Reasoning about Complex Actions*

Steffen Hölldobler and Hans–Peter Störr
Artificial Intelligence Institute
Dresden University of Technology
D–01062 Dresden
Germany
Email: {sh,haps}@inf.tu-dresden.de

Abstract

Many problems addressed within the field of Cognitive Robotics and related areas can only be solved by complex plans including conditional and recursive actions as well as non–deterministic choice operators. In this paper we present a planning language which allows for the specification of complex plans. We define its semantics and give a provably complete and correct completed equational logic program with an unification complete equational theory. The approach is independent of the representation of states; they may be sets of propositional fluents as in the situation calculus or multisets of resources as in the fluent calculus. Finally, we give an instantiation within the fluent calculus.

1 Introduction

Imagine an autonomous agent performing a task in the real world. Its performance is based on an internal plan. From a low level point of view, the world is its own representation and the actions of the aforementioned plan are simple commands controlling the effectors of the agent. At a higher level of abstraction, the world is internally represented by states and (primitive) actions are transformations on the space of states. In this paper we do not want to discuss how actions on the higher level lead to commands on the lower level or whether the abstract level is needed or not, although these are very interesting and active open research problems. We also do not want to deal with another burning question of how the agent got hold of its

*A version of this paper was presented on the Action and Causality track of the NM'98 Workshop.

plan. The plan may be given to it by a programmer, it may have (semi-)automatically generated the plan from the initial state, the goal state and the descriptions of the primitive actions it is able to perform or it may have learned it from examples. For the purpose of this paper we just assume that a plan at the abstract level is given.

But how should such a plan look like? In most studies performed so far, such a plan is simply a sequence of so–called primitive actions, which transform one state into another. For example, in the blocks world a simple plan may be used to stack all blocks into a tower. In such a plan the sequence in which the blocks are stacked is often fixed. E.g., if block a, b and c are on the table, then the plan "stack block b on top of a and stack block c on top of b" leads to the desired tower. This plan, however, cannot be used if there is a different number of blocks on the table. Moreover, the plan is unnecessarily precise, because the sequence of blocks is not important if we are just interested in building a tower of all blocks. Intuitively, we would like to add another layer of abstraction. We would like to abstract from the sequence of blocks used and from the number of available blocks. However, this involves more complex actions, viz. non–deterministic choice operators and recursive actions.

More formally, the question addressed in this paper is whether the execution of a given plan p in a given initial situation s yields a given goal g. Thereby the fact that the execution of p in s leads to the fulfillment of g should be a logical consequence of an appropriate axiomatization of situations, actions and causality (see [10, 11]). This problem was extensively studied for *simple* plans consisting of sequences or partial orderings of primitive actions (e.g. [12, 2]). However less attention was given to *complex* plans including conditional and recursive actions as well as non–deterministic choice operators (see Section "Relation to Other Approaches".).

This paper is organized as follows. First, we demonstrate the need for complex plans by a simple example, the omelette baking problem. Then, we present the planning language used in this paper as well as its semantics. Thereafter, we discuss and formally define the notions of executability, correctness and termination of plans. A formalization of complex plans by a completed equational logic program with an unification complete equational theory is given. Up to this point the whole approach is developed independently of a concrete representation of states. We instantiate our approach by specifying states as multisets of fluents. This leads to a formalization of reasoning about complex plans in the fluent calculus [3]. Finally, we relate our solution to other approaches known from the literature and discuss some open problems and outline future research.

2 The Omelette Baking Problem

A cook has a finite supply of eggs, at least one of which is good, and a saucer. He can break an egg into the saucer, can smell at the saucer to determine whether the egg is good and can empty the saucer into the sink. His goal is to have a good egg in the saucer and nothing else. This is simplified version of a problem presented in [6].

For several reasons a simple plan cannot solve this problem:

- The action of breaking the egg is non–deterministic in that the cook does not know in advance whether the egg is good, and consequently has to react on the outcome of the action.
- The cook does not know in advance how may eggs he has to break before the first good one is in the saucer. In other words, he does not know in advance how many break and throw away actions he has to perform.

However, a complex plan can give an apparently straightforward solution:

$$\begin{aligned} &\mathit{egg2saucer} \\ &\qquad \text{where} \\ &\mathit{egg2saucer} \ \Leftarrow\ \mathit{break}\,;\ \mathit{if}(\mathit{badegg}\,,\ \mathit{emptysc}\,;\mathit{egg2saucer}\,,\ \varepsilon). \end{aligned} \tag{1}$$

Intuitively, the action $\mathit{egg2saucer}$ is a recursive procedure, whose body consists of the simple action break, representing the breaking of an egg, followed by a conditional action. Within the condition badegg the egg is tested to see if is bad and depending on the outcome either the saucer is emptied ($\mathit{emptysc}$) followed by a recursive call to $\mathit{egg2saucer}$ or the procedure terminates with the empty plan ε .

