Cognitive Architectures

What is Cognitive Architecture?

Blueprint for Intelligent Agents

It proposes (artificial) computational processes that act like cognitive systems (human)

An approach that attempts to model behavioral as well as structural properties of the modeled system.

Aim:

- to summarize the various results of cognitive psychology in a comprehensive computer model
- to model systems that accounts for the whole of cognition.

1970	• GPS (Ernst & Newell, 1969) Means-ends analysis, recursive subgoals • ACT (Anderson, 1976) Human semantic memory • CAPS (Thibadeau, Just, Carpenter) Production system for modeling reading
1975	**• Soar (Laird, & Newell, 1983) Multi-method problem solving, production systems, and problem spaces** • Theo (Mitchell et al., 1985) Frames, backward chaining, and EBL
1980	• PRS (Georgeff & Lansky, 1986) Procedural reasoning & problem solving • BB1/AIS (Hayes-Roth & Hewitt 1988) Blackboard architecture, meta-level control
1985	• Prodigy (Minton et al., 1989) Means-ends analysis, planning and EBL • MAX (Kuokka, 1991) Meta-level reasoning for planning and learning
1990	• Icarus (Langley, McKusick, & Allen,1991) Concept learning, planning, and learning • 3T (Gat, 1991) Integrated reactivity, deliberation, and planning
1995	• CIRCA (Musliner, Durfee, & Shin, 1993) Real-time performance integrated with planning • AIS (Hayes-Roth 1995) Blackboard architecture, dynamic environment
2000	• EPIC (Kieras & Meyer, 1997) Models of human perception, action, and reasoning • APEX (Freed et al., 1998) Model humans to support human computer designs

Unified Theory of Cognition

Book by Allen Newell

Newell's aim:

To define the architecture of human cognition, which is the way that humans process information. This architecture must explain how we react to stimuli, exhibit goal directed behavior,acquire rational goals, represent knowledge, and learn.

Newell's Cognitive Model

Newell introduces Soar: architecture for general cognition.

Soar is a problem solver that creates its own subgoals and learns from its own experience.

Soar operates with real-time constraints: immediate-response, item-recognition tasks, etc..

What is Soar?

Soar is a symbolic cognitive architecture:

- AI programming framework
- Cognitive architectural framework to define and exploit cognitive models
- Architecture for knowledge-based problem solving, learning, and interaction with external environments

History of Soar

Created by John Laird, Allen Newell, and Paul Rosenbloom at Carnegie Mellon University in 1983.

John Laird

Allen Newell

Paul Rosenbloom

SOAR

Historically, Soar was for **State, Operator And Result**, because problem solving in Soar is a search through a problem space in which you apply an operator to a state to get results

Over time, the community no longer regarded Soar as an acronym: this is why it is no longer written in upper case

Problem Spaces

Soar represents all tasks as collections of problem spaces

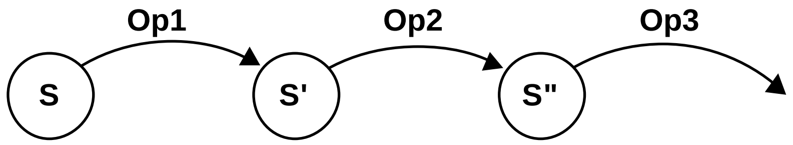
Problem spaces are made up of a set of states and operators that manipulate the states.

Soar begins work on a task by choosing a problem space, then an initial state in the space

Soar represents the goal of the task as some final state in the problem space

Problem Space Level

- Behaviour seen as occurring in a problem spaces: made up of States (S) and Operators (O or Op). (The implementation, however, has changed somewhat from Newell’s 1990 book.)
- Fluent behaviour is a cycle in which an *operator* is selected, and is applied to the current state to give a new (i.e. modified) current state



Problem Space Level

- Once the situation is set up and running, the main activity is the repeated *selection* and then *application* of one operator after another

- If something prevents that process from continuing (e.g., no operators to apply to *that* state, or no knowledge of how to choose) an *impasse occurs*

Some Definitions

- G*oal*: is a desired situation.
- S*tate*: representation of a problem solving situation.
- P*roblem space:* set of states and operators for the task.
- O*perator*: transforms the state by some action.

Problem Solving

- Soar is based upon a theory of human problem solving:
  - problem solving activity is formulated as the selection and application of operators to a state, to achieve some goal.

Problem Solving

Newell introduces the *problem space principle* as follows.

"The rational activity in which people engage to solve a problem can be described in terms of (1) a set of states of knowledge, (2) operators for changing one state into another, (3) constraints on applying operators and (4) control knowledge for deciding which operator to apply next.“

Problem spaces (e.g. STRIPS domain) are commonly composed of a set of goals, a state or set of states, and a set of valid operators which contain the constraints under which the operator can be applied.

The state consists of a set of literals that describe the knowledge of the agent and the present model of the world.

Structure of Soar

Soar can be layered into 3 levels :

