The Situation Calculus and Golog

— A Tutorial —

Gerhard Lakemeyer

Dept. of Computer Science
RWTH Aachen University
Germany

2nd Hybris Workshop, Freiburg, May 27–28, 2013
1. Motivation

2. The Situation Calculus

3. Golog
Knowledge-Based Agents

Example: An Office Robot

“If there are people in the office who want coffee, go to the kitchen, pick up coffee, and serve it to everyone who wants some. Alternatively, choose some room that is not clean and clean it. Alternatively, . . . “

Want a high-level representation that accounts for . . .

- incomplete knowledge
  - some room is dirty, but do not know which
  - agent gathers new knowledge at runtime (sensing)

- intelligent decision-making
  - programmer cannot foresee every eventuality
  - agent has to reason about actions
    (preconditions+effects)
“Broadly speaking, the newly emerging field of cognitive robotics has, as its long term objectives, the provision of a uniform theoretical and implementation framework for autonomous robotic or software agents that reason, act and perceive in changing, incompletely known, unpredictable environments.”

[Ray Reiter, Knowledge in Action, 2001]

The situation calculus is such a uniform framework. Goes back to John McCarthy (1963).

Here: variant developed by Reiter and his colleagues.

The Situation Calculus in a Nutshell

- **Situations** are sequences of actions like $do(goto(Frank), S_0)$.
- **Fluents** characterizing situations: $\neg HasCoffee(Frank, S_0)$
- **Axioms about change:**

$$HasCoffee(p, do(a, s)) \equiv a = giveCoffee \lor HasCoffee(p, s)$$
The situation calculus is a **sorted, second-order language with equality**.

- There are three sorts: *situations*, *actions*, and *ordinary objects*.
- The logical connectives are $\neg$, $\land$, $\forall$.
  (We also use $\lor$, $\supset$, $\equiv$, $\exists$ as the usual abbreviations.)
- There are two function symbols of sort situation:
  - the constant $S_0$ denoting the initial situation
  - the function $do(a, s)$ denoting the situation obtained by doing action $a$ in situation $s$
- A special binary predicate $s \sqsubseteq s'$ with the intended interpretation that $s$ is a subsequence of $s'$. 
A special binary predicate $Poss(a, s)$ with the intended interpretation that action $a$ is executable in situation $s$.

Relations and functions whose last argument is a situation are called fluents. E.g., in a world where objects can be painted $Color(x, c, s)$ may mean that $x$ has the color $c$ in situation $s$.

**Notation:** In the following, free variables are implicitly assumed to be universally quantified, e.g.,

$$Poss(drop(r, x), s) \equiv Holding(r, x, s)$$ stands for $$\forall r, s, x. Poss(drop(r, x), s) \equiv Holding(r, x, s).$$
Hybris

G Lakemeyer

Motivation

The Situation Calculus

Golog

Action Preconditions (Qualifications)

Actions have preconditions:

A robot $r$ can lift object $x$ in situation $s$ iff $r$ is not holding anything, is located next to $x$, and $x$ is not too heavy:

$$\text{Poss}(\text{pickup}(r, x), s) \equiv \forall z. \neg \text{Holding}(r, z, s) \land \neg \text{Heavy}(x) \land \text{NextTo}(r, x, s).$$

$r$ can repair an object iff the object is broken and he has glue:

$$\text{Poss}(\text{repair}(r, x), s) \equiv \text{HasGlue}(r, s) \land \text{Broken}(x, s)$$

Note: Defining Poss this way makes a strong and simplifying assumption, ignoring other possible qualifications such as lifting an object is possible only if it is not glued to the ground, not slippery, etc. This is the qualification problem, which we simply ignore here.
Changes in the world are specified by **effect axioms** which describe how actions change fluents.

The effect on *Broken* when *r* drops an object:

\[ \text{Fragile}(x) \implies \text{Broken}(x, \text{do}(\text{drop}(r, x), s)). \]

*(positive effect axiom)*

Repairing an object causes the object not to be broken any more:

\[ \neg \text{Broken}(x, \text{do}(\text{repair}(r, x), s)). \]

*(negative effect axiom)*
It takes more than effect axioms to describe a changing world. One needs so-called frame axioms to specify which fluents do not change when an action is executed.

