

Technische Universität Dresden

Fakultät Informatik

Diplomarbeit

Translation of Domain Specifications between  
Situation Calculus and Fluent Calculus

Stephan Schiffel

born on September 15th 1980

Supervisor: Prof. Michael Thielscher

submitted the 9th of March 2005



## **Abstract**

Situation Calculus and Fluent Calculus are two action formalisms used for describing dynamic domains. Both calculi provide different solutions to the frame problem and therefore the programming languages based on them—Golog and Flux—have different properties. For this reason, one might be suited better than the other to solve certain classes of problems. In order to compare the two calculi or the two programming languages a formal relation between both is needed. In this thesis I develop such a relation in form of a translation function between domain axiomatisations of Situation Calculus and Fluent Calculus and prove the correctness of this translation. Furthermore, I present an implementation of this translation function in Prolog which translates Golog domain descriptions to Flux and vice versa.



# Contents

<b>Contents</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Preliminaries</b>	<b>3</b>
2.1 The Situation Calculus . . . . .	3
2.1.1 Simplifying Assumptions . . . . .	4
2.1.2 Formal Definitions . . . . .	4
2.1.3 Domain Specifications . . . . .	7
2.1.4 An Example Domain . . . . .	9
2.2 The Fluent Calculus . . . . .	12
2.2.1 Formal Definitions . . . . .	12
2.2.2 Domain Specifications . . . . .	15
2.2.3 An Example Domain . . . . .	17
<b>3 Translation of Domain Specifications</b>	<b>19</b>
3.1 Formulas . . . . .	19
3.2 Knowledge Expressions . . . . .	20
3.3 Initial State Descriptions . . . . .	21
3.4 Domain Constraints . . . . .	23
3.5 Action Preconditions . . . . .	24
3.6 Effects of Actions . . . . .	26
3.6.1 Physical Effects . . . . .	26
3.6.2 Cognitive Effects . . . . .	31
3.7 Auxiliary Axioms . . . . .	33
<b>4 Proof of Conservation of Equivalence</b>	<b>35</b>

<b>5</b>	<b>Implementation in Prolog</b>	<b>43</b>
5.1	Foundations . . . . .	43
5.2	Another Example Domain . . . . .	44
5.3	Fluents . . . . .	45
5.4	Initial State . . . . .	48
5.5	Precondition Axioms . . . . .	50
5.6	Formulas . . . . .	50
5.7	Effect Axioms . . . . .	52
5.8	Auxiliary Axioms . . . . .	55
<b>6</b>	<b>Discussion</b>	<b>57</b>
<b>A</b>	<b>Golog to Flux Translator</b>	<b>61</b>
<b>B</b>	<b>Flux to Golog Translator</b>	<b>69</b>
<b>C</b>	<b>Example Golog Domain</b>	<b>77</b>
<b>D</b>	<b>Example Flux Domain</b>	<b>81</b>
	<b>Bibliography</b>	<b>85</b>
	<b>Selbständigkeitserklärung</b>	<b>87</b>

# Chapter 1

## Introduction

Situation Calculus and Fluent Calculus are action formalisms. These formalisms provide theories for intelligent systems to reason about actions and how they change the state of the world. They enable autonomous systems to decide on actions and execute them. Those decisions are based on the state of the world and the effects that actions have on this state. There are several other action formalisms, e.g. the Event Calculus [Sha99] and Causal Theories [GLL<sup>+</sup>04].

Situation Calculus and Fluent Calculus are special among these formalisms because there are two programming languages for programming high-level robot control programs based on them—Golog and Flux. Both calculi provide a different solution to the well known frame problem [MH69]. In Situation Calculus effects of action are defined by successor state axioms, describing the value of a fluent (an atomic property of the world) in a situation after executing an action in terms of the fluent's value in the situation before the action. Inferences of properties of the world in some situation are then made by regressing the properties to the initial situation with the help of the successor state axioms and evaluating the regressed properties in the initial situation. In Fluent Calculus effects of actions are described by action-based state update axioms. Those axioms update an explicit description of the state of the world upon execution of an action. Inferences of properties of the world can then be made by looking up the properties in the explicit state representation.

Because of these different approaches both calculi and the programming languages based on them have different properties. Therefore one might be suited better than the other to solve certain classes of problems [Thi05].

However there exists no proper analysis about that issue until now. For comparing the different properties of both calculi a formal relation between both calculi is necessary. Such a relation is given in this thesis in terms of a translation function between domain descriptions in both calculi. An important property of the developed relation is that inferences of facts that can be made under a domain description of one calculus are valid exactly if they are also valid under the domain description translated to the other calculus.

For analysing the suitability of Golog and Flux for solving certain problems a translation of domain descriptions of both programming languages to each other is necessary, in order to allow to apply both languages to the same problem. Ideally, this translation is just an implementation of the translation function between the two calculi. But in reality it differs from that as much as Golog and Flux are not just implementations of the calculi they are based on. Both programming languages pose certain restrictions on the domain axiomatization in order to make an automatic inference of facts possible and feasible. Thus Golog and Flux only allow to describe a smaller set of domains than their respective calculi.

Another benefit of a formal relation between Situation Calculus and Fluent Calculus is the possibility to compare extensions made separately for both calculi, such as concurrency. Furthermore it might even allow to translate current or future extensions made only for one of the calculi to the other.

The goal of this thesis is to provide a translation of Situation Calculus domain axiomatizations to Fluent Calculus and vice versa and to prove its correctness. That means that the translation preserves the validity of inferences. Furthermore a proof of concept implementation of this translation function will be provided, but only for a restricted class of domains.

The thesis is structured as follows:

In chapter 2 I will introduce Situation Calculus and Fluent Calculus and the domain axiomatizations in both calculi as a necessary prerequisite for the following chapters. In chapter 3 the translation function is developed and some restrictions to the domain descriptions are made in order to make the translation possible. The conservation of equivalence, i.e. the preservation of the validity of inferences, of the translation function is proved in chapter 4. In chapter 5 I present a proof of concept implementation of the translation function. I conclude with a discussion and presentation of open issues in chapter 6.

## Chapter 2

# Preliminaries

In this chapter I will introduce the Situation Calculus and the Fluent Calculus and show how domain axiomatizations look in both calculi. These are the necessary foundations of the following chapters.

### 2.1 The Situation Calculus

The Situation Calculus as described in [Rei01b] is a second order language designed for axiomatizing dynamic domains. All changes to the environment are caused by *actions*. A sequence of actions is called a *situation*. The *initial situation*  $S_0$  of the world is the empty sequence of actions. The situation resulting from executing action  $a$  in situation  $s$  is denoted by  $Do(a, s)$ . Formally actions are distinct function symbols and may have arguments, e.g.  $goto(x)$  could be an action where  $x$  denotes the place to go to.

The execution of an action  $a$  in situation  $s$  is only possible if its preconditions are fulfilled in situation  $s$ . This is denoted by a predicate  $Poss(a, s)$ .  $Poss(a, s)$  is defined by a set of so called precondition axioms. The precondition axiom for some action  $A(\vec{x})$  is of the form  $Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$ . The formula  $\Pi_A$  places certain conditions on the properties of the world in situation  $s$ , which have to be fulfilled in order that the action  $A(\vec{x})$  is possible.

Only actions may change certain properties of the world. Those “non-static” properties are denoted by so called *fluents*. Fluents are relations or functions with a situation term as last argument, so they may have different values in different situations. The change of these values is the effect of actions, which are axiomatized by negative and positive effect axioms, describing the negative (fluents becoming false) and positive effects (fluents becoming true) accordingly. Positive and negative effect axioms are combined to *successor state axioms*, which define the value of a fluent after some action in terms of the value of the fluent before the action. Successor state axioms provide a solution to the well known frame problem ([Rei01b, chapter 3.1.4]).

With the description given above each relational fluent is in every situation either true or false and each functional fluent has a distinct value in every

situation. Thus it is not possible to describe uncertainty about a fluent's value in a situation, e.g. it is not possible to describe the fact that one does not know whether a fluent is true or false. For achieving this Fluent Calculus is enhanced by the notion of knowledge, and therefore also lack of knowledge. Therefore, a set of possible situations instead of one distinct situation is considered. Thus if we do not know a fluents value then there is a situation in the set for each possible value of the fluent. As we consider all of these situations possible we do not know the value of the fluent. But we do know of course the values of all the fluents that have the same value in every possible situation, because there is no alternative value for these fluents in that case.

To describe such a set of possible situations [Rei01b, chapter 11] introduces a new binary predicate  $K(s', s)$  with the intuitive meaning that if  $s$  is a possible situation than  $s'$  is also considered possible. With the help of this relation it is possible to describe knowledge by a suggestive notation  $Knows(\phi, \sigma) \stackrel{\text{def}}{=} (\forall s')K(s', \sigma) \supset \phi[s']$ . Intuitively that means that we know a formula  $\phi$  to be true in situation  $\sigma$  exactly if  $\phi$  is true in every situation connected to  $\sigma$  by the  $K$ -relation, i.e. in every situation possible besides  $\sigma$ .

To gain new knowledge there are two additional types of actions besides the "normal" ones: actions sensing whether a condition  $\phi(\vec{x}, s)$  holds or not in a situation,  $sense_\phi(\vec{x})$ , and actions sensing the value of a functional fluent  $f(\vec{x}, s)$ ,  $read_f(\vec{x})$ . Those sensing actions do not have any effect on the value of any fluent, they just reduce the set of possible situations so their effect is only knowledge.

### 2.1.1 Simplifying Assumptions

For the sake of simplicity I adopt two assumptions from [Rei01a]. I will abandon functional fluents, i.e. functions that take situation arguments. One may represent functional fluents by relational fluents with the functional fluent's value as additional argument and enforce that there is a unique value in each situation by the initial situation and successor state axioms.

Furthermore I assume that there are no non-fluent predicates other than equality ( $=$ ),  $\sqsubset$ , and  $Poss$ . One may represent such a predicate  $R(\vec{x})$  as a fluent  $R(\vec{x}, s)$  with a successor state axiom stating that  $R(\vec{x}, s)$  is not changed by any action. All assertions about  $R(\vec{x})$  have to be made in terms of  $R(\vec{x}, S_0)$  in the initial state description.

### 2.1.2 Formal Definitions

In this section I will give a formal definition of the Situation Calculus and Situation Calculus domain descriptions.

The language  $L_{sitcalc}$  with knowledge is a second order language with equality and has three disjoint sorts: *action*, *situation* and *object* for everything apart from actions and situations. The alphabet consists of the standard logical symbols and the following items [Rei01b, chapter 4, chapter 11]:

- Countably infinitely many variable symbols of each sort. By convention  $a$  and  $s$  are used for actions and situations, respectively.
- Two function symbols of sort situation:
  - $S_0$  : *situation* stands for the initial situation.
  - $Do$  :  $action \times situation \rightarrow situation$  has the intuitive meaning that  $Do(a, s)$  denotes the situation obtained from executing action  $a$  in situation  $s$ .
- A binary predicate symbol  $\sqsubset$  :  $situation \times situation$  defining an ordering relation on situations. The intended interpretation is that  $s \sqsubset s'$  denotes that  $s$  is a proper subsequence of  $s'$ .
- A binary predicate symbol  $K$  :  $situation \times situation$ , defining a accessibility relation on situations. i.e. a relation that defines which situations are considered possible.
- A binary predicate symbol  $Poss$  :  $action \times situation$ , defining if an action is possible in a situation.
- for each  $n \geq 0$  countably infinitely many function symbols of sort  $(action \cup object)^n \rightarrow object$  for situation independent functions
- for each  $n \geq 0$  countably infinitely many function symbols of sort  $(action \cup object)^n \rightarrow action$  for actions
- for each  $n \geq 0$  countably infinitely many predicate symbols of sort  $(action \cup object)^n \times situation$  for relational fluents

With the help of the accessibility relation  $K$  one can define knowledge of a fact  $\phi$  in a situation  $s$  as follows:

$$Knows(\phi, s) \stackrel{\text{def}}{=} (\forall s'). K(s, s') \supset \phi[s']$$

Here  $\phi$  is a situation-suppressed expression (see definition 2.4) and  $\phi[s']$  is a Situation Calculus formula about  $s'$  obtained by  $\phi$  by restoring the suppressed situation argument with  $s'$ .

The foundational axioms of Situation Calculus with knowledge,  $\Sigma_{sit}$ , describe the basic properties of situations in any domain axiomatization. They consists of the following axioms ([Rei01b, chapter 11.5]):

- $Do(a_1, s_1) = Do(a_2, s_2) \supset a_1 = a_2 \wedge s_1 = s_2$  (unique name axiom for situations)
- $(\forall P). (\forall s)[Init(s) \supset P(s)] \wedge (\forall a, s)[P(s) \supset P(Do(a, s))] \supset (\forall s)P(s)$  (second order induction axiom saying that situations are finite sequences of actions)
- $\neg(s \sqsubset S_0)$

- $s \sqsubset Do(a, s') \equiv s \sqsubseteq s'$
- $K(s, s') \supset [Init(s) \equiv Init(s')]$  (only initial situations can be  $K$ -related to an initial situation)

Here  $Init(s)$  is an abbreviation defined as follows:

$$Init(s) \stackrel{\text{def}}{=} \neg(\exists a, s') s = Do(a, s')$$

The predicate  $Init(s)$  means that  $s$  is an initial situation, i.e. there is no situation preceding  $s$ .

Intuitively, those axioms say that situations are finite sequences of actions and that situations are ordered, i.e. there is a precedence between certain situations. Namely any situation  $s'$  succeeds a situation  $s$  ( $s \sqsubset s'$ ) if it can be obtained from  $s$  by adding one or more actions to  $s$ . In consequence all situations except for initial situations have a well defined predecessor, which is necessary for regression over situations, which in turn is used for inferring the values of fluents.

Now I define some kinds of formulas, which are used for different purpose.

Formulas uniform in a situation term  $\sigma$  are formulas stating facts about the situation  $\sigma$  but that do not say anything about any situation other than  $\sigma$ . They are defined as follows:

**Definition 2.1 (formulas uniform in  $\sigma$ ).** *Let  $\sigma$  be a term of sort situation and  $\vec{t}$  be terms not mentioning any terms of sort situation.*

- *Each relational fluent  $F(\vec{t}, \sigma)$  is a formula uniform in  $\sigma$ .*
- *If  $t_1$  and  $t_2$  are terms not of sort situation, then  $t_1 = t_2$  is a formula uniform in  $\sigma$ .*
- *All formulas formed from formulas uniform in  $\sigma$  and standard logical connectives that do not quantify over situations are formulas uniform in  $\sigma$ .*

Note that due to the assumption made in section 2.1.1 this definition differs slightly from the original one of [Rei01b], in that there are no special “terms uniform in a situation” because there are no fluent functions.

Formulas about a situation term  $\sigma$  are formulas stating facts about the situation  $\sigma$  and the agents knowledge in this situation. They also do not say anything about any situation other than  $\sigma$  (except those,  $K$ -related to  $\sigma$ ). In fact they are just formulas uniform in  $\sigma$  extended by knowledge statements:

**Definition 2.2 (formulas about  $\sigma$ ).** *Let  $\sigma$  be a term of sort situation.*

- *Each relational fluent  $F(\vec{t}, \sigma)$  is a formula about  $\sigma$ .*
- *If  $t_1$  and  $t_2$  are terms not of sort situation, then  $t_1 = t_2$  is a formula about  $\sigma$ .*

- If  $\phi$  is an objective situation-suppressed expression (see definition 2.4), then  $\text{Knows}(\phi, \sigma)$  is a formula about  $\sigma$ .
- All formulas formed from formulas about  $\sigma$  and standard logical connectives that do not quantify over situations are formulas about  $\sigma$ .

Expressions used in knowledge statements are defined here:

**Definition 2.3 (Admissible situation-suppressed expressions).**

- For each relational fluent  $F(\vec{t}, \sigma)$ ,  $F(\vec{t})$  is an admissible situation-suppressed expression.
- If  $t_1$  and  $t_2$  are terms not of sort situation, then  $t_1 = t_2$  is an admissible situation-suppressed expression.
- If  $\phi$  is an admissible situation-suppressed expression, then  $\text{Knows}(\phi)$  is, too.
- All expressions formed from admissible situation-suppressed expressions and standard logical connectives that do not quantify over situations are admissible situation-suppressed expressions.

**Definition 2.4 (Objective situation-suppressed expressions).**

- For each relational fluent  $F(\vec{t}, \sigma)$ ,  $F(\vec{t})$  is an objective situation-suppressed expression.
- If  $t_1$  and  $t_2$  are terms not of sort situation, then  $t_1 = t_2$  is an objective situation-suppressed expression.
- All expressions formed from objective situation-suppressed expressions and standard logical connectives that do not quantify over situations are objective situation-suppressed expressions.

The only difference between admissible and objective situation-suppressed expressions is, that admissible situation-suppressed expressions may express facts about the agent's mental state, i.e. they may state that the agent knows something about its knowledge.

### 2.1.3 Domain Specifications

Domain specifications in the Situation Calculus are called *basic action theories*. A basic action theory is a collection of axioms  $D$  with the *functional fluent consistency property*, where  $D$  consists of:

- $\Sigma_{sit}$ , the foundational axioms for Situation Calculus with knowledge

- $D_{ss}$ , a set of successor state axioms including an axiom for  $K$
- $D_{ap}$ , a set of action precondition axioms
- $D_{una}$ , a set of unique names axioms for actions
- $D_{S_0}$ , a set of sentences about  $S_0$ , the *initial database*
- $K_{Init}$ , a set of initial accessibility axioms specifying the  $K$  relation in the initial situation

**Definition 2.5 (Successor state axiom).** *A successor state axiom for an  $(n+1)$ -ary relational fluent  $F$  is a Situation Calculus formula of the form:*

$$F(\vec{x}, Do(a, s)) \equiv \gamma_F^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s)$$

where the domain axiomatisation entails:

$$\neg(\exists\vec{x}, a, s)\gamma_F^+(\vec{x}, a, s) \wedge \gamma_F^-(\vec{x}, a, s)$$

Here  $\gamma_F^+(\vec{x}, a, s)$  and  $\gamma_F^-(\vec{x}, a, s)$  are formulas uniform in  $s$  with free variables among  $\vec{x}, a$ . If  $\gamma_F^+(\vec{x}, a, s)$  holds then the fluent  $F(\vec{x})$  is a positive effect of action  $a$  in situation  $s$ . The Fluent is a negative effect, i.e. it does no longer hold in the resulting situation  $Do(a, s)$ , if  $\gamma_F^-(\vec{x}, a, s)$  holds.

Note that I restrict the syntactic form of successor state axioms more than is done in most publications about Situation Calculus. The reason for this is that it will make transformations between successor state and state update axioms (see definition 2.15) much easier. Furthermore, the restriction will be met in most practical domains anyway.

There is a special successor state axiom, the one for the  $K$  relation:

$$\begin{aligned} K(s', Do(a, s)) &\equiv (\exists s^*).s' = Do(a, s^*) \wedge K(s^*, s) \wedge \\ &(\forall \vec{x}_1)[a = \text{sense}_{\Psi_1}(\vec{x}_1) \supset \Psi_1(\vec{x}_1, s^*) \equiv \Psi_1(\vec{x}_1, s)] \\ &\wedge \dots \wedge \\ &(\forall \vec{x}_n)[a = \text{sense}_{\Psi_n}(\vec{x}_n) \supset \Psi_n(\vec{x}_n, s^*) \equiv \Psi_n(\vec{x}_n, s)] \end{aligned}$$

Every sense action  $\text{sense}_{\Psi_i}(\vec{x}_i)$  is associated with a condition  $\Psi_i(\vec{x}_i)$ , that is an objective situation-suppressed expression. The successor state axiom says that after executing the sensing action  $\text{sense}_{\Psi_i}(\vec{x}_i)$  all situations that are  $K$ -related to another have to agree in the property  $\Psi_i(\vec{x}_i)$ . Thus, it reduces the set of possible situations and it results in knowing  $\Psi_i(\vec{x}_i)$  after  $\text{sense}_{\Psi_i}(\vec{x}_i)$ .

**Definition 2.6 (Action precondition axiom).** *An action precondition axiom is a Situation Calculus formula of the form:*

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

where  $A$  is an  $n$ -ary action function symbol and  $\Pi_A$  is a Situation Calculus formula about  $s$  with free variables among  $\vec{x}$  and  $s$ .