1. Memory Level
2. Decision Level
3. Goal Level

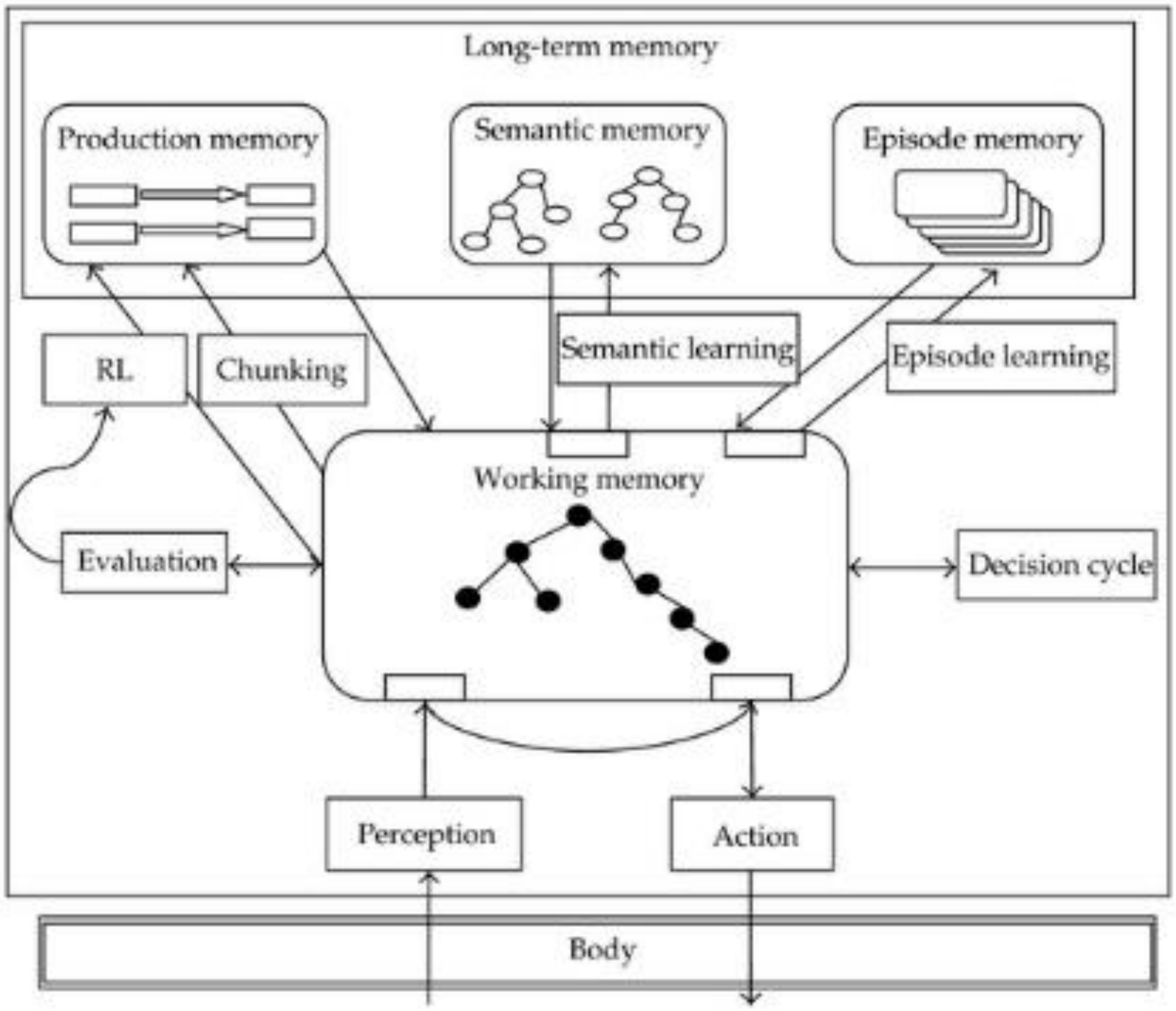
Memory Level

A general intelligence requires a memory with a large capacity for the storage of knowledge.

A variety of types of knowledge must be stored, including :

- Declarative knowledge
- Procedural knowledge
- Episodic knowledge

Soar Architecture



Long-term Production Memory

All of Soar's long-term knowledge is stored in a single production memory.

Each production is a condition-action structure that performs its actions when its conditions are met.

Memory access consists of the execution of these productions.

During the execution of a production, variables in its actions are instantiated with value.

Working Memory

The result of memory access is the retrieval of information into a global working memory.

It is the temporary memory that contains all of Soar's short-term processing context. It has 3 components :

- The context stack specifies the hierarchy of active goals, problem spaces, states and operators
- Objects, such as goals and states (and their subobjects)
- Preferences that encode the procedural search-control knowledge

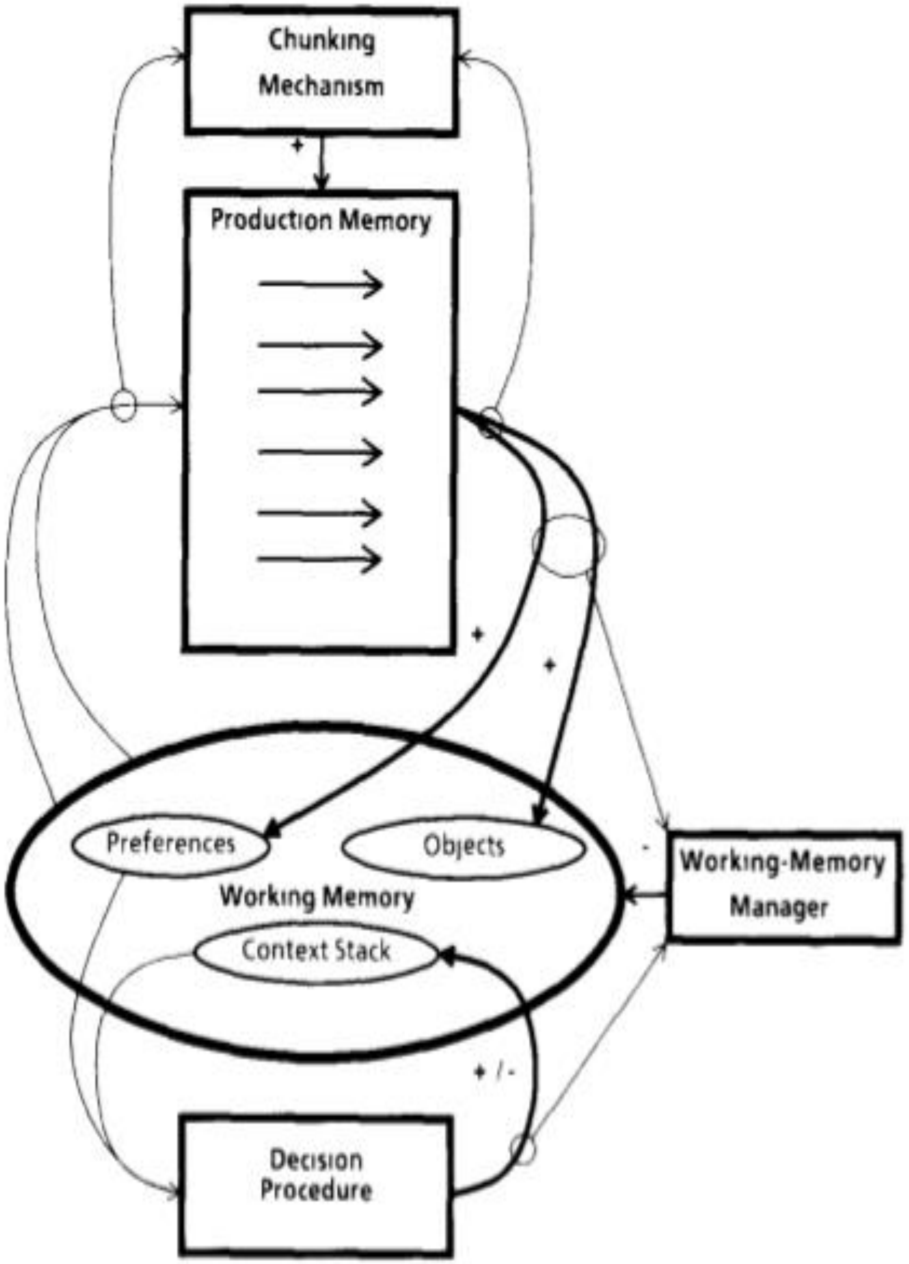


FIG 5 Architectural structure of SOAR

Preferences

There is one special type of working memory structure - “the preference”

Preferences encode control knowledge about the acceptability and desirability of actions

Acceptability preferences determine which actions should be considered as candidates

Desirability preferences define a partial ordering on the candidate actions.