- **Positive frame axioms:** e.g. dropping an object does not change its color:
  \[
  \text{Color}(x, c, s) \supset \text{Color}(x, c, \text{do}(\text{drop}(r, y), s)).
  \]

- **Negative frame axioms:** e.g. things do not break if they are not dropped:
  \[
  \neg \text{Broken}(x, s) \land [x \neq y \lor \neg \text{Fragile}(x)] \supset \neg \text{Broken}(x, \text{do}(\text{drop}(r, y), s)).
  \]
The Frame Problem

There are far too many frame axioms: only very few actions have an effect on any given fluent. All others remain unchanged.

- The color of an object does not change by lifting objects, opening doors, turning on lights, elect a president, etc.

There are about $2 \times A \times F$ frame axioms, where $A$ is the number of actions and $F$ the number of fluents.

Why is this a problem?

- Coding that many frame axioms by hand is awkward and error prone.
- It is very difficult to deal with that many axioms when reasoning (like planning a sequence of actions).

Goal: Find a compact representation of the frame axioms and derive them automatically from the effect axioms.
Reiter’s Solution to the Frame Problem

Example: Suppose we have two positive effect axioms for \textit{Broken}:

\[
\text{Fragile}(x) \supset \text{Broken}(x, \text{do}(\text{drop}(r, x), s)),
\]
\[
\text{NextTo}(b, x, s) \supset \text{Broken}(x, \text{do}(\text{explode}(b), s)).
\]

Then these can be transformed into an equivalent sentence:

\[
[\exists r. (a = \text{drop}(r, x) \land \text{Fragile}(x)) \lor \\
\exists b. (a = \text{explode}(b) \land \text{NextTo}(b, x, s))] \supset \text{Broken}(x, \text{do}(a, s)).
\]

(1)

Similarly, the negative effect axiom

\[
\neg \text{Broken}(x, \text{do}(\text{repair}(r, y), s))
\]

can be rewritten as:

\[
\exists r. (a = \text{repair}(r, x)) \supset \neg \text{Broken}(x, \text{do}(a, s))
\]

(2)
Now we make the following completeness assumption: Axiom (1) characterizes all possible conditions when an action $a$ can break an object $x$. That means: if $\neg \text{Broken}(x, s)$ and $\text{Broken}(x, \text{do}(a, s))$ are true, then the truth value of $\text{Broken}$ can only have changed because

$$\exists r. (a = \text{drop}(r, x) \land \text{Fragile}(x))$$

or

$$\exists b. (a = \text{explode}(b) \land \text{NextTo}(b, x, s))$$

were true.

Similarly, we assume that Axiom (2) characterizes the only possibility when an action $a$ can make an object $x$ whole.
**Successor State Axioms**

The effect axioms together with the completeness assumption can be combined in a so-called successor state axiom $\text{SSA}_{Br}$ for $\text{Broken}$:

\[
\text{Broken}(x, do(a, s)) \equiv \\
\exists r. (a = \text{drop}(r, x) \land \text{Fragile}(x)) \lor \\
\exists b. (a = \text{explode}(b) \land \text{NextTo}(b, x, s)) \lor \\
(\text{Broken}(x, s) \land \neg (\exists r. a = \text{repair}(r, x))).
\]

From the SSA one can deduce all effect axioms and all frame axioms: Examples:

- $\text{SSA}_{Br} \models \text{Fragile}(x) \supset \text{Broken}(x, do(\text{drop}(r, x), s))$
- $\text{SSA}_{Br} \land \text{UNA} \models \neg \text{Broken}(x, s) \land \neg \text{Fragile}(x) \supset \\
\quad \neg \text{Broken}(x, do(\text{drop}(r, y), s)),$

where UNA is $\text{drop}(r, x) \neq \text{explode}(b)$. 

