

# **Executive Control**

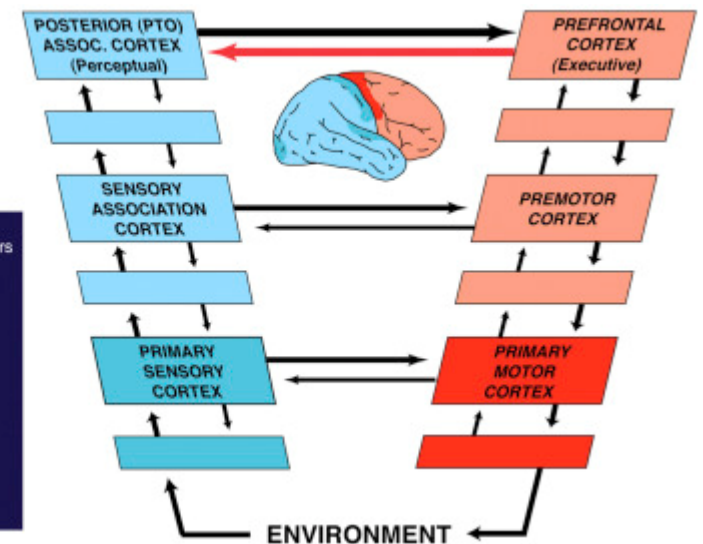
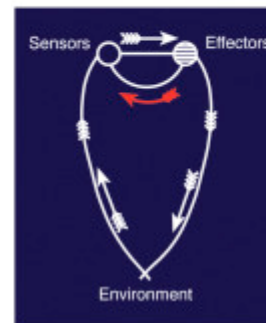
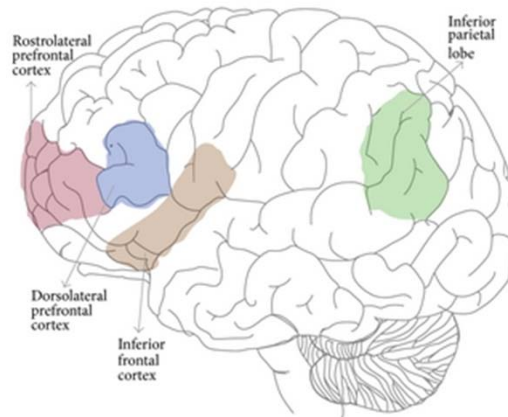
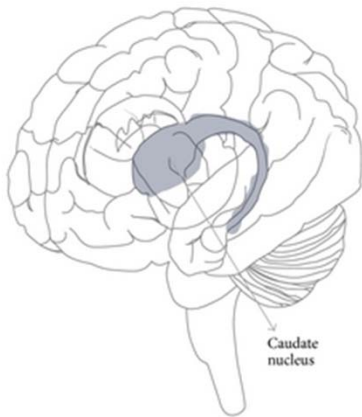
## Robot Architecture

# The Magic Middle: Executive Control

- The middle layer has a hard job:
  - 1) **compensate for the limitations** of both the planner and the reactive system
  - 2) reconcile their **different time-scales**.
  - 3) deal with their **different representations**.
  - 4) reconcile **any contradictory commands** between the two.
- This is **the challenge** of hybrid systems
  - => **achieving the right compromise between the two ends.**

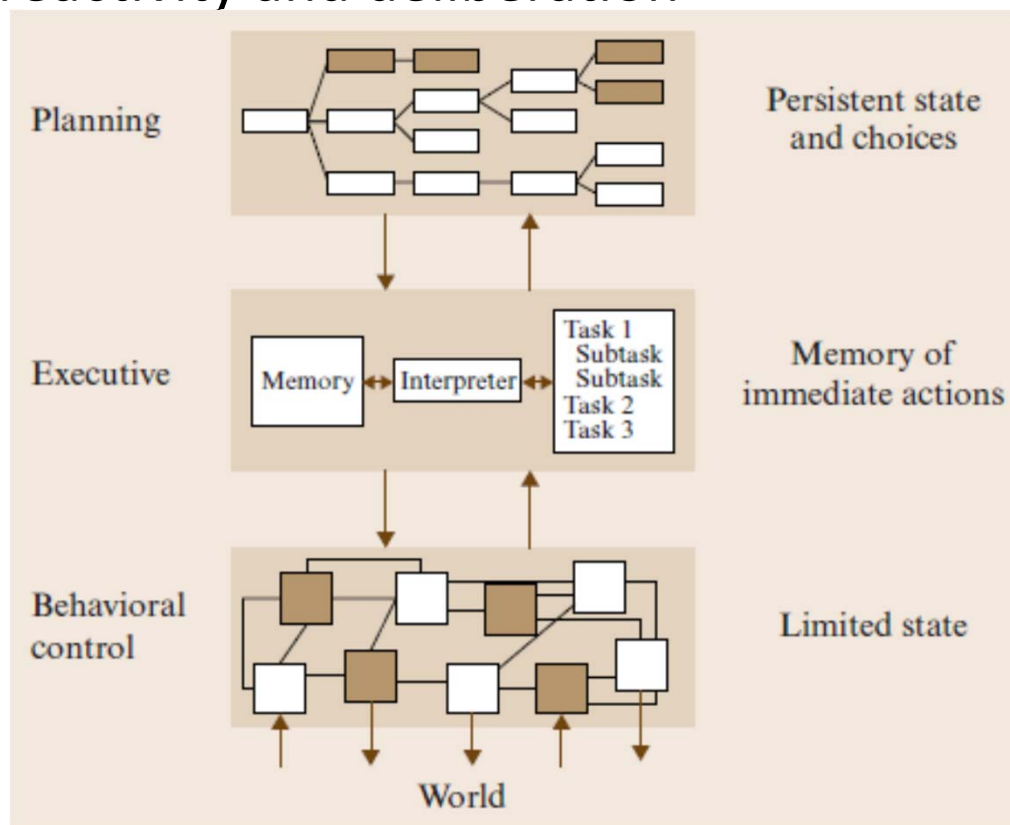
# The Magic Middle: Executive Control

- Executive Functions in humans
  - Flexible, goal-oriented execution
  - Novel situations, recover
  - Attention, Cognitive Control



# Integration

- Three layered architectures
- Integrate reactivity and deliberation



# Executive Frameworks



RAP (<http://people.cs.uchicago.edu/~firby/raps>)

Firby, J “Task Networks for Controlling Continuous Processes”, Proceedings of Artificial Intelligence Planning conference, 1994.



TCA (<http://www-2.cs.cmu.edu/afs/cs/project/TCA/release/tca.org.html>)

Simmons, R. “Structured Control for Autonomous Robots”, IEEE Transactions on Robotics and Automation, Feb 1994.



IRMA

Intelligent Resource-bounded Machine Architecture, Bartman, Israel, Pollack



PRS (<http://www.ai.sri.com/~prs>)

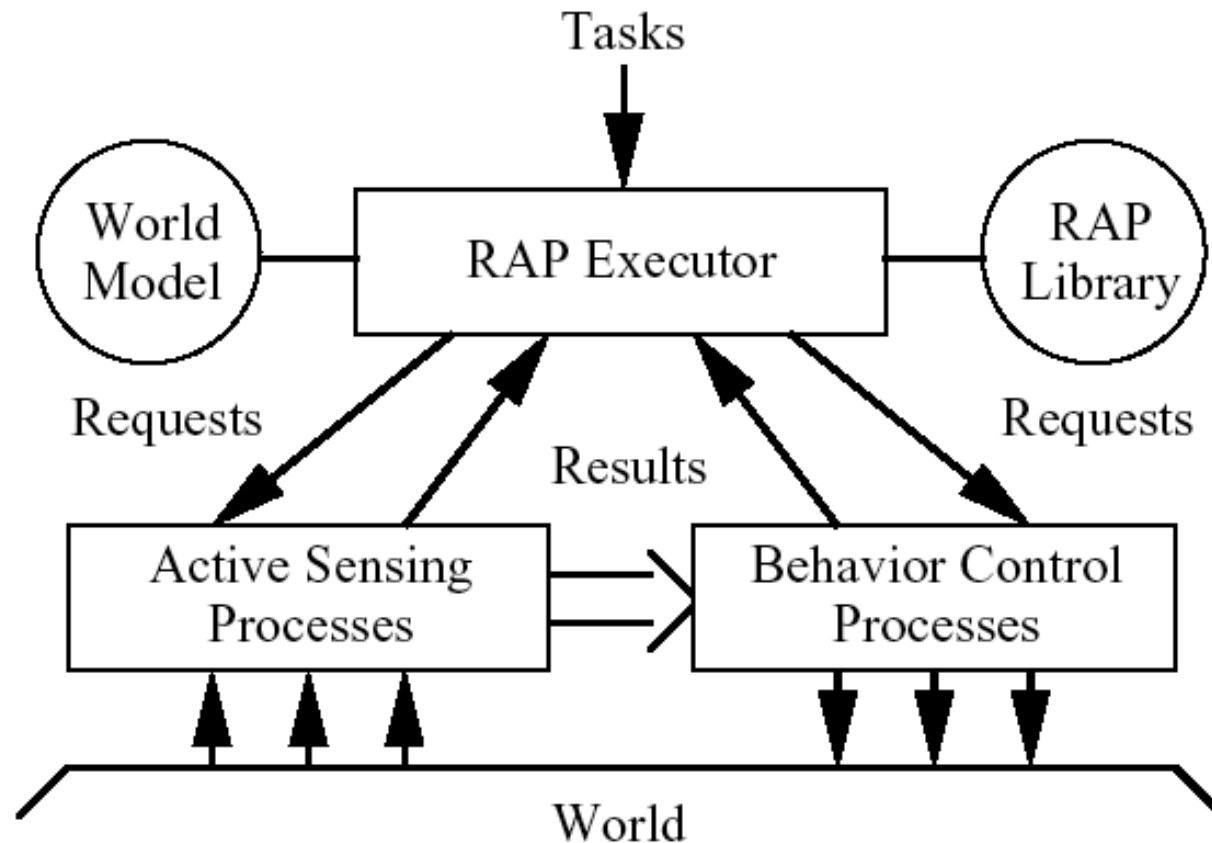
Reactive reasoning and planning: an experiment with a mobile robot, M. Georgeff and A. Lansky, in Proceedings of AAAI, 1987.



RMPL

Reactive Model-based Programming Language, Brian Williams

# Reactive Action Packages (RAP)



R.J. Firby: Adaptive Execution in Complex Dynamic Worlds. Ph.D. Thesis (Yale Univ., New Haven 1989)

R.J. Firby: Task Networks for Controlling Continuous Processes, Proc. of the Second International Conference on AI Planning Systems (1994)

# Reactive Actions Packages

To cope with unpredictable details:

Choose the appropriate plan completions at execution time.

- Instead of detailed plans, one should construct sketchy plans, with unspecified low level actions.
- The gaps in the plans are filled by RAPs execution system at run-time.

# What is a RAP?

- A RAP is a representation that groups together and describes all known ways to carry out a task in different situations.
- “Situation-driven execution”

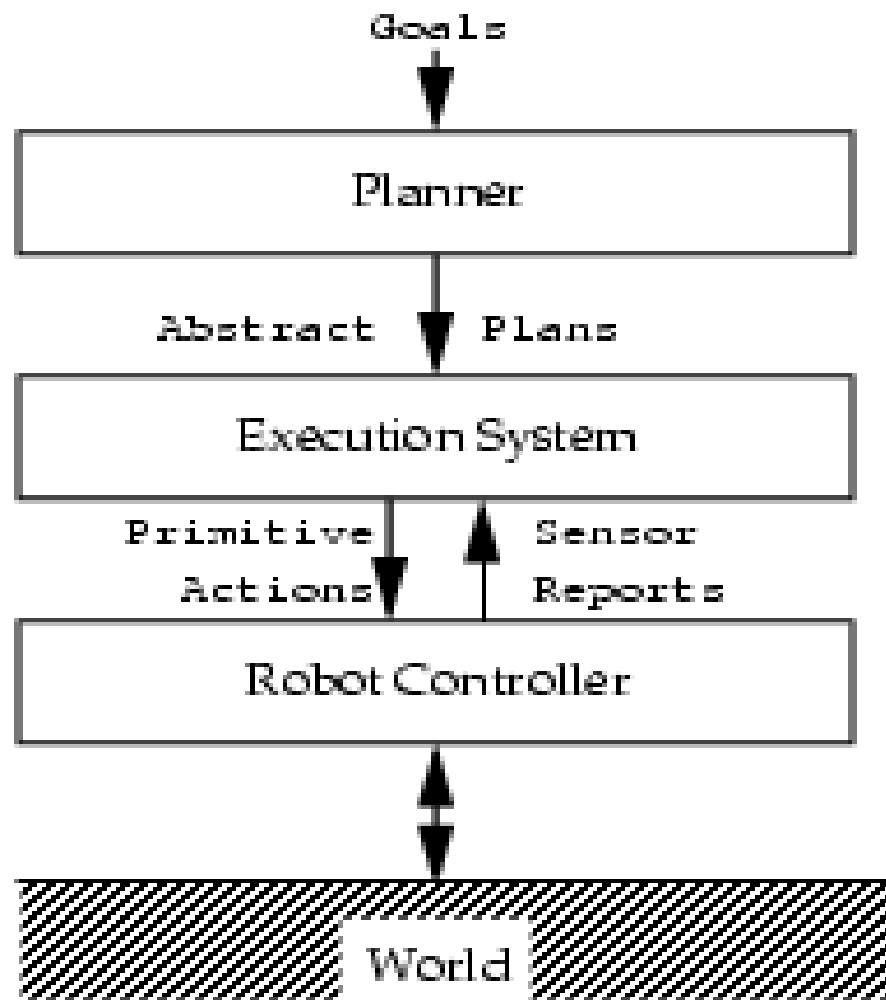


# Situation-driven execution

Tasks should have a:

- Satisfaction test
- Window of activity
- Set of execution methods

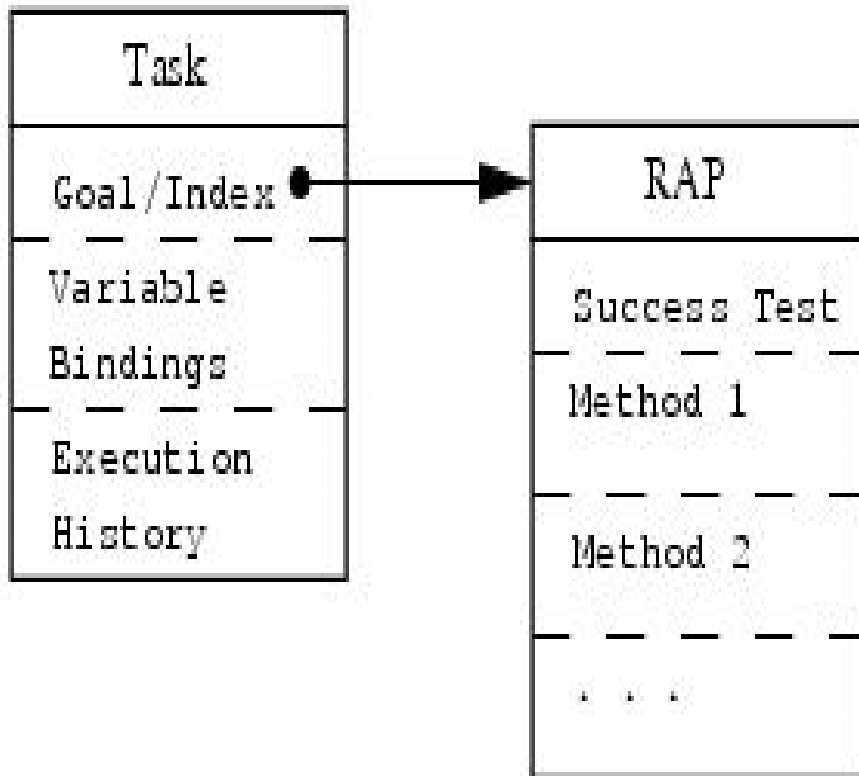
# The RAPs architecture



# Example of RAP

```
(DEFINE-RAP
  (INDEX (load-into-truck ?object))
  (SUCCESS (location ?object in-truck))
  (METHOD
    (CONTEXT (or (not (size-of ?object ?size))
                 (and (size-of ?object ?size)
                      (<= ?size arm-capacity))))
    (TASK-NET
      (t1 (pickup ?object)
           ((holding arm ?object) for t2))
      (t2 (putdown ?object in-truck))))
  (METHOD
    (CONTEXT (and (size-of ?object ?size)
                  (> ?size arm-capacity)))
    (TASK-NET
      (t1 (pickup lifting-aid)
           ((holding arm lifting-aid) for t2))
      (t2 (pickup ?object)
           ((holding arm ?object) for t3))
      (t3 (putdown ?object in-truck))))))
```

# Reactive Action Packages



- RAPs are basic blocks
- Representation of task in all situations

When a task is active, from its corresponding RAP in the library, the appropriate methods for that situation are executed.

Success test is not defined in methods, but it is defined in the description of the task.

# General Structure of a RAP

(DEFINE-RAP *index*

(MONITOR-STATE *query*)

(MONITOR-TIME *reference delay deadline*)

(MONITOR-EVENT *event repetition*)

(REPEAT-WHILE *query*)

(SUCCEED *query*)

(FAIL *query*)

(PRECONDITIONS *query*)

(CONSTRAINTS *query*)

(DURATION *estimated execution time*)

(TIMEOUT *delay result*)

(FUTILITY-THRESHOLD *count*)

(RESOURCES *internal resource definitions*)

(METHOD *plan for carrying out RAP behavior*)

(METHOD *plan for carrying out RAP behavior*)

...)

# Execution of RAPs

- Make a sketchy plan
- Choose a task
- Assess current situation
- Choose method
- Execute

**Plan: set of partially ordered tasks**

**Task: Complex actions**

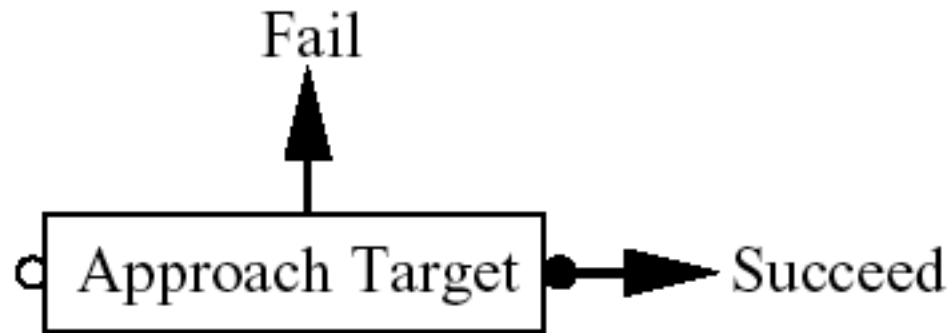
**Success criterion**

**Method: Set of actions**

↓  
No guarantee of success

The tasks has success criterion embedded, so they should check if the task is succeeded or not. Task executes as long as their time window is valid.

# A Symbolic Discrete Task



```
(task-net
  (t1 (approach-target ?target)))
```

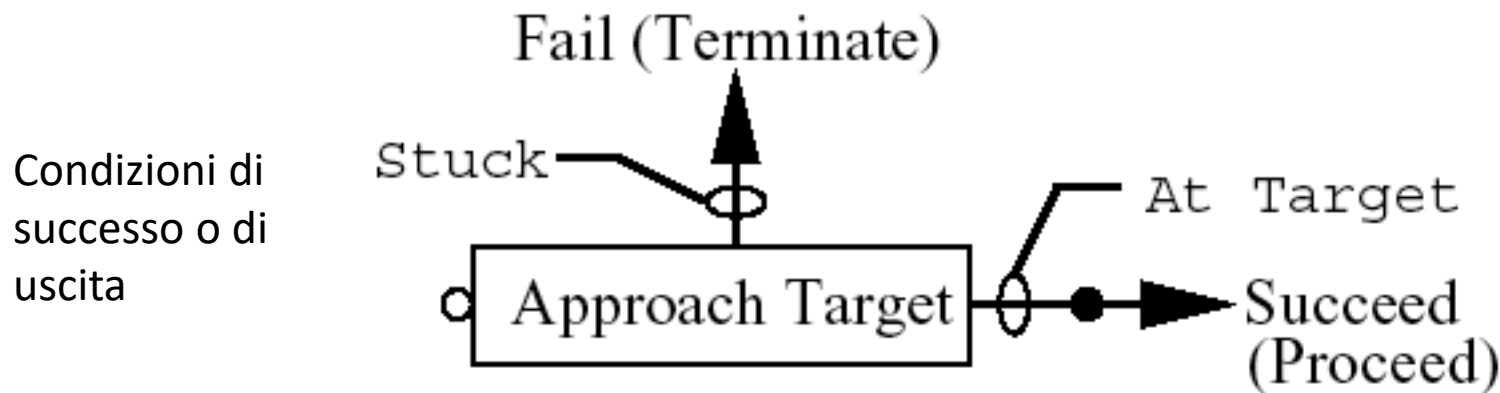
The subtask will either succeed, in which case the interpreter should continue processing the task network after the task or it will fail, in which case the method should fail and all of its subtasks should be terminated ...

It assumes that a subtask actually has a well-defined finish and will know when it has succeeded or failed.

# Waiting for a signal to proceed

However, we have sets of processes, hence detecting goal completion depends on the appropriate interpretation of signals from those processes. RAP methods must explicitly define which signals mean a subtask has succeeded and which signals mean it has failed.

