Evolution Prospection

Luís Moniz Pereira
Han The Anh

CENTRIA – UNL, Portugal
Prospective Logic Programming - PLP

• PLP is our starting point.

• It enables evolving programs to look ahead prospectively into its possible future states, in order prefer among them and better satisfy goals.

• Pre-existing implementation is ACORDA:
  – Based on EVOLP update language for LPs
  – Relies on XSB-XASP and its interface to Smodels
  – Uses Smodels to compute abductive models
Prospective Agent Architecture
Evolution Prospecting Agents - EPA

- Re-implements ACORDA in NegAbdual – an XSB system combining constructive negation and abduction over the well-founded semantics of the ABDUAL language.

- Provides extended ways to model evolving prospective agents. These can now look further into their possible futures, and prefer on the basis of evolution history too.

- Provides base constructs for the design and the implementation of the system.
Constructs of the EPA system

• Abducibles
• \textit{a priori} preferences
• \textit{a posteriori} preferences
• Active goals
• Context-sensitive integrity constraints
• Levels of commitment
Abducibles

• Each program has a set of abducibles.
• Abducibles can be seen as possible explanations of given queries.
• An abducible A can be assumed if it is expected in the given situation and there is no expectation to the contrary:
  
  consider(A) <- expect(A), not expect_not(A), A

• Program abducibles are declared by reserved NegAbdual predicate `abds/1`
a priori preferences

- Preferences over abducibles, having the form:
  \[ a \prec b \prec L_1, \ldots, L_n \]
  where: \( a, b \) - abducibles; \( L_i \) - domain literals
- Meaning: if \( b \) is assumed then \( a \prec b \)
  forces \( a \) to be assumed too
- The preference rule is coded as:
  \[
  \text{expect_not}(b) \prec L_1, \ldots, L_n, \text{not consider}(a)
  \]
  where, for any abducible \( Ab \):
  \[
  \text{consider}(Ab) \prec \text{expect}(Ab), Ab, \text{not expect_not}(Ab)
  \]
**a posteriori** preferences

- Preference over abductive solutions has the form:

  \[ Ai \ll Aj \leftarrow \text{holds}_{\text{given}}(Li,Ai), \text{holds}_{\text{given}}(Lj,Aj) \]

  where: \( Ai, Aj \) - abductive solutions

- Meaning:

  \( Ai \) is preferred to \( Aj \) *a posteriori* if \( Li \) and \( Lj \) are true as the side-effects of \( Ai \) and \( Aj \), respectively; i.e. without further abductions.
Active goals

- At each cycle, the agent has a set of active goals and desires that it wants to satisfy. An active goal AG is coded by the rule:

  \[
  \text{on\_observe}(AG) \leftarrow L_1, \ldots, L_n
  \]

  meaning “on observing \( L_1, \ldots, L_n \) trigger goal \( AG \) “.

- An active goal may be triggered by some events, previous commitments, or history-related information.
Context-sensitive integrity constraint - IC

- Typically, all integrity constraints coded in the KB must be satisfied. But when considering evolving agents, ICs that depend on context (time points, external environment) are necessary.

- An IC depending on context C is coded by an active goal:

  ```
  on_observe(not icName) <- C
  icName <- icBody
  ```

  If C is true, active goal `not icName` is launched, forcing the IC

  ```
  <- icBody
  ```

  to be satisfied.
Levels of commitment

• Each prospective cycle is completed by committing to preferred remaining ongoing abductive solutions.

• History of an evolution is kept by setting time stamps for committed to abducibles.

• Based on their influence on the future, we classify commitments in 3 categories:
  – **Hard**: indefeasible, permanently affect the future
  – **Ongoing**: defeasible, keep on affecting the future till defeated
  – **Temporary**: momentary, affect only the current state
Prospective Agents - PA

• Depending on capabilities and current need, a PA may wish or has to satisfy:
  – only active goals at hand – single-step PA
  – or long-term active goals – multiple-step PA
    whose preferences may depend on history
Single-step agents – example

John is buying a ticket to travel. Next we model the following:
He has 2 choices: saver or flexible ticket.
Flexible one is expensive. But if he has money, he does not wish a saver one, as he could never change or return it, and therefore would loose money if it’s not used.
For the saver ticket, after buying it, the reverse action of returning is disallowed (hard commitment). If John has not much money, he doesn’t expect buying an expensive one. Moreover, if John bought a new car he isn’t supposed to have much money left.
Later on, waiting for the flight, John finds his mother is ill. He wants to stay to take care of her. So he wishes to cancel the ticket and not loose money.
Abducibles declaration

- Abducibles
  
  abds([saver_ticket/0, flexible_ticket/0, cancel_ticket/0, loose_money/0]).