Decision Level

- The decision level is based on the memory level plus an architecturally provided, fixed, decision procedure.
- The decision level proceeds in a two phase elaborate-decide cycle.
- During elaboration, the memory is accessed repeatedly, in parallel, until quiescence is reached (no more productions can execute).
- This results in the retrieval into working memory of all of the accessible knowledge that is relevant to the current decision.
- After quiescence has occurred, the decision procedure selects one of the retrieved actions based on the preferences that were retrieved into working memory.

Decision Level

Two phase decision cycle: elaboration and decision. The two phases are repeated until the goal of the current task is reached.

- Elaboration phase:
  - all productions which match the current working memory fire. All productions fire in parallel.
  - The elaboration phase runs to Quiescence (until no more productions fire).

- Decision phase:
  - examines any preferences put into preference memory (either in this phase, or previous ones), and chooses the next problem space, state, operator or goal to place in the context stack.

Decision Level

If there is not enough information (or contradictory) for the decision phase to choose the next value, then an impasse results.

- There are four types of impasses:
  - When two are more elements have equal preference, then there is a "tie impasse".
  - When no preferences are in working memory, this causes a "no-change impasse"
  - When the only preferences in working memory are rejected by other preferences, then there is a "reject impasse".
  - A "conflict impasse" results when preferences claim that two or more elements are each better choices then the others.

- When Soar reaches an impasse, it chooses a new problem space in an attempt to resolve the impasse.

Goal Level

- A general intelligence must be able to set and work towards goals.This level is based on the decision level.

- Goals are set whenever a decision cannot be made; that is, when the decision procedure reaches an impasse.

- Impasses occur when there are no alternatives that can be selected (no-change and rejection impasses) or when there are multiple alternatives that can be selected, but insufficient discriminating preferences exist to allow a choice to be made among them (tie and conflict impasses).

Impasse Resolution

- Whenever an impasse occurs, the architecture generates the goal of resolving the impasse which becomes the subgoal.

- Along with this goal, a new performance context is created.

- The creation of a new context allows decisions to continue to be made in the service of achieving the goal of “resolving the impasse”.

- A stack of impasses is possible.

- The original goal is resumed after all the impasse stack is cleared.

Learning

- Chunking: new chunks to overcome empasses

- Reinforcement Learning: better operator selection

- Episodic and Semantic Learing: working memory re-organization

Learning through Chunking

- In addition to all above levels, a general intelligence requires the ability to learn.

- All learning occurs by the acquisition of chunks--productions that summarize the problem solving that occurs in subgoals, a mechanism called “Chunking”

- The actions of a chunk represent the knowledge generated during the subgoal; that is, the results of the subgoal.

Soar 9

- Unifying Cognitive Functions and Emotional Appraisal

- The functional and computational role of emotion is open to debate.

- Appraisal theory is the idea that emotions are extracted from our evaluations (appraisals) of events that cause specific reactions in different people.

- The main controversy surrounding these theories argues that emotions cannot happen without physiological arousal.

Appraisal's Detector

This theory proposes that an agent continually evaluates a situation and that evaluation leads to emotion.

The evaluation is hypothesized to take place along multiple dimensions, such as

- goal relevance
- goal conduciveness
    - causality and control

These dimensions are exactly what an intelligent agent needs to compute as it pursues its goals while interacting with an environment.