Methods must have a way to let time pass and synchronize further task expansion with process progress.



```
(task-net
  (t1 (approach-target ?target)
    (wait-for (at-target) :proceed)
    (wait-for (stuck) :terminate)))
```



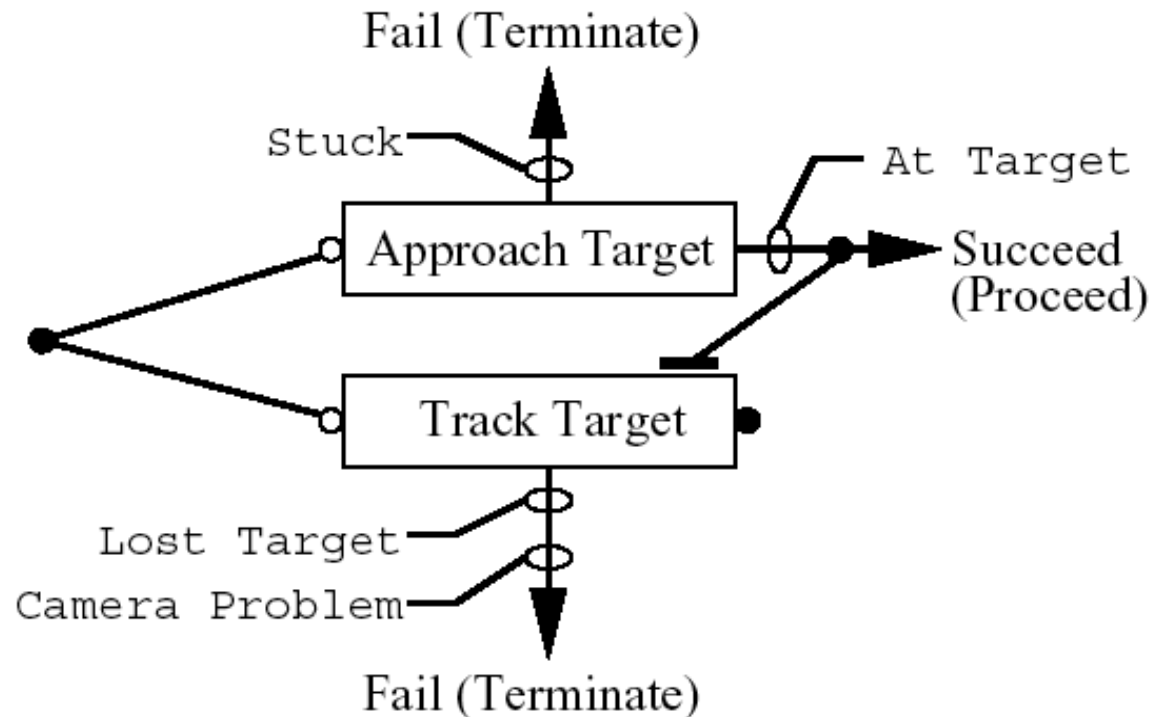
# Concurrent tasks

Both **approach-target** and **track-target** must be active to get the robot to visually servo to a selected fixed object.

The task net does not order these tasks and the interpreter follows two threads of execution.

The *until-end* annotation tells the interpreter that subtask t2 should be terminated whenever subtask t1 completes.

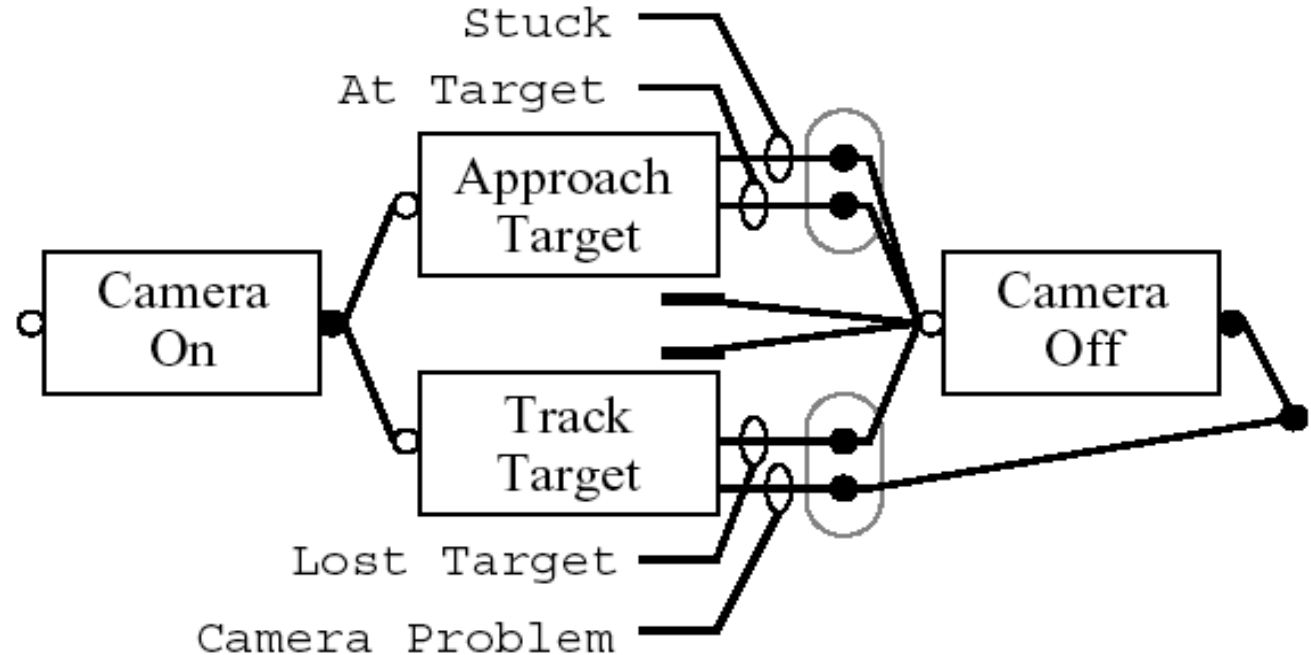
The only way this method can complete successfully is to receive the *at-target* signal from process **approach-target**



```
(task-net
  (t1 (approach-target ?target)
    (wait-for (at-target) :proceed)
    (wait-for (stuck) :terminate))
  (t2 (track-target ?target)
    (wait-for (lost-target) :terminate)
    (wait-for (camera-problem) :terminate)
    (until-end t1)))
```

# More Complex Task Networks

Here, there is no longer any notion of failure, failure stops the process.



```
(task-net
  (t0 (camera-on) (wait-for :success t1) (for t2))
  (t1 (approach-target ?target)
    (wait-for (at-target) t3)
    (wait-for (stuck) t3)
    (until-start t3))
  (t2 (track-target ?target)
    (wait-for (lost-target) t3)
    (wait-for (camera-problem) :terminate)
    (until-start t3))
  (t3 (camera-off)))
```

# Execution support language (ESL)

- ESL (Execution Support Language) is a language for encoding execution knowledge in embedded autonomous agents
- Similar to RAP, but more utilitarian (flexible, and easy-to-use tool)
- Notion of *cognizant failure*:
  - systems should be designed to detect failures when they occur so that the system can respond appropriately (behavior should not continue to bang its head against the wall)
  - contingency-handling constructs of ESL are a means of signaling that a failure has occurred, and a means of specifying a recovery procedure for a particular type of failure
- Task Net as in RAP
- A logical database is provided to assert and retract facts

ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents. Gat,  
In: Proceedings of the IEEE Aerospace Conference, 1997

# Task Control Architecture (TCA)

- Vertical task decomposition: several task-specific modules communicate through a central control module
- Deliberation: top-down task-subtask, resolve constraints
- Central control routes messages
  - Inform, query, command, monitor

# TDL (Simmons 1998)

- TDL (Task Definition Language):
  - Extension of C++ that simplifies the development of robot control programs by including explicit syntactic support for task-level control capabilities
  - TDL directly supports task decomposition, synchronization of subtasks, execution monitoring, and exception handling
  - Compiler transforms TDL code into efficient, platform-independent C++ code that invokes a Task Control Management (TCM) library to manage task-control aspects of the robot system
  - TDL and TCM extend the TCA control constructs to include additional synchronization capabilities

# TDL (Simmons 1998)

- Task Trees:
  - Task trees are the basic representation underlying TDL and TCM
  - Each task tree node has an action associated with it (parameterized piece of code); actions can perform computations, dynamically add child nodes to the task tree, perform some physical action in the world
  - Task trees are generated dynamically, the actions associated with nodes can use current perception data to make decisions about what nodes to add to the tree and how to parameterize their actions
  - Task nodes are goals or commands (plus monitors and exceptions)
    - After a monitor is enabled, events can activate it, which cause a separate invocation of its action; a monitor's action can issue a trigger event
    - Exception handlers are associated with a given node in the task tree and a specific reason (some user-defined string).
  - Task are disabled, enabled, active or completed

# TDL (Simmons 1998)

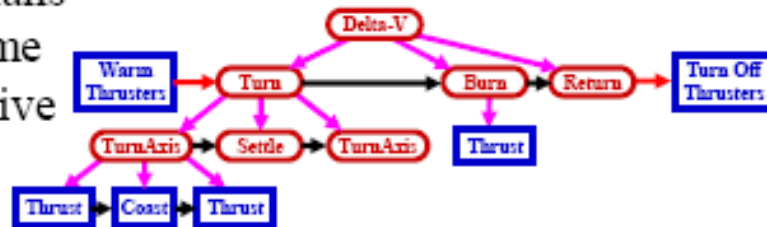
- TDL Maintains and Coordinates *Task Trees*

- *Execution trace* of hierarchical plans

- Created dynamically at run time
    - Can be conditional and recursive

- Temporal constraints (partially) order task execution

- Planning and sensing treated as schedulable activities; Concurrent planning, sensing, and execution



- Task Trees Augmented with *Reactive* Elements

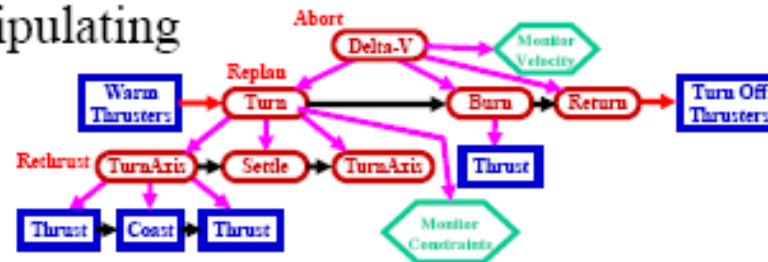
- Task-specific execution monitors

- Context-dependent, hierarchical exception handlers

- Replan by Analyzing and Manipulating Task Trees

- Terminate subtrees

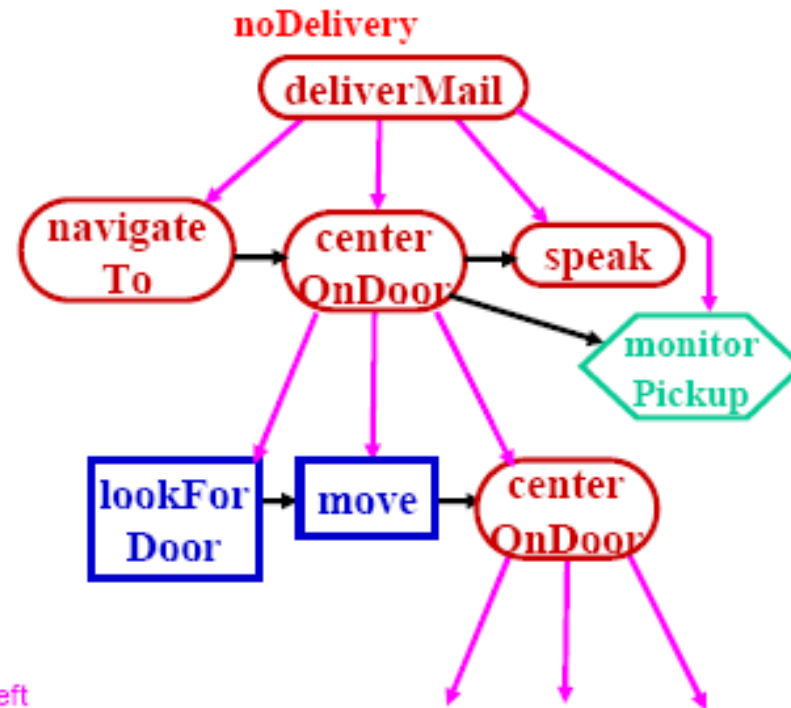
- Add new nodes and/or temporal constraints



# TDL

```
Goal deliverMail (int room)
  Exception Handler noDelivery
{
  double x, y;
  getRoomCoordinates(room, &x, &y);
  spawn navigateTo(x, y);
  spawn centerOnDoor(x, y)
    with sequential execution previous,
    terminate in 30.0;
  spawn speak("Xavier here with your mail")
    with sequential execution centerOnDoor,
    terminate at monitorPickup completed;
  spawn monitorPickup()
    with sequential execution centerOnDoor;
}
```

```
Goal centerOnDoor(double x, double y)
{
  int whichSide;
  spawn lookForDoor(&whichSide) with wait;
  if (whichSide != 0) {
    if (whichSide < 0) spawn move(-10); // move left
    else spawn move(10); // move right
    spawn centerOnDoor(x, y)
      with disable execution until
      previous execution completed;
  }
}
```

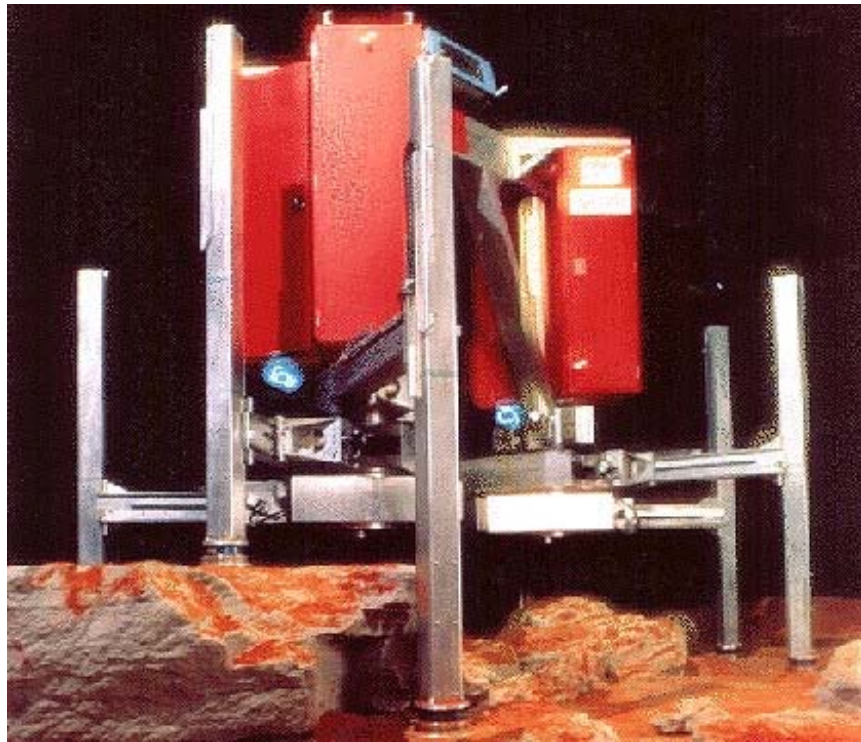


- “spawn” to add a child node to the task tree:
  - non-blocking, the child subtask may not be handled by the time control returns to the parent task. Spawned tasks can be synchronized using the “with” clause
- “wait” makes “spawn” blocking:
  - control is not returned until the spawned task, and all of its descendants, have been handled



# Ambler Walking Robot

Ambler was a walking robot that enabled energy-efficient overlapping gaits. Developed as a testbed for research in walking robots operating in rugged terrain.



David Wettergreen, H. Thomas and Chuck Thorpe. Conference Paper, Proceedings of the 1990 IEEE International Conference on Systems Engineering, pp. 198 - 203, 1990

# Ambler Modules

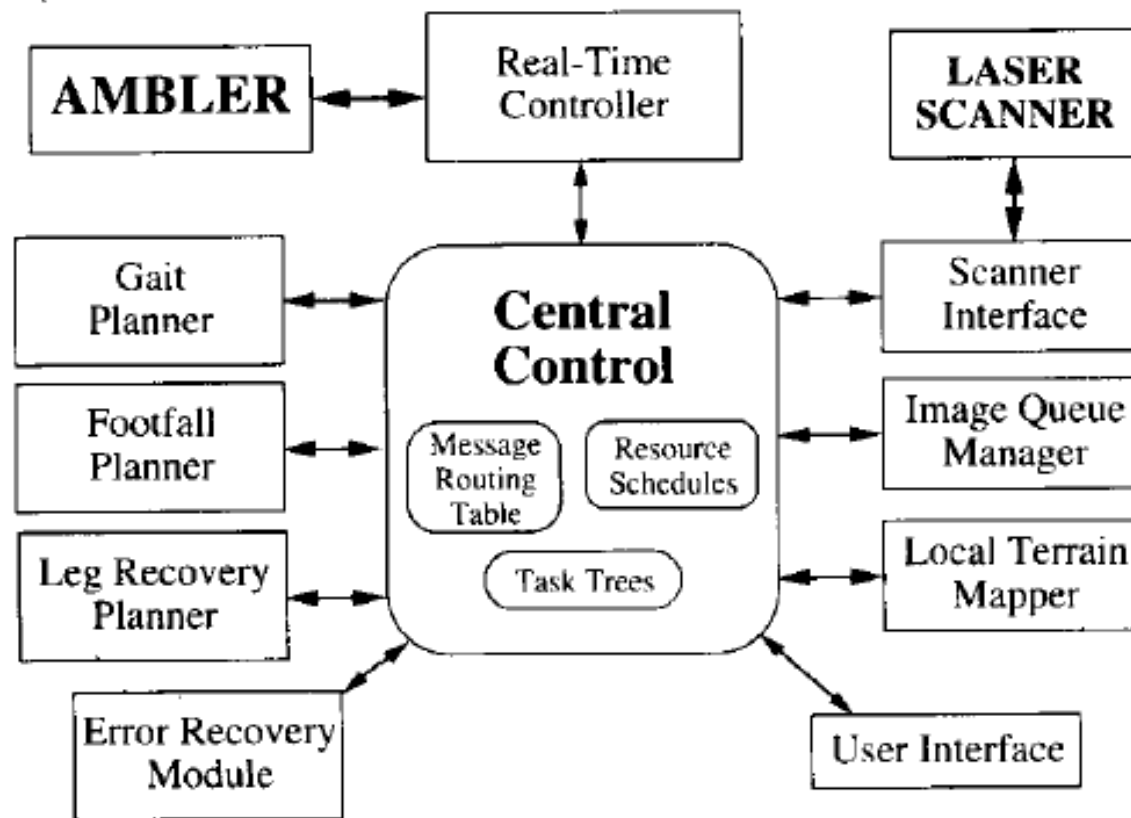


Fig. 3. Modules for the Ambler walking system.

# Ambler Task Tree

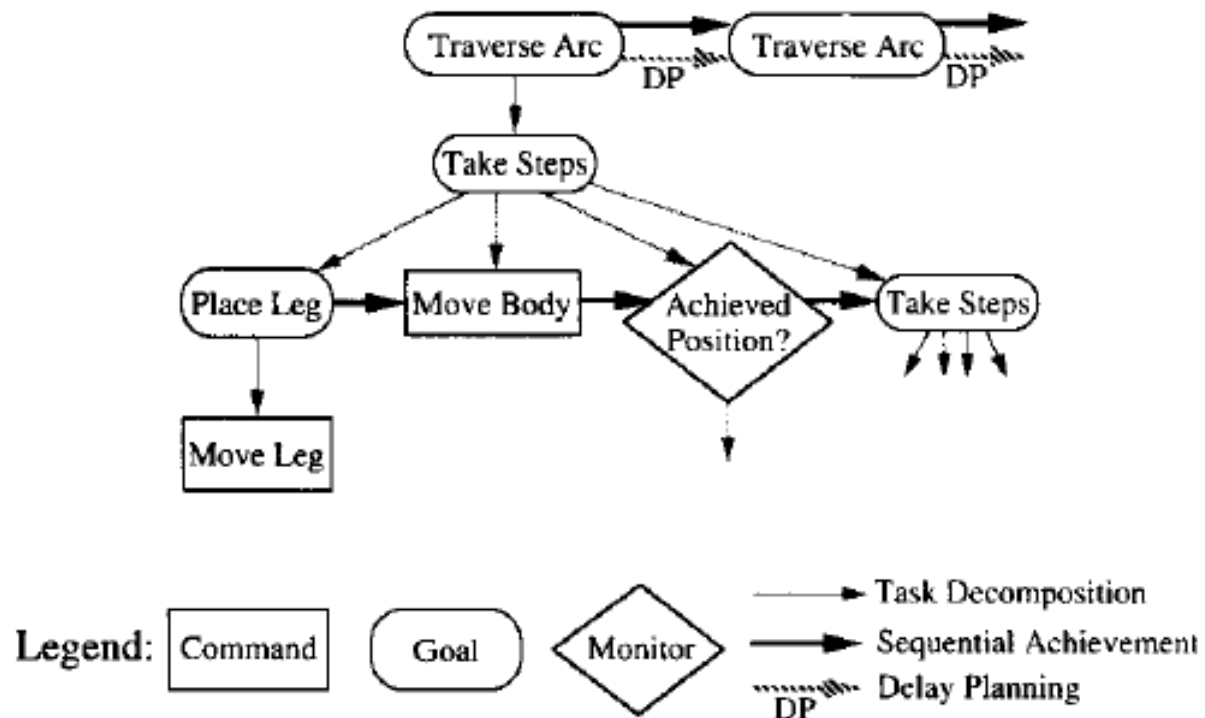


Fig. 4. Task tree for autonomous walking.