- Ongoing commitments
  
  ongoing_commitment([flexible_ticket]).

- Hard commitments
  
  hard_commitment([saver_ticket]).

- Abducibles not declared hard or ongoing are temporary by default.
Phase 1 – Buy ticket to travel

expect(saver_ticket).
expect(flexible_ticket).

on_observe(ticket) ← travel.
ticket ← saver_ticket.
ticket ← flexible_ticket.

expensive(flexible_ticket).
expect_not(saver_ticket) ← have_money.
expect_not(X) ← empty_pocket, expensive(X).
empty_pocket ← buy_new_car.
have_money ← not empty_pocket.
Phase 2 – Mother ill, stay at home

\begin{align*}
\text{expect}(\text{cancel\_ticket}). & \quad \text{expect}(\text{loose\_money}). \\
\text{on\_observe}(\text{stay\_home}) & \leftarrow \text{mother\_ill}. \\
\text{stay\_home} & \leftarrow \text{cancel\_ticket}. \\
\text{stay\_home} & \leftarrow \text{loose\_money}. \\
\text{change\_ticket} & \leftarrow \text{mother\_ill}. \\
\end{align*}

% Two context-sensitive integrity constraints

\begin{align*}
\text{on\_observe}(\text{not saver\_ticket\_ic}) & \leftarrow \text{change\_ticket}. \\
\text{saver\_ticket\_ic} & \leftarrow \text{saver\_ticket, cancel\_ticket}. \\
\text{on\_observe}(\text{not cancel\_ticket\_ic}) & \leftarrow \text{change\_ticket}. \\
\text{cancel\_ticket\_ic} & \leftarrow \text{cancel\_ticket, ticket}. \\
\end{align*}
An *a posteriori* preference

- Prefer abductive solutions leading to cancelling ticket to ones leading to loosing money

\[
Ai \ll Aj \leftarrow \text{holds\_given(cancel\_ticket,Ai),}
\text{holds\_given(loose\_money,Aj).}
\]
Running the example - 1

1) consult main program: ?- [metaAb].
2) load example file (name without extension): ?- load('ticket').
3) assert temporal event travel: ?- event(travel).
4) update/1 takes integer argument, representing # of steps to be updated. It finds at each cycle step the abductive solutions satisfying the active goals of the cycle (using acordaProspection/2): ?- update(2). % update 2 steps
5) show evolution so far, defined by abductions made: ?- timeStamps.

   1 - [flexible_ticket]  2 - [flexible_ticket]

6) assert temporal event mother_ill: ?- event(mother_ill).
7) update 1 step: ?- update(1).
8) show evolution: ?- timeStamps.

   1 - [flexible_ticket]  2 - [flexible_ticket]  3 - [cancel_ticket]

9) update 1 step: ?- update(1).
10) show evolution: ?- timeStamps.

   1 - [flexible_ticket]  2 - [flexible_ticket]
   3 - [cancel_ticket]  4 - []
Running the example (2)

% another run with the same initial scenario, but new permanent event buy_new_car

1) reload:     ?- load('ticket').
2) enter event travel:    ?- event(travel).
3) assert permanent event: ?- asserts(buy_new_car).
4) update 2 steps:  ?- update(2).
5) show evolution: ?- timeStamps.
   1 - [saver_ticket]  2 - [saver_ticket]
6) enter event mother_ill:    ?- event(mother_ill).
7) update 1 step:  ?- update(1).
8) show evolution: ?- timeStamps.
   1 - [saver_ticket]  2 - [saver_ticket]  3 - [loose_money, saver_ticket]
9) update 1 step:  ?- update(1).
10) show evolution: ?- timeStamps.
    1 - [saver_ticket]                  2 - [saver_ticket]
    3 - [loose_money, saver_ticket]    4 - [saver_ticket]
Sophie’s example

