*

1. $Holds(f_i^+, z_{new})$ *for all* $i = 1, \ldots, m$
2. $\neg Holds(f_j^-, z_{new})$ *for all* $j = 1, \ldots, n$
3. $(\forall f)\,(\bigwedge_i f \neq f_i^+ \wedge \bigwedge_j f \neq f_j^- \supset [Holds(f, z_{new}) \equiv Holds(f, z_{old})]\,)$

Proof:

1. Consider any $i \in \{1, \ldots, m\}$. Let $z = z_{old} \circ \vartheta^+$, then $Holds(f_i^+, z)$. After n-fold application of (9), decomposition in conjunction with $f_i^+ \neq f_j^-$ implies $Holds(f_i^+, z - \vartheta^-)$, hence $Holds(f_i^+, z_{new})$.

2. Follows immediately by n-fold application of (9).

3. Suppose $\bigwedge_i f \neq f_i^+$. Let $z = z_{old} \circ \vartheta^+$, then by m-fold application of decomposition, $Holds(f, z) \equiv Holds(f, z_{old})$. After n-fold application of (9) to $z_{new} = z - \vartheta^-$, decomposition and $\bigwedge_j f \neq f_j^-$ imply that $Holds(f, z_{new}) \equiv Holds(f, z)$. Hence, $Holds(f, z_{new}) \equiv Holds(f, z_{old})$.

■

To illustrate the design and use of state update axioms, we recall the five actions of our delivery scenario. Taking into account the precondition axioms of (5), the actions can be specified by this collection of state update axioms:

$$\begin{array}{l}
Poss(Go(d), s) \wedge Holds(AtDoor(d'), s) \wedge d' \neq d \supset \\
\quad State(Do(Go(d), s)) = State(s) \circ AtDoor(d) - AtDoor(d') \\
Poss(Go(d), s) \wedge [\, Holds(AtDoor(d), s) \vee \neg(\exists d')\, Holds(AtDoor(d'), s)\,] \supset \\
\quad State(Do(Go(d), s)) = State(s) \circ AtDoor(d) \\
Poss(Open(d), s) \supset State(Do(Open(d), s)) = State(s) - Closed(d) \\
Poss(Enter(r), s) \wedge Holds(InRoom(r'), s) \supset \\
\quad State(Do(Enter(r), s)) = State(s) \circ InRoom(r) - InRoom(r') \\
Poss(Pickup(x), s) \supset State(Do(Pickup(x), s)) = State(s) \circ Carries(x) \\
Poss(Drop(x), s) \wedge Holds(Request(r_1, x, r_2), s) \supset \\
\quad State(Do(Drop(x), s)) = State(s) - Carries(x) - Request(r_1, x, r_2)
\end{array} \tag{10}$$

Action Go has a conditional effect, depending on whether the robot happens to be at some other door initially. The reader may further notice that by executing a Drop action the robot successfully completes the corresponding delivery request.

State update axioms meet the representational requirements of the Frame Problem since the state equation in their consequent mentions only fluents that change. Moreover, state update axioms lay the foundation for efficiently inferring the result of actions if constrained equations of the following form are used to encode state knowledge:

$$(\exists \vec{x}, z)\,(\mathit{State}(s) = f_1 \circ \ldots \circ f_n \circ z) \,\wedge\, \Phi(\vec{x}, z)\,) \tag{11}$$

where $\Phi(\vec{x}, z)$ is a pure state formula in z. For example, recall $\mathit{State}(S_0) = \mathit{InitialState}$. The following formula is logically equivalent to the macro expansion of (4):

$$\begin{aligned}(\exists x, z)\,(\,\mathit{State}(S_0) = {} & \mathit{InRoom}(\mathit{R401}) \circ \mathit{AtDoor}(\mathit{D12}) \circ \mathit{Closed}(\mathit{D12}) \circ \\ & \mathit{HasKeyCode}(\mathit{D12}) \circ \mathit{HasKeyCode}(\mathit{DA4}) \circ \\ & \mathit{Request}(\mathit{Office}(\mathit{Alice}), x, \mathit{Office}(\mathit{Bob})) \circ \\ & z \\ \wedge\, \neg \mathit{Holds}&(\mathit{Closed}(\mathit{DA1}), z) \\ \wedge\, (\forall y)\, \neg \mathit{Holds}&(\mathit{Carries}(y), z) \\ \wedge\, (\forall d)\, \neg \mathit{Holds}&(\mathit{HasKeyCode}(d), z) \\ \wedge\, (\forall r_1, y, r_2)\, \neg \mathit{Holds}&(\mathit{Request}(r_1, y, r_2), z)\,)\end{aligned} \tag{12}$$

(Equivalence to (4) follows by decomposition and the axioms concerning uniqueness-of-names of Figure 2.) We have already derived $\mathit{Poss}(\mathit{Go}(\mathit{DA1}), S_0)$. Furthermore, $\mathit{Holds}(\mathit{AtDoor}(\mathit{D12}), S_0)$ (by (12)) and $\mathit{D12} \neq \mathit{DA1}$. Thus the instance $\{d/\mathit{DA1}, d'/\mathit{D12}, s/S_0\}$ of the first update axiom in (10) for Go implies

$$\mathit{State}(\mathit{Do}(\mathit{Go}(\mathit{DA1}), S_0)) = \mathit{State}(S_0) \circ \mathit{AtDoor}(\mathit{DA1}) - \mathit{AtDoor}(\mathit{D12})$$

Let $S_1 = \mathit{Do}(\mathit{Go}(\mathit{DA1}), S_0)$. Since $\mathit{Holds}(\mathit{AtDoor}(\mathit{D12}), \mathit{State}(S_0))$, macro definition (8) implies

$$\mathit{State}(S_1) \circ \mathit{AtDoor}(\mathit{D12}) = \mathit{State}(S_0) \circ \mathit{AtDoor}(\mathit{DA1}) \;\wedge\; \neg \mathit{Holds}(\mathit{AtDoor}(\mathit{D12}), \mathit{State}(S_1))$$

Replacing sub-term $\mathit{State}(S_0)$ by an equal term according to (12) yields

$$\begin{aligned}(\exists x, z)\, \mathit{State}(S_1) \circ \mathit{AtDoor}(\mathit{D12}) = {} & \mathit{InRoom}(\mathit{R401}) \circ \mathit{AtDoor}(\mathit{D12}) \circ \mathit{Closed}(\mathit{D12}) \circ \\ & \mathit{HasKeyCode}(\mathit{D12}) \circ \mathit{HasKeyCode}(\mathit{DA4}) \circ \\ & \mathit{Request}(\mathit{Office}(\mathit{Alice}), x, \mathit{Office}(\mathit{Bob})) \circ \\ & z \circ \\ & \mathit{AtDoor}(\mathit{DA1}) \\ \wedge\, \neg \mathit{Holds}(\mathit{AtDoor}(\mathit{D12}), \mathit{State}(S_1))&\end{aligned}$$

Since $\neg \mathit{Holds}(\mathit{AtDoor}(\mathit{D12}), \mathit{State}(S_1))$ and because state variable z in (12) can be chosen such that $\neg \mathit{Holds}(\mathit{AtDoor}(\mathit{D12}), z)$, fluent $\mathit{AtDoor}(\mathit{D12})$ can be canceled out on both sides of the equation, which yields

$$\begin{aligned}(\exists x, z)\,(\,\mathit{State}(S_1) = {} & \mathit{InRoom}(\mathit{R401}) \circ \mathit{AtDoor}(\mathit{DA1}) \circ \mathit{Closed}(\mathit{D12}) \circ \\ & \mathit{HasKeyCode}(\mathit{D12}) \circ \mathit{HasKeyCode}(\mathit{DA4}) \circ \\ & \mathit{Request}(\mathit{Office}(\mathit{Alice}), x, \mathit{Office}(\mathit{Bob})) \circ \\ & z \\ \wedge\, \neg \mathit{Holds}&(\mathit{AtDoor}(\mathit{D12}), z)\,)\end{aligned}$$

Besides the positive effect $AtDoor(DA1)$, the right hand side of the equation includes all fluents which remain unchanged by the action. Moreover, knowledge specified in (12) as to which fluents do not hold in z applies to the new state, which includes z, just as well. Thus all unchanged knowledge continues to hold without the need to apply extra inference steps.

Automated reasoning with equational theories is, however, known to be notoriously difficult. An efficient implementation of the Fluent Calculus therefore requires special techniques for dealing with both state equations and constraints on STATE variables. We will present such an implementation in the following Section 4.

The core concepts of the Fluent Calculus are summarized in the two notions of its basic signature and of axiomatizations of simple action domains as follows.

Definition 8 The *simple Fluent Calculus* is a sorted first-order logic language with equality which includes

1. Sorts
$$\text{FLUENT} < \text{STATE},\ \text{ACTION},\ \text{SIT}$$

2. Functions
$$\begin{array}{ll} \emptyset\colon\ \mapsto \text{STATE} & State\colon \text{SIT} \mapsto \text{STATE} \\ \circ\colon \text{STATE} \times \text{STATE} \mapsto \text{STATE} & Do\colon \text{ACTION} \times \text{SIT} \mapsto \text{SIT} \end{array}$$

3. Predicate
$$Poss\colon\ \text{ACTION} \times \text{STATE}$$

A *simple Fluent Calculus domain axiomatization* consists of a set of state constraints, a unique simple action precondition axiom for each function symbol with range ACTION, a set of simple state update axioms, foundational axioms $\mathcal{F}_{state}$, and possibly further domain-specific axioms. □

Axiomatizations of domains in the Fluent Calculus serve a variety of purposes.

- Entailing statements about situations, a domain theory can be used to predict the outcome of given action sequences. Our axiomatization of the delivery robot, for instance, entails

$$(\exists x)\, Holds(Carries(x), Do(Pickup(x), Do(Enter(Office(Alice)), Do(Open(D12), S_0))))$$

  by which a sequence of three actions is predicted to execute the task of picking up something in Alice's office.

- Entailing relational statements among situations, a domain theory can be used to explain observations. For instance, this is a valid implication in our example domain:

$$\begin{array}{l} Poss(Enter(R404), Do(Go(DA4), Do(Enter(Alley), Do(Go(DA1), S_0)))) \\ \quad \supset \neg Holds(Closed(DA4), S_0) \end{array} \tag{13}$$

  It says that if the robot was able to enter $R404$ right after walking to $DA4$, then that door must have been open.

- Finally, the planning problem can be modeled as the problem of finding a situation in which certain goal conditions are met. Our axiomatization of the delivery robot, for instance, entails that

  $$(\exists s)\,\neg(\exists r_1, r_2, x)\, \mathit{Holds}(\mathit{Request}(r_1, x, r_2), s)$$

  A constructive proof of this statement should yield a plan by which all given requests are met. A correct solution is,

  $$\begin{array}{l} s/\mathit{Do}(\mathit{Drop}(x), \mathit{Do}(\mathit{Enter}(\mathit{R404}), \mathit{Do}(\mathit{Open}(\mathit{DA4}), \mathit{Do}(\mathit{Go}(\mathit{DA4}), \\ \quad \mathit{Do}(\mathit{Enter}(\mathit{Alley}), \mathit{Do}(\mathit{Go}(\mathit{DA1}), \mathit{Do}(\mathit{Enter}(\mathit{R401}), \mathit{Do}(\mathit{Pickup}(x), \\ \quad\quad \mathit{Do}(\mathit{Enter}(\mathit{R402}), \mathit{Do}(\mathit{Open}(\mathit{D12}), S_0)))))))))) \end{array} \tag{14}$$

  Notice that in order to be entailed, the plan must be successful under any circumstances, that is, under any initial state of affairs which is consistent with our partial knowledge of it (cf. (4)). This is achieved only by letting the robot leave *R402* through *R401* and by having it open *DA4*.

Meta-statements about non-achievability of goals[6] can be proved with the help of the foundational axioms on situations, which include induction on situations, introduced in [53] in the context of the Situation Calculus.

4 FLUX—A Constraint Logic Programming Implementation

FLUX—the Fluent Calculus Executor—is based on the logic programming paradigm [24] with constraints [23]. FLUX is distinguished by a conceptually simple and computationally efficient way of dealing with incomplete states. An open state description with finitely many fluents known to hold, $z = f_1 \circ \ldots \circ f_n \circ z_1$ $(n \geq 0)$, is encoded by a list whose tail is a variable:

```
Z = [F1,...,Fn | Z1]                                        (15)
```

In addition, the fact that a certain fluent does not hold in a state, $\neg\mathit{Holds}(f, z)$, is encoded as constraint of the form

```
not_holds(F, Z)
```

Due to the incompleteness of state descriptions, a special constraint handling mechanism is needed which stores constraints in the background and checks their satisfiability whenever new bindings, e.g., for the tail variable in (15), become effective.

4.1 Constraint Handling Rules

Prolog systems with Constraint Handling Rules [11] support high-level programming of constraint solvers. Constraints are processed on the basis of declarative rules of the following kind.

Definition 9 Consider a signature for constraints, consisting of predicate and function symbols plus variables, which includes the special constraint `false`. A *constraint handling rule* (*CHR*) is an expression of the form

```
H1,...,Hm <=> G1,...,Gk | B1,...,Bn.                        (16)
```

where

[6] An example would be to prove that the robot can never solve a request from or to room *R403* if doors *DA3* and *D34* are closed.

- the *head* $H_1, \ldots, H_m$ are constraints ($m \geq 1$);
- the *guard* $G_1, \ldots, G_k$ are Prolog literals ($k \geq 0$);
- the *body* $B_1, \ldots, B_n$ are constraints ($n \geq 0$).

An empty guard is omitted; the empty body is denoted by `true`. □

As an example, consider the constraint $\mathit{Neq}(f_1, f_2)$ with the intended meaning that the arguments are two unequal terms. Suitable CHRs for this constraint are,

```
neq(F,F) <=> false.
neq(F1,F2) <=> \+ F1=F2 | true.
```
(17)

Informally speaking, the first rule causes $\mathit{Neq}(f_1, f_2)$ to fail if the two arguments are equal, whereas by the second rule, $\mathit{Neq}(f_1, f_2)$ is solved if the two arguments cannot be unified. In case of insufficiently instantiated arguments, neither of the two rules applies and the evaluation of the constraint is delayed.

The declarative interpretation of a CHR of the form (16) is given by the formula

$$(\forall \vec{x})\,(\,G_1 \wedge \ldots \wedge G_k \,\supset\, [H_1 \wedge \ldots \wedge H_m \,\equiv\, (\exists \vec{y})\,(B_1 \wedge \ldots \wedge B_n)]\,) \tag{18}$$

where $\vec{x}$ are the variables in both guard and head and $\vec{y}$ are the variables which additionally occur in the body. Thus the two rules in (17), for instance, mean the following, which shows that the rules agree with the intuition behind the constraint Neq:

$$\mathit{True} \supset [\,\mathit{Neq}(f, f) \equiv \mathit{False}\,]$$
$$f_1 \neq f_2 \supset [\,\mathit{Neq}(f_1, f_2) \equiv \mathit{True}\,]$$

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 thereafter be derived *without binding variables in the head*, then the constraints of the head are replaced by the constraints of the body. More formally, consider a set of constraints $\mathcal{C}$ and suppose there exists a CHR (16) along with two substitutions σ and θ such that

1. $H_1\sigma, \ldots, H_m\sigma \in \mathcal{C}$;
2. θ is a computed answer substitution for $G_1\sigma, \ldots, G_k\sigma$;
3. $H_i\sigma\theta = H_i\sigma$ (for $i = 1, \ldots, m$).

Then a transition is possible from $\mathcal{C}$ to $(\mathcal{C} \setminus \{H_1\sigma, \ldots, H_m\sigma\}) \cup \{B_1\sigma\theta, \ldots, B_n\sigma\theta\}$.

For example, the constraint $\mathit{Neq}(\mathit{Closed}(x), \mathit{Closed}(x))$ evaluates to False according to the first one of our CHRs in (17) whereas $\mathit{Neq}(\mathit{Closed}(x), \mathit{InRoom}(y))$ leads to the empty constraint store following our second rule. The constraint $\mathit{Neq}(\mathit{Closed}(x), \mathit{Closed}(\mathit{DA1}))$, on the other hand, cannot be rewritten by either rule because the head of the first rule does not match while the instantiated guard $\backslash\!+ \mathit{Closed}(x) = \mathit{Closed}(\mathit{DA1})$ of the second rule fails. Hence the constraint is kept in store for later evaluation.

The basic computation mechanism for logic programs with CHRs is standard SLDNF-resolution (see, e.g., [35]). Constraints that are selected in the course of a derivation are added to the constraint store. At any stage, the store is processed using the given CHRs until no further

```
handler fluent.
constraints neq/2, neq_all/3, not_holds/2, not_holds_all/3, duplicate_free/1.

neq(F,F) <=> false.
neq(F1,F2) <=> \+ F1=F2 | true.

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

neq_all(X,F1,F2) <=> copy_term_vars(X,F1,F), F=F2 | false.
neq_all(X,F1,F2) <=> \+ F1=F2 | true.

not_holds_all(X,F,[F1|Z]) <=> neq_all(X,F,F1), not_holds_all(X,F,Z).
not_holds_all(_,_,[]) <=> true.

duplicate_free([F|Z]) <=> not_holds(F,Z), duplicate_free(Z).
duplicate_free([]) <=> true.

not_holds_all(X,F1,Z) \ not_holds(F2,Z) <=> copy_term_vars(X,F1,F), F=F2 | true.
not_holds_all(X1,F1,Z) \ not_holds_all(_,F2,Z) <=> copy_term_vars(X1,F1,F), F=F2
                                                    | true.
```

Figure 3: The module `fluent.chr` contains the fundamental CHRs for FLUX.[7]

transition applies. If the special constraint "`false`" is introduced into the constraint store, then the entire derivation fails. If a derivation is successful, then the computed answer substitution is accompanied by all constraints that have remained in the store. The declarative reading of such an answer is that the instantiated query is entailed under the provision that the constraints hold [23].

4.2 Constraint Handling Rules for Flux

The complete constraint handling core of FLUX is shown in Figure 3.

4.2.1 NotHolds/2

Representing states by lists, the fundamental constraint in FLUX, $\mathit{NotHolds}$, requires a list to not contain a particular element. E.g., $\mathit{NotHolds}(\mathit{InRoom}(x), [\mathit{InRoom}(\mathit{Alley}), \mathit{AtDoor}(y) \,|\, z])$ yields the pending constraints $\mathit{Neq}(\mathit{InRoom}(x), \mathit{InRoom}(\mathit{Alley}))$ and $\mathit{NotHolds}(\mathit{InRoom}(x), z)$. These constraints may fail later, e.g., if x becomes Alley or if z is bound to $[\mathit{InRoom}(x) \,|\, z']$.

The CHR for this constraint can be justified, on the basis of its declarative interpretation (cf. (18)), by the foundational axioms of the Fluent Calculus, as the following proposition shows.

Proposition 10 $\mathcal{F}_{state}$ *entails,*

$$\neg \mathit{Holds}(f, f_1 \circ z) \;\equiv\; f \neq f_1 \wedge \neg \mathit{Holds}(f, z)$$

Proof: We prove that $\mathit{Holds}(f, f_1 \circ z) \;\equiv\; f = f_1 \vee \mathit{Holds}(f, z)$:

"$\Rightarrow$": Follows by the decomposition axiom (3).

"$\Leftarrow$": If $f = f_1$, then $f_1 \circ z = f \circ z$, hence $Holds(f, f_1 \circ z)$. Likewise, if $Holds(f, z)$, then $z = f \circ z'$ for some z', hence $f_1 \circ z = f_1 \circ f \circ z'$, hence $Holds(f, f_1 \circ z)$.

■

4.2.2 NotHoldsAll/3

Variables in constraints are generally treated as existentially quantified. To model universal quantification in negative *Holds* statements, e.g., as in $(\forall y)\, \neg Holds(Carries(y), z)$, the constraint $NotHoldsAll([x_1, \ldots, x_n], f, z)$ encodes the formula $(\forall \vec{x})\, \neg Holds(f, z)$. The auxiliary constraint $NeqAll(\vec{x}, f_1, f_2)$ denotes that no instance of f_1 wrt. variables $\vec{x}$ equals f_2. The Eclipse built-in $CopyTermVars(\vec{x}, f_1, f)$ used in the first CHR for this constraint means that f is a variant of f_1 in which all variables of list $\vec{x}$ have been renamed. Hence the guard succeeds without binding variables in the head just in case $f_1\sigma = f_2$ for some σ whose domain is a subset of $\vec{x}$. E.g., the constraint $NotHoldsAll([r, x], Request(r, x, R402), [Request(R401, x', r') \mid z])$ succeeds with $NeqAll([r, x], Request(r, x, R402), Request(R401, x', r'))$ among the pending constraints. The latter is solved if, say, r' becomes $R404$ whereas failure occurs if r' is bound to $R402$.