3 A Complex Planning Language

We now give a formal description of syntax and semantics of a language for complex plans which allows for conditional execution of plans, recursion and the choice of objects as parameters for actions, while at the same time minimizing complexity.

We intend to formalize reasoning about complex plans in a model, where *states* are snapshots of the universe at a given moment and *actions* are the only means to change from one state to another (see [10, 11]). For the purpose of this paper we abstract from a specific representation of states.

For example, states could be sets of propositional fluents as in the situation calculus or multisets of resources as in the fluent calculus.

Let $\mathcal{S}$ be the set of states, $\mathcal{A}$ be a set of parameterized action names denoting *primitive actions*, $\mathcal{C}$ a set of parameterized condition names denoting *Conditions* and $\mathcal{H}$ be a set of parameterized procedure heads. Now we are ready to introduce the set $\mathcal{P}$ of *complex plans*, whose elements p are defined by the following expression in BNF:

$p = \varepsilon \mid a; p \mid \mathit{if}(c, p, p); p \mid h; p$

where	$;$	is the concatenation operator
	ε	is the empty plan,
	a	is a primitive action,
	if	is a conditional,
	c	is a condition and
	h	is the head of a procedure definition.

A complex plan may be augmented by *procedures*, which are described by expressions of the form

$h \stackrel{c}{\Leftarrow} p$

where	h	is the head of a procedure definition,
	c	is a condition and
	p	is a complex plan.

Let $\mathcal{D}$ denote the set of procedure definitions an agent uses for the description of his plan.

One should observe that the last action in each plan is ε. For notational convenience we often omit this occurrence of ε. We also omit the condition c in a procedure definition if c denotes a tautology.

The meaning of a primitive action $a \in \mathcal{A}$ is given by a set of triples of the form $\langle s, a, s' \rangle$, where $\{s, s'\} \subseteq \mathcal{S}$. The set of such triples is denoted by $\mathcal{E}$. Intuitively, $\langle s, a, s' \rangle$ denotes that action a transforms state s into state s'.

Similarly, the meaning of the conditions is given by a set $\mathcal{V}$ of pairs $\langle c, s \rangle$, where $s \in \mathcal{S}$, $c \in \mathcal{C}$. $\langle c, s \rangle$ denotes that condition c holds in state s.

The stepwise execution of a plan p in a state s is defined by the following transformation $\leadsto : \mathcal{S} \times \mathcal{P} \to \mathcal{S} \times \mathcal{P}$:

$$
\begin{array}{ll}
(s, a; p) \leadsto (s', p) & \text{if } \langle s, a, s' \rangle \in \mathcal{E}, \\
(s, \mathit{if}(c, p_1, p_2); p) \leadsto (s, \mathit{app}(p_1, p)) & \text{if } \langle c, s \rangle \in \mathcal{V}, \\
(s, \mathit{if}(c, p_1, p_2); p) \leadsto (s, \mathit{app}(p_2, p)) & \text{if } \langle c, s \rangle \notin \mathcal{V}, \\
(s, h; p) \leadsto (s, \mathit{app}(p'\sigma, p)) & \text{if } \exists h', \sigma \ : \ h' \overset{c}{\Leftarrow} p' \in \mathcal{D} \wedge \\
 & \qquad h = h'\sigma \wedge \langle c\sigma, s \rangle \in \mathcal{V},
\end{array}
$$

where app appends complex plans and σ denotes a substitution.

One should observe that the definition of $\leadsto$ allows several forms of non–determinism.

- $\mathcal{E}$ may contain triples of the form $\langle s, a, s_1 \rangle$ and $\langle s, a, s_2 \rangle$, where $s_1 \neq s_2$. E.g., this allows the specification of a break action which requires an egg as a condition and yields non–deterministically either a good or a bad egg in the saucer.

- In the execution of a procedure the condition may be used as a non–deterministic choice operator, which chooses one object out of all objects satisfying this condition.[1] For example, consider the procedure

$$\mathit{fillbox}(x) \overset{\mathrm{ontable(y)}}{\Leftarrow} \mathit{putin}(y, x);\ \mathit{if}(\mathit{full}(x), \varepsilon, \mathit{fillbox}(x)) \ ,$$

  used to fill a box x with blocks y standing on a table. While executing the plan $\mathit{fillbox}(a)$ in a state where the blocks $b_1, \ldots, b_n$ are on the table, the sequence in which the blocks are selected is not determined, since the substitution σ used in the definition of $\leadsto$ may be any substitution which binds x to a and y to some block b_i, $i \in \{1, \ldots, n\}$, on the table, thus satisfying the choice operator $\mathit{ontable}(y)$.