• Sophie and her two children are at a Nazi concentration camp.
• A guard confronts Sophie and tells her that one of her children will be allowed to live and one will be killed.
• Sophie must decide which child will be killed. She can prevent the death of either of her children, but only by condemning the other to be killed.
• If she chooses neither both will be killed.
• For each child Sophie has an equally strong reason to save him or her.
Modelling the example - 1

- Declaration of abducibles

  \[
  \text{abds([letting\_both\_die/0, kill/1, flip\_a\_coin/0])}.
  \]

- There is always expectation for the abducibles declared

  \[
  \text{expect(kill(\_)).}
  \]
  \[
  \text{expect(flip\_a\_coin).}
  \]
  \[
  \text{expect(letting\_both\_die).}
  \]
Modelling the example - 2

% Sophie has to decide, on being given the choice by the guard

on_observe(decide) <- Sophie_choice.

% Possible decisions for Sophie

decide <- letting_both_die, not kill, not flip.
decide <- choose, not flip.
decide <- flip.

choose <- kill(child_1), not kill(child_2).
choose <- kill(child_2), not kill(child_1).

kill <- kill(child_1).
kill <- kill(child_2).

flip   <- flip_a_coin.
Modelling the example - 3

% Special reason for a child

expect_not(kill(C)) \leftarrow\text{special reason}(C).

% If there is special reason for only one child

expect_not(flip_a_coin) \leftarrow\text{special reason}(child_1),
\hspace{1cm}\text{not\ special\ reason}(child_2).

expect_not(flip_a_coin) \leftarrow\text{special reason}(child_2),
\hspace{1cm}\text{not\ special\ reason}(child_1).
a posteriori preferences

die(2) <- letting_both_die.
die(1) <- flip. die(1) <- choose.

% Prefer abductive solutions with less people dying
Ai << Aj <- holds_given(die(N), Ai),
    holds_given(die(K), Aj), N < K.

pr(feel_guilty, 1) <- kill(X).
pr(feel_guilty, 0.5) <- flip_a_coin.

% Prefer abductive solutions with smaller probability of feeling guilty
Ai << Aj <- holds_given( pr(feel_guilty,Pi), Ai),
    holds_given( pr(feel_guilty,Pj), Aj),
    Pi < Pj.
Running the example

1) consult the main program:  ?- [metaAb].
2) load file (note that file name without extension):  ?- load('mother').
3) show the abductive solutions:  ?- ab(decide,L).
   L = [ flip_a_coin ] ;
   L = [ kill(child_1), not flip_a_coin, not kill(child_2) ] ;
   L = [ kill(child_2), not flip_a_coin, not kill(child_1) ] ;
   L = [ letting_both_die, not flip_a_coin, not kill(child_1), not kill(child_2) ] ;
   no
4) show the list of abductive solutions:  ?- allAbds(decide,L).
   L = [ [flip_a_coin], [letting_both_die], [kill(child_1)], [kill(child_2)] ] ;  no
5) show the final decision:  ?- acordaProspection(decide,L).
   L = [ [flip_a_coin] ] ;  no
Time-sensitive preferences

• As an agent is evolving its preferences may change, depending on the evolution time point agent is at, or even on the whole evolution history or part of it taken into account.

• Example: John has lunch everyday, fast food or fruit. His favourite is fast food, to save time for work. He wants not to be fat, and avoids fast food 3 days in a row. After 2 days he prefers fruit instead.
Modelling the example - 1

abds([fast_food/0, fruit/0]).
expect(fast_food).
expect(fruit).

on_observe(lunch).
lunch <- fast_food, not fruit.
lunch <- fruit, not fast_food.

save_time <- fast_food.
gathering <- fruit.
waste_time <- gathering.
Modelling the example - 2

\[
\begin{align*}
Ai &<\!< Aj <- \text{not fat_prolog,} \\
& \quad \text{holds\_given(save\_time,}\ Ai), \\
& \quad \text{holds\_given(waste\_time,}\ Aj). \\
Ai &<\!< Aj <- \text{fat_prolog,} \\
& \quad \text{holds\_given(fruit,}\ Ai), \\
& \quad \text{holds\_given(fast\_food,}\ Aj).
\end{align*}
\]

