Úpravy přednášky v letech 2011/12 a 2012/13 byly částečně podpořeny projektem CZ.2.17/3.1.00/33274, který je financován Evropským sociálním fondem a rozpočtem hlavního města Prahy.

Evropský sociální fond
Praha & EU: Investujeme do vaší budoucnosti

Human-like artificial creatures

2. Reactive planning

Cyril Brom

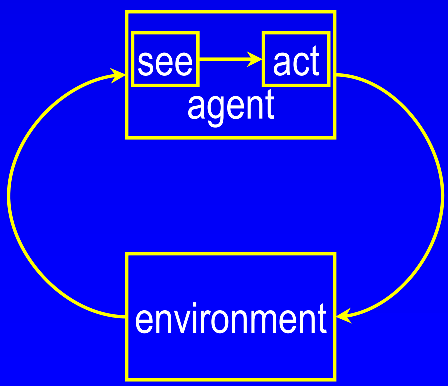
Faculty of Mathematics and Physics
Charles University in Prague
brom@ksvi.mff.cuni.cz

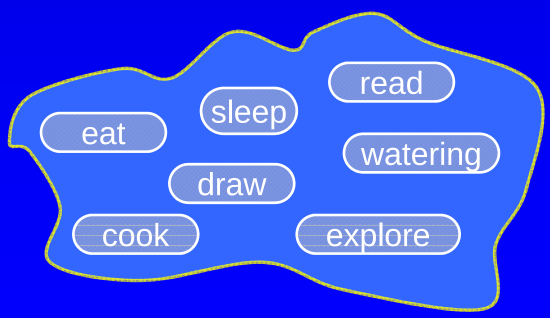
Outline

1. Recapitulation
   - action selection problem, artificial mind, architecture of a virtual being
2. Reactive planning
3. If-then rules
   - simple reactive planning
   - simple hierarchical reactive planning
   - limitations
4. Finite state machines
   - basic
   - hierarchical
   - probabilistic

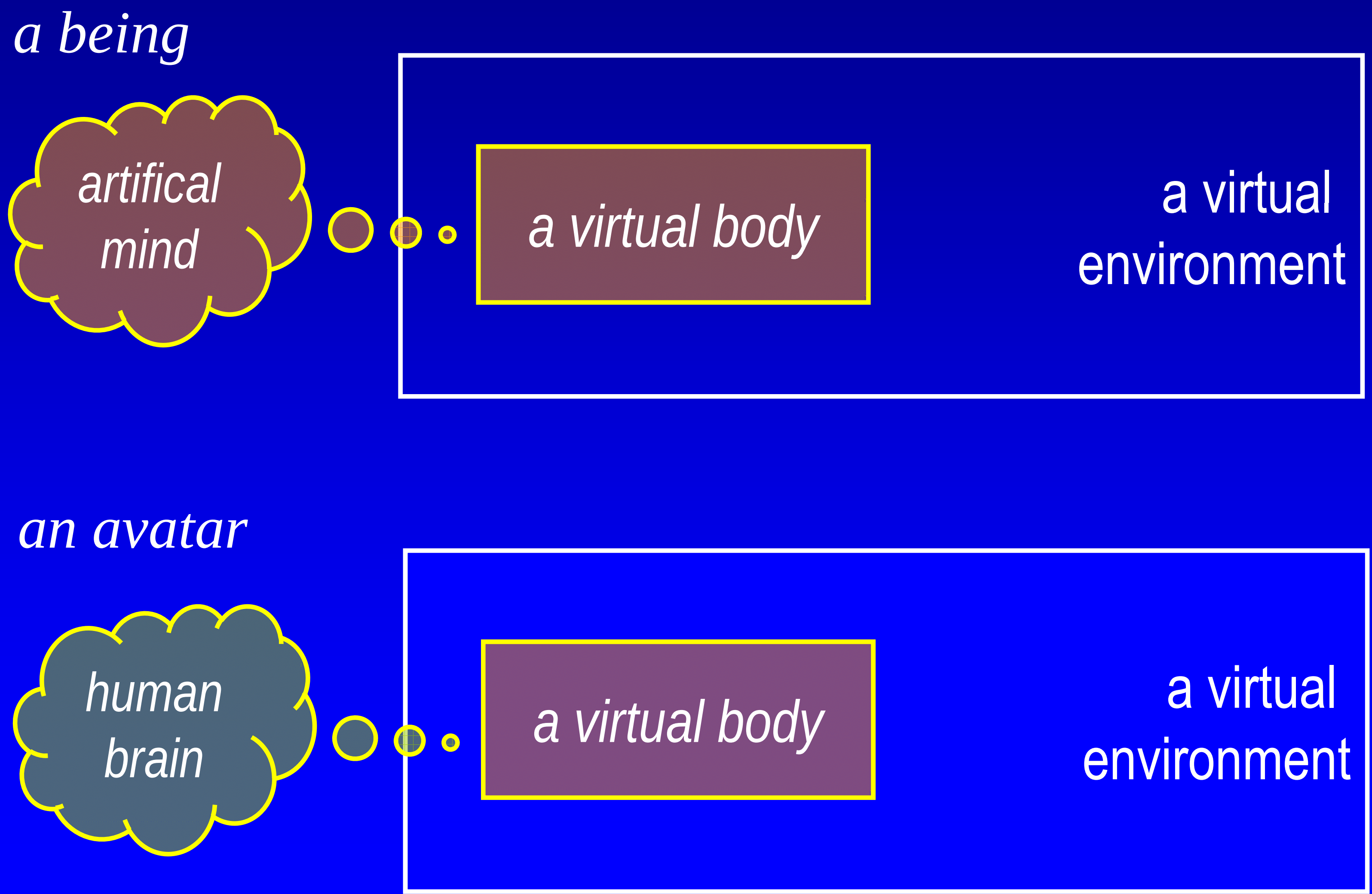
Action selection problem

- Artificial mind is a piece of code that decides "what to do next"
- The problem of deciding what to do next is called the action selection problem
- To decide what to do next, the creature must perceive its environment
- An action causes a change in the environment, and it usually has a feedback on the creature
- Typically, all possible actions are predefined