4.2.3 DuplicateFree/2

The third and final foundational constraint of FLUX is used to prevent multiple occurrence of fluents in lists representing a state. E.g., processing $DuplicateFree([Closed(DA1), Closed(x) \mid z])$ yields the four delayed constraints $Neq(Closed(DA1), Closed(x))$, $NotHolds(Closed(DA1), z)$, $NotHolds(Closed(x), z)$, and $DuplicateFree(z)$.

4.3 The Basic Flux Language

Building on the constraint solving core, the basic FLUX system shown in Figure 4 consists of further clauses reflecting the algebraic foundation of the Fluent Calculus. The program culminates in a definition for updating states by which the inferential Frame Problem is solved.

4.3.1 Holds/2

The two clauses defining this standard predicate are justified by

$$\mathcal{F}_{state} \models Holds(f, z) \subset (\exists z_1)\, z = f \circ z_1 \vee (\exists f_1, z_1)\, (z = f_1 \circ z_1 \wedge Holds(f, z_1))$$

which follows from Proposition 10. The Prolog built-in $NonVar(z)$ used in the second clause avoids non-terminating recursion by making sure that the state argument z is reducible.

A query of successive *Holds* atoms all appealing to the same state variable, results in an incomplete state description of the form (15). In this way, FLUX expands a state specification into a representation of the form (11), which enables a solution to the inferential Frame Problem.

If the arguments of fluent terms are not fully instantiated, then there may be several ways of having a fluent hold in a state. E.g., the query $Holds(Closed(DA1), z_0)$, $Holds(Closed(x), z_0)$ admits the two answers $\{z_0/[Closed(DA1) \mid z], x/DA1\}$ and $\{z_0/[Closed(DA1), Closed(x) \mid z]\}$. (In the latter case, constraint $DuplicateFree(z_0)$ would ensure $Neq(Closed(DA1), Closed(x))$.)

[7] The last two rules are used to remove subsumed constraints. They follow the syntax `H1 \ H2 <=> G | B`, which is an abbreviation for `H1,H2 <=> G | H1,B`.

```
:- lib(chr).
:- chr2pl(fluent), [fluent].

holds(F, [F|_]).
holds(F, Z) :- nonvar(Z), Z=[F1|Z1], \+ F==F1, holds(F, Z1).

holds(F, [F|Z], Z).
holds(F, Z, [F1|Zp]) :- nonvar(Z), Z=[F1|Z1], \+ F==F1, holds(F, Z1, Zp).

equal(Z1, Z2) :- (var(Z1) ; var(Z2)), Z1=Z2.
equal(Z1, Z2) :-
   nonvar(Z1), nonvar(Z2), ( Z1=[F|Z3], holds(F, Z2, Z4), equal(Z3, Z4) ;
                             Z1=[], Z2=[] ).

plus(Z, [], Z).
plus(Z, [F|Fs], Zp) :- (not_holds(F, Z), Z1=[F|Z] ; holds(F, Z), Z1=Z),
                       plus(Z1, Fs, Zp).

minus(Z, [], Z).
minus(Z, [F|Fs], Zp) :- (holds(F, Z, Z1) ; not_holds(F, Z), Z1=Z),
                        minus(Z1, Fs, Zp).

update(Z1, ThetaP, ThetaN, Z2) :-
   plus(Z1, ThetaP, Z), minus(Z, ThetaN, Zp), equal(Zp, Z2).
```

Figure 4: The core of `flux.pl`, into which also the fundamental CHRs are loaded.

4.3.2 Holds/3

Predicate $\mathit{Holds}(f, z, z_1)$ means $\mathit{Holds}(f, z) \wedge z_1 \circ f = z \wedge \neg \mathit{Holds}(f, z_1)$. The two clauses are justified under the assumption that the list representing state z does not contain multiple occurrences of f.

4.3.3 Equal/2

Predicate $\mathit{Equal}(z_1, z_2)$ means that the arguments denote equal states. The Prolog built-ins used in the clauses defining this predicate ensure that the fluent-wise comparison is performed only in case neither of the arguments is a variable. The second clause is justified by this implication, which is self-evident:

$$z_1 = z_2 \subset z_1 = f \circ z_3 \wedge z_2 = f \circ z_4 \wedge z_3 = z_4$$

4.3.4 Update/4

Predicate $\mathit{Update}(z_1, \vartheta^+, \vartheta^-, z_2)$ means $z_2 = z_1 \circ \vartheta^+ - \vartheta^-$, that is, update of state z_1 to state z_2 by means of positive and negative, resp., effects ϑ^+, ϑ^-. Its clause uses the auxiliary predicates $\mathit{Plus}/3$ and $\mathit{Minus}/3$, whose definitions preserve the property of lists not containing multiple occurrences of fluents.

The clauses defining *Plus* are justified by

$$\begin{aligned} & z \circ \emptyset = \emptyset \\ & z \circ f \circ \vartheta = z' \subset (\exists z_1)\,(\, [\neg \mathit{Holds}(f, z) \supset z_1 = f \circ z] \wedge \\ & \qquad\qquad\qquad\qquad [\mathit{Holds}(f, z) \supset z_1 = z] \wedge \\ & \qquad\qquad\qquad\qquad z' = z_1 \circ \vartheta\,) \end{aligned}$$

which follows from ACI1.

The clauses defining *Minus* are justified by

$$\begin{aligned} & z - \emptyset = \emptyset \\ & z - f \circ \vartheta = z' \subset (\exists z_1)\,(\, [\mathit{Holds}(f, z) \supset z_1 \circ f = z \wedge \neg \mathit{Holds}(f, z_1)] \wedge \\ & \qquad\qquad\qquad\qquad [\neg \mathit{Holds}(f, z) \supset z_1 = z] \wedge \\ & \qquad\qquad\qquad\qquad z' = z_1 - \vartheta\,) \end{aligned}$$

which follows from ACI1 and (9), (8).

The clause defining *Update* is justified by

$$z_2 = z_1 \circ \vartheta^+ - \vartheta^- \subset (\exists z, z')\,(z = z_1 \circ \vartheta^+ \wedge z' = z - \vartheta^- \wedge z' = z_2)$$

The definition of *Update* implements the solution to the inferential Frame Problem of the Fluent Calculus: Provided that argument z_2 is not substituted by an explicit list description of a state when using the head in a resolution step, z_2 can be directly bound, via the first clause for *Equal*, to the result of addition and removal of positive and negative effects. These operations leave all unaffected fluents in the list. Moreover, the original and the updated state share their tail variable. Hence, all knowledge of the form that a fluent does not hold, represented by delayed constraints on this tail variable, applies to the successor state just as well.

It is important to note, however, that if variable z_2 in the clause defining *Update* is already bound to an explicit description of a state, then the definition for *Equal* requires element-wise comparison of this list and the update result. Hence, the above definition of state update should be used in a progression-like reasoning mode, which means to start with an initial state and to successively infer the result of action sequences. For the reverse operation of regression, a different but analogous clause defining *Update* is more suited in which z_1 is inferred from a given open list z_2.

4.4 Programming in Flux

Based on the general FLUX module of Figures 3 and 4, domain axiomatizations in the simple Fluent Calculus can be encoded according to the following programming scheme. (See Appendices A.1 and A.2 for the complete FLUX specification of our example of a delivery robot.)

Prolog in general and our CHRs for the constraint *Neq* in particular take two terms as unequal whenever they are not unifiable. In so doing, FLUX assumes universal uniqueness of names.

The state constraints of a domain are summarized in a clause of the form $\mathit{Consistent}(z) \leftarrow \Gamma_1(z), \ldots, \Gamma_n(z)$. Some state constraints require the introduction of domain-specific CHRs. Examples are the last three of the axioms (7) for our delivery scenario: They are encoded by the constraints $\mathit{AtDoorUnique}(z)$, $\mathit{DoorOfRoom}(r, z)$, and $\mathit{RequestUnique}(z)$ along with the CHRs shown in Appendix A.1.

A simple action precondition axiom is encoded as $\mathit{Poss}(A(\vec{x}), z) \leftarrow \Pi_A(\vec{x}, z)$ along with $\mathit{NotPoss}(A(\vec{x}), z) \leftarrow \neg\Pi_A(\vec{x}, z)$. The latter is necessary because in the presence of incomplete state knowledge, non-executability cannot simply be inferred by negation-as-failure. A simple state update axiom is encoded as $\mathit{StateUpdate}(z_1, A(\vec{x}), z_2) \leftarrow \Delta(\vec{x}, z_1),\ \mathit{Update}(z_1, \vartheta^+, \vartheta^-, z_2)$. Since it is often necessary to verify action preconditions independently from applying a state update axiom, its body does not include this check. Rather it is assumed that the axiom is applied only in a context where the preconditions hold (cf. (19) below).

Based on a specification of preconditions and effects for each single action, the result of sequences of actions may be inferred using the following clauses, which define the predicate $\mathit{DO}(s, z_0, z)$ with the intended meaning that s is a (possibly empty) list of actions $[a_1, \ldots, a_n]$ such that $z_0 = \mathit{State}(S_0)$ and $z = \mathit{State}(\mathit{Do}(a_n, \ldots, \mathit{Do}(a_1, S_0) \ldots))$:

```
do([], Z, Z).
do([A|S], Z0, Z) :- poss(A, Z0), state_update(Z0, A, Z1), do(S, Z1, Z).
```
(19)

For the sake of efficiency, satisfaction of state constraints is not verified within this definition. Rather the state update axioms are assumed to preserve state consistency. It then suffices to stipulate consistency, as well as freeness of duplicates, of initial states only, assuming clauses (19) are used for progressing a state z_0.

It is important to realize that successful derivations of queries with incomplete states indicate mere satisfiability. For example, the query

$$\begin{array}{l}\mathit{Holds}(\mathit{InRoom}(\mathit{R401}), z_0),\ \mathit{Holds}(\mathit{AtDoor}(\mathit{D12}), z_0),\ \mathit{Consistent}(z_0),\ \mathit{DuplicateFree}(z_0),\\ \mathit{DO}([a], z_0, z_1),\ \mathit{Holds}(\mathit{InRoom}(\mathit{R402}), z_1),\ \mathit{Holds}(\mathit{Carries}(\mathit{Projector}), z_1)\end{array}$$

has a successful derivation, whose answer substitution includes the bindings $a/\mathit{Enter}(\mathit{R402})$ and $z_0/[\mathit{InRoom}(\mathit{R401}), \mathit{AtDoor}(\mathit{D12}), \mathit{Carries}(\mathit{Projector}) \,|\, z]$ and is accompanied by the constraint $\mathit{NotHolds}(\mathit{Closed}(\mathit{D12}), z)$. The initial state has thus been 'tuned' in view of the goal by stipulating that the robot carries the projector already and that door $\mathit{D12}$ is open. Proving that a statement is a logical consequence therefore requires to ensure that the negation of the statement is unsatisfiable. In turn, unsatisfiability can be easily expressed using Prolog's negation-as-failure. Recall, for instance, implication (13) at the end of Section 3. To prove that door $\mathit{DA4}$ must have been open, we ask whether assuming the contrary is unsatisfiable:

$$\begin{array}{l}\mathit{Holds}(\mathit{InRoom}(\mathit{R401}), z_0),\ \mathit{NotHolds}(\mathit{Closed}(\mathit{DA1}), z_0),\ \mathit{Consistent}(z_0),\ \mathit{DuplicateFree}(z_0),\\ \mathit{DO}([\mathit{Go}(\mathit{DA1}), \mathit{Enter}(\mathit{Alley}), \mathit{Go}(\mathit{DA4})], z_0, z_1),\\ \mathit{Poss}(\mathit{Enter}(\mathit{R404}), z_1), \backslash\!+\mathit{Holds}(\mathit{Closed}(\mathit{DA4}), z_0)\end{array}$$

This query is successful if posed to our program of Appendix A.2.

A similar double-check needs to be performed in the context of planning problems. Having established a plan which satisfies the goal, this action sequence should be verified against both not achieving the goal and not being executable at all. The latter means the existence of a model such that at some point during the execution of the plan, the action to be performed next fails.[8] This is encoded in the following clause, which defines the predicate $\mathit{NonExecutable}(s, z)$ where s is a sequence of actions and z a state:

```
non_executable([A|S], Z0) :-
  not_poss(A, Z0) ;
  poss(A, Z0), state_update(Z0, A, Z1), non_executable(S, Z1).
```
(20)

[8]In case of nondeterministic actions, the unsatisfiability check also concerns alternative outcomes of such actions as a cause for a plan to fail; see Section 5.

As an example, consider the following specification of the initial state of a planning problem (c.f. (4)):

$$\begin{array}{rl} \mathit{Init}(z_0, x) \leftarrow & \mathit{Holds}(\mathit{InRoom}(\mathit{R401}), z_0),\ \mathit{Holds}(\mathit{AtDoor}(\mathit{D12}), z_0),\ \mathit{Holds}(\mathit{Closed}(\mathit{D12}), z_0), \\ & \mathit{Holds}(\mathit{HasKeyCode}(\mathit{D12}), z_0, z'),\ \mathit{Holds}(\mathit{HasKeyCode}(\mathit{DA4}), z', z''), \\ & \mathit{NotHoldsAll}([d], \mathit{HasKeyCode}(d), z''), \\ & \mathit{Holds}(\mathit{Request}(\mathit{R402}, x, \mathit{R404}), z_0, z'''), \\ & \mathit{NotHoldsAll}([r_1, y, r_2], \mathit{Request}(r_1, y, r_2), z'''), \\ & \mathit{NotHolds}(\mathit{Closed}(\mathit{DA1}), z_0),\ \mathit{NotHoldsAll}([y], \mathit{Carries}(y), z_0), \\ & \mathit{Consistent}(z_0),\ \mathit{DuplicateFree}(z_0) \end{array}$$

Notice how we encode, for instance, the fact that the robot has two and no more key codes by two *Holds*/3 atoms followed by a *NotHoldsAll* constraint. The planning problem at the end of Section 3 can be formalized as the following query:

$$\begin{array}{l} \mathit{Init}(z_0, x),\ \mathit{DO}(s, z_0, z),\ \mathit{NotHoldsAll}([r_1, y, r_2], \mathit{Request}(r_1, y, r_2), z), \\ \mathit{Init}(z_0', x),\ \backslash{+}\mathit{NonExecutable}(s, z_0'),\ \backslash{+}(\mathit{DO}(s, z_0', z'),\ \mathit{Holds}(\mathit{Request}(r_1, y, r_2), z')) \end{array}$$

This query admits a successful derivation which yields a substitution for s that corresponds to the situation term of (14).

In Prolog systems, naive planning—which means to search the whole space of executable action sequences—requires to either restrict the search depth or to perform breadth-first search. For planning problems of practical size, it is essential to add heuristics to cut down considerably the search space. One of the most intuitive and expressive ways of doing so is by writing nondeterministic high-level robot programs [30], which can be easily adapted to FLUX. A further improvement regarding efficiency concerns the executability check of plans generated by satisfiability. Rather than performing it subsequently, each chosen action can immediately be verified to be necessarily possible in order to avoid unnecessary search. We will raise this issue in the context of knowledge and sensing actions (Section 6).

5 Nondeterministic Actions

Actions are nondeterministic if there are always several possible outcomes or if some general vagueness is involved with their effect. In this sense, actions in the simple Fluent Calculus are deterministic because their effect is determined solely by the fluent values of the current state. A straightforward generalization of simple state update axioms uses disjunction, along the lines of [33, 67], and existential quantification as means to express uncertainty about effects.

Definition 11 Let A be a function symbol with range ACTION. A *simple disjunctive state update axiom* for A is of the form

$$\begin{array}{rl} \mathit{Poss}(A(\vec{x}), s) \wedge \Delta(\vec{x}, \mathit{State}(s)) \supset & (\exists \vec{y}_1)\,(\mathit{State}(\mathit{Do}(A(\vec{x}), s)) = \mathit{State}(s) \circ \vartheta_1^+ - \vartheta_1^- \wedge \Theta_1) \\ & \vee \ldots \vee \\ & (\exists \vec{y}_n)\,(\mathit{State}(\mathit{Do}(A(\vec{x}), s)) = \mathit{State}(s) \circ \vartheta_n^+ - \vartheta_n^- \wedge \Theta_n) \end{array}$$

where $\Delta(\vec{x}, z)$ is a pure state formula in z, ϑ_i^+ and ϑ_i^- are fluent collections, and Θ_i is a first-order formula without terms of any reserved sort ($1 \leq i \leq n;\ n \geq 1$). □

Alternative outcomes of actions are thus modeled by a finite disjunction of possible equational relations between a state and its successor. Vagueness is modeled by existentially quantifying (and possibly restricting by sub-formula Θ) one or more parameters of the effects.

As an example for the first kind of nondeterminism, suppose that our delivery robot can perform the action of asking a person for access to the room she is sending it, provided that that room is adjacent to her office and that the connecting door is closed and the robot does not have the code. She will then either tell the robot the key code or open the door in question. Let $\mathit{Ask}\colon \textsc{person} \times \textsc{door} \mapsto \textsc{action}$ denote this action. Its precondition is defined by

$$\begin{aligned}\mathit{Poss}(\mathit{Ask}(p,d),z) \equiv\ & \mathit{Holds}(\mathit{InRoom}(\mathit{Office}(p)),z) \land \\ & (\exists d,r,x)\,(\,\mathit{Holds}(\mathit{Request}(\mathit{Office}(p),x,r),z) \land \mathit{Connects}(d,\mathit{Office}(p),r) \land \\ & \qquad \mathit{Holds}(\mathit{Closed}(d),z) \land \neg \mathit{Holds}(\mathit{HasKeyCode}(d),z)\,)\end{aligned}$$

The following disjunctive state update axiom specifies the uncertain outcome of performing an *Ask* action:

$$\begin{aligned}\mathit{Poss}(\mathit{Ask}(p,d),s) \supset\ & \mathit{State}(\mathit{Do}(\mathit{Ask}(p,d),z)) = \mathit{State}(s) \circ \mathit{HasKeyCode}(d) \\ & \lor \\ & \mathit{State}(\mathit{Do}(\mathit{Ask}(p,d),z)) = \mathit{State}(s) - \mathit{Closed}(d)\end{aligned}$$

As an example for the second kind of nondeterminism, suppose the exact physical position of the robot is modeled using a two-dimensional, real-valued coordinate system in conjunction with the fluent $\mathit{Position}\colon \textsc{real} \times \textsc{real} \mapsto \textsc{fluent}$. Consider the action $\mathit{Move}\colon \textsc{real} \times \textsc{real} \mapsto \textsc{action}$ of moving towards a specific point in space. Since the effectors of real robots will never be absolutely precise, the effect of this action is vague insofar as the actual resulting position of the robot differs from the position it is aiming at up to an uncertainty factor ς. This uncertainty is captured by the following state update axiom:

$$\begin{aligned}& \mathit{Poss}(\mathit{Move}(x,y),s) \land \mathit{Holds}(\mathit{Position}(x_0,y_0),s) \supset \\ & \quad (\exists x',y')\,(\,\mathit{State}(\mathit{Do}(\mathit{Move}(x,y),s)) = \mathit{State}(s) \circ \mathit{Position}(x',y') - \mathit{Position}(x_0,y_0) \\ & \qquad \land \sqrt{(x'-x)^2 + (y'-y)^2} \leq \varsigma\,)\end{aligned}$$

Planning problems in domains with nondeterministic actions require to find plans that solve the problem under any outcome of the intended actions. For example, our axiomatization of the delivery robot entails

$$\begin{aligned}& \mathit{Holds}(\mathit{InRoom}(\mathit{R402}),S_0) \land \mathit{Holds}(\mathit{AtDoor}(\mathit{D23}),S_0) \land \mathit{Holds}(\mathit{Closed}(\mathit{D23}),S_0) \\ & \land \mathit{Holds}(\mathit{Request}(\mathit{R402},\mathit{Projector},\mathit{R403}),S_0) \\ & \land \neg \mathit{Holds}(\mathit{HasKeyCode}(\mathit{D23}),S_0) \land \neg \mathit{Holds}(\mathit{Carries}(\mathit{Projector}),S_0) \\ & \quad \supset \neg \mathit{Holds}(\mathit{Request}(\mathit{R402},\mathit{Projector},\mathit{R403}),s)\end{aligned}$$

under the substitution

$$\begin{aligned}& s/\mathit{Do}(\mathit{Drop}(\mathit{Projector}),\mathit{Do}(\mathit{Enter}(\mathit{R403}), \\ & \quad \mathit{Do}(\mathit{Open}(\mathit{D23}),\mathit{Do}(\mathit{Ask}(\mathit{Alice},\mathit{D23}),\mathit{Do}(\mathit{Pickup}(\mathit{Projector}),S_0)))))\end{aligned} \tag{21}$$

Notice that omitting action $\mathit{Open}(\mathit{D23})$ from this situation would not yield a correct plan since the actual effect of asking Alice cannot be predicted with certainty. While this example of a planning problem with nondeterministic actions can thus be solved by a plain sequence of actions, a more elaborate concept of a plan is needed in general which allows a robot to condition its actions on the actual outcome of a nondeterministic action. We will raise this issue in the context of modeling knowledge and sensing actions (Section 6).