Because of these non–determinisms the execution of a complex plan can sometimes take one of several execution paths. Formally, these paths are represented by *derivations* wrt $\leadsto$. Derivations and their length are defined as usual. A derivation $((s_i, p_i))_{i=0}^{n}$ wrt $\leadsto$ *terminates* if $p_n = \varepsilon$. It *fails* if $p_n \neq \varepsilon$ and there is no (s_{n+1}, p_{n+1}) such that $(s_n, p_n) \leadsto (s_{n+1}, p_{n+1})$.

[1] This is similar to the $\pi(x)\sigma$ construct of Golog [7].

4 Executability, Correctness and Termination of Plans

Returning to the initial question posed in the introduction we consider an initial state s_0 and a complex plan p_0. Furthermore, let g be a predicate over states which represents the conditions that should be satisfied in a goal state. Several questions arise immediately:

- When is p_0 executable in s_0?
- Does p_0 terminate if it is executed in s_0?
- Is p_0 correct?

A short reflection reveals that the questions are ill–posed without further considerations. What precisely is meant by executability, termination and correctness of plans? Since complex plans may contain non–deterministic actions and choice operators, the precise definition of these notions depends on the risks an agent is willing to take.

From the viewpoint of a sceptical agent who takes no risks, all possible derivations $((s_i, p_i))_{i=0}^n$ are to be considered. A plan p_0 is *executable* in state s_0 iff no derivation $((s_i, p_i))_{i=0}^n$ wrt $\rightsquigarrow$ fails. It *terminates* iff there is an upper bound for the length of possible derivations. It is *correct* wrt the condition g iff for all terminating derivations $((s_i, p_i))_{i=0}^n$ the condition g holds in s_n.

To express these properties we introduce a relation $* \overset{*}{\underset{*}{\longrightarrow}} *$ on $\mathcal{S} \times \mathcal{P} \times \mathbb{N} \times (\mathcal{S} \cup \{\perp\})$ which describes the complete execution of the program under a limited number of steps. It expresses successful execution:

$s_0 \overset{p_0}{\underset{m}{\longrightarrow}} s_n$ with $s_n \in \mathcal{S}$ holds iff there is a terminating derivation $((s_i, p_i))_{i=0}^n$ of length $n \leq m$,

and it expresses failure as well:

$s_0 \overset{p_0}{\underset{m}{\longrightarrow}} \perp$ holds iff there exists a derivation $((s_i, p_i))_{i=0}^n$ which either fails or is of length $n > m$.

Using this relation, we establish the following proposition:

Proposition 1. *A complex plan $p_0 \in \mathcal{P}$ is executable in a state $s_0 \in \mathcal{S}$, terminates and is correct wrt. the condition g iff*

$$\exists m \in \mathbb{N} \;\; \forall s_g \in (\mathcal{S} \cup \{\perp\}) \;\; \left[s_0 \overset{p_0}{\underset{m}{\longrightarrow}} s_g \quad \rightarrow \quad s_g \neq \perp \wedge g(s_g) \right].$$

(The proof is straightforward by case analysis.)

5 A Logical Formalization of Complex Plans

Let $\mathcal{T}$ be a representation of states as terms and $\cdot^{\mathcal{T}}$ be a mapping from $\mathcal{S}$ to $\mathcal{T}$ assigning to each state its term representation.[2] Furthermore, assume that conditions are represented as terms and let the predicate $\mathit{holds}(c, s)$ denote the fact that the condition c holds in the state s and let $\mathit{goalstate}(s)$ represent the goal g of the planning problem. Let Δ_D be a set of first order formulas such that the following holds:

$$\begin{array}{lll}
\Delta_D \models s_1^{\mathcal{T}} = s_2^{\mathcal{T}} & \text{iff} & s_1 = s_2 \\
\Delta_D \models \mathit{causes}(s_1^{\mathcal{T}}, a, s_2^{\mathcal{T}}) & \text{iff} & \langle s_1, a, s_2 \rangle \in \mathcal{E} \\
\Delta_D \models \mathit{holds}(c, s^{\mathcal{T}}) & \text{iff} & \langle c, s \rangle \in \mathcal{V} \\
\Delta_D \models \mathit{goalstate}(s^{\mathcal{T}}) & \text{iff} & g(s) \text{ holds.}
\end{array}$$