**Definition 2.7** ( $K_{Init}$ ).  $K_{Init}$  is any set of initial accessibility axioms specifying  $K$  in the initial situation. They have to fulfill certain properties described in [Rei01a]. In general  $K_{Init}$  may state any of the following properties for the  $K$  relation in the initial situation: reflexive, symmetric, transitive, euclidean. The successor state axiom for  $K$  guarantees that those properties hold in all situations, although only stated for initial situations.

For the translation developed in the following chapter to work reflexivity of the  $K$  relation is required. Thus  $K_{Init}$  must contain the following axiom:

$$(\forall s)(Init(s) \supset K(s, s))$$

One result of reflexivity of the  $K$  relation is the following property:

**Proposition 2.1 (truth of knowledge).**

$$(\forall s)Knows(\phi, s) \supset \phi[s]$$

That means that every fact  $\phi$  that is known in some situation  $s$  has to be true in  $s$ .

#### 2.1.4 An Example Domain

In this section I will present an example Situation Calculus domain axiomatisation for making the above definitions clearer.

Consider a cleaning robot in an office building whose task it is to empty all the waste bins in the offices of the floor depicted in figure 2.1. Waste bins are assumed to be in every office and every square of the corridor. The Room at position (1,1) is the home of the cleanbot. The cleaning robot shall only enter offices that are not currently occupied by anyone because it is quite noisy. Therefore, the robot is equipped with a light sensor, that enables him to sense whether there is light in or adjacent to the robots current location. If there is light then at least one of the adjacent rooms (or the one the robot is in) has to be occupied. At the beginning the robot doesn't know which of the offices are in use.

The state of the world can be described by the following four fluents:

$At(x, y, s) : \mathbb{N} \times \mathbb{N} \times situation$	the robot's position is $(x, y)$
$Facing(d, s) : \mathbb{N} \times situation$	the robot faces direction $d$
$Occupied(x, y, s) : \mathbb{N} \times \mathbb{N} \times situation$	room $(x, y)$ is occupied
$Cleaned(x, y, s) : \mathbb{N} \times \mathbb{N} \times situation$	dust bin in $(x, y)$ has been cleaned

The direction  $d$  of  $Facing(d, s)$  is encoded by numbers 1 to 4 standing for the directions north, south east and west.

The robot can execute the following three actions:

$Go : action$	go forward to adjacent square
$Turn : action$	turn clockwise by $90^\circ$
$Clean : action$	empty waste bin at current location

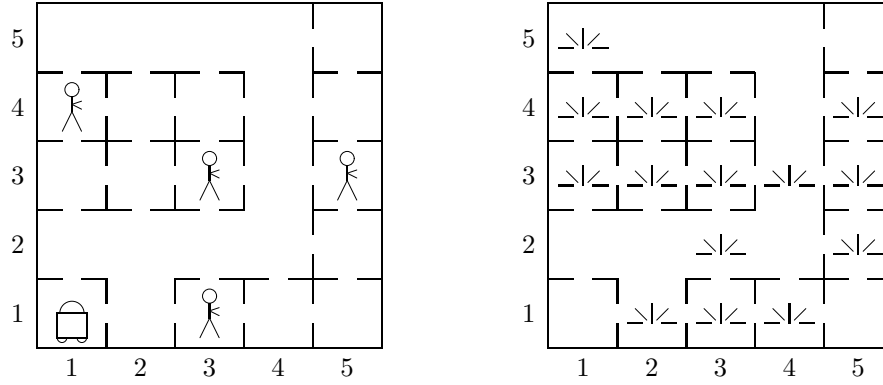


Figure 2.1: Layout of an office floor (from [Thi04]). In the scenario depicted in the left diagram, four offices are still in use. The diagram on the right hand side shows the locations in which a light sensor would be activated.

The *Go* action should have the additional effect of sensing whether there is light at the new position. Because this conflicts with the condition put on sensing actions in Situation Calculus not to have physical effects, I add a pure sensing action  $Sense_{Light}$  which should be executed directly after each *Go* action.

$Sense_{Light}$  : *action* sense whether there is light at the current position

The property *Light* is defined as follows:

$$\begin{aligned}
 Light(s) &\stackrel{\text{def}}{=} (\exists x, y) At(x, y, s) \wedge \\
 & (Occupied(x, y, s) \vee Occupied(x + 1, y, s) \vee Occupied(x - 1, y, s) \vee \\
 & Occupied(x, y + 1, s) \vee Occupied(x, y - 1, s))
 \end{aligned}$$

This property says that the the robot perceiving light means that the room at the robots current location or any of the rooms adjacent to it is occupied.

Those actions defined above have the following precondition axioms:

$$\begin{aligned}
 Poss(Go, s) &\equiv (\forall d, x, y) At(x, y, s) \wedge Facing(d, s) \supset \\
 & (\exists x', y') Adjacent(x, y, d, x', y') \\
 Poss(Turn, s) &\equiv \top \\
 Poss(Clean, s) &\equiv \top \\
 Poss(Sense_{Light}, s) &\equiv \top
 \end{aligned}$$

Thus, the robot may go forward only if he does not face a wall and he may turn, empty a waste bin and sense light at any time.

Here  $Adjacent(x, y, d, x', y', S_0)$  is an auxiliary axiom with the property that

$x', y'$  is the location the robot is facing if he is at  $x, y$  and faces direction  $d$ :

$$\begin{aligned} \text{Adjacent}(x, y, d, x', y', S_0) \equiv & 1 \leq d \leq 4 \wedge 1 \leq x, x', y, y' \leq 5 \wedge \\ & [d = 1 \wedge x' = x \wedge y' = y + 1 \vee \\ & d = 2 \wedge x' = x + 1 \wedge y' = y \vee \\ & d = 3 \wedge x' = x \wedge y' = y - 1 \vee \\ & d = 4 \wedge x' = x - 1 \wedge y' = y] \end{aligned}$$

According to the simplifying assumptions of section 2.1.1 there are no non-fluent predicates. Therefore, this auxiliary axiom has to be rewritten as a fluent, i.e. a situation added as last argument, with the trivial successor state axiom:

$$\text{Adjacent}(x, y, d, x', y', Do(a, s)) \equiv \text{Adjacent}(x, y, d, x', y', s)$$

The successor state axioms for the other fluents are the following:

$$\begin{aligned} \text{At}(x, y, Do(a, s)) \equiv & \\ a = Go \wedge \text{At}(x', y', s) \wedge \text{Facing}(d', s) \wedge \text{Adjacent}(x', y', d', x, y, s) \vee & \\ \text{At}(x, y, s) \wedge \neg a = Go & \end{aligned}$$

$$\begin{aligned} \text{Facing}(d, Do(a, s)) \equiv a = Turn \wedge \text{Facing}(d', s) \wedge d' \bmod 4 + 1 = d \vee & \\ \text{Facing}(d, s) \wedge \neg a = Turn & \end{aligned}$$

$$\text{Occupied}(x, y, Do(a, s)) \equiv \perp \vee \text{Occupied}(x, y, s) \wedge \top$$

$$\begin{aligned} \text{Cleaned}(x, y, Do(a, s)) \equiv a = Clean \wedge \text{At}(x, y, s) \vee & \\ \text{Cleaned}(x, y, s) \wedge \top & \end{aligned}$$

That is to say, the robots position only changes by means of the *Go* action, the direction the robot faces is only changed by the *Turn* action, whether a room is occupied or not does not change at all (at least during the time the robot is active) and a dust bin is emptied on a *Clean* action in the same square and stays so (during the time the robot is active).

Note, that there is no successor state axiom of a fluent mentioning the action *SenseLight* because it does not have any physical effect. The cognitive effect of *SenseLight* is achieved by the successor state axiom for the *K* relation. Which states that only those situations  $Do(\text{SenseLight}, s^*)$  and  $Do(\text{SenseLight}, s')$  are *K*-related to each other that share the property *Light*, i.e. where  $\text{Light}(s^*)$  and  $\text{Light}(s')$  are equivalent:

$$\begin{aligned} K(s', Do(a, s)) \equiv (\exists s^*).s' = Do(a, s^*) \wedge K(s^*, s) \wedge & \\ a = \text{SenseLight} \supset \text{Light}(s^*) \equiv \text{Light}(s) & \end{aligned}$$

## 2.2 The Fluent Calculus

The Fluent Calculus is an extension of the Situation Calculus. The basic concepts of situations, fluents for describing the world's state and actions for changing it are similar. But there is an explicit notion of a state of the world. Fluents are no longer considered as relations or functions in Fluent Calculus but instead they are terms of sort *fluent* which are connected by a binary function symbol  $\circ$  to form a term of sort *state*. A fluent is defined to hold, i.e. to be true, in a state if it is part of the state. Each situation  $s$  has a state connected to it via a function symbol  $State(s)$ .

The Fluent Calculus provides another solution to the frame problem different from that of the Situation Calculus. Instead of successor state axioms the effect axioms of the actions are combined to *state update axioms*, which describe, for each action  $a$ , the state  $State(Do(a, s))$  in terms of the state  $State(s)$  and the action's positive and negative effects.

In Fluent Calculus uncertainty and knowledge are expressed similar as in Situation Calculus. Instead of a set of possible situations there is just one situation  $s$  but a set of possible states  $z$  in that situation, defined by a predicate  $KState(s, z)$ .

Instead of having special sense actions normal actions can have sensing results.

### 2.2.1 Formal Definitions

The Fluent Calculus is a first order language with equality and has four sorts: *action*, *situation*, *fluent* and *state*, where *fluent* is a sub-sort of sort *state*. The signature of Fluent Calculus with knowledge consists of the following items [Thi04]:

- Countably infinitely many variable symbols of each sort. By convention  $a$ ,  $s$ ,  $f$  and  $z$  are used for actions, situations, fluents and states, respectively.
- A finite number of function symbols into sort fluent:  $\mathcal{F}$
- A finite number of function symbols into sort action:  $\mathcal{A}$
- Three function symbols of sort state:
  - $\emptyset$  : *state* stands for the empty state.
  - $\circ$  :  $state \times state \rightarrow state$ , where  $z_1 \circ z_2$  means the state obtained by merging the properties of both states. It can be compared to the union operation for sets.
  - $State : Exp$  assigns a state to each situation.
- Two function symbols of sort situation:  $S_0$  : *situation* and  $Do$  :  $action \times situation \rightarrow situation$ . As in Situation Calculus  $S_0$  denotes the initial situation and  $Do(a, s)$  denotes the situation obtained by executing the action  $a$  in situation  $s$ .

- A binary predicate symbol  $KState : situation \times state$  defining a relation between a situation and all states that are considered possible in that situation. The  $KState$ -relation can be compared to the  $K$ -relation of Situation Calculus.
- A binary predicate symbol  $Poss : action \times state$  defining the precondition of an action in a state similar to  $Poss$  in Situation Calculus.

A macro  $Holds(f, z)$  defines what it means for a fluent  $f$  to hold in a state  $z$ :

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

For handling states there are two operations defined as macros:

- $z + z_p$  for adding all fluents of state  $z_p$  to  $z$
- $z - z_n$  for removing all fluents of state  $z_n$  from  $z$

The foundational axioms for Fluent Calculus with knowledge describe basic properties of states:

**Definition 2.8 (foundational axioms).** *The foundational axioms for Fluent Calculus with knowledge are  $\Sigma_{state} \cup \Sigma_{knows}$ , where  $\Sigma_{state}$  consists of the following axioms:*

1. *Associativity, commutativity and idempotency of  $\circ$ :*

$$\begin{aligned} (\forall z_1, z_2, z_3) (z_1 \circ z_2) \circ z_3 &= z_1 \circ (z_2 \circ z_3) \\ (\forall z_1, z_2) z_1 \circ z_2 &= z_2 \circ z_1 \\ ((\forall z_1, z_2) z_1 \circ z_1 &= z_1) \end{aligned}$$

*Note that idempotency is a logical consequence of the other axioms and does not have to be stated as an axiom of its own.*

2. *Empty state axiom*

$$\neg Holds(f, \emptyset)$$

3. *Irreducibility*

$$Holds(f, g) \supset f = g$$

4. *Decomposition*

$$Holds(f, z_1 \circ z_2) \supset (Holds(f, z_1) \vee Holds(f, z_2))$$

5. *State Equality*

$$(\forall f) (Holds(f, z_1) \equiv Holds(f, z_2)) \supset z_1 = z_2$$

## 6. Existence of states

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

Here  $z_1, z_2, z_3$  are states,  $f, g$  are fluents and  $P$  is a second order predicate variable of sort fluent.

$\Sigma_{\text{knows}}$  contains the single axiom

## 7. Truth of knowledge

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

where  $s$  is a situation.

**Definition 2.9 (state formula).** A state formula  $\Delta(z)$  is a first order formula with free state variable  $z$  and without any occurrence of states other than in expressions of the form  $\text{Holds}(f, z)$  and without actions or situations.

**Definition 2.10 (situation formula).** If  $\Delta(z)$  is a state formula and  $s$  a situation variable, then  $\Delta(z)z/State(s)$ , i.e.  $\Delta$  with all occurrences of  $z$  replaced by  $State(s)$ , is a situation formula, written  $\Delta(s)$ .

**Definition 2.11 (knowledge expression).** A knowledge expression is composed of the standard logical connectives and atoms of the form

- $f$ , where  $f$  is a term of sort fluent
- $Poss(a)$ , where  $a$  is an action
- atoms  $P(\vec{t})$  without terms of sort state or situation

Knowledge is defined with the help of the  $KState$  relation as follows:

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

Here  $\phi$  is a knowledge expression and  $HOLDS(\phi, z)$  is obtained from  $\phi$  by replacing every fluent  $f$  by  $\text{Holds}(f, z)$  and every  $Poss(a)$  by  $Poss(a, z)$ .

Two important properties of this axiomatization of knowledge are that everything the agent knows is actually true and furthermore the agent is logically omniscient, i.e. he knows every consequence of its knowledge.

**Proposition 2.2 (truth of knowledge and logical omniscience).** Let  $\phi$  be a knowledge expression, then the foundational axioms  $\Sigma_{\text{state}} \cup \Sigma_{\text{knows}}$  entail

- $Knows(\phi, s) \supset HOLDS(\phi, State(s))$
- $Knows(\phi, s) \supset Knows(\psi, s)$ , if  $HOLDS(\phi, z) \models HOLDS(\psi, z)$

$$(\forall s)Knows(\phi, s) \supset \phi[s]$$

A knowledge state describes everything an agent knows in a certain situation.

**Definition 2.12 (knowledge state).** A knowledge state (for situation  $s$ ) is a formula

$$KState(s, z) \equiv \Phi(z)$$

where  $\Phi(z)$  is a state formula with free variable  $z$ .

### 2.2.2 Domain Specifications

A domain axiomatization  $\Sigma$  in Fluent Calculus for knowledge is a finite set of axioms consisting of

- $\Sigma_{dc}$  a set of domain constraints
- $\Sigma_{poss}$  a set of precondition axioms, one for each action
- $\Sigma_{sua}$  a set of state update axioms, one for each action
- $\Sigma_{kua}$  a set of knowledge update axioms, one for each action, such that all axioms agree with the corresponding state update axiom in  $\Sigma_{sua}$
- $\Sigma_{aux}$  a set of auxiliary axioms, independent of situations, actions and states, except for fluents
- $\Sigma_{init} = KState(S_0, z) \equiv \Phi(z)$  the initial knowledge state, where  $\Phi(z)$  is a state formula
- $\Sigma_{state} \cup \Sigma_{knows}$  the foundational axioms of Fluent Calculus for knowledge

**Definition 2.13 (domain constraint).** *A domain constraint is a formula of the form  $(\forall s)Knows(\phi, s)$ , where  $\phi$  is a knowledge expression without actions, i.e. a knowledge expression without the occurrence of *Poss*.*

Domain constraints restrict the space of possible states by stating general knowledge the agent has about the world.

**Definition 2.14 (precondition axiom).** *A precondition axiom for action  $A$  is a formula of the form  $Poss(A(\vec{x}), z) \equiv \Pi(\vec{x}, z)$ , where  $\Pi(\vec{x}, z)$  is a state formula with free variables among  $\vec{x}$  and  $z$ .*

**Definition 2.15 (state update axiom).** *A state update axiom for action  $A$  is a formula of the form*

$$\begin{aligned}
 Poss(A(\vec{x}), s) \supset & (\exists \vec{y}_1)(\Delta_1(s) \wedge State(Do(A(\vec{x}), s)) = State(s) - \vartheta_1^- + \vartheta_1^+) \\
 & \vee \dots \vee \\
 & (\exists \vec{y}_n)(\Delta_n(s) \wedge State(Do(A(\vec{x}), s)) = State(s) - \vartheta_n^- + \vartheta_n^+)
 \end{aligned}$$

where

- $n \geq 1$
- for each  $i = 1, \dots, n$ 
  - $\Delta_i(s)$  is a situation formula with free variables among  $\vec{x}, \vec{y}_i$  and  $s$
  - $\vartheta_i^-, \vartheta_i^+$  are finite states with free variables among  $\vec{x}$  and  $\vec{y}_i$

The terms  $\vartheta_i^-$  and  $\vartheta_i^+$  are called, respectively, negative and positive effects of  $A(\vec{x})$  under condition  $\Delta_i(s)$ .

State update axioms are used to calculate the new state after executing an action.

**Definition 2.16 (knowledge update axiom).** A knowledge update axiom for action  $A$  is a formula of the form

$$\text{Poss}(A(\vec{x}), s) \supset (\exists \vec{y})(\forall z')(K\text{State}(Do(A(\vec{x}), s), z') \equiv (\exists z)(K\text{State}(s, z) \wedge \Psi(z', z)) \wedge \Pi(z', Do(A(\vec{x}), s)))$$

where  $\Psi(z', z)$  specifies the physical effect and  $\Pi(z', Do(A(\vec{x}), s))$  the cognitive effect.

The physical effect  $\Psi(z', z)$  is a formula of the form

$$\begin{aligned} & (\exists \vec{y}_1)(\Delta_1(s) \wedge z' = z - \vartheta_1^- + \vartheta_1^+) \\ & \quad \vee \dots \vee \\ & (\exists \vec{y}_n)(\Delta_n(s) \wedge z' = z - \vartheta_n^- + \vartheta_n^+) \end{aligned}$$

If this formula corresponds with the state update axiom for action  $A$ , then knowledge update axiom and state update axiom agree.

The cognitive effect  $\Pi(z', Do(A(\vec{x}), s))$  is a formula of the form

$$\begin{aligned} & [\Pi_1(z') \equiv \Pi_1(Do(A(\vec{x}), s))] \wedge \dots \wedge [\Pi_k(z') \equiv \Pi_k(Do(A(\vec{x}), s))] \\ & \quad \wedge \\ & \text{Holds}(F_1(\vec{t}_1), z') \wedge \text{Holds}(F_1(\vec{t}_1), Do(A(\vec{x}), s)) \\ & \quad \wedge \dots \wedge \\ & \text{Holds}(F_l(\vec{t}_l), z') \wedge \text{Holds}(F_l(\vec{t}_l), Do(A(\vec{x}), s)) \\ & \quad \wedge \Pi(\vec{x}, \vec{y}) \end{aligned}$$

where

- $k, l \geq 0$
- for  $i = 1, \dots, k$ ,  $\Pi_i(z')$  is a state formula with free variables among  $\vec{x}$
- for  $i = 1, \dots, l$ ,  $F_i(\vec{t}_i)$  is a fluent with variables among  $\vec{x}$  and  $\vec{y}$
- $\Pi(\vec{x}, \vec{y})$  is an auxiliary formula without states and with variables among  $\vec{x}$  and  $\vec{y}$

Knowledge update axioms describe how the agent's knowledge state changes after executing an action. For non-sensing actions there is no cognitive effect and therefore knowledge update axioms and state update axioms contain essentially the same information. If the action is a sensing action, then there is a cognitive effect which may be knowledge about a property  $\Pi_i(Do(A(\vec{x}), s))$  of the situation after the action or knowledge about the value of a fluent  $F_i(\vec{t}_i)$ . The formula  $\Pi(\vec{x}, \vec{y})$  is used to put restrictions on the values of the fluents  $F_i(\vec{t}_i)$  depending on the arguments of the action  $A$ .

### 2.2.3 An Example Domain

The cleaning robot domain from section 2.1.4 can be defined in Fluent Calculus as follows.