Nondeterministic Actions in Flux

State update axioms of the form of Definition 11 are encoded in FLUX by a clause

$$\begin{aligned} \mathit{StateUpdate}(z_1, A(\vec{x}), z_2) \leftarrow \Delta(z_1), (\ & \mathit{Update}(z_1, \vartheta_1^+, \vartheta_1^-, z_2),\ \Theta_1\ ; \\ & \ldots\ ; \\ & \mathit{Update}(z_1, \vartheta_n^+, \vartheta_n^-, z_2),\ \Theta_n\ ) \end{aligned}$$

For illustration, the program for the delivery robot of Appendix A.2 includes the formalization of our example action of asking for access.

As in the simple case, solving planning problems includes verifying that a generated plan does not admit a model in which it is not executable or does not achieve the goal. Correctness of a plan is thus established also as regards alternative outcomes of nondeterministic actions. Consider, for example, the initial specification

$$\begin{aligned} \mathit{Init}(z_0) \leftarrow\ & \mathit{Holds}(\mathit{InRoom}(\mathit{R402}), z_0),\ \mathit{Holds}(\mathit{AtDoor}(\mathit{D23}), z_0),\ \mathit{Holds}(\mathit{Closed}(\mathit{D23}), z_0), \\ & \mathit{Holds}(\mathit{Request}(\mathit{R402}, \mathit{Projector}, \mathit{R403}), z_0), \\ & \mathit{NotHolds}(\mathit{HasKeyCode}(\mathit{D23}), z_0),\ \mathit{NotHolds}(\mathit{Carries}(\mathit{Projector}), z_0), \\ & \mathit{Consistent}(z_0),\ \mathit{DuplicateFree}(z_0) \end{aligned}$$

The query

$$\begin{aligned} & \mathit{Init}(z_0),\ \mathit{DO}(s, z_0, z),\ \mathit{NotHolds}(\mathit{Request}(\mathit{R402}, \mathit{Projector}, \mathit{R403}), z), \\ & \mathit{Init}(z_0'),\ \backslash{+}\mathit{NonExecutable}(s, z_0'), \\ & \qquad \backslash{+}(\mathit{DO}(s, z_0', z'),\ \mathit{Holds}(\mathit{Request}(\mathit{R402}, \mathit{Projector}, \mathit{R403}), z')) \end{aligned}$$

admits a successful derivation which yields a substitution for s that corresponds to the situation term of (21).

6 Knowledge and Sensing

Autonomous, mobile robots often have to condition their actions on the state of their environment. As their knowledge of the world state is limited, robots are equipped with sensors for the purpose of acquiring information about the external world. The use of sensing actions is often an integral part of a successful plan, and in order to devise these plans robots need an explicit representation of what they know of a state and how sensing affects their knowledge [45].

Our delivery robot, for example, thus far performs an *Open* action whenever it does not know whether a door is closed or not, in order to guarantee that a subsequent *Enter* action be possible. Yet suppose a slightly different setting in which the robot does not have the key codes but the electronic door system is designed in such a way that by sending out a special identification code the robot can alter the state of any door. While in principle this should provide access to any room, the robot always needs to know if a particular door is currently closed in order to decide whether to send out the identification. This poses planning problems that go beyond the simple Fluent Calculus. To see why, consider the action $\mathit{SendId} : \mapsto \textsc{action}$, which shall be possible at any time (that is, $\mathit{Poss}(\mathit{SendId}, z) \equiv \mathit{True}$), along with these three state update axioms:

$$\begin{aligned} & \mathit{Poss}(\mathit{SendId}, s) \wedge \mathit{Holds}(\mathit{AtDoor}(d), s) \wedge \mathit{Holds}(\mathit{Closed}(d), s) \supset \\ & \quad \mathit{State}(\mathit{Do}(\mathit{SendId}, s)) = \mathit{State}(s) - \mathit{Closed}(d) \\ & \mathit{Poss}(\mathit{SendId}, s) \wedge \mathit{Holds}(\mathit{AtDoor}(d), s) \wedge \neg\mathit{Holds}(\mathit{Closed}(d), s) \supset \\ & \quad \mathit{State}(\mathit{Do}(\mathit{SendId}, s)) = \mathit{State}(s) \circ \mathit{Closed}(d) \\ & \mathit{Poss}(\mathit{SendId}, s) \wedge (\neg\exists d)\, \mathit{Holds}(\mathit{AtDoor}(d), s) \supset \mathit{State}(\mathit{Do}(\mathit{SendId}, s)) = \mathit{State}(s) \end{aligned} \tag{22}$$

Put in words, if the robot happens to be at a door which is closed (resp. open) then the door opens (resp. closes), otherwise nothing happens. Suppose, for instance, that the robot is in the alley at door $\mathit{DA3}$:

$$\mathit{Holds}(\mathit{InRoom}(\mathit{Alley}), S_0) \wedge \mathit{Holds}(\mathit{AtDoor}(\mathit{DA3}), S_0)$$

Since the state of $\mathit{DA3}$ is not known, there is no provably executable sequence of actions by which the robot is guaranteed to achieve the goal of being in room $\mathit{R403}$. For, simply performing $\mathit{Enter}(\mathit{R403})$ is not possible in case $\mathit{Holds}(\mathit{Closed}(\mathit{DA3}), S_0)$ while $\mathit{Enter}(\mathit{R403})$ after a SendId action is not possible in case $\neg \mathit{Holds}(\mathit{Closed}(\mathit{DA3}), S_0)$. Solving this problem thus requires the ability to sense the states of doors and to condition actions on the (unpredictable) outcome of sensing.

6.1 State Knowledge

From the perspective of a non-omniscient robot, there are always several possible states of the world, constrained only by what the robot currently knows. Whenever the value of a certain fluent is unknown to the robot, then both states in which it is true and states in which it is false are conceivable. Knowledge, on the other hand, emerges if all states considered possible satisfy a certain property.

In order to formally represent this notion of a possible state, the signature of the simple Fluent Calculus is extended by the predicate $\mathit{KState}(s, z)$, which denotes that in situation s the robot considers z to be a possible world state. An example is depicted in Figure 5, where at the beginning the robot knows it is in front of door $\mathit{DA3}$ but does not know the state of this door. Hence, only those states are considered possible which are in accordance with this knowledge. After sensing that the door is closed, the set of possible states shrinks due to the newly acquired information. Finally, after transmitting the identification code, the possible states are those in which door $\mathit{DA3}$ is now open, assuming that the robot is aware of the effect of its action.

With the help of the KState relation, we can define a property of the world state to be known in s iff the property holds in all possible states for s. To this end, the macro $\mathit{Knows}(\psi, s)$ is introduced where ψ is a so-called fluent formula:

Definition 12 A fluent formula is an expression composed of

1. non-variable terms of sort FLUENT and

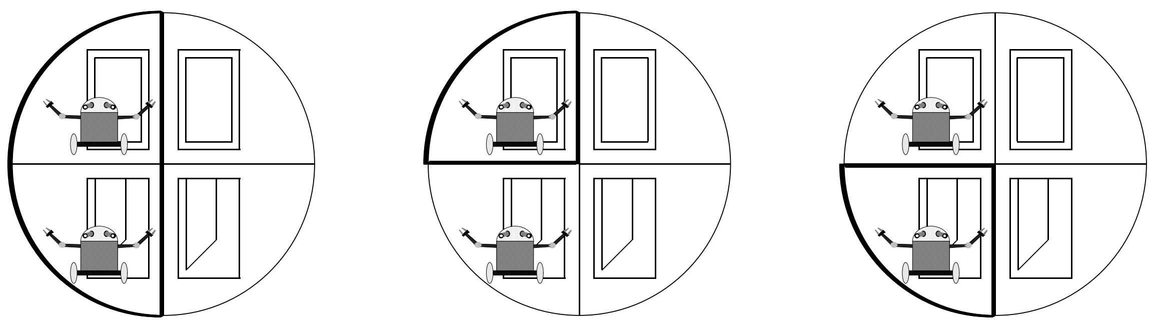
2. atoms without terms of any reserved sort

joined together with the standard logical connectives. □

The notion of a fluent formula is just syntactic sugar, which allows to conveniently express statements like, "The robot knows it is at some open door that leads to the alley." The macro is inductively defined as follows:

$$\mathit{Knows}(\psi, s) \stackrel{\text{def}}{=} (\forall z)\,(\mathit{KState}(s, z) \supset \mathit{HOLDS}(\psi, z))$$

$$\begin{aligned}
\mathit{HOLDS}(f, z) &\stackrel{\text{def}}{=} \mathit{Holds}(f, z) && \text{if } f \text{ is a FLUENT} \\
\mathit{HOLDS}(A, z) &\stackrel{\text{def}}{=} A && \text{if } A \text{ is an atom} \\
\mathit{HOLDS}(\neg\psi, z) &\stackrel{\text{def}}{=} \neg \mathit{HOLDS}(\psi, z) \\
\mathit{HOLDS}(\psi_1 \wedge \psi_2, z) &\stackrel{\text{def}}{=} \mathit{HOLDS}(\psi_1, z) \wedge \mathit{HOLDS}(\psi_2, z) \\
\mathit{HOLDS}((\forall x)\,\psi, z) &\stackrel{\text{def}}{=} (\forall x)\,\mathit{HOLDS}(\psi, z)
\end{aligned} \tag{23}$$



$KState(S_0, z) \supset$
$\quad Holds(AtDoor(DA3), z)$

$KState(S_1, z) \supset$
$\quad Holds(AtDoor(DA3), z)$
$\quad \wedge\, Holds(Closed(DA3), z)$

$KState(S_2, z) \supset$
$\quad Holds(AtDoor(DA3), z)$
$\quad \wedge\, \neg Holds(Closed(DA3), z)$

Figure 5: The set of states in our delivery world can be divided into four categories, namely, those in which the robot is at $DA3$ with this door being closed (upper left quarter), those in which the door is closed but the robot is elsewhere (upper right quarter), and those two in which $DA3$ is open and the robot is present (lower left quarter) and absent (lower right quarter), resp. The three pictures represent different mental states, characterizing, resp., the situation S_0 prior to sensing whether the door is closed, the situation $S_1 = Do(Sense(Closed(DA3)), S_0)$ after sensing that the door is closed, and the situation $S_2 = Do(SendId, S_1)$ after further sending the identification code to change the state of the door.

(Likewise for the other connectives.) For example, $Knows((\exists d)\,(AtDoor(d) \wedge \neg Closed(d) \wedge (\exists r)\, Connects(d, r, Alley)), S_0)$ expands into $(\forall z)\,(KState(S_0, z) \supset (\exists d)\,(Holds(AtDoor(d), z) \wedge \neg Holds(Closed(d), z) \wedge (\exists r)\, Connects(d, r, Alley)))$.

It is worth mentioning that by different quantifier structures it can be distinguished between knowledge *de re* and mere knowledge *de dicto*; for instance, knowing of some closed door—$(\exists d)\, Knows(Closed(d), s)$—implies to know some door is closed—$Knows((\exists d)\, Closed(d), s)$—but not vice versa. It is convenient to also introduce the macro $Kwhether(\psi, s)$, indicating whether the truth value of a statement is known:

$$Kwhether(\psi, s) \stackrel{\text{def}}{=} Knows(\psi, s) \vee Knows(\neg\psi, s) \tag{24}$$

With an atomic foundational axiom $\mathcal{F}_{knows}$ we establish the basic property of knowledge to be true:

$$KState(s, State(s))$$

Hence, by requiring that the actual state shall always be among those that are considered possible, every statement known by the robot indeed holds in the actual world. A further consequence of the foundational axiom is consistency of mental states, as the following proposition shows.

Proposition 13 $\quad \mathcal{F}_{knows} \models \neg(\exists f, s)\,(Knows(f, s) \wedge Knows(\neg f, s))$.

Proof: Suppose $Knows(f, s) \wedge Knows(\neg f, s)$ for some f and s, that is,

$$(\forall z)\,(KState(s, z) \supset Holds(f, z)) \;\wedge\; (\forall z)\,(KState(s, z) \supset \neg Holds(f, z))$$

This implies $(\forall z)\,\neg KState(s, z)$, which contradicts $\mathcal{F}_{knows}$. ■

The concept of possible states leaves open the possibility to specify robots that have arbitrarily limited knowledge of state constraints. To stipulate that the robot is aware of a particular constraint $\Gamma(z)$, the axiom $\mathit{KState}(s,z) \supset \Gamma(z)$ needs to be added.

6.2 Only Knowing

When specifying the initial knowledge of a robot, one usually wishes to make some completeness assumption to be able to derive statements also as to what a robot does not know. With the help of the fundamental relation KState, this notion of "only knowing" (so named by [26]) can be modeled by a specification of the form

$$\mathit{KState}(\sigma, z) \equiv \Sigma(z)$$

Put in words, in situation σ the robot knows Σ, and this is all it knows since every state is possible if only it satisfies Σ. Assuming that knowledge is given in this form, a property of the world state can be defined as unknown just in case there is at least one possible state in which the statement is true and one in which it is false:

$$\mathit{Unknown}(\psi, s) \stackrel{\text{def}}{=} (\exists z_1)\,(\mathit{KState}(s,z_1) \wedge \mathit{HOLDS}(\psi, z_1)) \wedge (\exists z_2)\,(\mathit{KState}(s,z_2) \wedge \mathit{HOLDS}(\neg\psi, z_2))$$

(where ψ is a fluent formula). However, this definition alone is insufficient because models may not contain enough states to ensure that a certain proposition can go both ways. Suppose, for example, the robot only knows that no year before 2001 lies in the future. It should then be unknown to the robot whether all other years are in the future. Yet this does not follow:

Observation 14 *Let σ be a constant of sort* SIT*, then $\mathcal{F}_{state}$ plus*

$$\mathit{KState}(\sigma, z) \equiv (\forall n)\,(n < 2001 \supset \neg \mathit{Holds}(\mathit{FutureYear}(n), z)) \tag{25}$$

and $\neg \mathit{Unknown}((\forall n)\,(n \geq 2001 \supset \mathit{FutureYear}(n)), \sigma)$ is satisfiable.

Proof: We construct a model $\mathcal{M}$ as follows. Let the domain elements of sort FLUENT be all singleton sets of the form $\{\mathit{FutureYear}(n)\}$ and let the domain elements of sort STATE be all *finite* sets of elements of the form $\mathit{FutureYear}(n)$, where n is a natural number. Let $\emptyset$ and $\circ$ be interpreted by the empty set and the union operation, resp. Then $\mathcal{M}$ is a model of $\mathcal{F}_{state}$ (cf. the proof for Theorem 3). Furthermore, let $\mathit{KState}^{\mathcal{M}}$ consist of all pairs $(\sigma^{\mathcal{M}}, \mathcal{Z})$ such that $\mathcal{Z}$ is a finite (possibly empty) set containing only fluents $\mathit{FutureYear}(n)$ with $n \geq 2001$. Then $\mathcal{M}$ is a model of (25). Finiteness of all states implies

$$\mathcal{M} \models \neg(\exists z)(\mathit{KState}(\sigma, z) \wedge (\forall n)\,(n \geq 2001 \supset \mathit{Holds}(\mathit{FutureYear}(n), z))$$

which proves the claim. ■

To achieve the intended result, a second-order axiom is needed by which is guaranteed the existence of sufficiently many states. Let $F_1, \ldots, F_n$ be all functions of a domain with range FLUENT, then the following axiom stipulates that for all truth-value distributions a corresponding state exists:

$$(\forall \Phi_1, \ldots, \Phi_n)(\exists z)(\forall \vec{x}_1, \ldots, \vec{x}_n) \left\{ \begin{array}{c} [\,\Phi_1(\vec{x}_1) \equiv \mathit{Holds}(F_1(\vec{x}_1), z)\,] \\ \wedge \ldots \wedge \\ [\,\Phi_n(\vec{x}_n) \equiv \mathit{Holds}(F_n(\vec{x}_n), z)\,] \end{array} \right\} \tag{26}$$

This axiom is consistent with our foundational axioms on states, as the following theorem shows.

Theorem 15 $\mathcal{F}_{state} \cup \{(26)\}$ *is consistent.*

Proof: We construct a model $\mathcal{M}$ as follows. Let the domain elements of sort FLUENT be all singleton sets over an arbitrary, non-empty set F and let the domain elements of sort STATE be all sets over F. Let $\emptyset$ and $\circ$ be interpreted by the empty set and the union operation, resp. Then $\mathcal{M}$ is a model of $\mathcal{F}_{state}$ (cf. the proof for Theorem 3). Furthermore, for each truth-value distribution of fluents, there exists a set containing just the true fluents; hence, $\mathcal{M}$ is also a model of (26). ∎

It is worth noting that, unlike the approach of [26, 27], no presuppositions are made regarding the domain objects inhabiting a world. As a consequence, universally quantified statements can be proved unknown only if the domain axioms stipulate the existence of sufficiently many objects of a certain sort. Consider, for example, the fluent $Out\colon \text{PERSON} \mapsto \text{FLUENT}$, denoting whether a person is out of office, along with

$$KState(S_0, z) \equiv Holds(Out(Alice), z) \wedge Holds(Out(Bob), z)$$

It does not follow, then, that $Unknown((\forall p)\, Out(p), S_0)$. For we might live in a world with Alice and Bob being the only inhabitants, in which case knowing that the two are out suffices to know that everybody is out; hence, the latter is not necessarily unknown.

For practical purposes, it is important to note that in settings in which consistent states consist of finitely many fluents only, the second-order axiom (26) can be omitted since all necessary states can be constructed by connecting fluents via "$\circ$". As an example, suppose the robot knows that in situation S_0 it is in the alley and at least one of $DA3$ and $DA4$ is not closed, but it is unknown which of the two. This combination of knowledge with ignorance is formally specified by,

$$\begin{array}{l} KState(S_0, z) \equiv \\ \quad Holds(InRoom(Alley), z) \wedge [\,\neg Holds(Closed(DA3), z) \vee \neg Holds(Closed(DA4), z)\,] \end{array} \tag{27}$$

Then the robot knows that some door is open, that is, $Knows((\exists d)\, \neg Closed(d), S_0)$. This can be easily seen from the macro expansion

$$(\forall z)\,(KState(S_0, z) \supset (\exists d)\, \neg Holds(Closed(d), z))$$

which follows directly from (27). However, the robot does not know of any particular open door, that is, $(\forall d)\; Unknown(Closed(d), S_0)$. This follows from the macro expansion

$$\begin{array}{l} (\forall d)\,(\,(\exists z_1)\,(KState(S_0, z_1) \wedge Holds(Closed(d), z_1)) \wedge \\ \qquad\qquad (\exists z_2)\,(KState(S_0, z_2) \wedge \neg Holds(Closed(d), z_2))\,) \end{array}$$

For in case $d = DA3$ the states $z_1 = InRoom(Alley) \circ Closed(DA3)$ and $z_2 = InRoom(Alley)$ satisfy the conjunct; in case $d = DA4$ the states $z_1 = InRoom(Alley) \circ Closed(DA4)$ and $z_2 = InRoom(Alley)$ satisfy the conjunct; and in case $d \neq DA3$ and $d \neq DA4$ the states $z_1 = InRoom(Alley) \circ Closed(x)$ and $z_2 = InRoom(Alley)$ satisfy the conjunct.