In addition, Δ_D is assumed to be consistent and complete, i.e.,

$$\begin{array}{lll}
\Delta_D \not\models \mathit{causes}(s_1^{\mathcal{T}}, a, s_2^{\mathcal{T}}) & \text{iff} & \Delta_D \models \neg \mathit{causes}(s_1^{\mathcal{T}}, a, s_2^{\mathcal{T}}) \\
\Delta_D \not\models \mathit{holds}(c, s^{\mathcal{T}}) & \text{iff} & \Delta_D \models \neg \mathit{holds}(c, s^{\mathcal{T}}) \\
\Delta_D \not\models \mathit{goalstate}(s^{\mathcal{T}}) & \text{iff} & \Delta_D \models \neg \mathit{goalstate}(s^{\mathcal{T}}).
\end{array}$$

Δ_D should contain an unification complete equational theory[3] , thus

$$\Delta_D \not\models s_1^{\mathcal{T}} = s_2^{\mathcal{T}} \quad \text{iff} \quad \Delta_D \models s_1^{\mathcal{T}} \neq s_2^{\mathcal{T}} \; .$$

Furthermore, Δ_D should not contain any of the predicate symbols *app* , *proc* , *world* , *continuable* , *doplan* , *planfails* , and *natnum* , which will be used in the following to model the execution of plans.

Δ_D can be specified in detail as soon as we have committed ourselves to the representation of states. Since our planning language and its semantics is indifferent to this commitment, we have just given conditions for Δ_D . In the following section we instantiate our approach within the fluent calculus by using multisets of fluents to represent states; the mapping $\cdot^{\mathcal{T}}$ and the concrete Δ_D are given there. For the moment we concentrate on the formalization of complex plans.

Plans are concatenated with the help of a ternary predicate *app* as usual.

$$\begin{array}{l}
\mathit{app}(\varepsilon, p, p) \\
\mathit{app}(\; h;t \;, p, \; h;t' \;) \leftarrow \mathit{app}(t, p, t').
\end{array} \tag{2}$$

[2] The term representation of a state may be either a term describing the history of the present state, as in the situation calculus, or a direct representation of the state, as in the fluent calculus (see next section.)

[3] In the case of the fluent calculus (see below), an E-unification complete equational theory.

The set $\mathcal{D}$ of procedure definitions is represented by a ternary predicate *proc* :

$$proc\,(h,c,b). \quad \text{for} \quad h \stackrel{c}{\Leftarrow} b \in \mathcal{D} \tag{3}$$

The transformation $\leadsto$ is represented by the predicate *world* :

$$\begin{aligned}
&world\,(s_1,\ h;t\ , s_2, t) \leftarrow causes\,(s_1,h,s_2).\\
&world\,(s,\ if(h,p,p');t\ , s, t') \leftarrow holds\,(h,s), app\,(p,t,t').\\
&world\,(s,\ if(h,p,p');t\ , s, t') \leftarrow \neg holds\,(h,s), app\,(p',t,t').\\
&world\,(s,\ h;t\ , s, t') \leftarrow proc\,(h,c,b), holds\,(c,s), app\,(b,t,t').
\end{aligned} \tag{4}$$

The predicate *continuable* describes the possibility to continue the execution of a plan p in a state s .

$$continuable\,(s,p) \leftarrow world\,(s,p,s',p'). \tag{5}$$

Derivations are represented with the help of the predicate *doplan* . $\perp$ denotes that a plan has failed, 0 is a constant denoting the number zero and s is a unary function representing the successor function on natural numbers. The predicate $doplan\,(s_0^\tau, p_0^\tau, s^m(0), s_n^\tau)$ is true iff there exists either a terminating derivation $\big((s_i,p_i)\big)_{i=0}^{n}$ of length $n \le m$, or $s_n = \perp$ and there exists a failing derivation or a derivation of a length greater n , i.e. iff $s_0 \xrightarrow[m]{p_0} s_n$.

$$\begin{aligned}
&doplan\,(s_0,\varepsilon,n,s_0).\\
&doplan\,(s_0,p,0,\perp) \leftarrow p \neq \varepsilon.\\
&doplan\,(s_0,p_0,s(n),s_n) \leftarrow p_0 \neq \varepsilon, world\,(s_0,p_0,s_1,p_1),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad doplan\,(s_1,p_1,n,s_n).\\
&doplan\,(s_0,p_0,s(n),\perp) \leftarrow p_0 \neq \varepsilon, \neg continuable\,(s_0,p_0).
\end{aligned} \tag{6}$$

A plan fails, iff it may lead to a state that is not a goal state:

$$planfails\,(s_0,p_0,n,s_n) \leftarrow doplan\,(s_0,p_0,n,s_n), \neg goalstate\,(s_n). \tag{7}$$

The natural numbers are represented by the predicate *natnum* supported by the unification complete equational theory contained in Δ_D : [4]

$$\begin{aligned}
&natnum\,(0).\\
&natnum\,(s(n)) \leftarrow natnum\,(n).
\end{aligned} \tag{8}$$

[4] Induction axioms are not necessary for our purpose; more comments on that are given in the discussion.