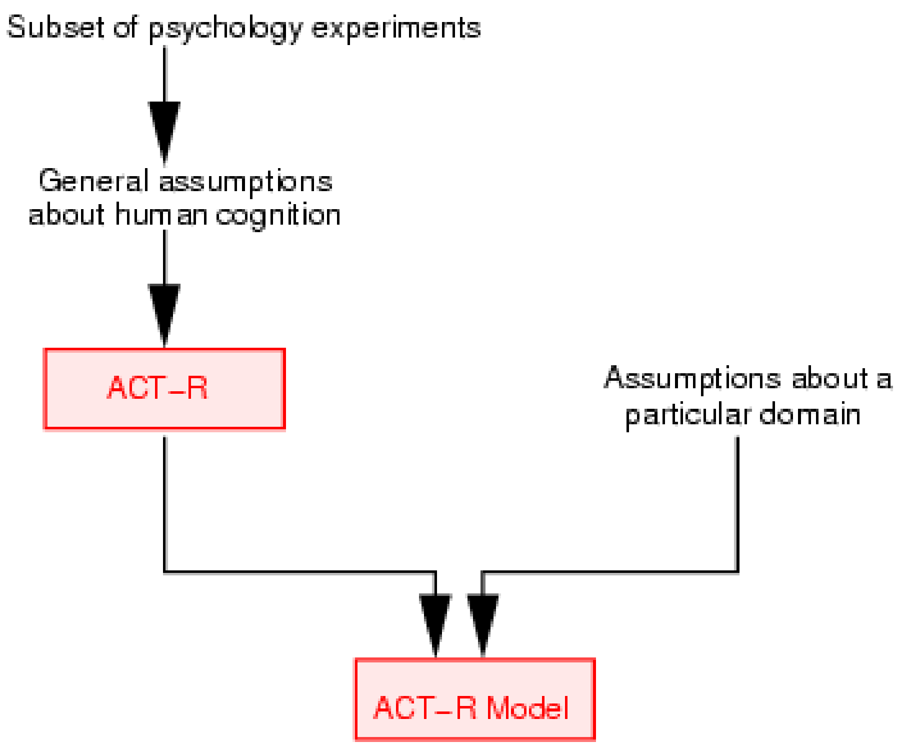
ACT-R

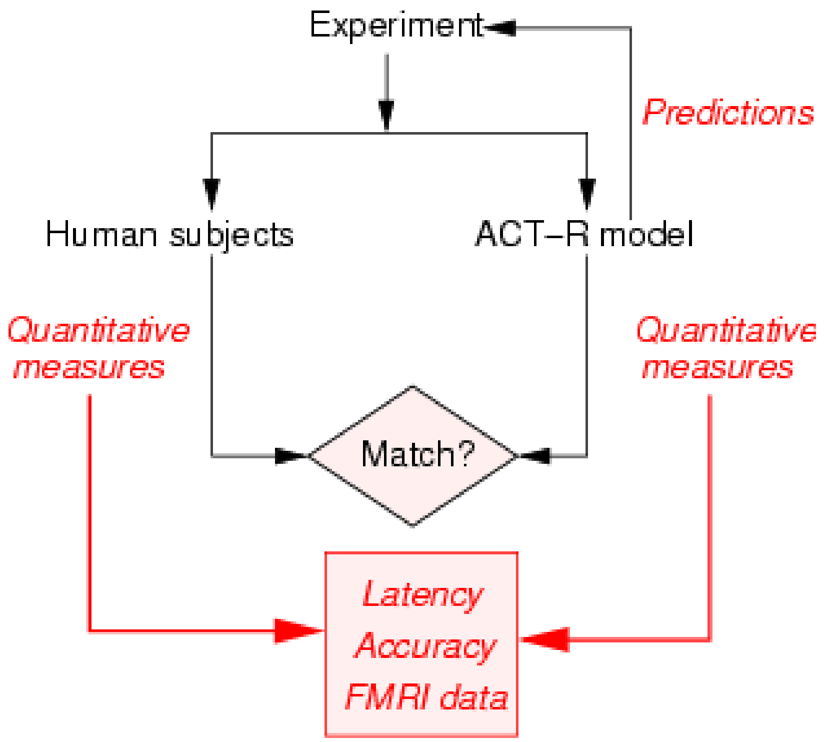
- ACT-R is a cognitive architecture, a theory about how human cognition works.
  - Looks like a (procedural) programming language.
  - Constructs based on assumptions about human cognitions

ACT-R

- ACT-R is a framework
  - Researchers can create models that are written in ACT-R including
    - ACT-R's assumptions about cognition.
    - The researcher's assumptions about the task.
  - The assumptions are tested against data.
    - Reaction time
    - Accuracy
    - Neurological data (fMRI)

ACT-R


Subset of psychology experiments
General assumptions about human cognition
ACT–R
Assumptions about a particular domain
ACT–R Model


Experiment
Predictions
Human subjects
ACT–R model
Quantitative measures
Quantitative measures
Match?
Latency
Accuracy
FMRI data

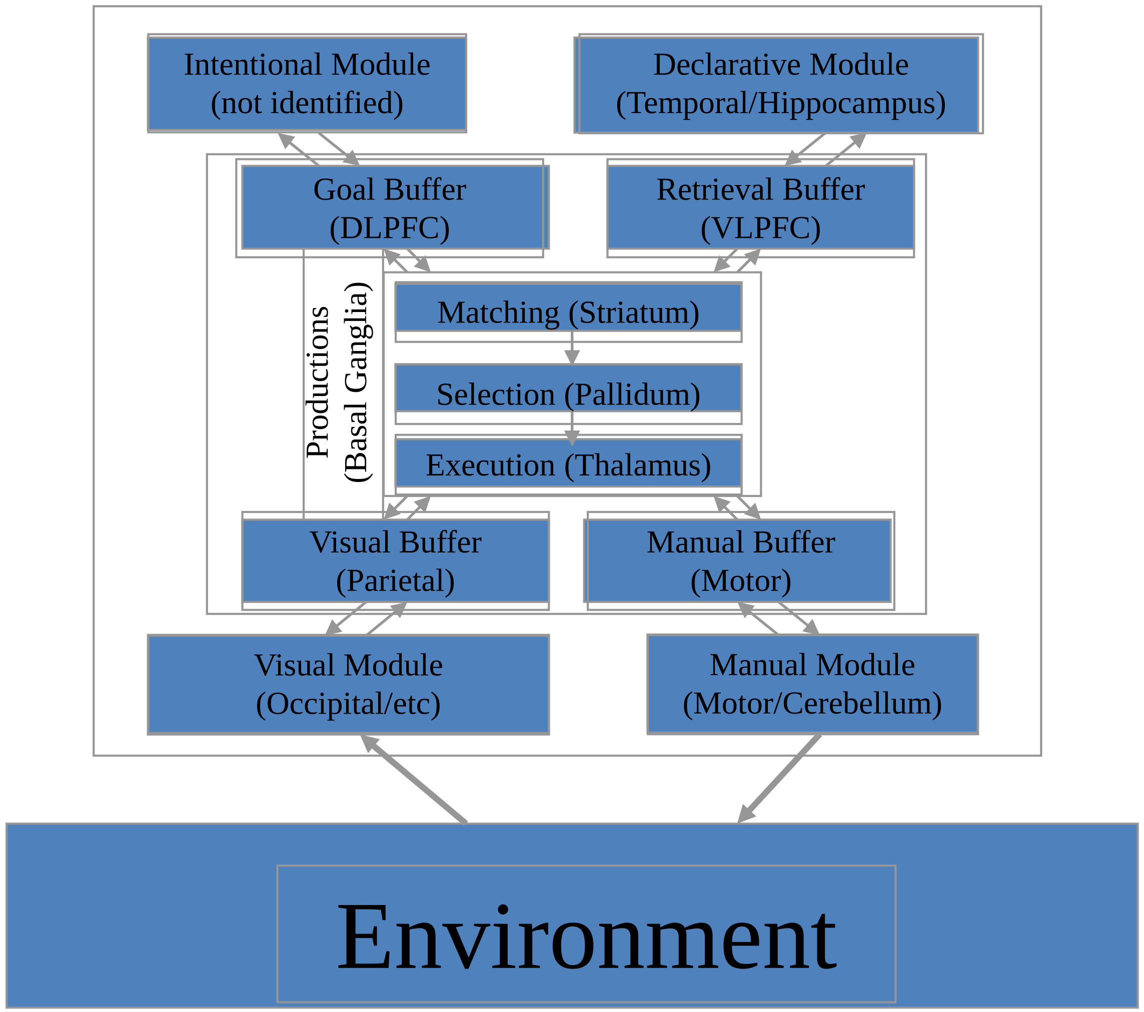
ACT-R

- ACT-R is an integrated cognitive architecture.
  - Brings together not just different aspects of cognition, but of
    - Cognition
    - Perception
    - Action
  - Runs in real time.
  - Learns.
  - Robust behavior in the face of error, the unexpected, and the unknown.