6.3 Knowledge Update Axioms

Generally, the effect of an action a on the mental state of a robot in a situation s is specified by an update axiom which defines how the states satisfying $KState(s, z)$ relate to the states satisfying $KState(Do(a, s), z)$. Actions can be both sensing and actively manipulating the world. We will discuss their effect on knowledge in turn.

6.3.1 Knowledge update by sensing actions

The effect of sensing on the mental state of a robot is to reduce the set of possible states to the effect that whatever is sensed becomes known.

Definition 16 Let A be a function symbol with range ACTION. A *knowledge update axiom for accurate sensing* for A is of the form

$$\begin{array}{l} Poss(A(\vec{x}), s) \wedge \Delta(\vec{x}, State(s)) \supset \\ \quad [\, KState(Do(A(\vec{x}), s), z) \equiv KState(s, z) \wedge (\forall \vec{y})\, (\Psi(\vec{x}, \vec{y}, z) \equiv \Psi(\vec{x}, \vec{y}, State(s)))\,] \end{array} \tag{28}$$

where $\Delta(\vec{x}, z)$ and $\Psi(\vec{x}, \vec{y}, z)$ are pure state formulas in z.

A *state update axiom for sensing* has the form $Poss(A(\vec{x}), s) \supset State(Do(A(\vec{x}), s)) = State(s)$.
□

Put in words, among all states considered possible prior to sensing only those are still possible afterwards which agree with the actual value of the sensed property, Ψ. Sensing does not affect the actual world state.

A crucial consequence of our definition is that after accurate sensing, the sensed property is known, as the following proposition shows.

Proposition 17 *Let $\Psi(\vec{x}, \vec{y}, z)$ be a pure state formula in z and ψ the fluent formula obtained from Ψ be replacing each $Holds(f, z)$ by f. For any knowledge update axiom (28),*

$$Poss(A(\vec{x}), s) \wedge \Delta(\vec{x}, State(s)) \supset (\forall \vec{y})\, Kwhether(\psi, Do(A(\vec{x}), s))$$

Proof: Assume $Poss(A(\vec{x}), s) \wedge \Delta(\vec{x}, State(s))$, then (28) entails

$$KState(Do(A(\vec{x}), s), z) \supset (\forall \vec{y})\, (HOLDS(\psi, z) \equiv \Psi(\vec{x}, \vec{y}, State(s)))$$

Therefore, if $\Psi(\vec{x}, \vec{y}, State(s))$ then $(\forall z)\, (KState(Do(A(\vec{x}), s), z) \supset HOLDS(\psi, z))$, otherwise if $\neg\Psi(\vec{x}, \vec{y}, State(s))$ then $(\forall z)\, (KState(Do(A(\vec{x}), s), z) \supset \neg HOLDS(\psi, z))$. Put together, we obtain $Knows(\psi, Do(A(\vec{x}), s)) \vee Knows(\neg\psi, Do(A(\vec{x}), s))$, which proves the claim. ■

As an example, suppose that our delivery robot can sense ($Sense$: FLUENT $\mapsto$ ACTION) whether a door is closed if being next to the door. This sensing action can be axiomatized as follows:

$$\begin{array}{l} Poss(Sense(f), z) \equiv (\exists d)\, (f = Closed(d) \wedge Holds(AtDoor(d), z)) \\ Poss(Sense(f), s) \supset \\ \quad [\, KState(Do(Sense(f), s), z) \equiv KState(s, z) \wedge (Holds(f, z) \equiv Holds(f, s))\,] \\ Poss(Sense(f), s) \supset State(Do(Sense(f), s)) = State(s) \end{array} \tag{29}$$

Sensing as considered in Definition 16 is called accurate because it results in full knowledge of the sensed property. Much like effectors in the real world, however, the sensing apparatus of real robots is never absolutely precise. Hence, if it is a quantitative property that is being sensed, then we cannot expect to gain perfect knowledge.

Definition 18 Let A be a function symbol with range ACTION. A *knowledge update axiom for sensing* for A is of the form

$$\begin{array}{l} Poss(A(\vec{x}), s) \wedge \Delta(\vec{x}, State(s)) \supset \\ \quad [\, KState(Do(A(\vec{x}), s), z) \equiv KState(s, z) \wedge \Psi(\vec{x}, z, State(s))\,] \end{array} \tag{30}$$

where $\Delta(\vec{x}, z)$ is a pure state formula in z and $\Psi(\vec{x}, z, z')$ is a first-order formula whose atomic sub-formulas are each pure in z or z'. □

As an example, consider the action Read_{pos} of sensing the current (two-dimensional) physical position of a robot. Assuming that the inaccuracy of the sensors is given by an uncertainty factor ϱ, this is a suitable knowledge update axiom for this action:

$$\begin{array}{l} \mathit{Poss}(\mathit{Read}_{pos}, s) \supset \\ \quad [\, \mathit{KState}(\mathit{Do}(\mathit{Read}_{pos}, s), z) \equiv \\ \qquad \mathit{KState}(s, z) \wedge (\forall x, y, x', y')\, (\mathit{Holds}(\mathit{Position}(x, y), s) \wedge \mathit{Holds}(\mathit{Position}(x', y'), z) \\ \qquad\qquad \supset |x - x'| \leq \varrho \wedge |y - y'| \leq \varrho)\,] \end{array}$$

The crucial property of general knowledge update axioms for sensing is that they solve the representational Frame Problem for knowledge in that everything known before a sensing action is still known afterwards, as the following proposition shows.

Proposition 19 *Let ψ be a fluent formula. For any knowledge update axiom (30),*

$$\mathit{Poss}(A(\vec{x}), s) \wedge \Delta(\vec{x}, \mathit{State}(s)) \supset \mathit{Knows}(\psi, s) \supset \mathit{Knows}(\psi, \mathit{Do}(A(\vec{x}), s))$$

Proof: Assume $\mathit{Poss}(A(\vec{x}), s) \wedge \Delta(\vec{x}, \mathit{State}(s))$, then (30) entails

$$(\forall z)\, (\mathit{KState}(\mathit{Do}(A(\vec{x}), s), z) \supset \mathit{KState}(s, z))$$

Suppose $\mathit{Knows}(\psi, s)$, then $(\forall z)\, (\mathit{KState}(s, z) \supset \mathit{HOLDS}(\psi, z))$. Combining the two implications we obtain $(\forall z)\, (\mathit{KState}(\mathit{Do}(A(\vec{x}), s), z) \supset \mathit{HOLDS}(\psi, z))$, which proves the claim. ■

6.3.2 Knowledge update by physical actions

Knowledge update axioms for physical actions should reflect what a robot knows about the effects of the respective action. To this end, the possible states after acting are obtained by considering all previously possible states and inferring the effect of the action on them. If the action is nondeterministic, then all conceivable outcomes lead to possible states. A knowledge update axiom also combines all conditional effects of an action. Finally, besides knowing that one of the possible effects must materialize, a robot may learn other things about the state by performing an action, which is reflected in the following definition by the additional condition Δ.

Definition 20 Let A be a function symbol with range ACTION. A *knowledge update axiom* for A is of the form

$$\begin{array}{l} \mathit{Poss}(A(\vec{x}), s) \supset \\ \quad [\, \mathit{KState}(\mathit{Do}(A(\vec{x}), s), z) \equiv (\exists z')\, (\, \mathit{KState}(s, z') \wedge \Delta(\vec{x}, z, z', \mathit{State}(s)) \wedge \\ \qquad\qquad [\, \Delta_1(\vec{x}, z') \supset \Upsilon_1(\vec{x}, z, z')\,] \\ \qquad\qquad \wedge \ldots \wedge \\ \qquad\qquad [\, \Delta_m(\vec{x}, z') \supset \Upsilon_m(\vec{x}, z, z')\,]\,] \end{array} \tag{31}$$

where $\Delta(\vec{x}, z, z', z'')$ is a first-order formula whose atomic sub-formulas are each pure in z, z', or z''; $\Delta_i(\vec{x}, z)$ is a pure state formula in z ($1 \leq i \leq m$; $m \geq 1$) and each (possibly nondeterministic) update $\Upsilon_i(\vec{x}, z, z')$ is of the form

$$\begin{array}{l} (\exists \vec{y}_1)\, (z = z' \circ \vartheta_1^+ - \vartheta_1^- \wedge \Theta_1) \\ \vee \ldots \vee \\ (\exists \vec{y}_n)\, z = z' \circ \vartheta_n^+ - \vartheta_n^- \wedge \Theta_n) \end{array}$$

where ϑ_j^+ and ϑ_j^- are fluent collections and Θ_j is a first-order formula without terms of any reserved sort ($1 \leq j \leq n$; $n \geq 1$). □

It should be noted that knowledge update axioms for sensing (Definition 18) are obtained as a special case by taking $\Delta(\vec{x}, z, z', State(s))$ as $\Psi(\vec{x}, z', State(s))$ and setting $m = n = 1$ and $\vartheta_1^+ = \vartheta_1^- = \emptyset$.

Since the formalization of knowledge update is separated from state update axioms, it is possible to specify robots that are arbitrarily limited in what they know about the effects of their actions. This feature supports reasoning about possibly restricted goal achievability of a robot in the sense of [34] but without the need for a meta-theory.

On the other hand, when designing robots with accurate knowledge of the effects of an action, a knowledge update axiom (31) should reflect the underlying specification of state update in that there is a one-to-one correspondence to the conditions Δ_i, the effects ϑ_j^+, ϑ_j^-, and the constraining formulas Θ_j of the $m \geq 1$ (disjunctive) state update axioms (cf. Definition 11). Moreover, in case of accurate knowledge the mere executability of an action teaches the robot that the action must have been possible, hence that its precondition must have held in the previous state. This is stipulated by setting condition Δ in (31) to $Poss(A(\vec{x}), z')$. For example, the following axiom specifies the knowledge update for an informed robot when sending out the identification code (cf. (22)):

$$\begin{aligned}
&Poss(SendId, s) \supset \\
&(\forall z)\,(KState(Do(SendId, s), z) \equiv \\
&\qquad (\exists z')\,(KState(s, z') \wedge Poss(SendId, z') \wedge \\
&\qquad\qquad [\,Holds(AtDoor(d), z') \wedge Holds(Closed(d), z') \supset z = z' - Closed(d)\,] \wedge \\
&\qquad\qquad [\,Holds(AtDoor(d), z') \wedge \neg Holds(Closed(d), z') \supset z = z' \circ Closed(d)\,] \wedge \\
&\qquad\qquad [\,(\neg\exists d)\, Holds(AtDoor(d), z') \supset z = z'\,]\,)
\end{aligned} \tag{32}$$

6.3.3 The inferential Frame Problem for knowledge

Much like state update axioms, knowledge update axioms lay the foundation for a solution to the inferential Frame Problem for knowledge. Consider, for example, a scenario in which the robot knows it is in the alley at door $DA3$ and that $DA4$ is open. Suppose further that in fact, unknown to the robot, $DA3$ is closed:

$$\begin{aligned}
&[\,KState(S_0, z) \equiv \\
&\quad Holds(InRoom(Alley), z) \wedge Holds(AtDoor(DA3), z) \wedge \neg Holds(Closed(DA4), z)\,] \\
&\wedge Holds(Closed(DA3), S_0)
\end{aligned} \tag{33}$$

Consider the action of sensing whether or not door $DA3$ is closed. From (29) we conclude $Poss(Sense(Closed(DA3)), S_0)$. Thus the knowledge update axiom for $Sense(f)$ in (29) entails, after replacing the instance $KState(S_0, z)$ by the equivalent formula given in (33),

$$\begin{aligned}
&KState(Do(Sense(Closed(DA3)), S_0), z) \equiv \\
&\quad Holds(InRoom(Alley), z) \wedge Holds(AtDoor(DA3), z) \wedge \neg Holds(Closed(DA4), z) \\
&\quad \wedge [\,Holds(Closed(DA3), z) \equiv Holds(Closed(DA3), S_0)\,]\,)
\end{aligned}$$

From (33), $Holds(Closed(DA3), S_0)$; hence,

$$\begin{aligned}
&KState(Do(Sense(Closed(DA3)), S_0), z) \equiv \\
&\quad Holds(InRoom(Alley), z) \wedge Holds(AtDoor(DA3), z) \wedge \neg Holds(Closed(DA4), z) \\
&\quad \wedge Holds(Closed(DA3), z)
\end{aligned}$$

Besides the newly acquired information, $\mathit{Closed}(\mathit{DA3})$, the right hand side of the equivalence includes all previously available knowledge of the state. Thus all unaffected knowledge continues to hold without the need to apply extra inference steps.

The reader may verify that we can likewise infer the knowledge state after a subsequent physical SendId action using knowledge update axiom (32):

$$\begin{array}{c}\mathit{KState}(\mathit{Do}(\mathit{SendId}, \mathit{Do}(\mathit{Sense}(\mathit{Closed}(\mathit{DA3})), S_0)), z) \equiv \\ \mathit{Holds}(\mathit{InRoom}(\mathit{Alley}), z) \wedge \mathit{Holds}(\mathit{AtDoor}(\mathit{DA3}), z) \\ \wedge \neg \mathit{Holds}(\mathit{Closed}(\mathit{DA3}), z) \wedge \neg \mathit{Holds}(\mathit{Closed}(\mathit{DA4}), z)\end{array}$$

6.4 Conditional Actions

Considering plans as mere sequences of elementary actions is insufficient for planning with sensing actions, as we have illustrated at the beginning of this section. Robots need to be able to condition their course of actions on the outcome of sensing. To this end, we further extend the signature of the basic Fluent Calculus by elements that support the representation of branching actions. For the sake of simplicity, we confine ourselves to conditioning on the value of single fluents. The generalization to arbitrary fluent formulas is straightforward but requires to formally introduce the concept of fluent formulas into the signature.

Definition 21 The *Fluent Calculus for sensing* is a sorted second-order logic language which extends the language of the simple Fluent Calculus by the predicates

$$\mathit{KState}: \ \text{SIT} \times \text{STATE} \qquad\qquad \mathit{Poss}: \ \text{ACTION} \times \text{SIT}$$

and the functions

$$\begin{array}{rl} \epsilon: & \mapsto \text{ACTION} \\ ;: & \text{ACTION} \times \text{ACTION} \mapsto \text{ACTION} \\ \mathit{If}: & \text{FLUENT} \times \text{ACTION} \times \text{ACTION} \mapsto \text{ACTION} \end{array}$$

□

Note that Poss is overloaded (cf. Definition 8); the expression $\mathit{Poss}(a, s)$, which thus far has been used as a mere macro (recall (6)), becomes part of the language because the possibility to execute actions may depend on the mental state of the robot and not just on the world state as in the simple Fluent Calculus.[9] The constant "ϵ" (read: "no-op") shall denote the empty action. The symbol ";" shall denote the consecutive performance of two actions. Performing the conditional action $\mathit{If}(f, a_1, a_2)$ means to perform a_1 if f holds and to perform a_2 otherwise. An example of an action in the extended delivery domain is, $\mathit{Sense}(\mathit{Closed}(\mathit{DA3})); \mathit{If}(\mathit{Closed}(\mathit{DA3}), \mathit{SendId}, \epsilon); \mathit{Enter}(\mathit{R403})$.

Preconditions and effects of the special actions are characterized by the foundational axiom set $\mathcal{F}_{cond}$:

$$\begin{array}{l} \mathit{Poss}(\epsilon, s) \equiv \mathit{True} \\ \mathit{Poss}(a_1; a_2, s) \equiv \mathit{Poss}(a_1, s) \wedge \mathit{Poss}(a_2, \mathit{Do}(a_1, s)) \\ \mathit{Poss}(\mathit{If}(f, a_1, a_2), s) \equiv \mathit{Kwhether}(f, s) \wedge \\ \qquad\qquad\qquad [\, \mathit{Holds}(f, s) \supset \mathit{Poss}(a_1, s) \,] \wedge [\, \neg \mathit{Holds}(f, s) \supset \mathit{Poss}(a_2, s) \,] \end{array}$$

[9] In what follows, for each action $A(\vec{x})$ not accompanied by an explicit axiom for $\mathit{Poss}(A(\vec{x}), s)$, we tacitly assume given the axiom $\mathit{Poss}(A(\vec{x}), s) \equiv \mathit{Poss}(A(\vec{x}), \mathit{State}(s))$.

$$
\begin{array}{l}
\mathit{State}(\mathit{Do}(\epsilon,s)) = \mathit{State}(s) \wedge [\mathit{KState}(\mathit{Do}(\epsilon,s),z) \equiv \mathit{KState}(s,z)] \\
\mathit{Poss}(a_1;a_2,s) \supset \mathit{State}(\mathit{Do}(a_1;a_2,s)) = \mathit{State}(\mathit{Do}(a_2,\mathit{Do}(a_1,s))) \wedge \\
\qquad\qquad [\mathit{KState}(\mathit{Do}(a_1;a_2,s),z) \equiv \mathit{KState}(\mathit{Do}(a_2,\mathit{Do}(a_1,s)),z)] \\
\mathit{Poss}(\mathit{If}(f,a_1,a_2),s) \supset \\
\quad [\,\mathit{Holds}(f,s) \supset \mathit{State}(\mathit{Do}(\mathit{If}(f,a_1,a_2),s)) = \mathit{State}(\mathit{Do}(a_1,s)) \wedge \\
\qquad\qquad [\mathit{KState}(\mathit{Do}(\mathit{If}(f,a_1,a_2),s),z) \equiv \mathit{KState}(\mathit{Do}(a_1,s),z)]\,] \wedge \\
\quad [\,\neg \mathit{Holds}(f,s) \supset \mathit{State}(\mathit{Do}(\mathit{If}(f,a_1,a_2),s)) = \mathit{State}(\mathit{Do}(a_2,s))\,] \wedge \\
\qquad\qquad [\mathit{KState}(\mathit{Do}(\mathit{If}(f,a_1,a_2),s),z) \equiv \mathit{KState}(\mathit{Do}(a_2,s),z)]\,]
\end{array}
$$

Notice in particular that a conditional action is possible only if the truth-value of the condition is known at the time of performance because otherwise the robot would not know what to do. With the help of conditionals it is possible to devise plans that provably achieve goals in a setting where incomplete knowledge requires active sensing and where considering a single action sequence is insufficient. As an example, recall the initial state (33), where the robot knows it is in the alley at door $\mathit{DA3}$. We prove that this plan achieves the goal of being in room $\mathit{R403}$:

$$
\mathit{Do}(\mathit{Sense}(\mathit{Closed}(\mathit{DA3})); \mathit{If}(\mathit{Closed}(\mathit{DA3}), \mathit{SendId}, \epsilon); \mathit{Enter}(\mathit{R403}), S_0) \tag{34}
$$

From (29) and $\mathcal{F}_{knows}$ we conclude that $\mathit{Poss}(\mathit{Sense}(\mathit{Closed}(\mathit{DA3})), S_0)$. Then according to (29), $\mathit{Kwhether}(\mathit{Closed}(\mathit{DA3}), \mathit{Do}(\mathit{Sense}(\mathit{Closed}(\mathit{DA3})), S_0))$. Thus, since ϵ and SendId are always possible, $\mathit{Poss}(\mathit{If}(\mathit{Closed}(\mathit{DA3}), \mathit{SendId}, \epsilon), \mathit{Do}(\mathit{Sense}(\mathit{Closed}(\mathit{DA3})), S_0))$. From (29) it follows that

$$
\begin{array}{ll}
 & \mathit{Holds}(\mathit{Closed}(\mathit{DA3}), \mathit{Do}(\mathit{Sense}(\mathit{Closed}(\mathit{DA3})), S_0)) \supset \\
 & \quad \neg \mathit{Holds}(\mathit{Closed}(\mathit{DA3}), \mathit{Do}(\mathit{SendId}, \mathit{Do}(\mathit{Sense}(\mathit{Closed}(\mathit{DA3})), S_0))) \\
\text{and} & \neg \mathit{Holds}(\mathit{Closed}(\mathit{DA3}), \mathit{Do}(\mathit{Sense}(\mathit{Closed}(\mathit{DA3})), S_0)) \supset \\
 & \quad \neg \mathit{Holds}(\mathit{Closed}(\mathit{DA3}), \mathit{Do}(\epsilon, \mathit{Do}(\mathit{Sense}(\mathit{Closed}(\mathit{DA3})), S_0)))
\end{array}
$$

Therefore, $\mathit{Poss}(\mathit{Enter}(\mathit{R403}), \mathit{Do}(\mathit{If}(\mathit{Closed}(\mathit{DA3}), \mathit{SendId}, \epsilon), \mathit{Do}(\mathit{Sense}(\mathit{Closed}(\mathit{DA3}), S_0))))$. Then (10) and $\mathcal{F}_{cond}$ entail,

$$
\begin{array}{l}
\mathit{Holds}(\mathit{InRoom}(\mathit{R403}), \\
\qquad \mathit{Do}(\mathit{Sense}(\mathit{Closed}(\mathit{DA3})); \mathit{If}(\mathit{Closed}(\mathit{DA3}), \mathit{SendId}, \epsilon); \mathit{Enter}(\mathit{R403}), S_0))
\end{array}
$$

Our account of knowledge and sensing is summarized in the following extension of the notion of a domain axiomatization in the Fluent Calculus.