The domain can be described by the following four fluents:

- $At(x, y) : \mathbb{N} \times \mathbb{N}$  the robot's position is  $(x, y)$
- $Facing(d) : \mathbb{N}$  the robot faces direction  $d$
- $Occupied(x, y) : \mathbb{N} \times \mathbb{N}$  room  $(x, y)$  is occupied
- $Cleaned(x, y) : \mathbb{N} \times \mathbb{N}$  dust bin in  $(x, y)$  has been cleaned

The robot can execute the following three actions:

- $Go : action$  go forward to adjacent square
- $Turn : action$  turn clockwise by  $90^\circ$
- $Clean : action$  empty waste bin at current location

Those actions have the following precondition axioms:

$$\begin{aligned}
 Poss(Go, z) &\equiv (\forall d, x, y) Holds(At(x, y), z) \wedge Holds(Facing(d), z) \supset \\
 &\quad (\exists x', y') Adjacent(x, y, d, x', y') \\
 Poss(Turn, z) &\equiv \top \\
 Poss(Clean, z) &\equiv \top
 \end{aligned}$$

The state update axioms are the following:

$$\begin{aligned}
 Poss(Go, s) &\supset (\exists d, x, y, x', y') (Holds(At(x, y), s) \wedge \\
 &\quad Holds(Facing(d), s) \wedge Adjacent(x, y, d, x', y') \wedge \\
 &\quad State(Do(Go, s)) = State(s) - At(x, y) + At(x', y')) \\
 Poss(Turn, s) &\supset (\exists d) (Holds(Facing(d), s) \wedge \\
 &\quad State(Do(Turn, s)) = \\
 &\quad State(s) - Facing(d) + Facing(d \bmod 4 + 1)) \\
 Poss(Clean, s) &\supset (\exists x, y) (Holds(At(x, y), s) \wedge \\
 &\quad State(Do(Clean, s)) = State(s) + Cleaned(x, y))
 \end{aligned}$$

The knowledge update axioms for *Turn* and *Clean* are essentially the same as the state update axioms:

$$\begin{aligned}
Poss(Turn, s) \supset KState(Do(Turn, s), z') &\equiv (\exists z)[KState(s, z) \wedge \\
&(\exists d)(Holds(Facing(d), z) \wedge \\
&z' = z - Facing(d) + Facing(d \bmod 4 + 1))] \\
Poss(Clean, s) \supset KState(Do(Clean, s), z') &\equiv (\exists z)[KState(s, z) \wedge \\
&(\exists x, y)(Holds(At(x, y), z) \wedge z' = z + Cleaned(x, y))]
\end{aligned}$$

Whereas, the knowledge update axiom for *Go* differs from its state update axiom by the cognitive effect of sensing whether there is light at the current position:

$$\begin{aligned}
Poss(Go, s) \supset KState(Do(Go, s), z') &\equiv (\exists z)[KState(s, z) \wedge \\
&(\exists d, x, y, x', y')(Holds(At(x, y), z) \wedge Holds(Facing(d), z) \wedge \\
&Adjacent(x, y, d, x', y') \wedge \\
&z' = z - At(x, y) + At(x', y'))] \wedge \\
&[\Pi_{Light}(z') = \Pi_{Light}(Do(Go, s))]
\end{aligned}$$

Here  $\Pi_{Light}$  says that the robot perceives light at its current position, which means that at least one of the adjacent offices is occupied:

$$\begin{aligned}
\Pi_{Light}(z) &\stackrel{\text{def}}{=} (\exists x, y)Holds(At(x, y), z) \wedge \\
&(Holds(Occupied(x, y), z) \vee \\
&Holds(Occupied(x + 1, y), z) \vee Holds(Occupied(x, y + 1), z) \vee \\
&Holds(Occupied(x - 1, y), z) \vee Holds(Occupied(x, y - 1), z))
\end{aligned}$$

Therefore, only those states  $z'$  are considered possible in situation  $Do(Go, s)$  which share the property of perceiving light with the true state  $State(Do(Go, s))$  in that situation.

## Chapter 3

# Translation of Domain Specifications

In this chapter I present translations between Situation Calculus and Fluent Calculus domain axiomatizations and prove the equivalence of translated domain axiomatizations to the original ones. I start with translating Situation Calculus formulas uniform in a situation into Fluent Calculus situation formulas and continue with translating the parts of the domain axiomatizations described in sections 2.1.3 and 2.2.2. In the following there will often appear formulas of both Fluent Calculus and Situation Calculus. As there are predicates with the same name, but different definition in both calculi I will use a superscript of  $s$  for Situation Calculus and  $f$  for Fluent Calculus to distinguish them.

### 3.1 Formulas

The translation of a Situation Calculus formula  $\Pi^s$  uniform in some situation term  $\sigma$  to a Fluent Calculus situation formula  $\Pi^f$  can be achieved by replacing each fluent atom  $P(\vec{x}, \sigma)$  in  $\Pi^s$  by  $Holds(P(\vec{x}), \sigma)$  and vice versa, as described in [Thi99, Appendix A], where  $\vec{x}$  are the arguments of the fluent  $P$ . This leads to the following definition of a translation function  $t$  between Situation Calculus and Fluent Calculus formulas:

$$\begin{aligned} t(P(\vec{x}, \sigma)) &= Holds(P(\vec{x}), \sigma) \\ t(t_1 = t_2) &= t_1 = t_2 \\ t(\neg\Pi) &= \neg t(\Pi) \\ t(\Pi_1 \wedge \Pi_2) &= t(\Pi_1) \wedge t(\Pi_2) \\ &\dots \end{aligned}$$

Here  $\vec{x}$  are terms of appropriate sorts for the arguments of fluent  $P$ ,  $t_1$  and  $t_2$  are terms and  $\Pi$ ,  $\Pi_1$  and  $\Pi_2$  are formulas uniform in the situation term  $\sigma$ . The function  $t$  has the intuitive meaning that  $t(a)$  denotes the translation of

a part  $a$  of a Situation Calculus domain axiomatization to its Fluent Calculus counterpart. Analogously, the reverse function  $t'(b)$  denotes the translation of a part  $b$  of a Fluent Calculus domain axiomatization to its Situation Calculus counterpart.

The reverse function  $t'$  for translating Fluent Calculus situation formulas to Situation Calculus uniform formulas is defined as follows:

$$\begin{aligned} t'(\text{Holds}(P(\vec{x}), \sigma)) &= P(\vec{x}, \sigma) \\ t'(t_1 = t_2) &= t_1 = t_2 \\ t'(\neg\Pi) &= \neg t'(\Pi) \\ t'(\Pi_1 \wedge \Pi_2) &= t'(\Pi_1) \wedge t'(\Pi_2) \\ &\dots \end{aligned}$$

**Example 3.1.** *The property  $Light$  of the cleaning robot domain (see section 2.1.4) is defined in Situation Calculus as follows:*

$$\begin{aligned} \text{Light}(s) &\stackrel{\text{def}}{=} (\exists x, y) \text{At}(x, y, s) \wedge \\ &(\text{Occupied}(x, y, s) \vee \text{Occupied}(x + 1, y, s) \vee \text{Occupied}(x - 1, y, s) \vee \\ &\text{Occupied}(x, y + 1, s) \vee \text{Occupied}(x, y - 1, s)) \end{aligned}$$

A translation to Fluent Calculus with  $t$  as defined above replaces the fluent  $\text{At}(x, y, s)$  by  $\text{Holds}(\text{At}(x, y), s)$  and acts similar for each occurrence of the fluent  $\text{Occupied}$ . The result is this formula:

$$\begin{aligned} \text{Light}(s) &\stackrel{\text{def}}{=} (\exists x, y) \text{Holds}(\text{At}(x, y), s) \wedge \\ &(\text{Holds}(\text{Occupied}(x, y), s) \vee \\ &\text{Holds}(\text{Occupied}(x + 1, y), s) \vee \text{Holds}(\text{Occupied}(x - 1, y), s) \vee \\ &\text{Holds}(\text{Occupied}(x, y + 1), s) \vee \text{Holds}(\text{Occupied}(x, y - 1), s)) \end{aligned}$$

*Translating this formula back to Situation Calculus by using  $t'$  yields the original formula.*

## 3.2 Knowledge Expressions

Assume a valid translation of the Situation Calculus basic action theory  $D$  to the Fluent Calculus domain axiomatization  $\Sigma$  and vice versa. Intuitively, under this assumption a translation function  $t$  of Situation Calculus objective situation-suppressed expressions to Fluent Calculus knowledge expressions should have the following property:

$$D \models \text{Knows}^s(\phi, s) \quad \text{iff} \quad \Sigma \models \text{Knows}^f(t(\phi), s)$$

A similar property should hold for the inverse function  $t'$ :

$$\Sigma \models \textit{Knows}^f(\phi, s) \quad \text{iff} \quad D \models \textit{Knows}^s(t'(\phi), s)$$

Provided that  $\textit{Knows}^s$  and  $\textit{Knows}^f$  have equivalent definitions in their calculi, a translation from Situation Calculus to Fluent Calculus that fullfills this property is:

$$t(\phi) = \phi$$

This is so simple because each Situation Calculus objective expression (see definition 2.4) is also a Fluent Calculus knowledge expression (see definition 2.11).

The inverse translation is a bit more difficult because a Fluent Calculus knowledge expression may contain non-fluent predicates, but the assumption made in section 2.1.1 does not allow non-fluent predicates in Situation Calculus and therefore also not in Situation Calculus objective expressions. There are two possibilities to deal with non-fluent relations of a Fluent Calculus domain. Either don't allow them, i.e. pose the same assumption as for Situation Calculus, or automatically transform them to fluents as described in section 2.1.1. The second solution would involve to incorporate the assertions made about the predicates in the intial database  $D_{S_0}$ . Because an implementation for this would require several restrictions on the definition of such a predicate<sup>1</sup>, I prefer the first solution for now. If later the restriction for Situation Calculus is lifted one can easily discard it for Fluent Calculus as well. That means I restrict Fluent Calculus domains not to contain non-fluent predicates, like it is done for Situation Calculus in section 2.1.1.

With the new restriction on a Fluent Calculus domain, a Fluent Calculus knowledge expression does not contain non-fluent predicates any longer, except for equality. Therefore the inverse translation function is the identity function as well:

$$t'(\phi) = \phi$$

### 3.3 Initial State Descriptions

The initial database of Situation Calculus  $D_{S_0}$  is a set of formulas about  $S_0$ . The initial knowledge state  $\Sigma_{init}$  of Fluent Calculus is a formula of the form:  $KState(S_0, z) \equiv \Phi(z)$ , where  $\Phi(z)$  is a Fluent Calculus state formula. So how can we make a relation between the two? The goal of both axiomatizations is to provide the the agent with specific knowledge about the initial situation of the world. If instead we had two formulas  $\textit{Knows}^s(\phi^s, S_0)$  and  $\textit{Knows}^f(\phi^f, S_0)$  we could reduce the problem to the translation between objective situation-suppressed expressions  $\phi^s$  and Fluent Calculus knowledge expressions  $\phi^f$ .

---

<sup>1</sup>The version of Golog with knowledge from [Rei01b], an implementation of Situation Calculus with knowledge, for example allows only finite domains for fluents. Therefore, describing non-fluent predicates by fluents would impose this restriction to non-fluent predicates as well.

Gladly, we can safely restrict  $D_{S_0}$  to  $D_{S_0} = \{Knows^s(\phi^s, S_0)\}$ , where  $\phi^s$  is objective, because properties of the initial situation that the agent does not know are of no use for the agent, when executing a strategy. For the implementation of Golog interpreters this is done anyways (see [Rei01b, 11.7.7]).

Together with the closed world assumption for knowledge<sup>2</sup> the Fluent Calculus expression  $Knows^f(\phi^f, S_0)$  is equivalent to  $KState(S_0, z) \equiv HOLDS(\phi^f, z)$  (see definition of  $Knows^f$  in section 2.2). Every state formula  $\Phi(z)$  can be transformed to an equivalent formula of the form  $HOLDS(\phi^f, z)$  and vice versa, if  $\phi^f$  is a knowledge expression without actions, i.e. without any occurrence of  $Poss(a)$ . Therefore, each knowledge state  $KState(S_0, z) \equiv \Phi(z)$  is equivalent to a formula  $Knows^f(\phi^f, S_0)$  for some  $\phi^f$ .

We can now define the translation between the restricted Situation Calculus initial database  $D_{S_0}$  and the Fluent Calculus initial knowledge state  $\Sigma_{init}$  to be:

$$\begin{aligned} t(D_{S_0} = \{Knows^s(\phi, S_0)\}) = \\ (\Sigma_{init} = \{(\forall z)KState(S_0, z) \equiv HOLDS(t(\phi), z)\}) \\ \text{and} \\ t'(\Sigma_{init} = \{(\forall z)KState(S_0, z) \equiv HOLDS(\phi, z)\}) = \\ (D_{S_0} = \{Knows^s(t'(\phi), S_0)\}) \end{aligned}$$

**Example 3.2.** *The Situation Calculus initial database of the example of section 2.1.4, where the robot knows his initial position and direction as well as the position of the corridor, is the following:*

$$\begin{aligned} Knows^s(At(1, 1) \wedge Facing(1) \wedge \\ \neg Occupied(1, 1) \wedge \neg Occupied(1, 2) \wedge \dots \wedge \neg Occupied(4, 5) \wedge \\ (\forall x, y)\neg Cleaned(x, y), S_0) \end{aligned}$$

*The translation function of a objective expression to a Fluent Calculus knowledge expression is the identity function, that means:*

$$\begin{aligned} t(At(1, 1) \wedge Facing(1) \wedge \dots \wedge (\forall x, y)\neg Cleaned(x, y)) = \\ At(1, 1) \wedge Facing(1) \wedge \dots \wedge (\forall x, y)\neg Cleaned(x, y) \end{aligned}$$

*Therefore a translation of this initial database to Fluent Calculus results in the following initial knowledge state:*

$$\begin{aligned} (\forall z)KState(S_0, z) \equiv HOLDS(At(1, 1) \wedge Facing(1) \wedge \\ \neg Occupied(1, 1) \wedge \neg Occupied(1, 2) \wedge \dots \wedge \neg Occupied(4, 5) \wedge \\ (\forall x, y)\neg Cleaned(x, y), z) \end{aligned}$$

*The translation of this Fluent Calculus initial knowledge state back to Situation Calculus yields the original initial database.*

---

<sup>2</sup>The closed world assumption for knowledge states that  $Knows(\phi, S_0)$  characterizes everything that is known initially. Every fact that does not follow from this is not known. [Rei01b]

### 3.4 Domain Constraints

The Fluent Calculus domain axiomatization contains domain constraints of the form  $(\forall s)Knows^f(\phi^f, s)$ . That are formulas restricting the set of possible states in each situation by defining general knowledge, i.e. invariant facts about the fluents and their values that do hold in every possible state.

In a Situation Calculus domain description as defined in section 2.1.3 there is nothing allowing to quantify a formula over all situations. Therefore, I add a set of domain constraints  $D_{dc}$  to the basic action theory of Situation Calculus.

**Definition 3.1 (domain constraints in Situation Calculus).** *A domain constraint in Situation Calculus with knowledge is a formula of the form*

$$(\forall s)(executable(s) \supset Knows^s(\phi^s, s))$$

where  $executable(s)$  denotes that every action leading to  $s$  was possible in its respective situation.

The macro  $executable(s)$  is defined in [Rei01b, 4.2.4] as follows:

$$executable(s) \stackrel{\text{def}}{=} (\forall a, s^*). Do(a, s^*) \sqsubseteq s \supset Poss(a, s^*) \quad (3.1)$$

Restricting the domain constraints to executable situations is required because otherwise they would contradict the other parts of the basic action theory, namely  $D_{sua} \cup D_{S_0} \cup D_{una}$ . The reason for this is, that the successor state axioms can be applied to a situation regardless of whether the situation is actually reachable by an executable sequence of actions. This problem does not exist in Fluent Calculus because all state update axioms and knowledge update axioms depend on the action to be possible in the current situation.

Now that we have introduced domain constraints for Situation Calculus we can define the translation function for them:

$$t((\forall s)(executable(s) \supset Knows^s(\phi, s))) = ((\forall s)Knows^f(t(\phi), s))$$

and

$$t'((\forall s)Knows^f(\phi, s)) = ((\forall s)(executable(s) \supset Knows^s(t'(\phi), s)))$$

**Example 3.3.** *Domain constraints in the cleaning robot domain consist of the following properties about fluents:*

- *There is always a unique position and direction of the robot, i.e.  $x, y, d$  in  $At(x, y)$  and  $Facing(d)$  are unique in every situation.*
- *Values for the position and direction of the robot as well as of the other fluents are restricted to the actual domain, i.e.  $x, y, d$  in  $At(x, y)$ ,  $Facing(d)$ ,  $Occupied(x, y)$  and in  $Cleaned(x, y)$  are restricted to  $\{1, 2, 3, 4, 5\}$  for  $x, y$  and  $\{1, 2, 3, 4\}$  for  $d$ .*

The domain constraints for the first item can be encoded in the following Situation Calculus domain constraints:

$$(\forall s)(executable(s) \supset Knows^s( \\ (\forall x, y, x', y') At(x, y) \wedge At(x', y') \supset x = x' \wedge y = y', s))$$

$$(\forall s)(executable(s) \supset Knows^s( \\ (\forall d, d') Facing(d) \wedge Facing(d') \supset d = d', s))$$

A translation to Fluent Calculus yields:

$$(\forall s)(Knows^f( \\ (\forall x, y, x', y') At(x, y) \wedge At(x', y') \supset x = x' \wedge y = y', s))$$

$$(\forall s)(Knows^f( \\ (\forall d, d') Facing(d) \wedge Facing(d') \supset d = d', s))$$

### 3.5 Action Preconditions

The action precondition axioms for Situation Calculus and Fluent Calculus defined in chapter 2 are incompatible. The Situation Calculus definition allows to describe knowledge of some property as a precondition for an action. But the Fluent Calculus definition does not, because it talks only about a state but not about a situation, and knowledge about a property can only be stated for a situation and not for a state.

On the other hand, Situation Calculus allows to describe properties which have to hold in situation  $s$ , but which are not known to hold, which is not of much use regarding an implementation, where the agent can't decide on things he doesn't know. The same problem exists in Fluent Calculus. Therefore, in Flux, the implementation of Fluent Calculus, a valid implementation of a precondition axiom has to entail that the agent knows that this precondition holds.

To overcome the differences between the definitions of precondition axioms I propose these definitions for both calculi:

**Definition 3.2 (Situation Calculus action precondition axiom).** *An action precondition axiom is a Situation Calculus formula of the form:*

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

where  $A$  is an  $n$ -ary action function symbol and  $\Pi_A^s$  is a Situation Calculus formula with free variables among  $\vec{x}$  and  $s$ , which consists of the standard logical connectives and whose only atoms are of the form  $Knows^s(\phi^s, s)$  (with  $\phi^s$  being objective).

**Definition 3.3 (Fluent Calculus action precondition axiom).** *An action precondition axiom is a Fluent Calculus formula of the form:*

$$Poss(A(\vec{x}), s) \equiv \Pi_A^f(\vec{x}, s)$$

where  $A$  is an  $n$ -ary action function symbol and  $\Pi_A^f$  is a Fluent Calculus formula with free variables among  $\vec{x}$  and  $s$ , which consists of the standard logical connectives and whose only atoms are of the form  $Knows^f(\phi^f, s)$  (with  $\phi^f$  being a Fluent Calculus knowledge expression).

I also have to redefine the notion of a knowledge expression not to include  $Poss(A(\vec{x}))$  anymore because this would translate to  $Poss(A(\vec{x}), z)$ , which is no longer defined in Fluent Calculus. It is easy to see that we do not lose but gain expressiveness in Fluent Calculus by this. We can express more facts by saying  $Poss(A(\vec{x}), s)$  with the new definition of precondition axioms than by saying  $Knows^f(Poss(A(\vec{x})), s)$  with the old definition, e.g. it was not possible to express a conjunction of positive and negative knowledge (lack of knowledge) with the old definition.