# TCA: Monitoring

- Central control traverses tree and handles messages:
  - asks gait planner to traverse arc,
  - gait planner asks terrain mapper for elevation map in order to take steps
  - Gait planner asks leg recovery planner to place leg, move leg, move body,
  - Gait planner activates monitor whether achieved position

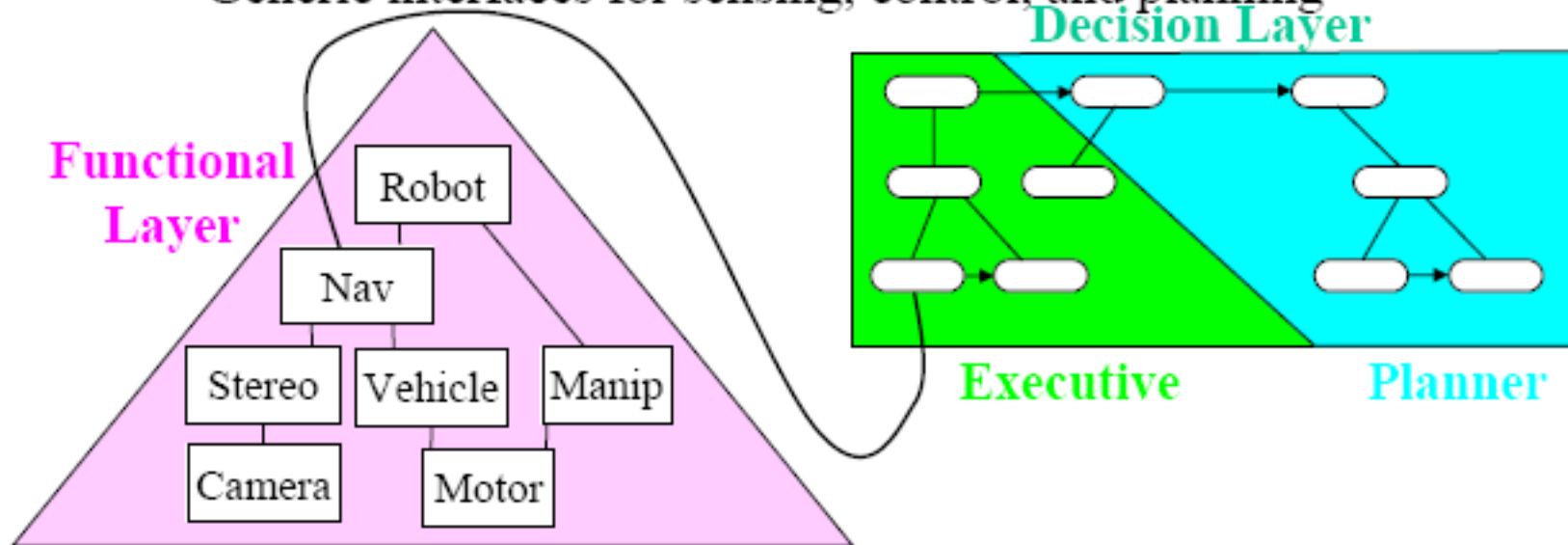
# TCA: Control

- Ordering and temporal constraints
- Delay planning constraint: goal cannot be issued until previous task achieved
  - Can do place leg planning while monitoring achieve position
- Exception handling: error recovery modules examine and modify task trees
  - Eg: if position not achieved, add take steps subtask



# Clarity

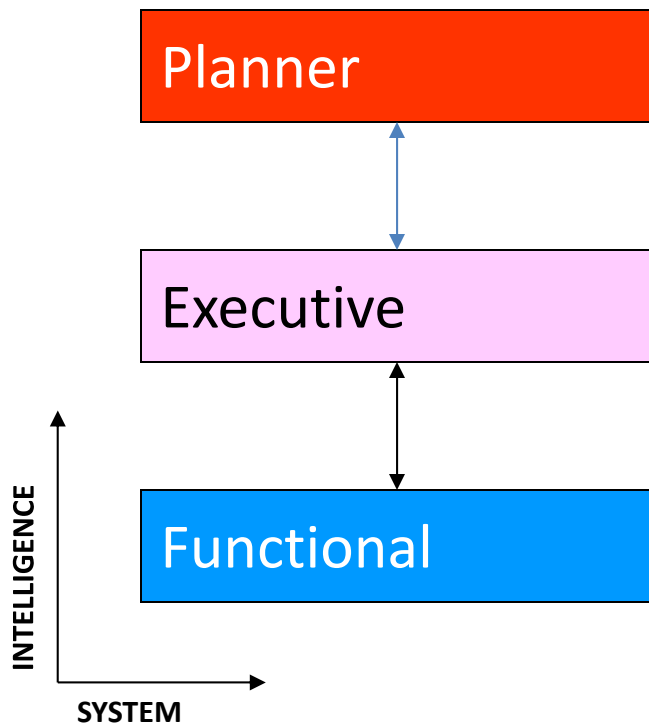
- Two-Tiered Architecture
  - Functional layer: Object oriented, reusable
  - Decision layer: Tightly integrates planner (ASPEN) and executive (TDL)
- Developed at NASA for Next-Generation Mars Rovers
  - Generic interfaces for sensing, control, and planning



# A New View of Architecture Hierarchy

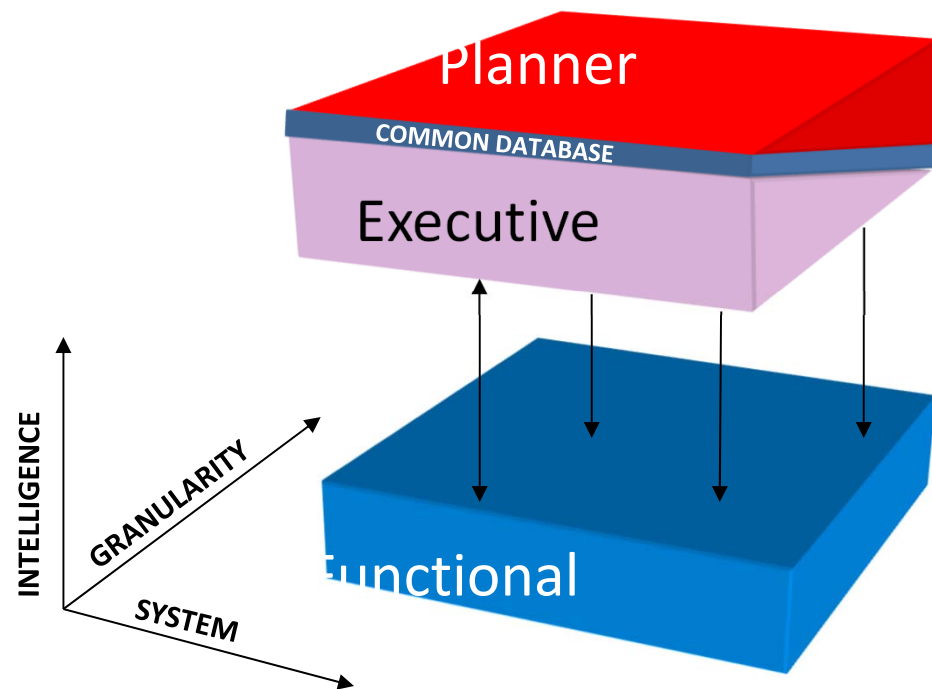
## Typical 3 Level Architecture

- *Functional Level is flat – typically a thin layer over the hardware*
- *Planner has no access to Functional Layer.*
- *Abstraction and granularity is mixed with intelligence.*



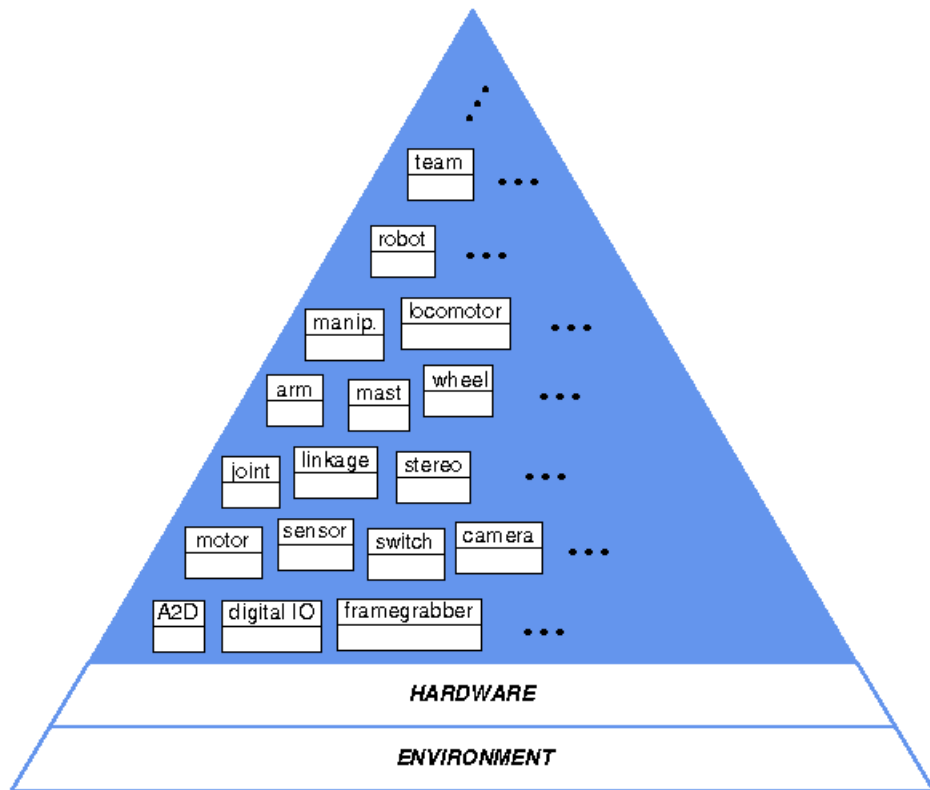
## Proposed 2 Layer Architecture

- *Functional Layer contains object-oriented abstraction of hardware at all levels of system granularity.*
- *Planner and Exec are similar, dominating at different levels of granularity, sharing a common database.*
- *Planner does not have direct access to the Functional Layer for execution, but executive may be minimized.*





# The Functional Layer



- **Object-oriented Hierarchy:** captures granularity and abstraction of the system.
- **Resident State:** the state of the system components is contained in the appropriate objects and obtained from it by query. This includes state variables values, object state machine status, resource usage, health monitoring, etc.
- **Local Planners:** part of the appropriate object. For instance, trajectory planners for arm motion or driving.
- **Resource Usage Predictors:** part of the appropriate objects. The prediction can have specified levels of fidelity. For high levels, access of subordinate objects may be needed.
- **Encoded Functionality:** objects contain basic functionality for themselves, accessible from within the Functional Layer, as well as the Decision Layer.
- **Simulation:** system simulation can be obtained at various levels of fidelity by disconnecting subordinate objects from superior ones.
- **Test and Debug:** objects contain test and debug interfaces and have external exercisers.

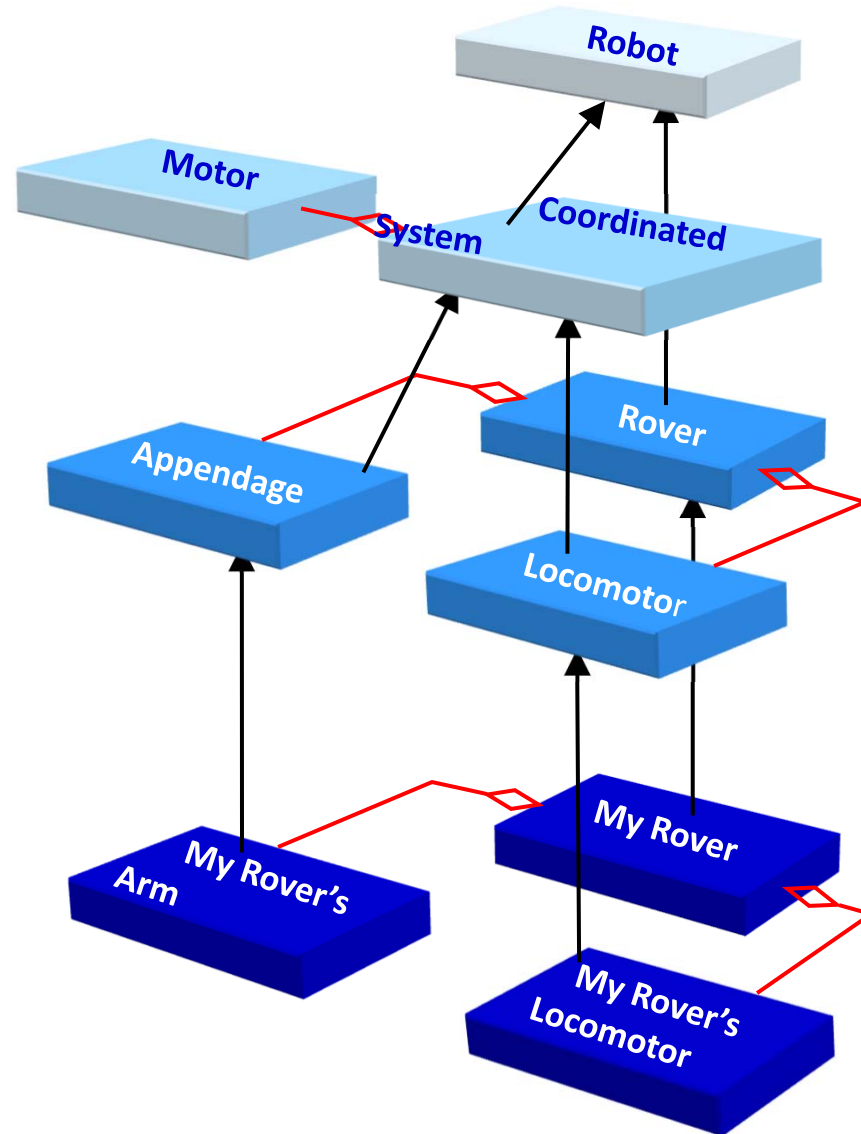
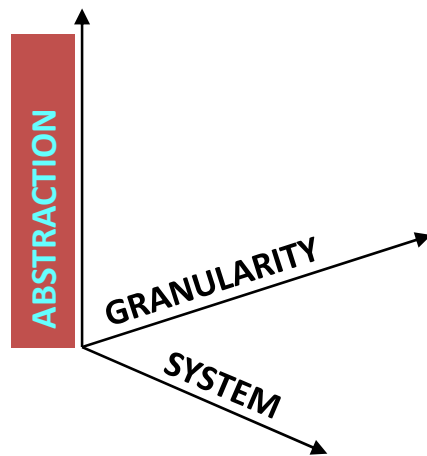
# Functional Layer Object-Oriented Hierarchy

**Implementation Instance:** *Functional Layer* software controlling the system is an instance of software derived from more abstract classes.

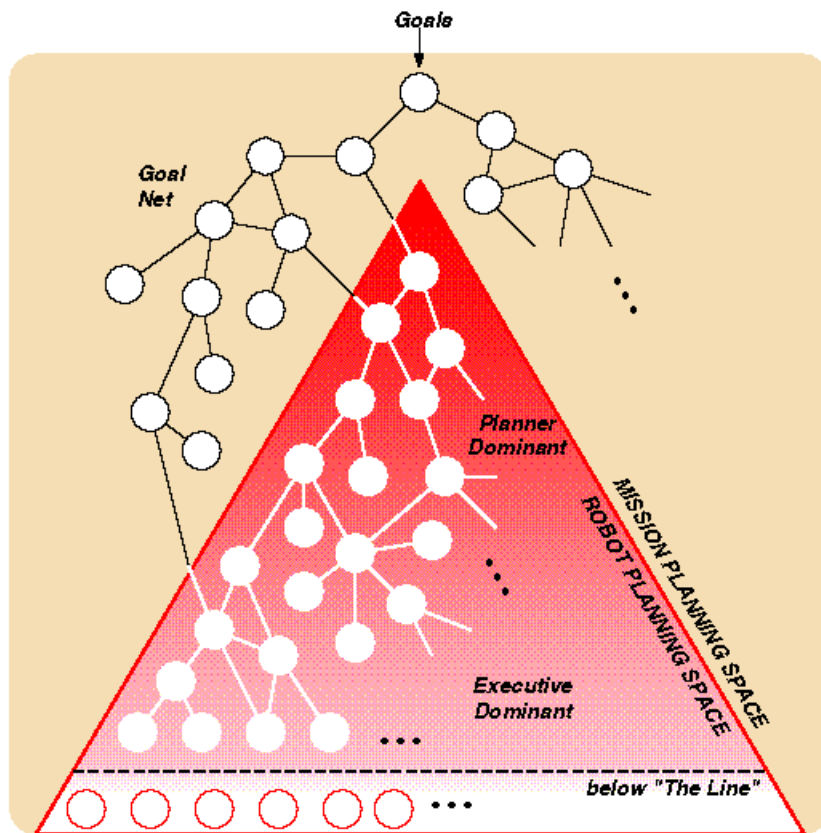
**Aggregate Functionality:** Centralizes and enhances capabilities.

**Abstraction dimension:** In addition to system, granularity, and intelligence.

**Inheritance:** Abstraction dimension is best described by the inheritance properties of an object oriented description



# The Decision Layer



- **Goal Net:** temporal constraint network at planning level, task tree at executive level.
- **Goals:** specified as constraints on state over time. Consistent with MDS, and ASPEN/CASPER. Runtime loadable. ("What not to do.")
- **Tasks:** specified as explicitly parallel or sequential activities that are tightly linked. Consistent with TDL. Compiled in. ("What to do.")
- **Commands:** termination fringes of goal net where the Functional Layer is accessed.
- **The Line:** the lower border of the elaborated goal net. It is floating, according to the current elaboration. When projected on the Functional Layer, it denotes the border below which the system is a "black box".
- **State:** the state of the Functional Layer is obtained by query. The state of the Decision Layer, which is essentially its plan, the active elaboration, and history of execution, is maintained by this layer. It may be save, or reloaded, in whole or part.

# Decision/Functional Layer Connectivity

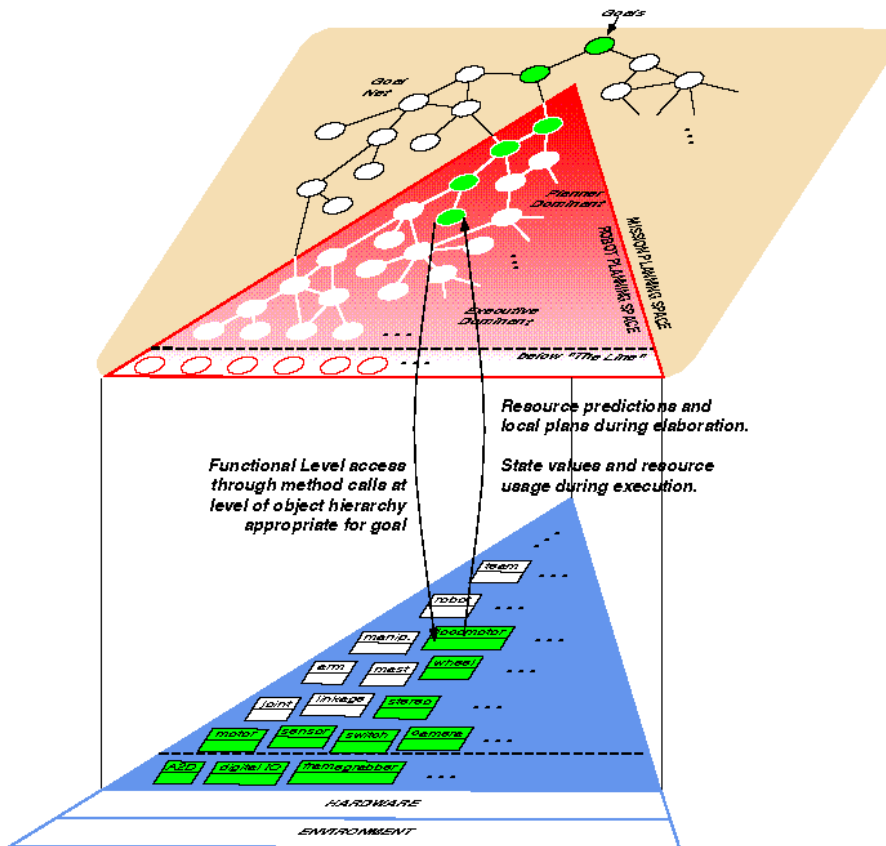
## Lower Granularity

- Enable global decisions that are lacking in Functional Layer.
- Utilize better decision making capability for smaller scale issues.
- Bypass built-in default capabilities of Functional Layer.
- Take advantage of faster planner or exec.

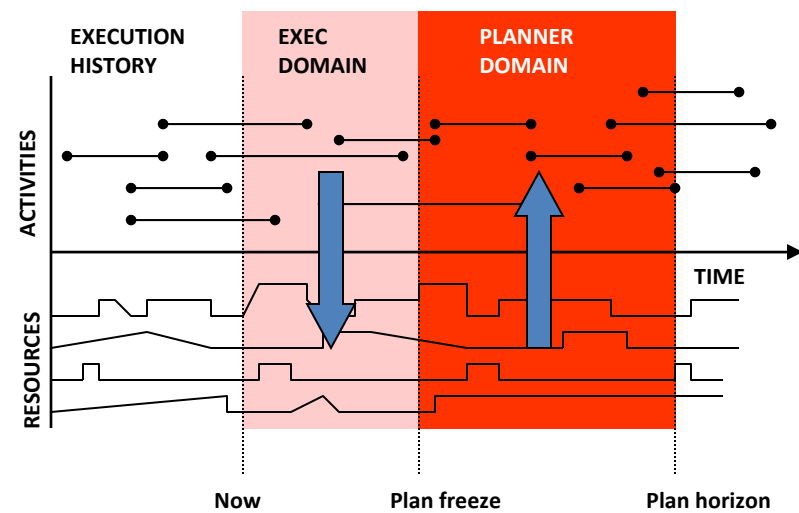
## Higher Granularity

- Take advantage of built in capabilities of Functional Layer
- Avoid second guessing Functional Layer algorithms and error handling.
- Approach planning and commanding at appropriate level of granularity for the problem – don't micro-manage the system.
- Allow the use of slower Decision Layer tools.

# Timeline Interaction



- **Elaboration/Planning:** expansion of goals/activities, along with resource prediction requests from Functional Layer
- **Scheduling:** rearrangement of activities based on resource constraints.
- **Execution:** Executive expands activities into task trees or directly into commands. Conditional activities plus state feedback cause changes in resource usage values.
- **Plan Repair:** Planner iteratively repairs the plan based on the new projections of resources.