Definition 22 A *Fluent Calculus domain axiomatization with knowledge and sensing* consists of a set of state constraints, a unique simple action precondition axiom as well as a unique knowledge update axiom for each function symbol with range ACTION, a set of state update axioms, foundational axioms $\mathcal{F}_{state}$, $\mathcal{F}_{knows}$, $\mathcal{F}_{cond}$ and possibly (26), plus possible further domain-specific axioms. □

6.5 Knowledge in FLUX

For the sake of simplicity and efficiency, the encoding of knowledge and sensing in FLUX presented in the following concentrates on the major purpose to program a planning agent. We presuppose that the robot is aware of all preconditions and effects of its action and of all state constraints.

Moreover, we assume the strictly reflective stance regarding state knowledge: A property is known just in case it can be derived from an incomplete state specification. With these two assumptions it is possible to avoid the separate introduction of knowledge update. It rather suffices to extend the existing state update computation by a definition of knowledge wrt. state representations and of how sensing affects this representation. This leads to a clean and effective computation mechanism for knowledge.

6.5.1 Inferring state knowledge

The reflective stance concerning state knowledge means to take a given state specification $\Sigma(\mathit{State}(S))$ as what the robot knows in situation S, that is, $\mathit{KState}(S,z) \equiv \Sigma(z)$. Justified by definitions (23) and (24), we can thus define the value of a fluent to be known just in case Σ entails either the fluent or its negation. Speaking in terms of FLUX, a fluent f is entailed to hold in a state z iff adding the constraint $\mathit{NotHolds}(f,z)$ leads to failure. Likewise, the fluent is entailed to not hold iff the goal $\mathit{Holds}(f,z)$ fails.

Consider, for example, the initial state specification

$$\begin{aligned}\mathit{Init}(z_0) \leftarrow\ & \mathit{Holds}(\mathit{InRoom}(\mathit{Alley}), z_0),\ \mathit{Holds}(\mathit{AtDoor}(\mathit{DA3}), z_0), \\ & \mathit{NotHolds}(\mathit{Closed}(\mathit{DA4}), z_0),\ \mathit{Consistent}(z_0),\ \mathit{DuplicateFree}(z_0)\end{aligned} \tag{35}$$

then the robot knows that initially it is at door $\mathit{DA3}$ and door $\mathit{DA4}$ is not closed; accordingly, both $\mathit{Init}(z_0)$, $\backslash\!+\mathit{NotHolds}(\mathit{AtDoor}(\mathit{DA3}), z_0)$ and $\mathit{Init}(z_0)$, $\backslash\!+\mathit{Holds}(\mathit{Closed}(\mathit{DA4}), z_0)$ succeed. On the other hand, the robot does not know whether door $\mathit{DA3}$ is closed or not; accordingly, $\mathit{Init}(z_0), \backslash\!+\mathit{NotHolds}(\mathit{Closed}(\mathit{DA3}), z_0)$ and $\mathit{Init}(z_0), \backslash\!+\mathit{Holds}(\mathit{Closed}(\mathit{DA3}), z_0)$ both fail. The robot is also aware of all state constraints: Query $\mathit{Init}(z_0)$, $\backslash\!+\mathit{Holds}(\mathit{AtDoor}(\mathit{DA4}), z_0)$ succeeds—the robot knows it cannot be at door $\mathit{DA4}$ because it knows it is in front of $\mathit{DA3}$ and cannot be at both places simultaneously.

Identifying state specifications with knowledge, the effect of sensing the value of a fluent is to leave a state itself unchanged but to affect the specification:

```
state_update(Z, sense(F), Z, SV) :-
   holds(F, Z), SV = F ; not_holds(F, Z), SV = -(F).
```

(For later purposes we record the sensed value in an additional argument of $\mathit{StateUpdate}$ for all sensing actions.)

A consequence of this clause is that after a $\mathit{Sense}(f)$ action the value of fluent f is known. For example, if $\mathit{Init}(z_0), \mathit{StateUpdate}(z_0, \mathit{Sense}(\mathit{Closed}(\mathit{DA3})), z_1, \_)$ is queried wrt. (35), then two answers result, one of which includes $z_1/[\mathit{InRoom}(\mathit{Alley}), \mathit{AtDoor}(\mathit{DA3}), \mathit{Closed}(\mathit{DA3})|z]$, whereas the other one includes $z_1/[\mathit{InRoom}(\mathit{Alley}), \mathit{AtDoor}(\mathit{DA3})|z]$ along with the constraint $\mathit{NotHolds}(\mathit{Closed}(\mathit{DA3}), z)$. In both cases it becomes known whether $\mathit{Closed}(\mathit{DA3})$ holds while the state itself does not change.

A correct account of knowledge as derivability requires, however, to resolve a conflict caused by the way actions with conditional effects are treated in FLUX. Consider, for example, the following encoding of the SendId action (cf. (22)):

$$\begin{aligned}&\mathit{StateUpdate}(z_1, \mathit{SendId}, z_2) \leftarrow \\ &\quad \mathit{Holds}(\mathit{AtDoor}(d), z_1),\ \mathit{Holds}(\mathit{Closed}(d), z_1),\ \mathit{Update}(z_1, [\,], [\mathit{Closed}(d)], z_2)\ ; \\ &\quad \mathit{Holds}(\mathit{AtDoor}(d), z_1),\ \mathit{NotHolds}(\mathit{Closed}(d), z_1),\ \mathit{Update}(z_1, [\mathit{Closed}(d)], [\,], z_2)\ ; \\ &\quad \mathit{NotHoldsAll}([d], \mathit{AtDoor}(d), z_1),\ \mathit{Equal}(z_1, z_2).\end{aligned}$$

The disjunctive body may cause knowledge to emerge as a side-effect of applying this clause. If, for instance, $\mathit{Init}(z_0), \mathit{StateUpdate}(z_0, \mathit{SendId}, z_1)$ is queried wrt. (35), then two answers result, the first of which includes the substitution $z_1/[\mathit{InRoom}(\mathit{Alley}), \mathit{AtDoor}(\mathit{DA3})|z]$ along with the constraint $\mathit{NotHolds}(\mathit{Closed}(\mathit{DA3}), z)$, while the second one includes the substitution $z_1/[\mathit{Closed}(\mathit{DA3}), \mathit{InRoom}(\mathit{Alley}), \mathit{AtDoor}(\mathit{DA3})|z]$. In both cases $\mathit{Closed}(\mathit{DA3})$ is known wrt. the respective state representations.

Knowledge therefore depends not only on the current state representation but also on the previously performed actions. For this reason, the predicate $\mathit{Kwhether}(f, z, s)$ used in Flux carries the situation argument, represented as a list of actions. Fluent f is defined to be known if it is either true or false in state z and if it cannot go both ways wrt. the actions in s under the same results of all performed sensing actions:

```
kwhether(F, Z, S) :-
   is_fluent(F),
   (\+ not_holds(F, Z) ; \+ holds(F, Z)),
   \+ ( init(Z0a), result(S, SensedValues, Z0a, Z1), holds(F, Z1),
        init(Z0b), result(S, SensedValues, Z0b, Z2), not_holds(F, Z2) ).

result([], [], Z, Z).
result([A|S], SensedValues, Z0, Z) :-
   state_update(Z0, A, Z1), result(S, SensedValues, Z1, Z).
result([A|S], [SV|SensedValues], Z0, Z) :-
   state_update(Z0, A, Z1, SV), result(S, SensedValues, Z1, Z).
```

For instance, the query $\mathit{Init}(z_0), \mathit{Kwhether}(\mathit{Closed}(\mathit{DA3}), z_0, [\,])$ fails wrt. (35) just like the query $\mathit{Init}(z_0), \mathit{StateUpdate}(z_0, \mathit{SendId}, z_1), \mathit{Kwhether}(\mathit{Closed}(\mathit{DA3}), z_1, [\mathit{SendId}])$ does; whereas

$$\begin{aligned}&\mathit{Init}(z_0),\ \mathit{StateUpdate}(z_0, \mathit{Sense}(\mathit{Closed}(\mathit{DA3})), z_1),\\ &\qquad \mathit{Kwhether}(\mathit{Closed}(\mathit{DA3}), z_1, [\mathit{Sense}(\mathit{Closed}(\mathit{DA3}))])\end{aligned}$$

is successful just like

$$\begin{aligned}&\mathit{Init}(z_0),\ \mathit{StateUpdate}(z_0, \mathit{Sense}(\mathit{Closed}(\mathit{DA3})), z_1),\ \mathit{StateUpdate}(z_1, \mathit{SendId}, z_2),\\ &\qquad \mathit{Kwhether}(\mathit{Closed}(\mathit{DA3}), z_2, [\mathit{Sense}(\mathit{Closed}(\mathit{DA3})), \mathit{SendId}])\end{aligned}$$

It is worth stressing that our encoding supports indirect sensing; e.g., figuring out whether a solution is acidic by sensing whether a litmus strip turned red, to mention a well-known example [45].

6.5.2 Planning with conditional actions

Plans involving conditional actions and branching are represented in Flux as nested lists of actions. Focusing on the planning problem, the previously used predicate DO is modified to the effect that actions are only considered if they are *known* to be possible at the time of their performance. This avoids considering plan steps which are not provably possible and which thus would be refuted during the subsequent verification of a plan. Moreover, to be able to verify that the condition of a conditional action is known, the argument structure is extended to $\mathit{DO}(s, s_0, z_0, s_n, z_n)$ with the intended reading that the actions of situation s are provably executable in situation s_0 with world state z_0, and that the execution may lead to situation s_n with world state z_n:

```
do([], S, Z, S, Z).
do([if(F,S1,S2)|S], S0, Z0, Sn, Zn) :-
   kwhether(F, Z0, S0),
   ( holds(F, Z0), append(S1, S, S1S), do(S1S, S0, Z0, Sn, Zn) ;
     not_holds(F, Z0), append(S2, S, S2S), do(S2S, S0, Z0, Sn, Zn) ).
do([A|S], S0, Z0, Sn, Zn) :-
   primitive_action(A),
   \+ not_poss(A, Z0), append(S0, [A], S1),
   ( state_update(Z0, A, Z1) ; state_update(Z0, A, Z1, _) ),
   do(S, S1, Z1, Sn, Zn).
```

Predicate *PrimitiveAction* is used to define the domain-specific robot actions. In order to test correctness of a conditional plan, the predicate $\mathit{NonExecutable}(s, s_0, z_0)$ is extended (c.f. (20)), now representing that action sequence s is not executable in situation s_0 with state z_0:

```
non_executable([if(F,S1,S2)|S], S0, Z0) :-
   \+ kwhether(F, Z0, S0) ;
   holds(F, Z0), append(S1, S, S1S), non_executable(S1S, S0, Z0) ;
   not_holds(F, Z0), append(S2, S, S2S), non_executable(S2S, S0, Z0).
non_executable([A|S], S0, Z0) :-
   primitive_action(A),
   ( not_poss(A, Z0) ;
     poss(A, Z0), ( state_update(Z0, A, Z1) ; state_update(Z0, A, Z1, _) ),
     append(S0, [A], S1), non_executable(S, S1, Z1) ).
```

For illustration, Appendix A.3 includes the modified FLUX program for our sensing delivery robot. Recall, for example, specification (35). The query

$$\begin{array}{l} \mathit{Init}(z_0),\ \mathit{DO}(s, [\,], z_0, s_n, z_n),\ \mathit{Holds}(\mathit{InRoom}(\mathit{R403}), z_n), \\ \mathit{Init}(z_0'),\ \backslash{+}\mathit{NonExecutable}(s, [\,], z_0'), \\ \qquad\qquad \backslash{+}(\mathit{DO}(s, [\,], z_0', s_n', z_n'),\ \mathit{NotHolds}(\mathit{InRoom}(\mathit{R403}), z_n')) \end{array}$$

admits a successful derivation whose answer includes the substitution (cf. (34))

$$s/[\mathit{Sense}(\mathit{Closed}(\mathit{DA3})), \mathit{If}(\mathit{Closed}(\mathit{DA3}), [\mathit{SendId}], [\,]), \mathit{Enter}(\mathit{R403})]$$

6.5.3 Knowledge goals

An outstanding feature of our approach, e.g., in comparison to [14, 25, 54], is that planning problems can be solved where the goal is to gain knowledge. A simple example is the task to figure out whether a door, say $\mathit{DA1}$, is closed:

$$\begin{array}{l} \mathit{Init}(z_0),\ \mathit{DO}(s, [\,], z_0, s_n, z_n),\ \mathit{Kwhether}(\mathit{Closed}(\mathit{DA1}), z_n), \\ \mathit{Init}(z_0'),\ \backslash{+}\mathit{NonExecutable}(s, [\,], z_0') \end{array}$$

with the straightforward solution $s/[\mathit{Sense}(\mathit{Closed}(\mathit{DA1})), \mathit{Go}(\mathit{DA1})]$.

7 Ramifications: Indirect Effects

A specification of update by means of explicit collections of positive and negative effects, as in simple (disjunctive) state update axioms, always formalizes the entire effect of an action. In

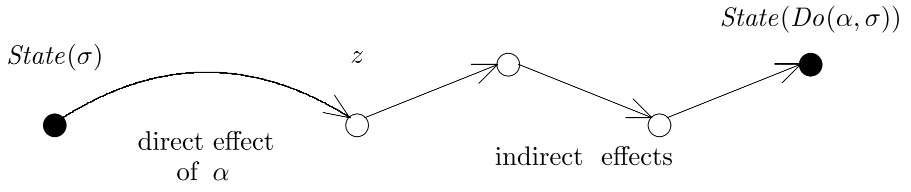


Figure 6: Ramifications as causal chains: The result of the direct effect of action α, state z, is the root of a path through several intermediate states, linked by a causal relation. An example would be the robot picking up an object in $State(\sigma)$, in which case this object is carried in z. The further states are obtained by concluding, one-by-one, that the robot carries all objects directly or indirectly attached to the primary one.

many circumstances, however, it is desirable to distinguish between direct changes and further, indirect effects [55]: Firstly, axiomatizations of large domains can be structured much better, hence be made more concise and elaboration tolerant. Secondly, actions may cause unbounded chains of effects, which are generally difficult to summarize in a direct formalization of update. As an example, consider the possibility that objects are somehow attached to other objects, like a cable being plugged into an adapter which is in turn connected to an Ethernet card. The effect of picking up, say, the card is then not only to carry the card itself but also the adapter, hence also the cable. Likewise, dropping one of them means to get rid of the other two at the same time as indirect effect, which may in turn cause the completion of a request concerning the attached objects. Since attachment between objects may vary freely, the effect of moving things cannot be specified as a fixed effect of an action.

The challenge of extending solutions to the Frame Problem so as to cope with indirect effects is known as the Ramification Problem [15]. Adapting our general theory of [63], the approach taken in the Fluent Calculus is to infer indirect effects with the help of causal chains: Starting with the direct effect of an action, further changes are successively determined on the basis of a causal relation. Figure 6 gives a schematic illustration of this approach.

The various causal relations of a domain are axiomatized with the help of the expression $Causes(\varepsilon,\varrho,z)$ where ε (the *triggering effect*) and ϱ (the *ramification*) are possibly negated atomic fluent formulas and z is a state. The intuitive meaning is that in state z, indirect effect ϱ is triggered by a preceding effect ε. As an example, consider the fluent $Attached$: $\text{OBJECT} \times \text{OBJECT} \mapsto \text{FLUENT}$ along with the state constraints

$$\begin{aligned}&Holds(Attached(x,y),s) \equiv Holds(Attached(y,x),s)\\&Holds(Attached(x,y),s) \supset [Holds(Carries(x),s) \equiv Holds(Carries(y),s)]\end{aligned} \tag{36}$$

The dynamic aspect of the extended domain is expressed by these four axioms:

$$\begin{aligned}&Holds(Attached(x,y),z) \supset Causes(Carries(x), Carries(y), z)\\&Holds(Attached(x,y),z) \supset Causes(\neg Carries(x), \neg Carries(y), z)\\&Holds(Request(r_1,x,r_2),z) \supset Causes(\neg Carries(x), \neg Request(r_1,x,r_2), z)\\&Holds(InRoom(r),z) \wedge r \neq r_2 \supset Causes(\neg Request(r_1,x,r_2),z), Request(r,x,r_2),z)\end{aligned} \tag{37}$$

Put in words, if x and y are attached, then the positive effect $Carries(x)$ causes the indirect effect $Carries(y)$. Conversely, if again x and y are attached, then the negative effect of

Carries(x) becoming false causes *Carries*(y) to not hold, too. Dropping an object incidentally cancels a possible request concerning this object. If, however, the object is dropped in the wrong office, then a new request is caused, which asks for taking the object from its current location to the desired one.

The distinction between a context of a causal relation and a triggering effect (which represent a kind of 'momentum') is essential when considering intermediate states. To appreciate this, consider a state z in which two objects a and b are attached and neither is carried. If then *Carries*(a) occurs as direct or indirect effect, we expect *Carries*(b) to be additionally caused, that is,

$$z \quad \longrightarrow \quad z \circ \mathit{Carries}(a) \quad \longrightarrow \quad z \circ \mathit{Carries}(a) \circ \mathit{Carries}(b) \tag{38}$$

On the other hand, if a and b are attached in $z \circ \mathit{Carries}(a) \circ \mathit{Carries}(b)$ and *Carries*(b) becomes false, then we expect *Carries*(a) to no longer hold, that is,

$$z \circ \mathit{Carries}(a) \circ \mathit{Carries}(b) \quad \longrightarrow \quad z \circ \mathit{Carries}(a) \quad \longrightarrow \quad z \tag{39}$$

Since the intermediate states in (38) and (39) are identical, getting the indirect effect right relies on knowing the preceding effect.

The expression *Causes* as used above is just syntactic sugar to allow for convenient specifications of cause-effect pairs. Actually, the extended signature of the Fluent Calculus for ramifications is as follows.

Definition 23 The *Fluent Calculus with ramifications* is a sorted second-order logic language which includes the reserved predicates

$$\begin{aligned} \mathit{Causes}&: \text{STATE} \times \text{STATE} \times \text{STATE} \times \text{STATE} \times \text{STATE} \times \text{STATE} \\ \mathit{Ramify}&: \text{STATE} \times \text{STATE} \times \text{STATE} \times \text{STATE} \end{aligned}$$

□

An instance $\mathit{Causes}(z_1, e_1^+, e_1^-, z_2, e_2^+, e_2^-)$ means that if state z_1 is the result of positive effects e_1^+ and negative effects e_1^-, then an additional effect is caused which leads to state z_2 (now the result of positive and negative effects e_2^+, e_2^-, resp.).[10] This predicate is used to define the ternary macro $\mathit{Causes}(\varepsilon, \varrho, z)$ as follows:

$$\begin{aligned} \mathit{Causes}(f, f', z_1) &\overset{\text{def}}{=} (\forall e^+, e^-)\, \mathit{Causes}(z_1, e^+ \circ f, e^-, z_1 \circ f', e^+ \circ f \circ f', e^- - f') \\ \mathit{Causes}(f, \neg f', z_1) &\overset{\text{def}}{=} (\forall e^+, e^-)\, \mathit{Causes}(z_1, e^+ \circ f, e^-, z_1 - f', e^+ \circ f - f', e^- \circ f') \\ \mathit{Causes}(\neg f, f', z_1) &\overset{\text{def}}{=} (\forall e^+, e^-)\, \mathit{Causes}(z_1, e^+, e^- \circ f, z_1 \circ f', e^+ \circ f', e^- \circ f - f') \\ \mathit{Causes}(\neg f, \neg f', z_1) &\overset{\text{def}}{=} (\forall e^+, e^-)\, \mathit{Causes}(z_1, e^+, e^- \circ f, z_1 - f', e^+ - f', e^- \circ f \circ f') \end{aligned} \tag{40}$$

Notice how the collections of negative and positive effects are guaranteed to remain disjoint by subtracting, if necessary, a newly established positive (resp. negative) indirect effect from the preceding negative (resp. positive) effects.