With those new definitions precondition axioms can be translated fairly easy by just translating the Situation Calculus formula  $\Phi_A^s$  to the Fluent Calculus formula  $\Phi_A^f$  and vice versa:

$$t(Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)) = (Poss(A(\vec{x}), s) \equiv t(\Pi_A(\vec{x}, s)))$$

and

$$t'(Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)) = (Poss(A(\vec{x}), s) \equiv t'(\Pi_A(\vec{x}, s)))$$

The Situation Calculus formula  $\Pi_A^s$  is translated to the Fluent Calculus formula  $\Pi_A^f$  by replacing each  $Knows^s(\phi^s, s)$  by  $Knows^f(\phi^f, s)$  and vice versa, where  $\phi^f = t(\phi^s)$  or  $\phi^s = t'(\phi^f)$  respectively:

$$\begin{aligned} t(Knows^s(\phi^s, s)) &= Knows^f(t(\phi^s), s) \\ t'(Knows^f(\phi^f, s)) &= Knows^s(t'(\phi^f), s) \end{aligned}$$

**Example 3.4.** *The precondition axioms of the cleaning robot domain have to be rewritten according to the new form of precondition axioms defined above. Thus, in Situation Calculus the new precondition axiom for the Go action is:*

$$\begin{aligned} Poss(Go, s) \equiv Knows^s((\forall d, x, y) At(x, y) \wedge Facing(d) \supset \\ (\exists x', y') Adjacent(x, y, d, x', y'), s) \end{aligned}$$

*The translation to Fluent Calculus yields:*

$$\begin{aligned} Poss(Go, s) \equiv Knows^f((\forall d, x, y) At(x, y) \wedge Facing(d) \supset \\ (\exists x', y') Adjacent(x, y, d, x', y'), s) \end{aligned}$$

### 3.6 Effects of Actions

Translating the effect descriptions of actions is the most difficult part of the translation of the domain descriptions because successor state axioms of Situation Calculus are fluent centered and state and knowledge update axioms of Fluent Calculus are action centered. I will deal with physical and cognitive effects separately. This is no problem in Situation Calculus as there are separate actions for sensing and normal effects. In Fluent Calculus as defined in chapter 2 an action can have both kinds of effects. For simplification I restrict the actions of Fluent Calculus to be either pure sensing actions (sensing some property of the domain), or pure physical actions (without any cognitive effect). One can easily transform an action  $A$ , which has both kinds of effects, to two actions  $A_{physical}$  and  $A_{cognitive}$ , with only physical or cognitive effects, respectively, which are than always executed directly one after the other.

#### 3.6.1 Physical Effects

In Situation Calculus physical effects of an action are defined by the successor state axioms of all fluents the action affects. In Fluent Calculus physical effects appear in two axioms, namely the state update and the knowledge update axiom of the action. For actions without cognitive effects both contain the same information, so it is sufficient to translate only the state update axioms.

For translating successor state axioms to state update axioms I will closely follow [Thi99] and I will reverse the obtained procedure for translating state update axioms to successor state axioms.

Assume, we have a (finite) set of successor state axioms of the form:

$$F_i(\vec{y}_i, Do(a, s)) \equiv \epsilon_{F_i}^+(\vec{y}_i, a, s) \vee F_i(\vec{y}_i, s) \wedge \neg \epsilon_{F_i}^-(\vec{y}_i, a, s)$$

Where  $\epsilon_{F_i}^+(\vec{y}_i, a, s)$  and  $\epsilon_{F_i}^-(\vec{y}_i, a, s)$  are conditions of positive and negative effect axioms of the form:

$$\epsilon_{F_i}^+(\vec{y}_i, a, s) \supset F_i(\vec{y}_i, Do(a, s))$$

and

$$\epsilon_{F_i}^-(\vec{y}_i, a, s) \supset \neg F_i(\vec{y}_i, Do(a, s))$$

Those general effect axioms can be transformed to effect axioms for a specific action  $A(\vec{x})$ :

$$\begin{aligned} \epsilon_{A, F_i}^+(\vec{x}, \vec{y}_i, s) &\supset F_i(\vec{y}_i, Do(A(\vec{x}), s)) \\ \epsilon_{A, F_i}^-(\vec{x}, \vec{y}_i, s) &\supset \neg F_i(\vec{y}_i, Do(A(\vec{x}), s)) \end{aligned}$$

Now we can apply the method described in [Thi99] and get a set of axioms of the form:

$$\Delta_k(s) \supset State(Do(A(\vec{x}), s)) \circ \vartheta_k^- = State(s) \circ \vartheta_k^+$$

Here  $\Delta_k(s)$  is one possible combination of positive and negative effect conditions  $\epsilon_{A,F_i}^+(\vec{x}, \vec{y}_i, s)$  and  $\epsilon_{A,F_i}^-(\vec{x}, \vec{y}_i, s)$  translated to Fluent Calculus formulas. The states  $\vartheta_k^+$  and  $\vartheta_k^-$  contain the fluents that are the positive and negative effects of action  $A(\vec{x})$  under condition  $\Delta_k(s)$ .

Those axioms can be easily transformed to

$$\Delta_k(s) \supset State(Do(A(\vec{x}), s)) = State(s) - \vartheta_k^- + \vartheta_k^+$$

by using the macros  $-$  and  $+$ . Because all those  $\Delta_k(s)$  are mutually exclusive, i.e. exactly one of them is true if the action  $A$  is possible, we can combine the axioms above to a state update axiom for action  $A$ :

$$\begin{aligned} Poss(A(\vec{x}), s) \supset & (\exists \vec{y}_1)(\Delta_1(s) \wedge State(Do(A(\vec{x}), s)) = State(s) - \vartheta_1^- + \vartheta_1^+) \\ & \vee \dots \vee \\ & (\exists \vec{y}_n)(\Delta_n(s) \wedge State(Do(A(\vec{x}), s)) = State(s) - \vartheta_n^- + \vartheta_n^+) \end{aligned}$$

The resulting axioms can be reduced by removing those parts where  $\Delta_k(s)$  is a contradiction. Which means that although the procedure described seems to result in an exponential blow-up in the size of the axioms, in a practical domain the axioms will not be much bigger than the successor state axioms. Typically an action influences only a small set of fluents, which means, that for most of the fluents  $F_i$  the conditions  $\epsilon_{A,F_i}^-(\vec{x}, \vec{y}_i, s)$  and  $\epsilon_{A,F_i}^+(\vec{x}, \vec{y}_i, s)$  are false, regardless of  $\vec{x}, \vec{y}_i$  and  $s$ .

**Example 3.5.** *The successor state axioms of the cleaning robot domain (including the trivial one for the auxiliary axiom  $Adjacent(x, y, d, x', y')$ ) are the following:*

$$Adjacent(x, y, d, x', y', Do(a, s)) \equiv Adjacent(x, y, d, x', y', s)$$

$$\begin{aligned} At(x, y, Do(a, s)) & \equiv \\ a = Go \wedge At(x', y', s) \wedge Facing(d', s) \wedge Adjacent(x', y', d', x, y, s) \vee \\ At(x, y, s) \wedge \neg a & = Go \end{aligned}$$

$$\begin{aligned} Facing(d, Do(a, s)) & \equiv a = Turn \wedge Facing(d', s) \wedge d \bmod 4 + 1 = d' \vee \\ Facing(d, s) \wedge \neg a & = Turn \end{aligned}$$

$$Occupied(x, y, Do(a, s)) \equiv Occupied(x, y, s)$$

$$Cleaned(x, y, Do(a, s)) \equiv a = Clean \wedge At(x, y, s) \vee Cleaned(x, y, s)$$

*These successor state axioms can be transformed to positive and negative effect axioms for each action. For example, for the Turn action we get the following*

axioms:

$$\begin{aligned}
Facing(d', s) \wedge d' \bmod 4 + 1 = d &\supset Facing(d, Do(Turn, s)) \\
Facing(d, s) &\supset \neg Facing(d, Do(Turn, s)) \\
\perp &\supset Adjacent(x, y, d, x', y', Do(Turn, s)) \\
\perp &\supset \neg Adjacent(x, y, d, x', y', Do(Turn, s)) \\
\perp &\supset At(x, y, Do(Turn, s)) \\
\perp &\supset \neg At(x, y, Do(Turn, s)) \\
&\dots
\end{aligned}$$

As can be seen, all axioms except the first two can be discarded. Because the Turn action changes only one fluent, namely  $Facing(d)$ , there is only one positive and one negative effect axiom. Following the method of [Thi99] we obtain the following four axioms, as combination of the two effect axioms:

$$\begin{aligned}
\neg(Holds(Facing(d'_1), s) \wedge d'_1 \bmod 4 + 1 = d_1) \wedge \neg Holds(Facing(d_2), s)) &\supset \\
State(Do(Turn, s)) \circ \emptyset = State(s) \circ \emptyset
\end{aligned}$$

$$\begin{aligned}
\neg(Holds(Facing(d'_1), s) \wedge d'_1 \bmod 4 + 1 = d_1) \wedge Holds(Facing(d_2), s)) &\supset \\
State(Do(Turn, s)) \circ Facing(d_2) = State(s) \circ \emptyset
\end{aligned}$$

$$\begin{aligned}
Holds(Facing(d'_1), s) \wedge d'_1 \bmod 4 + 1 = d_1 \wedge \neg Holds(Facing(d_2), s) &\supset \\
State(Do(Turn, s)) \circ \emptyset = State(s) \circ Facing(d_1)
\end{aligned}$$

$$\begin{aligned}
Holds(Facing(d'_1), s) \wedge d'_1 \bmod 4 + 1 = d_1 \wedge Holds(Facing(d_2), s) &\supset \\
State(Do(Turn, s)) \circ Facing(d_2) = State(s) \circ Facing(d_1)
\end{aligned}$$

According to the domain constraints the robot is facing a unique direction in every situation (although he does not necessarily know which one), therefore the premises of all but the last axiom are not satisfiable and we can discard those axioms. Because of the uniqueness of the direction the variables  $d'_1$  and  $d_2$  in the last axiom are identical, which leads to the following simplified axiom:

$$\begin{aligned}
Holds(Facing(d_2), s) \wedge d_2 \bmod 4 + 1 = d_1 &\supset \\
State(Do(Turn, s)) \circ Facing(d_2) = State(s) \circ Facing(d_1)
\end{aligned}$$

This remaining axiom results in the following state update axiom for the Turn action:

$$\begin{aligned}
Poss(Turn, s) &\supset (\exists d_1, d_2)(Holds(Facing(d_2), s) \wedge d_2 \bmod 4 + 1 = d_1 \wedge \\
&State(Do(Turn, s)) = \\
&State(s) - Facing(d_2) + Facing(d_1))
\end{aligned}$$

Now we can reverse this procedure and translate state update axioms to successor state axioms.

We can extract effect axioms

$$\begin{aligned}\epsilon_{A,F_i}^+(\vec{x}, \vec{y}_i, s) &\supset F_i(\vec{y}_i, Do(A(\vec{x}), s)) \\ \epsilon_{A,F_i}^-(\vec{x}, \vec{y}_i, s) &\supset \neg F_i(\vec{y}_i, Do(A(\vec{x}), s))\end{aligned}$$

for each action  $A$  out of the state update axiom with

$$\begin{aligned}\epsilon_{A,F_i}^+(\vec{x}, \vec{y}_i, s) &\stackrel{\text{def}}{=} \bigvee_{j \in I^+} [t'(\Delta_j(s))] \\ \epsilon_{A,F_i}^-(\vec{x}, \vec{y}_i, s) &\stackrel{\text{def}}{=} \bigvee_{j \in I^-} [t'(\Delta_j(s))]\end{aligned}$$

Here  $I^+$  and  $I^-$  are the sets of indices of those  $\vartheta_j^+$  and  $\vartheta_j^-$ , that contain  $F(\vec{y}_i)$ . That means  $\epsilon_{A,F_i}^+$  is the disjunction of all those conditions  $\Delta_j$  for which  $F(\vec{y}_i)$  is a positive effect of the action  $A$ , similar for  $\epsilon_{A,F_i}^-$ . It is easy to see that if the state update axiom is build from effect axioms as described above, then the disjunction of  $\Delta_j$  is equivalent to the original effect axiom.

Now we have positive and negative effect axioms for each pair of action  $A_j$  and fluent  $F_i$ . Those can be combined to general effect axioms for each fluent:

$$\begin{aligned}\epsilon_{F_i}^+(\vec{x}_i, a, s) &\supset F_i(\vec{x}_i, Do(a, s)) \\ \epsilon_{F_i}^-(\vec{x}_i, a, s) &\supset \neg F_i(\vec{x}_i, Do(a, s))\end{aligned}$$

where  $\epsilon_{F_i}^+$  and  $\epsilon_{F_i}^-$  are defined as follows:

$$\begin{aligned}\epsilon_{F_i}^+(\vec{x}_i, a, s) &\stackrel{\text{def}}{=} \bigvee_j [\epsilon_{A_j, F_i}^+(\vec{x}, s) \wedge a = A_j(\vec{x})] \\ \epsilon_{F_i}^-(\vec{x}_i, a, s) &\stackrel{\text{def}}{=} \bigvee_j [\epsilon_{A_j, F_i}^-(\vec{x}, s) \wedge a = A_j(\vec{x})]\end{aligned}$$

Now the general effect axioms can be combined to a successor state axiom for each fluent  $F_i$ :

$$F_i(\vec{x}_i, Do(a, s)) \equiv \epsilon_{F_i}^+(\vec{x}_i, a, s) \vee F_i(\vec{x}_i, s) \wedge \neg \epsilon_{F_i}^-(\vec{x}_i, a, s)$$

Note, that the method described above does not work if an action has so-called open effects. That means each effect axiom

$$\epsilon_{A,F_i}^+(\vec{x}, \vec{y}_i, s) \supset F_i(\vec{y}_i, Do(A(\vec{x}), s))$$

has to ensure that the variables  $\vec{y}_i$  are bound by  $\epsilon_{A,F_i}^+(\vec{x}, \vec{y}_i, s)$  if  $\vec{x}$  and  $s$  are given. Otherwise it is not possible to give an explicit definition of the effect  $\vartheta_i^+$ . The same applies for negative effect axioms. This restriction could possibly be lifted by allowing only finite domains for fluents and collecting all instances of a fluent that are implied by the effect axiom.

**Example 3.6.** *The state update axioms of the cleaning robot domain are the following:*

$$\begin{aligned} Poss(Go, s) \supset & (\exists d, x, y, x', y')(Holds(At(x, y), s) \wedge \\ & Holds(Facing(d), s) \wedge Adjacent(x, y, d, x', y') \wedge \\ & State(Do(Go, s)) = State(s) - At(x, y) + At(x', y')) \end{aligned}$$

$$\begin{aligned} Poss(Turn, s) \supset & (\exists d)(Holds(Facing(d), s) \wedge \\ & State(Do(Turn, s)) = \\ & State(s) - Facing(d) + Facing(d \bmod 4 + 1)) \end{aligned}$$

$$\begin{aligned} Poss(Clean, s) \supset & (\exists x, y)(Holds(At(x, y), s) \wedge \\ & State(Do(Clean, s)) = State(s) + Cleaned(x, y)) \end{aligned}$$

From these axioms one obtains the following effect axioms regarding the fluent  $At(x, y)$  and the action  $Go$ . For every other action the premises of the effect axioms are false, because only the  $Go$  action influences the fluent  $At(x, y)$ .

$$\begin{aligned} At(x, y, s) \wedge Facing(d, s) \wedge Adjacent(x, y, d, x', y', s) & \supset At(x', y', Do(Go, s)) \\ At(x, y, s) \wedge Facing(d, s) \wedge Adjacent(x, y, d, x', y', s) & \supset \neg At(x, y, Do(Go, s)) \end{aligned}$$

Now those effect axioms for specific actions can be transformed to general effect axioms:

$$\begin{aligned} At(x, y, s) \wedge Facing(d, s) \wedge Adjacent(x, y, d, x', y', s) \wedge a = Go & \supset \\ At(x', y', Do(a, s)) & \end{aligned}$$

$$\begin{aligned} At(x, y, s) \wedge Facing(d, s) \wedge Adjacent(x, y, d, x', y', s) \wedge a = Go & \supset \\ \neg At(x, y, Do(a, s)) & \end{aligned}$$

The combination of the positive and negative effect axiom and renaming of conflicting variables leads to the successor state axiom for  $At(x, y)$ :

$$\begin{aligned} At(x, y, Do(a, s)) \equiv & At(x', y', s) \wedge Facing(d, s) \wedge \\ & Adjacent(x', y', d, x, y, s) \wedge a = Go \\ & \vee \\ & At(x, y, s) \wedge \neg (At(x, y, s) \wedge Facing(d, s) \wedge \\ & Adjacent(x, y, d, x', y', s) \wedge a = Go) \end{aligned}$$

This axiom can be simplified when taking the domain constraints and the pre-condition axiom for *Go* into account. It is then:

$$\begin{aligned} At(x, y, Do(a, s)) &\equiv At(x', y', s) \wedge Facing(d, s) \wedge \\ &\quad Adjacent(x', y', d, x, y, s) \wedge a = Go \\ &\vee \\ &\quad At(x, y, s) \wedge \neg a = Go \end{aligned}$$

### 3.6.2 Cognitive Effects

In Situation Calculus cognitive effects are defined by the special successor state axiom for the  $K$  relation (see section 2.1.3) and the objective expression  $\Psi$  of each sense action  $Sense_{\Psi}$ . This is similiar for Fluent Calculus where each action that senses some property  $\Pi$  of the world has a knowledge update axiom stating that after executing the action each possible world state has to agree with the actual world state on the property  $\Pi$ .

For the sake of simplicity I placed the restriction on the Situation Calculus domain not to have functional fluents (see section 2.1.1). Because of this there are also no actions sensing the value of some fluent. A consequence of this is that if we want to translate the cognitive effects of actions from Fluent Calculus to the Situation Calculus, we have to restrict the actions in Fluent Calculus also to sense only properties of the world, but not values of fluents. To overcome this restricting it would be necessary to either change the successor state axiom for the  $K$  relation and with that the domain axiomatization of Situation Calculus. The other solution would be to directly map relational fluents of Fluent Calculus to functional fluents of Situation Calculus in such a way that one can have a Situation Calculus action  $Read_f(\vec{x})$  sensing the value  $y$  of fluent  $f(\vec{x}) = y$  for a Fluent Calculus action that senses  $y$  of  $f(\vec{x}, y)$ . This mapping would depend on additional information, which arguments of a fluent are mapped to the value of its functional fluent counterpart. Even then we still had to restrict the Fluent Calculus domain. Therefore I just add the restriction that Fluent Calculus sensing actions can sense only properties of the world (but not values of fluents) for the moment and leave the lifting of this restricting for future work.

A restricted knowledge update axiom for a sensing action  $A$  looks like this:

**Definition 3.4 (restricted knowledge update axiom).** *A knowledge update axiom for an action  $A$  sensing some property  $\Pi$  is a formula of the form*

$$\begin{aligned} Poss(A(\vec{x}), s) &\supset (\exists \vec{y})(\forall z')(KState(Do(A(\vec{x}), s), z) \equiv \\ &\quad (KState(s, z') \wedge [\Pi(z') \equiv \Pi(Do(A(\vec{x}), s))])) \end{aligned}$$

where  $\Pi(z')$  is a state formula with free variables among  $\vec{x}$ .

For further simplification I apply the same naming schema for sensing actions that is used in Situation Calculus to Fluent Calculus, i.e. a Fluent Calculus action sensing some property  $\Pi$  is named  $Sense_{\Pi}$  in the following.

The domains restricted in that way can now easily be translated:

- *From Situation Calculus to Fluent Calculus:*  
For each Situation Calculus sensing action  $Sense_{\Psi}(\vec{x})$  introduce a Fluent Calculus sensing action  $Sense_{\Pi}(\vec{x})$  with the following knowledge update axiom:

$$\begin{aligned} Poss(Sense_{\Pi}(\vec{x}), s) \supset & (\exists \vec{y})(\forall z')(KState(Do(Sense_{\Pi}(\vec{x}), s), z) \equiv \\ & (KState(s, z') \wedge \\ & [\Pi(z') \equiv \Pi(Do(Sense_{\Pi}(\vec{x}), s))])) \end{aligned}$$

where  $\Pi \stackrel{\text{def}}{=} t(\Psi)$  and  $\Pi(z) \stackrel{\text{def}}{=} HOLDS(\Pi, z)$ .

- *From Fluent Calculus to Situation Calculus:*  
For each Fluent Calculus sensing action  $Sense_{\Pi}(\vec{x})$  introduce a Situation Calculus sensing action  $Sense_{\Psi}(\vec{x})$  and add the following formula to the successor state axiom for the  $K$  relation:

$$(\forall \vec{x}_1)[a = Sense_{\Psi_1}(\vec{x}_1) \supset \Psi_1(\vec{x}_1, s^*) \equiv \Psi_1(\vec{x}_1, s)]$$

where  $\Psi \stackrel{\text{def}}{=} t'(\Pi)$ .