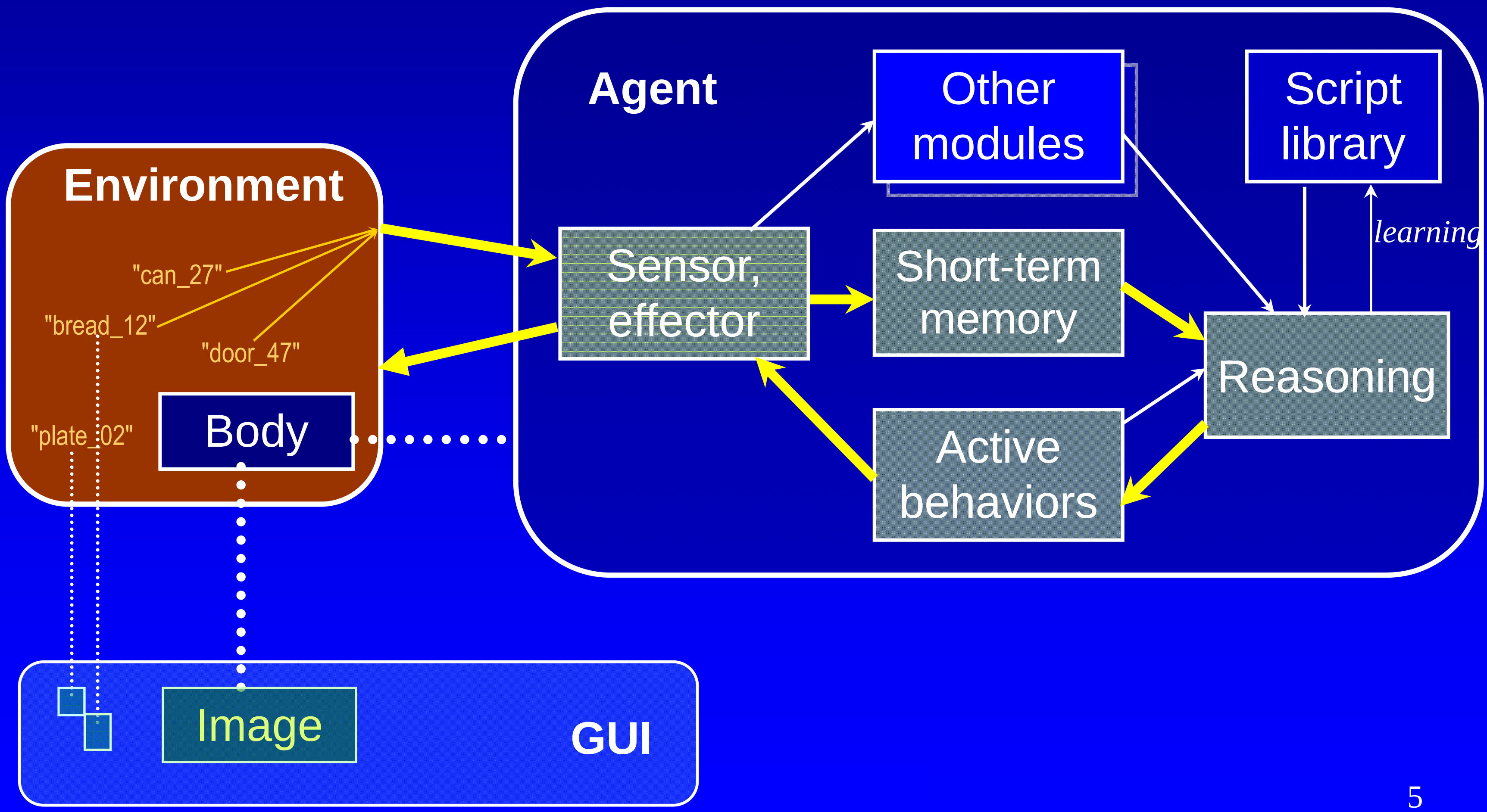




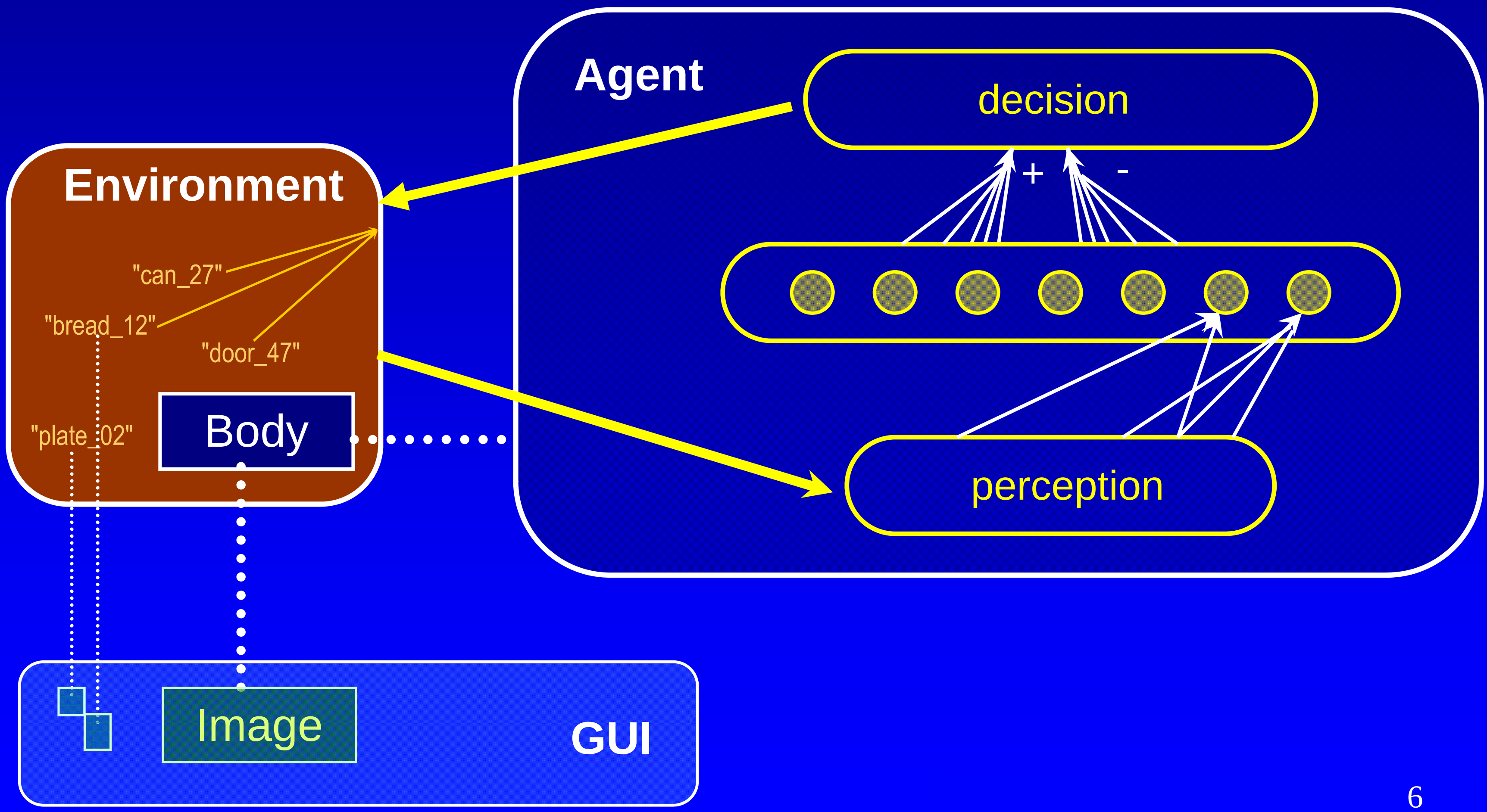
A virtual being vs. an avatar



Overall architecture of a symbolic beast



Overall architecture of a connectionist beast



An artificial environment

recapitulation

- accessible/inaccessible
  - an agent cannot obtain accurate up-to-date information about the whole environment
- deterministic/non-deterministic
  - the outcome of some actions is not uniquely defined
- static/dynamic
  - the environment changes in ways beyond the agent's control
- discrete/continuous in time/space:
  - finite number of discrete states is guaranteed
- real-time/step-based
  - the agent has theoretically infinite time to make a decision
- interactive/non-interactive
  - the user can alter the simulation

[Russell and Norwig, 1995]

Outline

1. Recapitulation
    - action selection problem, artificial mind, architecture of a virtual being
2. Reactive planning
3. If-then rules
    - simple reactive planning
    - simple hierarchical reactive planning
    - limitations
4. Finite state machines
    - basic
    - hierarchical
    - probabilistic

Reactive planning

- An approach to action selection problem
- Instead of calculating a plan in advance, the planner finds just the next action in every instant
- No unified definition

- „Reactive planning ... chooses only the immediate **next action**, and bases this choice on **the current context**. In most architectures utilizing this technique, reactive planning is facilitated by the presence of **reactive plans**. Reactive plans are stored structures which, given the current context, determine the next act.“

  [Bryson & Stein, 2000]
- The choice must be made in a "**timely fashion**"

Reactive planning

- A reactive planner realizes a function: $S \times P \longrightarrow A$
  - S – the set of all possible internal states (including memory)
  - P – the set of all possible actual percepts
  - A – the set of all possible actions
    - notice: A vs. P(A)

- Techniques
  - production rules
    - flat, hierarchical, heterarchical
  - finite state machines
  - fuzzy modifications, probabilistic modifications
  - free-flow hierarchies
  - neural networks
  - ...

Outline

1. Recapitulation
   - action selection problem, artificial mind, architecture of a virtual being
2. Reactive planning
3. <u>If-then rules</u>
   - simple reactive planning
   - simple hierarchical reactive planning
   - limitations
4. Finite state machines
   - basic
   - hierarchical
   - probabilistic
5. Conclusion

If-then rules

if p then A

a precondition, an antecedent

an action, an effect, a consequent...

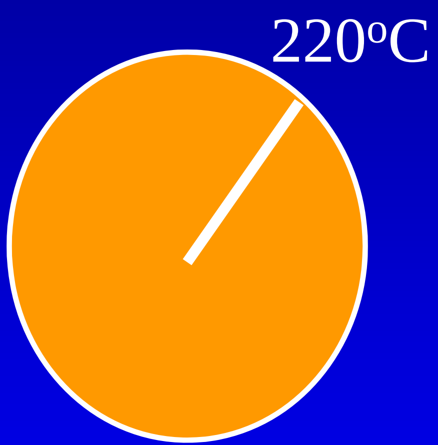
If-then rules

- A rule fires if its condition holds
- A reactive plan consist of tens of if-then rules
- All rules are "evaluated at once"
  - think in parallel!
- Technically, the parallelism must be "translated" to a serial program.

A thermostat

The regulator is set on 220^{o}C:

1. IF temperature > 225^{o}C,
   THEN switch the heater off.
2. IF temperature < 215^{o}C,
   THEN switch the heater on.



Why is the temperature tested for 225 / 215 instead of 220?

What to do when more rules fires in the same instant?

Simple reactive planning

- Assign a priority to each rule:

A robot picking up mushrooms:

```
# When starts: not at home && be in picking state
1. if see_obstacle then change_direction
2. if basketful_of_m. and picking then stop_picking
3. if see_mush. and picking then pick_up_the_mush.
4. if midday and picking then stop_picking
5. if home then END
6. if picking then move_random
7. if not_picking then move_home
```

What does the robot do when it sees a mushroom, but it is returning home?

subsumption architecture:
[Brooks, 1986; Wooldridge, 2002]

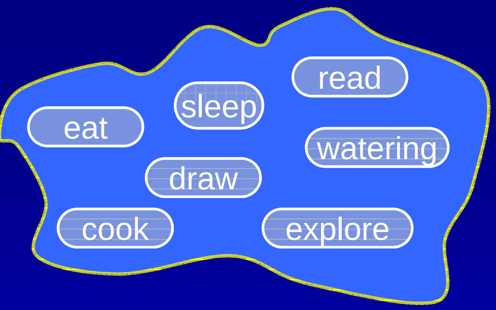
Simple hierarchical reactive planning

1. **if** bla1 **and** bla2 **then** SubGoal1
2. **if not** bla1 **and** bla3 **then** SubGoal2
3. **if** bla4 **then** SubGoal3
   - **3.1 if** A **then** Sub²GoalA
   - **3.2 if** B **then** Sub²GoalB
   - **3.3 if** C **then** Sub²GoalC
     - .
     - .
     - .
   - **3.4 if** D **then** Sub²GoalD
4. **if not** bla3 **and** bla2 **then** SubGoal4
5. **if** bla1 **and** bla3 and bla8 **then** SubGoal5
6. **if** blabla **then** SubGoal6
7. **if** bla2 **or (** bla3 **and not** bla7 **) then** SubGoal7

- Think hierarchically!

[Bryson, 2001; Nilsson, 1994; etc.]

Simple hierarchical reactive planning



- Behaviour is decomposed hierarchically
  - top-level goals, sub-goals, tasks, atomic actions
- Every reactive plan is expressed by means of a set of trees
- Every root of a tree corresponds to a top-level goal
  - AND trees, AND-OR trees
- How to create a decomposition?

Simple hierarchical reactive planning

(a hierarchical top-down decomposition)

Watering: goal: the garden is watered

Appetitive ⇄ Taxis ⇄ Consumatory → Clean

- **Appetitive**
  - Find & take a can
  - Fill the can
- **Taxis**
  - Go next to a dry bed
- **Consumatory**
  - Water the bed
- **Clean**
  - Empty the can
  - Put down the can

...cycles are possible!

...an ethology model

Simple hierarchical reactive planning

a decomposition example (watering)

- the highest priority has the goal condition, the second highest is the cleaning
- order the task in the normal/the reverse order [Bryson, 2001]

```
1.  if garden_watered and cleaned then COMMIT
2.  if garden_watered then subGoal_Clean
```

Clean

```
3.  if not_hold_any_can then subGoal_FindTakeCan
4.  if can_in_hands and empty then subGoal_FillUpTheCan
```

App.

```
5.  if know_about_dry_bed & not_stand_nextTo_theBed
    then subGoal_GoThere
```

Taxi

```
6.  if stand_nextTo_theBed and theBad_dry then
    atomicWatering
```

Cons.