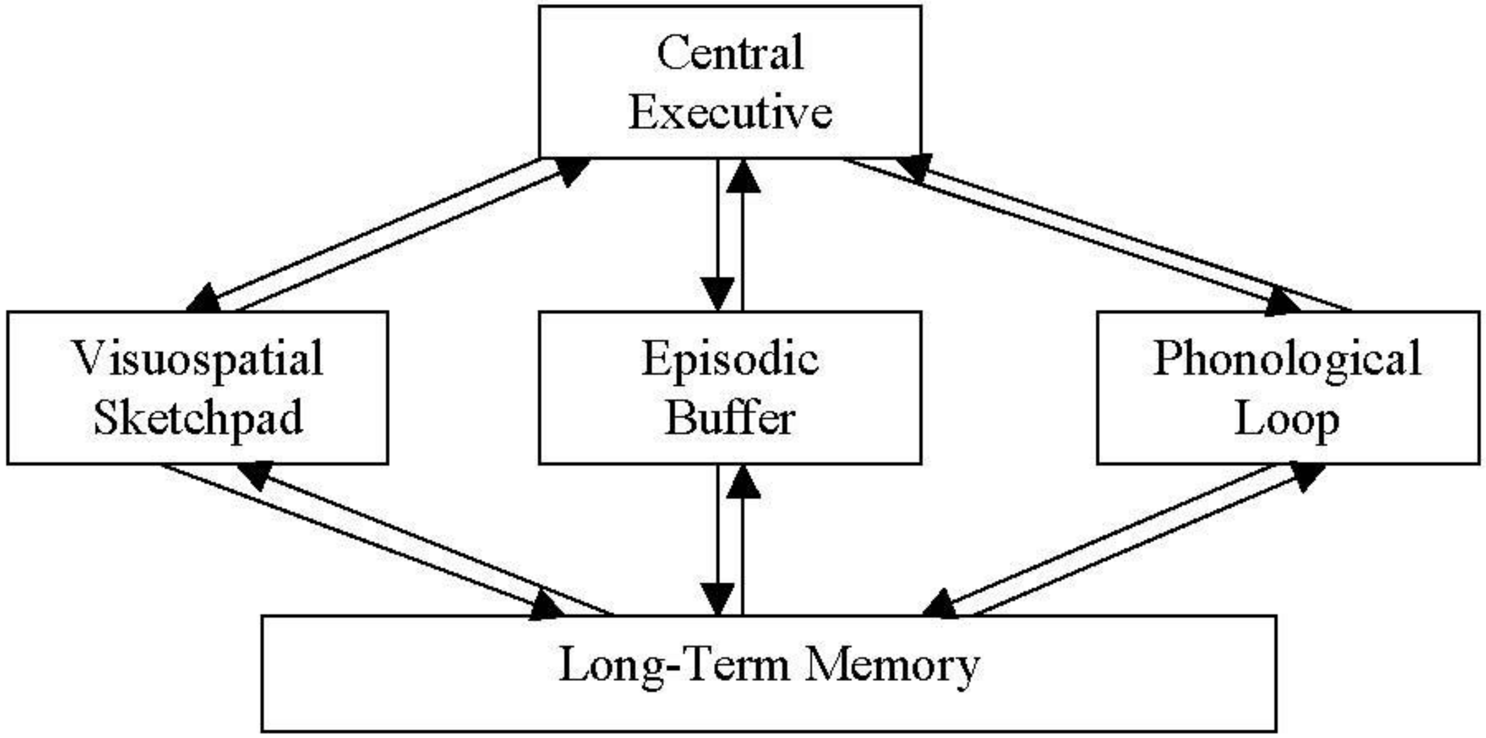
Overview of ACT-R

- ACT-R is made up of
  - Modules.
  - Buffers.
  - A subsymbolic level.

ACT-R: Architecture

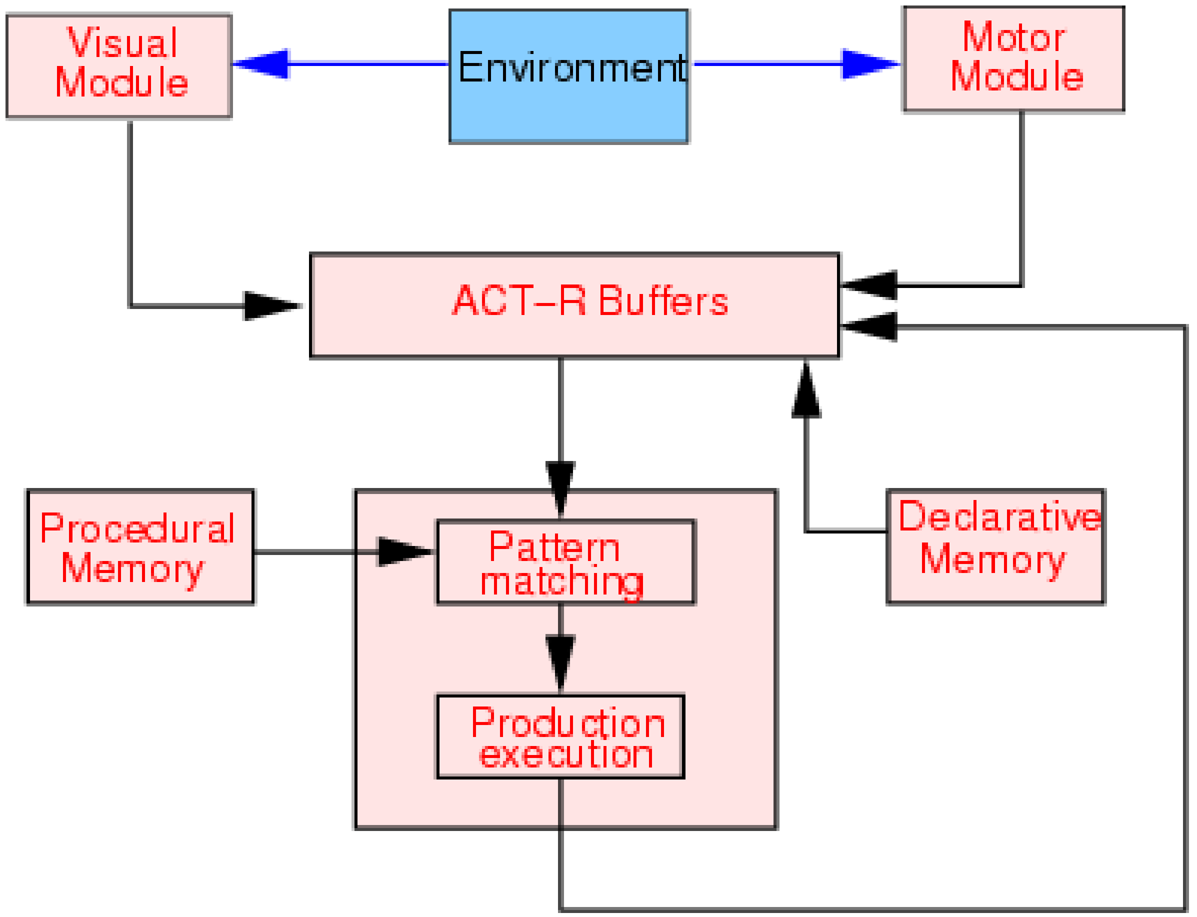


Baddeley Model



ACT-R: Cycle

ACT-R accesses its modules (except for the procedural-memory module) through buffers. For each module, a dedicated buffer serves as the interface with that module. The contents of the buffers at a given moment in time represents the state of ACT-R at that moment.