**Example 3.7.** *In the cleaning robot domain for Fluent Calculus the action Go does not comply with the restriction, that a sensing action must not have any physical effect. Therefore the action has to be split into a new Go action with only physical effects and the action Sense<sub>Light</sub> with the effect of sensing whether there is light in the room reached by the Go action.*

The knowledge update axiom for  $Sense_{Light}$ , according to the translation defined above, is:

$$\begin{aligned} Poss(Sense_{Light}, s) \supset & (\exists \vec{y})(\forall z')(KState(Do(sense_{Light}, s), z) \equiv \\ & (KState(s, z') \wedge \\ & [\Pi_{Light}(z') \equiv \Pi_{Light}(Do(Sense_{Light}, s))])) \end{aligned}$$

The property *Light* in Situation Calculus is defined in section 2.1.4 and is the translation of the property  $\Pi_{Light}$  of Fluent Calculus as defined in section 2.2.3.

The successor state axiom for  $K$ , according to the translation defined above, is:

$$\begin{aligned} K(s', Do(a, s)) \equiv & (\exists s^*).s' = Do(a, s^*) \wedge K(s^*, s) \wedge \\ & a = Sense_{Light} \supset Light(s^*) \equiv Light(s) \end{aligned}$$

This is the same axiom as defined in the example in section 2.1.4.

### 3.7 Auxiliary Axioms

Because we do not allow the domain axiomatization to contain non-fluent predicates other than *Poss* and equality, we don't have any auxiliary, i.e. situation independent, axioms. Instead assertions about those predicates are made in the initial state description in terms of their fluent counterpart (see sections 2.1.1 and 3.2).



## Chapter 4

# Proof of Conservation of Equivalence

In this chapter I will prove the correctness of the translation presented in the previous chapter. That means I will prove that a fact about a situation is entailed by a Situation Calculus domain axiomatization if and only if the translation of this fact to Fluent Calculus is entailed by the domain axiomatization translated to Fluent Calculus.

First I will prove the equivalence of the domain descriptions regarding knowledge of facts and truth of formulas about the situation  $S_0$ . I will then prove it for all reachable situations by induction.

**Proposition 4.1.**

$$D_{S_0} \cup \Sigma_{sit} \cup K_{init} \models Knows^s(\Pi, S_0) \quad (4.1)$$

*iff*

$$\Sigma_{init} \cup \Sigma_{state} \cup \Sigma_{knows} \models Knows^f(\Pi, S_0) \quad (4.2)$$

where

- $\Pi$  is a Situation Calculus objective expression (and therefore a Fluent Calculus knowledge expression at the same time)
- $D_{S_0} = Knows^s(\phi, S_0)$
- $\Sigma_{init} = (\forall z) KState(S_0, z) \equiv HOLDS(\phi, z)$ , ( $\Sigma_{init}$  is therefore  $D_{S_0}$  translated to a Fluent Calculus initial knowledge state and vice versa)

*Proof.* Because of the closed world assumption about knowledge, our only source of knowledge to be  $D_{S_0}$  and because of logical omniscience the equation (4.1) holds iff

$$\Sigma_{sit} \cup K_{init} \cup \{\phi\} \models \Pi \quad (4.3)$$

Neither  $\phi$  nor  $\Pi$  contain any notion of a situation or the  $K$  relation. Hence none of the axioms of  $\Sigma_{sit} \cup K_{init}$  contribute to the deduction of  $\phi \supset \Pi$ . Therefore equation (4.3) holds iff

$$\phi \models \Pi \quad (4.4)$$

Because  $\Sigma_{state} \cup \Sigma_{knows}$  is consistent and does not contain any axiom contributing to the deduction of  $\phi \supset \Pi$  either, equation (4.4) is equivalent to

$$\Sigma_{state} \cup \Sigma_{knows} \cup \{\phi\} \models \Pi \quad (4.5)$$

Under the closed world assumption about knowledge the initial knowledge state  $\Sigma_{init}$  is equivalent to  $Knows^f(\phi, S_0)$ . Therefore and because of logical omniscience (4.5) implies (4.2). Because there is no other way to deduce  $Knows^f(\Pi, S_0)$  than from  $\Sigma_{init}$  from the definition of  $Knows^f$  follows that (4.2) implies (4.5), too. Hence (4.5) is equivalent to (4.2).  $\square$

**Proposition 4.2.**

$$D_{S_0} \cup \Sigma_{sit} \cup K_{init} \models \Pi(S_0) \quad (4.6)$$

iff

$$\Sigma_{init} \cup \Sigma_{state} \cup \Sigma_{knows} \models t(\Pi(S_0))$$

where

- $\Pi(S_0)$  is a Situation Calculus formula uniform in  $S_0$
- $D_{S_0} = Knows^s(\phi, S_0)$
- $\Sigma_{init} = (\forall z)KState(S_0, z) \equiv HOLDS(\phi, z)$

Note that because of the reversibility of the translation function  $t$  ( $t(t'(x)) = x$  and  $t'(t(x)) = x$ ) this proposition applies to both directions of translation.

*Proof.* The only fact about  $S_0$  is  $Knows^s(\phi, S_0)$  therefore the only way to deduce  $\Pi$  to be true in  $S_0$  is if  $\Pi$  is known to be true. On the other hand proposition 2.1 says that everything that is known has to be true. Hence (4.6) holds iff

$$D_{S_0} \cup \Sigma_{sit} \cup K_{init} \models Knows^s(\Pi, S_0) \quad (4.7)$$

By proposition 4.1 equation (4.7) is equivalent to

$$\Sigma_{init} \cup \Sigma_{state} \cup \Sigma_{knows} \models Knows^f(\Pi, S_0) \quad (4.8)$$

Proposition 2.2 and the closed world assumption about knowledge now imply that (4.8) is equivalent to

$$\Sigma_{init} \cup \Sigma_{state} \cup \Sigma_{knows} \models HOLDS(\Pi, S_0)$$

Finally by the definition of  $HOLDS$  and the translation function  $t$  for uniform formulas follows that  $HOLDS(\Pi, S_0) = t(Pi(S_0))$ , which proves the claim.  $\square$

**Proposition 4.3.**

$$D_{S_0} \cup \Sigma_{sit} \cup K_{init} \models \Psi(S_0)$$

iff

$$\Sigma_{init} \cup \Sigma_{state} \cup \Sigma_{knows} \models t(\Psi(S_0))$$

where

- $\Psi(S_0)$  is a Situation Calculus formula about  $S_0$ , i.e. it may contain atoms of the form  $Knows^s(\phi, S_0)$
- $D_{S_0} = Knows^s(\phi, S_0)$
- $\Sigma_{init} = (\forall z)KState(S_0, z) \equiv HOLDS(\phi, z)$

*Proof.* The claim can be proved by induction over the structure of  $\Psi(S_0)$  with the help of propositions 4.1 and 4.2.  $\square$

Now I will show that adding the other parts of the domain axiomatization doesn't change the equivalence of deductions of formulas about the initial situation.

**Proposition 4.4.**

$$D_{S_0} \cup D_{dc} \cup \Sigma_{sit} \cup K_{Init} \models \Psi(S_0) \quad (4.9)$$

iff

$$\Sigma_{init} \cup \Sigma_{dc} \cup \Sigma_{state} \cup \Sigma_{knows} \models t(\Psi(S_0)) \quad (4.10)$$

where

- $\Psi(S_0)$  is a Situation Calculus formula about  $S_0$
- $D_{S_0} = Knows^s(\phi, S_0)$
- $\Sigma_{init} = (\forall z)KState(S_0, z) \equiv HOLDS(\phi, z)$
- $\Sigma_{dc}$  is the translation of  $D_{dc}$  (and vice versa)

*Proof.* The set of Situation Calculus domain constraints  $D_{dc}$  is a set of formulas of the form

$$(\forall s)(executable(s) \supset Knows^s(\Pi_i, s))$$

Because every initial situation is executable by definition, restricting the domain constraints to  $S_0$  leads to a set of formulas of this form  $Knows^s(\Pi_i, S_0)$ . These formulas can be integrated into the initial database:

$$D'_{S_0} = Knows^s(\phi \wedge \bigwedge_i \Pi_i, S_0)$$

It is easy to see that this new initial database  $D'_{S_0}$  is equivalent to  $D_{S_0} \cup D_{dc}$  regarding formulas about the initial situation, i.e. equation (4.9) holds iff

$$D'_{S_0} \cup \Sigma_{sit} \cup K_{Init} \models \Psi(S_0)$$

Similarly, the set of Fluent Calculus Domain constraints

$$\Sigma_{dc} = \{(\forall s)Knows^f(\Pi_i, s)\}$$

can be integrated into the initial knowledge state if restricted to  $S_0$ :

$$\Sigma'_{init} = (\forall z)KState(S_0, z) \equiv HOLDS(\phi \wedge \bigwedge_i \Pi_i, z)$$

The new initial knowledge state  $\Sigma'_{init}$  is equivalent to  $\Sigma_{init} \cup \Sigma_{dc}$  regarding formulas about  $S_0$ , i.e. equation (4.10) holds iff:

$$\Sigma'_{init} \cup \Sigma_{state} \cup \Sigma_{knows} \models t(\Psi(S_0))$$

From the definition of the translation function follows that  $D'_{S_0}$  and  $\Sigma'_{init}$  are translatable into each other, that means  $\Sigma'_{init} = t(D'_{S_0})$  and  $D'_{S_0} = t'(\Sigma'_{init})$ . Proposition 4.3 yields:

$$D'_{S_0} \cup \Sigma_{sit} \cup K_{init} \models \Psi(S_0)$$

iff

$$\Sigma'_{init} \cup \Sigma_{state} \cup \Sigma_{knows} \models t(\Psi(S_0))$$

This together with the two facts stated above proves the claim.  $\square$

**Proposition 4.5.**

$$D \models \Psi(S_0)$$

iff

$$\Sigma \models t(\Psi(S_0))$$

where

- $\Psi(S_0)$  a Situation Calculus formula about  $S_0$
- $D = D_{S_0} \cup D_{dc} \cup D_{ss} \cup D_{ap} \cup \Sigma_{sit} \cup D_{una} \cup K_{Init}$
- $\Sigma = \Sigma_{init} \cup \Sigma_{dc} \cup \Sigma_{sua} \cup \Sigma_{kua} \cup \Sigma_{poss} \cup \Sigma_{state} \cup \Sigma_{knows}$  is  $D$  translated to Fluent Calculus (and vice versa)

*Proof.* Because neither  $D_{ss}$  nor  $\Sigma_{sua} \cup \Sigma_{kua}$  make any claim about  $S_0$  and  $\Psi(S_0)$  does not mention *Poss* the claim follows from proposition 4.4.  $\square$

**Proposition 4.6.**

$$D \models Poss(A, S_0)$$

iff

$$\Sigma \models Poss(A, S_0)$$

for every action term  $A$  where

- $D = D_{S_0} \cup D_{dc} \cup D_{ss} \cup D_{ap} \cup \Sigma_{sit} \cup D_{una} \cup K_{Init}$

- $\Sigma = \Sigma_{init} \cup \Sigma_{dc} \cup \Sigma_{sua} \cup \Sigma_{kua} \cup \Sigma_{poss} \cup \Sigma_{state} \cup \Sigma_{knows}$  is  $D$  translated to Fluent Calculus (and vice versa)

*Proof.* Because a precondition axiom is a formula with the only atoms  $Knows(\Pi, s)$ ,  $Poss(A, S_0)$  is just a special case of a formula about  $S_0$ . Therefore the claim follows from the definition of the precondition axioms in both calculi and proposition 4.5.  $\square$

One main difference between the Fluent Calculus and the Situation Calculus in describing knowledge is the usage of possible states as opposed to possible situations. In Situation Calculus there is a set of possible situations, which are  $K$  related to each other, for each Fluent Calculus situation. In Fluent Calculus a set of possible states is connected with each situation by the  $KState$  relation. I will call the Fluent Calculus situation the *shared* situation because this situation exists in both calculi, whereas the situations  $K$  related to it are not defined within Fluent Calculus. That means  $S_0$  or any of its subsequent situation are *shared* situations.

An important fact is that for each possible situation  $s'$  in a shared situation  $s$  there is a possible state in  $s$  which agrees on an arbitrary property of  $s'$ , and vice versa.

**Proposition 4.7.** *Let  $s$  be an executable shared situation. For each objective expression  $\Pi$  the following property holds:*

$$D \models (\exists s') K(s', s) \wedge \Pi[s'] \quad (4.11)$$

*iff*

$$\Sigma \models (\exists z') KState(s, z') \wedge HOLDS(\Pi, z') \quad (4.12)$$

where  $\Sigma$  is the translation of  $D$ .

*Proof.* By induction over  $s$ .  
 $s = S_0$ :

Because there is no other information about any fluent in a initial situation except for  $D_{S_0} = Knows^s(\phi, S_0)$  the only way to deduce  $\Pi[s']$  is by knowing it. Hence (4.11) holds iff

$$D \models Knows^s(\Pi, S_0)$$

By proposition 4.5 this is equivalent to

$$\Sigma \models Knows^f(\Pi, S_0)$$

which is defined as

$$\Sigma \models (\forall z) KState(S_0, z) \supset HOLDS(\Pi, z) \quad (4.13)$$

By definition there is at least on state  $z$  with  $KState(S_0, z)$  namely  $State(S_0)$ . On the other hand there is no way to deduce  $HOLDS(\Pi, z')$  in an initial state, other than from  $\Sigma_{init}$ . Hence equation (4.13) holds iff equation (4.12) does.

Induction step ( $s = Do(a, s^*)$ ):

Because  $s$  is an executable situation  $a$  is possible in  $s^*$ , that means the precondition axiom for  $a$  is true. First assume  $\Pi$  to be a fluent term  $F(\vec{x})$ .

$$D \models (\exists s')K(s', Do(a, s^*)) \wedge F(\vec{x}, s') \quad (4.14)$$

Now make a case differentiation between sensing and non-sensing actions  $a$ . First let  $a$  be a non-sensing action. By the successor state axiom for  $K$  follows that equation (4.14) holds iff

$$D \models (\exists s')(\exists s'^*)s' = Do(a, s'^*) \wedge K(s'^*, s^*) \wedge F(\vec{x}, s')$$

By the successor state axiom for  $F$  this is equivalent to

$$D \models (\exists s'^*)s' = Do(a, s'^*) \wedge K(s'^*, s^*) \wedge (\gamma_F^+(\vec{x}, a, s'^*) \vee F(\vec{x}, s'^*) \wedge \neg\gamma_F^-(\vec{x}, a, s'^*))$$

Now from the induction hypotheses follows that this holds iff

$$\Sigma \models (\exists z'^*)KState(s^*, z'^*) \wedge HOLDS(\gamma_F^+(\vec{x}, a) \vee F(\vec{x}) \wedge \neg\gamma_F^-(\vec{x}, a), z'^*) \quad (4.15)$$

The result of [Thi99] says that equation (4.15) is equivalent to

$$\Sigma \models (\exists z')(\exists z'^*)KState(s^*, z'^*) \wedge \Psi(z', z'^*) \wedge Holds(F(\vec{x}), z')$$

where  $\Psi$  is the physical effect of action  $a$ , which is the same as the state update axiom. Now by the knowledge update axiom for  $a$  this is equivalent to

$$\Sigma \models (\exists z')KState(Do(a, s^*), z') \wedge Holds(F(\vec{x}), z')$$

Now let  $a$  be a sensing action  $sense_\psi(\vec{y})$ . Then equation (4.14) together with the successor state axioms for  $K$  and  $F$  is equivalent to

$$D \models (\exists s'^*)s' = Do(a, s'^*) \wedge K(s'^*, s^*) \wedge (\Psi(\vec{y}, s'^*) \equiv \Psi(\vec{y}, s^*)) \wedge F(\vec{x}, s'^*)$$

By the induction hypotheses this holds iff

$$\Sigma \models (\exists z'^*)KState(s^*, z'^*)(HOLDS(\Psi(\vec{y}), z'^*) \equiv HOLDS(\Psi(\vec{y}), s^*)) \wedge Holds(F(\vec{x}), z'^*)$$

With the knowledge update axiom of  $a$  this is equivalent to

$$\Sigma \models (\exists z')KState(Do(a, s^*), z') \wedge Holds(F(\vec{x}), z')$$

This proves the claim for  $\Pi = F(\vec{x})$ . The result can easily be extended to arbitrary objective expressions  $\Pi$  by structural induction. □

Now the induction step follows. Assume that propositions 4.5 and 4.6 hold for a reachable shared situation  $s$ , i.e. a situation which is executable according to equation (3.1) and which is a descendent of  $S_0$ . Now I prove that those propositions also hold for each executable situation  $Do(a, s)$ , for any action  $a$  which is possible in  $s$ .

**Proposition 4.8.**

$$D \models \text{Knows}^s(\Pi, Do(a, s)) \quad (4.16)$$

iff

$$\Sigma \models \text{Knows}^f(\Pi, Do(a, s)) \quad (4.17)$$

where  $\Sigma$  is  $D$  translated to *Fluent Calculus* (and vice versa) and  $Do(a, s)$  is an executable shared situation.

*Proof.* Per definition equation (4.16) holds iff

$$D \models (\forall s') K(s', Do(a, s)) \supset \Pi[s']$$

By proposition 4.7 this is equivalent to

$$\Sigma \models (\forall z') KState(Do(a, s), z') \supset HOLDS(\Pi, z')$$

Which is equation (4.17) by definition.  $\square$

**Proposition 4.9.**

$$D \models \Pi(Do(a, s)) \quad (4.18)$$

iff

$$\Sigma \models t(\Pi(Do(a, s))) \quad (4.19)$$

where  $\Sigma$  is  $D$  translated to *Fluent Calculus* (and vice versa) and  $Do(a, s)$  is an executable shared situation.

*Proof.* The only relevant fact to prove is that

$$D \models F(\vec{x}, Do(a, s)) \quad (4.20)$$

iff

$$\Sigma \models Holds(F(\vec{x}), Do(a, s)) \quad (4.21)$$

for every fluent term  $F(\vec{x})$ . The claim can then easily be proved for general Situation Calculus uniform formulas  $\Pi(Do(a, s))$  by structural induction.

For a non-sensing action  $a$  equation (4.20) is equivalent to

$$D \models \gamma_F^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg \gamma_F^-(\vec{x}, a, s) \quad (4.22)$$

By [Thi99] this is equivalent to equation (4.21).

For a sensing action  $a$  equation (4.20) is equivalent to

$$D \models F(\vec{x}, s) \quad (4.23)$$

which is equivalent to equation (4.21) by the induction hypotheses and the state update axiom for  $a$ .  $\square$

**Proposition 4.10.**

$$D \models \Psi(Do(a, s))$$

iff

$$\Sigma \models t(\Psi(Do(a, s)))$$

where

- $Do(a, s)$  is an executable shared situation
- $\Psi(Do(a, s))$  is a Situation Calculus formula about  $Do(a, s)$
- $\Sigma$  is  $D$  translated to Fluent Calculus (and vice versa)

*Proof.* The claim follows from propositions 4.8 and 4.9 by structural induction.  $\square$

**Proposition 4.11.**

$$D \models Poss(a_2, Do(a_1, s))$$

iff

$$\Sigma \models Poss(a_2, Do(a_1, s))$$

where  $\Sigma$  is  $D$  translated to Fluent Calculus (and vice versa) and  $Do(a_1, s)$  is an executable shared situation.

*Proof.* The claim follows from proposition 4.10 and the structure of precondition axioms.  $\square$

**Proposition 4.12.**

$$D \models \Psi(s)$$

iff

$$\Sigma \models t(\Psi(s))$$

and

$$D \models Poss(A, s)$$

iff

$$\Sigma \models Poss(A, s)$$

where

- $s$  is an executable shared situation
- $A$  is an action term
- $\Psi(s)$  is a Situation Calculus formula about  $s$
- $\Sigma$  is  $D$  translated to Fluent Calculus (and vice versa)

*Proof.* The claim follows by induction over  $s$  with the help of propositions 4.5, 4.6, 4.10 and 4.11.  $\square$

## Chapter 5

# Implementation in Prolog

In this chapter I will implement a translation between Situation Calculus and Fluent Calculus domain axiomatizations given in form of Golog and Flux terms. Golog is an implementation of the Situation Calculus in Prolog together with an abstract language for programming high-level strategies for intelligent agents. Flux is an implementation of the Fluent Calculus in Prolog. In contrast to Golog, which has its own syntax, formulas in Flux are encoded directly as Prolog clauses.

