Reasoning about Actions in the Situation Calculus – An Introduction

Yves Lespérance

[slides adapted from Giuseppe De Giacomo]

Sapienza University of Rome, April 2024

Outline

Outline

Knowledge Representation

Aims at building systems that know about their world and are able to act in an informed way in it. The key points of KR are:

- knowledge is represented **formally**;
- **reasoning** procedures are able to extract consequences of such knowledge;
- reasoning is used to deliberate in an informed fashion about how to **act**.

Knowledge Representation

KR is actually a radical idea, developed only recently, see e.g., Hector J. Levesque: On our best behaviour. Artif. Intell. 212: 27-35 (2014). It comes after a long gestation:

- Aristotle, who developed the initial notion of logic;
- Leibniz, who brought forward a notion of "thinking as computation" ("*calculemus*" [Leibniz's The Art of Discovery (1685)]);
- Gottlob Frege, who developed the notion of symbolic logic;
- Alonzo Church, Kurt Gödel, and Alan Turing, who set the bases for bounding logic and computation, ultimately giving rise to Computer Science, though even them did not think about logic as a way of representing knowledge.
- **John McCarthy**, who finally came up with the idea of using logic for representing knowledge and reason about it, at the end of the 50's, e.g., McCarthy, J. Programs with common sense. Proceedings of the Teddington Conference on the Mechanization of Thought Processes, 756-791 (1959)

KR suggests that we should put aside any idea of tricks and shortcuts, and focus instead on what needs to be known, how to represent it symbolically, and how to use the representations.

Knowledge-based systems should be able to apply what they know in new contexts and be easy to extend, in contrast to systems where knowledge is hard-coded.

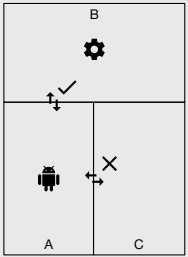
Running Example

Scenario

A robot, $self$, inhabits an environment formed by various rooms connected by doors that are open or closed. Some rooms are control rooms and contain a button to open all the doors.

Instance

- Rooms: A, B, C
- Control room: B
- Doors: d_{AB} between A and B, d_{AC} between A and C.
- Initially: robot $self$ is in room A, door d_{AB} is open, and d_{AC} is closed.



Running Example

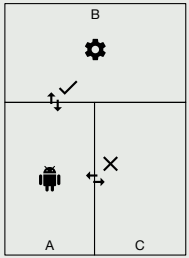
Scenario

A robot, $self$, inhabits an environment formed by various rooms connected by doors that are open or closed. Some rooms are control rooms and contain a button to open all the doors. Robot $self$ can move across rooms and when in a control room $self$ can press the button to open all doors.

Actions

Action available to robot $self$:

- $goto(x)$ where x is a room.
  - PRE: To perform this action, there must be an open door between the room where $self$ is and room x.
  - EFF: The effect is that $self$ gets to room x.
- $openAllDoors()$ to open all closed doors.
  - PRE: To perform this action, $self$ needs to be in a room that is a control room.
  - EFF: The effect is that all closed doors get opened.



Running Example

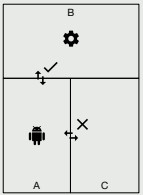
Predicates

- Static Predicates
  - $Room(x)$ denoting that x is a room;
  - $ControlRoom(x)$ denoting that the room x is a control room;
  - $Door(x, y, z)$ denoting that x is a door between room y and room z
    (note that $Door(x, y, z)$ is symmetric, that is, if $Door(x, y, z)$ holds also $Door(x, z, y)$ holds).
- Dynamic Predicates, or Fluents:
  - $Open(x)$ denotes that the door x is open;
  - $SelfIn(x)$ denotes that the $self$ is in room x.

Instance - Using Models

Let's represent the instance with a first-order logic model $\mathcal{I}$
($\mathcal{I}$ is essentially a relational database)

- Static Predicates:
  - $Room^{\mathcal{I}} = \{A, B, C\}$
  - $ControlRoom^{\mathcal{I}} = \{B\}$
  - $Door^{\mathcal{I}} = \{(d_{AB}, A, B), (d_{AB}, B, A), (d_{AC}, A, C), (d_{AC}, C, A)\}$
- Dynamic Predicates, or Fluents:
  - $Open^{\mathcal{I}} = \{d_{AB}\}$
  - $SelfIn^{\mathcal{I}} = \{A\}$



Running Example

Predicates

- Static Predicates
  - $Room(x)$ denoting that x is a room;
  - $ControlRoom(x)$ denoting that the room x is a control room;
  - $Door(x, y, z)$ denoting that x is a door between room y and room z
    (note that $Door(x, y, z)$ is symmetric, that is, if $Door(x, y, z)$ holds also $Door(x, z, y)$ holds).
- Dynamic Predicates, or Fluents:
  - $Open(x)$ denotes that the door x is open;
  - $SelfIn(x)$ denotes that the $self$ is in room x.

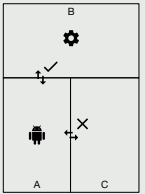
Instance - **Using Theories**

Static Predicates:

- $Room(x) \equiv (x = A \vee x = B \vee x = C)$
- $ControlRoom(x) \equiv (x = B)$
- $Door(x, y, z) \equiv ((x = d_{AB} \wedge y = A \wedge z = B) \vee (x = d_{AB} \wedge y = B \wedge z = A) \vee$
  $(x = d_{AC} \wedge y = A \wedge z = C) \vee (x = d_{AC} \wedge y = C \wedge z = A))$

Dynamic Predicates, or Fluents:

- $Open(x) \equiv (x = d_{AB})$
- $SelfIn(x) \equiv (x = A)$



Notation: for simplicity, we often omit writing the outermost universal quantification: i.e. we write $ControlRoom(x) \equiv (x = B)$ *instead of* $\forall x.(ControlRoom(x) \equiv (x = B))$.