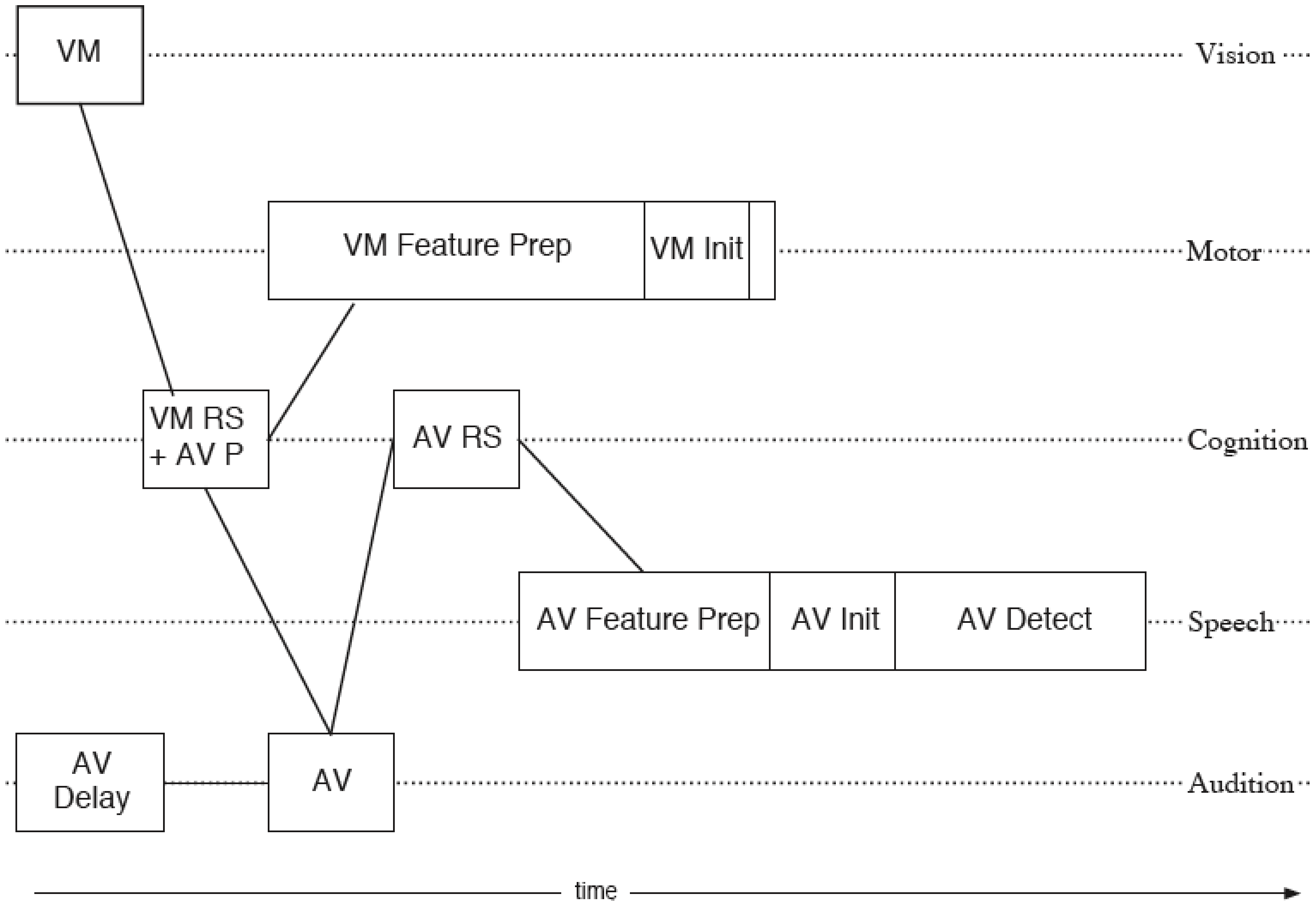
Perceptual-Motor Modules

- Takes care of the interface with the “real” world.
    - Visual module
    - Auditory module
    - Motor module
    - etc

Perceptual-Motor Modules

- 3 tones: low, med, high
  - 445ms
- 3 positions: left, middle, right
  - 279ms
- Tones and positions
  - 456ms
  - 283ms

Perceptual-Motor Modules



Declarative Module

- Declarative memory:
  - Facts
    - Washington, D.C. is the capital of the U.S.
    - 2+3=5.
  - Knowledge a person might be expected to have to solve a problem.
  - Called chunks

Declarative Module

(CHUNK-TYPE NAME SLOT1 SLOT2 SLOTN)

(b

isa	count-order
first	1
second	2

)

Procedural Module

- Procedural memory: Knowledge about how to do something.
  - How to type the letter “Q”.
  - How to drive.
  - How to perform addition.

Procedural Module

- Made of condition-action data structures called production rules.
- Each production rule takes 50ms to fire.
- Serial bottleneck in this parallel system.

Procedural Module

	(p name
condition part	Specification of Buffer Tests
delimiter	==>
action part	Specification of Buffer Transformations
	)

Procedural Module

```
( p    example-counting
       =goal>
          isa count
          state counting
          number =num1
       =retrieval>
          isa count-order
          first =num1
          second =num2
  ==>
       =goal>
          number =num2
       +retrieval>
          isa count-order
          first =num2
 )
```

IF the goal is
    to count
    the current state is counting
    there is a number called =num1
    and a chunk has been retrieved
    of type count-order
    where the first number is =num1
    and it is followed by =num2
THEN
    change the goal
    to continue counting from =num2
    and request a retrieval
    of a count-order fact
    for the number that follows =num2

Buffers

- The procedural module accesses the other modules through buffers.
- For each module (visual, declarative, etc), a dedicated buffer serves as the interface with that module.
- The contents of the buffers at any given time represent the state of ACT-R at that time.