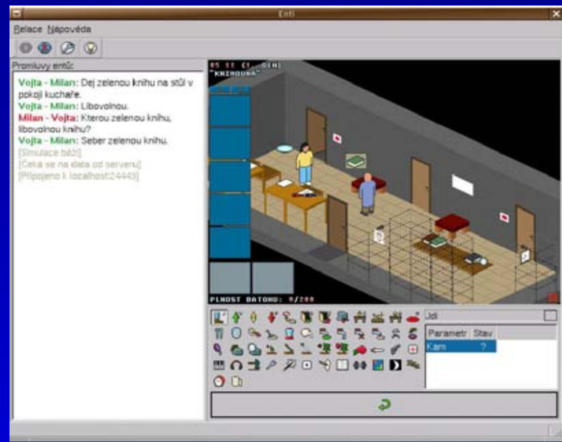
Simple hierarchical reactive planning

top-level goals

- How to select a top-level goal to perform?
  - a schedule + interrupts
  - drives + interrupts
  - a drama manager (Façade)
  - planning and future-directed intentions (BDI)

ENTs

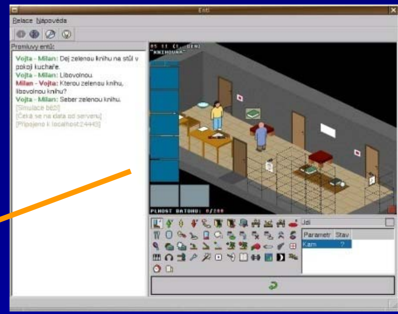
- Chess-like topology, 2½ D world
- Discrete time (time-steps)
  - a step = 20 sec.
- 20 internal drives
  - hunger, thirst,...
- 60 atomic actions
  - aWalk, aPickUp, aWater, aEat,...
- Two hands + an inventory
- Face no particular direction in the world
  - an illusion of orientation is caused by the GUI only
- Understand a simplified version of Czech language
- Driven by scripts in E language

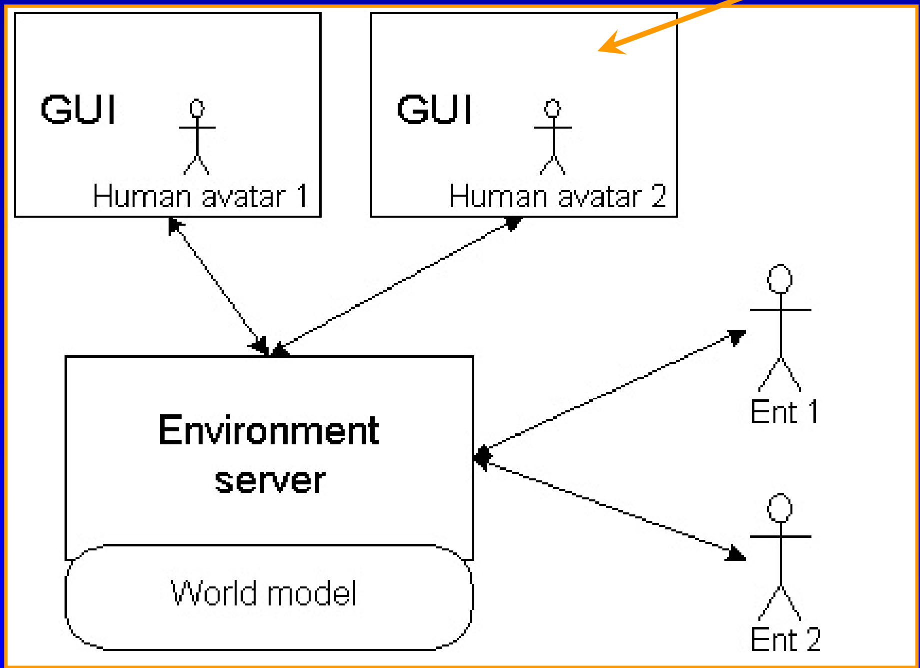


[Bojar et al., 2002; 2005]

ENTs

system architecture





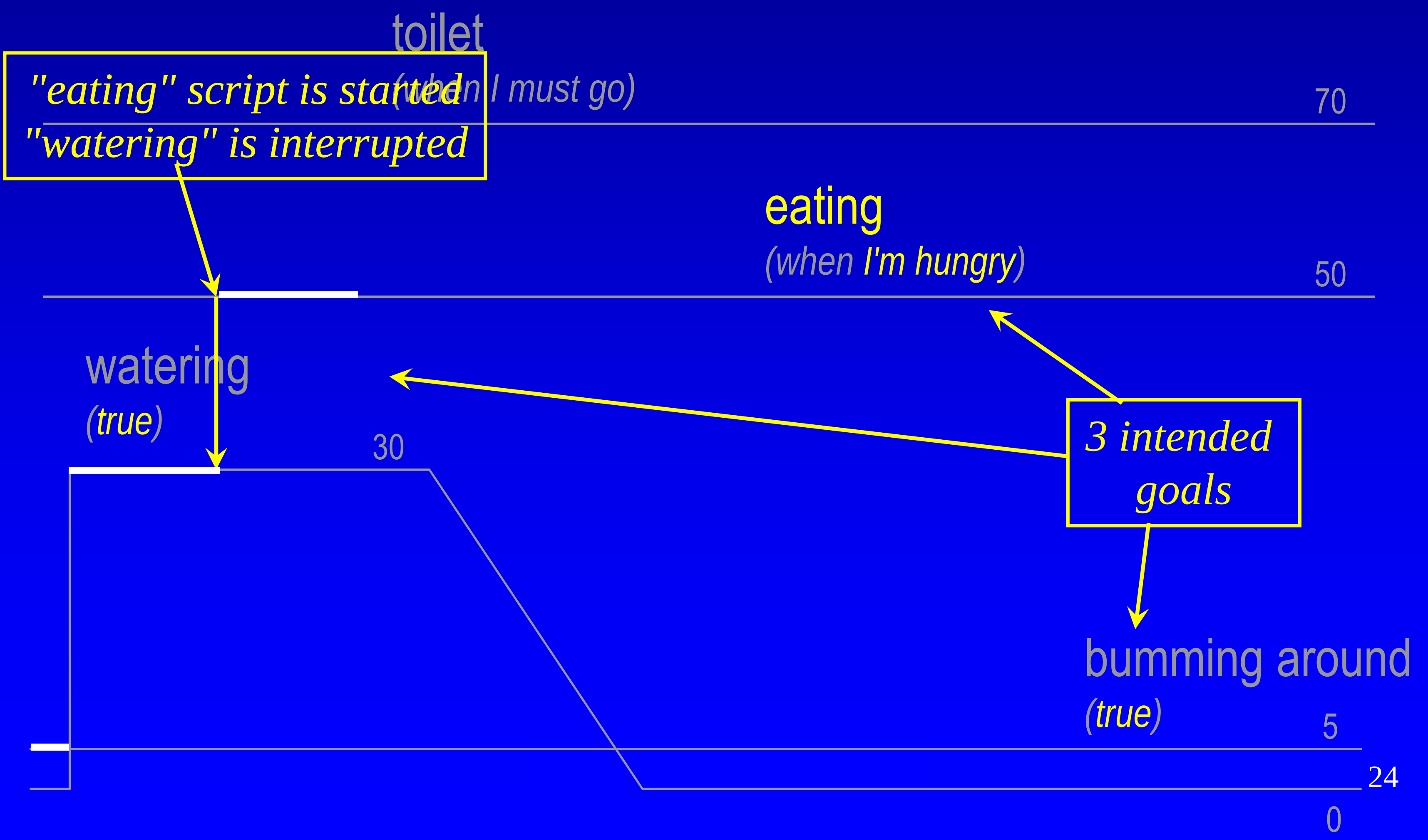
- 3 independent programs for Linux
  - entiserver (ES): the server of a virtual world
  - entiprohlizec: the graphical user interface
  - ent: the ent's control program (artificial mind)
- It is possible to instantiate different world models
  - we will use a model of a family house

Top-level goals

Four intended top-level goals of the gardener…

What is on the top?

Three active goals



What is on the top?

Bumming around

Reactive planner in action
PyPOSH in Unreal Tournament

Unreal

Gamebots
API

PyPOSH

PyPOSH
creature

[IGN Entertainment,
1996-2006]

[Adobatti et al., 2000]

[Kwong, 2003]

POSH - control structure I

- Action pattern
  - a sequence of actions that cannot be interrupted
  - e.g., "baa" and look at it (sheep)
- A competence: { *s*; *s* is a competence step }
  - steps that can be performed in different orders (i.e., a set of sequences)
  - one of the steps can be a goal step
  - the competence returns a value: $\top$ if the goal is accomplished, $\perp$ if none of its steps fire
  - a competence step: <p, r, a, [n]>
    - a priority, a releaser, an action, a number of retries
    - the action can be another competence

[Bryson, 2001]

POSH - control structure II

- A drive collection: { *d*; *d* is a drive element }
  - the root of the hierarchy
  - a drive element: <p, r, a, A, [f]>
    - *p* – a priority
    - *r* – a releaser
    - *a* – a currently active element of the drive element (a sub-element)
    - *A* – the top element (i.e., a collection, action pattern, or an action) of the drive element $\rightarrow$ slip-stack
    - *f* – a maximum frequency at which this drive element is visited
      - e.g., jump every five seconds
  - for any cycle of the action selection, only the drive collection itself and at most one other POSH element will have their releasers examined
- One drive element can suspend temporarily another drive element
  - a competence step cannot interrupt another competence step
- When the suspending drive element terminates, the suspended drive element continues

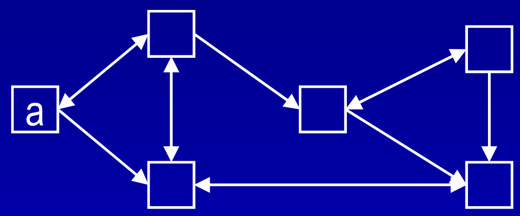
PyPOSH

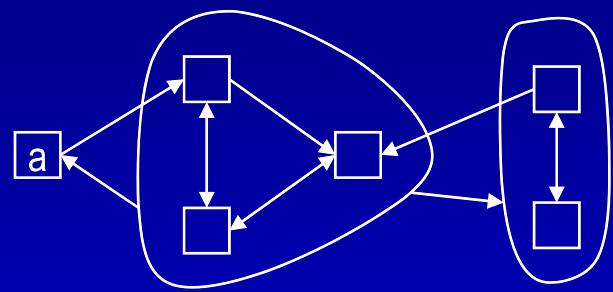
```
(RDC life (goal( (fail) ) )
    ( drives
        (( hit( trigger( (hit-object) (is-rotating False) ) ) avoid ))
        (( follow( trigger( (see-player) ) ) follow-player ))
        (( wander( trigger( (succeed) ) ) wander-around ))
    ) )
```

Outline

1. Recapitulation
   - action selection problem, artificial mind, architecture of a virtual being
2. Reactive planning
3. If-then rules
   - simple reactive planning
   - simple hierarchical reactive planning
   - limitations
4. Finite state machines
   - basic
   - hierarchical

FSM & HFSM (1)





Standard "finite-state machine" (FSM) is a tuple:

< { <label, T, script> }, a >

- <label, T, script> is a *state*
  - a *label* is a name of the state
  - a *script* is a code associated with the state
  - *T* is a set of rules that trigger transition to another state (i.e. transition function)
- *a* is a currently active state