Running Example

Actions

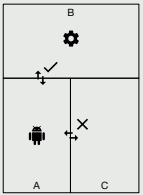
Action available to robot $self$:

- $goto(x)$ where x is a room.
  - PRE: To perform this action, there must be an open door between the room where $self$ is and x.
  - EFF: The effect is that $self$ gets to room x.
- $openAllDoors()$ to open all doors.
  - PRE: To perform this action, $self$ needs to be in a room that is a control room.
  - EFF: The effect is that all closed doors get opened.

Formalization

- $goto(x)$:
  - PRE: $\exists r.SelfIn(r) \wedge \exists d.Door(d, r, x) \wedge Open(d)$
  - EFF: $selfIn(x) \wedge \neg\exists r.r \neq x \wedge SelfIn(r)$
- $openAllDoors()$:
  - PRE: $\exists r.SelfIn(r) \wedge ControlRoom(r)$
  - EFF: $\forall d.\text{PRE}[Closed(d)] \supset Open(d)$

(Notation: $\varphi \supset \psi$ denotes φ implies ψ, i.e., $\neg\varphi \vee \psi$.)



Problem 1: we need to formally talk about two different states:
the state s (just) before the action ($\text{PRE}[Closed(d)]$) and the state s' just after it ($Open(d)$)! How?

Problem 2: Is this specification enough? How do we specify that "nothing else changes"? Frame Problem!

Running Example

Actions

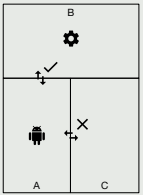
Action available to robot $self$:

- $goto(x)$ where x is a room.
  - PRE: To perform this action, there must be an open door between the room where $self$ is and x.
  - EFF: The effect is that $self$ gets to room x.
- $openAllDoors()$ to open all doors.
  - PRE: To perform this action, $self$ needs to be in a room that is a control room.
  - EFF: The effect is that all closed doors get opened.

Formalization

- $goto(x)$:
  - PRE: $\exists r.SelfIn(r) \wedge \exists d.Door(d, r, x) \wedge Open(d)$
  - EFF: $selfIn(x) \wedge \neg\exists r.r \neq x \wedge SelfIn(r)$
- $openAllDoors()$:
  - PRE: $\exists r.SelfIn(r) \wedge ControlRoom(r)$
  - EFF: $\forall d.\text{PRE}[Closed(d)] \supset Open(d)$

(Notation: $\varphi \supset \psi$ denotes φ implies ψ, i.e., $\neg\varphi \vee \psi$.)



Problem 1: we need to formally talk about two different states:
the state s (just) before the action ($\text{PRE}[Closed(d)]$) and the state s' just after it ($Open(d)$)! How?

Problem 2: Is this specification enough? How do we specify that "nothing else changes"? Frame Problem!

Running Example

Actions

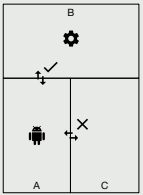
Action available to robot $self$:

- $goto(x)$ where x is a room.
  - PRE: To perform this action, there must be an open door between the room where $self$ is and x.
  - EFF: The effect is that $self$ gets to room x.
- $openAllDoors()$ to open all doors.
  - PRE: To perform this action, $self$ needs to be in a room that is a control room.
  - EFF: The effect is that all closed doors get opened.

Formalization

- $goto(x)$:
  - PRE: $\exists r.SelfIn(r) \wedge \exists d.Door(d, r, x) \wedge Open(d)$
  - EFF: $selfIn(x) \wedge \neg\exists r.r \neq x \wedge SelfIn(r)$
- $openAllDoors()$:
  - PRE: $\exists r.SelfIn(r) \wedge ControlRoom(r)$
  - EFF: $\forall d.\text{PRE}[Closed(d)] \supset Open(d)$

(Notation: $\varphi \supset \psi$ denotes φ implies ψ, i.e., $\neg\varphi \vee \psi$.)



Problem 1: we need to formally talk about two different states:
the state s (just) before the action ($\text{PRE}[Closed(d)]$) and the state s' just after it ($Open(d)$)! How?

Problem 2: Is this specification enough? How do we specify that "nothing else changes"? Frame Problem!

Outline

Situation Calculus

SitCalc [McCarthy63,McCarthyHayes69,Reiter01] is very well-developed formalism for reasoning about actions.

SitCalc (Reiter's version)

First-Order multi sorted language - but over inductively defined situations (i.e,. situations are defined in Second-Order Logic). Sorts:

- Objects: representing the objects of the domain of interest - *e.g., rooms* A, B, C, *doors* d_{AB}, *and* d_{AC}
- Actions: progress the system - *e.g.,* $goto(x)$, *and* $openAllDoors()$
- Situations: representing the current state and the history that lead to that state - *see next*
- Fluents: assert a property of objects in situations - *predicates with an extra situation argument, see next*

Situation Calculus Language

Situations

- Situations denote states resulting from possible world histories.
- Situations are formed by making use of a distinguished constant S_0 and function symbol $do(a, s)$:
  - S_0 is the initial situation, before any action has been performed
  - $do(a, s)$ is the situation that results from doing action a in situation s
- Formally, the set of situations is defined by induction (in Second-Order Logic) inducing an infinite tree of situations.
- Each situation represent the state resulting from the history that leads to it from the initial situation S_0.

Example

Consider the situation term

$$do(goto(C), do(goto(A), do(openAlldoors(), do(goto(B), S_0))))$$

it represents situation that results by starting in the initial situation S_0 and then going to room B, pushing the button to open all closed doors, then going to room A, and finally going to room C.