\% Representing time-dependent predicate fat_prolog/0 (prolog code).

\[
\begin{align*}
\text{fat\_prolog} &:- \text{times(fast\_food,3).} \\
\text{times(X,N)} &:- \text{current\_state(S),} \ M = S - N + 1, \\
& \quad M > 0, \text{have\_from\_to(X,}\ M, S). \\
\text{have\_from\_to(X,M,S)} &:- M > S, !. \\
\text{have\_from\_to(X,M,S)} &:- \text{timeStamp(As,}\ M), \\
& \quad \text{member(X,}\ As), \ M1 = M+1, \\
& \quad \text{have\_from\_to(X,}\ M1, S). 
\end{align*}
\]
Running the example

1) consult the main program:    ?- [metaAb].
2) load file:                 ?- load('lunch').
3) update 8 steps:          ?- update(8).
4) see evolution:           ?- timeStamps.

1 - [fast_food]
2 - [fast_food]
3 - [fruit]
4 - [fast_food]
5 - [fast_food]
6 - [fruit]
7 - [fast_food]
8 - [fast_food]
Inevitable actions

• Abducibles that belong to all abductive solutions are called inevitable. They are committed to whatever the final abductive solution is.

• Actually, committing to some abducible changes the KB and may trigger new preferences which may help rule out irrelevant abductive solutions.
Inevitable actions - example

• John, a student, wants to get some money.
• He can go to one of 3 banks: the one in Monte, the one in Maths, or the university branch. All 3 are the same distance from the Residence.
• Moreover John needs a book for project work. His only choice is the library.
• At first he can’t decide which bank to go. Then he realizes he must go to the library in any case, and so does it first.
• Arrived there he notices the university branch is now the nearest. So he then chooses to go there.
Modelling the example - 1 committing to inevitable abducibles

abds([library/0, monte_bank/0, math_bank/0, univ_bank/0]).
expect(library).         expect(monte_bank).
expect(math_bank).       expect(univ_bank).

% Go to bank to get money
on_observe(take_money).
take_money <- monte_bank, not math_bank, not univ_bank.
take_money <- math_bank, not monte_bank, not univ_bank.
take_money <- univ_bank, not math_bank, not monte_bank.

% Go to library to find book
on_observe(find_book).
find_book <- library.
Modelling the example - 2 committing to inevitable abducibles

% Commit to library (prolog code)
go_to(library) :- commit_to(library).
cur_pos(C) :- go_to(C).
dif_distance :- cur_pos(library).

% This preference is triggered when library is committed to
Ai << Aj <- dif_distance,
    hold(dist(Di),Ai),
    hold(dist(Dj),Aj), Di < Dj.

% Facts about distances
dist(0)  <- prolog(cur_pos(library)), univ_bank.
dist(10) <- prolog(cur_pos(library)), monte_bank.
dist(5)  <- prolog(cur_pos(library)), math_bank.
Running the example

1) consult the main program:  
?- [metaAb].

2) load file:  
?- load('inevit_action').

3) find all abductive solutions for active goals: return_book and take_money
   (query is conjunction):  
?- allAbds(query,L).
   L = [ [library, math_bank], [library, monte_bank], [library, univ_bank] ];  no

4) acordaProspection/2 predicate does this: first commits to inevitable abducibles
   if any, then take into account \textit{a posteriori} preferences to rule out less preferred
   abductive solutions obtained by allAbds/2:  
?- acordaProspection(query,L).
   L = [ [library, math_bank] ];  no

5) update 1 step:  
?- update(1).  yes

6) time stamped commitments at each cycle:  
?- timeStamps.
   1 - [library, math_bank]
Multiple-step prospective agents

• When looking ahead a number of steps into the future, simple \((a \text{ posteriori})\) local preferences aren’t appropriate enough.

• The agent must be able to prefer amongst evolutions by:
  – quantitatively or qualitatively evaluating their consequences \((a \text{ posteriori evolution preferences})\)
  – or evaluating their historical information \((evolution \text{ history preferences})\)
a posteriori Evolution Preferences

- Preferences over evolutions, by quantitatively or qualitatively evaluating their consequences, have the form:
  \[ E_i <<< E_j \Leftarrow \text{holds}_\text{in}_\text{evol}(L_i,E_i), \]
  \[ \text{holds}_\text{in}_\text{evol}(L_j,E_j) \]
  where: \( E_i, E_j \) - evolutions; \( L_i, L_j \) - domain literals.