### 5.1 Foundations

A translation of domain axiomatizations from Golog with knowledge as defined in [Rei01b] to General Flux [Thi05] is possible only for a very restricted set of Golog domain axiomatizations. The reason for this is, that Flux-expressible formulas are only a subset of First Order Logic in order to make the computation much more efficient. More precisely within Flux it is only possible to describe the following forms of knowledge:

- existentially quantified positive and negated fluents
- universally quantified negated fluents
- positive disjunctions of fluents
- properties about the values of fluents

In particular, it is not possible to describe disjunctions of positive and negative fluents, which would be necessary for the translation of arbitrary Golog knowledge expressions.

A translation of such a restricted set of Golog domain axiomatizations is not considered very useful because for most of the domains it would involve to change the Golog domain descriptions manually before the translation. But this is not much less complex than a manual translation to Flux. An alternative is a translation algorithm that guides the user through the translation

process in such a way that the user gets a good presentation of changes which are needed and which effect they might have on other parts of the domain description or on strategies the agent might execute. Such an algorithm is beyond the scope of this thesis. Therefore, I will present a translation of domain axiomatizations with complete information, i.e. without the notion of knowledge or lack of knowledge. This can then be a basis for future work regarding a more comprehensive translation algorithm.

As basis for the translation I use the IndiGolog interpreter from the LeGolog distribution [Leg03] and Special Flux, a restricted Flux system that supports only complete information [Thi04]. This is the same as in [Sch04], where an interpreter for Golog strategies was developed. This made an execution of Golog strategies with Flux possible, but the domain axiomatizations had to be translated manually. Therefore together with the result of [Sch04] a complete translation of a Golog program, including the domain description and strategy, will be possible.

The restriction to complete information yields the following consequences regarding the domain axiomatizations given in chapter 2 (and redefined partly in chapter 3):

- There is no notion of knowledge, i.e. no  $Knows(\phi, s)$ , only one possible situation (or state respectively) and no knowledge update axioms. Because every fact is known, so there is no need to distinguish between facts that hold in a situation and facts that are known to hold.
- Because of complete information, i.e. every interesting aspect of the world state is known, and the assumption of the truth of the information there is no need for sensing actions.
- There are no domain constraints. In a domain with complete information domain constraints are defined implicitly by the initial state and the effects of actions.

## 5.2 Another Example Domain

Because the previous example from section 2.1.4, or 2.2.3 respectively, does not meet the restriction to complete information, I introduce another example to illustrate the implementation of the translation. It is a slightly adapted version of the mail delivery robot domain from [Thi04].

Consider a robot at an office floor, with offices numbered from 1 to 6, whose task it is to deliver packages from some offices to others. The robot has 3 slots or bags that can contain at most one package each. A possible initial situation of this scenario is depicted in Figure 5.1.

The robot keeps track of its current position, the packages it carries in its bags, and the requests (to bring a package from one room to another) still left to fulfill. To describe all this, there are the following fluents:

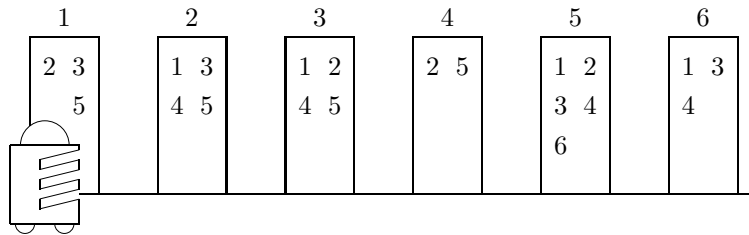


Figure 5.1: The initial state of a sample mail delivery problem (from [Thi04]), with a total of 21 requests.

- $\text{at}(\mathbf{R})$  means the robot is currently in front of room  $\mathbf{R}$ , where  $\mathbf{R}$  is a valid office number.
- $\text{carries}(\mathbf{B}, \mathbf{R})$  means the robot carries a package in bag  $\mathbf{B}$  for room  $\mathbf{R}$ , if  $\mathbf{R}$  is a valid office number, or the bag is empty if  $\mathbf{R} = \text{nothing}$ .
- $\text{empty}(\mathbf{B})$ , mail bag  $\mathbf{B}$  does not contain a package.
- $\text{request}(\mathbf{R}_1, \mathbf{R}_2)$ , there is an (unfulfilled) request to bring a package from room  $\mathbf{R}_1$  to  $\mathbf{R}_2$ .

The fluents  $\text{at}$  and  $\text{carries}$  are functional fluents whereas the other two are relational fluents. Note that the fluent  $\text{empty}(\mathbf{B})$  is redundant, because  $\text{carries}(\mathbf{B}, \text{nothing})$  carries the same information. So it has only illustrative purpose.

To accomplish its task the robot can execute three actions:

- $\text{go}(\mathbf{D})$ , where  $\mathbf{D}$  is either  $\text{up}$  or  $\text{down}$ , has the intended effect of moving the robot to the next office up or down the hallway.
- $\text{pickup}(\mathbf{B}, \mathbf{R})$  has the intended effect of picking up the package at the current room destined for room  $\mathbf{R}$  and putting in in mail bag  $\mathbf{B}$ .
- $\text{deliver}(\mathbf{B})$  finally delivers the package in mail bag  $\mathbf{B}$  to the current room.

The full definition of the domain in Golog and Flux including action precondition and effect axioms is listed in appendices C and D.

## 5.3 Fluents

In contrast to the assumption made in section 2.1.1, that there are no functional fluents, in IndiGolog there are none but functional fluents. Relational fluents are described by functional fluents with binary logical values (true or false). Fluents  $f(x_1, \dots, x_n)$  are defined in Golog by a predicate `prim_fluent/1`.

**Example 5.1.** *The definition of the fluents of the mailbot domain in Golog looks like this:*

```

prim_fluent(at).
prim_fluent(carries(Bag)) :- mailbag(Bag).

prim_fluent(request(FromRoom, ToRoom)) :-
    room(FromRoom), room(ToRoom).
prim_fluent(empty(Bag)) :- mailbag(Bag).

```

Here `room/1` and `mailbag/1` are predicates for auxiliary axioms defining valid room numbers and valid mail bags.

Note, in contrast to the definition of the fluents in section 5.2, the fluents `at(R)` and `carries(B,R)` are defined as functional fluents, i.e. as `at = R` and `carries(B) = R`. Therefore, the argument `R` that serves as the value of the fluent is left out of the definition of the fluents with `prim_fluent/1`. The fluents `request(FromRoom,ToRoom)` and `empty(Bag)` are also functional fluents, because there are only functional fluents in IndiGolog. But the value of these fluents is a binary boolean one, therefore they are functional fluents describing relational ones.

In Flux (and in Fluent Calculus) there are only relational fluents, but functional fluents can be described by relational fluents with the functional fluent's value as last argument. In that case the initial state and the state update axioms have to satisfy the condition that there is always a unique value for the fluent. There is no special predicate defining fluents, but as it is needed for the translation I introduce the same predicate `prim_fluent/1` as in Golog.

**Example 5.2.** *The definition of the fluents of the mailbot domain in Flux look like this:*

```

prim_fluent(at(Room)) :- room(Room).
prim_fluent(carries(Bag,Room)) :-
    mailbag(Bag), (room(Room) ; Room=nothing).

prim_fluent(request(FromRoom, ToRoom)) :-
    room(FromRoom), room(ToRoom).
prim_fluent(empty(Bag)) :- mailbag(Bag).

```

*The definitions of the fluents `at` and `carries` differ from the ones of Golog above, because there are only relational fluents in Flux, but `at` and `carries` are functional fluents. That means they have a unique value for the argument `Room` in each situation.*

Now it is possible to translate a (situation suppressed) Golog fluent  $f(x_1, \dots, x_n) = v$  to a Flux fluent  $f(x_1, \dots, x_n, v)$  by translating the initial state and the effect axioms in such a way that the new domain description satisfies the following condition:

**Condition 5.1.** *Let  $f$  be an  $n$ -ary Golog fluent function and  $\vec{x} = (x_1, \dots, x_n)$ .*

$$(\forall s, \vec{x}) \left[ \begin{array}{c} (\exists v) \text{Holds}(f(\vec{x}, v), s) \\ \wedge \\ (\forall v_1, v_2) \text{Holds}(f(\vec{x}, v_1), s) \wedge \text{Holds}(f(\vec{x}, v_2), s) \supset v_1 = v_2 \end{array} \right]$$

**Example 5.3.** *Thus the Golog fluent  $\text{at} = \mathbf{R}$  is translated to the Flux fluent  $\text{at}(\mathbf{R})$ .*

Translating a Golog fluent that serves as relational fluent, i.e. a functional fluent with binary logical value, to Flux in that way leads to an inefficiency. The reason is that in Flux a state typically consists only of those fluents that hold in a state and does not list fluents that do not hold, thereby reducing the size of the state and making state updates and inference more efficient. If we translate “relational” Golog fluents in the way described, than every instance of a fluent will be in the resulting state whether it is true or false. This results in a bigger Flux state which means that inference and updating is slower.

In order to avoid this inefficiency, I introduce a predicate `rel_fluent/1` in Golog that can be defined for all fluents that serve as relational fluents, and that are never used in a functional context, i.e. the fluents must only occur in place of a predicate and never in place of a function. Note, that defining this predicate is optional. The translation works without defining this predicate but the result might be less efficient. If `rel_fluent/1` is used, relational fluents can be directly translated to Fluent Calculus without adding arguments.

**Example 5.4.** *We define `empty` as relational fluent:*

`rel_fluent(empty(Bag)).`

*Then the Golog fluent `empty(B)` is identical to the Flux fluent `empty(B)`. Otherwise it would be translated to `empty(B, Value)`, with `Value` being either `true` or `false`. This would result in a bigger Flux state. If, for example, mail bag 2 was empty then `empty(2, false)` would be part of the state, but `empty(2)` would not be.*

The translation of fluents from Flux to Golog is more or less the reverse process. That means a Flux fluent  $f(\mathbf{x}_1, \dots, \mathbf{x}_n)$  can be translated to a Golog fluent  $f(\mathbf{x}_1, \dots, \mathbf{x}_n) = y$  with a binary logical value depending on whether  $f(\mathbf{x}_1, \dots, \mathbf{x}_n)$  holds or not in the state of question.

**Example 5.5.** *The Flux fluent `request(R1, R2)` is identical to the Golog fluent. That means it is translated to the (functional) Golog fluent `request(R1, R2)` with a binary logical value—true or false—depending on whether the Flux fluent `request(R1, R2)` holds or not.*

Now this translation is also inefficient for functional Flux fluents  $f(\mathbf{x}_1, \dots, \mathbf{x}_n, v)$ , i.e. fluents that satisfy condition 5.1, because it would result

in one Golog fluent  $f(x_1, \dots, x_n, v) = y$  with binary  $y$  for each instance of  $v$  although there is only one instance of  $v$  for which  $f(x_1, \dots, x_n, v)$  holds. The result if this would be bigger effect axioms and therefore slower inference.

To dissolve that inefficiency, I introduce an (optional) predicate `func_fluent/1` in Flux which can be defined for all fluents that serve as functional fluents with the last argument as value. Now functional fluents  $f(x_1, \dots, x_n, v)$  are translated to Golog as  $f(x_1, \dots, x_n) = v$ .

**Example 5.6.** *We define carries as functional fluent:*

```
func_fluent(carries(Bag,Room)).
```

*Then the Flux fluent carries(B,R) is translated to the Golog fluent carries(B) = R. Otherwise it would be translated to carries(B,R) = true and carries(B,V) = false for every value of V besides R. That means the Flux fluent carries(1,3) would be translated to Golog fluents carries(1,3) = true and carries(1,1) = false, carries(1,2) = false, carries(1,4) = false, etc., instead of just carries(1) = 3.*

## 5.4 Initial State

The initial state in Golog, i.e. the value  $v$  of each fluent  $f(x_1, \dots, x_n)$  in the initial situation  $s_0$ , is defined by a predicate `initially(f(x1, ..., xn), v)`. In (Special) Flux the initial state is typically defined by a predicate `init(Z0)`, where  $Z0$  is a list of ground fluents that hold in  $s_0$ .

Now the translation of the initial state from Golog to Flux can be easily achieved by building a list  $Z0$  in the following way:

- start with the empty list  $Z0 = []$
- for each instance of a functional Golog fluent  $f(x_1, \dots, x_n)$  add  $f(x_1, \dots, x_n, v)$  to  $Z0$  if `initially(f(x1, ..., xn), v)` holds
- for each instance of a relational Golog fluent  $f(x_1, \dots, x_n)$ , i.e. one for which `rel_fluent(f(x1, ..., xn))` is true, add  $f(x_1, \dots, x_n)$  to  $Z0$  if `initially(f(x1, ..., xn), true)` holds

It is easy to see, that if the definition of `initially` implies a unique value  $v$  for a functional fluent  $f(x_1, \dots, x_n) = v$  then the resulting initial state  $Z0$  does, too.

**Example 5.7.** *The following is an example initial state description in Golog:*

```
initially(at,1).
initially(empty(Bag),true).
initially(request(FromRoom, ToRoom), true) :-
    member((FromRoom,ToRoom),
    [(1,2), (1,3), (1,5), (2,1), (2,3), (2,4), (2,5), (3,1), (3,2),
```

```

    (3,4), (3,5), (4,2), (4,5), (5,1), (5,2), (5,3), (5,4), (5,6),
    (6,1), (6,3), (6,4)]).
initially(request(FromRoom, ToRoom), false) :-
    \+ initially(request(FromRoom, ToRoom), true).
initially(carries(Bag), nothing).

```

*The translation to Flux following the method described above results in:*

```

init(Z0) :-
    Z0 = [at(1), empty(1), empty(2), empty(3),
        request(1,2), request(1,3), request(1,5), request(2,1),
        request(2,3), request(2,4), request(2,5), request(3,1),
        request(3,2), request(3,4), request(3,5), request(4,2),
        request(4,5), request(5,1), request(5,2), request(5,3),
        request(5,4), request(5,6), request(6,1), request(6,3),
        request(6,4),
        carries(1, nothing), carries(2, nothing),
        carries(3, nothing)].

```

The translation of the Flux initial state Z0 to a definition of initially is similar:

- for each instance of a relational Flux fluent  $f(x_1, \dots, x_n)$  add `initially(f(x1, ..., xn), v)` with `v = true` if `holds(f(x1, ..., xn), Z0)` holds and `v = false` otherwise
- for each instance of  $x$  of a functional Flux fluent  $f(x_1, \dots, x_n, v)$ , i.e. one for which `func_fluent(f(x1, ..., xn, v))` is true, add `initially(f(x1, ..., xn), v)` if `holds(f(x1, ..., xn, v), Z0)`

**Example 5.8.** *The translation of the initial Flux state from the previous example to Golog results in:*

```

initially(at,1).
initially(empty(1),true). initially(empty(2),true).
initially(empty(3),true).
initially(request(1,2),true). initially(request(1,3),true).
initially(request(1,5),true). initially(request(2,1),true).
...
initially(request(6,3),true). initially(request(6,4),true).

initially(request(1, 1), false). initially(request(1, 4), false).
...
initially(request(6, 5), false). initially(request(6, 6), false).
initially(carries(1), nothing). initially(carries(2), nothing).
initially(carries(3), nothing).

```

*This description of the initial state is equivalent to the one of the previous example as long as one pays attention to the domains of the fluents defined by the `prim_fluent/1` predicate. That means for every ground fluent  $F$  with `prim_fluent(F)` both definitions of `initially(F,V)`, this one and the one of example 5.7, are equivalent.*

## 5.5 Precondition Axioms

Precondition axioms are encoded in Golog as a predicate `poss(a(X1, ..., Xn), πa(X1, ..., Xn))`. Here  $\pi_a(X_1, \dots, X_n)$  is a Golog condition term, or more specifically the Golog encoding of the situation suppressed expression associated to the uniform formula  $\Pi_A(x_1, \dots, x_n, s)$  of the precondition axiom  $Poss(A(x_1, \dots, x_n), s) \equiv \Pi_A(x_1, \dots, x_n, s)$  for action  $A$  in Situation Calculus. In Flux precondition axioms are encoded as predicates `poss(a(X1, ..., Xn), Z) : -π'a(X1, ..., Xn, Z)` where  $\pi'_a$  is the Prolog encoding of the Fluent Calculus state formula of the precondition axiom for  $A$ .

Therefore, the translation of precondition axioms between Golog and Flux is almost identical to the translation of Golog condition terms to Prolog encodings of Fluent Calculus state formulas and vice versa. This translation is presented in the following section.

## 5.6 Formulas

Formulas in Golog are encoded as condition terms which are processed by the Golog interpreter at runtime and tested against the current situation. A condition term is a Prolog term encoding a Situation Calculus situation suppressed expression. A condition term can be built of the following constructs (see [GL00] for details):

- `r(t1, ..., tn)`, for relational fluents  $r$ ,
- `p(t1, ..., tn)`, for non-fluent predicates  $p$ ,
- `not(c1)`, `or(c1, c2)` and `and(c1, c2)` for condition terms  $c_1$  and  $c_2$ ,
- and `some(v, c)` for condition terms  $c$ , where  $v$  is a Prolog constant encoding a variable

Because there are functional fluents in Golog, in contrast to definition 2.4 the terms  $t_1, \dots, t_n$  of these expressions might also contain functional fluents.

A formula in Flux is just a Prolog encoding of a Fluent Calculus state formula, i.e. a logical formula which contains fluents only in the form of atoms `holds(f, Z)`. Therefore, upon translating Golog condition expressions to Flux, each functional fluent  $f(x_1, \dots, x_n)$  contained in the terms  $t_1, \dots, t_n$  has to be replaced by a variable  $V$  and this variable has to be bound by `holds(f(x1, ..., xn, V), Z)`.

Golog condition terms can be translated to Flux formulas for state  $Z$  as follows:

- A relational fluent atom  $r(t_1, \dots, t_n)$  is translated to  $(\Pi, \text{holds}(r(t_1, \dots, t'_n), Z))$ , where  $t_1, \dots, t'_n$  is  $t_1, \dots, t_n$  with each occurrence of a functional fluent  $f(x_1, \dots, x_n)$  replaced by a new variable  $V$  and  $\Pi$  is the conjunction of  $\text{holds}(f(x_1, \dots, x_n, V), Z)$  for all those fluents.
- A non-fluent predicate  $p(t_1, \dots, t_n)$  is translated to  $(\Pi, p(t_1, \dots, t'_n))$ , where  $t_1, \dots, t'_n$  is  $t_1, \dots, t_n$  with each occurrence of a functional fluent  $f(x_1, \dots, x_n)$  replaced by a new variable  $V$  and  $\Pi$  is the conjunction of  $\text{holds}(f(x_1, \dots, x_n, V), Z)$  for all those fluents.
- Condition terms  $\text{not}(c_1)$ ,  $\text{or}(c_1, c_2)$  and  $\text{and}(c_1, c_2)$  are translated to  $\setminus + c'_1$ ,  $(c'_1 ; c'_2)$  and  $(c'_1 , c'_2)$  respectively, where  $c'_1, c'_2$  are the translations of  $c_1, c_2$ .
- A condition term  $\text{some}(v, c)$ , is translated to  $c'$  where  $c'$  is the translation of  $c$  with each occurrence of  $v$  replaced by the same new variable  $V$ .

**Example 5.9.** *The following is the Golog precondition axiom for the pickup action:*

`poss(pickup(Bag,Room), and( request(at, Room), empty(Bag) ) ) .`

*Following the algorithm above we translate the condition term `and(request(at,Room),empty(Bag))` to a Flux state formula and get the following Flux precondition axiom:*

`poss(pickup(Bag, Room), Z) :-  
 holds(at(Pos), Z), holds(request(Pos, Room), Z),  
 holds(empty(Bag), Z) .`

*Note, that the functional fluent `at` was replaced by a new variable `Pos` and this variable gets its value from `holds(at(Pos),Z)`.*

Flux formulas for state  $Z$  can be translated to Golog condition terms as follows:

- $\text{holds}(r(x_1, \dots, x_n), Z)$  for some relational fluent  $r$  is translated to  $r(x_1, \dots, x_n)$ .
- $\text{holds}(f(x_1, \dots, x_n, v), Z)$  for some functional fluent  $f$  is translated to  $f(x_1, \dots, x_n) = v$ .
- formulas  $\setminus + c_1$ ,  $(c_1 ; c_2)$ ,  $(c_1 , c_2)$ ,  $(c_1 - > c_2)$  and  $(c_1 - > c_2 ; c_3)$  are translated to  $\text{not}(c'_1)$ ,  $\text{or}(c'_1, c'_2)$ ,  $\text{and}(c'_1, c'_2)$ ,  $\text{and}(c'_1, c'_2)$  and  $\text{or}(\text{and}(c'_1, c'_2), \text{not}(\text{and}(c'_1, c'_3)))$  respectively, where  $c'_1, c'_2$  and  $c'_3$  are the translations of  $c_1, c_2$  and  $c_3$ .