Situation Calculus Language

Fluents

Predicates (or functions – but we do not consider fluents functions here) whose values may vary from situation to situation are called fluents.

These are written using predicate symbols whose last argument is a situation

Example

In our example we add to the dynamic predicates an extra situation parameter.

- $Open(d, s)$, which denotes that the door d is open in situation s.
- $SelfIn(r, s)$, which denotes that robot $self$ is in room r in situation s.
- Hence, the initial values of these fluents can now be described
  - $Open(x, S_0) \equiv (x = d_{AB})$
  - $SelfIn(x, S_0) \equiv (x = A)$

Situation Calculus Language

The distinguished predicate $Poss(a, s)$

A distinguished predicate symbol $Poss(a, s)$ is used to state that a may be performed, i.e., is executable, in situation s.

Example

For example:

- $Poss(goto(B), S_0)$ denotes that it is possible for robot $self$ to move from its current location (in situation S_0 robot $self$ is in room A) to room B.

  $Poss(goto(B), S_0)$ *actually holds in our example*

- $Poss(openAllDoors(), S_0)$ denotes that robot $self$ can push the button to open all closed doors in situation S_0

  $Poss(openAllDoors(), S_0)$ *does NOT hold in our example since* A *is not a control room*

This is the entire language of SitCalc (a first-order language over the second-order defined set of situations, i.e., the situation tree)

Running Example

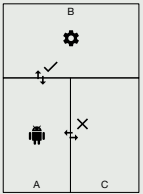
Actions

Action available to robot $self$:

- $goto(x)$ where x is a room.
  - PRE: To perform this action, there must be an open door between the room where $self$ is and x.
  - EFF: The effect is that $self$ gets to room x.
- $openAllDoors()$ to open all doors.
  - PRE: To perform this action, $self$ needs to be in a room that is a control room.
  - EFF: The effect is that all closed doors get opened.

Formalization

- $goto(x)$:
  - PRE: $Poss(goto(x), s) \equiv \exists r.SelfIn(r, s) \wedge \exists d.Door(d, r, x) \wedge Open(d, s)$
  - EFF: $SelfIn(x, do(goto(x), s)) \wedge \neg\exists r.r \neq x \wedge SelfIn(r, do(goto(x), s))$
- $openAllDoors()$:
  - PRE: $Poss(openAllDoors(), s) \equiv \exists r.SelfIn(r, s) \wedge ControlRoom(r)$
  - EFF: $Closed(d, s) \supset Open(d, do(openAllDoors(), s))$
    (Note, Problem 1 solved. This formula talks about two situations: s in which the action is executed and the one, and $do(openAllDoors(), s))$ resulting from executing the action.)

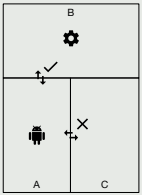


Running Example

Precondition Axioms (PRE)

One precondition axioms per action:

- $Poss(goto(x), s) \equiv \exists r.SelfIn(r, s) \wedge \exists d.Door(d, r, x) \wedge Open(d, s)$
- $Poss(openAllDoors(), s) \equiv \exists r.SelfIn(r, s) \wedge ControlRoom(r)$

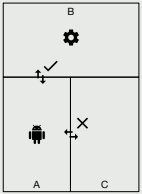


Effect Axioms (EFF)

Some effect axioms for each action:

- $SelfIn(x, do(goto(x), s)) \wedge \neg\exists r.(r \neq x \wedge SelfIn(r, do(goto(x), s)))$
- $Closed(d, s) \supset Open(d, do(openAllDoors(), s))$

What about Problem 2? How do specify that "nothing else" changes? Frame Problem!



Frame Problem

To really know how the world works, it is also necessary to know what fluents are unaffected by performing an action.

Example

- Opening the doors does not affect the room robot *self* is in

$$SelfIn(r, s) \supset SelfIn(r, do(openAllDoors(), s)$$

- As *self* moves around the state of the doors remains unchanged:

$$Open(d, s) \supset Open(d, do(goto(x), s))$$
$$\neg Open(d, s) \supset \neg Open(d, do(goto(x), s))$$

These are sometimes called frame axioms.

The Frame Problem

- We need to know a vast number of such frame axioms ...
- ... because few actions affect the value of a given fluent; most leave it invariant

  e.g.,: an object's colour is unaffected by picking things up, opening a door, using the phone, turning on a light, electing a new Prime Minister of UK, etc.
- The agent needs to reason efficiently with them

Reiter's Solution to the Frame Problem

Many ways of solving the frame problems have been explored by the AI scientific community in the past, promoting important scientific developments like default logics, autoepistemic logics, nonmonotonic reasoning.

Ray Reiter's Simple Solution to The Frame Problem in SitCalc

A simple solution to the frame problem (due to Ray Reiter) yields the following axioms:

- Use "successor state axioms" instead of effect axioms - one successor state axiom per fluent
- Use precondition axioms for specifying preconditions - one precondition axiom per action

Interestingly, we do not get fewer axioms at the expense of prohibitively long ones: the length of a successor state axioms is roughly proportional to the number of actions which affect the truth value of the fluent

The conciseness and perspicuity of the solution relies on

- *quantification over actions*
- *the assumption that relatively few actions affect each fluent*
- *the completeness assumption for effects*

Reiter's Solution to the Frame Problem

Ray Reiter [Reiter1991,Reiter2021], building on work by Pednault, Haas, Schubert, developed a simple solution to the frame problem for SitCalc theories of where effects are completely characterized by effect axioms *(as the ones we are looking at)*.