Now let

$$\Delta = \Delta_D \cup Comp(\{(2),(3),(4),(6),(7),(8)\})$$

be our axiomatization of complex plans, where *Comp* denotes the Clark completion (see e.g.[8]). The following proposition states that Δ is correct and complete. Recall that g is a predicate over states denoting the condition to be satisfied in the final state.

Proposition 2. *p_0 is executable in s_0, terminates and is correct wrt g iff there exists an $n \in I\!N$ such that*

$$\Delta \models \forall s_g \ [doplan(s_0^\tau, t_0^\tau, s^n(0), s_g) \rightarrow s_g \neq \perp \wedge g(s_g)] \ .$$

Sketch of proof. [5] First, one can be prove by induction over the structure of plans that the predicate *app* models the concatenation of plans. Second, $proc(h, c, b)$ is implied by Δ iff it represents ground instantiations of procedures $h \stackrel{c}{\Leftarrow} b$ in $\mathcal{D}$. Based on these lemmata one can prove by case analysis that *world* represents the execution relation $\leadsto$:

$$\forall s_1, s_2 \in \mathcal{S}, p_1, p_2 \in \mathcal{P}$$
$$(s_1, p_1) \leadsto (s_2, p_2) \text{ iff } \Delta \models world(s_1^\tau, p_1, s_2^\tau, p_2).$$

Induction over n shows that *doplan* represents the execution relation $* \xrightarrow[*]{*} *$:

$$\forall s_0 \in \mathcal{S}, p_0 \in \mathcal{P}, s_g \in (\mathcal{S} \cup \{\perp\})$$
$$s_0 \xrightarrow[n]{p_0} s_g \quad \text{iff} \quad \Delta \models doplan(s_0^\tau, p, s^n(0), s_g^\tau).$$

Considering proposition 1 at this point leads to the proof of our proposition. $\Box$

In order to reason about plans we use the predicate *planfails* as in the following theorem:

Theorem 3. *p_0 is executable in s_0, terminates and is correct wrt the goal condition encoded by the predicate goalstate iff*

$$\Delta \models \exists n \ natnum(n) \wedge \neg planfails(s_0^\tau, p_0^\tau, n).$$

(The proof is straightforward by case analysis.)

[5] The full proof can be obtained along the lines presented in in [15].

6 A Fluent Calculus Formalization of the Omelette Problem

The formalization presented in the previous section is independent of the representation of states and actions. In order to completely solve the omelette baking problem, we have to choose an appropriate representation. We opt for the fluent calculus [3], a purely first order formalization, in which the frame problem is solved without the need to state any frame axioms or laws of inertia.

Within the fluent calculus, a state is a multiset of fluents. Fluents are represented by first order terms $f_1, \ldots, f_n$, which contain no occurrence of the symbols $\circ$ and 1. Hence, a state is a multiset of the form $\dot{\{}\, f_1, \ldots, f_n \,\dot{\}}$.[6] Multisets of fluents are represented with the help of a binary function symbol $\circ$ which is written infix and satisfies the equational theory AC1, which consists of the equations

$$\begin{array}{ll} x \circ (y \circ z) = (x \circ y) \circ z & \text{(associativity)} \\ x \circ y = y \circ x & \text{(commutativity)} \\ x \circ 1 = x & \text{(unit element)} \end{array}$$

together with the axioms of equality. In order to reason about negated equalities, this theory is turned into a so–called unification complete theory $\mathrm{AC1}^*$. $\mathrm{AC1}^*$ is then built into the unification computation and SLDENF–resolution can be used to determine whether a query follows from a normal logic program (see [4]). There is a straightforward mapping $\cdot^\tau$ from multisets of fluents to their corresponding term representations. Let $\mathcal{M}$ be a multiset of terms, then

$$\mathcal{M}^\tau = \begin{cases} 1 & \text{if } \mathcal{M} = \dot{\emptyset}\,, \\ f \circ \mathcal{M}_1^\tau & \text{if } \mathcal{M} = \dot{\{} f \dot{\}} \,\dot{\cup}\, \mathcal{M}_1\,. \end{cases}$$

Actions are represented in the fluent calculus with the help of a frame assumptions. Consequently, each triple in $\mathcal{E}$ is represented by a ternary predicate *action*, whose arguments encode the condition, the name and the effect of the action. Condition and effect are multisets of those fluents, which are affected by the action. For example, in the omelette baking problem the

[6] The symbols used to denote sets and the usual operations on sets are also used here to denote multisets and the operations defined on multisets, but we stack a $\cdot$ on top of them.

actions are represented by

$$\begin{aligned}
&\mathit{action}\,(\text{☺}, \mathit{break}\,, \text{◡}).\\
&\mathit{action}\,(\text{☹}, \mathit{break}\,, \text{♨}).\\
&\mathit{action}\,(\text{♨}, \mathit{emptysc}\,, \varepsilon).\\
&\mathit{action}\,(\text{◡}, \mathit{emptysc}\,, \varepsilon).
\end{aligned} \tag{9}$$

where ☺ and ☹ represent a good and a bad egg in the supply, and ◡ and ♨ represent a good and a bad egg in the saucer respectively.