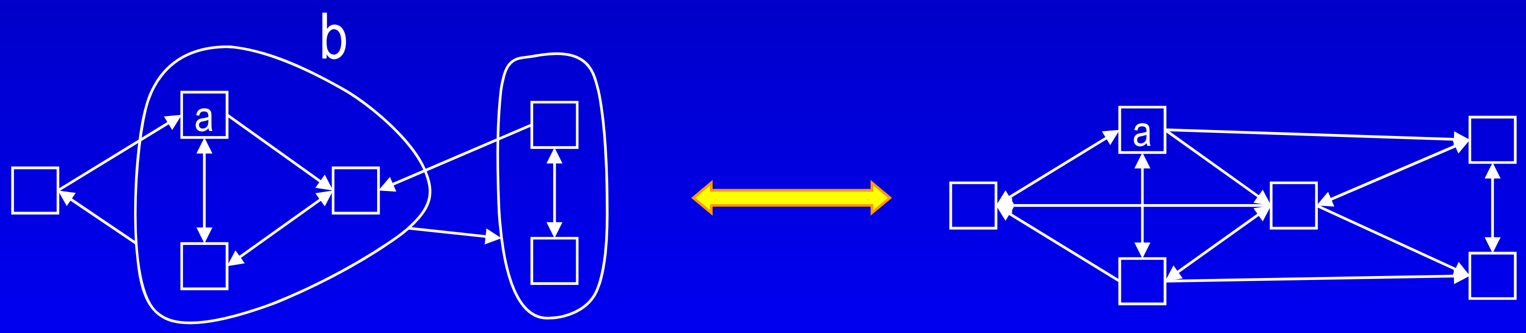
Hierarchical "finite-state machine" (HFSM) is a tuple:

< { <label, T, sc> }, A >

- <label, T, sc> is a *state*
  - a *label* is a name of the state
  - a *sc* is either a code associated with the state (i.e. a *script*), or a set of the names of the state's substates
  - *T* is a set of rules that trigger transition to another state (i.e. transition function)
- *A* is a set of currently active states
  - a path from a root-state to a leaf-state

FSM & HFSM (2)

- FSM and HFSM are computationally equivalent
  - HFSM avoids "spaghetti design"



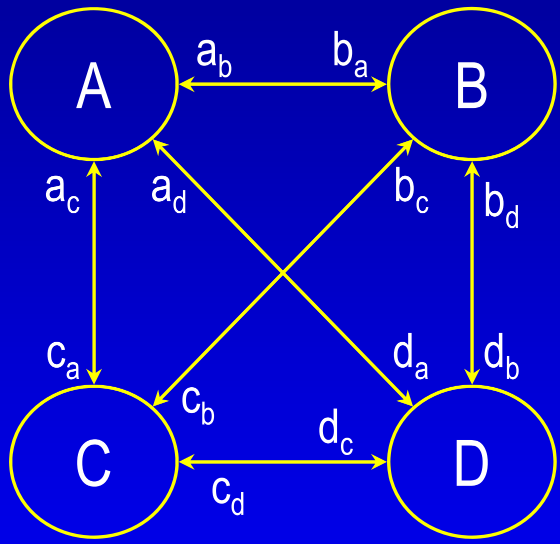
[Isla, 2005]

[Champandard, 2003]

Are finite state machines computationally equivalent to Touring machines?

SRP vs. FSM

```
1.  if ac or bc or dc then C
2.  if ab or cb or db then B
3.  if ba or ca or da then A
4.  if ad or bd or cd then D
```



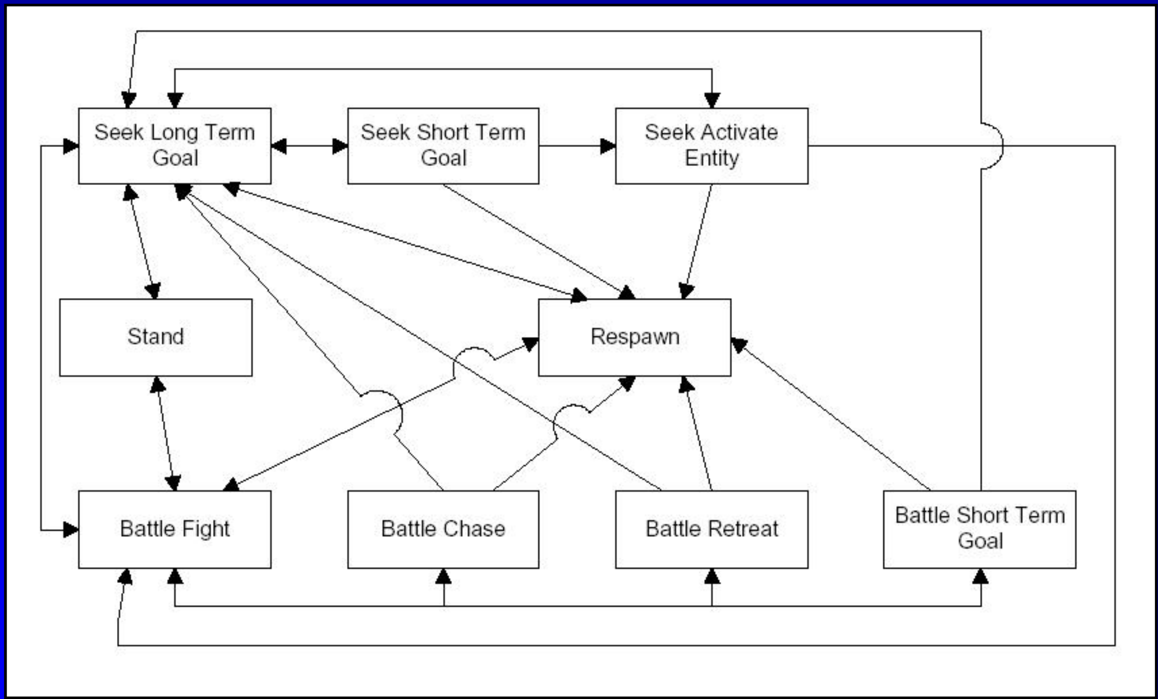
a note: z_x also tests whether the FSM is in state *Z*

- priorities
- "spaghetti design"

FSM example

Quake bot

- High level decision control only
- In each FSM-node, a bot chooses among possible goals associated with the node
- Standard FSM
- The if-then rules "in each node" are written in C



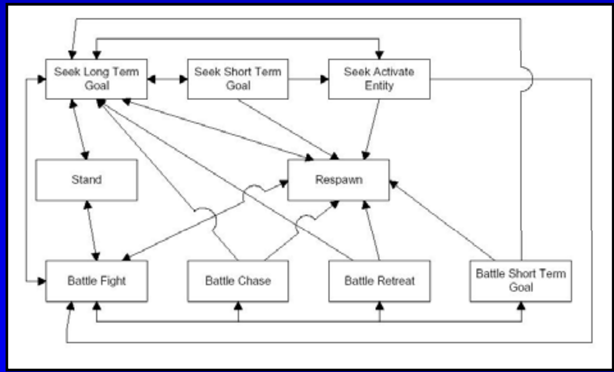
van Waveren (c) 2001

[van Waveren, 2001]

HFSM example

Quake bot

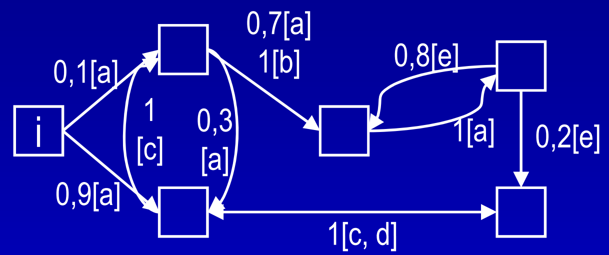
- In each FSM-node, a bot chooses among possible goals associated with the node
  - fuzzy decision (how much do I want to pick this weapon up?)
  - long term-goals vs. short term goals
- E.g. "battle fight":
  - acquiring enemy
  - selecting weapon
  - aiming and approaching
  - shooting
- Different techniques can be used in each node
  - low-level navigation
  - voting system
  - planning



van Waveren (c) 2001

Time (seconds)	Event or decision	Current AI node	Current goal
18.1	The bot named Grunt enters the game.	Stand	-
	Bot spawns.	Stand	-
		Seek LTG	
	Bot decides to retrieve item.	Seek LTG	Retrieve rocket launcher
	Bot decides to retrieve nearby item.	Seek LTG	Retrieve rocket launcher
		Seek NBG	Retrieve bullets
19.9	Picked up bullets.	Seek NBG	Retrieve bullets
		Seek LTG	Retrieve rocket launcher
20.6	Bot decides to retrieve nearby item.	Seek LTG	Retrieve rocket launcher
		Seek NBG	Retrieve shotgun
21.5	Enemy in sight.	Seek NBG	Retrieve shotgun
		Battle NBG	Kill the enemy & retrieve shotgun.
22.7	Picked up shotgun & bot wants to retreat.	Battle NBG	Kill the enemy & retrieve shotgun.
		Battle Retreat	Retreat & retrieve rocket launcher.
23.8	Bot decides to retrieve nearby item.	Battle Retreat	Retreat & retrieve rocket launcher.
		Battle NBG	Retrieve armor shard.
25.5	Picked up armor shard.	Battle NBG	Retrieve armor shard.
	Enemy out of sight & bot decides to chase.	Battle Retreat	Retreat & retrieve rocket launcher.
		Battle Chase	Chase enemy.
25.9	Bot decides to retrieve nearby item.	Battle Chase	Chase enemy.
		Battle NBG	Retrieve armor shard.
28.2	Picked up armor shard.	Battle NBG	Retrieve armor shard.
		Battle Chase	Chase enemy.
31.9	Enemy in sight.	Battle Chase	Chase enemy.
		Battle Fight	Kill the enemy.
32.3	Enemy out of sight.	Battle Fight	Kill the enemy.
		Battle Chase	Chase the enemy.
33.4	Enemy in sight.	Battle Chase	Chase the enemy.
		Battle Fight	Kill the enemy.
33.5	Enemy out of sight.	Battle Fight	Kill the enemy.
		Battle Chase	Chase the enemy.
35.4	Enemy in sight.	Battle Chase	Chase the enemy.

Probabilistic FSM models



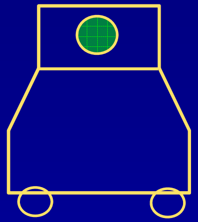
- Probabilistic "finite-state machine" (PFSM) is a tuple:
  < { <label, T^p, script> }, a >
- <label, T^p, script> is a *state*
  - a *label* is a name of the state
  - a *script* is a code associated with the state
  - T^p is a set of rules that trigger a transition to another state with a given probability
- *a* is the currently active state

Reactive planning - recapitulation

Recapitulation

- Reactive planning is a group of methods of driving behaviour of virtual beings
- Each method determines the next action in every instant in "a timely fashion"
- SHRP
  - if-then rules
  - priorities
  - AND-OR trees
- FSM
  - states
  - transitions

Implementation



- Special-purpose languages:
  - rules
    - JAM [Hubber, 1999]
    - E [Bojar et al., 2002]
    - PyPOSH [Kwong, 2003]
    - ABL [Mateas, 2002]
    - (Soar)
  - FSM
    - AI. Implant...
    - Softimage

rationale:

```
step
if someone-shoot-at-me do { .. }
if someone-asked-me do { .. }
if I-am-hungry do { .. }
if I-need-toilet do { .. }
if I-am-sleepy do { .. }
step
if someone-shoot-at-me do { .. }
if someone-asked-me do { .. }
if I-am-hungry do { .. }
if I-need-toilet do { .. }
if I-am-sleepy do { .. }
pick-up-mark
if someone-shoot-at-me do { .. }
if someone-asked-me do { .. }
if I-am-hungry do { .. }
if I-need-toilet do { .. }
if I-am-sleepy do { .. }
pick-up-mark
if someone-shoot-at-me do { .. }
...
```

Simple hierarchical reactive planning
problems

- Failures → AS memory
- Perceptual aliasing problem → AS memory [Brooks]
- Transition → hard-coding, if-then + FSM [Sengers]
- It behaves in the same way → probabilistic approach
- Compromise action → free-flow, voting [Tyrrell, 1993]
- Proscription → negative links, networks
- Modification of a behavior → metaparameters, floating priorities?
- Integrating concurrent behavior → modifieres [Blumberg, 1995]
- Interleaving → classical planning
- Sharp timeout → BDI, fuzzy, perceptual motivation ???