Reiter's solution to the Frame Problem is based on 3 steps:

- Step 1: adopt Normal Form for Effect Axioms
- Step 2: enforce Explanation Closure
- Step 3: replace effect axioms with Successor State Axioms

Reiter's Solution to the Frame Problem

Step 1: Normal Form for Effect Axioms

In general, for any fluent F, we can rewrite all the effect axioms as as two formulas of the form

- POSITIVE: $\Phi_F^+(\vec{x}, a, s) \supset F(\vec{x}, do(a, s))$
- NEGATIVE: $\Phi_F^-(\vec{x}, a, s) \supset \neg F(\vec{x}, do(a, s))$ *(Notation: $\vec{x}$ is a shorthand for $x_1, \ldots, x_n$.)*

Note that it must be the case that the following consistency condition holds:

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

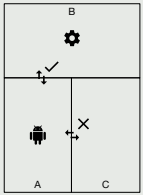
Otherwise we could instruct to make the fluent F true and false at the same time.

Running Example

Normal Form Effect Axioms (EFF)

Splitting effect axioms per fluent:

- POSITIVE: $(a = goto(x)) \supset SelfIn(x, do(a, s))$
- NEGATIVE: $(\exists y.a = goto(y) \wedge y \neq x) \supset \neg SelfIn(x, do(a, s))$
- POSITIVE: $(a = openAllDoors() \wedge Closed(d, s)) \supset Open(d, do(a, s))$
- NEGATIVE: $none$ *(under no circumstances an a -among ours- closes a door d, i.e., $False \supset Open(d, do(a, s))$.)*



Another Example

Example

Suppose we have two positive effect axioms for the fluent $Broken$:

$$Fragile(x) \supset Broken(x, do(drop(r, x), s))$$
$$NextTo(b, x, s) \supset Broken(x, do(explode(b), s))$$

These can be rewritten as a single normal form positive effect axiom:

$$(\exists r.a = drop(r, x) \wedge Fragile(x)) \vee (\exists b.a = explode(b) \wedge NextTo(b, x, s)) \supset Broken(x, do(a, s))$$

Similarly, suppose we have a single negative effect axiom:

$$\neg Broken(x, do(repair(r, x), s))$$

This can be rewritten as a single normal form negative effect axiom:

$$(\exists r.a = repair(r, x)) \supset \neg Broken(x, do(a, s))$$

Reiter's Solution to the Frame Problem

Step 2: Explanation Closure

Make a completeness assumption effect axioms:

- POSITIVE: $\Phi_F^+(\vec{x}, a, s) \supset F(\vec{x}, do(a, s))$
- NEGATIVE: $\Phi_F^-(\vec{x}, a, s) \supset \neg F(\vec{x}, do(a, s))$

This can be formalized by **explanation closure axioms**:

- if F was false and was made true by doing action a then condition $\Phi_F^+(\vec{x}, a, s)$ must have been true:

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

- if F was true and was made false by doing action a then condition $\Phi_F^-(\vec{x}, a, s)$ must have been true:

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

Note that explanation closure axioms are in fact disguised versions of frame axioms:

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

Reiter's Solution to the Frame Problem

Step 3: Successor State Axioms

It is easy to see that the conjunction of

- POSITIVE: $\Phi_F^+(\vec{x}, a, s) \supset F(\vec{x}, do(a, s))$
- NEGATIVE: $\Phi_F^-(\vec{x}, a, s) \supset \neg F(\vec{x}, do(a, s))$
- EXPL POS: $\neg F(\vec{x}, s) \wedge F(\vec{x}, do(a, s)) \supset \Phi_F^+(\vec{x}, a, s)$
- EXPL NEG: $F(\vec{x}, s) \wedge \neg F(\vec{x}, do(a, s)) \supset \Phi_F^-(\vec{x}, a, s)$

is equivalent to the following axiom, called Successor State Axiom (SSA) for $F(\vec{x}, s)$:

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

Note that if new actions or effects are introduced, the SSAs must be recomputed.

Running Example

We drop effect axioms in favor of successor state axioms.

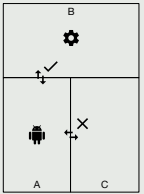
Normalized Effect Axioms (EFF)

Splitting effect axioms per fluent:

- POSITIVE: $(a = goto(x)) \supset SelfIn(x, do(a, s))$
- NEGATIVE: $(\exists y.a = goto(y) \wedge y \neq x) \supset \neg SelfIn(x, do(a, s))$

- POSITIVE: $(a = openAllDoors() \wedge Closed(d, s)) \supset Open(d, do(a, s))$
- NEGATIVE: $none$ i.e., $False \supset \neg Open(d, do(a, s))$

We still need to specify that "nothing else" changes, i.e., solve the Frame Problem!

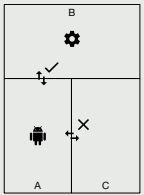


Successor State Axioms (SSA)

Successor state axioms, one per fluent:

- $SelfIn(x, do(a, s)) \equiv$
  $(a = goto(x)) \vee$
  $(selfIn(x, s) \wedge \neg(\exists y.a = goto(y) \wedge y \neq x))$

- $Open(d, do(a, s)) \equiv$
  $(a = openAllDoors() \wedge Closed(d, s)) \vee$
  $Open(d, s) \wedge \neg(False)$



How Does STRIPS Planning Framework Deal with the Frame Problem?

STRIPS Operator for Action $goto$

$goto(f, t, d)$ - robot goes from room f to room t using door d

- PRE: $SelfIn(f) \wedge Door(d, f, t) \wedge Open(d)$ - preconditions
- ADD: $SelfIn(t)$ - add list
- DEL: $SelfIn(f)$ - delete list

STRIPS Representation

- STRIPS represents states as databases of atoms
- actions' effects are represented as updates to this database
- only works if initial state is completely known
- is not a logic where one can reason about action

How Does STRIPS Planning Framework Deal with the Frame Problem?