The predicates *causes* and *holds* can now be represented by the clauses

$$\mathit{causes}\,(c \circ r, a, e \circ r) \leftarrow \mathit{action}\,(c, a, e). \tag{10}$$

and

$$\begin{aligned}
&\mathit{holds}\,(\top, s).\\
&\mathit{holds}\,(\mathit{badegg}\,, \text{☹} \circ s).
\end{aligned} \tag{11}$$

respectively, where the constant $\top$ represents the truth value "true".

The procedure *egg2saucer*, as defined in (1), is represented by the following clause:

$$\mathit{proc}\,(\mathit{egg2saucer}\,, \top, \mathit{break}\,;\ \mathit{if}(\mathit{badegg}\,,\ \mathit{emptysc}\,; \mathit{egg2saucer}\,,\ \varepsilon)). \tag{12}$$

To represent the goal we use the auxiliary predicate $\mathit{ac1match}\,(m, s)$, which is true iff s contains m:

$$\mathit{ac1match}\,(m, s) \leftarrow s = m \circ r. \tag{13}$$

The actual goal is that there is a good egg in the saucer, and no bad one:

$$\mathit{goalstate}\,(s) \leftarrow \mathit{ac1match}\,(\text{◡}, s), \neg \mathit{ac1match}\,(\text{♨}, s). \tag{14}$$

Now let

$$\Gamma = \mathrm{AC1}^{*} \cup \mathit{Comp}(\{(2), (3), (4), (6), (7), (8), (9), (10), (11), (12), (13), (14)\})$$

be our fluent calculus formalization of the omelette baking problem. It is straightforward to verify that

$$\forall s\ \big[\mathit{doplan}\,(\text{☹} \circ \text{☹} \circ \text{☺} \circ \text{☹}, \mathit{egg2saucer}\,, s^{16}(0), s) \\ \rightarrow \\ s \neq \top \wedge \exists s'[s = \text{◡} \circ s'] \wedge \neg \exists s''[s = \text{♨} \circ s'']\big]$$

is a logical consequence of Γ, and thus

$$\exists n\ \mathit{natnum}(n) \wedge \neg \mathit{planfails}(\frownie \circ \frownie \circ \smiley \circ \frownie, \mathit{egg2saucer}, n).$$

as well. (An upper bound for the number of steps in the derivation can easily be given as the number of available eggs times the number of steps required for one execution of *egg2saucer*).

There is a straightforward implementation of Γ in Prolog by computing with the clause form of the definitions and using SLDENF–resolution. In fact, by checking carefully the cases, where equational reasoning is really needed, and giving a Prolog predicate for computing AC1–unifiers, SLDNF–resolution can be used. The Prolog program is given in the Appendix.

7 Relation to Other Approaches

There is a number of other approaches to the problem of reasoning about complex plans. The work of Manna and Waldinger [9] is in a sense quite uncompromising. They establish a very complex plan theory whose main purpose is the ability to represent complex plans as terms, thus enabling generation of plans from the constructive proof of a logical formula stating the existence of a plan.[7] To be able to prove termination, this approach uses first order logic augmented by well–founded induction. Unfortunately, their formalism seems to be much too complex to be of practical use.

Stephan and Biundo [13, 14] use variants of a modal logic DL for the interactive generation of recursive plans. They focus on the generation of recursive plans by refinement of domain specific plan schemata with the help of the interactive theorem prover KIV.

A close relation exists to the situation calculus based GOLOG language defined by Levesque et al. [7]. The core idea of GOLOG is to give the user a means of specifying high level programs, which is translated into a second order logic formula. A proof of this formula in conjunction with a specification of the environment yields an action sequence, which can be executed by the agent.

Our approach avoids the use of second order constructs for two reasons:

- The procedure definitions are translated into clauses of the logical formulas instead of being terms as in [9]. With this translation one

[7] A further reason for the high complexity seems to be the use of the situation calculus: a large part of the plan theory is used to bind plan terms to specific situations during the execution of a plan by introducing the extra situation argument.

loses the possibility of quantification over procedure definitions and thus it is impossible to generate plans by proving a formula stating the existence of a plan. On the other hand, this avoids of complexity in comparison to [9] and avoids the use of second order formulas as in [7] for this purpose.