The second predicate introduced in Definition 23, $\mathit{Ramify}(z, e^+, e^-, z')$, shall mean that state z' can be reached by iterated application of the underlying causal relation, starting in

[10]While formally collections of effects are terms of sort STATE, they should not be viewed as corresponding to an actual complete state of the world. In what follows, all variables e with sub- or superscripts are of sort STATE.

state z with 'momentum' e^+, e^-. A foundational axiom $\mathcal{F}_{ramify}$ defines *Ramify* as fixpoints of *Causes*:

$$Ramify(z_1, e_1^+, e_1^-, z_2) \equiv (\exists e_2^+, e_2^-)\,(z_1, e_1^+, e_1^-, z_2, e_2^+, e_2^-) \in \mu[Causes]$$

where $(\vec{x}, \vec{y}) \in \mu[P]$ abbreviates the following formula, which is a standard second-order schema to axiomatize that $(\vec{x}, \vec{y})$ belongs to the reflexive and transitive closure of predicate P and that $\vec{y}$ is a fixpoint:

$$\forall \Phi \left\{ \begin{array}{c} (\forall \vec{u})\, \Phi(\vec{u}, \vec{u}) \,\wedge\, (\forall \vec{u}, \vec{v}, \vec{w})\, [\, \Phi(\vec{u}, \vec{v}) \wedge P(\vec{v}, \vec{w}) \,\supset\, \Phi(\vec{u}, \vec{w})\,] \\ \supset\, \Phi(\vec{x}, \vec{y}) \end{array} \right\} \wedge\, (\forall \vec{z})\, (P(\vec{y}, \vec{z}) \supset \vec{y} = \vec{z})$$

Using the transitive closure relies on the assumption that the underlying *Causes* relation is completely specified. To this end, we circumscribe [41] this predicate wrt. a given axiomatization of cause-effect pairs. If in these axioms *Causes* occurs only as the single consequent of implications, like in (37), then second-order circumscription is equivalent to first-order completion [31].

On the basis of a causal relation and its closure, the following generalization of simple state update axioms accounts for actions with ramifications.

Definition 24 Let A be a function symbol with range ACTION. A *state update axiom with ramifications* for A is of the form

$$\begin{array}{l} Poss(A(\vec{x}), s) \,\wedge\, \Delta(\vec{x}, State(s)) \,\supset \\ \quad (\exists z)\, (\, z = State(s) \circ \vartheta^+ - \vartheta^- \wedge Ramify(z, \vartheta^+, \vartheta^-, State(Do(A(\vec{x}), s)))\,) \end{array}$$

where $\Delta(\vec{x}, z)$ is a pure state formula in z and ϑ^+ and ϑ^- are fluent collections.

A *disjunctive state update axiom with ramifications* for A is of the form

$$\begin{array}{ll} Poss(A(\vec{x}), s) \,\wedge\, \Delta(\vec{x}, State(s)) \,\supset & (\exists \vec{y}_1, z)\, (z = State(s) \circ \vartheta_1^+ - \vartheta_1^- \wedge \Theta_1 \wedge \\ & \qquad Ramify(z, \vartheta_1^+, \vartheta_1^-, State(Do(A(\vec{x}), s)))\,) \\ & \vee \ldots \vee \\ & (\exists \vec{y}_n, z)\, (z = State(s) \circ \vartheta_n^+ - \vartheta_n^- \wedge \Theta_n \wedge \\ & \qquad Ramify(z, \vartheta_n^+, \vartheta_n^-, State(Do(A(\vec{x}), s)))\,) \end{array}$$

where $\Delta(\vec{x}, z)$ is a pure state formula in z, ϑ_i^+ and ϑ_i^- are fluent collections, and Θ_i is a first-order formula without terms of any reserved sort ($1 \leq i \leq n$; $n \geq 1$). □

It is worth mentioning that there might be more than one successor state satisfying *Ramify* such state so that actions with ramifications can be nondeterministic even in case of a non-disjunctive update axiom.[11]

As an example, recall the two actions *Pickup* and *Drop* of our delivery robot. To account for possible indirect effects, they are generalized as follows:

$$\begin{array}{l} Poss(Pickup(x), s) \,\supset \\ \quad (\exists z)\, (z = State(s) \circ Carries(x) \,\wedge\, Ramify(z, Carries(x), \emptyset, State(Do(Pickup(x), s)))\,) \\ Poss(Drop(x), s) \,\supset \\ \quad (\exists z)\, (z = State(s) - Carries(x) \,\wedge\, Ramify(z, \emptyset, Carries(x), State(Do(Drop(x), s)))\,) \end{array}$$

[11]Suppose, for instance, we add to (37) the possible indirect effect that $Attached(x, y)$ becomes false if $Carries(x)$ occurs as positive effect while y is not carried. With this extension, picking up an object attached to another one may result either in both being carried, or in the two objects becoming detached as indirect effect.

Notice that the power of ramification allows us to simplify the direct update for *Drop* by ignoring the (now indirect) effect that a request is completed. The following, for instance, is a logical consequence of our extended delivery scenario:

$$
\begin{array}{l}
[\,\mathit{Holds}(\mathit{Attached}(x,y),S_0)\;\equiv \\
\qquad x=\mathit{Cable}\wedge y=\mathit{Adapter}\;\vee\; x=\mathit{Adapter}\wedge y=\mathit{EthernetCard}\;\vee \\
\qquad x=\mathit{Adapter}\wedge y=\mathit{Cable}\;\vee\; x=\mathit{EthernetCard}\wedge y=\mathit{Adapter}\,] \\
\wedge\,(\forall x)\,\neg\mathit{Holds}(\mathit{Carries}(x),S_0) \\
\quad\supset\; \mathit{Holds}(\mathit{Carries}(\mathit{Cable}),\mathit{Do}(\mathit{Pickup}(\mathit{EthernetCard}),S_0))\,\wedge \\
\qquad \neg\mathit{Holds}(\mathit{Carries}(\mathit{EthernetCard}),\mathit{Do}(\mathit{Drop}(\mathit{Adapter}),\mathit{Do}(\mathit{Pickup}(\mathit{EthernetCard}),S_0)))
\end{array}
\tag{41}
$$

Our solution to the Ramification Problem is summarized in the following extension of the notion of a domain axiomatization in the Fluent Calculus.

Definition 25 A *Fluent Calculus domain axiomatization with ramifications* consists of a set of state constraints, a circumscribed axiomatization of causal relations, a unique simple action precondition axiom for each function symbol with range ACTION, a set of state update axioms, possibly with ramifications, foundational axioms $\mathcal{F}_{state}$ and $\mathcal{F}_{ramify}$, plus possible further domain-specific axioms. □

The notion of knowledge update axioms according to Definition 20 can be straightforwardly generalized to include ramifications known by the robot. On this basis, Definition 22 for domains with knowledge and sensing can be easily reconciled with Definition 25.

Ramifications in Flux

The causal relations of a domain are encoded in FLUX by defining the ternary predicate *Causes*. To infer fixpoints of causal chains, an additional clause is needed which defines the predicate $\mathit{NotCauses}(z,e^+,e^-)$, representing that no further indirect effects apply in state z wrt. effects e^+,e^-. The encoding of the indirect effects in the extended delivery scenario (cf. (37)), together with the additional state constraints, is shown in Appendix A.4.

Positive and negative indirect effects define the predicate $\mathit{Causes}(z_1,e_1^+,e_1^-,z_2,e_2^+,e_2^-)$ as follows (c.f. (40)):

```
causes(Z1, EP1, EN1, EP2, EN2, Z2) :-
   causes(EF, RA, Z),
   ( \+ EF = -(_), \+ RA = -(_), holds(EF, EP1),
     plus(Z1, [RA], Z2), plus(EP1, [RA], EP2), minus(EN1, [RA], EN2) ;

     \+ EF = -(_), RA = -(R), holds(EF, EP1)
     minus(Z1, [R], Z2), minus(EP1, [R], EP2), plus(EN1, [R], EN2) ;

     EF = -(E), \+ RA = -(_), holds(E, EN1),
     plus(Z1, [RA], Z2), plus(EP1, [RA], EP2), minus(EN1, [RA], EN2) ;

     EF = -(E), RA = -(R), holds(E, EN1)
     minus(Z1, [R], Z2), minus(EP1, [R], EP2), plus(EN1, [R], EN2) ).
```

Based on the specification of a causal relation, ramification is defined recursively:

```
ramify(Z1, EP, EN, Z2) :- not_causes(Z1, EP, EN), equal(Z1, Z2).
ramify(Z1, EP1, EN1, Z) :-
   causes(Z1, EP1, EN1, Z2, EP2, EN2), ramify(Z2, EP2, EN2, Z).
```

Let our running example program be augmented by an encoding of the state constraints in (36), represented by the constraints $AttachedSymmetric(z)$ and $CarriesSymmetric(z)$ along with the corresponding CHRs shown in Appendix A.1, and let the update axioms for *Pickup* and *Drop* be modified as shown in Appendix A.4. The following query (cf. (41)) has a successful derivation wrt. the extended program:

$$\begin{array}{l}
Holds(Attached(Cable, Adapter), z_0, z'),\ Holds(Attached(Adapter, EthernetCard), z', z'')\\
Holds(Attached(Adapter, Cable), z'', z'''),\ Holds(Attached(EthernetCard, Adapter), z''', z''''),\\
NotHoldsAll([x, y], Attached(x, y), z''''),\\
NotHoldsAll([x], Carries(x), z_0),\ Consistent(z_0),\ DuplicateFree(z_0),\\
StateUpdate(z_0, Pickup(EthernetCard), z_1),\ \backslash{+}NotHolds(Carries(Cable), z_1),\\
StateUpdate(z_1, Drop(Adapter), z_2),\ \backslash{+}Holds(Carries(EthernetCard), z_2)
\end{array}$$

8 Concurrency

It may be desirable to have a robot execute several actions concurrently, for two reasons. Firstly, performing actions in parallel whenever possible leads to shorter plans with less execution time. Secondly, certain effects may be achievable only by simultaneous execution of actions. Suppose, for example, that all doors are equipped with a springlock which our delivery robot can open only by running into them and at the same time sending out the identification code to unlock the mechanism.

Concurrent actions are represented in the Fluent Calculus using a binary function, denoted "$\cdot$", by which singleton actions are joined together to denote their simultaneous execution. The constant "ϵ" ("no-op"), introduced in Section 6.4, will play the role as the empty concurrent action when decomposing a compound one. In the concurrent setting, the standard function *Do* and predicate *Poss* both range over the new sort of concurrent actions. Furthermore, the predicate $Affects(c, c_1)$ is introduced to denote that concurrent action c affects the usual effect of concurrent action c_1. Finally, to model actions with ramifications, the functions *DirState*, $DirEffect^+$, and $DirEffect^-$ map a concurrent action and a situation to, resp., the resulting state after the direct effects, all positive, and all negative effects of a concurrent action. These modifications of the signature are summarized in the following definition.

Definition 26 The *Fluent Calculus for concurrency* is a sorted second-order logic language which includes

1. Sort CONCURRENT such that ACTION < CONCURRENT
2. Functions

$$\begin{aligned}
\epsilon :&\ \mapsto \text{CONCURRENT}\\
\cdot :&\ \text{CONCURRENT} \times \text{CONCURRENT} \mapsto \text{CONCURRENT}\\
Do :&\ \text{CONCURRENT} \times \text{SIT} \mapsto \text{SIT}\\
DirState, DirEffect^+, DirEffect^- :&\ \text{CONCURRENT} \times \text{SIT} \mapsto \text{STATE}
\end{aligned}$$

3. Predicates

$$\mathit{Poss}\colon \text{CONCURRENT} \times \text{STATE} \qquad\qquad \mathit{Affects}\colon \text{CONCURRENT} \times \text{CONCURRENT}$$

□

In what follows, variables of sort CONCURRENT are denoted by the letter c, possibly with sub- or superscripts. For example, $\mathit{Pickup}(\mathit{Projector}) \cdot \mathit{SendId} \cdot c$ denotes a concurrent action. Similar to abbreviation (1) we will use the macro $\mathit{In}(c_1, c)$ to denote that concurrent action c_1 is included in concurrent action c:

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

To capture the intended properties of the connection function which combines single actions into concurrent ones, we add axioms for $\cdot$ and ϵ which have the same form as the foundational axioms for states:

$$\begin{aligned}(c_1 \cdot c_2) \cdot c_3 &= c_1 \cdot (c_2 \cdot c_3)\\ c_1 \cdot c_2 &= c_2 \cdot c_1\\ c \cdot c &= c\\ c \cdot \epsilon &= c\end{aligned}$$

$$\mathit{In}(a, a_1 \cdot c) \supset a = a_1 \vee \mathit{In}(a, c)$$

The foundational axioms $\mathcal{F}_{conc}$ of the Fluent Calculus for concurrency comprises this equational theory along with the following axioms, which stipulate that doing nothing has no effect and that a successor state is the result of ramifying the direct effects of a concurrent action:

$$\mathit{DirState}(\epsilon, s) = \mathit{State}(s) \wedge \mathit{DirEffect}^+(\epsilon, s) = \emptyset \wedge \mathit{DirEffect}^-(\epsilon, s) = \emptyset$$
$$\mathit{Ramify}(\mathit{DirState}(c, s), \mathit{DirEffect}^+(c, s), \mathit{DirEffect}^-(c, s), \mathit{State}(\mathit{Do}(c, s)))$$

The possibility to perform actions concurrently is specified using a combined precondition axiom.

Definition 27 A *concurrent action precondition axiom* is of the form

$$\mathit{Poss}(c, z) \equiv \Pi(c, z)$$

where $\Pi(c, z)$ is a pure state formula in z. □

For example, let $\mathit{RunInto} : \text{DOOR} \mapsto \text{ACTION}$ denote the action of running into a door, and consider the simple action precondition axiom $\mathit{Poss}(\mathit{RunInto}(d), z) \equiv \mathit{Holds}(\mathit{Closed}(d), z) \wedge \mathit{Holds}(\mathit{AtDoor}(d), z)$, that is, running into a door is possible only if the door is closed and the robot is next to it. Regarding our robot, we may assume that it can always send out the identification code simultaneously with any other action, hence the following precondition axiom:

$$\mathit{Poss}(c, z) \equiv (\exists a)\,(c = a \vee c = \mathit{SendId} \cdot a) \wedge (\forall a)\,(\mathit{In}(a, c) \supset \mathit{Poss}(a, z)) \tag{42}$$

In case two or more actions are performed simultaneously and do not interfere, the combined effect is the sum of the effects of the singleton actions. The accumulation of effects is modeled on the basis of recursive effect specifications [66]. This notion allows for specifying the (direct) effect of an action relative to the effect of arbitrary other actions performed concurrently.

Definition 28 Let $A_1, \ldots, A_k$ be function symbols with range ACTION $(k \geq 1)$. A *recursive update axiom* for $A_1, \ldots, A_k$ is of the form

$$\begin{array}{l} Poss(\alpha(\vec{x}) \cdot c, s) \,\wedge\, \Delta(\vec{x}, State(s)) \,\supset \\ \quad DirState(\alpha(\vec{x}) \cdot c, s) = DirState(c, s) \circ \vartheta^+ - \vartheta^- \wedge \\ \quad DirEffect^+(\alpha(\vec{x}) \cdot c, s) = DirEffect^+(c, s) \circ \vartheta^+ \wedge \\ \quad DirEffect^-(\alpha(\vec{x}) \cdot c, s) = DirEffect^-(c, s) \circ \vartheta^- \end{array} \tag{43}$$

where $\alpha(\vec{x}) = A_1(\vec{x}_1) \cdot \ldots \cdot A_k(\vec{x}_k)$, $\Delta(\vec{x}, z)$ is a pure state formula in z and ϑ^- and ϑ^+ are fluent collections.

A *disjunctive recursive update axiom* is of the form

$$\begin{array}{l} Poss(\alpha(\vec{x}) \cdot c, s) \,\wedge\, \Delta(\vec{x}, State(s)) \,\supset \\ \quad (\exists \vec{y}_1)\,(\,DirState(\alpha(\vec{x}) \cdot c, s) = DirState(c, s) \circ \vartheta_1^+ - \vartheta_1^- \wedge \\ \qquad\qquad DirEffect^+(\alpha(\vec{x}) \cdot c, s) = DirEffect^+(c, s) \circ \vartheta_1^+ \wedge \\ \qquad\qquad DirEffect^-(\alpha(\vec{x}) \cdot c, s) = DirEffect^-(c, s) \circ \vartheta_1^- \,\wedge\, \Theta_1\,) \\ \quad \vee \ldots \vee \\ \quad (\exists \vec{y}_n)\,(\,DirState(\alpha(\vec{x}) \cdot c, s) = DirState(c, s) \circ \vartheta_n^+ - \vartheta_n^- \wedge \\ \qquad\qquad DirEffect^+(\alpha(\vec{x}) \cdot c, s) = DirEffect^+(c, s) \circ \vartheta_n^+ \wedge \\ \qquad\qquad DirEffect^-(\alpha(\vec{x}) \cdot c, s) = DirEffect^-(c, s) \circ \vartheta_n^- \,\wedge\, \Theta_n\,) \end{array}$$

where $\alpha(\vec{x}) = A_1(\vec{x}_1) \cdot \ldots \cdot A_k(\vec{x}_k)$, $\Delta(\vec{x}, z)$ is a pure state formula in z, ϑ_i^+ and ϑ_i^- are fluent collections, and Θ_i is a first-order formula without terms of any reserved sort ($1 \leq i \leq n$; $n \geq 1$). □

Put in words, ϑ^+ and ϑ^- are the *additional* positive and negative, resp., effects which occur if $A_1, \ldots, A_k$ are performed *besides* c. Since under specific circumstances actions may interfere when performed simultaneously, the condition Δ of a recursive update axiom may include qualifications based on the *Affects* predicate. The combined effect of interfering actions can then be specified by a separate update axiom.

For example, the only interference among actions in our modified delivery domain (where doors are equipped with a springlock) happens when *SendId* and *RunInto* are performed concurrently:

$$\begin{array}{l} Affects(c, c_1) \,\equiv \\ \quad (\exists d)\, c_1 = RunInto(d) \wedge In(SendId, c) \,\vee\, c_1 = SendId \wedge (\exists d)\, In(RunInto(d), c) \end{array} \tag{44}$$

Consider the actions *Pickup*, *SendId*, and *RunInto*, which in view of concurrent execution are suitably specified by the following collection of (deterministic) recursive update axioms:

$$\begin{array}{l} Poss(Pickup(x) \cdot c, s) \,\supset \\ \quad DirState(Pickup(x) \cdot c, s) = DirState(c, s) \circ Carries(x) \wedge \\ \quad DirEffect^+(Pickup(x) \cdot c, s) = DirEffect^+(c, s) \circ Carries(x) \wedge \\ \quad DirEffect^-(Pickup(x) \cdot c, s) = DirEffect^-(c, s) \\ Poss(RunInto(d) \cdot c, s) \wedge \neg Affects(c, RunInto(d)) \,\supset \\ \quad DirState(RunInto(d) \cdot c, s) = DirState(c, s) \wedge \\ \quad DirEffect^+(RunInto(d) \cdot c, s) = DirEffect^+(c, s) \wedge \\ \quad DirEffect^-(RunInto(d) \cdot c, s) = DirEffect^-(c, s) \end{array}$$

$$
\begin{array}{l}
\mathit{Poss}(\mathit{SendId} \cdot c, s) \wedge \mathit{Holds}(\mathit{AtDoor}(d), s) \wedge \neg \mathit{Holds}(\mathit{Closed}(d), s) \ \supset \\
\quad \mathit{DirState}(\mathit{SendId} \cdot c, s) = \mathit{DirState}(c, s) \circ \mathit{Closed}(d) \wedge \\
\quad \mathit{DirEffect}^{+}(\mathit{SendId} \cdot c, s) = \mathit{DirEffect}^{+}(c, s) \circ \mathit{Closed}(d) \wedge \\
\quad \mathit{DirEffect}^{-}(\mathit{SendId} \cdot c, s) = \mathit{DirEffect}^{-}(c, s) \\
\mathit{Poss}(\mathit{SendId} \cdot c, s) \wedge [\, \neg(\exists d)\, \mathit{Holds}(\mathit{AtDoor}(d), s) \vee \\
\qquad\qquad\qquad\qquad \mathit{Holds}(\mathit{AtDoor}(d), s) \wedge \mathit{Holds}(\mathit{Closed}(d), s) \wedge \neg \mathit{Affects}(c, \mathit{SendId})] \ \supset \\
\quad \mathit{DirState}(\mathit{SendId} \cdot c, s) = \mathit{DirState}(c, s) \wedge \\
\quad \mathit{DirEffect}^{+}(\mathit{SendId} \cdot c, s) = \mathit{DirEffect}^{+}(c, s) \wedge \\
\quad \mathit{DirEffect}^{-}(\mathit{SendId} \cdot c, s) = \mathit{DirEffect}^{-}(c, s) \\
\mathit{Poss}(\mathit{RunInto}(d) \cdot \mathit{SendId} \cdot c, s) \ \supset \\
\quad \mathit{DirState}(\mathit{RunInto}(d) \cdot \mathit{SendId} \cdot c, s) = \mathit{DirState}(c, s) - \mathit{Closed}(d) \wedge \\
\quad \mathit{DirEffect}^{+}(\mathit{RunInto}(d) \cdot \mathit{SendId} \cdot c, s) = \mathit{DirEffect}^{+}(c, s) \wedge \\
\quad \mathit{DirEffect}^{-}(\mathit{RunInto}(d) \cdot \mathit{SendId} \cdot c, s) = \mathit{DirEffect}^{-}(c, s) \circ \mathit{Closed}(d)
\end{array}
$$

That is, picking up an object has the usual effect no matter what actions are performed concurrently. (The other actions Go, Enter, Drop of our robot can be reformulated in a similar fashion.) Running into a door does not have any effect if not affected by a concurrent action. Transmitting the identification code at an open door causes the latter to close; otherwise, the action has no effect if not affected by some concurrent action. Finally, running into a door and sending out the code simultaneously has the effect that the door opens.