STRIPS Operator for Action $goto$

$goto(f, t, d)$ - robot goes from room f to room t using door d

- PRE: $SelfIn(f) \wedge Door(d, f, t) \wedge Open(d)$ - preconditions
- ADD: $SelfIn(t)$ - add list
- DEL: $SelfIn(f)$ - delete list

STRIPS Representation

- STRIPS represents states as databases of atoms
- actions' effects are represented as updates to this database
- only works if initial state is completely known
- is not a logic where one can reason about action

Exercise

Let's specify the following dynamic domain in the SitCalc

Suppose that we have a fluent $LightOn(x, s)$ that is true iff light x is on in situation s and a fluent $PowerOn(s)$ that is true iff lthe power is on in situation s. Suppose also that we have an action $flipSwitch(x)$ that flips the switch of light x; the only effects of $flipSwitch(x)$ are (1) that it will turn light x on if x is currently off (i.e. not on) and the power is on, as well as (2) that it will turn light x off if x is currently on. Finally, assume that $flipSwitch$ is the only action that affects the fluent $LightOn$.

a) Write effect axioms for the action $flipSwitch$; your axioms should capture all the effects of the action.

b) Write frame axiom(s) for the action $flipSwitch$ and the fluent $LightOn$; your axioms should handle all cases where the fluent does not change when $flipSwitch$ is performed.

c) Write a successor state axiom for the fluent $LightOn$.

Exercise

Let's specify the following dynamic domain in the SitCalc

Suppose that we have a fluent $LightOn(x, s)$ that is true iff light x is on in situation s and a fluent $PowerOn(s)$ that is true iff lthe power is on in situation s. Suppose also that we have an action $flipSwitch(x)$ that flips the switch of light x; the only effects of $flipSwitch(x)$ are (1) that it will turn light x on if x is currently off (i.e. not on) and the power is on, as well as (2) that it will turn light x off if x is currently on. Finally, assume that $flipSwitch$ is the only action that affects the fluent $LightOn$.

a) Write effect axioms for the action $flipSwitch$; your axioms should capture all the effects of the action.

$$PowerOn(s) \wedge \neg LightOn(x, s) \supset LightOn(x, do(flipSwitch(x), s))$$
$$LightOn(x, s) \supset \neg LightOn(x, do(flipSwitch(x), s))$$

b) Write frame axiom(s) for the action $flipSwitch$ and the fluent $LightOn$; your axioms should handle all cases where the fluent does not change when $flipSwitch$ is performed.

$$LightOn(x, s) \wedge x \neq y \supset LightOn(x, do(flipSwitch(y), s))$$
$$\neg LightOn(x, s) \wedge (x \neq y \vee \neg PowerOn(s)) \supset \neg LightOn(x, do(flipSwitch(y), s))$$

c) Write a successor state axiom for the fluent $LightOn$.

$$LightOn(x, s) \equiv (a = flipSwitch(x) \wedge PowerOn(s) \wedge \neg LightOn(x, s)) \vee LightOn(x, s) \wedge \neg(a = flipSwitch(x))$$

Exercise

Let's specify the following dynamic domain in the SitCalc

Suppose that we have a fluent $LightOn(x, s)$ that is true iff light x is on in situation s and a fluent $PowerOn(s)$ that is true iff lthe power is on in situation s. Suppose also that we have an action $flipSwitch(x)$ that flips the switch of light x; the only effects of $flipSwitch(x)$ are (1) that it will turn light x on if x is currently off (i.e. not on) and the power is on, as well as (2) that it will turn light x off if x is currently on. Finally, assume that $flipSwitch$ is the only action that affects the fluent $LightOn$.

a) Write effect axioms for the action $flipSwitch$; your axioms should capture all the effects of the action.

$$PowerOn(s) \wedge \neg LightOn(x, s) \supset LightOn(x, do(flipSwitch(x), s))$$
$$LightOn(x, s) \supset \neg LightOn(x, do(flipSwitch(x), s))$$

b) Write frame axiom(s) for the action $flipSwitch$ and the fluent $LightOn$; your axioms should handle all cases where the fluent does not change when $flipSwitch$ is performed.

$$LightOn(x, s) \wedge x \neq y \supset LightOn(x, do(flipSwitch(y), s))$$
$$\neg LightOn(x, s) \wedge (x \neq y \vee \neg PowerOn(s)) \supset \neg LightOn(x, do(flipSwitch(y), s))$$

c) Write a successor state axiom for the fluent $LightOn$.

$$LightOn(x, s) \equiv (a = flipSwitch(x) \wedge PowerOn(s) \wedge \neg LightOn(x, s)) \vee LightOn(x, s) \wedge \neg(a = flipSwitch(x))$$

Exercise

Let's specify the following dynamic domain in the SitCalc

Suppose that we have a fluent $LightOn(x, s)$ that is true iff light x is on in situation s and a fluent $PowerOn(s)$ that is true iff lthe power is on in situation s. Suppose also that we have an action $flipSwitch(x)$ that flips the switch of light x; the only effects of $flipSwitch(x)$ are (1) that it will turn light x on if x is currently off (i.e. not on) and the power is on, as well as (2) that it will turn light x off if x is currently on. Finally, assume that $flipSwitch$ is the only action that affects the fluent $LightOn$.

a) Write effect axioms for the action $flipSwitch$; your axioms should capture all the effects of the action.