- Meaning:
  \( E_i \) is preferred to \( E_j \) if \( L_i \) and \( L_j \) are true as side-effects of evolving according to \( E_i \) or \( E_i \), respectively; i.e. without any further abductions.
During war time David, a good general, needs to decide to save one of his cities, a or b, from an attack.

He does not have enough military resources to save both. If a city is saved, its citizens are saved.

A bad general, who just sees the situation at hand, would prefer to save the city with most population, but a good one looks ahead a number of steps into the future, in order to choose the best strategy for the war as a whole.
Example (cont.)

Having been scheduled for the next day to make an opportunity counter-attack on one of two cities of the enemy, either a small or a big city, the prior action of saving a city should take this foreseen future into account.

A successful attack on a small city is always expected, but a successful attack on a big city leads to a better probability of further wins in the war.

It is expected to successfully attack the big city only if the person who knows secret enemy information is alive in the city to be saved.
Phase 1 – Save a city from attack

```
abds([save/1, big_city/0, small_city/0]).
on_going_commitment([save(_)]).
expect(save(_)).

% Save a city from attack
on_observe(save_place) <- be_attacked.
save_place <- save(a). save_place <- save(b).

% Context-sensitive IC
on_observe(not save_atmost_one_ic) <- lack_of_resources.
save_atmost_one_ic <- save(a), save(b).
save_men(P) <- save(City), population(City, P).
alive(X) <- person(X), live_in(X, City), save(City).

% Facts
population(a, 1000). population(b, 2000).
person(john). live_in(john, a). knows(john, secret_inf).
```
Phase 2 – Make an counter-attack

on_observe(attack) <- good_opportunity.
attack <- big_city.
attack <- small_city.
expect(small_city).
expect(big_city) <- alive(Person),
                 knows(Person, secret_inf).
pr(win, 0.9)  <- big_city.
pr(win, 0.01) <- small_city.
Running the example

1) consult the main program:  
   `?- [metaAb].`

2) load example file:  
   `?- load('saving_city').`

3) update/1 is for single-step prospective agent:  
   `?- update(2).`

4) single-step prospective agent decision:  
   `?- timeStamps.`
   
   1 - [save(b)]
   2 - [small_city, save(b)]

5) reload:  
   `?- load('saving_city').`

6) look_ahead_and_update(Prefs, EndState) for multiple-step prospection
   (resulting big city instead of small city):
   `?- look_ahead_and_update([], 2).
   Final Plan: [ [save(a), not save(b)], [big_city, save(a)] ]`

7) reload:  
   `?- load('saving_city').`

8) whole evolution tree till state 2 - first evolution preferred:  
   `?- evolution_tree(2, T, []).`

   \[ T = [ [save(a), not save(b)], [big_city, save(a)] ],
   [ [save(a), not save(b)], [small_city, save(a)] ],
   [ [save(b), not save(a)], [small_city, save(b)] ] ].`
a posteriori vs. Evolution a posteriori

- **a posteriori** preference – single-step PA (bad general)

  \[ Ai << Aj <- \text{holds\_given}(\text{save\_men}(Ni), Ai), \]
  \[ \text{holds\_given}(\text{save\_men}(Nj), Aj), Ni > Nj \]

- Evolution **a posteriori** preference – multiple-step PA (good general)

  \[ Ei <<< Ej <- \text{holds\_in\_evol}(\text{pr}(\text{win}, Pi), Ei), \]
  \[ \text{holds\_in\_evol}(\text{pr}(\text{win}, Pj), Ej), Pi > Pj \]

- With the 1st preference the general can attack only the small city, while with the 2nd he can attack the big city.

- Agents can make more reasonable decisions with evolution **a posteriori** preferences.
Evolution history preferences

• Takes into account commitment history information from the evolutions.

• It can be quantitative (e.g. having maximal or minimal number of some type of commitment) or qualitative (e.g. time order of commitments along an evolution).