May 28, 2013
The previous example can be generalized. Suppose we have the following normal form for effect axioms:

One positive effect axiom for each fluent $F$:

$$\gamma_F^+ (\vec{x}, a, s) \supset F(\vec{x}, do(a, s))$$

One negative effect axiom for each fluent $F$:

$$\gamma_F^- (\vec{x}, a, s) \supset \neg F(\vec{x}, do(a, s))$$

Here $\gamma_F^+ (\vec{x}, a, s)$ and $\gamma_F^- (\vec{x}, a, s)$ are wffs whose free variables are $\vec{x}$, $a$, and $s$. Such a normal form can always be obtained assuming we have unique names axioms for actions. Then the SSA has the following form:

$$F(\vec{x}, do(a, s)) \equiv \gamma_F^+ (\vec{x}, a, s) \lor [F(\vec{x}, s) \land \neg \gamma_F^- (\vec{x}, a, s)]$$
Our solution yields the following axioms:

- For each fluent $F$ there is a single SSA:
  \[ F(\vec{x}, do(a, s)) \equiv \gamma_F^+(\vec{x}, a, s) \lor (F(\vec{x}, s) \land \neg \gamma_F^- (\vec{x}, a, s)) \]

- For each action $A$ there is a single precondition axiom:
  \[ Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s). \]

- Unique names axioms for all actions.

Ignoring unique names axioms, we have $F + A$ axioms, compared to $2 \times F \times A$ frame axioms!

**Note:** SSAs are fairly short in real applications.

- The length of a SSA is roughly proportional to the number of actions which have an effect on the fluent.
- In practice only very few actions have an effect on a particular fluent.
Basic Action Theories

**Action Preconditions:** $\Sigma_{\text{pre}}$

- $\text{Poss}(\text{goto}(x), s) \equiv \text{true}$
- $\text{Poss}(\text{loadCoffee}, s) \equiv \neg\text{HasCoffee}(R, s)$
- $\text{Poss}(\text{giveCoffee}(p), s) \equiv \text{HasCoffee}(R, s)$

**Successor State Axioms for Fluents:** $\Sigma_{\text{post}}$

- $\text{HasCoffee}(p, \text{do}(a, s)) \equiv$
  - $[p = R \land a = \text{loadCoffee}] \lor [a = \text{giveCoffee}(p)] \lor$
  - $[\text{HasCoffee}(p, s) \land \neg(\exists p'. p = R \land a = \text{giveCoffee}(p'))]$

- $\text{Loc}(\text{do}(a, s)) = x \equiv$
  - $[a = \text{goto}(x)] \lor [\text{Loc}(s) = x \land \neg(\exists y. y \neq x \land a = \text{goto}(y))]$

**What is true initially:** $\Sigma_0$

- $\text{Loc}(S_0) = Rm6304, \neg\text{HasCoffee}(\text{Frank}, S_0), \text{HasCoffee}(R, S_0)$.

$\Sigma = \Sigma_0 \cup \Sigma_{\text{pre}} \cup \Sigma_{\text{post}} \cup \text{UNA} \cup \text{found. axioms}$

$\Sigma \models \text{HasCoffee}(\text{Frank}, \text{do}(\text{giveCoffee}(\text{Frank}, \text{goto}(\text{Frank}, S_0))))$. 
Besides domain dependent axioms to characterize actions, we need a few domain independent axioms which specify, roughly, that situations are all and the only sequences of actions starting at \( S_0 \).

1. Situations have unique names:
   \[
   do(a_1, s_1) = do(a_2, s_2) \supset a_1 = a_2 \land s_1 = s_2
   \]

2. There are no situations other than those reachable from \( S_0 \):
   \[
   \forall P. P(S_0) \land \forall a, s. [P(s) \supset P(do(a, s))] \supset \forall s. P(s)
   \]

3. The definition of \( \sqsubseteq \) as subhistory:
   \[
   \neg s \sqsubseteq S_0
   \]
   \[
   s \sqsubseteq do(a, s') \equiv s \sqsubseteq s'
   \]
   where \( s \sqsubseteq s' \) is an abbreviation for \( s \sqsubseteq s' \lor s = s' \).
Reasoning about Action by Regression

Let us consider the subproblem $\Sigma \models G(\sigma)$,

where $\sigma = do([\alpha_1, \ldots, \alpha_n], S_0)$ is the only situation mentioned in $G$ (regressable formula).