# PLEXIL

PLEXIL (Plan Execution Interchange Language) development by NASA:

- Language for representing plans
- Technology for executing these plans on real or simulated systems

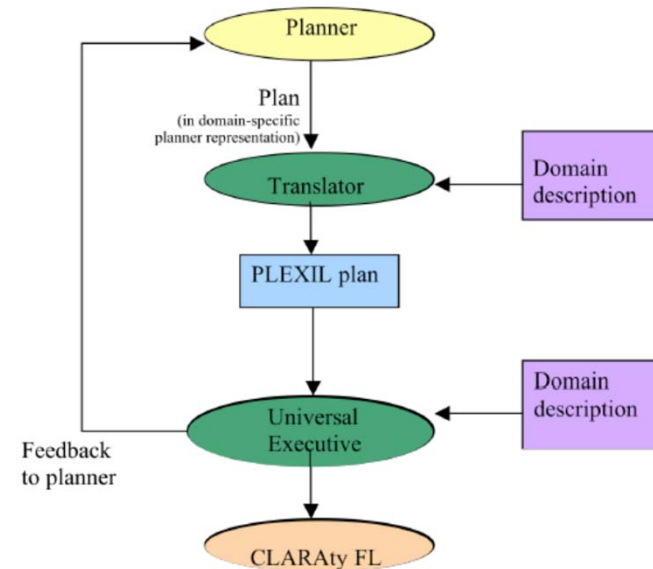
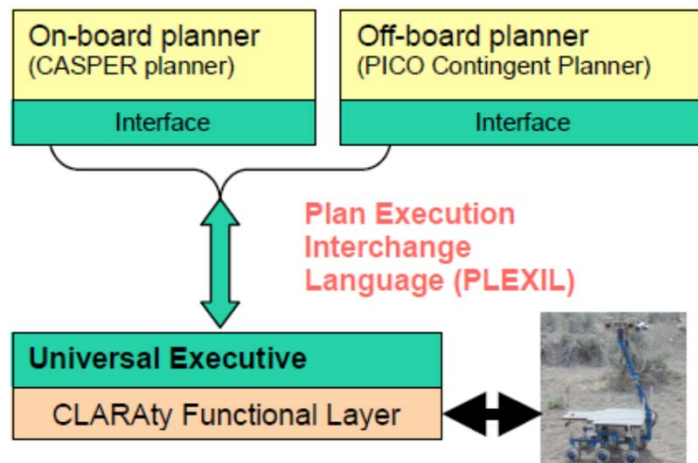


Figure 2: PLEXIL - Executive Interface

V. Verma, T. Estlin, A. Jónsson, C. Pasareanu, R. Simmons, K. Tso: Plan Execution Interchange Language (PLEXIL) for Executable Plans and Command Sequences, Proc. 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space

# PLEXIL

PLEXIL (Plan Execution Interchange Language) development by NASA:

- The PLEXIL plan execution language is based on a hierarchical representation of execution nodes.
- Execution nodes describe both initiation of real-world actions, and the control of their execution.
- The nodes are arranged into hierarchical trees where leaf nodes are action nodes and internal nodes are control nodes
- The execution of each node is governed by a set of conditions, such as when the node gets activated and when it is done

```
Node:{  
  NodeId: DriveUntilStuck;  
  RepeatUntilCondition: LookupOnChange{"Rover:wheelStuck"}==false;  
  NodeList:{  
    Node:{ DriveOneMeter  
      Command: Rover:Drive(1);  
    }  
  }  
}
```

V. Verma, T. Estlin, A. Jónsson, C. Pasareanu, R. Simmons, K. Tso: Plan Execution Interchange Language (PLEXIL) for Executable Plans and Command Sequences, Proc. 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space, 2005

# PLEXIL

PLEXIL (Plan Execution Interchange Language) development by NASA:

## **NodeAttributes:**

**Identifier (NodeID):** A unique symbolic name

**StartCondition:** Boolean expression

**EndCondition:** Boolean expression

**PreCondition:** Boolean expression

**PostCondition:** Boolean expression

**InvariantCondition:** Boolean expression

**RepeatUntilCondition:** Boolean expression

**Priority:** Integer

**Variables:** List of local variable declarations

**Interface:** List of variables “passed” to node

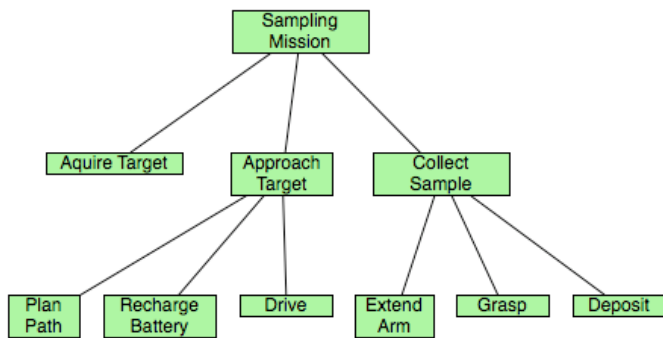
The execution of a node driven by node conditions (boolean expressions). Conditions capture internal/external information and temporal relationships

V. Verma, T. Estlin, A. Jónsson, C. Pasareanu, R. Simmons, K. Tso: Plan Execution Interchange Language (PLEXIL) for Executable Plans and Command Sequences, Proc. 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space, 2005

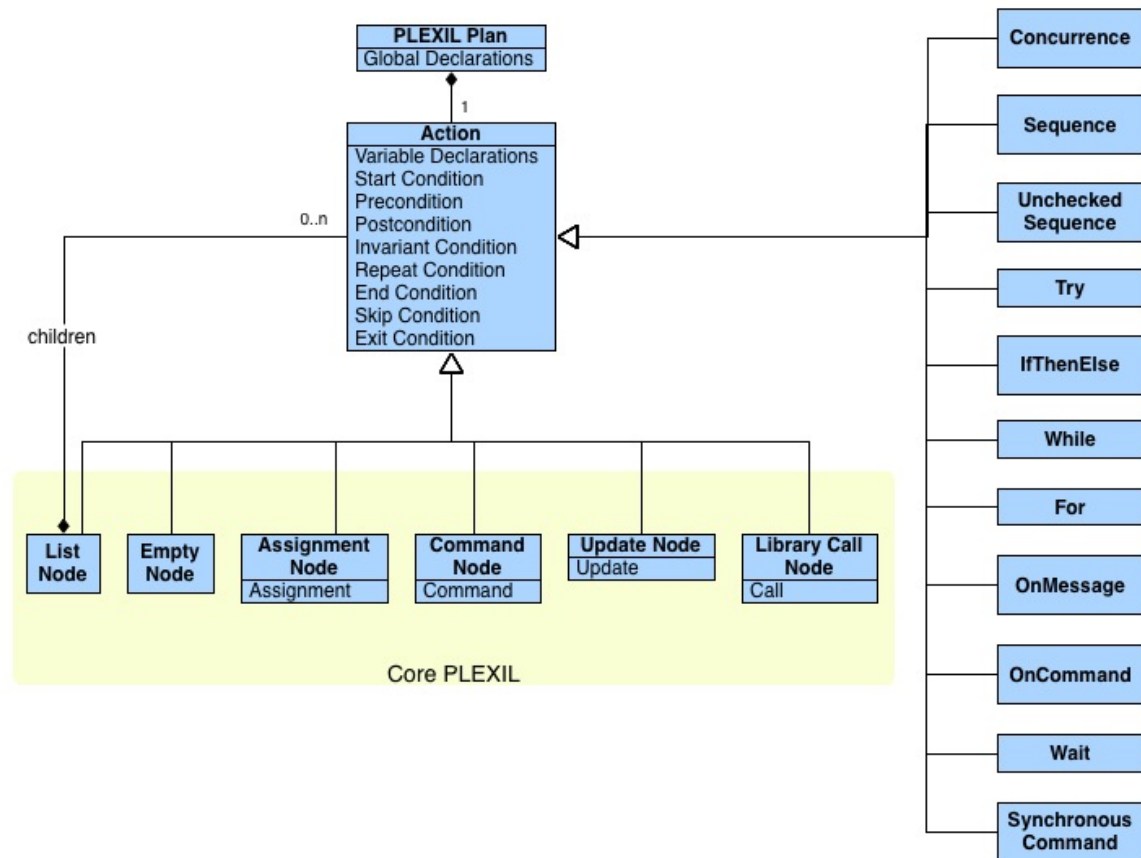


# PLEXIL

PLEXIL (Plan Execution Interchange Language) development by NASA:



A PLEXIL plan is a tree of actions with a single root action. This tree represents a hierarchical decomposition of tasks. High level tasks are closer to the root; leaf actions represent primitive behaviors such as assigning to a variable or sending a command to the external system.



# PLEXIL

PLEXIL (Plan Execution Interchange Language) development by NASA:

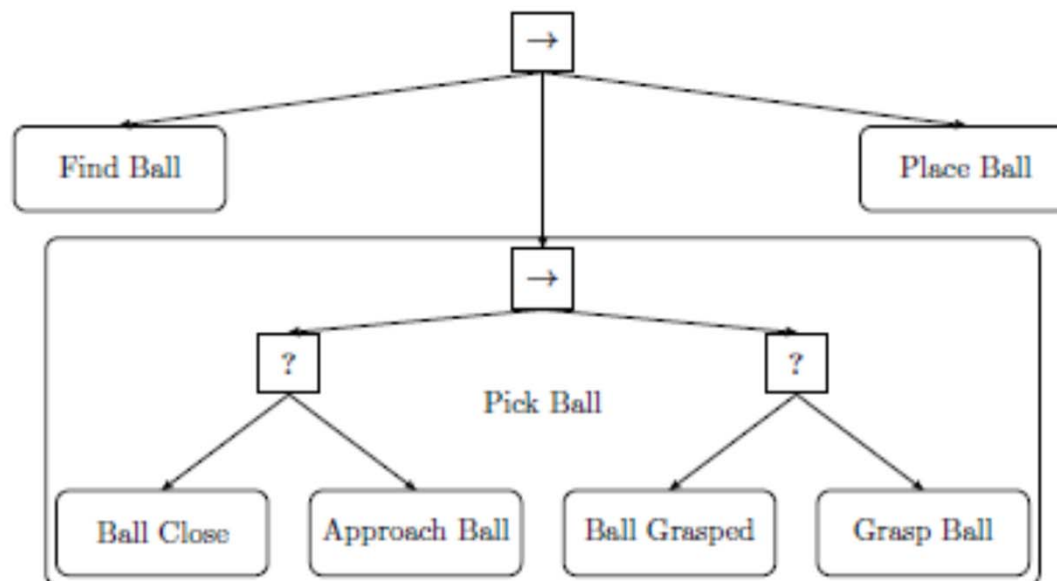
The *PLEXIL Executive* is an implementation of the PLEXIL language.

It is also a runtime environment that provides or facilitates the following capabilities:

- loading and execution of PLEXIL plans and libraries
- commanding and querying state of external systems
- notifications about execution status:
  - to the Plexil Viewer
  - to external systems

# Behavior Trees

A Behavior Tree (BT) is a way to structure the switching between different tasks in an autonomous agent, such as a robot or a virtual entity in a computer game



# Behavior Trees

Developed in the computer game industry to increase modularity in the control structures of Non-Player Characters (NPCs)

At Carnegie Mellon University, BTs have been used extensively to do robotic manipulation

FSMs have long been the standard choice when designing a task switching structure, but they lack of modularity and flexibility

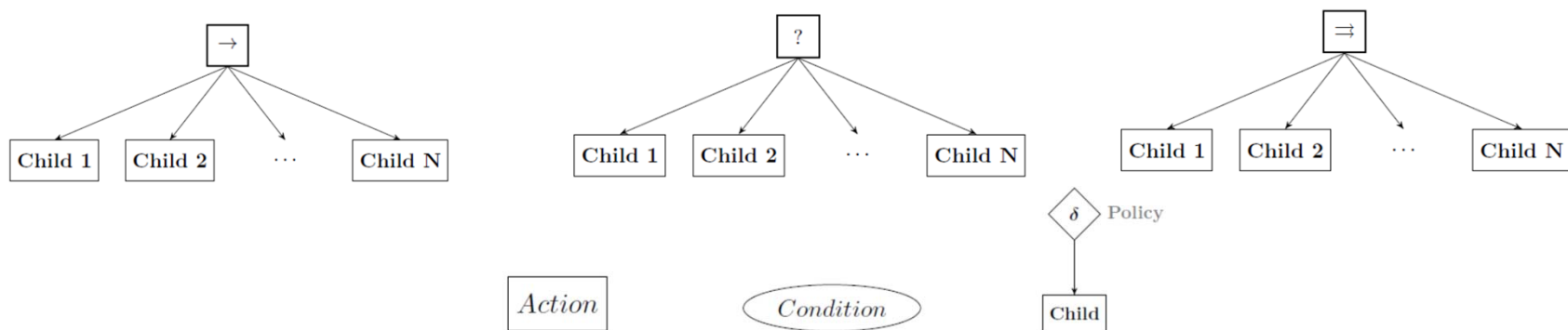
# Behavior Trees

The execution starts from the root which sends enabling signals (ticks) that allows the execution of a child

- A node is executed if and only if it receives ticks.
- The child immediately returns **Running** to the parent, if its execution is under way, **Success** if it has achieved its goal, or **Failure** otherwise.

Nodes classified as **root**, **control flow** nodes, or **execution** nodes:

- Control flow nodes (Sequence, Fallback, Parallel, and Decorator)
- Execution nodes (Action and Condition)



- (a) Action node. The label describes the action performed.
- (b) Condition node. The label describes the condition verified.
- (c) Decorator node. The label describes the user defined policy.

# Behavior Trees

The execution starts from the root which sends enabling signals (ticks) that allows the execution of a child

Nodes classified as **root**, **control flow** nodes, or **execution** nodes:

- Control flow nodes (Sequence, Fallback, Parallel, and Decorator)
- Execution nodes (Action and Condition)

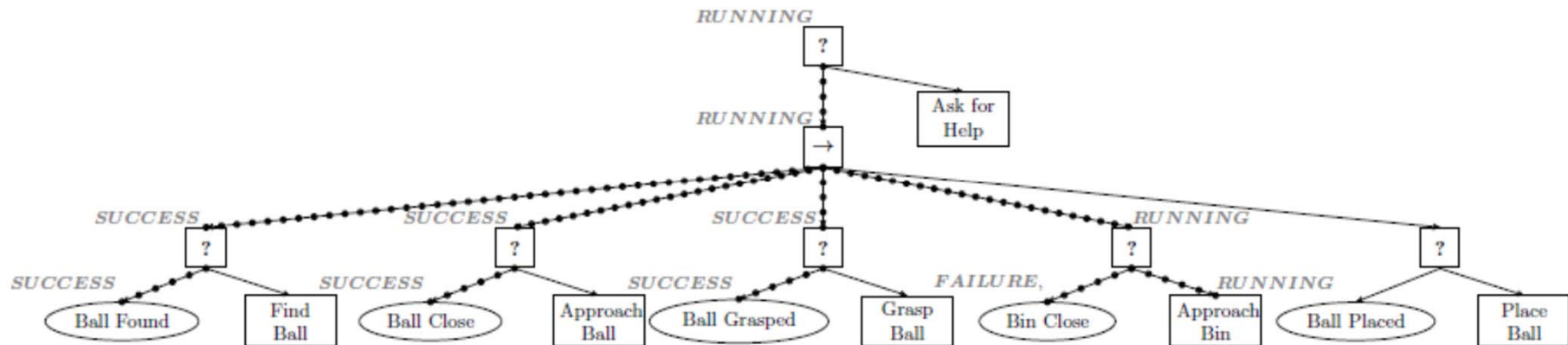
Node type	Symbol	Succeeds	Fails	Running
Fallback	?	If one child succeeds	If all children fail	If one child returns Running
Sequence	→	If all children succeed	If one child fails	If one child returns Running
Parallel	⇒	If $\geq M$ children succeed	If $> N - M$ children fail	else
Action	text	Upon completion	If impossible to complete	During completion
Condition	text	If true	If false	Never
Decorator	◇	Custom	Custom	Custom

# Behavior Trees

The execution starts from the root which sends enabling signals (ticks) that allows the execution of a child

Nodes classified as **root**, **control flow** nodes, or **execution** nodes:

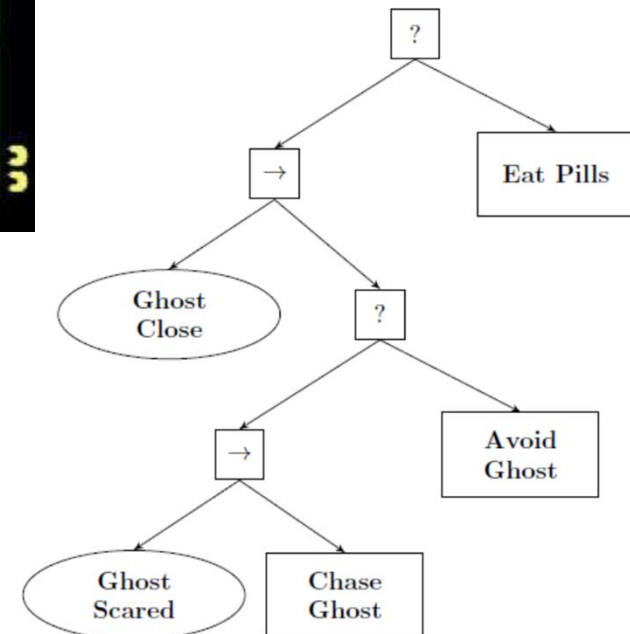
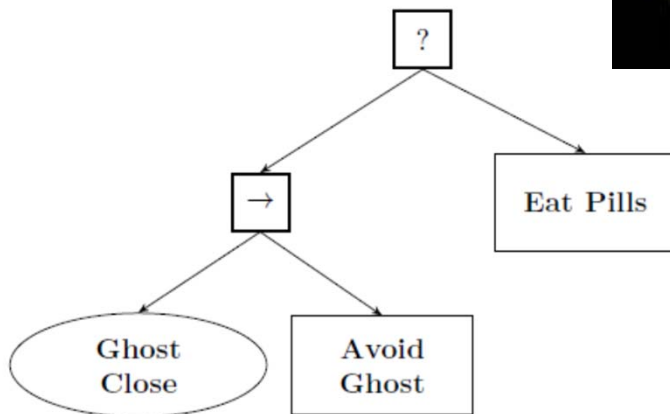
- Control flow nodes (Sequence, Fallback, Parallel, and Decorator)
- Execution nodes (Action and Condition)



# Behavior Trees

The execution starts from the root which sends enabling signals (ticks) that allows the execution of a child

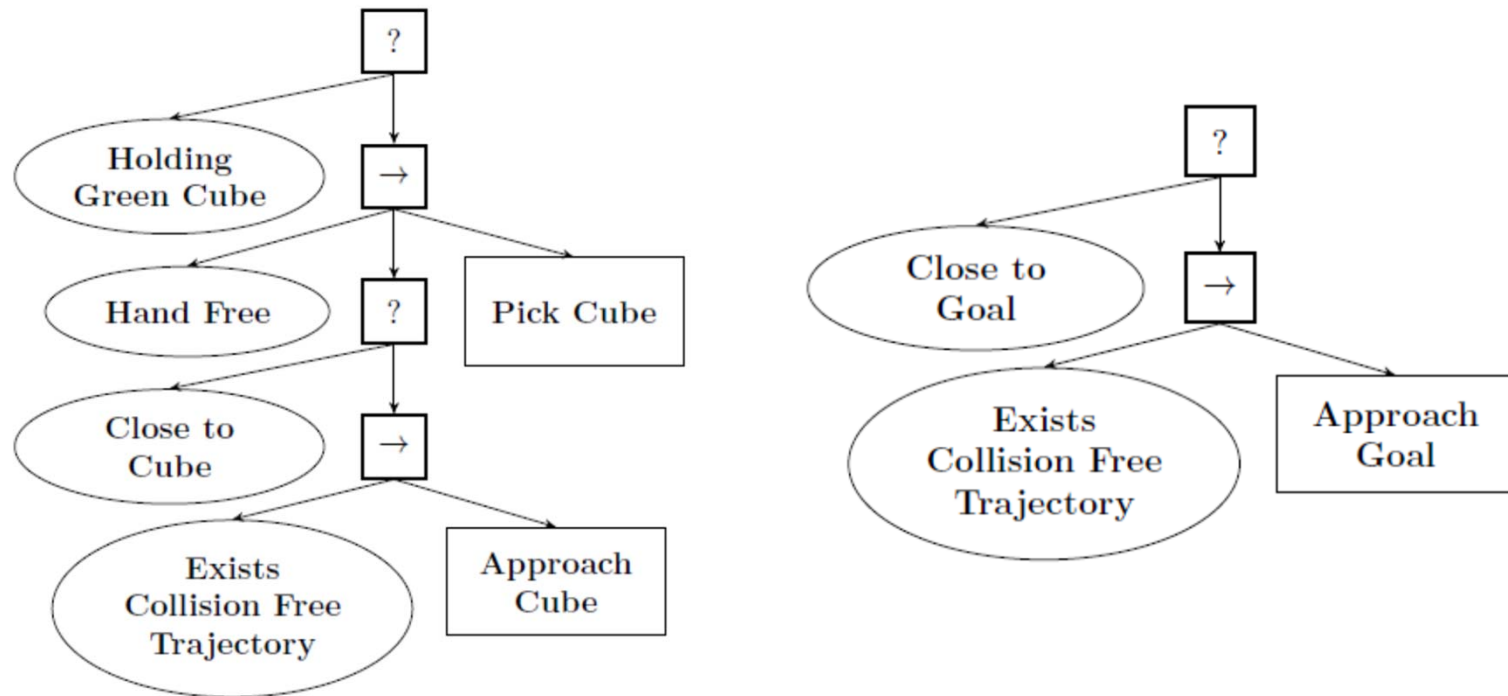
Pack-man example





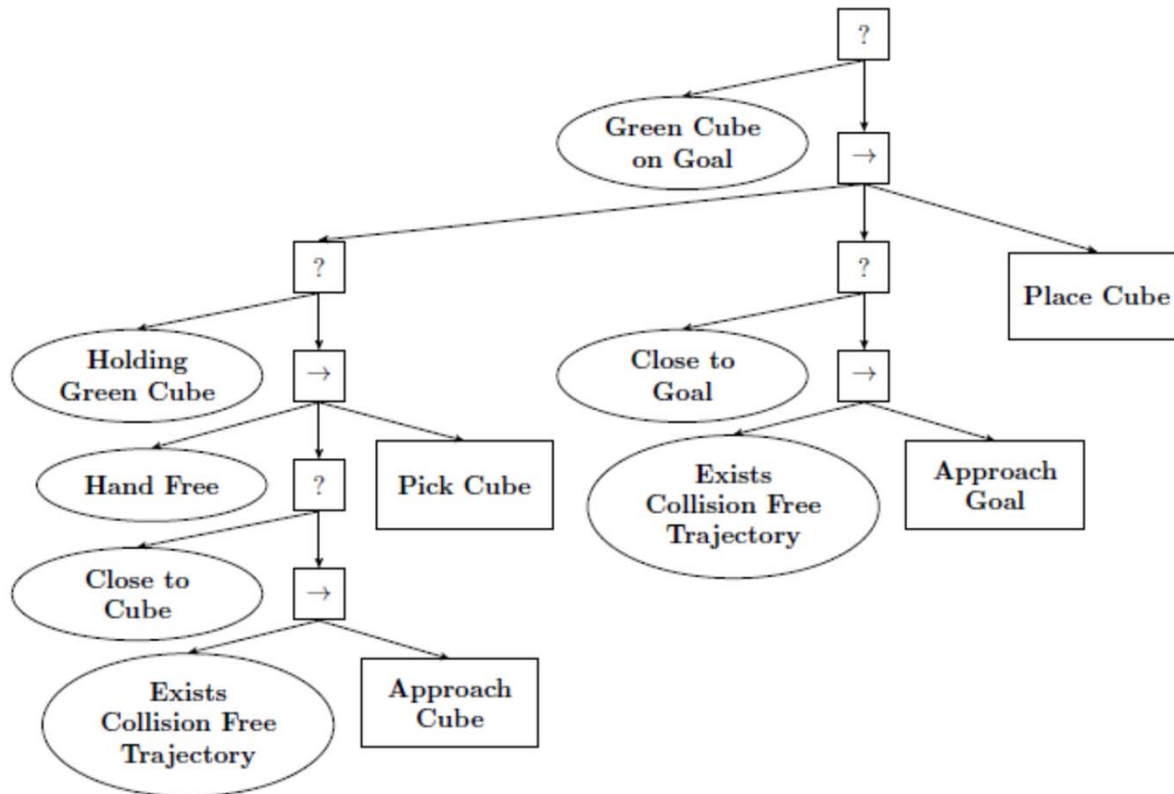
# Behavior Trees

Pick and Place scenario in CoppeliaSim <https://btirai.github.io/>



# Behavior Trees

The execution starts from the root which sends enabling signals (ticks) that allows the execution of a child