• Can be used *a priori* (upon finding possible evolutions) or *a posteriori* (upon interaction with users, if there is more than one evolution left).
Evolution history preferences – 2
Forms

• \text{max}(C): \text{find evolutions with maximal } \# \text{ of commitments to } C.

• \text{min}(C): \text{find evolutions with minimal } \# \text{ of commitments to } C.

• \text{greater}(C, N): \text{find evolutions having the } \# \text{ of commitments to } C \text{ greater than } N.

• \text{times}(C, N): \text{find evolutions having exactly } N \text{ commitments to } C.

• \text{smaller}(C, N): \text{find evolutions having } \# \text{ commitments to } C \text{ smaller than } N.

• \text{next}(C1,C2): \text{find evolutions with commitment } C1 \text{ next to } C2 \text{ in time.}

• \text{prec}(C1,C2): \text{find evolutions with commitment } C1 \text{ preceding } C2 \text{ in time.}
Evolution history preferences - 3

• Having applied all preferences, if there is still more than one possible evolution, an interaction mode with the user is turned on to ask for the user’s additional preferences in the form of a list.

• A preference standing before another takes priority. If some preference is not satisfied by any evolution the system will jump over it and apply the next one.
Evolution history preferences - example

John is a football fan, with a project to finish.

He must schedule his daily actions to finish it on time.

Each day he either works or relaxes.

He can relax by going to the beach, going to a movie, or watching football.

Since he is a football fan, whenever there is a football match on TV he chooses to relax only, by watching football.
Modelling the example - 1

\[ \text{abds([go\_to\_beach/0, go\_to\_movie/0, work/0, watch\_football/0])} \]

\[ \text{expect(go\_to\_beach).} \]
\[ \text{expect(go\_to\_movie).} \]
\[ \text{expect(work).} \]

\[ \text{on\_observe(everyday\_act).} \]
\[ \text{everyday\_act <- work.} \]
\[ \text{everyday\_act <- relax.} \]

\[ \text{relax <- go\_to\_beach.} \]
\[ \text{relax <- go\_to\_movie.} \]
\[ \text{relax <- watch\_football.} \]
Modelling the example - 2

• If there is football relax only, by watching football:

\[
\begin{align*}
\text{expect(watch\_football)} & \leftarrow \text{prolog(have\_football)}. \\
\text{expect\_not(go\_to\_beach)} & \leftarrow \text{prolog(have\_football)}. \\
\text{expect\_not(work)} & \leftarrow \text{prolog(have\_football)}. \\
\text{expect\_not(go\_to\_movie)} & \leftarrow \text{prolog(have\_football)}. \\
\end{align*}
\]

• There is football in the first and second states (prolog code):

\[
\text{have\_football} :- \text{current\_state(S)}, \text{member(S, [1,2])}.
\]
Evolution history preferences - *a priori* use

```prolog
on_observe(on_time).

on_time <- deadline(Deadline), project_work(Days),
           prolog(working_days(Deadline, Days)).

deadline(5).  project_work(2).
```

- Reserved `plan_pref/1`, `plan_ending/1` allow asserting *a priori* evolution history preferences and the necessary number of look ahead steps:

```prolog
working_days(Deadline, Days) :-
    assert(plan_pref(times(work, Days))),
    assert(plan_ending(Deadline)).
```

- At the beginning, the agent tentatively runs the active goals to collect all *a priori* evolution preferences and decide how many steps are needed to look ahead.
Evolution history preferences
user interaction mode - 1

There are 6 possible evolutions left:

E1= [[go_to_beach], [watch_football], [watch_football], [work], [work]]
E2= [[go_to_movie], [watch_football], [watch_football], [work], [work]]
E3= [[work], [watch_football], [watch_football], [go_to_beach], [work]]
E4= [[work], [watch_football], [watch_football], [go_to_movie], [work]]
E5= [[work], [watch_football], [watch_football], [work], [go_to_beach]]
E6= [[work], [watch_football], [watch_football], [work], [go_to_movie]]
User interaction mode - 2

• Since there are several possible evolutions the interaction mode is turned on.

• Suppose the agent prefers evolutions with maximal number of going to the beach, entering the list \([\text{max(} \text{go to beach})]\). Three possible evolutions E1, E3 and E5 remain.