- Authoring vs. Learning
  - perform a task in a new situation
  - learn a new task
  - adapt a task to a modified situation
  - how long to try to perform a task

End

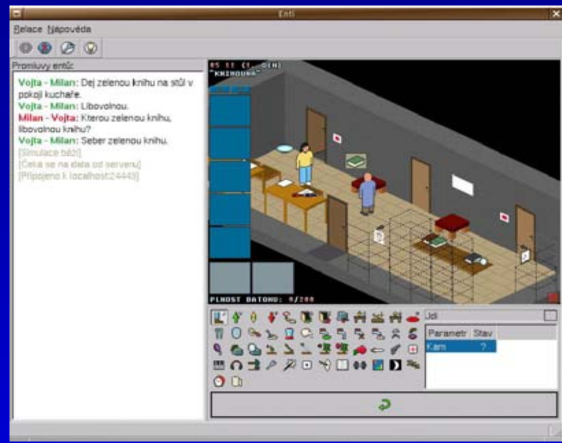
Implementation

- How exactly does it work?
  - it depends on the implementation...
- Special-purpose languages
- "Emulation"...

ENTs

→ an example

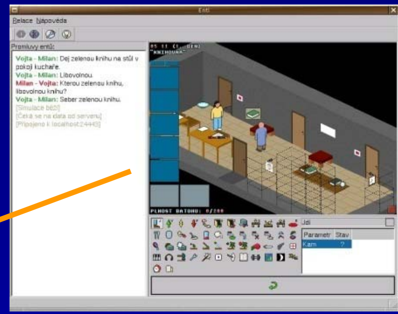
- Chess-like topology, 2½ D world
- Discrete time (time-steps)
  - a step = 20 sec.
- Embodied
- 20 internal drives
  - hunger, thirst,...
- 60 atomic actions
  - aWalk, aPickUp, aWater, aEat,...
- Two hands + an inventory
- Face no particular direction in the world
  - an illusion of orientation is caused by the GUI only
- Understand a simplified version of Czech language
- Driven by scripts in E language

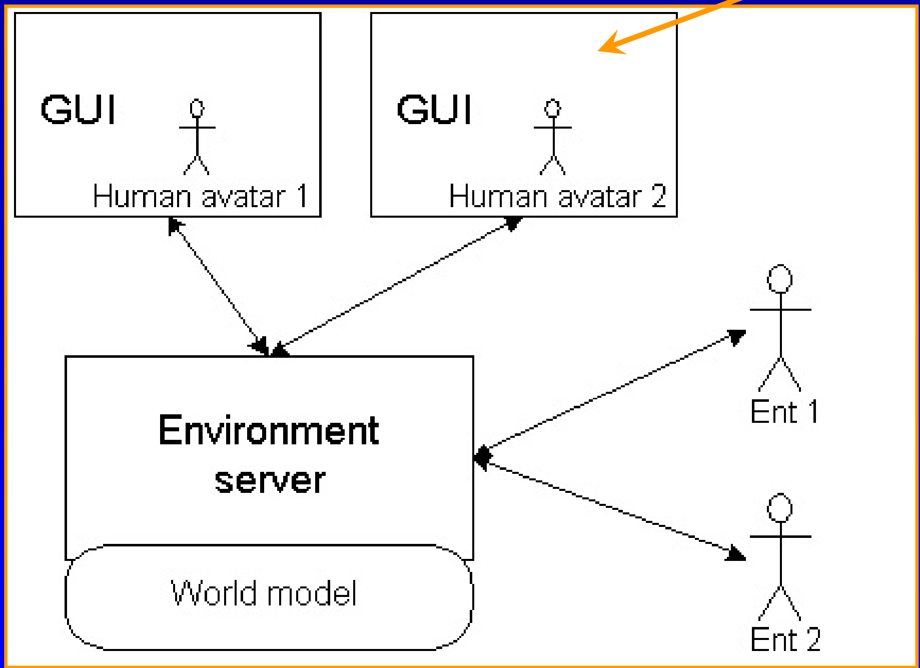


[Bojar et al., 2002; 2005]

ENTs

system architecture





- 3 independent programs for Linux
  - entiserver (ES): the server of a virtual world
  - entiprohlizec: the graphical user interface
  - ent: the ent's control program (artificial mind)
- It is possible to instantiate different world models
  - we will use a model of a family house

ENTs – the control cycle
one time-step

1. Every ent sends one a-action to the ES at the beginning of a time-step
2. ES waits till all a-actions are sent
3. ES computes the result of the time-step
4. Every ent receives “a world Δupdate” at the end of the time step

How to instruct an ent?

Watering a garden

- A simple behavioral script (b-script) in E language:

```
waterTheBedByTheCanOnce( hBed, hCan ):-

  aWaterPlants( hBed, hCan )
  .
```

an atomic instruction

input parameters – variables

How to instruct an ent?
Watering a garden

- Watering is a continuous action, it lasts about 10 time steps!
- A b-script with a conditional cycle:

```
waterTheBedByTheCan( hBed, hCan ):-
   if state( hBed, "already watered" ) then
       COMMIT
   fi,
   aWaterPlants( hBed, hCan ),
   RERUN
   .
```

a memory query

it finishes the script

it runs the script once again

A memory

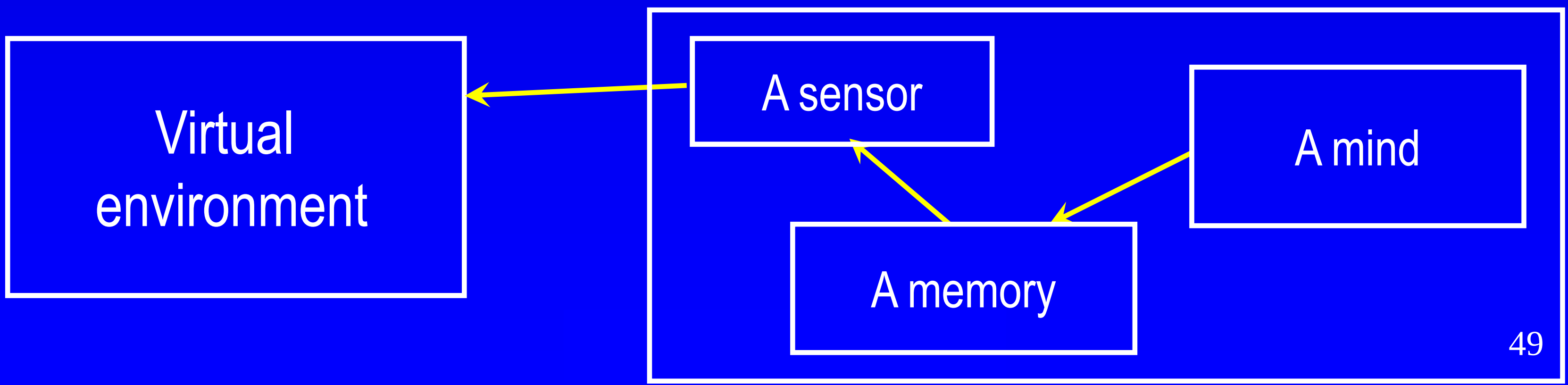
- The memory is a list of facts, e.g.:

`to_be_what_where_since( object, position, time )`

Can the ent look at the world-map directly?

No, because the ent is an autonomous being!

however, "cheating" may help a lot!



How to query the memory?

Find a dry bed

a general handle

```
waterTheBedByTheCan( hBed, hCan ):-
  query_ObjectsAnywhere( [ "object" = "bed" :
                           "room" = "garden" :
                           "special1" = "dry"],
                         [],
                         sListDryBeds
                       ) ,
```

a memory query

an output: a list of dry beds

EXIST x : d(x) ?
where d *means*
"a bed" &
"in the garden" &
"a dry object"

```
  returnTheClosestOne( hDryBed, sListDryBeds ) ,
```

an output

How to come next to the bed?

We need subgoaling…

- Assigning a subgoal

```
subGoal_goTo( hBed ) OR { ... },

aWaterPlants( hBed, hCan ),
```

a set of scripts that accomplish the subgoal

a script for a fail-case

Think hierarchically!

- Generally speaking, a task can be decomposed to subtasks recursively, until some atomic actions are reached.

How to come next to the bed?

We need subgoaling…

- Assigning a subgoal

a set of scripts that accomplish the subgoal

```
subGoal_goTo( hBed ) OR { ... },

aWaterPlants( hBed, hCan ),
```

a script for a fail-case

What should an ent do if someone begins watering the bed that the ent has just chosen? The bed might be already watered when the ent comes next to it!

- reactive planning!!!

A structure of a subgoal

- Every subgoal has:
  - prerequisites – a conjunction of atomic conditions that must hold before the subgoal is executed
  - a context – a conjunction of atomic conditions that must hold until the subgoal is accomplished
  - an effect – an expected result of the goal

planning background

How to test the context and prerequisites?