Buffers

1. Goal Buffer (=goal, +goal)
   -represents where one is in the task
   -preserves information across production cycles

2. Retrieval Buffer (=retrieval, +retrieval)
   -holds information retrieval from declarative memory
   -seat of activation computations

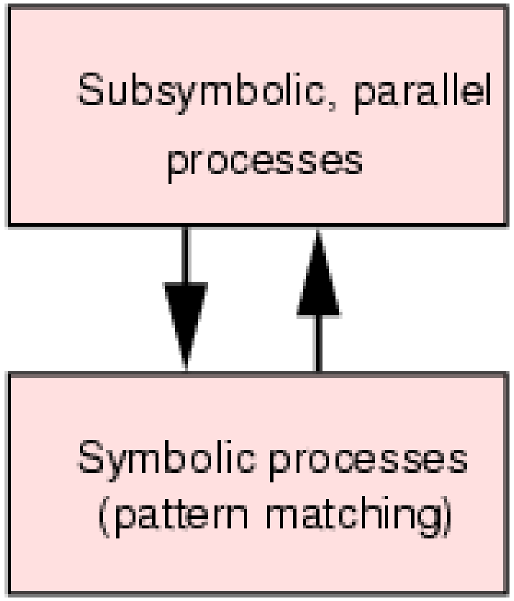
3. Visual Buffers
   -location (=visual-location, +visual-location)
   -visual objects (=visual, +visual)
   -attention switch corresponds to buffer transformation

4. Auditory Buffers (=aural, +aural)
   -analogous to visual

5. Manual Buffers (=manual, +manual)
   -elaborate theory of manual movement include feature preparation, Fitts law, and device properties

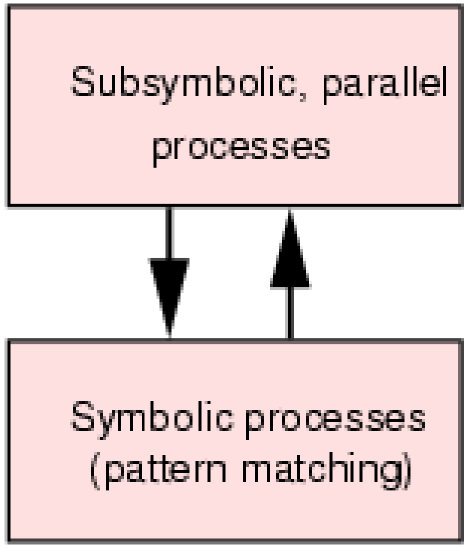
Subsymbolic Level

- The production system is symbolic.
- The subsymbolic structure is a set of parallel processes that can be summarized by a number of mathematical equations.
- The subsymbolic equations control many of the symbolic processes.


Subsymbolic, parallel processes
Symbolic processes (pattern matching)

Subsymbolic Level

- The subsymbolic equations control many of the symbolic processes.
- If several productions match the state of the buffers, a subsymbolic utility equation estimates the relative cost and benefit associated with each production and decides to select for the production with the highest utility.
- Similarly, whether (or how fast) a fact can be retrieved from declarative memory depends on subsymbolic retrieval equations, which take into account the context and the history of usage of that fact.
- Subsymbolic mechanisms are also responsible for most learning processes in ACT-R.


Subsymbolic, parallel processes
Symbolic processes (pattern matching)

Subsymbolic Level

- If several productions match the state of the buffers, a subsymbolic utility equation estimates the relative cost and benefit associated with each production and selects the production with the highest utility.

Production Utility

Expected Gain = $E = PG - C$

P expected probability of success
G value of goal
C expected cost

Probability of choosing i = $$\frac{e^{E_i/t}}{\sum_j e^{E_j/t}}$$

t noise in evaluation (temperature in the Bolztman equation)

$$P = \frac{\text{Successes}}{\text{Successes} + \text{Failures}}$$

α prior successes
m experienced successes
β prior failures
n experienced failures

Successes = $\alpha + m$
Failures = $\beta + n$

Subsymbolic Level

- Whether and how fast a chunk can be retrieved from declarative memory depends on the subsymbolic retrieval equations, which take into account the context and the history of usage of that fact.

Chunk Activation

- The activation of a chunk is a sum of base-level activation, reflecting its general usefulness in the past, and an associative activation, reflecting it’s relevance in the current context.