# Behavior Trees

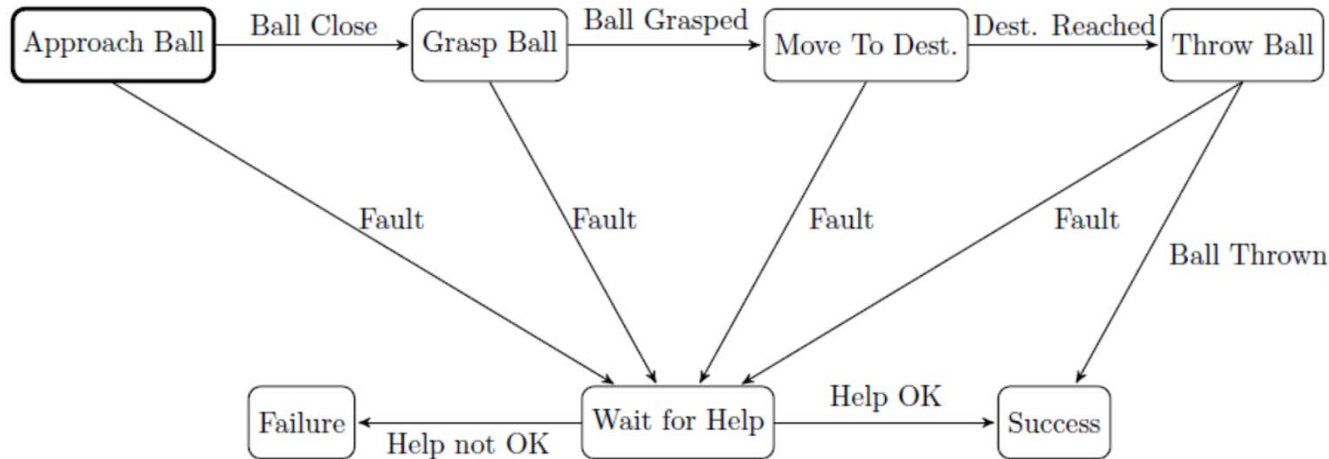
Amazon Picking Competition

The Amazon Robotics/Picking Challenge <http://amazonpickingchallenge.org/>



M. Colledanchise and P. Ogren Behavior Trees in Robotics and AI. An Introduction

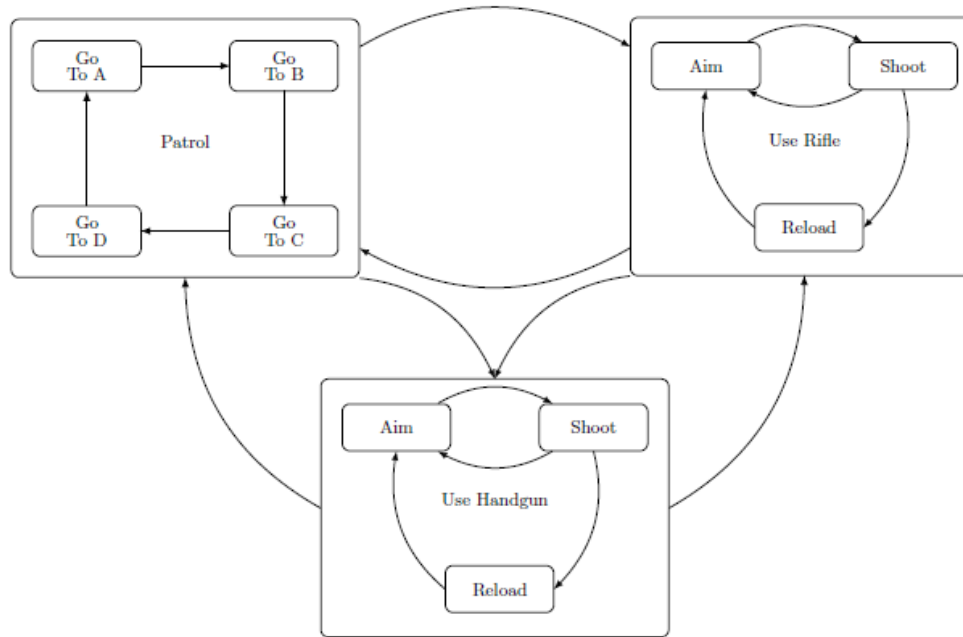
# BTs vs FSMs



Finite State Machines

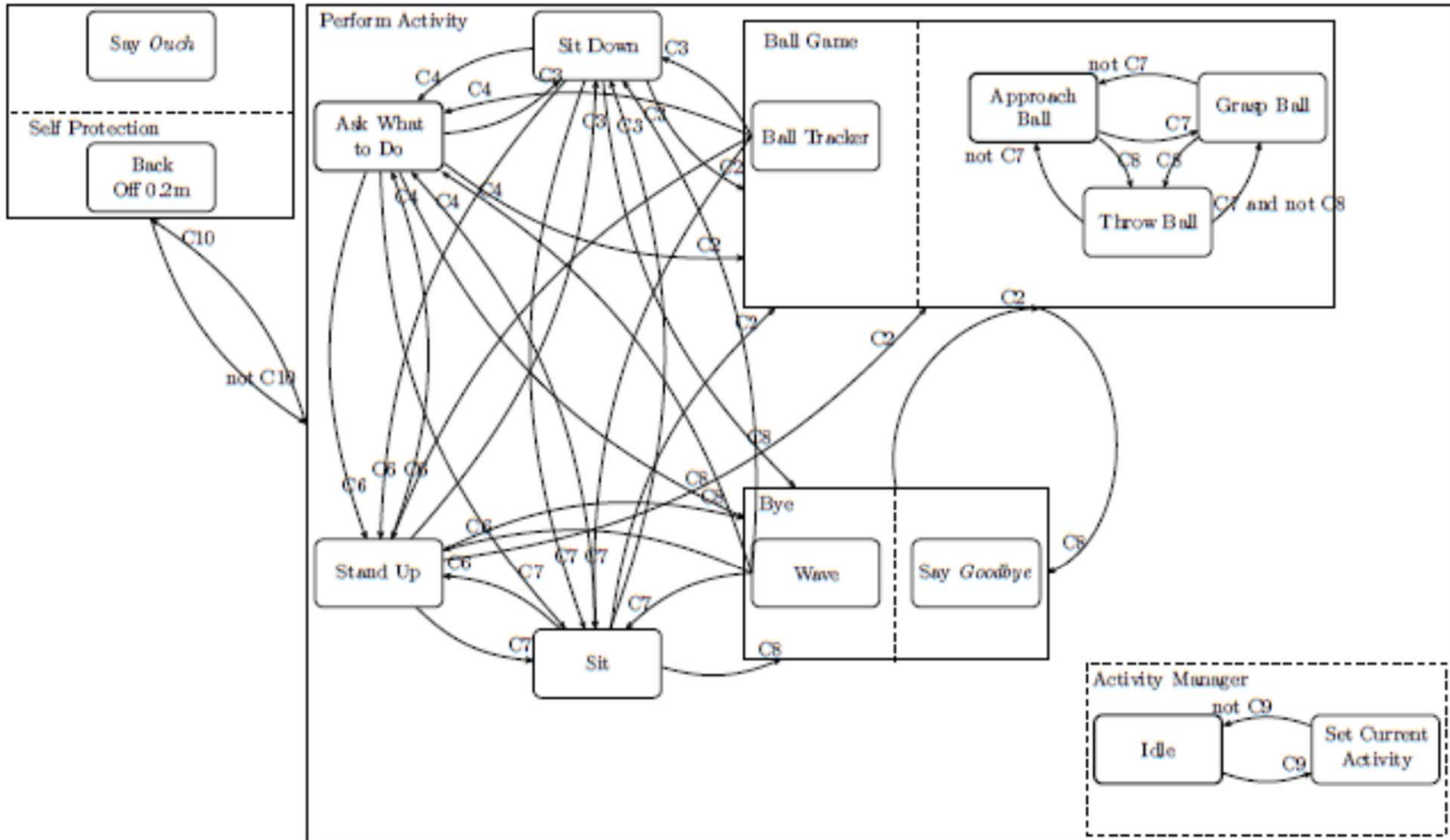
- **Maintainability:** Adding or removing states requires the re-evaluation a potentially large number of transitions and internal states of the FSM
- **Scalability:** FSMs with many states and many transitions between them are hard to modify, for both humans and computers.
- **Reusability:** The transitions between states may depend on internal variables, making it unpractical to reuse the same sub-FSM in multiple projects

# BTs vs HFSMs

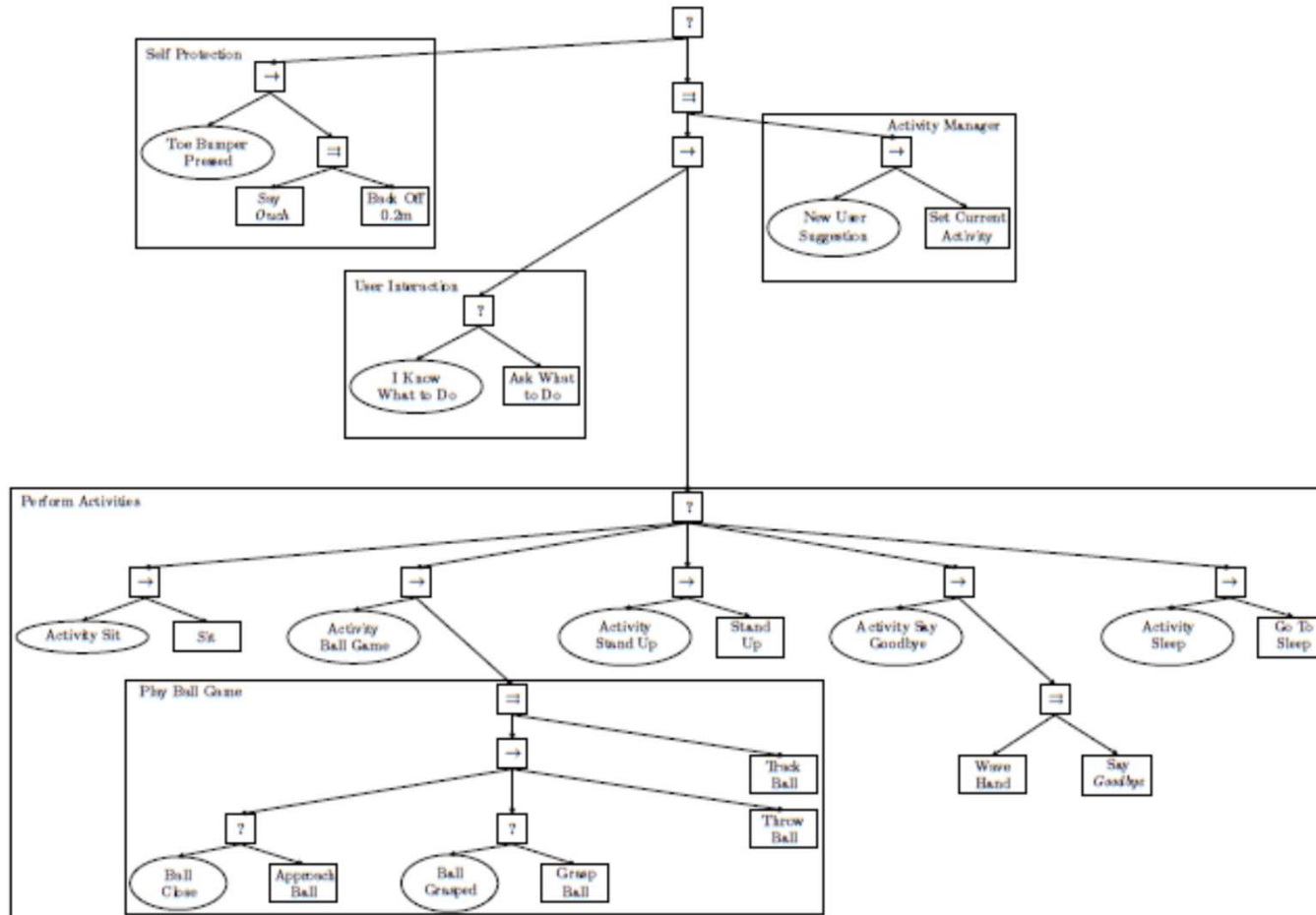


- **Maintainability:** Adding or removing states is complex. Long sequence of actions and interactions requires a fully connected subgraph
- **Manually created hierarchy:** The hierarchy resolves some problems, but a reactive HFSM results in some sub graphs being fully connected with many possible transitions

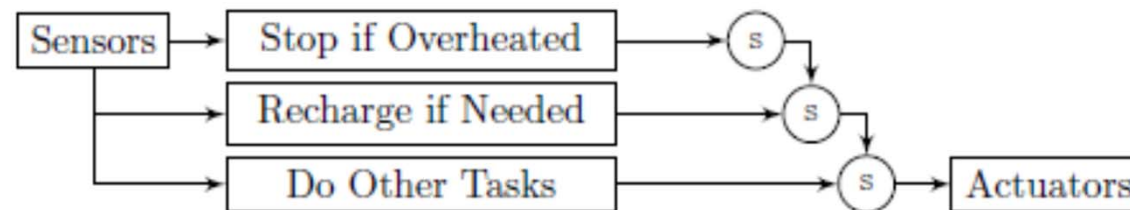
# BTs vs HFSMs



# BTs vs HFSMs

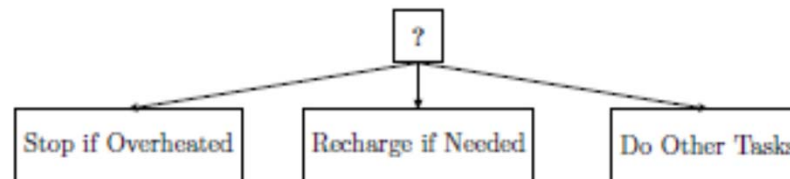


# BTs vs Subsumption



Fallback composition can be used to obtain an equivalent BT:

- Given a Subsumption architecture, an equivalent BT can be obtained by arranging the controllers as actions under a Fallback composition, in order from higher to lower priority
- the return status of the actions be Failure (if they do not need to execute) or Running. They never return Success



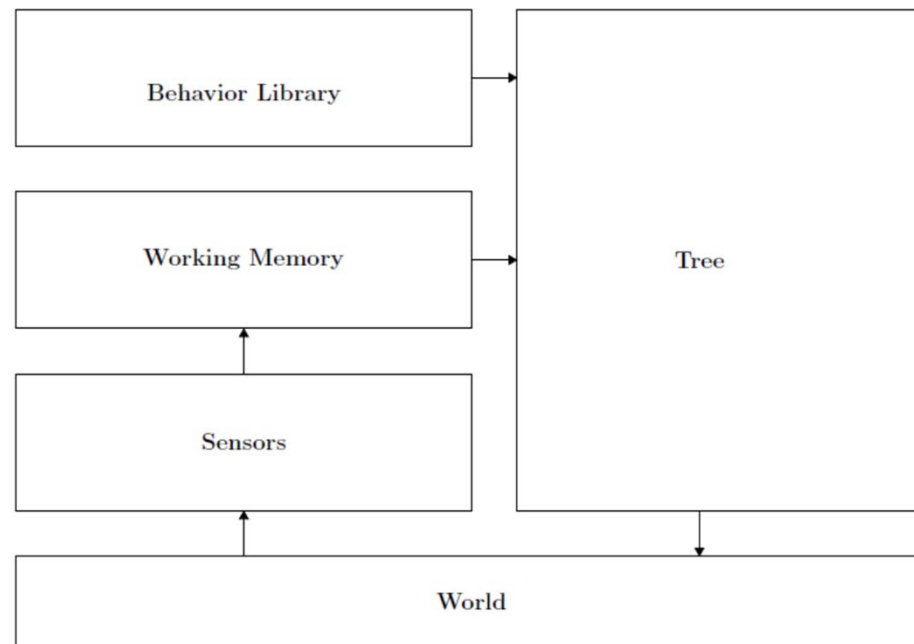


# ABL

Planning with a Behavior Language (ABL) [BG Weber et al 2011] for the façade video game:

- Players interact with the couple by wandering through their apartment and interacting with objects or by chatting with them directly

ABL Agent:



Michael Mateas and Andrew Stern. A Behavior Language for Story-based Believable Agents

# ABL

Planning with a Behavior Language (ABL) [BG Weber et al 2011] for the façade video game:

- Players interact with the couple by wandering through their apartment and interacting with objects or by chatting with them directly

ABL Agent:

```
sequential behavior OpenDoor() {
  precondition {
    (KnockWME doorID :: door)
    (PosWME spriteID == door pos :: doorPos)
    (PosWME spriteID == me pos :: myPos)
    (Util.computeDistance(doorPos, myPos) > 100)
  }
  specificity 2;
  // Too far to walk, yell for knocker to come in
  subgoal YellAndWaitForGuestToEnter(doorID);
}

sequential behavior OpenDoor() {
  precondition { (KnockWME doorID :: door) }
  specificity 1;
  // Default behavior - walk to door and open
  . . .
}
```



```
parallel behavior
YellAndWaitForGuestToEnter(int doorID) {
  precondition { (CurrentTimeWME t :: startT) }
  context_condition {
    (CurrentTimeWME t <= startT + 10000) }
  number_needed_for_success 1;

  with success_test {
    (DoorOpenWME door == doorID) } wait;
  with (persistent) subgoal YellForGuest(doorID);
}
```

Michael Mateas and Andrew Stern. A Behavior Language for Story-based Believable Agents

# ABL

Planning with a Behavior Language (ABL) [BG Weber et al 2011]

ABL Agent:

---

**Algorithm 8:** main loop - input(initial ABL tree)

---

```
1  $\mathcal{T} \leftarrow \text{ParallelNode}$ 
2 for subgoal in initial - tree do
3   |  $\mathcal{T}_g \leftarrow \text{GetBT}(\text{subgoal})$ 
4   |  $\mathcal{T}.\text{AddChild}(\mathcal{T}_g)$ 
5 while True do
6   |  $\text{Execute}(\mathcal{T})$ 
```

---

# ABL

## Planning with a Behavior Language (ABL) [BG Weber et al 2011]

ABL Agent:

---

**Algorithm 9:** GetBT - input(goal)

---

```
1  $\mathcal{T}_g \leftarrow \emptyset$ 
2 if goal.behavior is sequential then
3   |  $\mathcal{T}_g \leftarrow \text{SequenceNode}$ 
4 else
5   |  $\mathcal{T}_g \leftarrow \text{ParallelNode}$ 
6 Instructions  $\leftarrow \text{GetInstructions}(\text{goal})$ 
7 for instruction in Instructions do
8   | switch instruction do
9     | case act do
10    | |  $\mathcal{T}_g.\text{AddChild}(\text{ActionNode}(\text{act}))$ 
11    | case mental act do
12    | |  $\mathcal{T}_g.\text{AddChild}(\text{ActionNode}(\text{mental act}))$ 
13    | case spawngoal do
14    | |  $\mathcal{T}_g.\text{AddChild}(\text{PlaceholderNode}(\text{spawngoal}))$ 
15 if goal.precondition is not empty then
16   |  $\mathcal{T}_{g'} \leftarrow \text{SequenceNode}$ 
17   | for proposition in precondition do
18   | |  $\mathcal{T}_{g'}.\text{AddChild}(\text{ConditionNode}(\text{proposition}))$ 
19   | |  $\mathcal{T}_{g'}.\text{AddChild}(\mathcal{T}_g)$ 
20   | return  $\mathcal{T}_{g'}$ 
21 else
22   | return  $\mathcal{T}_g$ 
```