E language facilitates interrupts and conditions…

- An action can be triggered by activating an **interrupt**
- Prerequisites can be test by means of if-then condition

setting the interrupt

the local priority of the interrupt

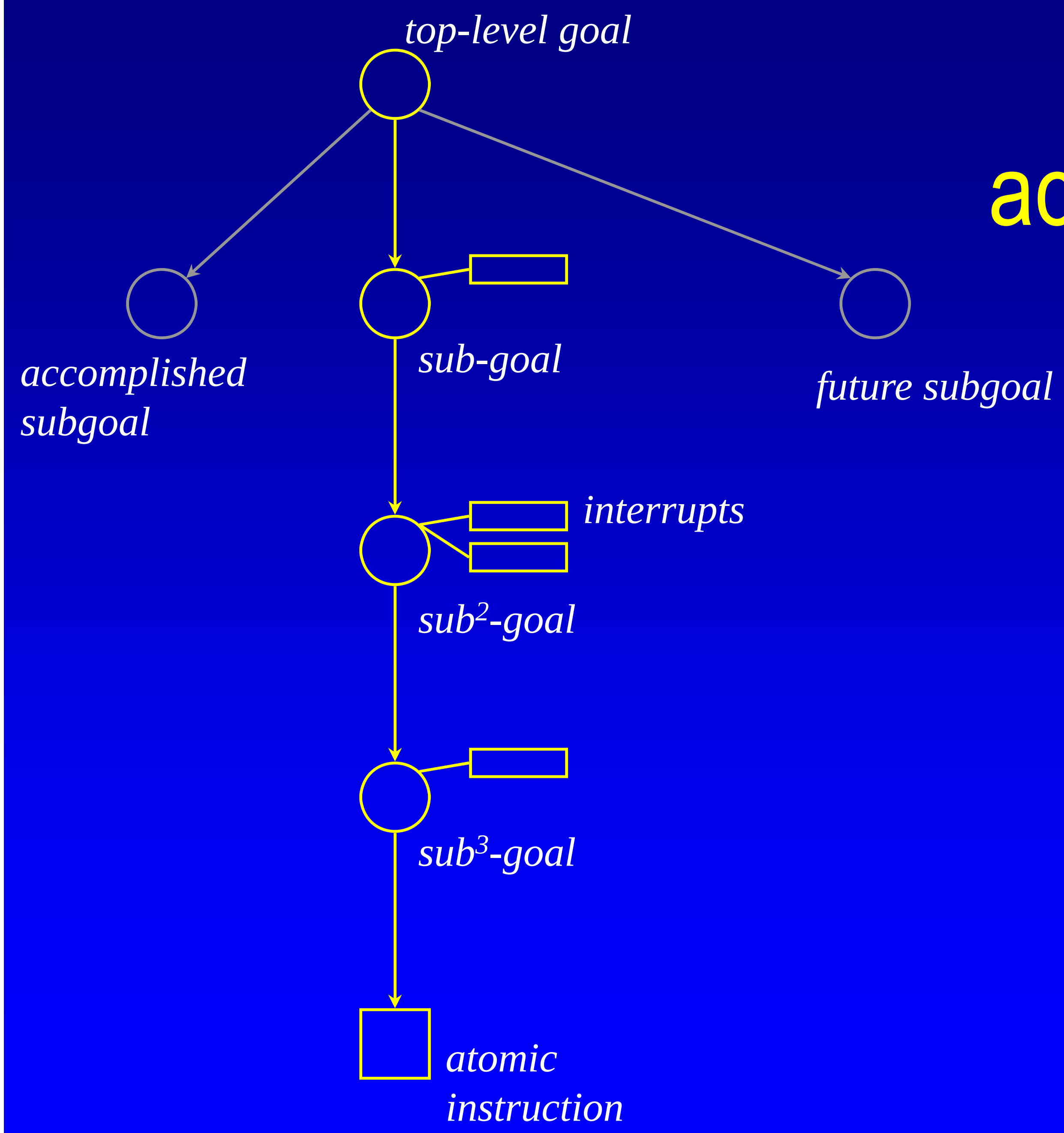
the trigger script

the id of the interrupt

```
if( not entNextTo( hBed )
{
  localHook( not state( hBed, "already watered" ),
             "PRIO_MAX",
             { ... },
             interruptNotWatered )
  subGoal_goTo( hBed ) OR { ... },
},
```

The tree of active subgoals

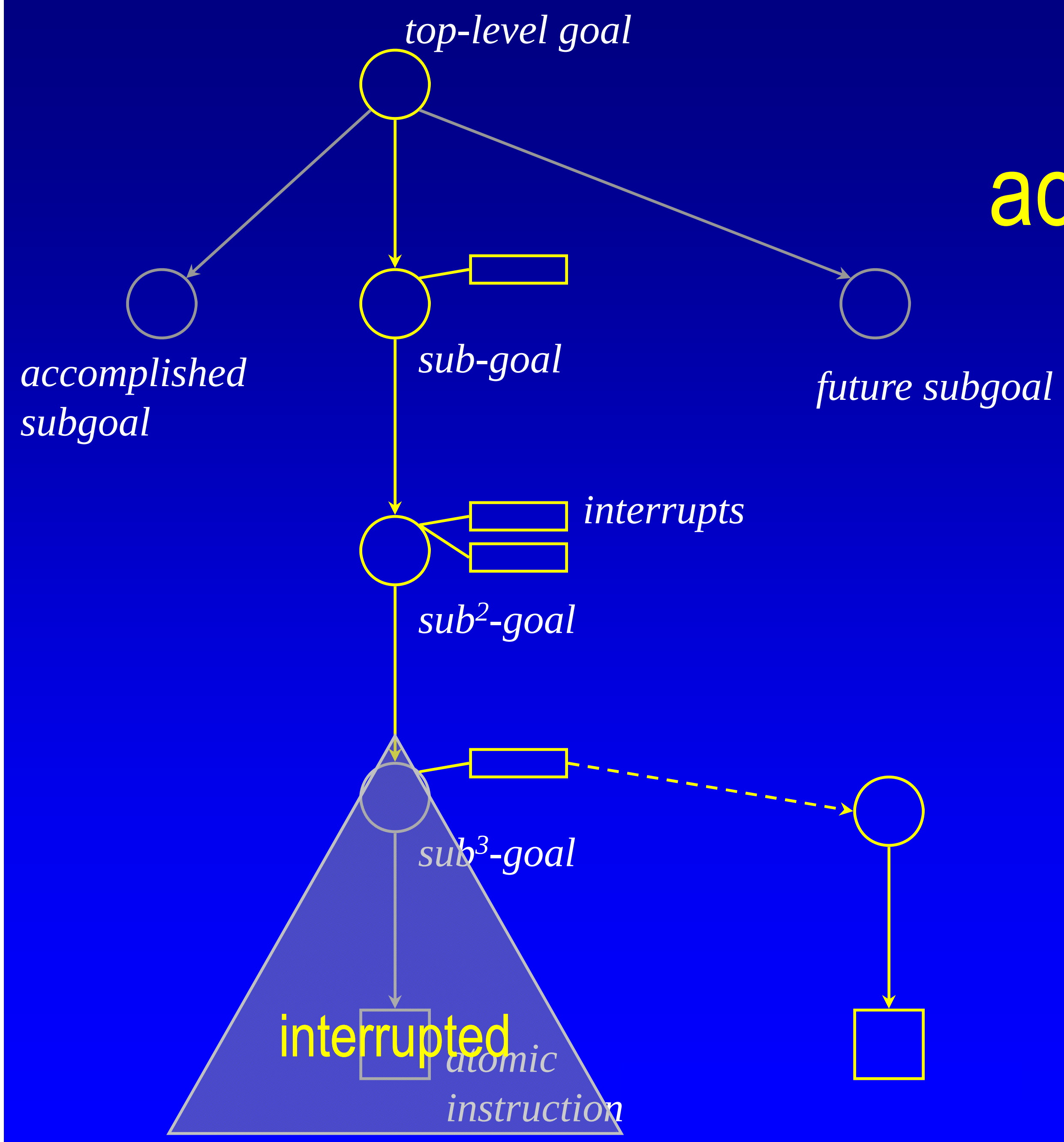
(in time t)



A stack of active subgoals

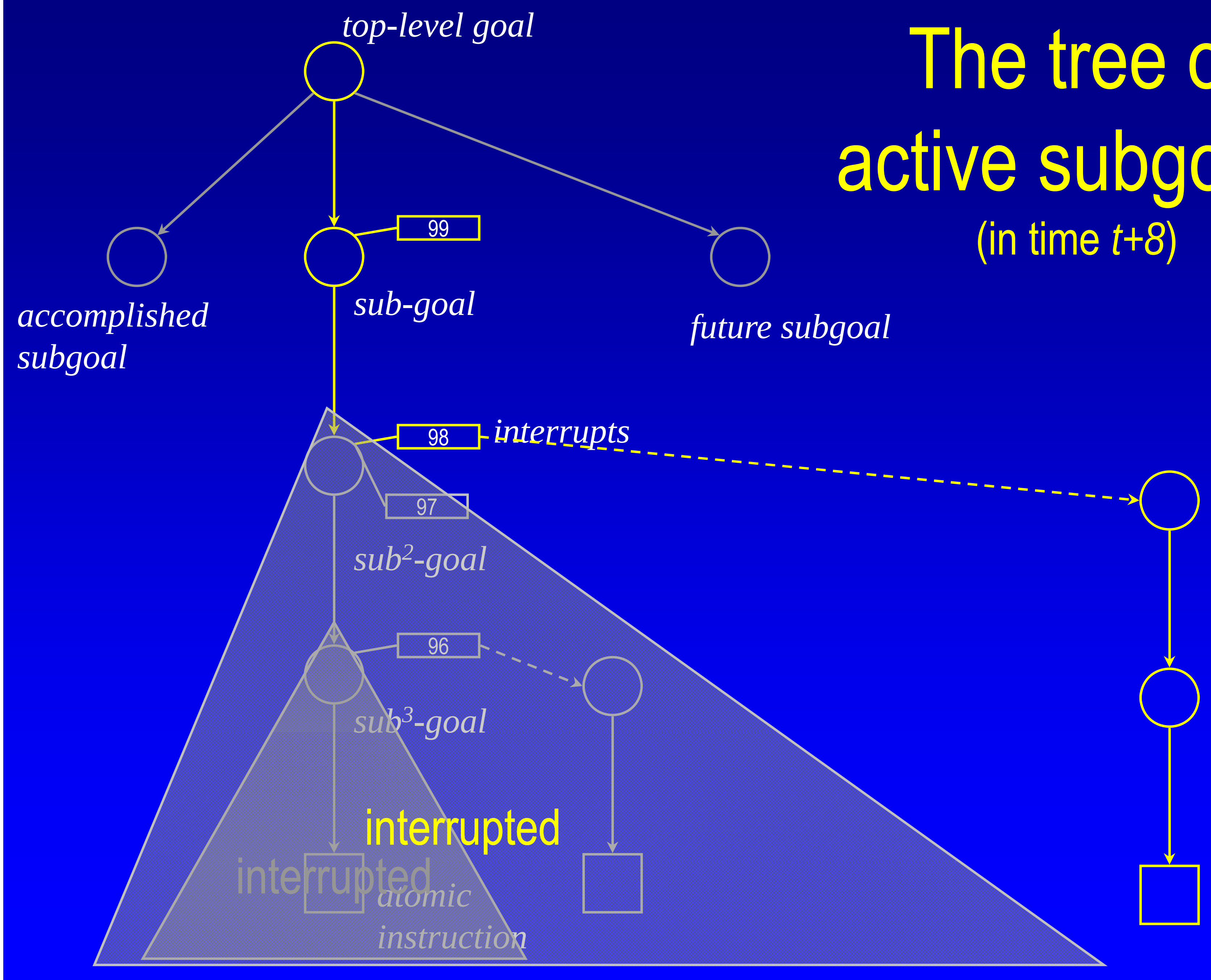
The tree of active subgoals

(in time t+5)