• The agent is asked again. Suppose he likes going to the beach after watching football, thereby entering \([\text{next(} \text{watch football, go to beach})]\).

• The only possible evolution is now E3.
Running the example - 1

1) consult the main file:  
   ?- [metaAb].

2) load example file:  
   ?- load('football'). yes

3) There are no initial preferences and no given number of look ahead steps. 
   They are chosen a priori:  
   ?- look_ahead_and_update([], []).

Possible plans:

1 - [ [go_to_beach], [watch_football], [watch_football], [work], [work] ]
2 - [ [go_to_movie], [watch_football], [watch_football], [work], [work] ]
3 - [ [work], [watch_football], [watch_football], [watch_football], [go_to_beach], [work] ]
4 - [ [work], [watch_football], [watch_football], [watch_football], [go_to_movie], [work] ]
5 - [ [work], [watch_football], [watch_football], [watch_football], [work], [go_to_beach] ]
6 - [ [work], [watch_football], [watch_football], [watch_football], [work], [go_to_movie] ]
Running the example - 2

Do you want to select a plan? Give the corresponding number if yes, otherwise enter ‘no’: no.

Do you have any preferences? [max(go_to_beach)].

Remaining plans:

1 - [ [go_to_beach], [watch_football], [watch_football], [work], [work] ]
2 - [ [work], [watch_football], [watch_football], [go_to_beach], [work] ]
3 - [ [work], [watch_football], [watch_football], [work], [go_to_beach] ]

Last chance to select a plan. Give the corresponding number if yes, otherwise enter ‘no’: 1. % consider one-cycle interaction mode

[ [go_to_beach], [watch_football], [watch_football], [work], [work] ]
Implementation - 1

• Our system is implemented on top of NegAbdual: a combination of Abdual with constructive negation.

• Abdual consists of two modules:
  – Preprocessor adds dual rules to the original program, and
  – Meta-interpreter allows for top-down abductive querying, answering queries using added rules.
Evolving program with Abdual

• The preprocessor was changed to make code dynamic, to allow adding or removing rules. Facts are changed by asserting or retracting a set of transformation-compatible rules. This enables programs to evolve.

• Committed to abducibles are timestamped for later use, e.g. to
  – Model evolution-level preferences
  – Represent time-related predicates
  – Hypothetically return to the past

• Different kinds of commitment are treated differently:
  – Hard and ongoing are added as facts
  – Abducible for a hard one is removed from abducibles list
  – Temporary ones are just time stamped
Implementation – 3
Preferences

• *a priori*: implemented by using the transformation provided previously.

• *a posteriori*:

\[ A_i \ll A_j \leftarrow \text{holds\_given}(L_i, A_i), \text{holds\_given}(L_j, A_j) \]

  – \text{holds\_given}(L, A) checks if domain literal \( L \) is a true side-effect of \( A \), without further abduction.

  – Implemented by checking if the abductive solution for query \( L \) is included in \( A \).
Preferences (cont.)

• Evolution result a posteriori

\[ E_i \lll E_j \leftarrow \text{holds\_in\_evol}(L_i,E_i), \]
\[ \text{holds\_in\_evol}(L_j,E_j) \]

– \text{holds\_in\_evol}(L,E) follows the evolution \( E \) till the end of last cycle and then checks whether \( L \) is a true side-effect of the abductive solution there (similar to a posteriori case).
Conclusions

• We have enacted
  – Local preferences: \textit{a priori ; a posteriori}
  – Evolution-level preferences: at evolution result \textit{a posteriori ; along evolution history.}
  – Time-sensitive preferences
  – Context-sensitive ICs
  – Inevitable actions

• Also exhibited a number of realistic illustrative examples.
Future Work - 1

• Agent relaxation by setting a scale of priorities for active goals: agent can focus on the most important ones if cannot satisfy all
  – This is implemented by preferring between on_observe/1 predicates used for representing the active goals

• Agent relaxation by setting a scale of priorities for ICs.
  – Similarly implemented
Future Work - 2

• Embedding some heuristic search algorithm for searching the evolution tree, e.g. best-first search.

• Using multi-threading of XSB
  – Independent threads can evolve on their own and they can communicate with each other to decide whether some thread should be canceled or kept evolving, based on the search algorithm used.