$$PowerOn(s) \wedge \neg LightOn(x, s) \supset LightOn(x, do(flipSwitch(x), s))$$
$$LightOn(x, s) \supset \neg LightOn(x, do(flipSwitch(x), s))$$

b) Write frame axiom(s) for the action $flipSwitch$ and the fluent $LightOn$; your axioms should handle all cases where the fluent does not change when $flipSwitch$ is performed.

$$LightOn(x, s) \wedge x \neq y \supset LightOn(x, do(flipSwitch(y), s))$$
$$\neg LightOn(x, s) \wedge (x \neq y \vee \neg PowerOn(s)) \supset \neg LightOn(x, do(flipSwitch(y), s))$$

c) Write a successor state axiom for the fluent $LightOn$.

$$LightOn(x, s) \equiv (a = flipSwitch(x) \wedge PowerOn(s) \wedge \neg LightOn(x, s)) \vee LightOn(x, s) \wedge \neg(a = flipSwitch(x))$$

Exercise

Let's specify the following dynamic domain in the SitCalc

Suppose that we have a fluent $LightOn(x, s)$ that is true iff light x is on in situation s and a fluent $PowerOn(s)$ that is true iff lthe power is on in situation s. Suppose also that we have an action $flipSwitch(x)$ that flips the switch of light x; the only effects of $flipSwitch(x)$ are (1) that it will turn light x on if x is currently off (i.e. not on) and the power is on, as well as (2) that it will turn light x off if x is currently on. Finally, assume that $flipSwitch$ is the only action that affects the fluent $LightOn$.

a) Write effect axioms for the action $flipSwitch$; your axioms should capture all the effects of the action.

$$PowerOn(s) \wedge \neg LightOn(x, s) \supset LightOn(x, do(flipSwitch(x), s))$$
$$LightOn(x, s) \supset \neg LightOn(x, do(flipSwitch(x), s))$$

b) Write frame axiom(s) for the action $flipSwitch$ and the fluent $LightOn$; your axioms should handle all cases where the fluent does not change when $flipSwitch$ is performed.

$$LightOn(x, s) \wedge x \neq y \supset LightOn(x, do(flipSwitch(y), s))$$
$$\neg LightOn(x, s) \wedge (x \neq y \vee \neg PowerOn(s)) \supset \neg LightOn(x, do(flipSwitch(y), s))$$

c) Write a successor state axiom for the fluent $LightOn$.

$$LightOn(x, s) \equiv (a = flipSwitch(x) \wedge PowerOn(s) \wedge \neg LightOn(x, s)) \vee LightOn(x, s) \wedge \neg(a = flipSwitch(x))$$

The Qualification Problem

is that of specifying the necessary and sufficient conditions for an action to be executable

E.g.: when Is the action of starting a car c possible?

- when there is petrol
- when the battery is not dead
- when there is no water infiltration in the wiring
- when there is no potato in the tailpipe
- ...

The simple approach seen earlier ignores all the minor qualifications

For a fully general solution, we need nonmonotonic/defeasible reasoning, where we infer conclusions tentatively, but may retract them based on further evidence

The Ramification Problem

is that of specifying all the indirect effects of actions

E.g.:when we paint an object x

- $Painted(x)$ becomes true - direct effect
- $Painted(y)$ becomes true if y is part of x - indirect effect
- applies to subparts as well

The simple solution to the frame problem assumes that all effects are explicitly specified.

State constraints such as $partOf(y, x) \supset Painted(y, do(paint(x), s))$ can specify indirect effects.

Again, for a fully general solution, we need nonmonotonic reasoning

Outline

Reiter's SitCalc Basic Action Theories

Basic Action Theories (BAT)

Reiter's SitCalc Basic Action Theories are as follows:

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{una} \cup \mathcal{D}_{pre} \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{S_0}$$

where

- Σ are the foundational axioms for situations - *second-order logic*
  Inductive definiton of the set of situations
- $\mathcal{D}_{una}$ is the set of unique names axioms for action - *first-order logic*
  For distinct action names A_1 and A_2, $A_1(\vec{x}) \neq A_2(\vec{y})$;
  identical actions have identical arguments: $A(x_1, \ldots, x_n) = A(y_1, \ldots, y_n) \supset x_1 = y_1 \wedge \cdots \wedge x_n = y_n$.
- $\mathcal{D}_{pre}$ is a set of action precondition axioms *first-order logic*
  $Poss(A(\vec{x}), s) \equiv \Phi_A^{pre}(\vec{x}, s)$ - *one for each action $A(\vec{x})$*
- $\mathcal{D}_{ssa}$ is a set of successor state axioms *first-order logic*
  $F(\vec{x}, do(a, s)) \equiv \Phi_F^{ssa}(\vec{x}, s)$ - *one for each fluent $F(\vec{x}, s)$*
- $\mathcal{D}_{S_0}$ is the initial situation description - *first-order logic*
  $\mathcal{D}_{S_0}$ is a set of first order sentences with the property that S_0 is the only term of sort $situation$ mentioned by the fluents of a sentence of $\mathcal{D}_{S_0}$. Thus, no fluent of a formula of $\mathcal{D}_{S_0}$ mentions a variable of sort $situation$ or the function symbol do. $\mathcal{D}_{S_0}$ will play the role of the initial situation of the world (i.e. the one we start off with, before any actions have been "executed").

SitCalc BAT Foundational Axioms

The SitCalc Foundational Axioms Σ are:

$$do(a_1, s_1) = do(a_2, s_2) \supset a_1 = a_2 \wedge s_1 = s_2 \quad (1)$$

$$\forall P.P(S_0) \wedge \forall a.\forall s.[P(s) \supset P(do(a, s))] \supset \forall s.P(s) \quad (2)$$

$$\neg s \sqsubset S_0 \quad (3)$$

$$s \sqsubset do(a, s') \equiv s \sqsubseteq s' \quad (4)$$

(2) is a second order logic axiom that says that the set of situations is the least set that contans the initial situation and such that doing an action in a situation produces a situation.