- in the resulting formula all unbound variables  $V$  are replaced by a new Prolog constant  $v$  and the formula  $c$  is replaced by `some(v, c)`. Unbound variables are free variables in the formula that are not bound by the axiom the formula occurs in, e.g. in `poss(a(X, Y), f(X, B) > Y)` the variables  $X$  and  $Y$  are bound variables, but  $B$  is not, therefore the axiom would be replaced by `poss(a(X, Y), some(b, f(X, b) > Y))`.

**Example 5.10.** *The following is the Flux precondition axiom for the deliver action:*

```
poss(deliver(Bag), Z) :-
    holds(at(Pos), Z), holds(carries(Bag, Pos), Z).
```

*Following the algorithm above, one can translate the state formula, i.e. the body of the predicate, to a Golog condition term and get the following Golog precondition axiom:*

```
poss(deliver(Bag), some(pos, and(at = pos, carries(Bag) = pos))).
```

*Note, that the condition term could be simplified to `carries(Bag) = at` in principle. But the unsimplified axiom is often the faster one, particularly if the variable for the value of the functional fluent (`pos`, in this case) occurs more than once in the condition term. Simplifying would then result in multiple inferences of the same fluent's value on evaluating the condition term, which would slow down the evaluation.*

## 5.7 Effect Axioms

In IndiGolog the effects of actions are encoded as causal laws `causes_val(a, f, v,  $\epsilon_{a,f}$ )` corresponding to the effect axioms  $\epsilon_{A,F}(\vec{x}, \vec{y}, s) \supset F(\vec{x}, Do(A(\vec{y}), s)) = v$ . In Flux the effects of actions are encoded as state update axioms very similar to those of Fluent Calculus:

$$\begin{aligned}
 \text{state\_update}(Z_1, a(X_1, \dots, X_n), Z_2) : - \\
 \Delta_1(Z_1) \rightarrow \text{update}(Z_1, \theta_1^+, \theta_1^-, Z_2) \\
 ; \\
 \dots \\
 \Delta_n(Z_1) \rightarrow \text{update}(Z_1, \theta_n^+, \theta_n^-, Z_2)
 \end{aligned} \tag{5.1}$$

Here  $\Delta_i(Z_1)$  is a Prolog encoding of a Fluent Calculus state formula for state  $Z_1$  and  $\theta_i^+$ ,  $\theta_i^-$  are Flux states containing only variables from  $X_1, \dots, X_n$  or variables that are ground instantiated by  $\Delta_i(Z_1)$ .

The translation closely follows the method described in section 3.6.1 and therefore we also have to restrict the causal laws not to allow open effects. That means, given that  $a$  is ground, evaluating  $\epsilon_{a,f}$  in any (executable) situation  $s$  must result in  $f$  and  $v$  being ground.

Following the method described in section 3.6.1 we would have to translate all Golog condition terms  $\epsilon_{a,f}$  to Flux state formulas and make  $\Delta_1(Z_1), \dots, \Delta_n(Z_1)$  all possible combinations of those state formulas. This results in a blow-up of the size of the axioms that makes the resulting state update axioms more inefficient than necessary. Assume action  $a$  affects  $m$  different fluents. Now translating those  $m$  causal laws for action  $a$  would result in one state update axiom with  $2^m$  disjunctive connected formulas each consisting of a combination of the  $m$  (possibly negated) translated conditions of the causal laws. That is pretty big even if we assume that each action changes only a small set of fluents. Of course most of the combinations are unsatisfiable and the conditions of different causal laws for the same action often contain equal facts but identifying those redundancies automatically is not trivial. Therefore, I change the form of the resulting state update axiom to something that is logically equivalent but looks different. The translation of a set of causal laws  $\text{causes\_val}(a, f_i, V, \epsilon_{a,f_i})$  for some action  $a$  to a Flux state update axiom for action  $a$  is the following:

$$\begin{aligned}
& \text{state\_update}(Z_1, a, Z_{n+1}) : - \\
& \quad (\mathfrak{t}(\epsilon_{a,f_1}, Z_1), \rightarrow \\
& \quad \quad \text{holds}(f_1(V_{\text{old}}), Z_1), \text{update}(Z_1, [f_1(V)], [f_1(V_{\text{old}})], Z_2) \\
& \quad \quad ; Z_2 = Z_1) \\
& \quad , \\
& \quad \dots \\
& \quad (\mathfrak{t}(\epsilon_{a,f_n}, Z_1) \rightarrow \\
& \quad \quad (V = \text{true} \rightarrow \text{update}(Z_n, [f_n], [], Z_{n+1}); \text{update}(Z_n, [], [f_n], Z_{n+1})) \\
& \quad \quad ; Z_{n+1} = Z_n)
\end{aligned} \tag{5.2}$$

Here  $\mathfrak{t}(\epsilon_{a,f_i}, Z_i)$  denotes the translation of the Golog condition term to a Flux state formula for state  $Z_i$ . Just for illustration  $f_1$  is a functional fluent and  $f_n$  is a relational fluent. For each functional Golog fluent  $f(x_1, \dots, x_n)$  the Flux fluent  $f(x_1, \dots, x_n, V_{\text{old}})$  is removed from the state and  $f(x_1, \dots, x_n, V)$  is added to the state if the translated condition is true and  $V, V_{\text{old}}$  is the new and respectively old value of the fluent. If the translated condition of the corresponding causal law for a relational fluent  $r(x_1, \dots, x_n)$  is true then  $r(x_1, \dots, x_n)$  is added to the state if  $V$  is true and removed from the state otherwise.

**Example 5.11.** *In the Golog mailbot domain axiomatization there are three causal laws related to the pickup action:*

```

causes_val(pickup(Bag,Room), carries(Bag), Room, true).
causes_val(pickup(Bag,Room), empty(Bag), false, true).
causes_val(pickup(Bag,Room), request(FromRoom, Room), false,
           FromRoom=at).

```

*Translating these causal laws results in the following Flux state update axiom for pickup:*

```

state_update(Z1, pickup(Bag, Room), Z4) :-
    holds(carries(Bag, Vold), Z1),
        update(Z1, [carries(Bag, Room)], [carries(Bag, Vold)], Z2),
    update(Z2, [], [empty(Bag)], Z3),
    ( holds(at(FromRoom), Z1) ->
        update(Z3, [], [request(FromRoom, Room)], Z4)
    ; Z4 = Z3).

```

*The first and second clauses are simplified because the condition is always true. Note, that this axiom is much smaller than the equivalent one of the form of equation 5.1. That one would consist of 8 clauses from which 6 are unsatisfiable (in fact its only 1 satisfiable clause, because the condition `holds(at(FromRoom), Z1)` is always true). With this example identifying the 6 clauses would be possible, but it would be hard to do (if possible at all), if the conditions were non-trivial.*

This new form of state update axiom (equation 5.2) is not the same as the original definition of a state update axiom, but it is easy to see that there is a logically equivalent axiom of the original form for each state update axiom of the new form, more precisely it is the one obtained by using the method described in section 3.6.1.

The difference to the original form of a state update axiom (equation 5.1) is that there are  $m$  conjunctive formulas that are each just the size of one of the effect axioms (or causal laws) instead of  $2^m$  bigger disjunctive formulas. Instead of one state update with all effects at once there are several consecutive state updates with only one effect each. Nevertheless, the new form of state update axiom (equation 5.2) is more efficient because the Prolog interpreter has to evaluate less and smaller formulas. Furthermore, due to the implementation of the `update` predicate, only the number of effects (but not the number of calls of `update`) is relevant for the execution time.

Translating Flux state update axioms to Golog causal laws is easier than the translation to successor state axioms described in section 3.6.1 because the effect axioms don't have to be combined to successor state axioms.

Assume we have a state update axiom of the form of equation 5.1. Now for each relational fluent  $r$  we add a causal law `causes_val(a, r, true,  $\tau'(\Delta_1)$ )` to the set of causal laws if  $r$  is a member of  $\theta_1^+$  and we add `causes_val(a, r, false,  $\tau'(\Delta_1)$ )` if  $r$  is a member of  $\theta_1^-$ . For each functional fluent  $f(x_1, \dots, x_n, v)$  we add the causal law `causes_val(a, f(x_1, \dots, x_n), v,  $\tau'(\Delta_1)$ )` if  $f(x_1, \dots, x_n, v)$  is a member of  $\theta_1^+$ . Here  $\tau'(\Delta)$  denotes the Golog condition term obtained by translating  $\Delta$ .

**Example 5.12.** *The following is the state update axioms for deliver:*

```

state_update(Z1, deliver(Bag), Z2) :-
    holds(carries(Bag, Room), Z1),
    update(Z1, [empty(Bag), carries(Bag, nothing)],
        [carries(Bag, Room)], Z2).

```

The fluent `empty(Bag)` is a relational one, therefore the following causal is obtained:

```
causes_val(deliver(Bag), empty(Bag), true,
           carries(Bag) = Room).
```

The fluent `carries(Bag,R)` is a functional fluent with  $R$  serving as the value of the fluent. Therefore it is translated to `carries(Bag) = R` and the translation of the state update axiom yields the following causal law:

```
causes_val(deliver(Bag), carries(Bag), nothing,
           carries(Bag) = Room).
```

Note, that both causal laws contain an unnecessary condition. This is an unintentional side effect of the translation that cannot be avoided completely. In this case it would be possible to infer that the condition is redundant because `carries` is a functional fluent and the condition is therefore always true and because the variable `Room` is not used. This type of inference gets soon complicated for bigger conditions and one cannot even find all redundant conditions, because some redundancies might depend on domain constraints that are only implicitly given by the initial state and effect axioms.

## 5.8 Auxiliary Axioms

Auxiliary axioms or axioms not mentioning situations, states or fluents. They are encoded in IndiGolog and Flux in the same way, as Prolog predicates. Therefore, there is no difference between the auxiliary axioms of Golog and Flux and they can just be copied and a translation is not necessary.

**Example 5.13.** *The following are auxiliary axioms used in both, Golog and Flux, domain axiomatizations.*

```
room(Room) :- member(Room, [1,2,3,4,5,6]).
mailbag(Bag) :- member(Bag, [1,2,3]).
```



# Chapter 6

## Discussion

With this chapter I will conclude this thesis by giving a summary and presenting issues for future work.

The goal of this thesis was to develop a translation of descriptions of dynamic domains between Situation Calculus and Fluent Calculus. This translation should have the property that inferences of facts on the basis of a domain description are valid in one calculus exactly if these inferences are valid in the other calculus on the basis of the translated domain description. Such a translation was presented in chapter 3 and the required property was proved in chapter 4. For the development of the translation function I imposed the following restrictions on the domain axiomatisations, which may be lifted (partially) in future work:

- **no functional fluents**

[Rei01b] imposed the restriction not to allow functional fluents but only relational fluents in order to allow an easy implementation of a Golog interpreter that allows to reason about knowledge. This restriction can easily be lifted for Situation Calculus (although it might be more difficult for the Golog interpreter). But there are no functional fluents in Fluent Calculus. That means, one has to translate functional fluents of the Situation Calculus domain to relational fluents of the Fluent Calculus domain. This can be accomplished by adding the value of the functional fluent as additional argument for the relational fluent as it is done within the implementation presented in section 5.3. Of course, it has to be ensured by the translation of the initial state and of the effects of actions, that there is a unique value for the fluent in each situation.

- **no non-fluent predicates**

This restriction was also imposed by [Rei01b] in order to allow an easy implementation of the Golog interpreter. It is not really necessary for a formal translation between the calculi, but it simplifies the translation in chapter 3 and the proof in chapter 4. This assumption does not really restrict the class of describable domains, because it can be easily met

by syntactical transformations of the non-fluent predicates to fluents (see section 2.1.1). Therefore it can probably be lifted quite easily.

- **no open effects**

For the translation of Situation Calculus successor state axioms to Fluent Calculus state update axioms it is essential that actions do not have so called “open effects”. That means, the effect axiom

$$\epsilon_{A,F}(\vec{x}, \vec{y}, s) \supset F(\vec{y}, Do(A(\vec{x}), s))$$

of an action  $A(\vec{x})$  regarding some fluent  $F(\vec{y})$  has to ensure that in a situation  $s$  for each instance of  $\vec{x}$  there is at most one instance of  $\vec{y}$  for which  $\epsilon_{A,F}(\vec{x}, \vec{y}, s)$  is true. That means, that if we allow only one effect axiom for each pair of action and fluent (which is the case if we obtain the effect axioms from successor state axioms as described in section 3.6.1), an action can only change one instance of a fluent at one time.

This restriction cannot be given up fully. That means, it is not possible to describe actions with open effects, i.e. actions that change an infinite number of instances of a fluent, in Fluent Calculus. That would result in an state update with an infinite state, i.e.  $State(Do(a, s)) = State(s) + \theta^+ - \theta^-$  with infinite states  $\theta^+$  or  $\theta^-$ , which is not possible. But one can lift the restriction to allow actions that affect a finite number of instances of a fluent. This could be accomplished by obtaining not only one pair of positive and negative effect axiom for each pair of action and fluent from the successor state axioms, but having one (positive and negative) effect axiom for each instance of the fluent. In order not to obtain too many effect axioms considerable intelligence of the domain is needed to reduce the number of effect axioms. This would be necessary because the number of effect axioms considerably affects the size of the state update axioms obtained by the translation.

The third part of the thesis was to implement a translation of domain descriptions between Golog and Flux. Existing implementations of Golog and Flux have very different approaches for the realization of uncertainty and knowledge. The inference in the Golog interpreter with knowledge presented by [Rei01b] is based on a theorem prover for propositional logic. In Flux a constraint solver is used that is complete only for a subset of first order logic. The translation between domain descriptions of both programming languages would require severe restrictions to the domains that would make the translation either very difficult or not much useful.

Therefore, I implemented the translation between a Golog variant without knowledge and Special Flux. This clearly restricts the use of the implementation to domains with complete information, i.e. without uncertainty. But it is usable, at least for this domains, and it can serve as a basis for a more sophisticated implementation of a translation between domains with uncertainty and knowledge. Because the Golog and Flux variants used are the same as the ones used in [Sch04] together with this work there exists now a way to completely execute a Golog agent program with the Flux system.

The following related issues are open for future work:

- One task is to prove that the implementation of the translation is correct. Then, together with the result of [Sch04], it will be proved that executing a Golog agent program with the Flux system has the same result as the execution with the Golog interpreter.
- Now that there is a translation between Golog and Flux (at least for deterministic domains without uncertainty) it is possible to analyse the run-time behaviour of both languages when applied to certain domains with different tasks (on-line execution, planning, ...). It might even be possible to develop mechanism to automatically choose the language that is more efficient for a given task at run-time.
- The translation function may be implemented to translate between Golog and Flux with knowledge and uncertainty. The problems described above may be solved by changes to the Flux system or by implementing the translation in such a way that it (intelligently) proposes changes to the domain that make the translation possible.



# Appendix A

## Golog to Flux Translator

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% The Golog to Flux translator
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

outputStream(stdout).

run_golog2flux :-
    trans_prim_fluent,
    trans_prim_action,
    trans_state,
    trans_poss,
    trans_ssa
    .

run_golog2flux(Filename) :-
    open(Filename,write,OS),
    retract_all(outputStream(_)),
    assert(outputStream(OS)),
    run_golog2flux,
    retract(outputStream(OS)),
    close(OS),
    assert(outputStream(stdout)).

% just in case it doesn't exist in the particular domain
% description:
proc(none,none) :- false.
rel_fluent(none) :- false.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% translate prim_fluent axioms
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

trans_prim_fluent :-
```



```

% translate effect descriptions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

trans_ssa :-
    findall([A_name,Len], (causes_val(Ac, _, _, _),
        Ac=..[A_name|Args], length(Args,Len)), Actions2),
    setof(A, member(A,Actions2),Actions),
    process_every_ssa_action(Actions),
    outputStream(OS), write(OS,"% all ssa actions:"),
    my_write_term(Actions), nl(OS).

process_every_ssa_action([]):- !.

process_every_ssa_action([[A_name,Len]|Actions]):-
    outputStream(OS),
    length(Args2,Len),
    A=..[A_name|Args2],
    findall([Args,F,V,C], (causes_val(Ac, F, V, C),
        Ac=..[A_name|Args]), L),
    my_write_term(state_update(Z, A, ZP)), write(OS, " :- "),
    process_every_causes_val(Args2, L, Z, Z, ZP),
    write(OS,"."), nl(OS), nl(OS),
    process_every_ssa_action(Actions).

process_every_causes_val(Args, [[Args2,F,V,C]], Z0, Z, ZP) :- !,
    process_one_causes_val(Args, Args2, F, V, C, Z0, Z, ZP),
    outputStream(OS),
    nl(OS).

process_every_causes_val(Args, [[Args2,F,V,C]|L], Z0, Z, ZP) :-
    process_one_causes_val(Args, Args2, F, V, C, Z0, Z, Z2),
    outputStream(OS),
    write(OS, ","),
    nl(OS),
    process_every_causes_val(Args, L, Z0, Z2, ZP).

process_one_causes_val(Args, Args2, F, V, C, Z0, Z, ZP) :-
    trans_uniform_formula(C, CP, Z0),
    outputStream(OS),
    write(OS, "( "),
    ( (\+ Args2 = []) ->
        my_write_term(Args2 = Args), write(OS, ", ")
        ;
        true
    ),
    ((\+ rel_fluent(F)) ->
        write_effect_func_fluent(F, V, CP, Z0 ,Z ,ZP)

```

```

        ;
        write_effect_rel_fluent(F, V, CP, Z0, Z, ZP)
    ),
    write(OS, ")").

write_effect_rel_fluent(F, V, CP, Z0, Z, ZP) :-
    trans_fluent_g2f(F, V, FP),
    % fluent is positive effect if new value is "true" and
    % negative effect if new value is "false"
    (V=true -> EP=[FP], EN=[] ; EP=[], EN=[FP]),
    ( (\+ CP = true) ->
        my_write_term((CP -> update(Z, EP, EN, ZP) ; ZP=Z))
    );
    my_write_term(update(Z, EP, EN, ZP))
).

write_effect_func_fluent(F, V, CP, Z0, Z, ZP) :-
    trans_fluent_g2f(F, V, FP),
    trans_fluent_g2f(F, Vold, Fold),
    ( (\+ CP = true) ->
        % remove old value from and add new value to state
        my_write_term(
            (CP ->
                holds(Fold, Z0), update(Z, [FP], [Fold], ZP)
            ; ZP=Z))
    );
    my_write_term(
        (holds(Fold, Z0), update(Z, [FP], [Fold], ZP))
    ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% translate initial state
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

trans_state :-
    assert(initial_state([])), retract_all(initial_state(_)),
    assert(initial_state([])),
    get_every_initially,
    initial_state(Z),
    retract(initial_state(Z)),
    my_write_clause(init(Z0) :- Z0=Z),
    outputStream(OS), nl(OS).

get_every_initially :-
    prim_fluent(F),
    initially(F,V),
    initial_state(Z),

```

```

    (rel_fluent(F) -> V=true ; true),
    % relational fluents with value "false" are not added
    % to the initial state for efficiency
    retract(initial_state(Z)),
    trans_fluent_g2f(F,V,FP),
    assert(initial_state([FP|Z])),
    fail.

get_every_initially :- !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% trans_fluent_g2f(Ffunc,Value,Frel)
%
% translate functional fluents (Golog)
% to relational fluents (Flux)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

trans_fluent_g2f(Ffunc,Value,Frel) :-
    (\+ rel_fluent(Ffunc)) ->
        Ffunc =.. [Fn|L],
        append(L,[Value],L2),
        Frel =.. [Fn|L2]
    ;
    Frel=Ffunc.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% trans_uniform_formula(SitCalcF, FluentCalcF, State)
%
% translate SitCalc uniform formulas to FluentCalc state formulas
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

trans_uniform_formula(and(P1, P2), P3, State) :- !,
    trans_uniform_formula(P1, P1F, State),
    trans_uniform_formula(P2, P2F, State),P3=(P1F, P2F).

trans_uniform_formula(or(P1, P2), (P1F; P2F), State) :- !,
    trans_uniform_formula(P1, P1F, State),
    trans_uniform_formula(P2, P2F, State).

trans_uniform_formula(neg(P), \+(PF), State) :- !,
    trans_uniform_formula(P, PF, State).

trans_uniform_formula(some(V, P), PF, State) :- !,
    subv(V, Var, P, P1),
    trans_uniform_formula(P1, PF, State).