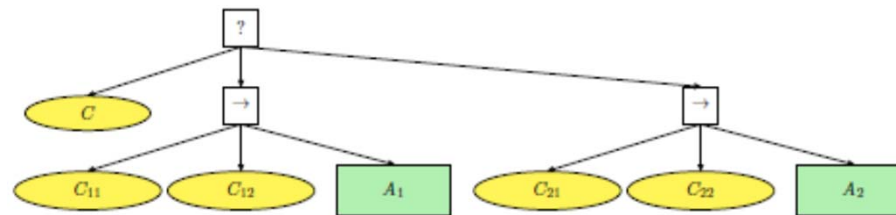
---

# PA-BT

## Planning and Acting using Behavior Trees (PA-BT)

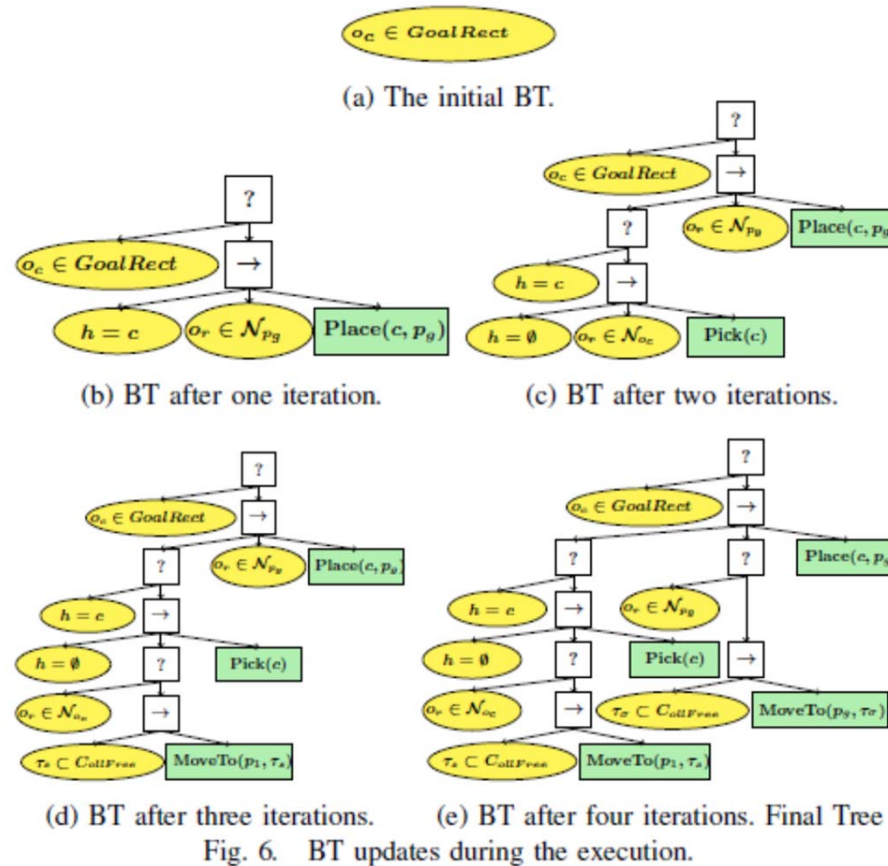
Actions	Preconditions	Postconditions
$A_1$	$C_{11}^{Pre}, C_{12}^{Pre}, \dots$	$C_{11}^{Post}, C_{12}^{Post}, \dots$
$A_2$	$C_{21}^{Pre}, C_{22}^{Pre}, \dots$	$C_{21}^{Post}, C_{22}^{Post}, \dots$
$\vdots$	$\vdots$	$\vdots$

$C_1^{Goal}, C_2^{Goal}, \dots$



# PA-BT

## Planning and Acting using Behavior Trees (PA-BT)



# PA-BT

## Planning and Acting using Behavior Trees (PA-BT)

---

**Algorithm 1:** Main Loop, finding conditions to expand and resolve conflicts

---

```
1  $\mathcal{T} \leftarrow \emptyset$ 
2 for  $c$  in  $\mathcal{C}_{goal}$  do
3    $\mathcal{T} \leftarrow \text{SequenceNode}(\mathcal{T}, c)$ 
4 while True do
5    $T \leftarrow \text{RefineActions}(T)$ 
6   do
7      $r \leftarrow \text{Tick}(T)$ 
8     while  $r \neq \text{Failure}$ 
9        $c_f \leftarrow \text{GetConditionToExpand}(T)$ 
10       $\mathcal{T}, \mathcal{T}_{new\_subtree} \leftarrow \text{ExpandTree}(\mathcal{T}, c_f)$ 
11      while  $\text{Conflict}(\mathcal{T})$  do
12         $\mathcal{T} \leftarrow \text{IncreasePriority}(\mathcal{T}_{new\_subtree})$ 
```

---

---

**Algorithm 2:** Replace failed condition with new Atomic BT

---

```
1 Function  $\text{ExpandTree}(\mathcal{T}, c_f)$ 
2    $A_T \leftarrow \text{GetAllActTemplatesFor}(c_f)$ 
3    $\mathcal{T}_{fall} \leftarrow c_f$ 
4   for  $a$  in  $A_T$  do
5      $\mathcal{T}_{seq} \leftarrow \emptyset$ 
6     for  $c_a$  in  $a.con$  do
7        $\mathcal{T}_{seq} \leftarrow \text{SequenceNode}(\mathcal{T}_{seq}, c_a)$ 
8      $\mathcal{T}_{seq} \leftarrow \text{SequenceNode}(\mathcal{T}_{seq}, a)$ 
9      $\mathcal{T}_{fall} \leftarrow \text{FallbackNode}(\mathcal{T}_{fall}, \mathcal{T}_{seq})$ 
10   $\mathcal{T} \leftarrow \text{Substitute}(\mathcal{T}, c_f, \mathcal{T}_{fall})$ 
11  return  $\mathcal{T}, \mathcal{T}_{fall}$ 
```

---

Back chaining: starting from a goal condition select actions to achieve that goal

# PA-BT

## Planning and Acting using Behavior Trees (PA-BT)

---

**Algorithm 1:** Main Loop, finding conditions to expand and resolve conflicts

---

```
1  $\mathcal{T} \leftarrow \emptyset$ 
2 for  $c$  in  $\mathcal{C}_{goal}$  do
3    $\mathcal{T} \leftarrow \text{SequenceNode}(\mathcal{T}, c)$ 
4 while True do
5    $T \leftarrow \text{RefineActions}(T)$ 
6   do
7      $r \leftarrow \text{Tick}(T)$ 
8     while  $r \neq \text{Failure}$ 
9        $c_f \leftarrow \text{GetConditionToExpand}(T)$ 
10       $\mathcal{T}, \mathcal{T}_{new\_subtree} \leftarrow \text{ExpandTree}(T, c_f)$ 
11      while  $\text{Conflict}(T)$  do
12         $\mathcal{T} \leftarrow \text{IncreasePriority}(\mathcal{T}_{new\_subtree})$ 
```

---

---

**Algorithm 3:** Get Condition to Expand

---

```
1 Function  $\text{GetConditionToExpand}(T)$ 
2   for  $c_{next}$  in  $\text{GetConditionsBFS}()$  do
3     if  $c_{next}.status = \text{Failure}$  and
4        $c_{next} \notin \text{ExpandedNodes}$  then
5          $\text{ExpandedNodes.push\_back}(c_{next})$ 
6         return  $c_{next}$ 
7   return None
```

---

Back chaining: starting from a goal condition select actions to achieve that goal



# Behavior Networks

Hierarchical Behavior-based systems [Nicolescu and Mataric 2002]

Abstract and Primitive Behaviors

- **world preconditions** - conditions that activate the behaviors based on a particular state of the environment.
- **sequential preconditions** - task-dependent conditions that must be met before activating the behavior:
  - *Permanent preconditions*: preconditions that must be met during the entire execution of the behavior
  - *Enabling preconditions*: preconditions that must be met immediately before the activation of a behavior
  - *Ordering constraints*: preconditions that must have been met at some point before the behavior is activated
- **Sequential execution**: for the task segments containing temporal ordering constraints;
- **Opportunistic execution**: for the task segments for which the order of execution does not matter.

# Behavior Networks

Hierarchical Behavior-based systems [Nicolescu and Mataric 2002]

Abstract and Primitive Behaviors

**NETWORK-DESCRIPTION =**  
    < Number of components (N),  
      { Component-Description }<sub>N</sub>,  
      Topology-Description >

where,

**Component-Description =**  
    < AB-Description | ABN-Description >

**AB-Description =**  
    < Component-ID,  
      BehaviorID, Number of Parameters (P),  
      { Parameter Name, Parameter Value }<sub>P</sub> >

**NAB-Description =**  
    < Component-ID,  
      NETWORK-DESCRIPTION >

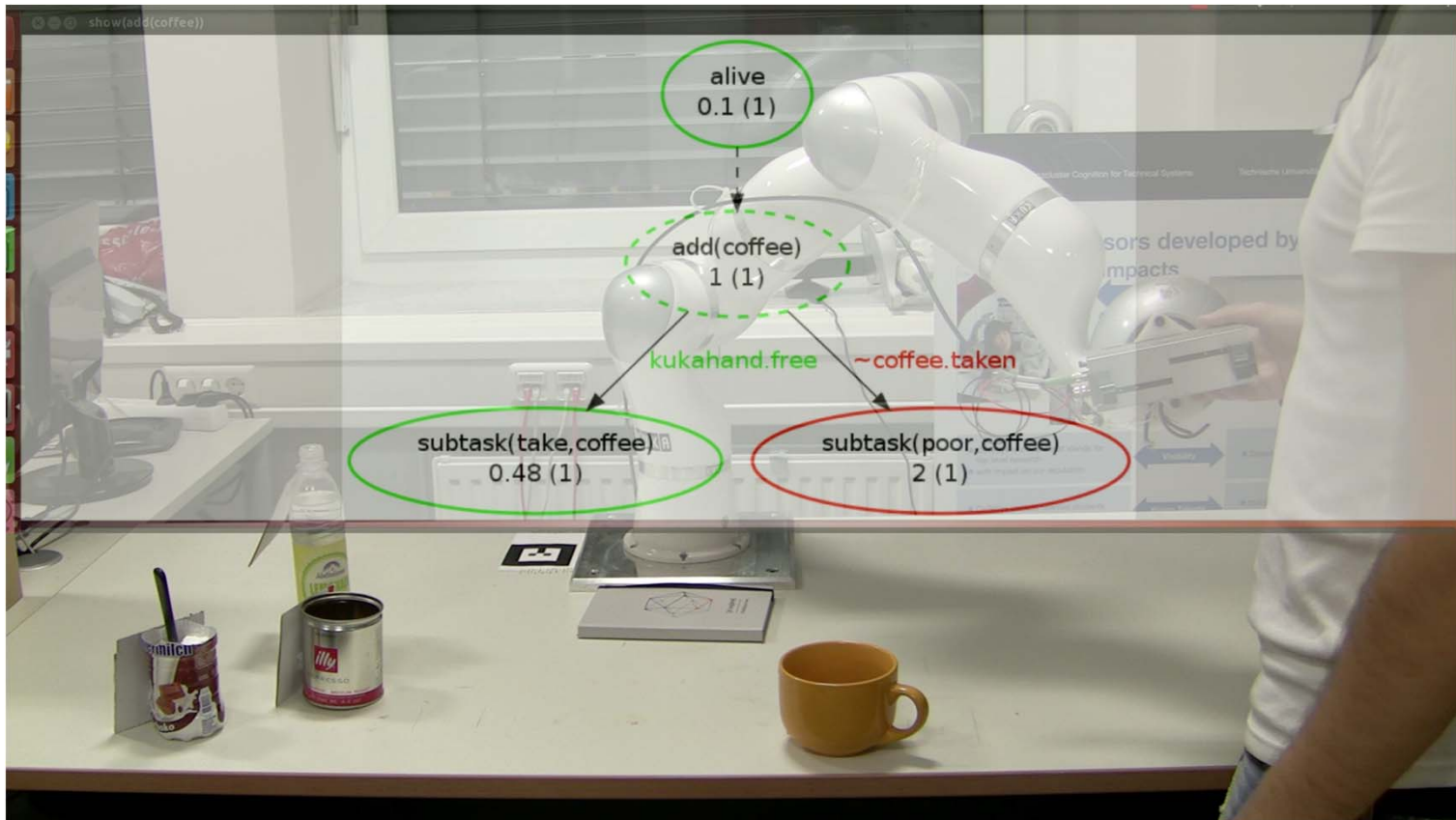
**Topology-Description =**  
    < Number of Links (L),  
      { FromComp-ID, ToComp-ID, Link-Type }<sub>L</sub> >

**Link-Type =**  
    < Ordering | Enabling | Permanent >

# Behavior Hierarchy

Hierarchical Behavior-based systems [Caccavale Finzi 2015, 2017, 2019]

Abstract and Primitive Behaviors (and activations)



# BDI Systems

[Bratman,1987]. Intention, Plans, and Practical Reason.

- BDI model inspired by the Michael Bratman's theory of human practical reasoning:
  - resource-bounded agent
  - intention and desire are proactive, intentions as commitments

## Core concepts

Beliefs = information the agent has about the world

Desires = state of affairs that the agent would wish to bring about

Intentions = desires (or actions) that the agent has committed to achieve

**Belief:** the agent knowledge about about the world (belief set)

**Desires:** motivational state, objectives, tasks to be acheived (goals)

**Intentions:** desires with commitment, i.e. plans ready for the execution (plans)

# BDI Systems

BDI particularly compelling because:

- **philosophical component** - based on a theory of rational actions in humans
- **software architecture** - it has been implemented and successfully used in a number of complex fielded applications
  - IRMA - Intelligent Resource-bounded Machine Architecture
  - PRS - Procedural Reasoning System
- **logical component** - the model has been rigorously formalized in a family of BDI logics
  - Rao & Georgeff, Wooldrige
  - $(\text{Int } A_i \varphi) \rightarrow \neg (\text{Bel } A_i \varphi)$

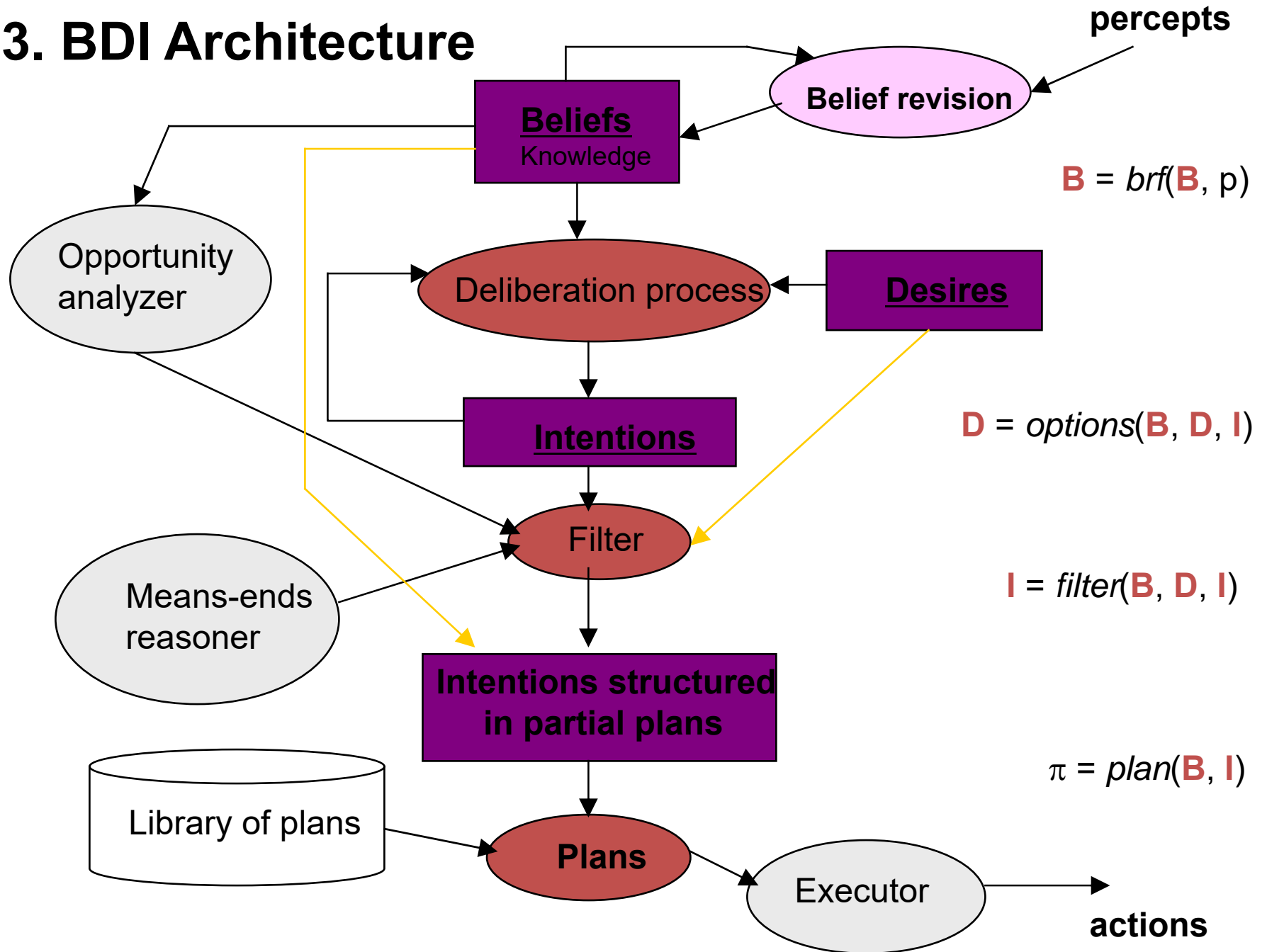
# Practical Reasoning Agents: Deliberation: Intentions and Desires

- intentions are stronger than desires
  - “My desire to play basketball this afternoon is merely a potential influencer of my conduct this afternoon. It must vie with my other relevant desires [. . . ] before it is settled what I will do. In contrast, once I intend to play basketball this afternoon, the matter is settled: I normally need not continue to weigh the pros and cons. When the afternoon arrives, I will normally just proceed to execute my intentions.” [*Bratman, 1990*]

# Practical Reasoning Agents: Intentions

1. agents are expected to **determine ways of achieving** intentions
  - *If I have an intention to  $\Phi$ , you would expect me to devote resources to deciding how to bring about  $\Phi$*
2. agents **cannot** adopt intentions which **conflict**
  - *If I have an intention to  $\Phi$ , you would not expect me to adopt an intention  $\Psi$  that was incompatible with  $\Phi$*
3. agents are inclined to **try again** if their attempts to achieve their intention fail
  - *If an agent's first attempt to achieve  $\Phi$  fails, then all other things being equal, it will try an alternative plan to achieve  $\Phi$*
4. agents **believe** their intentions are **possible**
  - *That is, they believe there is at least some way that the intentions could be brought about.*
5. agents do **not believe** they will **not bring about** their intentions
  - *It would not be rational of me to adopt an intention to  $\Phi$  if I believed that I would fail with  $\Phi$*
6. under certain circumstances, agents **believe** they **will bring about** their intentions
  - *If I intend  $\Phi$ , then I believe that under "normal circumstances" I will succeed with  $\Phi$*
7. agents need **not intend** all the expected **side effects** of their intentions
  - *I may believe that going to the dentist involves pain, and I may also intend to go to the dentist — but this does not imply that I intend to suffer pain!*

### 3. BDI Architecture





# Practical Reasoning Agents

- agent control loop

```
while true
  observe the world;
  update internal world model;
  deliberate about what intention to achieve next;
  use means-ends reasoning to get a plan for the
  intention;
  execute the plan
end while
```

# Practical Reasoning Agents

- agent control loop

```
while true
```

```
  observe the world;
```

```
  update internal world model;
```

```
  deliberate about what intention to achieve next;
```

```
  use means-ends reasoning to get a plan for the  
  intention;
```

```
  execute the plan
```

```
end while
```

- what are the options (desires) ?
- how to choose an option ?
- incl. filter
- chosen option → intention ...



# Practical Reasoning Agents

- agent control loop

```
while true
```

```
  observe the world;
```

```
  update internal world model;
```

```
  deliberate about what intention to achieve next;
```

```
  use means-ends reasoning to get a plan for the  
  intention;
```

```
  execute the plan
```

```
end while
```

- what are the options (desires) ?  
- how to choose an option ?  
- incl. filter  
- chosen option → intention ...

- when to reconsider intentions !?

# Implementing Practical Reasoning Agents

- Let's make the algorithm more formal:

```
Agent Control Loop Version 2
1.   $B := B_0$ ; /* initial beliefs */
2.  while true do
3.      get next percept  $\rho$ ;
4.       $B := brf(B, \rho)$ ;
5.       $I := deliberate(B)$ ;
6.       $\pi := plan(B, I)$ ;
7.      execute( $\pi$ )
8.  end while
```

# Implementing Practical Reasoning Agents

- Optimal behaviour if
  - deliberation and means-ends reasoning take a small amount of time;
  - the world is guaranteed to remain static while the agent is deliberating and performing means-ends reasoning;
  - an intention that is optimal when achieved at time  $t_0$  (the time at which the world is observed) is guaranteed to remain optimal until time  $t_2$  (the time at which the agent has found a course of action to achieve the intention).

# Deliberation

- The *deliberate* function can be decomposed into two distinct functional components:
  - *option generation*  
the agent generates a set of possible alternatives. A function, *options*, takes the agent's current beliefs and current intentions, and from them determines a set of options (= *desires*)
  - *filtering*  
the agent chooses between competing alternatives, and commits to achieving them. In order to select between competing options, an agent uses a *filter* function.

# Deliberation

Agent Control Loop Version 3

```
1.
2.    $B := B_0;$ 
3.    $I := I_0;$ 
4.   while true do
5.       get next percept  $\rho;$ 
6.        $B := brf(B, \rho);$ 
7.        $D := options(B, I);$  |
8.        $I := filter(B, D, I);$  | ←
9.        $\pi := plan(B, I);$ 
10.      execute( $\pi$ )
11.  end while
```

# Practical Reasoning Agents

If an option has successfully passed through the filter function and is chosen by the agent as an intention, we say that **the agent has made a commitment to that option.**

Commitment implies temporal persistence of intentions; once an intention is adopted, it should not be immediately dropped out.

How committed an agent should be to its intentions?