Second order logic is very expressive, but it is incomplete, i.e., the set of valid sentences is not recursively enumerable.

Reasoning 0: KB Satisfiability

A preliminary task for every logical theory is to check its consistency, i.e., it actually describes something meaningful, without contradicting itself.

Formally, this amounts in checking its satisfiability, i.e., if it admits at least one model.

Basic Action Theories enjoy a very strong property for this:

Theorem (Relative Satisfiability)

The second-order theory $\mathcal{D} = \Sigma \cup \mathcal{D}_{una} \cup \mathcal{D}_{pre} \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{S_0}$ is satisfiable iff the first-order theory $\mathcal{D}_{una} \cup \mathcal{D}_{S_0}$ is.

Reasoning 1: Projection Task

A first important reasoning task for an agent in a dynamic world is the projection task: determine what is true after performing a sequence of actions.

Projection

Given a sequence of actions, determine what would be true in the situation that results from performing that sequence.

This can be formalized as follows: Suppose that $\varphi(s)$ is a formula with a free situation variable s. To find out if $\varphi(s)$ would be true after performing $a_1, \ldots, a_{n-1}, a_n$ in the initial situation, we determine whether or not

$$\mathcal{D} \models \varphi((do(a_n, do(a_{n-1}, \ldots, do(a_1, S_0) \ldots)))$$

Example

For example, using the effect and frame axioms from before, it follows that $Open(d_{AC}, s) \wedge SelfIn(C, s)$ would hold after doing the sequence of actions $goto(B), openAllDoors(), goto(A), goto(C)$, i.e., with $s = do(goto(C), do(goto(A), do(openAllDoors(), do(goto(B), S_0))))$.

Reasoning 2: Executability Task

A second important reasoning task is the executability task: determine whether a sequence of action is executable.

A sequence of action is executable if every action satisfies its precondition in the situation in which is executed. Formally this be defined on the length of the sequence.

Executability

The executability task is the task of determining whether a sequence of actions is indeed executable.
This can be formalized as follows: To find out if the sequence $a_1, \ldots, a_n$ is executable in the initial situation, we determine whether $\mathcal{D} \models \bigwedge_{i \in \{1,\ldots,n\}} Poss(a_i, do(a_{i-1}, \ldots, do(a_1, S_0) \ldots))$, i.e.:

$$\mathcal{D} \models Poss(a_1, S_0)$$
$$\mathcal{D} \models Poss(a_2, do(a_1, , S_0))$$
$$\ldots$$
$$\mathcal{D} \models Poss(a_n, do(a_{n-1}, \ldots, do(a_1, S_0) \ldots))$$

Example

For example, the sequence of actions $goto(B), openAllDoors(), goto(A), goto(C)$ is executable, while the sequence $goto(B), goto(A), goto(C)$ is not.

Reasoning 3: Planning Task

A third important reasoning task is the planning task: find a sequence of actions that achieves a goal.

This can be formulated as follows:

Planning

Given a goal formula $\varphi_g(s)$, find a sequence of actions $a_1, \ldots, a_n$ such that

$$\mathcal{D} \models \varphi_g((do(a_n, do(a_{n-1}, \ldots, do(a_1, S_0)\ldots))) \\ \wedge executable((do(a_n, do(a_{n-1}, \ldots, do(a_1, S_0))$$

where $executable$ can be defined as shown earlier

We can obtain a plan from a constructive proof of $\mathcal{D} \models \exists s.\varphi_g(s) \wedge executable(s)$

Example

For example, the sequence of actions $goto(B), openAllDoors(), goto(A), goto(C)$, achieves the goal $SelfIn(C, s)$.

SitCalc Key Feature: Regression

But how do we reason in SitCalc considering that is a second-order theory?

We have a basic mechanism, called **regression**, that allows us to reduce reasoning about future situations (second-order) to reasoning on the initial situation only (first-order).

Regression

- Regression reduces reasoning formulas about the next situation to equivalent formulas about the current situation.
- It does so essentially by substituting the body of SSA for corresponding fluents.
- Analogous to computing **weakest precondition**.

- By iterating regression, we can reduce reasoning about a given future situation to reasoning about the initial situation.
- It greatly simplifies main forms of reasoning about actions:
  - **projection**: query the result situation after a given sequence of actions;
  - **executability**: check the executability of a sequence of actions.

Regression

Regressable Formulas

A regressable formula is a formula such that each of its *situation* terms are rooted at S_0, and therefore, one can tell, by inspection of such a term, exactly how many actions it involves.

Let φ be a regressable formula

The Regression Operator - Atoms

If φ is an atom, there are four possibilities:

- φ is a situation independent atom. Then
  $$\mathcal{R}[\varphi] = \varphi.$$
- φ is a fluent atom in S_0 of the form $F(\vec{t}, S_0)$. Then
  $$\mathcal{R}[\varphi] = \varphi.$$
- φ is a $Poss$ atom, see next.
- φ is a fluent atom not in S_0, see next.

Regression

The Regression Operator - Atoms - $Poss$

If φ is of the form $Poss(A(\vec{t}), \sigma)$ for terms $A(\vec{t})$ and σ of sort *action* and *situation* respectively. Then there must be an action precondition axiom for A of the form

$$Poss(A(\vec{x}), s) \equiv \Phi_A^{pre}(\vec{x}, s).$$

Then

$$\mathcal{R}[\varphi] = \mathcal{R}[\Phi_A^{pre}(\vec{t}, \sigma)].$$

In other words, replace the atom $Poss(A(\vec{t}), \sigma)$ by a suitable instance of the right hand side of the equivalence in A's action precondition axiom, and regress that expression.