% translate complex conditions encoded as procedures

```

```

trans_uniform_formula(P, PF, State) :-
    proc(P,P1), !,
    trans_uniform_formula(P1, PF, State).

% translate predicate P
trans_uniform_formula(P, PF, State) :- !,
    trans_term(P, P1, State, Constraints),
    append_constraints(Constraints, P1, PF).

% trans_term(T1, T2, State, Constraints)
% translate Golog term T1 to Flux term T2 plus additional
% constraints of the form "holds(F,State)"

% variable
trans_term(T1, T2, _, true) :-
    var(T1), !,
    T2 = T1.

% function
trans_term(P1, P3, State, Constraints) :-
    P1 =..[F|L1],
    trans_term_list(L1, L2, State, Constraints1),
    P2 =..[F|L2],
    trans_term2(P2, P3, State, Constraints2),
    append_constraints(Constraints1, Constraints2, Constraints).

% fluent function
trans_term2(T1, V, State, Constraints) :-
    \+ (not prim_fluent(T1)), !,
    trans_fluent_g2f(T1, V, T3),
    (\+ rel_fluent(T1) ; V=true),!,
    Constraints=holds(T3,State).

% non-fluent function
trans_term2(T1, T1, _, true) :-
    \+ prim_fluent(T1).

% translate list of terms
trans_term_list([], [], _, true) :- !.

trans_term_list([T1|L1], [T2|L2], State, Constraints) :-
    trans_term(T1, T2, State, Constraints1),
    trans_term_list(L1, L2, State, Constraints2),
    append_constraints(Constraints1, Constraints2, Constraints).

append_constraints(C1, C, C) :- C1==true, !.
append_constraints(C, C2, C) :- C2==true, !.

```

```

append_constraints(C1, C2, (C1, C2)) :- !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% subv(X1, X2, T1, T2)
%
% substitute X1 by X2 in T1, result is T2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

subv(_, _, T1, T2) :-
    var(T1), !, T2 = T1.
subv(X1, X2, T1, T2) :-
    T1 = X1, !, T2 = X2.
subv(X1, X2, T1, T2) :-
    T1 =..[F|L1], subvl(X1, X2, L1, L2), T2 =..[F|L2].

subvl(_, _, [], []).
subvl(X1, X2, [T1|L1], [T2|L2]) :-
    subv(X1, X2, T1, T2), subvl(X1, X2, L1, L2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% print terms to outputStream
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

my_write_term(T) :- outputStream(OS), my_write_term(OS,T).
my_write_term(OS,T) :-
    write_term(OS,T, [variables(full), depth(full), as(term),
                     attributes(full), depth(full)]).

my_write_clause(T) :- outputStream(OS), my_write_clause(OS,T).
my_write_clause(OS,T) :-
    write_term(OS,T, [variables(full), depth(full), as(clause),
                     attributes(full), depth(full)]),
    write(OS,','), nl(OS).

```



## Appendix B

# Flux to Golog Translator

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% The Flux to Golog translator
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- ['special_flux'].

outputStream(stdout).

run_flux2golog :-
    trans_prim_fluent,
    trans_prim_action,
    trans_state,
    trans_poss,
    trans_sua
    .

run_flux2golog(Filename) :-
    open(Filename,write,OS),
    retract_all(outputStream(_)),
    assert(outputStream(OS)),
    run_flux2golog,
    retract(outputStream(OS)),
    close(OS),
    assert(outputStream(stdout)).

% just in case it doesn't exist in the particular domain
% description:
func_fluent(none) :- false.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% translate prim_fluent axioms
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```

trans_prim_fluent :-
    get_every_prim_fluent,
    my_write_clause(prim_fluent(F):-func_fluent(F)),
    outputStream(OS), nl(OS).

get_every_prim_fluent :-
    prim_fluent(F),
    trans_fluent_f2g(F, Fp, V),
    (func_fluent(F) ->
        my_write_clause(func_fluent(Fp))
    ; my_write_clause(prim_fluent(Fp))),
    fail.

get_every_prim_fluent :- !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% translate prim_action axioms
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

trans_prim_action :-
    get_every_prim_action,
    outputStream(OS), nl(OS).

get_every_prim_action :-
    prim_action(A),
    my_write_clause(prim_action(A)),
    fail.

get_every_prim_action :- !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% translate precondition axioms
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

trans_poss :-
    get_every_action.

get_every_action :-
    prim_action(A),
    clause(poss(A,Z), X),
    trans_state_formula([A],X,Z,F),
    my_write_clause(poss(A,F)),
    outputStream(OS), nl(OS),
    fail.

get_every_action :- !.

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% translate effect descriptions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

trans_sua :-
    process_every_sua,
    outputStream(OS),nl(OS).

process_every_sua :-
    outputStream(OS),
    clause(state_update(Z1,A,Z2), F),
    process_single_updates(Z1,A,Z2,F),
    nl(OS),
    fail.

process_every_sua :- !.

% process_single_updates(Z1,A,Z2,F)
% processes every single "D -> update(Z1,T-,T+,Z2)"-line from F
% and writes appropriate effect axioms causes_val(A,Fluent,V,C)

% multiple updates (with different conditions)
process_single_updates(Z1,A,Z2,(F1;F2)) :- !,
    process_single_updates(Z1,A,Z2,F1),
    process_single_updates(Z1,A,Z2,F2).

% a single update
process_single_updates(Z1,A,Z2,F) :- !,
    % decompose F in condition D and update
    % update must be the last clause in F
    decompose(F,D,update(Z1,Ep,Em,Z2)),
    trans_state_formula([A,Em,Ep],D,Z1,Dp),
    process_effects(Em,negative,A,Dp),
    process_effects(Ep,positive,A,Dp).

decompose(update(Z1,Ep,Em,Z2),true,update(Z1,Ep,Em,Z2)) :- !.
decompose((F1,update(Z1,Ep,Em,Z2)),F1,update(Z1,Ep,Em,Z2)) :- !.
decompose((F1,F2),(F1,F2p),update(Z1,Ep,Em,Z2)) :- !,
    decompose(F2,F2p,update(Z1,Ep,Em,Z2)).

process_effects([],_,-,-) :- !.

process_effects([Fluent|Effects],Type,Action,Cond) :- !,
    trans_fluent_f2g(Fluent,Fp,Value),
    ((\+ func_fluent(Fluent)) ->
        (Type=positive -> Value=true ; Value=false),
        my_write_clause(causes_val(Action,Fp,Value,Cond))

```

```

; % else (Fluent is a functional fluent)
  % don't write effect axioms for negative effects of
  % functional fluents
  (Type=positive ->
    my_write_clause(causes_val(Action,Fp,Value,Cond))
  ; true)
),
process_effects(Effects,Type,Action,Cond).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% translate initial state
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

trans_state :-
  init(Z0),!,
  get_every_initially(Z0),
  outputStream(OS), nl(OS).

get_every_initially(Z0) :-
  prim_fluent(F),
  (func_fluent(F) ->
    holds(F,Z0)
  ; % else
    (holds(F,Z0) -> Value=true ; Value=false)
  ),
  trans_fluent_f2g(F,Fp,Value),
  my_write_clause(initially(Fp,Value)),
  fail.

get_every_initially(_) :- !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% trans_fluent_f2g(Frel,Ffunc,Value)
%
% translate relational fluents (Flux) to
% functional fluents (Golog)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

trans_fluent_f2g(Frel,Ffunc,Value) :-
  (\+ (\+ func_fluent(Frel)) ->
    Frel=..[Fname|Args],
    append(Args2,[Value],Args),
    Ffunc=..[Fname|Args2]
  ; % else
    Ffunc=Frel
  )
.

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% trans_state_formula(Boundby, FluentCalcF, State, SitCalcF)
%
% - all unbound variables are bound in SitCalcF by 'some(v,F)'
% - Boundby is a term that binds some variables of FluentCalcF
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% trans_state_formula(FluentCalcF, State, SitCalcF)
%
% translate SitCalc uniform formulas to FluentCalc state formulas
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

trans_state_formula(Boundby, P, Z, PS) :-
    trans_state_formula(P, Z, Pp),
    bind_unbound_vars(Boundby, Pp, PS).

trans_state_formula((P1, P2), Z, and(P1S,P2S)) :- !,
    trans_state_formula(P1, Z, P1S),
    trans_state_formula(P2, Z, P2S).

trans_state_formula((P1 -> P2 ; P3), Z,
    or(and(P1S,P2S), and(neg(P1S),P3S)) ) :- !,
    trans_state_formula(P1, Z, P1S),
    trans_state_formula(P2, Z, P2S),
    trans_state_formula(P3, Z, P3S).

trans_state_formula((P1; P2), Z, or(P1S,P2S)) :- !,
    trans_state_formula(P1, Z, P1S),
    trans_state_formula(P2, Z, P2S).

trans_state_formula((P1 -> P2), Z, and(P1S,P2S)) :- !,
    trans_state_formula(P1, Z, P1S),
    trans_state_formula(P2, Z, P2S).

trans_state_formula(not P, Z, neg(PS)) :- !,
    trans_state_formula(P, Z, PS).

trans_state_formula(holds(F,Z), Z, P) :- !,
    \+ (\+ prim_fluent(F) ),
    trans_fluent_f2g(F,Fp,Value),
    ((\+ func_fluent(F)) ->
        P=Fp
    ; % else (is a functional fluent)
        P=(Fp=Value)
    )
.

```



```

% substitute variable Var by the term Name in T1, result is T2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% subv(X1, X2, T1, T2)
%
% substitute X1 by X2 in T1, result is T2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

subst_var(Var, Name, T1, T2) :-
    var(T1), Var==T1, !, T2 = Name.
subst_var(Var, Name, T1, T2) :-
    var(T1), \+ Var==T1, !, T2 = T1.
subst_var(Var, Name, T1, T2) :-
    T1 =..[F|L1], subst_var_1(Var, Name, L1, L2), T2 =..[F|L2].

subst_var_1(_, _, [], []).
subst_var_1(X1, X2, [T1|L1], [T2|L2]) :-
    subst_var(X1, X2, T1, T2), subst_var_1(X1, X2, L1, L2).

subv(_, _, T1, T2) :-
    var(T1), !, T2 = T1.
subv(X1, X2, T1, T2) :-
    T1 = X1, !, T2 = X2.
subv(X1, X2, T1, T2) :-
    T1 =..[F|L1], subvl(X1, X2, L1, L2), T2 =..[F|L2].

subvl(_, _, [], []).
subvl(X1, X2, [T1|L1], [T2|L2]) :-
    subv(X1, X2, T1, T2), subvl(X1, X2, L1, L2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% print terms to outputStream
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

my_write_term(T) :- outputStream(OS), my_write_term(OS,T).
my_write_term(OS,T) :-
    write_term(OS,T, [variables(full), depth(full), as(term),
        attributes(full), depth(full)]).

my_write_clause(T) :- outputStream(OS), my_write_clause(OS,T).
my_write_clause(OS,T) :-
    write_term(OS,T, [variables(full), depth(full), as(clause),
        attributes(full), depth(full)]),
    write(OS,','), nl(OS).

```



# Appendix C

## Example Golog Domain

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Mail delivery robot for Golog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Actions
%
% prim_action(Action)
% Action is a primitive action and it can be executed
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% pickup package for Room in mailbag Bag
prim_action(pickup(Bag,Room)).

% deliver package in mailbag Bag
prim_action(deliver(Bag)).

% go in direction Direction (up or down)
prim_action(go(Direction)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Fluents
%
% prim_fluent(Fluent): Fluent is a primitive fluent
% rel_fluent(Fluent): Fluent is a (primitive) relational fluent
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% the robots current position (a room number)
prim_fluent(at).

% the content of mailbag Bag
```

```

% (either the destination room of a package or "nothing")
prim_fluent(carries(Bag)) :- mailbag(Bag).

% a request to bring a package from room FromRoom to ToRoom
rel_fluent(request(FromRoom, ToRoom)) :-
    room(FromRoom), room(ToRoom).

% mailbag Bag is empty
rel_fluent(empty(Bag)) :- mailbag(Bag).

% all relational fluents are also primitive fluents
prim_fluent(X) :- rel_fluent(X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Auxiliary axioms
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

room(Room) :- member(Room, [1,2,3,4,5,6]).
mailbag(Bag) :- member(Bag, [1,2,3]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Causal laws
%
% causes_val(Action, Fluent, Value, Cond)
% When Cond holds, doing Action causes Fluent to have Value
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

causes_val(go(Direction), at, NextRoom,
    and(
        PrevRoom=at
        , % AND
        or(
            and(Direction=up, NextRoom is PrevRoom+1)
            , % OR
            and(Direction=down, NextRoom is PrevRoom-1)
        )
    )
).

causes_val(pickup(Bag,Room), carries(Bag), Room, true).
causes_val(pickup(Bag,_), empty(Bag), false, true).
causes_val(pickup(_,Room), request(FromRoom, Room), false,
    FromRoom=at).

causes_val(deliver(Bag), carries(Bag), nothing, true).
causes_val(deliver(Bag), empty(Bag), true, true).

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Preconditions of prim actions
%
% poss(Action, Condition)
%   When Condition is true, Action is possible
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

poss(go(Direction),
     some(prev_room,
          and(
            prev_room=at
          , % AND
            some(next_room,
                 and( (
                    or( and( Direction=up , next_room is prev_room+1 )
                      , % OR
                    and( Direction=down , next_room is prev_room-1 ) )
                )
            , % AND
            room(next_room) )
          )
        )
      )
    ).

poss(pickup(Bag,Room), and( request(at, Room) , empty(Bag) ) ).

poss(deliver(Bag), carries(Bag) = at).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initial state
%
% initially(Fluent, Value)
%   Fluent has Value at s0 (the initial situation)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% in the initial situation the robot is at room 1,
% carries nothing and there are 4 requests

initially(at,1).
initially(empty(Bag),true).

initially(request(FromRoom, ToRoom), true) :-
    member((FromRoom,ToRoom),

```

```
[(1,2), (1,3), (1,5), (2,1), (2,3), (2,4), (2,5), (3,1), (3,2),  
 (3,4), (3,5), (4,2), (4,5), (5,1), (5,2), (5,3), (5,4), (5,6),  
 (6,1), (6,3), (6,4)]).
```

```
initially(request(FromRoom, ToRoom), false) :-  
 \+ initially(request(FromRoom, ToRoom), true).
```

```
initially(carries(Bag), nothing).
```

## Appendix D

# Example Flux Domain

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Mail delivery robot for Flux
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Actions
%
% prim_action(Action)
% Action is a primitive action and it can be executed
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% pickup package for Room in mailbag Bag
prim_action(pickup(Bag,Room)).

% deliver package in mailbag Bag
prim_action(deliver(Bag)).

% go in direction Direction (up or down)
prim_action(go(Direction)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Fluents
%
% prim_fluent(Fluent): Fluent is a primitive fluent
% func_fluent(Fluent):
%     Fluent is a (primitive) functional fluent, i.e. the last
%     argument serves as value
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Pos is the robots current position (a room number)
func_fluent(at(Pos)).
```

```

% Content is the content of mailbag Bag
% (either the destination room of a package or "nothing")
func_fluent(carries(Bag, Content)) :- mailbag(Bag).

% a request to bring a package from room FromRoom to ToRoom
prim_fluent(request(FromRoom, ToRoom)) :-
    room(FromRoom), room(ToRoom).

% mailbag Bag is empty
prim_fluent(empty(Bag)) :- mailbag(Bag).

% all functional fluents are also primitive fluents
prim_fluent(X) :- func_fluent(X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Auxiliary axioms
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

room(Room) :- member(Room, [1,2,3,4,5,6]).
mailbag(Bag) :- member(Bag, [1,2,3]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% State update axioms
%
% state_update(Z1, A, Z2)
%     executing action A in state Z1 results in state Z2
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

state_update(Z1, go(Direction), Z2) :-
    holds(at(PrevRoom), Z1),
    (
        Direction=down, NextRoom is PrevRoom-1
        ;
        Direction=up, NextRoom is PrevRoom+1
    ),
    update(Z1, [at(NextRoom)], [at(PrevRoom)], Z2)
.

state_update(Z1, pickup(Bag, Room), Z2) :-
    holds(at(ThisRoom), Z1),
    update(Z1, [carries(Bag, Room)],
        [empty(Bag), request(ThisRoom, Room), carries(Bag, nothing)],
        Z2).

state_update(Z1, deliver(Bag), Z2) :-

```

```

holds(carries(Bag,Room), Z1),
update(Z1, [empty(Bag), carries(Bag, nothing)],
       [carries(Bag, Room)], Z2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Preconditions of prim actions
%
% poss(Action, State)
%   if true Action is possible in State
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

poss(go(Direction), Z) :-
  holds(at(PrevRoom),Z),
  ( Direction=up ->
    NextRoom is PrevRoom+1
  ; NextRoom is PrevRoom-1 ),
  room(NextRoom).

poss(pickup(Bag,Room), Z) :-
  holds(at(ThisRoom),Z), holds(request(ThisRoom, Room), Z),
  holds(empty(Bag),Z).

poss(deliver(Bag), Z) :-
  holds(at(ThisRoom),Z), holds(carries(Bag, ThisRoom), Z).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initial state
%
% init(Z0): Z0 is a finite list of ground fluents that hold\
%           in the initial situation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% in the initial situation the robot is at room 1,
% carries nothing and there are 4 requests

init(Z0) :- Z0 = [at(1), empty(1), empty(2), empty(3),
  request(1,2), request(1,3), request(1,5), request(2,1),
  request(2,3), request(2,4), request(2,5), request(3,1),
  request(3,2), request(3,4), request(3,5), request(4,2),
  request(4,5), request(5,1), request(5,2), request(5,3),
  request(5,4), request(5,6), request(6,1), request(6,3),
  request(6,4),
  carries(1, nothing), carries(2, nothing), carries(3, nothing)].

```



# Bibliography

- [GL99] Giuseppe De Giacomo and Hector 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.
- [GL00] Giuseppe De Giacomo and Hector Levesque. ConGolog, a concurrent programming language based on the situation calculus. 121(1–2):109–169, 2000.
- [GLL<sup>+</sup>04] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. 153(1–2):49–104, 2004.
- [Höl01] Steffen Hölldobler. *Logik und Logikprogrammierung*. Kolleg Synchron, 2001.
- [Leg03] Legolog website, 2003.  
<http://www.cs.toronto.edu/cogrobo/Legolog/>.
- [LRL<sup>+</sup>97] Hector Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard Scherl. GOLOG: A logic programming language for dynamic domains. 31(1–3):59–83, 1997.
- [MH69] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.
- [Rei01a] Ray Reiter. On knowledge-based programming with sensing in the situation calculus. 2(4):433–457, 2001.
- [Rei01b] Raymond Reiter. *Knowledge in Action*. MIT Press, 2001.
- [Sch04] Stephan Schiffel. Development of a fluent calculus semantics for golog programs. Großer Beleg, Dresden University of Technology, 2004.
- [Sha99] Murray Shanahan. The event calculus explained. 1999. URL: <http://www.ida.liu.se/ext/emtek/post/1999/02/paper.ps>.

- [Thi99] Michael Thielscher. From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. 111(1–2):277–299, 1999.
- [Thi04] Michael Thielscher. *The Art and Science of Programming Reasoning Robotic Agents*. 2004.
- [Thi05] Michael Thielscher. FLUX: A logic programming method for reasoning agents. 2005. Available at: [www.fluxagent.org](http://www.fluxagent.org).

## **Selbständigkeitserklärung**

*Ich erkläre, daß ich die vorliegende Arbeit selbständig, unter Angabe aller Zitate und unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.*

Dresden, den 09. März 2005

Stephan Schiffel