- degrees of commitments
  - blind commitment
    - »  $\approx$  fanatical commitment: continue until achieved
  - single-minded commitment
    - » continue until achieved or no longer possible
  - open-minded commitment
    - » continue until no longer believed possible



# Commitment Strategies

- An agent has commitment both
  - to *ends* (i.e., the wishes to bring about)
  - and *means* (i.e., the mechanism via which the agent wishes to achieve the state of affairs)
- current version of agent control loop is overcommitted, both to means and ends
  - modification: *replan* if ever a plan goes wrong

## Agent Control Loop Version 4

```
1.
2.   $B := B_0;$ 
3.   $I := I_0;$ 
4.  while true do
5.      get next percept  $\rho;$ 
6.       $B := brf(B, \rho);$ 
7.       $D := options(B, I);$ 
8.       $I := filter(B, D, I);$ 
9.       $\pi := plan(B, I);$ 
10.     while not empty( $\pi$ ) do
11.          $\alpha := hd(\pi);$ 
12.         execute( $\alpha$ );
13.          $\pi := tail(\pi);$ 
14.         get next percept  $\rho;$ 
15.          $B := brf(B, \rho);$ 
16.         if not sound( $\pi, I, B$ ) then ← Reactivity, replan
17.              $\pi := plan(B, I)$ 
18.         end-if
19.     end-while
20. end-while
```

“Blind commitment”

# Commitment Strategies

- this version still overcommitted to intentions:
  - never stops to consider whether or not its intentions are appropriate

→ modification: stop for determining whether

intentions have succeeded or whether they are impossible:


*“Single-minded commitment”*

# Single-minded Commitment

Agent Control Loop Version 5

```
2.   $B := B_0$ ;  
3.   $I := I_0$ ;  
4.  while true do  
5.      get next percept  $\rho$ ;  
6.       $B := brf(B, \rho)$ ;  
7.       $D := options(B, I)$ ;  
8.       $I := filter(B, D, I)$ ;  
9.       $\pi := plan(B, I)$ ;  
10.     while not  $empty(\pi)$   
        or  $succeeded(I, B)$   
        or  $impossible(I, B)$  do  
11.          $\alpha := hd(\pi)$ ;  
12.          $execute(\alpha)$ ;  
13.          $\pi := tail(\pi)$ ;  
14.         get next percept  $\rho$ ;  
15.          $B := brf(B, \rho)$ ;  
16.         if not  $sound(\pi, I, B)$  then  
17.              $\pi := plan(B, I)$   
18.         end-if  
19.     end-while  
20. end-while
```

*Dropping intentions  
that are impossible  
or have succeeded*



*Reactivity, replan*



# Intention Reconsideration

- Our agent gets to reconsider its intentions when:
  - it has completely executed a plan to achieve its current intentions; or
  - it believes it has achieved its current intentions; or
  - it believes its current intentions are no longer possible.
- This is limited in the way that it permits an agent to *reconsider* its intentions
  - modification:  
Reconsider intentions after executing every action

*“Open-minded commitment”*

## Agent Control Loop Version 6

```
1.
2.   $B := B_0;$ 
3.   $I := I_0;$ 
4.  while true do
5.      get next percept  $\rho;$ 
6.       $B := brf(B, \rho);$ 
7.       $D := options(B, I);$ 
8.       $I := filter(B, D, I);$ 
9.       $\pi := plan(B, I);$ 
10.     while not ( $empty(\pi)$ 
11.             or  $succeeded(I, B)$ 
12.             or  $impossible(I, B)$ ) do
13.          $\alpha := hd(\pi);$ 
14.          $execute(\alpha);$ 
15.          $\pi := tail(\pi);$ 
16.         get next percept  $\rho;$ 
17.          $B := brf(B, \rho);$ 
18.          $D := options(B, I);$ 
19.          $I := filter(B, D, I);$ 
20.         if not  $sound(\pi, I, B)$  then
21.              $\pi := plan(B, I)$ 
22.         end-if
23.     end-while
24. end-while
```

## Open-minded Commitment

# Intention Reconsideration

- But intention reconsideration is *costly*!  
A dilemma:
  - an agent that does not stop to reconsider its intentions sufficiently often will continue attempting to achieve its intentions even after it is clear that they cannot be achieved, or that there is no longer any reason for achieving them
  - an agent that *constantly* reconsiders its intentions may spend insufficient time actually working to achieve them, and hence runs the risk of never actually achieving them
- Solution: incorporate an explicit *meta-level control* component, that decides whether or not to reconsider

## Agent Control Loop Version 7

```
1.
2.   $B := B_0;$ 
3.   $I := I_0;$ 
4.  while true do
5.      get next percept  $\rho;$ 
6.       $B := brf(B, \rho);$ 
7.       $D := options(B, I);$ 
8.       $I := filter(B, D, I);$ 
9.       $\pi := plan(B, I);$ 
10.     while not ( $empty(\pi)$ 
11.                or  $succeeded(I, B)$ 
12.                or  $impossible(I, B)$ ) do
13.          $\alpha := hd(\pi);$ 
14.          $execute(\alpha);$ 
15.          $\pi := tail(\pi);$ 
16.         get next percept  $\rho;$ 
17.          $B := brf(B, \rho);$ 
18.         if  $reconsider(I, B)$  then
19.              $D := options(B, I);$ 
20.              $I := filter(B, D, I);$ 
21.         end-if
22.         if not  $sound(\pi, I, B)$  then
23.              $\pi := plan(B, I)$ 
24.         end-if
25.     end-while
26. end-while
```

*meta-level control*





# Possible Interactions

- The possible interactions between meta-level control and deliberation are:

Situation number	Chose to deliberate?	Changed intentions?	Would have changed intentions?	<i>reconsider(...)</i> optimal?
1	No	—	No	Yes
2	No	—	Yes	No
3	Yes	No	—	No
4	Yes	Yes	—	Yes

# Intention Reconsideration

- Situations
  - In situation (1), the agent did not choose to deliberate, and as consequence, did not choose to change intentions.  
Moreover, if it *had* chosen to deliberate, it would not have changed intentions.  
the *reconsider(...)* function is behaving optimally.
  - In situation (2), the agent did not choose to deliberate, but if it had done so, it *would* have changed intentions.  
the *reconsider(...)* function is not behaving optimally.
  - In situation (3), the agent chose to deliberate, but did not change intentions.  
the *reconsider(...)* function is not behaving optimally.
  - In situation (4), the agent chose to deliberate, and did change intentions.  
the *reconsider(...)* function is behaving optimally.
- An important assumption: cost of *reconsider(...)* is *much* less than the cost of the deliberation process itself.

# Optimal Intention Reconsideration

- Kinny and Georgeff's experimentally investigated effectiveness of intention reconsideration strategies
- Two different types of reconsideration strategy were used:
  - *bold* agents  
never pause to reconsider intentions, and
  - *cautious* agents  
stop to reconsider after every action
- *Dynamism* in the environment is represented by the *rate of world change*,  $\gamma$

# Optimal Intention Reconsideration

- Results (not surprising):
  - If  $\gamma$  is low (i.e., the environment does not change quickly), bold agents do well compared to cautious ones.
    - cautious ones waste time reconsidering their commitments while bold agents are busy working towards — and achieving — their intentions.
  - If  $\gamma$  is high (i.e., the environment changes frequently), cautious agents tend to outperform bold agents.
    - they are able to recognize when intentions are doomed, and also to take advantage of serendipitous situations and new opportunities when they arise.

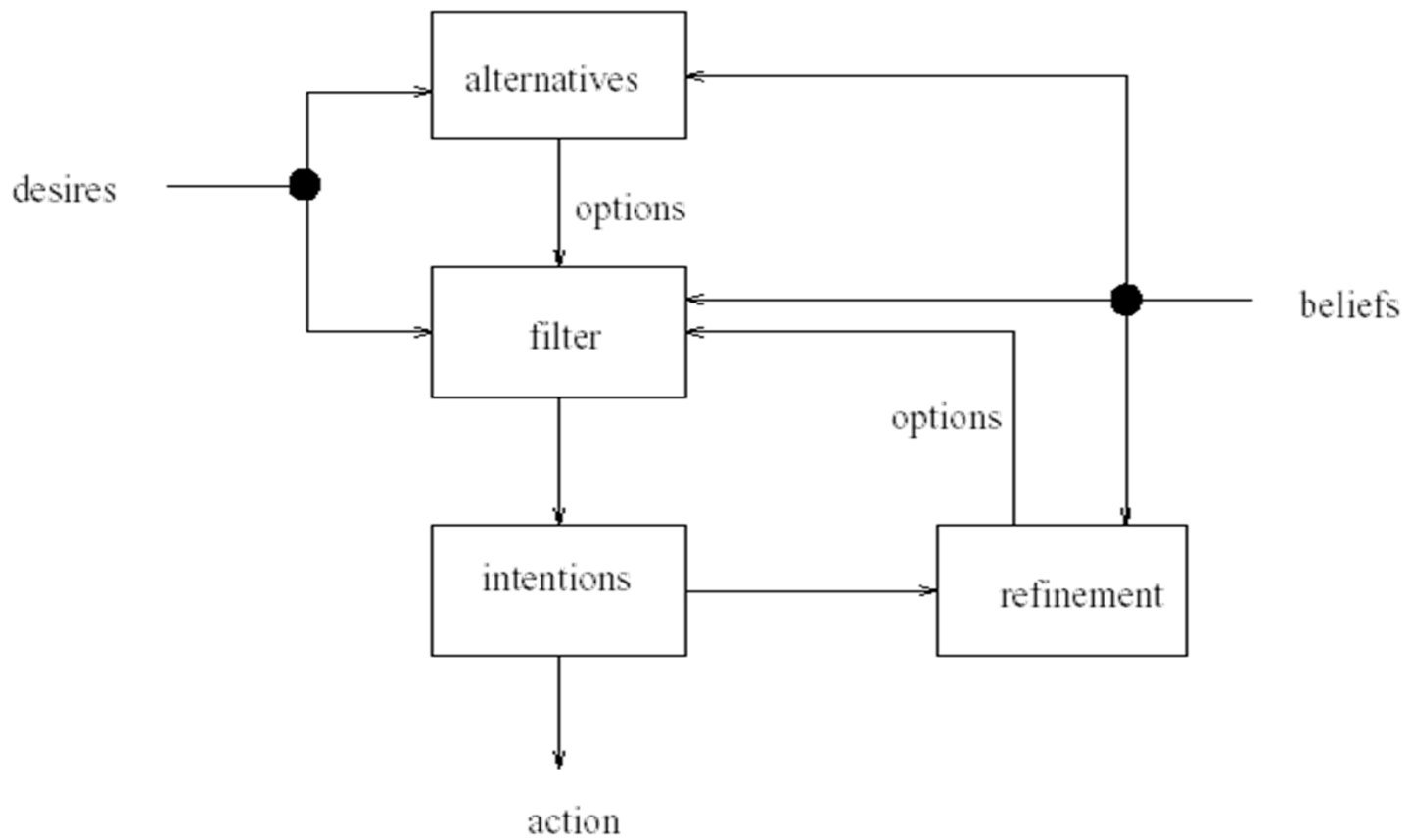
# Implemented BDI Agents: IRMA

- IRMA — Intelligent Resource-bounded Machine Architecture —  
Bratman, Israel, Pollack
- IRMA has four key symbolic data structures:
  - a *plan library*
  - explicit representations of
    - *beliefs*: information available to the agent — may be represented symbolically, but may be simple variables
    - *desires*: those things the agent would *like* to make true — think of desires as *tasks* that the agent has been allocated;
    - *intentions*: desires that the agent has *chosen* and *committed to*

# IRMA

- Additionally, the architecture has:
  - a *reasoner*
    - for reasoning about the world; an inference engine
  - a *means-ends analyzer*
    - determines which plans might be used to achieve intentions
  - an *opportunity analyzer*
    - monitors the environment, and as a result of changes, generates new options
  - a *filtering process*
    - determines which options are compatible with current intentions
  - a *deliberation process*
    - responsible for deciding upon the ‘best’ intentions to adopt

# IRMA



# Practical Reasoning Agents: Procedural Reasoning System (PRS)

– “BDI-architecture” (*beliefs / desires / intentions*)

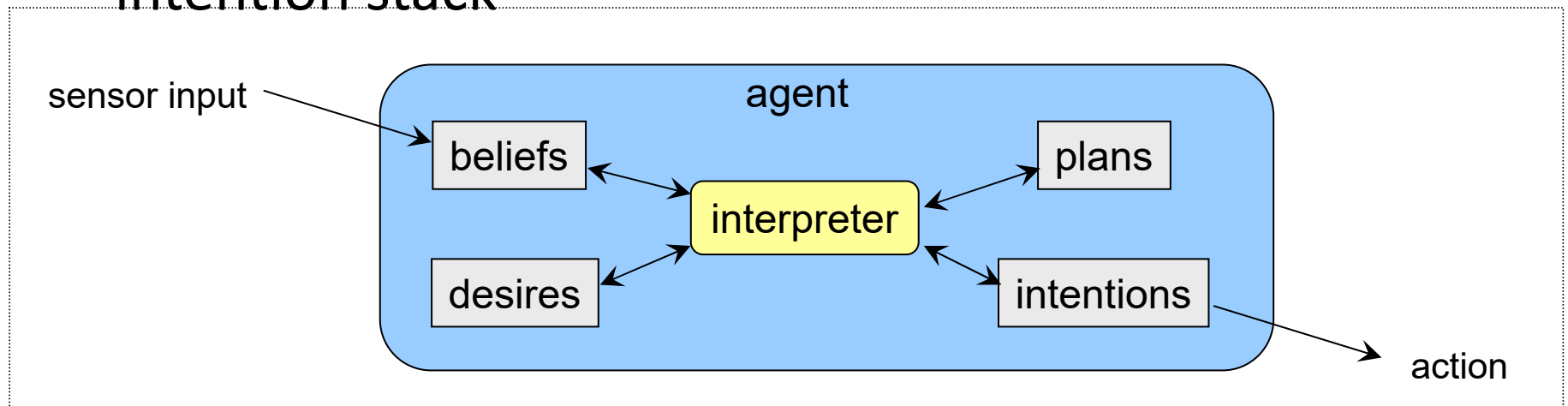
– explicit data structures for b/d/i

– planning

– no “on-the-fly” planning → plan libraries

» a plan:            goal            (post-condition)  
                         context            (pre-condition)  
                         body            (sequence of actions  
   / subgoals)

– intention stack





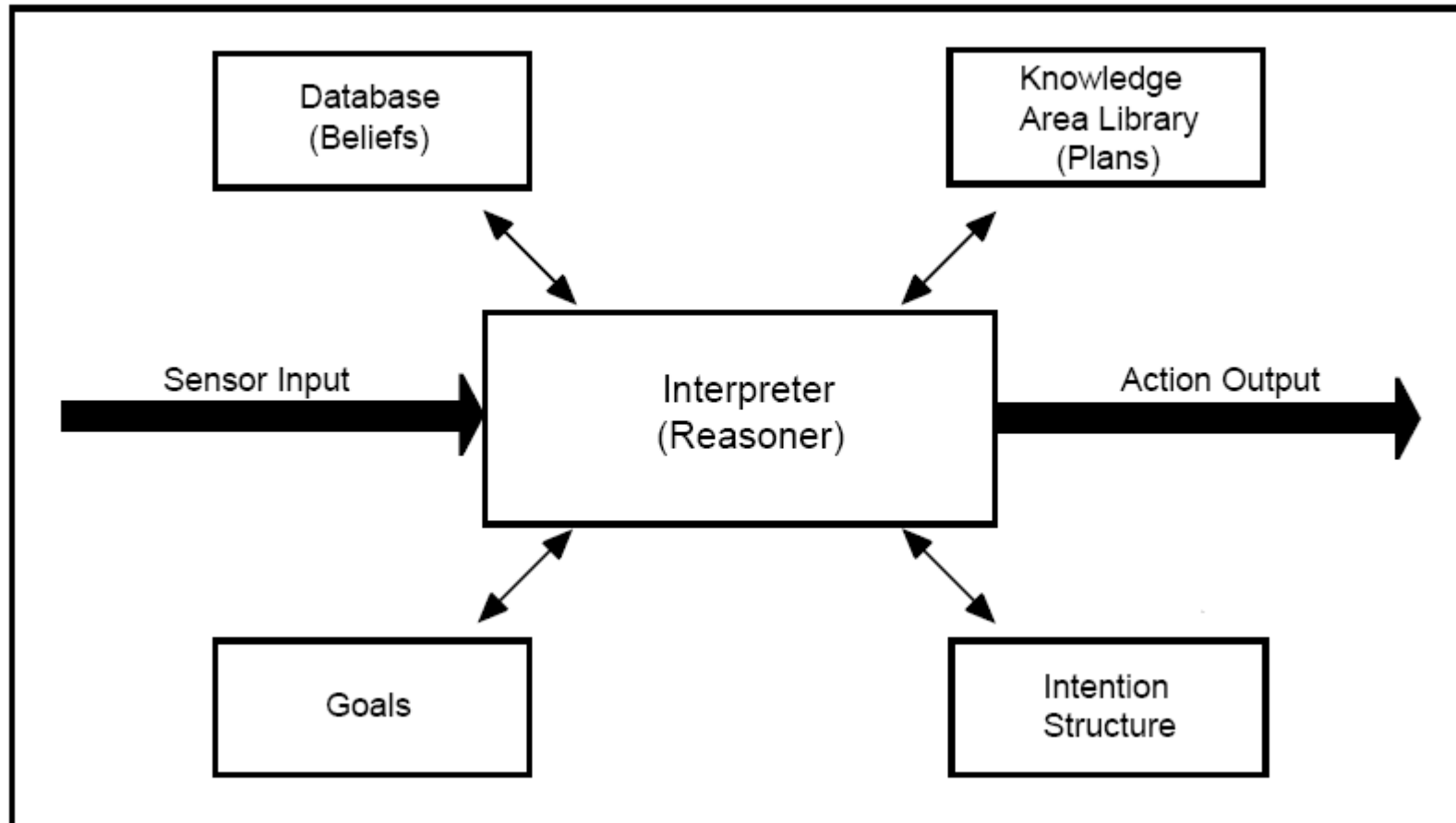
# Procedural Reasoning System (PRS)

- Framework for symbolic reactive control systems in dynamic environments
  - Eg. Mobile robot control
  - Eg. Diagnosis of the Space Shuttle's Reaction Controls System

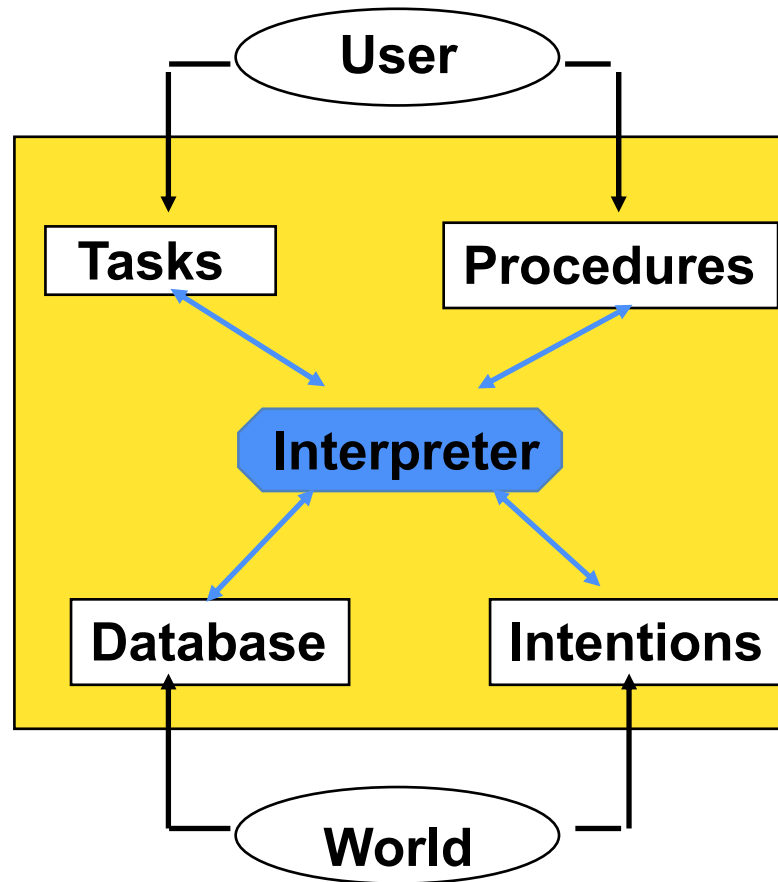
# PRS: Main Features

- Pre-compiled procedural knowledge
- BDI (Belief, Desires, Intentions) foundation
- Combines deliberative and reactive features
  - Plan selection, formation, execution, sensing
- Plans dynamically and incrementally
- Integrates goal-directed and event-driven behavior
- Can interrupt plan execution
- Meta-level reasoning
- Multi-agent planning