$\text{(do([\alpha_1, \alpha_2], s) \overset{\text{def}}{=} \text{do(\alpha_2, do(\alpha_1, s))})}$

**Idea:** Replace each fluent $F(\vec{t}, \sigma)$ with $F(\vec{x}, do(a, s)) \equiv \Phi(\vec{x}, a, s)$ by $\Phi(\vec{t}, \alpha_n, do([\alpha_1, \ldots, \alpha_{n-1}], S_0))$, and continue until $S_0$ is the only situation left.

Similarly, replace $\text{Poss}(A, s)$ by its definition.

Then we only need to reason about the initial situation.
\[ G = \text{Poss}(\text{giveCoffee}(F), S_0) \land \text{HasCoffee}(F, \text{do}(\text{giveCoffee}(F), S_0)). \]

\[ \text{Poss}(\text{giveCoffee}(p), s) \equiv \text{HasCoffee}(R, s) \]
\[ \text{HasCoffee}(p, \text{do}(a, s)) \equiv \]
\[ [p = R \land a = \text{loadCoffee}] \lor [a = \text{giveCoffee}(p)] \lor \]
\[ [\text{HasCoffee}(p, s) \land \neg (\exists p'. p = R \land a = \text{giveCoffee}(p'))] \]

Let \( \mathcal{R}[G] \) denote the formula obtained by applying regression.

Then \( \mathcal{R}[G] = \text{HasCoffee}(R, s) \land \)
\[ [F = R \land \text{giveCoffee}(F) = \text{loadCoffee}] \lor \]
\[ [\text{giveCoffee}(F') = \text{giveCoffee}(F)] \lor \]
\[ [\text{HasCoffee}(F, S_0) \land \neg (\exists p'. F = R \land \text{giveCoffee}(F) = \text{giveCoffee}(p'))] \]

which simplifies to \( \mathcal{R}[G] = \text{HasCoffee}(R, S_0). \)

Note, in particular, that \( \Sigma_0 \cup \text{UNA} \models \mathcal{R}[G]. \)

This is no coincidence!
Main Result about Regression

Let $G$ be a regresseeable sentence and $D$ a basic action theory. Then

1. $R[G]$ mentions only $S_0$.
2. $\Sigma \models G \equiv R[G]$.
3. $\Sigma \models G$ iff $\Sigma_0 \cup \text{UNA} \models R[G]$.

**Note:** (3) says that, after applying the regression operator, one only needs to consider $\Sigma_0$ and, most importantly, figuring out whether $G$ follows from $\Sigma$ reduces to ordinary first-order theorem proving!
Planning in the Situation Calculus

Let $\sigma = do([a_1, \ldots, a_n], S_0)$ and $\text{executable}(\sigma)$ stand for a formula expressing that each $a_i$ is executable.

For a given basic action theory $\Sigma$ and goal $G$ planning means finding a situation $\sigma$ such that

$$\Sigma \models \text{executable}(\sigma) \land G(\sigma).$$

For a particular $\sigma$ use regression to compute the entailment.
Planning is too Expensive

- While basic action theories allow to infer arbitrary plans in principle, this often does not work in practice.
- Even optimized planners fail in general as this is an inherently hard problem.
- Perhaps not even needed: Often the user has a good idea of what needs to be done.

Idea: Tell the robot what to do if possible. Leave (small) subtasks for planning where needed.

Coffee delivery robot

*Imagine a robot serving coffee to many people. The robot can carry only one cup at a time.*
proc DeliverCoffee
    while ∃x. WantsCoffee(x) do
        πx. WantsCoffee(x); ServeCoffee(x)
    endwhile
    goto(Home)
endProc

proc ServeCoffee(x)
    if ¬HasCoffee(R) then goto(CoffeeM); loadCoffee endIf
    goto(x); giveCoffee(x)
endProc

Semantics of programs given in the situation calculus:

- $Do(\rho, s, s')$: executing program $\rho$ leads from situation $s$ to $s'$.
- Program execution: $\Sigma \models \exists s. Do(DeliverCoffee, S_0, s)$
  (finds as a side effect a sequence of executable actions)

- Implementation in Prolog.
- Efficient if the initial state consists of a set of literals.
Complex Actions

Programming in the situation calculus is introduced by defining the following complex actions:

- **Sequence:**
  First execute $A$, then $B$.