The overall direct effect of a concurrent action is determined by a set of recursive equations which are obtained as the consequents of instances of the appropriate update axioms. For example, consider the initial specification

$$
\begin{array}{l}
\mathit{Holds}(\mathit{InRoom}(\mathit{R403}), S_0) \wedge \mathit{Holds}(\mathit{AtDoor}(\mathit{D23}), S_0) \\
\wedge\, \mathit{Holds}(\mathit{Request}(\mathit{R403}, \mathit{Projector}, \mathit{R404}), S_0) \wedge \neg \mathit{Holds}(\mathit{Closed}(\mathit{D23}), S_0) \\
\wedge\, (\forall x)\, \neg \mathit{Holds}(\mathit{Carries}(x), S_0) \wedge (\forall x)\, \neg \mathit{Holds}(\mathit{Attached}(x, \mathit{Projector}), S_0)\,)
\end{array}
$$

and the concurrent action $\mathit{Pickup}(\mathit{Projector}) \cdot \mathit{SendId}$, whose executability in S_0 is given by (42) in conjunction with (5). Because of $\neg \mathit{Affects}(\mathit{Pickup}(\mathit{Projector}), \mathit{SendId})$, which is due to (44), from the above recursive update axioms we can set up these state equations:

$$
\begin{array}{rcl}
\mathit{DirState}(\mathit{Pickup}(\mathit{Projector}) \cdot \mathit{SendId}, S_0) & = & \mathit{DirState}(\mathit{SendId}, S_0) \circ \mathit{Carries}(\mathit{Projector}) \\
\mathit{DirEffect}^{+}(\mathit{Pickup}(\mathit{Projector}) \cdot \mathit{SendId}, S_0) & = & \mathit{DirEffect}^{+}(\mathit{SendId}, S_0) \circ \mathit{Carries}(\mathit{Projector}) \\
\mathit{DirEffect}^{-}(\mathit{Pickup}(\mathit{Projector}) \cdot \mathit{SendId}, S_0) & = & \mathit{DirEffect}^{-}(\mathit{SendId}, S_0) \\
\mathit{DirState}(\mathit{SendId}, S_0) & = & \mathit{DirState}(\epsilon, S_0) \circ \mathit{Closed}(\mathit{D23}) \\
\mathit{DirEffect}^{+}(\mathit{SendId}, S_0) & = & \mathit{DirEffect}^{+}(\epsilon, S_0) \circ \mathit{Closed}(\mathit{D23}) \\
\mathit{DirEffect}^{-}(\mathit{SendId}, S_0) & = & \mathit{DirEffect}^{-}(\epsilon, S_0)
\end{array}
$$

Hence, the result of the concurrent action is inferred by decomposing the latter and successively inferring the effects of the components. From $\mathcal{F}_{\mathit{conc}}$ and the above initial specification it then follows that

$$
\begin{array}{l}
(\exists z)\ \mathit{State}(\mathit{Do}(\mathit{Pickup}(\mathit{Projector}) \cdot \mathit{SendId}, S_0)) \ = \\
\qquad \mathit{InRoom}(\mathit{R403}) \circ \mathit{AtDoor}(\mathit{D23}) \circ \mathit{Request}(\mathit{R403}, \mathit{Projector}, \mathit{R404}) \circ \\
\qquad \mathit{Carries}(\mathit{Projector}) \circ \mathit{Closed}(\mathit{D23}) \circ z
\end{array}
\tag{45}
$$

Our approach to concurrency is summarized in the following extended definition of axiomatizations of concurrent worlds.

Definition 29 A *Fluent Calculus domain axiomatization with concurrency and ramifications* consists of a set of state constraints, a circumscribed axiomatization of causal relations, a unique simple action precondition axiom for each function symbol with range ACTION, a concurrent action precondition axiom, a set of recursive update axioms, foundational axioms $\mathcal{F}_{state}$, $\mathcal{F}_{ramify}$, and $\mathcal{F}_{conc}$, plus possible further domain-specific axioms. □

Again, this definition can easily be reconciled with our approach to knowledge and sensing if the concept of knowledge update is suitably generalized to account for concurrent actions.

Concurrency in Flux

Concurrent actions are modeled in FLUX as lists of action terms. As opposed to states, we can reasonably assume that a robot has complete knowledge of its concurrent actions, which considerably simplifies their treatment.

A concurrent action precondition axiom $Poss(c, z) \equiv \Pi(c, z)$ is encoded by the clause

$$Poss(c, z) \leftarrow \Pi(c, z)$$

Action interference is encoded by a clause with head *Affects*; Appendix A.5 shows the encoding of (44) as example. For efficiency reasons, the state update is inferred only once, on the basis of the combined positive and negative direct effects. To this end, recursive update axioms (43) are encoded by clauses of the following form, where $DirEffect(c, z, e^+, e^-)$ means that in state z concurrent action c has positive and negative, resp., direct effects e^+, e^-:[12]

$$\begin{aligned}
&DirEffect(c, z, e^+, e^-) \leftarrow \\
&\quad Subset([A_1(\vec{x}_1), \ldots, A_k(\vec{x}_k)], c),\ Subtract(c, [A_1(\vec{x}_1), \ldots, A_k(\vec{x}_k)], c_1), \\
&\quad \Delta(\vec{x}_1, \ldots, \vec{x}_k, z),\ DirEffect(c_1, z, e_1^+, e_1^-),\ Plus(e_1^+, \vartheta^+, e^+),\ Plus(e_1^-, \vartheta^-, e^-)
\end{aligned}$$

(Appendix A.5 shows the FLUX encodings for the actions of our robot tailored to concurrency.) The base case of this recursion reflects foundational axioms $\mathcal{F}_{conc}$:

```
dir_effect([], _, [], []).
```

Disjunctive recursive update axioms are encoded along the same line. Finally, a general clause defines the update caused by concurrent actions:

```
state_update(Z1, C, Z2) :-
   dir_effect(C, Z1, ThetaP, ThetaN),
   update(Z1, ThetaP, ThetaN, Z), ramify(Z, ThetaP, ThetaN, Z2).
```

As an example, the modified FLUX program and the query

$$\begin{aligned}
&Holds(InRoom(R403), z_0),\ Holds(AtDoor(D23), z_0), \\
&Holds(Request(R403, Projector, R404), z_0), \\
&NotHolds(Closed(D23), z_0),\ NotHoldsAll([x], Carries(x), z_0), \\
&NotHoldsAll([x], Attached(x, Projector), z_0),\ Consistent(z_0),\ DuplicateFree(z_0) \\
&Poss([Pickup(Projector), SendId], z_0),\ StateUpdate(z_0, [Pickup(Projector), SendId], z_1)
\end{aligned}$$

yield the answer (cf. (45)),

$$\begin{aligned}
z_1 = [\,&Closed(D23), Carries(Projector), \\
&InRoom(R403), AtDoor(D23), Request(R403, Projector, R404) \mid z]
\end{aligned}$$

[12]The standard Eclipse predicates $Subset(l_1, l_2)$ and $Subtract(l_2, l_1, l_3)$ used below denote, resp., that all elements in list l_1 occur in list l_2 and that list l_3 contains all elements in l_2 but those in l_1.

9 Summary and Discussion

We have presented the Fluent Calculus as a comprehensive specification and programming language for Cognitive Robotics in which are combined a variety of challenging aspects of complex environments. Furnishing the calculus with a new axiomatic foundation, we have first of all overcome an important limitation of [65] caused by relying on the notion of unification completeness [62, 22]: Defining inequality of state terms as non-unifiability wrt. AC1 did not permit any domain-specific equalities like $\mathit{Office}(\mathit{Alice}) = \mathit{R402}$ since this leads to a contradiction given that, e.g., the state terms $\mathit{InRoom}(\mathit{Office}(\mathit{Alice}))$ and $\mathit{InRoom}(\mathit{R402})$ are not AC1-unifiable. The new, conceptually even simpler axiomatic foundation relates equality of state terms to equality of fluents, thus allowing the latter to be defined independently. Moreover, states directly correspond to sets of fluents under the new foundational axioms, whereas previously multisets of fluents have been used to represent states [21, 65].

We have proved that state update axioms solve the Frame Problem under the new algebraic foundation (Theorem 7). Moreover, the axiomatic characterization of states has also paved the way towards the system FLUX. The underlying constraint handling rules as well as the core logic program have been formally verified against the foundations of the Fluent Calculus.

FLUX is distinguished mainly by two features. First, it is especially designed for specifying and computing with incomplete state specifications. Second, it exploits the solution to the inferential Frame Problem of the Fluent Calculus: Effects of actions are computed as “local surgeries” [46] on a list of fluents and the accompanying constraints so that most of a state remains unchanged, which is the essence of a computationally effective solution to the Frame Problem.

The most well-known existing language for Cognitive Robotics, GOLOG [30],[13] is based on successor state axioms in the Situation Calculus. GOLOG includes the concept of high-level robot control programs to guide search, which is essential for solving problems of practical size. This concept furnishes a ready approach to programming heuristics for planning in FLUX as well. The main advantage of FLUX in comparison is that plain GOLOG as well as its extensions [13, 25, 5] apply the closed-world assumption [51] to an initial state specification, hence do not support incomplete state knowledge. This limitation is overcome in the two versions introduced in [10], where either a propositional theorem prover is used to establish entailment of formulas about the initial situation, or where an incomplete specification is compiled into a set of so-called prime implicates, which are then used for the same purpose. As it stands, either approach is restricted to essentially propositional domains and to incomplete knowledge of the initial situation only, although it seems feasible to extend both approaches in such a way that knowledge about other situations is first regressed to the initial situation and then processed analogously.

A further fundamental difference between FLUX and GOLOG concerns the way in which the value of a fluent in a particular situation is determined, e.g., in order to verify an action precondition or the satisfaction of a goal condition. While in FLUX the value is readily available from the current list of fluents and constraints, in GOLOG the current situation term needs to be unfolded either until the situation is reached where the fluent was caused true or false by the preceding action or completely down to the initial situation. An analogous difference applies to other existing systems for Cognitive Robotics, like those based on the Event Calculus [58, 59].

We have shown how disjunctive state update axioms along the line of [33, 67] and existential quantification can be used to model nondeterministic actions and uncertainty in the Fluent Calculus and FLUX. A related extension of GOLOG supports reasoning with probability distri-

[13]Among others, a recent robotics application of this system was the high-level end of an autonomous museum tour guide [20, 6].

butions over a discrete space of alternative results of actions [5]. Augmenting FLUX by elements of decision theory along this line is a promising direction of future.

We have developed and integrated into the Fluent Calculus a formal account of a robot's knowledge about the state of its environment. Our approach is kept representationally and inferentially simple in that it avoids non-classical extensions to standard predicate logic. The formalism accounts for both knowledge preconditions of actions and information gathering actions which enhance the state knowledge of a robot. Our theory also provides simple means to reason about what a robot does not know and about branching plans based on conditional actions. The effect of actions on the mental state of a robot is specified by so-called knowledge update axioms, by which is solved the representational Frame Problem for knowledge. Moreover, knowledge update axioms have been shown to lay the foundations for overcoming the inferential aspect of this variant of the Frame Problem, too.

Knowledge and sensing actions were first investigated in [45] in the context of the Situation Calculus, and in [57] this approach was combined with the solution to the Frame Problem provided by successor state axioms. Other approaches to planning with sensing exist using special-purpose logics, e.g., [17, 12]. The basic idea of [45, 57] is to represent state knowledge by a binary situation-situation relation $K(s, s')$, meaning that as far as the robot knows in situation s it could as well be in situation s'. Hence, every given fact about any such s' is considered possible by the robot. Having readily available the explicit notion of a state in the Fluent Calculus, our formalization avoids this indirect encoding of state knowledge, which is intuitively less appealing because it seems that a robot should always know exactly which situation it is in, that is, which sequence of actions it has taken. Apart from this clash of intuitions, there is a more crucial difference between our approach and that of [45, 57]: The latter defines the effect of a non-sensing action a on the robot's state knowledge via the equivalence relation $K(Do(a, s), s'') \equiv (\exists s')\,(K(s, s') \wedge s'' = Do(a, s'))$. Hence, the very same successor state axioms apply to both the state update (when moving from s to $Do(a, s)$) and the knowledge update (when moving from s' to $s'' = Do(a, s')$). In contrast, with independent specifications of state and knowledge update, our formalism furnishes a ready approach for representing and reasoning about the ability to achieve goals based on possibly limited knowledge of the effects of actions. This separating what a user knows from what a robot knows distinguishes our theory from other existing accounts of sensing action and knowledge, too, such as [36, 2], where also non-sensing actions have identical effect on the external and internal states. On the other hand, knowledge in terms of possible situations allows nested application of the knowledge modality, as in $\mathit{Knows}(\mathit{Knows}(\mathit{Closed}(d), s_2), s_1)$, which is not a valid expression in our theory. This feature is of particular interest in multi-agent settings. Knowledge of other robot's knowledge can, however, be achieved in our approach by extending the *Knows* macro and the underlying *KState* relation by a third argument to distinguish the knowledge states of different robots.

Representing and reasoning about non-knowledge has previously been investigated in the context of the Situation Calculus [26, 27]. Two approaches to 'only knowing' have been offered, one of which is by a non-standard semantics while the other one is an axiomatization in classical logic but with two complex second-order axioms involved. Exploiting the reification of fluents and states, knowledge and non-knowledge can be expressed in our approach by comparatively simple equivalences based on a straightforward second-order sentence along with the standard semantics of classical logic. A further difference is that in [26, 27] the set of objects inhabiting a domain is fixed (namely, the set of natural numbers), whereas our formalization does not make any presuppositions in this regard. Our notion of a branching plan was inspired by the conditional action trees of [25]. A minor difference is that the latter are introduced as a new sort while our conditional actions and sequences are just pre-defined actions.

We have shown how sensing actions can be specified in FLUX and, by exploiting the representation technique for incomplete states, generated and verified as elements of plans. By assuming the reflective stance on knowledge, we have obtained an effective test of whether a property is known at the current state of the computation. As an outstanding feature, our implementation thus allows to solve planning problems with knowledge goals. In contrast, existing accounts of sensing in logic programming, such as [14, 25], suffer from the closed world assumption to the effect that it is not possible to verify whether a condition is known in a situation. As a consequence, it is left to the responsibility of the programmer to restrict the search space in such a way that only correct plans are generated. In particular, it is not possible to solve fully automatically planning problems where the goal is to acquire knowledge.

For the sake of simplicity, we have only considered off-line sensing. In [14] it has been argued that robots cannot fully plan ahead solutions to large, complex tasks. This requires interleaving off-line planning with sometimes committing to certain actions based on local criteria and without foreseeing the consequences in every detail.

Finally, we have shown how indirect effects and concurrent actions can be uniformly modeled in the Fluent Calculus and programmed in FLUX. To this end, we have adapted our theory [63] based on causal propagation—a concept which provides the most general solution to the Ramification Problem known today [55]. In particular cyclic causal dependencies [8] are dealt with correctly, which goes beyond most alternative theories, e.g., [32, 19, 61, 44]. We refer to [63] for a detailed discussion and comparison with some of these approaches as well as the one of [38], which has evolved into [16, 68]. In the series of papers [47, 48, 49], the attempt is made to define a unifying semantics for approaches to ramifications, which covers causal propagation and in particular our causal relationships. The notion of recursive update axioms for concurrency has been introduced in [66] in the context of modeling continuous change with the Fluent Calculus [64].

A The Delivery Domain

A.1 delivery_constraints.chr

```
handler delivery_constraints.
constraints at_door_unique/1, request_unique/1, door_of_room/2,
            attached_symmetric/1,
            carries_implies/3, carries_symmetric/1, carries_symmetric/2.

% Robot cannot be at two doors simultaneously

at_door_unique([F1|Z1])         <=> \+ F1=at_door(_) | at_door_unique(Z1).
at_door_unique([at_door(_)|Z1]) <=> not_holds_all(D, at_door(D), Z1).

% Robot cannot be at door not belonging to the room it is in

door_of_room(R,[F1|Z1])        <=> \+ F1=at_door(_) | door_of_room(Z1).
door_of_room(R,[at_door(D)|_]) <=> connects(D,R,_) | true.
door_of_room(R,[at_door(D)|_]) <=> \+ connects(D,R,_) | false.

% No two request for the same object

request_unique([F1|Z1])             <=> \+ F1=request(_,_,_) | request_unique(Z1).
request_unique([request(_,X,_)|Z1]) <=> not_holds_all([R1,R2], request(R1,X,R2), Z1),
                                        request_unique(Z1).

% CHRs for ramification

% Whenever attached(x,y) then also attached(y,x)

attached_symmetric([F1|Z1])            <=> \+ F1=attached(_,_)
                                           | attached_symmetric(Z1).
attached_symmetric([attached(X,Y)|Z1]) <=> holds(attached(Y,X), Z1, Z2),
                                           attached_symmetric(Z2).

% Whenever carries(x) then also carries(y)

carries_implies(X,Y,[F1|Z1])         <=> \+ F1=carries(_) | carries_implies(X,Y,Z1).
carries_implies(X,Y,[carries(X)|Z1]) <=> holds(carries(Y), Z1).
carries_implies(X,Y,[carries(Y)|_]   <=> true.
carries_implies(X,Y,[carries(W)|Z1]) <=> \+ X=W | carries_implies(X,Y,Z1).

% Whenever attached(x,y) then carries_implies(x,y,z),
%    where z includes all fluents carries(_) of a state

carries_symmetric(Z) <=> carries_symmetric(Z, []).

carries_symmetric([F1|Z1], Zp)            <=> \+ F1=carries(_), \+ F1=attached(_,_)
                                              | carries_symmetric(Z1, Zp).
carries_symmetric([carries(X)|Z1], Zp)    <=> carries_symmetric(Z1, [carries(X)|Zp]).
carries_symmetric([attached(X,Y)|Z1], Zp) <=> append(Zp, Z1, Z2)
                                              | carries_implies(X, Y, Z2),
                                                carries_symmetric(Z1, Zp).
```