- For the sceptical agent, who wants to know in advance that the plan execution terminates after n steps, it is sufficient to use the n-th approximation of the transitive closure of the $\leadsto$ relation describing the stepwise execution of the plan. With the use of a unification complete equational theory this is first order definable in contrast to the use of the full transitive closure. Note however that it is not possible in our approach to prove that the plan does *not* terminate if it contains a loop. This would require proving that the plan fails for all limits $n \in \mathbb{N}$ of the execution length and thus require induction.

In contrast to the above–mentioned formalisms, our approach can cope with the issues which arise out of non–determinism.

Recently, White translated GOLOG into an extended linear logic [16]. While the transformation seems to be quite straightforward, the variant of linear logic used by White is non–standard. On the other hand, there is a close relation between the fluent calculus and linear logic as formally shown in [2]. Since the fluent calculus admits a standard and well understood semantics it seems to be preferable to the corresponding fragments of linear logic.

8 Open Problems and Future Research

We intend to use the planning language presented in this paper to program autonomous robots. In order to do so we have to solve a variety of open problems.

Actions like *break* or *emptysc* have to be translated into control commands for the effectors of the robot. On the other hand, the effectors of such an autonomous agent should not just receive commands from a high level planning language. Many tasks such as moving along a line should be performed within a low level circuit connecting the robots sensors to its effectors. But where precisely is the borderline between a so–called reflexive and a so–called deliberative behavior and how precisely do the two levels interact? Here we expect that we can learn from discrete control theory if we discover connections between this theory and reasoning about situations, actions and causality.

In a real environment the execution of an action may not lead to the expected effect. Constant monitoring based on an agent's sensors is required to determine states in which the plan must be modified or where replanning is needed. What precisely should a robot do if it observes an unexpected state change in the middle of a plan execution? First solutions to this problem were presented in [1]. The problem itself is not trivial and good solutions depend very likely on specific domains since from a theoretical point of view replanning is as complex as planning in the worst case. Moreover, what should the robot do if the unexpected change of the state involves some sort of concept the robot is unaware of. For example, the robot may be trained to behave in the blocks world. While moving around a block, the block slips out of the robot's arm, hits the ground and breaks into three parts. What should the robot do if it has never learned anything about broken blocks?

An approach to this problem can be found in [5] which proposes an architecture for detecting the "unusualness" of an event on the basis of distinctions between model-based anticipations and actual reality. Although this work only touches on the problem of concept creation, it does provides a mechanism which enables a robot to detect that none of its current concepts are sufficient and it therefore needs to discover new ones.

Related to the problem of unexpected changes ("miracles") is the problem of conceivable but improbable outcomes ("surprises"). The concept of a sceptical agent used in this paper seems too strong in a real world setting. If it was not known in the omelette baking problem that there is at least a good egg, there would be no plan which satisfies the cook as a sceptical agent. He would have to consider in advance going to a store to buy eggs, and because it is possible that there are no eggs in the store he would have to consider going to a farm ... ad infinitum. A cook as a brave agent would just break the eggs, and if they are no good, he would start planning again.

The complex plans considered in this paper are given by the user. In future we would like to see an automated reasoning system generating or synthesizing these plans either from examples or from a given specification. The synthesis of recursive plans is still a wide open field to be addressed in the future.

The use of SLDENF poses some problems as well. The SLDENF derivation tree may contain massive redundancy in some cases. For instance if a box is to be filled with blocks on the table the derivation tree would contain branches for every sequence of blocks put into the box. But the actual sequence of blocks is unimportant for the box to be full after the execution of the plan. The use of lemmata or reduction rules in such a case might help.

9 Acknowledgements

The authors would like to thank Michael Thielscher and the anonymous referees of ESSLLI and of the NMR'98 workshop for their valuable comments on earlier versions of this paper.

References

[1] G. De Giacomo, R. Reiter, and M. Soutchanski. Execution monitoring of high–level robot programs. In *Common Sense '98, The Fourth Symposium on Logical Formalizations of Commonsense Reasoning*, pages 277–297, University of London, 1998. Queen Mary and Westfield College.

[2] G. Große, S. Hölldobler, and J. Schneeberger. Linear deductive planning. *Journal of Logic and Computation*, 6(2):233–262, 1996.

[3] S. Hölldobler and J. Schneeberger. A new deductive approach to planning. *New Generation Computing*, 8:225–244, 1990. A short version appeared in the Proceedings of the German Workshop on Artificial Intelligence, Informatik Fachberichte *216*, pages 63-73, 1989.