- **Test actions:**
  Is $\phi$ true in the current situation?

- **If-then-else, While-loops**

- **Nondeterministic actions:**
  Execute $A$ or $B$.

- **Nondeterministic choice of the argument of an actions:**
  e.g. $\pi x\text{pickup}(x)$: pick up an arbitrary object.

- **procedures**
  Names for complex actions, parameters, recursion.
The Meaning of Complex Actions (1)

\( \text{Do}(a, s, s') \) stands for:
The execution of action a in situation s leads to situation \( s' \).

- **Primitive actions:**
  \[
  \text{Do}(a, s, s') \triangleq \text{Poss}(a[s], s) \land s' = \text{do}(a[s], s).
  \]

- **Sequence:**
  \[
  \text{Do}([a_1; a_2], s, s') \triangleq \exists s'' \text{Do}(a_1, s, s'') \land \text{Do}(a_2, s'', s').
  \]

- **Nondeterministic actions:**
  \[
  \text{Do}([a_1|a_2], s, s') \triangleq \text{Do}(a_1, s, s') \lor \text{Do}(a_2, s, s')
  \]

- **Nondeterministic choice:**
  \[
  \text{Do}((\pi x)a(x), s, s') \triangleq \exists x \text{Do}(a(x), s, s')
  \]

- **Test actions:**
  \[
  \text{Do}(\phi?, s, s') \triangleq \phi[s] \land s = s'
  \]
The Meaning of Complex Actions (2)

- **If-then-else:**

  \[
  Do(if(\phi, a_1, a_2), s, s') \equiv Do([[\phi?]; a_1][\neg\phi?]; a_2], s, s')
  \]

- ***-operator:** execute the action \(n\) times \((n \geq 0)\):

  \[
  Do(a^*, s, s') \equiv \forall P[\forall s_1 P(s_1, s_1) \land \forall s_1, s_2, s_3 (P(s_1, s_2) \land \neg\phi ?) \subset \forall P(s_1, s_3))] \subset P(s, s')
  \]

- **While-loops:**

  \[
  Do(while(\phi, a), s, s') \equiv Do(([[\phi?]; a]^*; \neg\phi?), s, s')
  \]

- **Procedures:**

  definition is second order because of recursion (omitted).

**Note:** The semantics of complex actions is completely described within the situation calculus.
Consider the coffee-delivery program together with the basic action theory as before and the initial state

\[ \Sigma_0 = \{ \forall x. \text{WantsCoffee}(x) \equiv (x = \text{Frank}), \neg \text{HasCoffee}(R, S_0) \} \].

\[ \Sigma \models \text{Do}(\text{DeliverCoffee}, S_0, \sigma) \],

where \( \sigma = \text{do}([\text{goto}(\text{CoffeeM}), \text{loadCoffee},
\text{goto}(\text{Frank}), \text{giveCoffee}(\text{Frank}), \text{goto}(\text{Home}]), S_0) \).
Comparison to conventional imperative programming:

- Procedures in “normal” imperative languages are ultimately reduced to *machine instructions* like assignment statements.
- Procedures in Golog actions are reduced to primitive actions which refer to *actions in the real world*, like picking up objects or moving. Instead of machine states we are interested in *world states*.
- Executing complex actions requires logical reasoning about what the world is like and how it evolves. For example, in order to execute
  \[
  \text{while } [\exists x. \text{WantsCoffee}(x)] \text{ do } \ldots
  \]
one needs to evaluate the test \([\exists x. \text{WantsCoffee}(x)]\) in the current situation.
Over the past years a number of extensions to Golog have been proposed to make it applicable in realistic domains, especially in robotics. These include

- concurrent actions
- time
- continuous change
- an explicit notion of knowledge
- sensing
- stochastic actions