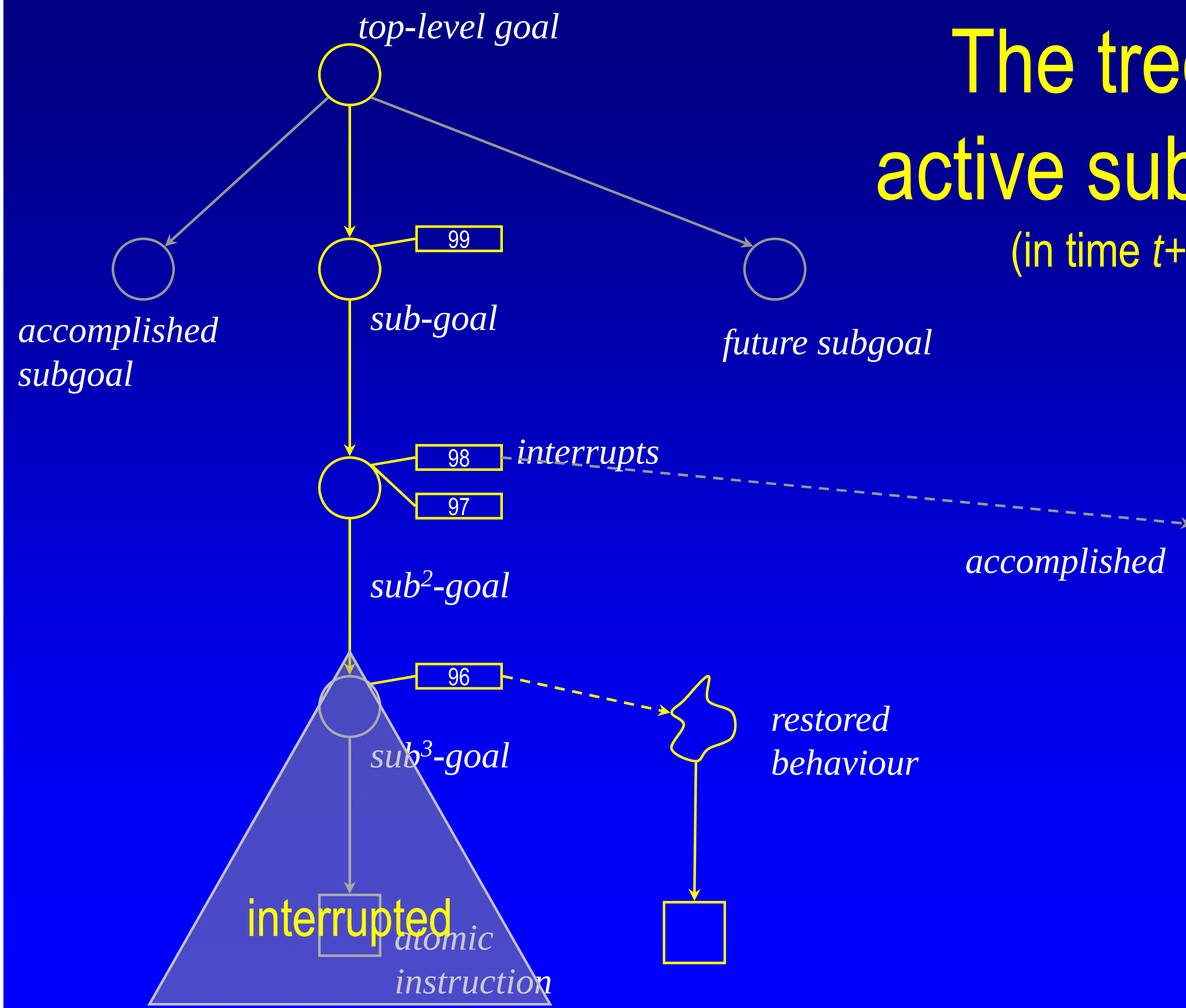
The tree of active subgoals

(in time *t*+8)



The tree of active subgoals

(in time $t+12$)



Hierarchical reactive planning in E

(a template)

```
top_levelGoal_WaterAllBeds :-
    // if everything is watered, try to put the can and commit
  if GOAL_COND then { try sgPutCan, COMMIT } fi,

    // if you are not holding a can, find it and take it; then activate the local interrupt that tests whether
        the can is still at hands -- if not, restart the watering
  if ! holdCan then sgFindAndTakeCan fi,
  localHook( ! holdCan, localPrioMax-1, { RERUN }, id1 ),

    // if  you are not holding an empty can, fill it; then activate the local interrupt that tests whether the
        can is not empty -- if it is, restart the watering
  if holdCan and canInHandEmpty then sgFillCan fi,
  localHook( holdCan and canInHandEmpty, localPrioMax-2, { RERUN }, id2 ),

    // the same follows for other subgoals...
```

- When an interrupt fires, restart the current script
- Perform the cleaning also as a transition

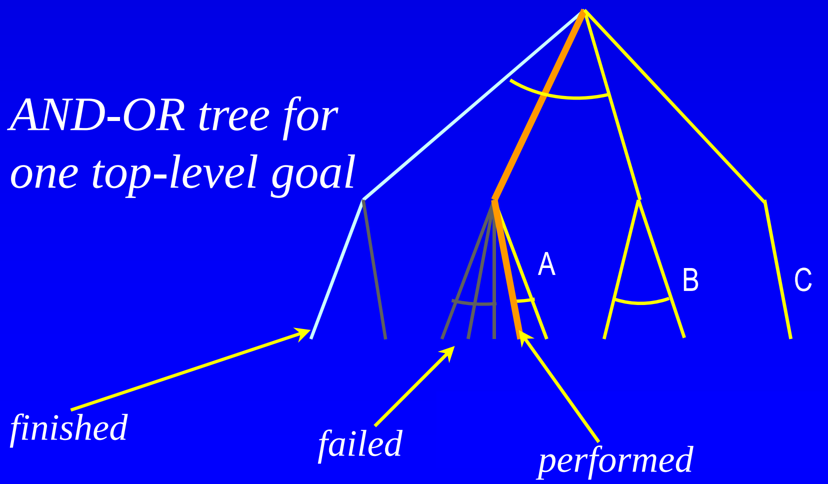
A subgoal is not a b-script
What is the difference?

- There may exist more ways of accomplishing a subgoal
- When the subgoal is instantiated, one b-script from a set of b-scripts is chosen to accomplish the task
  - a utility function
- If the b-script fails, another b-script is chosen
- The subgoal fails if all of its variants fail
- Remember: AND-OR trees vs. AND trees

```
subGoalEat

subGoalEat $-
  stateEnt( "hunger", hunger ),
  if hunger > 15 return 2
  else return 0 .

subGoalEat :-
  subGoalEatWhatever-
    FromTheFridge .
```

```
subGoalEat $-
  if lunchTime or DinnerTime
    return 1 .

subGoalEat :-
  subGoalEatInRoom .
```

a utility function



AND-OR tree for one top-level goal

What will be performed next in the case of a success/failure? A, B, C or nothing?

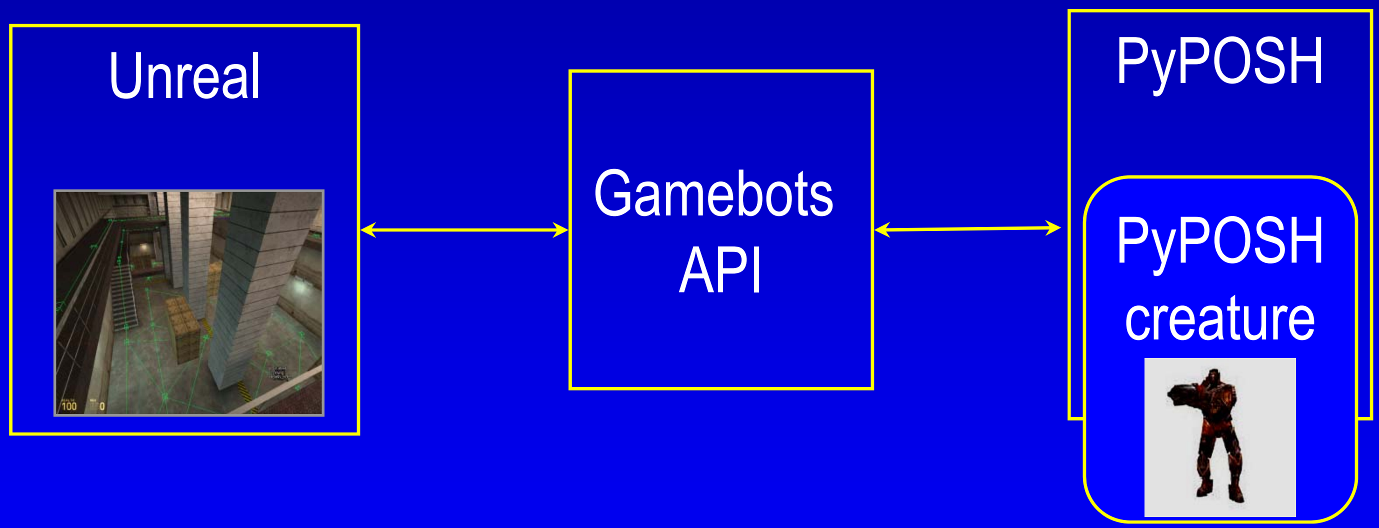
POSH

POSH & BOD

- Behavioural oriented design
  - behavioural decomposition
- POSH: Parallel-rooted, Ordered Slip-stack Hierarchical
  - a method that exploits hierarchical if-then rules
  - several languages
    - POSH: in lisp or C++
    - PyPOSH: Python implementation
    - jyPOSH: Jython implementation (interoperates with Java)

[Bryson et al., 2001 - 2006]