[4] S. Hölldobler and M. Thielscher. Computing change and specificity with equational logic programs. *Annals of Mathematics and Artificial Intelligence*, 14:99–133, 1995.

[5] Catriona M. Kennedy. A conceptual foundation for autonomous learning in unforeseen situations. Technical Report WV-98-01, Computer Science Department, Dresden University of Technology, 1998. (forthcoming).

[6] Hector J. Levesque. What is planning in the presence of sensing? Technical report, Department of Computer Science, University of Toronto, Canada, 1996. email: `hector@cs.toronto.edu`.

[7] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzehn Lin, and Richard B. Scherl. Golog: a logic programming language for dynamic domains. *Journal of Logic Programming*, 19(20):1–25, 1994.

[8] J. W. Lloyd. *Foundations of Logic Programming.* Springer, 1987.

[9] Zohar Manna and Richard Waldinger. How to clear a block: A theory of plans. *Journal of Automated Reasoning*, 4(3):343–377, 1987.

[10] J. McCarthy. Situations and actions and causal laws. Stanford Artificial Intelligence Project: Memo 2, 1963.

[11] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463 – 502. Edinburgh University Press, 1969.

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

[13] Werner Stephan and Susanne Biundo. A new logical framework for deductive planning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, page 32. 1993.

[14] Werner Stephan and Susanne Biundo. Deduction-based refinement planning. Technical Report RR-95-13, Germen Research Center for Artificial Intelligence (DFKI), {stephan,biundo}ddfki.uni-sb.de, 1995.

[15] H.-P. Störr. Bedingte und Rekursive Aktionen im Fluent–Kalkül. Master's thesis, Dresden University of Technology, 1997. (in German).

[16] Graham White. Golog and linear logic programming. Technical report, Dept. Computer Science, Queen Mary and Westfield College, University of London, January 1998. `http://www.dcs.qmw.ac.uk/~graham`.

A Appendix: A Prolog program

We give below a Prolog program for the simplified version of the omelette baking problem. In the Prolog program terms with $\circ$ as well as plans are represented by lists. The predicate `ac1_match(X,Y,Rest)` solves the AC1–equation $X \circ Rest = Y$.

A.1 Domain independent part

```
ac1_match([], T, T).
ac1_match([E|R], T, R1) :-
    multi_minus(T,E,T1), ac1_match(R,T1,R1).

multi_minus([E|R],E,R) :- !.
multi_minus([E1|R1],E,[E1|R2]) :- multi_minus(R1, E, R2).

world(S1,[A|P],S2,P) :- causes(S1,A,S2).
world(S1,[if(H,PA,PB)|P],S1,PN) :-
    holds(H,S1), append(PA,P,PN).
world(S1,[if(H,PA,PB)|P],S1,PN) :-
    \+ holds(H,S1), append(PB,P,PN).
world(S1,[H|P],S1,PN) :-
    proc(H,C,B), holds(C,S1), append(B,P,PN).

continuable(S1,P1) :- world(S1,P1,_S2,_P2).

doplan(S0,[],_M,S0).
doplan(_S0,_P0,0,fail).
doplan(S0,P0,M,SN) :- P0\=[],  M > 0, M1 is M-1,
    world(S0,P0,S1,P1), doplan(S1,P1,M1,SN).
doplan(S0,P0,M,fail) :- P0\=[], M > 0, \+ continuable(S0,P0).

planfails(S0,P0,N) :- doplan(S0,P0,N,SN), \+ goalstate(SN).

nat(0).
nat(X1) :- nat(X), X1 is X+1.
```

A.2 Domain dependent part

```
action([eg],break,[sg]).
action([eb],break,[sb]).
action([sg],emptysc,[]).
action([sb],emptysc,[]).

causes(S1,A,S2) :- action(C,A,E),
    ac1_match(C,S1,R), append(E,R,S2).

holds(true,S).
holds(badegg,S) :- ac1_match([sb],S,_).

proc(egg2saucer,true,
    [break, if(badegg,[emptysc,egg2saucer],[]) ]).

goalstate(S) :- SN \= fail,
    ac1_match([sg],S,_), \+ ac1_match([sb],S,_).
```

A.3 Query

To check that the plan *egg2saucer* solves the problem for the case of three bad eggs and one good egg the Prolog interpreter has to prove

```
:- nat(N), \+ planfails([eb,eb,eg,eb], [egg2saucer],N).
```

This yields the answer `N = 16`, i.e. the plan is executable, correct and terminates after at most 16 steps. The query

```
:- doplan([eb,eb,eg,eb],[egg2saucer],16,S).
```

yields the possibly reached final states within the limit of 16 steps:

```
S = [sg, eb, eb, eb] ;
S = [sg, eb, eb] ;
S = [sg, eb] ;
S = [sg] ;
```