A.2 delivery.pl

```
:- [flux].
:- chr2pl(delivery_constraints), [delivery_constraints].

c(d12, r401, r402).
c(d23, r402, r403).
c(d34, r403, r404).
c(da1, alley, r401).
c(da2, alley, r402).
c(da3, alley, r403).
c(da4, alley, r404).
connects(D, X, Y) :- c(D, X, Y) ; c(D, Y, X).

office(alice, r402).
office(bob, r404).

consistent(Z) :- holds(in_room(R), Z, Z1), not_holds_all(R, in_room(R), Z1),
                 at_door_unique(Z),
                 door_of_room(R, Z),
                 request_unique(Z).

poss(go(D), Z) :-
   holds(in_room(R), Z), connects(D, R, _).
poss(open(D), Z) :-
   holds(at_door(D), Z), ( holds(has_key_code(D), Z) ; not_holds(closed(D), Z) ).
poss(enter(R), Z) :-
   holds(at_door(D), Z), holds(in_room(R1), Z),
   connects(D, R1, R), not_holds(closed(D), Z).
poss(pickup(X), Z) :-
   holds(request(R,X,_), Z), holds(in_room(R), Z), not_holds(carries(X), Z).
poss(drop(X), Z) :-
   holds(carries(X), Z), holds(request(_,X,R), Z), holds(in_room(R), Z).
poss(ask(P, D), Z) :-
   office(P, R1), holds(in_room(R1), Z), holds(request(R1,_,R2), Z),
   connects(D, R1, R2), holds(closed(D), Z), not_holds(has_key_code(D), Z).

not_poss(go(D), Z) :-
   holds(in_room(R), Z), \+ connects(D, R, _).
not_poss(open(D), Z) :-
   not_holds(at_door(D), Z) ; not_holds(has_key_code(D), Z), holds(closed(D), Z).
not_poss(enter(R), Z) :-
   holds(in_room(R), Z) ; not_holds_all(D, at_door(D), Z) ;
   holds(at_door(D), Z), ( holds(closed(D), Z) ; \+ connects(D, _, R) ).
not_poss(pickup(X), Z) :-
   holds(carries(X), Z) ; not_holds_all([R1,R2], request(R1,X,R2), Z) ;
   holds(request(R,X,_), Z), not_holds(in_room(R), Z).
not_poss(drop(X), Z) :-
   not_holds(carries(X), Z) ;
   holds(request(_, X, R), Z), not_holds(in_room(R), Z).
not_poss(ask(P, D), Z) :-
   office(P, R1),
   ( \+ connects(D, R1, _) ;
     connects(D, R1, R2), ( not_holds(in_room(R1), Z) ;
                            not_holds_all(X, request(R1,X,R2), Z) ;
                            not_holds(closed(D), Z) ; holds(has_key_code(D), Z)
                          ) ).

state_update(Z1, go(D), Z2) :-
   holds(at_door(D1), Z1), neq(D1, D), update(Z1, [at_door(D)], [at_door(D1)], Z2) ;
   ( holds(at_door(D), Z1) ; not_holds_all(D1, at_door(D1), Z1) ),
     update(Z1, [at_door(D)], [], Z2).
state_update(Z1, open(D), Z2) :-
   holds(closed(D), Z1), update(Z1, [], [closed(D)], Z2) ;
   not_holds(closed(D), Z1), equal(Z1, Z2).
state_update(Z1, enter(R), Z2) :-
   holds(in_room(R1), Z1), update(Z1, [in_room(R)], [in_room(R1)], Z2).
state_update(Z1, pickup(X), Z2) :-
   update(Z1, [carries(X)], [], Z2).
state_update(Z1, drop(X), Z2) :-
   update(Z1, [], [carries(X),request(_,X,_)], Z2).
state_update(Z1, ask(_, D), Z2) :-
   update(Z1, [has_key_code(D)], [], Z2) ;
   update(Z1, [], [closed(D)], Z2).
```

A.3 delivery_sense.pl

```
is_room(alley). is_room(r401). is_room(r402). is_room(r403). is_room(r404).

is_door(da1). is_door(da2). is_door(da3). is_door(da4).
is_door(d12). is_door(d23). is_door(d34).

is_object(projector). is_object(document_folder).

is_person(alice). is_person(bob).

is_fluent(in_room(R)) :- is_room(R).
is_fluent(at_door(D)) :- is_door(D).
is_fluent(closed(D)) :- is_door(D).
is_fluent(has_key_code(D)) :- is_door(D).
is_fluent(carries(X)) :- is_object(X).
is_fluent(request(R1,X,R2)) :- is_room(R1), is_object(X), is_room(R2).

primitive_action(go(D)) :- is_door(D).
primitive_action(enter(R)) :- is_room(R).
primitive_action(pickup(X)) :- is_object(X).
primitive_action(drop(X)) :- is_object(X).
primitive_action(ask(P,D)) :- is_person(P), is_door(D).
primitive_action(send_id).
primitive_action(sense(closed(D))) :- is_door(D).

poss(send_id, _).
poss(sense(F), Z) :- F=closed(D), holds(at_door(D), Z).

not_poss(sense(F), Z) :- \+ F=closed(_) ; F=closed(D), not_holds(at_door(D), Z).
```

```
state_update(Z, sense(F), Z, SV) :-
   holds(F, Z), SV = F ; not_holds(F, Z), SV = -(F).

state_update(Z1, send_id, Z2) :-
   holds(at_door(D), Z1), holds(closed(D), Z1), update(Z1, [], [closed(D)], Z2) ;
   holds(at_door(D), Z1), not_holds(closed(D), Z1), update(Z1, [closed(D)], [], Z2) ;
   not_holds_all(D, at_door(D), Z1), equal(Z1, Z2).
```

A.4 delivery_ramification.pl

```
consistent(Z) :- holds(in_room(_), Z, Z1), not_holds_all(R, in_room(R), Z1),
                 at_door_unique(Z),
                 door_of_room(R, Z),
                 request_unique(Z),
                 attached_symmetric(Z),
                 carries_symmetric(Z).

causes(carries(X), carries(Y), Z) :- holds(attached(X,Y), Z).

causes(-(carries(X)), -(carries(Y)), Z) :- holds(attached(X,Y), Z).

causes(-(carries(X), -(request(R1,X,R2)), Z) :- holds(request(R1,X,R2), Z).

causes(-(request(_,X,R2), request(R,X,R2), Z) :- holds(in_room(R), Z), neq(R, R2).

not_causes(Z, EP, [E|EN]) :-
   ( \+ E=carries(_), \+ E=request(_,_,_) ;
     E=carries(X),
        (not_holds_all([R1,R2], request(R1,X,R2), Z); holds(request(_,X,_), EN)) ;
     E=request(_,X,R), (holds(in_room(R), Z); holds(request(_,X,_), EP)) ),
   not_causes(Z, EP, EN).

not_causes(Z, _, []) :- carries_symmetric(Z).

state_update(Z1, pickup(X), Z2) :-
   update(Z1, [carries(X)], [], Z), ramify(Z, [carries(X)], [], Z2).
state_update(Z1, drop(X), Z2) :-
   update(Z1, [], [carries(X)], Z),
   ramify(Z, [], [carries(X)], Z2).
```

A.5 delivery_concurrency.pl

```
affects(C, C1) :- subset([run_into(_)], C), C1=send_id ;
                  subset([send_id], C), C1=run_into(_).

poss(send_id, _).
poss(run_into(D), Z) :- holds(closed(D), Z), holds(at_door(D), Z).

not_poss(run_into(D), Z) :- not_holds(closed(D), Z) ; not_holds(at_door(D), Z).

poss(C, Z) :- ( C=[A] ; C=[send_id,A] ; C=[A,send_id] ),
              \+ (member(A, C), not_poss(A, Z)).
```

```
dir_effect(C, Z, Ep, En) :-
   subset([go(D)], C), subtract(C, [go(D)], C1),
   dir_effect(C1, Z, Ep1, En1),
   ( holds(at_door(D1), Z), neq(D1, D), append([at_door(D1)], En1, En) ;
     (holds(at_door(D), Z) ; not_holds_all(D1, at_door(D1), Z)), En=En1 ),
   append([at_door(D)], Ep1, Ep).
dir_effect(C, Z, Ep, En) :-
   subset([enter(R)], C), subtract(C, [enter(R)], C1),
   dir_effect(C1, Z, Ep1, En1),
   holds(in_room(R1), Z),
   append([in_room(R1)], En1, En), append([in_room(R)], Ep1, Ep).
dir_effect(C, Z, Ep, En) :-
   subset([pickup(X)], C), subtract(C, [pickup(X)], C1),
   dir_effect(C1, Z, Ep1, En), append([carries(X)], Ep1, Ep).
dir_effect(C, Z, Ep, En) :-
   subset([drop(X)], C), subtract(C, [drop(X)], C1),
   dir_effect(C1, Z, Ep, En1),
   append([carries(X),request(_,X,_)], En1, En).
dir_effect(C, Z, Ep, En) :-
   subset([run_into(D)], C), subtract(C, [run_into(D)], C1),
   \+ affects(run_into(D), C1), dir_effect(C1, Z, Ep, En).
dir_effect(C, Z, Ep, En) :-
   subset([send_id], C), subtract(C, [send_id], C1),
   dir_effect(C1, Z, Ep1, En1),
   ( holds(at_door(D), Z), not_holds(closed(D), Z),
     En=En1, append([closed(D)], Ep1, Ep) ) ;
   ( ( not_holds_all(D, at_door(D), Z) ;
       (holds(at_door(D), Z), holds(closed(D), Z), \+ affects(send_id, C1)) ),
     En=En1, Ep=Ep1 ).
dir_effect(C, Z, Ep, En) :-
   subset([run_into(D),send_id], C), subtract(C, [run_into(D),send_id], C1),
   dir_effect(C1, Z, Ep, En1), append([closed(D)], En, En1).
```

References

[1] Andrew B. Baker. A simple solution to the Yale Shooting problem. In R. Brachman, H. J. Levesque, and R. Reiter, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 11–20, Toronto, Kanada, 1989. Morgan Kaufmann.

[2] Chitta Baral and Tran Cao Son. Approximate reasoning about actions in presence of sensing and incomplete information. In J. Maluszynski, editor, *Proceedings of the International Logic Programming Symposium (ILPS)*, pages 387–401, Port Jefferson, NY, October 1997. MIT Press.

[3] Wolfgang Bibel. A deductive solution for plan generation. *New Generation Computing*, 4:115–132, 1986.

[4] Wolfgang Bibel. Let's plan it deductively! *Artificial Intelligence*, 103(1–2):183–208, 1998.

[5] Craig Boutilier, Ray Reiter, Mikhail Soutchanski, and Sebastian Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In H. Kautz and B. Porter, editors, *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 355–362, Austin, TX, July 2000.

[6] Wolfram Burgard, Armin B. Cremers, Dieter Fox, Dirk Hähnel, Gerhard Lakemeyer, Dieter Schulz, Walter Steiner, and Sebastian Thrun. Experiences with an interactive museum tour-guide robot. *Artificial Intelligence*, 14(1–2):3–55, 1999.

[7] Martin Davis. First order logic. In D. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, chapter 2, pages 31–65. Oxford University Press, 1993.

[8] Marc Denecker, Daniele Theseider Dupré, and Kristof Van Belleghem. An inductive definition approach to ramifications. *Electronic Transactions on Artificial Intelligence*, 2(1–2):25–67, 1998. URL: `http://www.ep.liu.se/ea/cis/1998/007/`.

[9] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[10] Alberto Finzi, Fiora Pirri, and Ray Reiter. Open world planning in the situation calculus. In H. Kautz and B. Porter, editors, *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 754–760, Austin, TX, July 2000.

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

[12] Giuseppe De Giacomo, Luca Iocchi, Daniele Nardi, and Riccardo Rosati. Planning with sensing for a mobile robot. In *Proceedings of the European Conference on Planning (ECP)*, volume 1348 of *LNAI*, pages 158–170. Springer, 1997.

[13] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. Reasoning about concurrent execution, prioritized inputs, and exogenous actions in the situation calculus. In M. Pollack, editor, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1221–1226, Nagoya, Japan, August 1997. Morgan Kaufmann.

[14] Giuseppe De Giacomo and Hector J. Levesque. An incremental interpreter for high-level programs with sensing. In H. Levesque and F. Pirri, editors, *Logical Foundations for Cognitive Agents*, pages 86–102. Springer, 1999.

[15] Matthew L. Ginsberg and David E. Smith. Reasoning about action I: A possible worlds approach. *Artificial Intelligence*, 35:165–195, 1988.

[16] Enrico Giunchiglia, G. Neelakantan Kartha, and Vladimir Lifschitz. Representing action: Indeterminacy and ramifications. *Artificial Intelligence*, 95:409–443, 1997.

[17] Keith Golden and Daniel Weld. Representing sensing actions: The middle ground revisited. In L. C. Aiello, J. Doyle, and S. Shapiro, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 174–185, Cambridge, MA, November 1996. Morgan Kaufmann.

[18] Gerd Große, Steffen Hölldobler, and Josef Schneeberger. Linear Deductive Planning. *Journal of Logic and Computation*, 6(2):233–262, 1996.

[19] Joakim Gustafsson and Patrick Doherty. Embracing occlusion in specifying the indirect effects of actions. In L. C. Aiello, J. Doyle, and S. Shapiro, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 87–98, Cambridge, MA, November 1996. Morgan Kaufmann.

[20] Dirk Hähnel, Wolfram Burgard, and Gerhard Lakemeyer. GOLEX: Bridging the gap between logic (GOLOG) and a real robot. In O. Herzog and A. Günter, editors, *Proceedings of the German Annual Conference on Artificial Intelligence (KI)*, volume 1504 of *LNAI*, pages 165–176, Bremen, Germany, September 1998. Springer.

[21] Steffen Hölldobler and Josef Schneeberger. A new deductive approach to planning. *New Generation Computing*, 8:225–244, 1990.

[22] Steffen Hölldobler and Michael Thielscher. Computing change and specificity with equational logic programs. *Annals of Mathematics and Artificial Intelligence*, 14(1):99–133, 1995.

[23] Joxan Jaffar and Michael J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[24] Robert A. Kowalski. Predicate logic as a programming language. In *Proceedings of the Congress of the International Federation for Information Processing (IFIP)*, pages 569–574. Elsevier, 1974.

[25] Gerhard Lakemeyer. On sensing and off-line interpreting GOLOG. In H. Levesque and F. Pirri, editors, *Logical Foundations for Cognitive Agents*, pages 173–189. Springer, 1999.

[26] Gerhard Lakemeyer and Hector J. Levesque. $\mathcal{AOL}$: A logic of acting, sensing, knowing, and only knowing. In A. G. Cohn, L. K. Schubert, and S. C. Shapiro, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 316–327, Trento, Italy, 1998.

[27] Gerhard Lakemeyer and Hector J. Levesque. Query evaluation and progression in $\mathcal{AOL}$ knowledge bases. In T. Dean, editor, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 124–131, Stockholm, Sweden, 1999.

[28] Hector Levesque, Fiora Pirri, and Ray Reiter. Foundations for a calculus of situations. *Electronic Transactions on Artificial Intelligence*, 3((1–2)):159–178, 1998. URL: `http://www.ep.liu.se/ea/cis/1998/018/`.

[29] Hector Levesque and Ray Reiter. High-level robotic control: beyond planning. In *Integrating Robotics Research: Taking the Next Big Leap*, AAAI Spring Symposia, Stanford University, March 1998.

[30] 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.

[31] Vladimir Lifschitz. Circumscription. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in AI and Logic Programming*, volume 3, pages 298–352. Oxford University Press, 1994.

[32] Fangzhen Lin. Embracing causality in specifying the indirect effects of actions. In C. S. Mellish, editor, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1985–1991, Montreal, Canada, August 1995. Morgan Kaufmann.

[33] Fangzhen Lin. Embracing causality in specifying the indeterminate effects of actions. In B. Clancey and D. Weld, editors, *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 670–676, Portland, OR, August 1996. MIT Press.

[34] Fangzhen Lin and Hector Levesque. What robots can do: Robot programs and effective achievability, 1997. (Manuscript).

[35] John W. Lloyd. *Foundations of Logic Programming.* Series Symbolic Computation. Springer, second, extended edition, 1987.

[36] Jorge Lobo, Gisela Mendez, and Stuart R. Taylor. Adding knowledge to the action description language $\mathcal{A}$. In B. Kuipers and B. Webber, editors, *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 454–459, Providence, RI, July 1997. MIT Press.

[37] Marcel Masseron, Christophe Tollu, and Jacqueline Vauzielles. Generating plans in linear logic I. Actions as proofs. *Journal of Theoretical Computer Science*, 113:349–370, 1993.

[38] Norman McCain and Hudson Turner. A causal theory of ramifications and qalifications. In C. S. Mellish, editor, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1978–1984, Montreal, Canada, August 1995. Morgan Kaufmann.

[39] John McCarthy. Programs with Common Sense. In *Proceedings of the Symposium on the Mechanization of Thought Processes*, volume 1, pages 77–84, London, November 1958. (Reprinted in: [42]).

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

[41] John McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[42] John McCarthy. *Formalizing Common Sense.* Ablex, 1990. (Edited by V. Lifschitz).

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

[44] Sheila McIlraith. An axiomatic solution to the ramification problem (sometimes). *Artificial Intelligence*, 116(1–2):87–121, 2000.

[45] Robert Moore. A formal theory of knowledge and action. In J. R. Hobbs and R. C. Moore, editors, *Formal Theories of the Commonsense World*, pages 319–358. Ablex, 1985.

[46] Judea Pearl. Causation, action, and counterfactuals. Technical Report R-223-T, Computer Science Department, University of California at Los Angeles, February 1996. (Published in *Proceedings of the Conference on Theoretical Aspects of Rationality and Knowledge (TARK)*, pages 51–73, March 1996.).

[47] Pavlos Peppas, Maurice Pagnucco, Mikhail Prokopenko, Norman Y. Foo, and Abhaya Nayak. Preferential semantics for causal systems. In T. Dean, editor, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 118–123. Morgan Kaufmann, Stockholm, Sweden, 1999.

[48] Mikhail Prokopenko, Maurice Pagnucco, Pavlos Peppas, and Abhaya Nayak. Causal propagation semantics—a study. In N. Foo, editor, *Proceedings of the Australian Joint Conference on Artificial Intelligence*, volume 1747 of *LNAI*, pages 378–3392, Sydney, Australia, December 1999. Springer.

[49] Mikhail Prokopenko, Maurice Pagnucco, Pavlos Peppas, and Abhaya Nayak. A unifying semantics for causal propagation. In *Proceedings of the Australian Joint Conference on Artificial Intelligence*, LNAI, Melbourne, Australia, August 2000. Springer.

[50] Willard Quine. *From a Logical Point of View: 9 Logico-Philosophical Essays.* Harvard University Press, 1953.

[51] Ray Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, New York, 1978.

[52] Ray Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*, pages 359–380. Academic Press, 1991.

[53] Ray Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64:337–351, 1993.

[54] Ray Reiter. On knowledge-based programming with sensing in the situation calculus. In *Cognitive Robotics Workshop at ECAI*, pages 55–61, Berlin, Germany, August 2000.

[55] Erik Sandewall. Assessments of ramification methods that use static domain constraints. In L. C. Aiello, J. Doyle, and S. Shapiro, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 99–110, Cambridge, MA, November 1996. Morgan Kaufmann.

[56] Erik Sandewall, editor. *Electronic Transactions on Artificial Intelligence 2(3–4). A Collection of Refereed Reference Articles*. 1998. URL: `http://www.ep.liu.se/ej/etai/1998`.

[57] Richard Scherl and Hector Levesque. The frame problem and knowledge-producing actions. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 689–695, Washington, DC, July 1993.

[58] Murray Shanahan. A circumscriptive calculus of events. *Artificial Intelligence*, 77:249–284, 1995.

[59] Murray Shanahan. Event calculus planning revisited. In *Proceedings of the European Conference on Planning (ECP)*, volume 1348 of *LNAI*, pages 390–402. Springer, 1997.

[60] Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press, 1997.

[61] Murray Shanahan. The ramification problem in the event calculus. In T. Dean, editor, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 140–146, Stockholm, Sweden, 1999. Morgan Kaufmann.

[62] John C. Shepherdson. SLDNF-resolution with equality. *Journal of Automated Reasoning*, 8:297–306, 1992.

[63] Michael Thielscher. Ramification and causality. *Artificial Intelligence*, 89(1–2):317–364, 1997.

[64] Michael Thielscher. Fluent Calculus planning with continuous change. *Electronic Transactions on Artificial Intelligence*, 1999. (Submitted.)
URL: `http://www.ep.liu.se/ea/cis/1999/011/`.

[65] 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.

[66] Michael Thielscher. The concurrent, continuous Fluent Calculus. *Studia Logica*, 2000.

[67] Michael Thielscher. Nondeterministic actions in the fluent calculus: Disjunctive state update axioms. In S. Hölldobler, editor, *Intellectics and Computational Logic*, pages 327–345. Kluwer Academic, 2000.

[68] Hudson Turner. A logic of universal causation. *Artificial Intelligence*, 113(1–2):87–123, 1999.