PyPOSH in Unreal - architecture

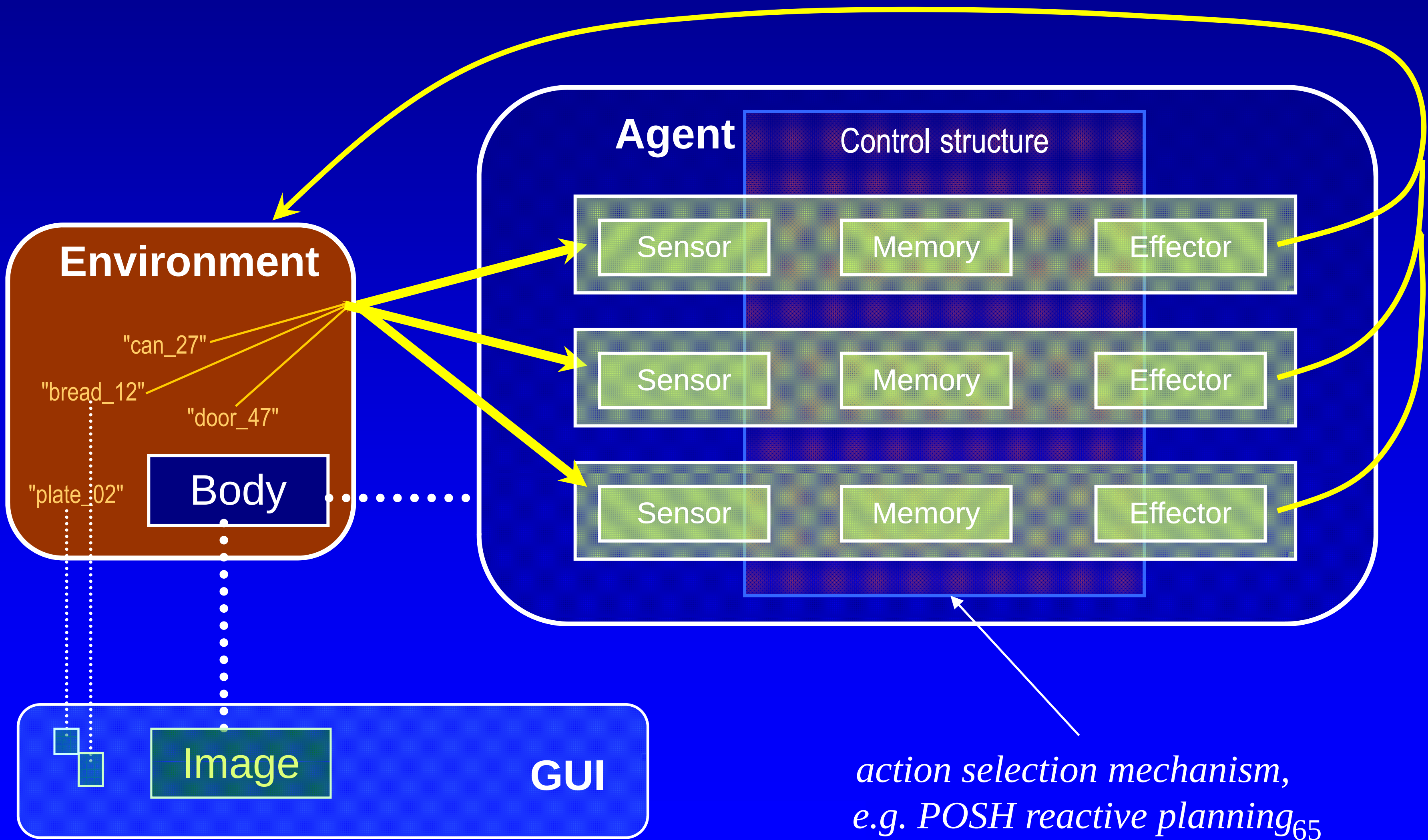


[IGN Entertainment, 1996-2006]

[Adobatti et al., 2000]

[Kwong, 2003]

Behavioural oriented creature



action selection mechanism,
e.g. POSH reactive planning

Behaviours as objects

- Object
  - properties/variables
  - methods
- Behaviour
  - states/variables (memory)
  - primitive elements of the reactive plan which present the interface to the behaviour
    - senses
    - acts
  - learning

POSH - control structure I

- Action pattern
  - a sequence of actions
  - e.g., "baa" and look at it (sheep)
- A competence: { *s*; *s* is a competence step }
  - steps that can be performed in different orders (i.e., a set of sequences)
  - one of the steps can be a goal step
  - the competence returns a value: $\top$ if the goal is accomplished, $\perp$ if none of its steps fire
  - a competence step: <p, r, a, [n]>
    - a priority, a releaser, an action, a number of retries
    - the action can be another competence

[Bryson, 2001]

POSH - control structure II

- A drive collection: { d; d is a drive element }
  - the root of the hierarchy
  - a drive element: <p, r, a, A, [f]>
    - *p* – a priority
    - *r* – a releaser
    - *a* – a currently active element of the drive element (a sub-element)
    - *A* – the top element (i.e., a collection, action pattern, or an action) of the drive element $\rightarrow$ slip-stack
    - *f* – a maximum frequency at which this drive element is visited
      - e.g., jump every five seconds
  - for any cycle of the action selection, only the drive collection itself and at most one other POSH element will have their releasers examined
- One drive element can suspend temporarily another drive element
  - a competence step cannot interrupt another competence step
- When the suspending drive element terminates, the suspended drive element continues

PyPOSH

```
def init_senses( self ):
    self.add_sense( "see-player", self.see_player )
    ...

def init_acts( self ):
    self.add_act( "move-player", self.move-player )
    ...

def see_player( self ):
    ...
```

Python

top-level

checking period

```
(RDC life (goal( (fail) ) )
    ( drives
prio: 1 (( hit( trigger( * (hit-object)(is-rotating False) ) ) avoid ))
      2 (( follow( trigger( (see-player) ) ) follow-player ))
      3 (( wander( trigger( (succeed) ) ) wander-around ))
    ) )
```

timeout condition *terminate condition*

```
(C wander-around (minutes 10) (goal( (see-player) ) )
    ( elements
        (( close-enough( trigger( (close-to-player) ) ) stop-bot ))
        (( move( trigger( (see-player) ) ) move-player ))
    ) )
```

if *then*

"Lisp"

PyPOSH

```
def init_senses( self ):
    self.add_sense( "see-player", self.see_player )
    ...

def init_acts( self ):
    self.add_act( "move-player", self.move-player )
    ...

def see_player( self ):
    ...


(RDC life (goal( (fail) ) )
    ( drives
        (( hit( trigger( (hit-object)(is-rotating False) ) ) avoid ))
        (( follow( trigger( (see-player) ) ) follow-player ))
        (( wander( trigger( (succeed) ) ) wander-around ))
    ) )

(C wander-around (minutes 10) (goal( (see-player) ) )
    ( elements
        (( close-enough( trigger( (close-to-player) ) ) stop-bot ))
        (( move( trigger( (see-player) ) ) move-player ))
    ) )
```

PyPOSH

```
def init_senses( self ):
    self.add_sense( "see-player", self.see_player )
    ...

def init_acts( self ):
    self.add_act( "move-player", self.move-player )
    ...

def see_player( self ):
    ...


(RDC life (goal( (fail) ) )
    ( drives
        (( hit( trigger( (hit-object)(is-rotating False) ) ) avoid ))
        (( follow( trigger( (see-player) ) ) follow-player ))
        (( wander( trigger( (succeed) ) ) wander-around ))
    ) )

(C wander-around (minutes 10) (goal( (see-player) ) )
    ( elements
        (( close-enough( trigger( (close-to-player) ) ) stop-bot ))
        (( move( trigger( (see-player) ) ) move-player ))
    ) )
```

PyPOSH

```
def init_senses( self ):
    self.add_sense( "see-player", self.see_player )
    ...

def init_acts( self ):
    self.add_act( "move-player", self.move-player )
    ...

def see_player( self ):
    ...


(RDC life (goal( (fail) ) )
    ( drives
        (( hit( trigger( (hit-object)(is-rotating False) ) ) avoid ))
        (( follow( trigger( (see-player) ) ) follow-player ))
        (( wander( trigger( (succeed) ) ) wander-around ))
    ) )

(C wander-around (minutes 10) (goal( (see-player) ) )
    ( elements
        (( close-enough( trigger( (close-to-player) ) ) stop-bot ))
        (( move( trigger( (see-player) ) ) move-player ))
    ) )
```

Evropský sociální fond
Praha & EU: Investujeme do vaší budoucnosti

Questions?

References

- BOD, POSH
  - Joanna Bryson. The Behavior-Oriented Design of Modular Agent Intelligence. In: *Proceedings of Agent Technologies, Infrastructures, Tools, and Applications for E-Services*, pages 61-79, Springer LNCS 2592, Berlin, Germany, 2003.
  - Kwong, A. *A Framework for Reactive Intelligence through Agile Component-Based Behaviours*. Master thesis, University of Bath (2003)
  - Joanna Bryson. *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents.* PhD thesis, Massachusetts Institute of Technology, 2001.
- Gamebots:
  - Adobbati, R., Marshall, A. N., Scholer, A., and Tejada, S.: Gamebots: A 3d virtual world test-bed for multi-agent research. In: Proceedings of the 2nd International Workshop on Infrastructure for Agents, MAS, and Scalable MAS, Canada (2001)
- ENTs
  - O. Bojar, C. Brom, M. Hladík, V. Toman: The Project ENTs: Towards Modeling Human-like Artificial Agents. In *SOFSEM 2005 Communications*, pages 111–122, Liptovský Ján, Slovak Republic, January 2005.
  - Project Ent homepage: `http://ckl.ms.mff.cuni.cz/~bojar/enti/`

References

- FSM
  - Waveren, J. M. P. van: The Quake III Arena Bot. Master thesis. Faculty ITS, University of Technology Delft (2001)
  - Champandard, A.J.: AI Game Development: Synthetic Creatures with learning and Reactive Behaviors. New Riders, USA (2003)
  - Softimage, Bahavior: http://www.softimage.com/products/behavior
- Façade, ABL
  - Mateas, M.: Interactive Drama, Art and Artificial Intelligence. Ph.D. Dissertation. Department of Computer Science, Carnegie Mellon University (2002)
- Other
  - Brooks, A. R.: Intelligence without reason. In: Proceedings of the 1991 International Joint Conference on Artificial Intelligence, Sydney (1991) 569-595
  - Huber, M. J.: JAM: A BDI-theoretic mobile agent architecture. In: Proceedings of the 3rd International Conference on Autonomous Agents (Agents'99). Seatle (1999) 236-243
  - Soar project: http://www.eecs.umich.edu/~soar/
  - Isla, D.: Handling Complexity in the Halo 2 AI. Game Developers Conference, GDC 2005, http://www.gamasutra.com/gdc2005/features/20050311/isla_01.shtml

References

- AI & agents
  - S. J. Russell and P. Norvig: *Artificial Intelligence: a Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ.
  - M. Wooldridge: *An Introduction to MultiAgent Systems*. John Wiley & Sons, 1995
- Other
  - Brooks, A. R.: Intelligence without reason. In: *Proceedings of the 1991 International Joint Conference on Artificial Intelligence*, Sydney (1991) 569-595
  - Huber, M. J.: JAM: A BDI-theoretic mobile agent architecture. In: *Proceedings of the 3rd International Conference on Autonomous Agents* (Agents'99). Seatle (1999) 236-243