The Regression Operator - Atoms - Fluents not in S_0

If φ is of the form $F(\vec{t}, do(\alpha, \sigma))$. Let F's successor state axiom in $\mathcal{D}_{ss}$ be

$$F(\vec{x}, do(a, s)) \equiv \Phi_F^{ssa}(\vec{x}, a, s).$$

Then

$$\mathcal{R}[\varphi] = \mathcal{R}[\Phi_F^{ssa}(\vec{t}, \alpha, \sigma)].$$

In other words, replace the atom $F(\vec{t}, do(\alpha, \sigma))$ by a suitable instance of the right hand side of the equivalence in F's successor state axiom, and regress this formula.

The Regression Operator - Non-atomic formulas

For non-atomic formulas, regression is defined inductively.

- $\mathcal{R}[\neg\varphi] = \neg\mathcal{R}[\varphi]$,
  $\mathcal{R}[\varphi_1 \wedge \varphi_2] = \mathcal{R}[\varphi_1] \wedge \mathcal{R}[\varphi_2]$,
  $\mathcal{R}[\exists v.\varphi] = \exists v.\mathcal{R}[\varphi]$.

Regression

Observations:

- The regression operator eliminates $Poss$ atoms in favour of their definitions as given by action precondition axioms, and replaces fluent atoms about $do(\alpha, \sigma)$ by logically equivalent expressions about σ as given by successor state axioms. Moreover, it repeatedly does this until it cannot make such replacements any further.

- Each $\mathcal{R}$-step reduces the depth of nesting of the function symbol do in the fluents of φ by substituting suitable instances of Φ_F^{ssa} for each occurrence of a fluent atom of φ of the form $F(t_1, \ldots, t_n, do(\alpha, \sigma))$. Since no fluent atom of Φ_F^{ssa} mentions the function symbol do, the effect of this substitution is to replace each such F by a formula whose fluents mention only the situation term σ, and this reduces the depth of nesting by one.

Theorem (Regression Theorem)

Suppose φ is a regressable sentence and $\mathcal{D}$ is a basic theory of actions. Then,

$$\mathcal{D} \models \varphi \text{ iff } \mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models \mathcal{R}[\varphi].$$

Observe: $\mathcal{D} \models \varphi$ is a second-order logical implication problem, while $\mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models \mathcal{R}[\varphi]$ is a first-order logical implication problem

Again, we are reducing second-order reasoning into first-order reasoning.

Implementation in Prolog

Reiter has shown that BATs with a closed initial database can straightforwardly be translated into Prolog and that the Prolog interpreter is a sound reasoner for answwering regressable queries on them.

Example (E.g. an education database application)

```
poss(drop(St,C),S) :- enrolled(St,C,S).
poss(register(St,C),S) :- not (prtereq(P,C), not (grade(St,P,G,S), G >= 50)).

enrolled(St,C,do(A,S)) :- A = register(St,C) ; enrolled(St,C,S), not A = drop(St,C).

enrolled(john,c100,s0).  grade(mary, c100,60,s0).  prereq(c100,m100).

...
```

Exercise

Example (Solve by Regression)

Check by regression that using the effect and frame axioms from before, it follows that $Open(d_{AC}, s) \wedge SelfIn(C, s)$ would hold after doing the sequence of actions $goto(B), openAllDoors(), goto(A), goto(C)$„ i.e., whether

$$\mathcal{D} \models (Open(d_{AC}, s_f) \wedge SelfIn(C, s_f))$$

where $s_f = do(goto(C), do(goto(A), do(openAllDoors(), do(goto(B), S_0))))$

Example (Solve by Regression)

Check by regression that:

- the sequence of actions $goto(B), openAllDoors(), goto(A), goto(C)$ is executable;
- the sequence $goto(B), goto(A), goto(C)$ is not executable.

SitCalc: Temporal Reasoning

Regression is not sufficient for more sophisticated temporal properties:

- There exists a future situation such that α
- For all (future) situations α holds
- Eventually whatever actions we do we have α
- Always when α then eventually β
- ...

Beyond Projection and Executability

When we deal with such temporal properties we need **verification techniques**, most of which assume finite number of states (i.e., finite object domain in the SitCalc).

- If we assume finite number of objects, then we can model check SitCalc Action Theories!

If the number of objects is infinite, then classical model checking techniques do not work, but recently there have been interesting advancements in SitCalc wrt verification:

- *Incomplete fixpoint approximation-based methods*
- *Complete temporal reasoning for special fragments (like two variable FOL fragment)*
- *Verification of "Bounded SitCalc theories" is decidable.*

SitCalc Limitations

This version of SitCalc has a number of limitations:

- no time: cannot talk about how long actions take, or when they occur
- only known actions: no hidden exogenous actions, no unnamed events
- no concurrency: cannot talk about doing two actions at once
- only discrete situations: no continuous actions, like pushing an object from A to B
- only hypotheticals: cannot say that an action has occurred or will occur
- actions have deterministic effects: can't model throwing a coin
- no knowledge-producing actions, e.g., sense the distance to the wall
- only primitive actions: no actions made up of other parts, like conditionals or iterations
- etc.

There has been work on extensions to handle most of these limitations.

References

R. Reiter, Knowledge in Action: *Logical Foundations for Specifying and Implementing Dynamical Systems,* MIT Press, 2001.

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

F. Pirri and R. Reiter. Some contributions to the metatheory of the situation calculus. *Journal of the ACM,* 46(3):261–325, 1999.