# PRS Architecture

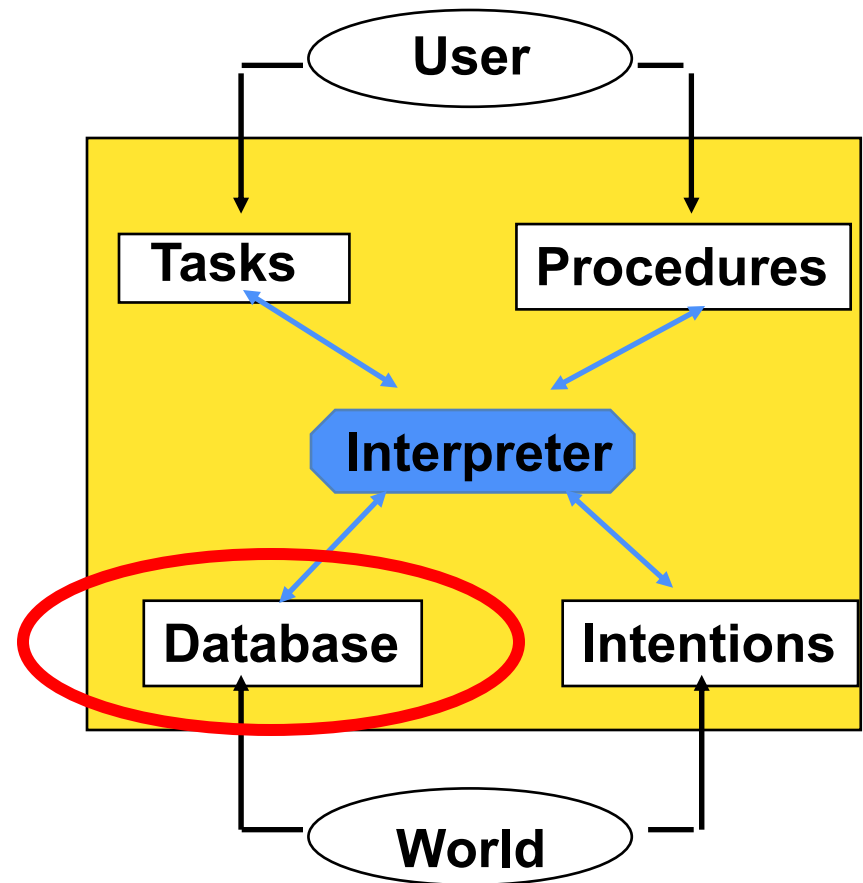


# PRS Architecture



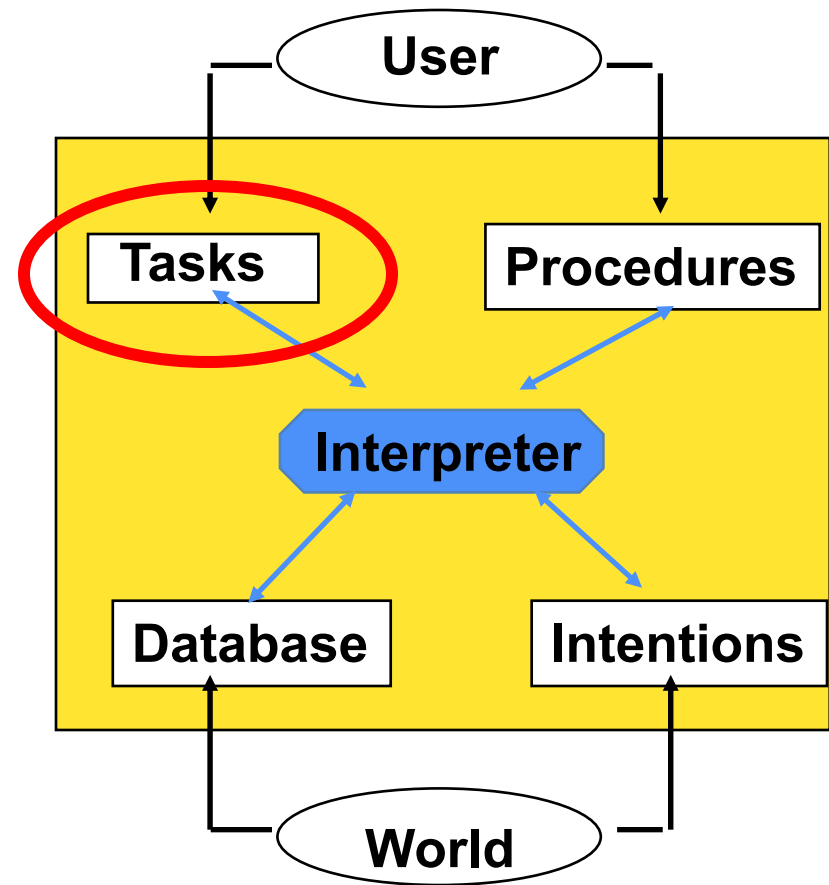
# PRS Architecture: Database

- Contains beliefs or facts about the world
- Includes meta-level information
  - Eg goal G is active



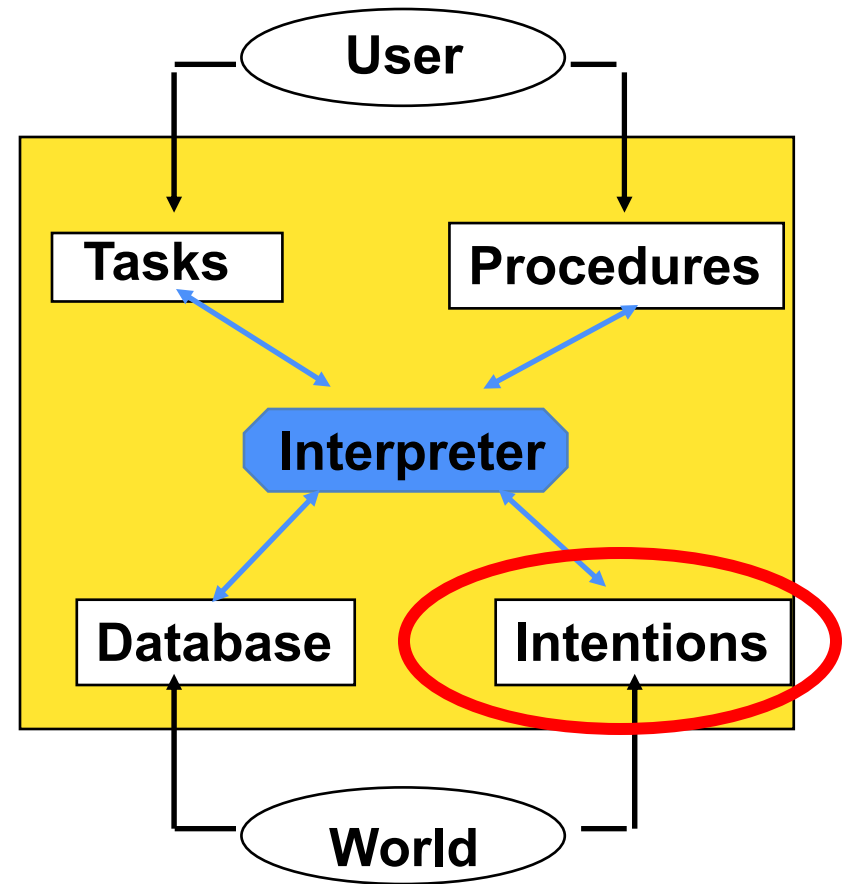
# PRS Architecture: Tasks

- Represent desired behavior
- Conditions over some time interval
  - eg (walk a b): set of behaviors in which agent walks from a to b)



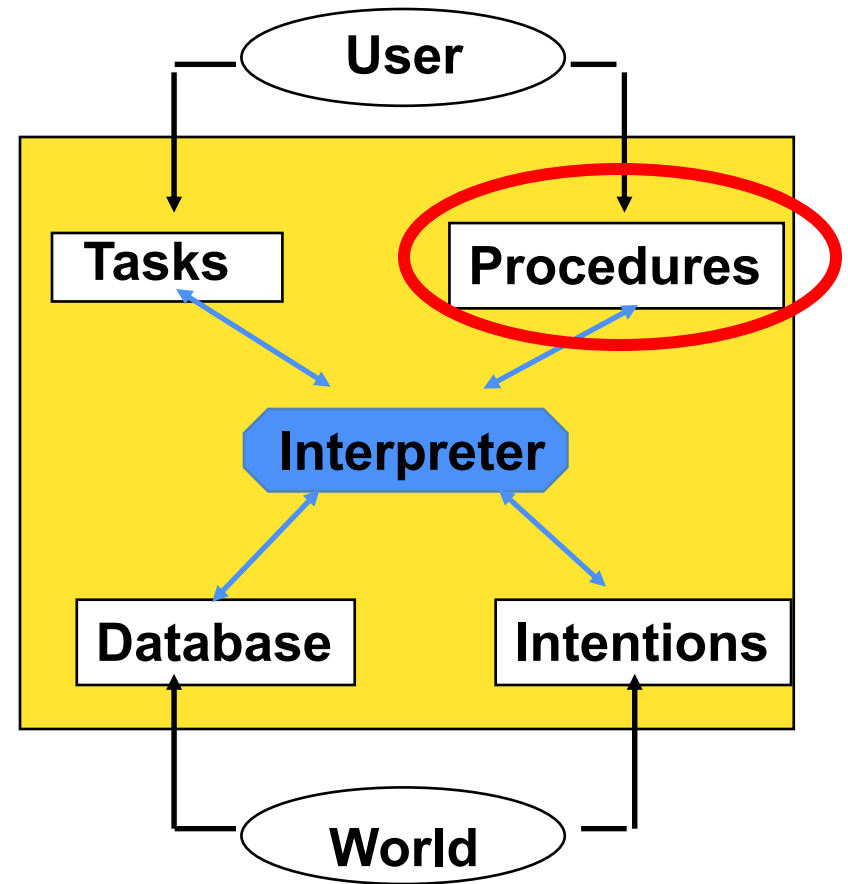
# PRS Architecture: Intentions

- Currently active procedures
- Procedure currently being executed



# PRS Architecture: Procedures

- Pre-compiled procedures
- Express actions and tests to achieve goals or to react to conditions





# PRS

- Beliefs, goals, intentions and plan library
- Agent perceive the world through external events
- Plans – procedural knowledge
  - Recipes for action (tree labeled with actions and formulas which evaluate to a boolean)
  - Trigger (what an agent must perceive)
  - Context (what an agent must believe)
  - If plans cannot proceed they post internal events

# From Plans to Intentions

- Agents respond to Internal and External events by selecting an appropriate plan in its plan based whose trigger and context is true
- When a plan is adopted it becomes an intention
- This intention become part of the agent's intention structure

# PRS Operation

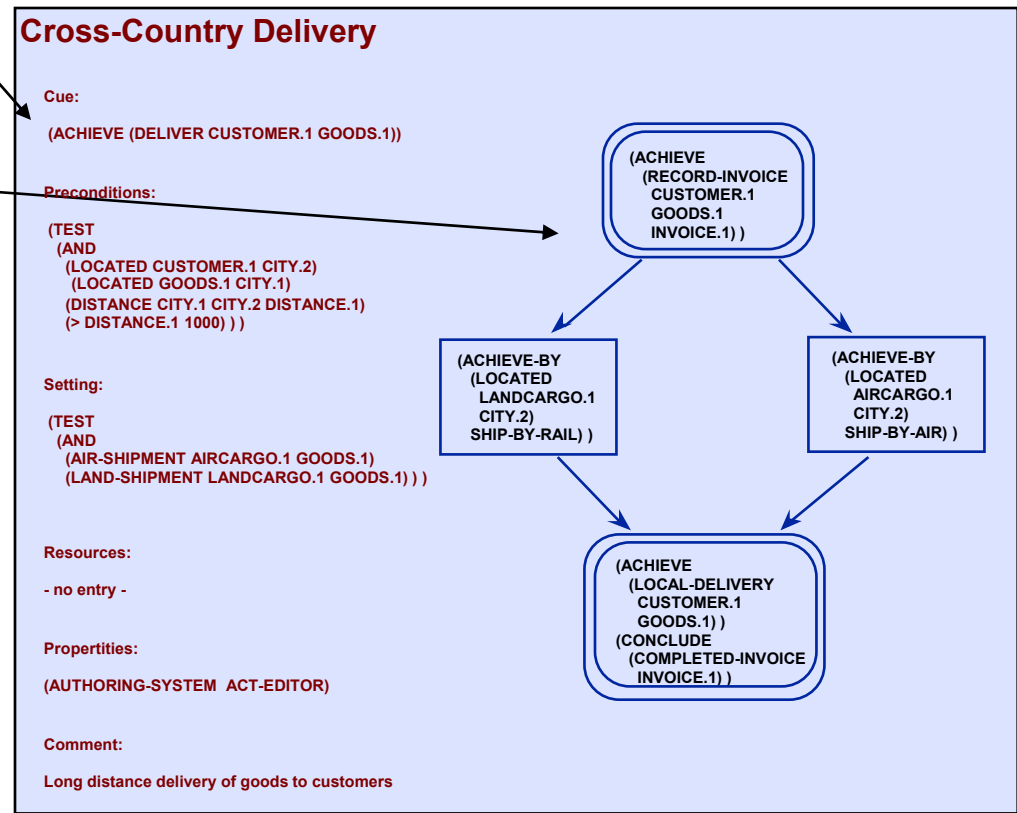
1. Perceive the world, and update the set of events.
2. For each event, generate the set of plans whose trigger condition matches the event. These are known as the *relevant* plans of an event.
3. For each event, select the subset of relevant plans whose context condition is satisfied by the agent's current beliefs. These plans are known as *active* plans.
4. From the set of active plans, select one for execution so that it is now an intention.
5. Include this new intention in the current intention structure either by creating a new intention stack or by placing it on the top of an existing stack.
6. Select an intention stack, take the topmost intention, and execute the next formula in it.

# Expressing Tasks in a Dynamic Environment

- $(! P)$  -- achieve  $P$
- $(? P)$  -- test  $P$
- $(\# P)$  -- maintain  $P$
- $(\wedge C)$  -- wait until  $C$
- $(-\> C)$  -- assert  $C$
- $(\sim\> C)$  -- retract  $C$

# Representing Procedures with Act Formalism

- **Environment conditions**
  - Purpose (goal or condition)
  - applicability criteria
- **Plot**
  - directed graph
  - partially ordered conditional & parallel actions, loops
  - Successful node execution by achievement of node's goals
  - If no body: primitive action



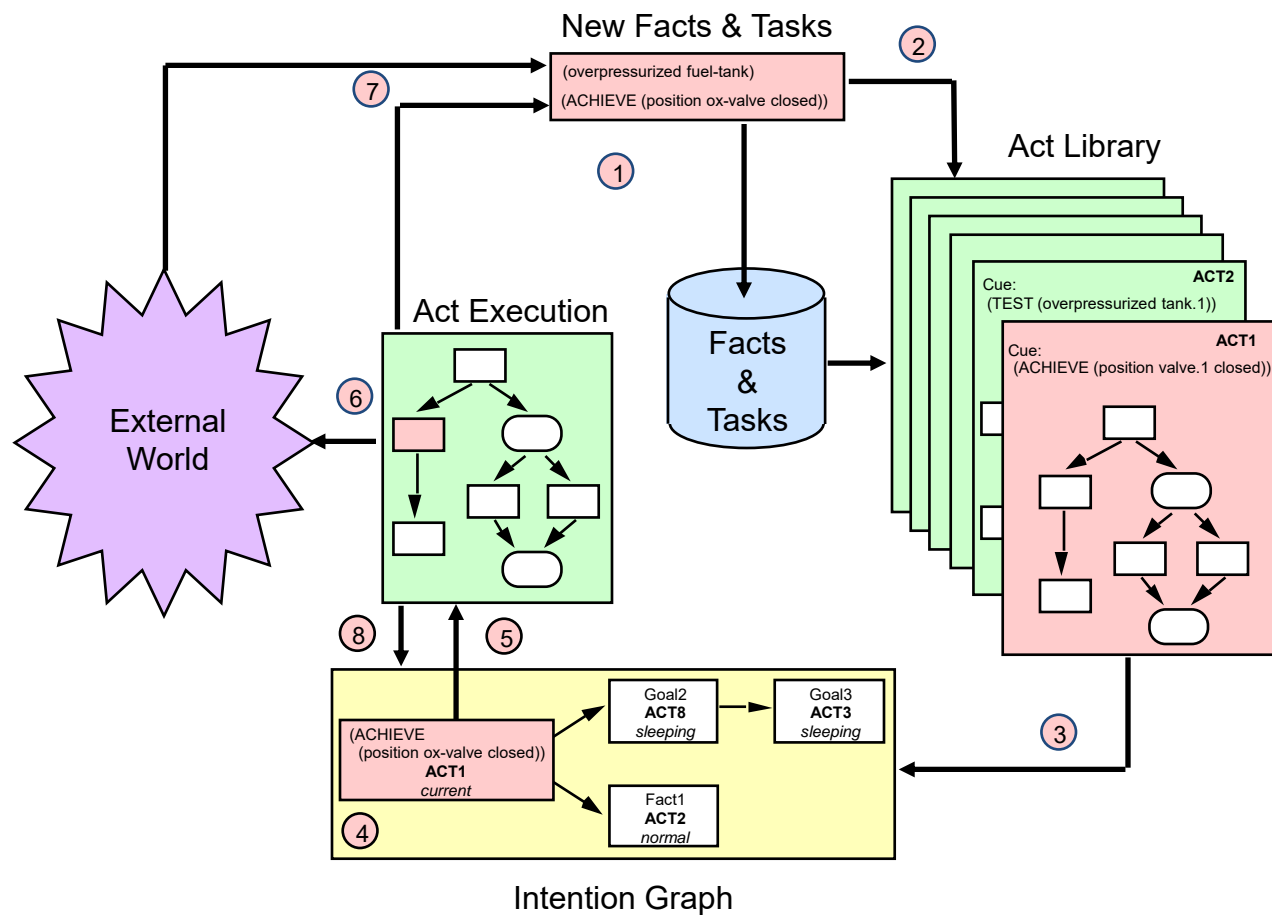
## Metapredicates

- Achieve
- Test
- Wait-Until
- Require-Until
- Achieve-By {proc}
- Conclude {effects}
- Use-Resource

# PRS Interpreter

## Execution Cycle

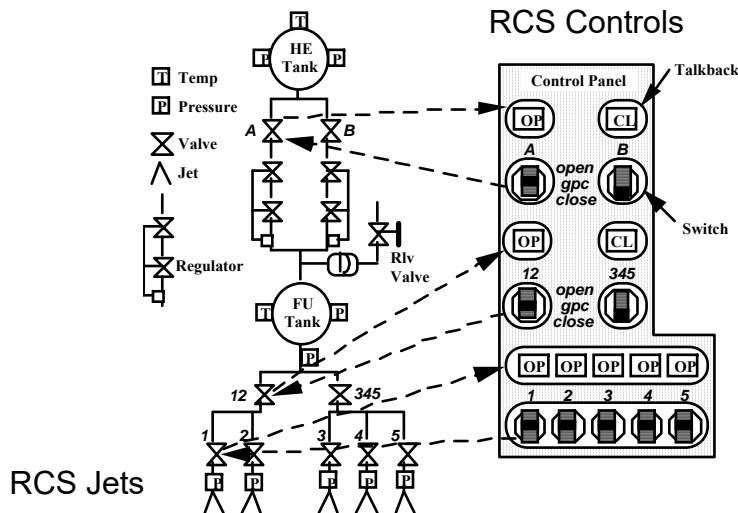
1. New information arrives that updates facts and/or tasks
2. Acts are triggered by new facts or tasks
3. A triggered Act is intended
4. An intended Act is selected
5. That intention is activated
6. An action is performed
7. New facts or tasks are posted
8. Intentions are updated



# Meta-Reasoning

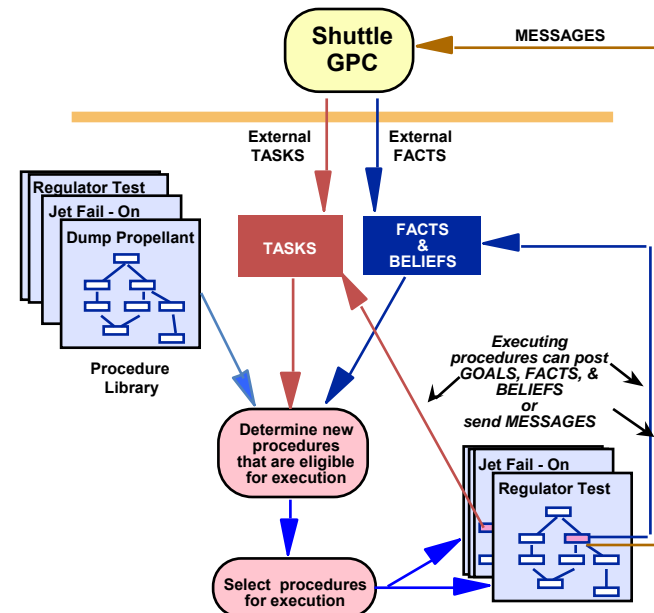
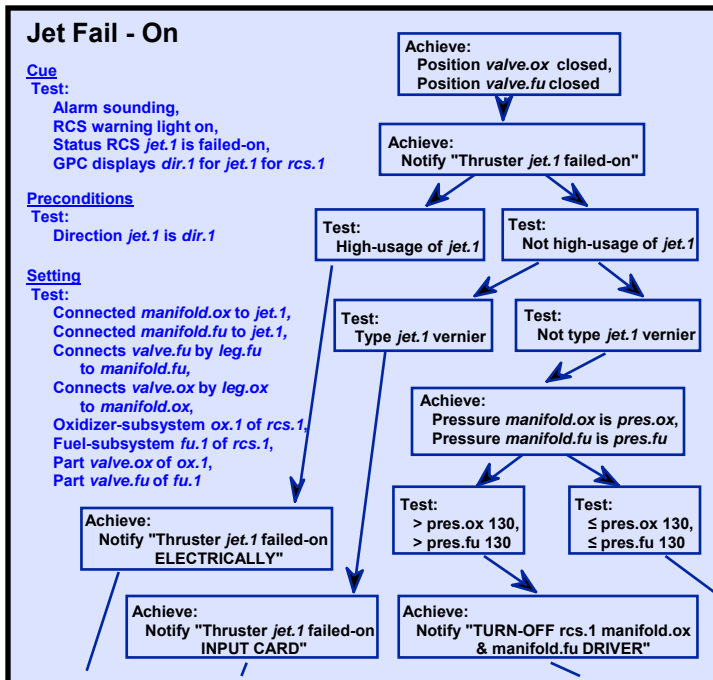
- Can include meta-level procedures
  - eg: choose among multiple applicable procedures
  - eg: evaluate how much more reasoning can be done within time constraints
  - eg: how to achieve a conjunction or disjunction of goals

# Shuttle's RCS Malfunction Handling



PRS's procedure library

- Automates specification and execution of RCS (Reaction Control System) malfunction procedures.
- Reacts to changes in RCS. Ensures safe operation while carrying out diagnosis and remediation procedures.





# PRS

## Versions of PRS:

- UM-PRS
- OpenPRS (formerly C-PRS and Propice)
- AgentSpeak
- Distributed Multi-Agent Reasoning System (dMARS)
- JAM
- JACK Intelligent Agents
- SRI Procedural Agent Realization Kit (SPARK)
- PRS-CL

# Multiple Tasks, Multiple Agents

- Multithreaded operation: multiple tasks being performed, runtime stacks where tasks are executed, suspended, and resumed
- Supports distributed planning: several PRS agents run asynchronously and communicate through message passing

# Model-based Programming

- High level programming vs. Planning
- Model-based execution and diagnosis
- RMPL [Kim et. al 2001, Williams et al. 2003]
- GOLOG [Hahnel et al. 1998, Finzi Orlandini 2005, Carbone et al. 2008]

# Summary

- Reactive Action Packages (RAPs)
  - Networks of “conditions” and “tasks”
- ESL (Execution Support Language)
- Task Control Architecture (TCA)
  - Network arranged according to “vertical capabilities”
- Claraty and PLEXIL
- Behavior Trees
- BDI
  - IRMA
  - Procedural Reasoning System (PRS)