Chunk Activation

Attentional weighting of
Element j of Chunk i

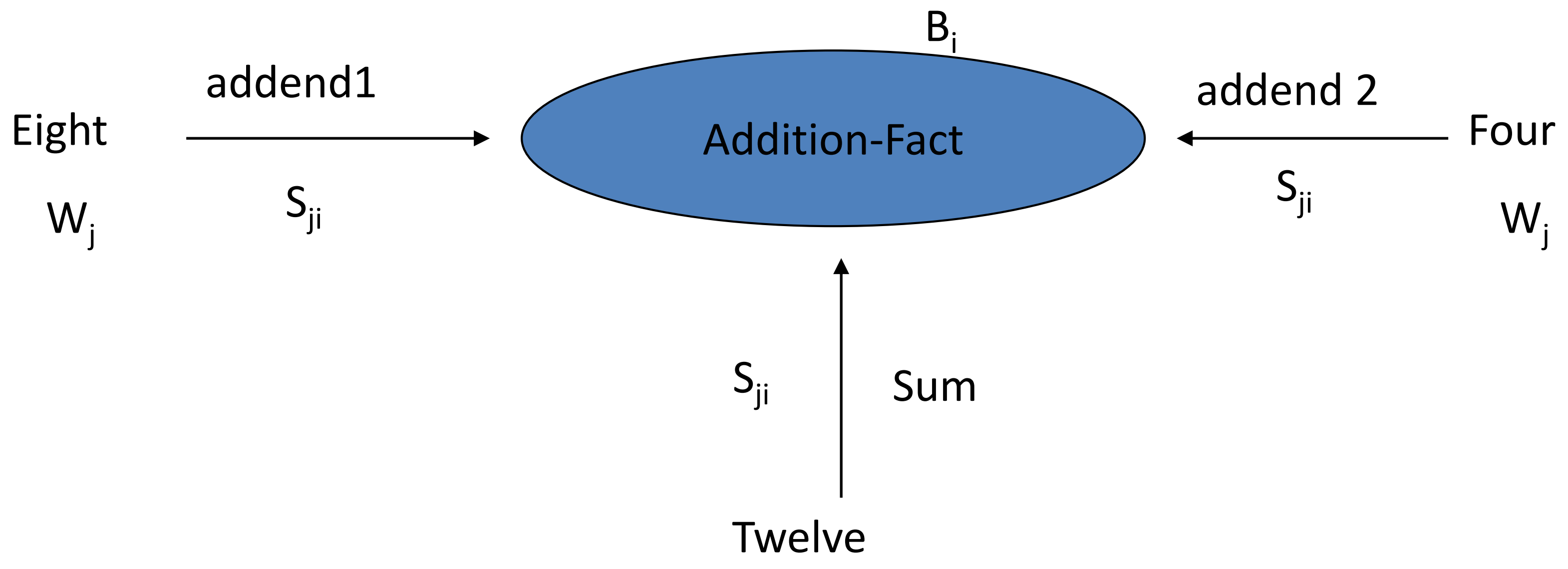
Activation of Chunk i

$$A_i = B_i + \sum_j W_j S_{ji}$$

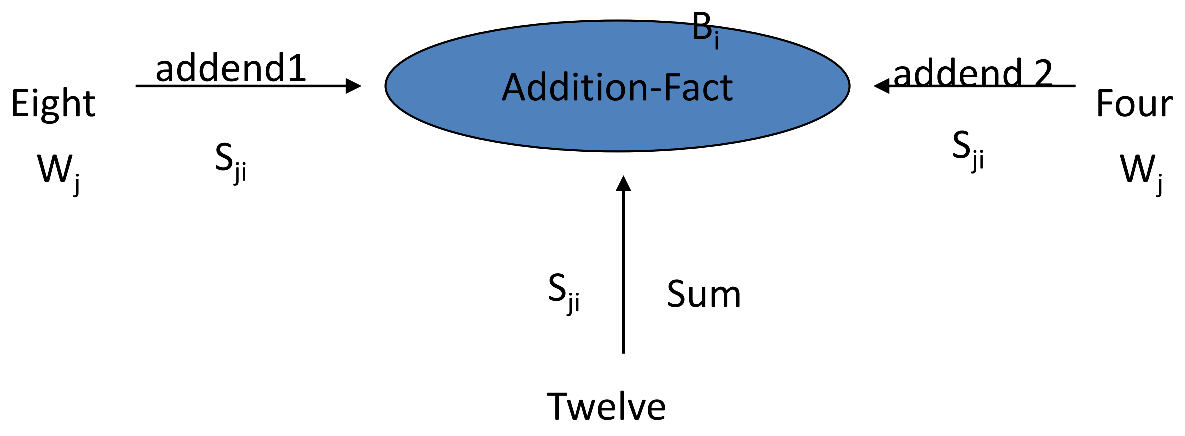
Strength of association
of Element j to Chunk i

Base-level activation
(Higher if used recently)

Chunk Activation



Chunk Activation



W_j decreases with the number of elements associated with Chunk i.

S_{ji} decreases with the number of chunks associated with the element.

Probability of Retrieval

- The probability of retrieving a chunk is given by

$$P_i = 1 / (1 + \exp(-(A_i - \tau)/s))$$

Retrieval Time

- The time to retrieve a chunk is given by

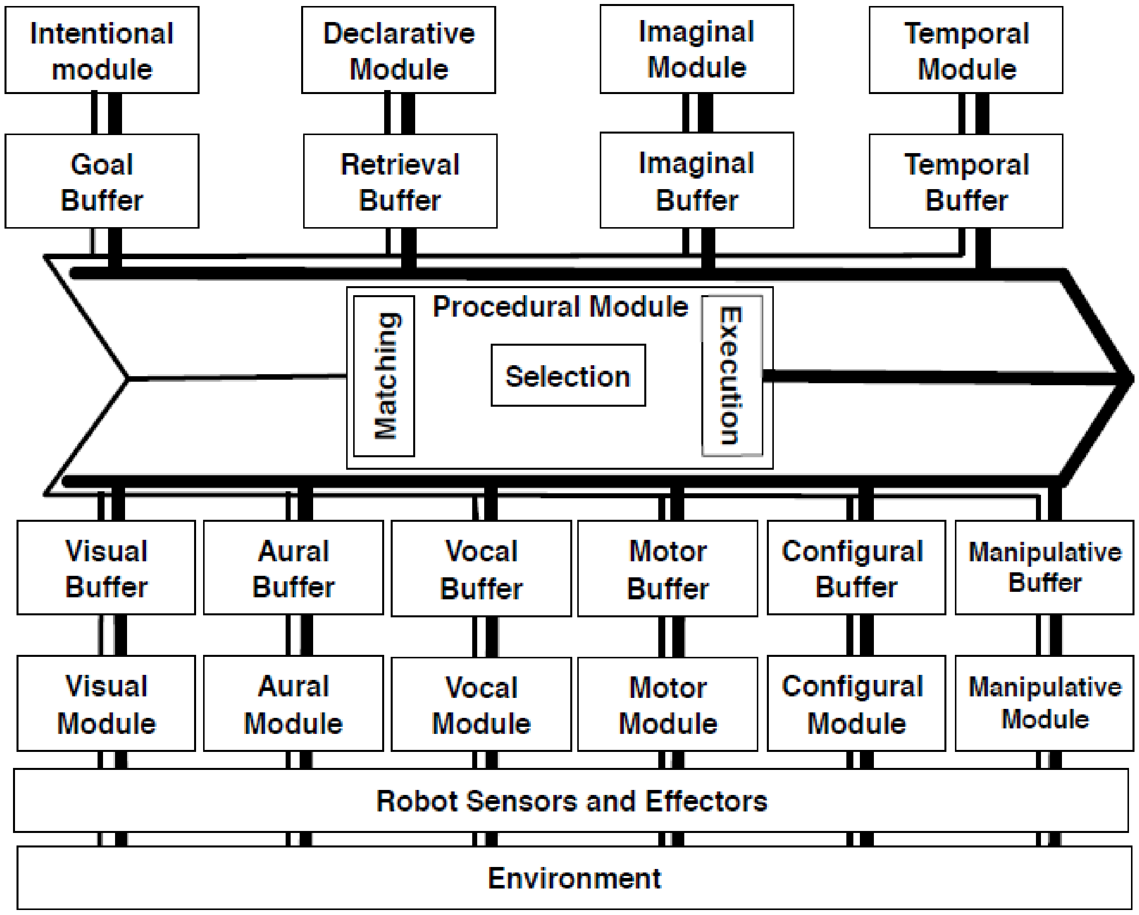
$$T_i = F \exp(-A_i)$$

Subsymbolic Level

- The equations that make up the subsymbolic level are not static and change with experience.
- The subsymbolic learning allows the system to adapt to the statistical structure of the environment.

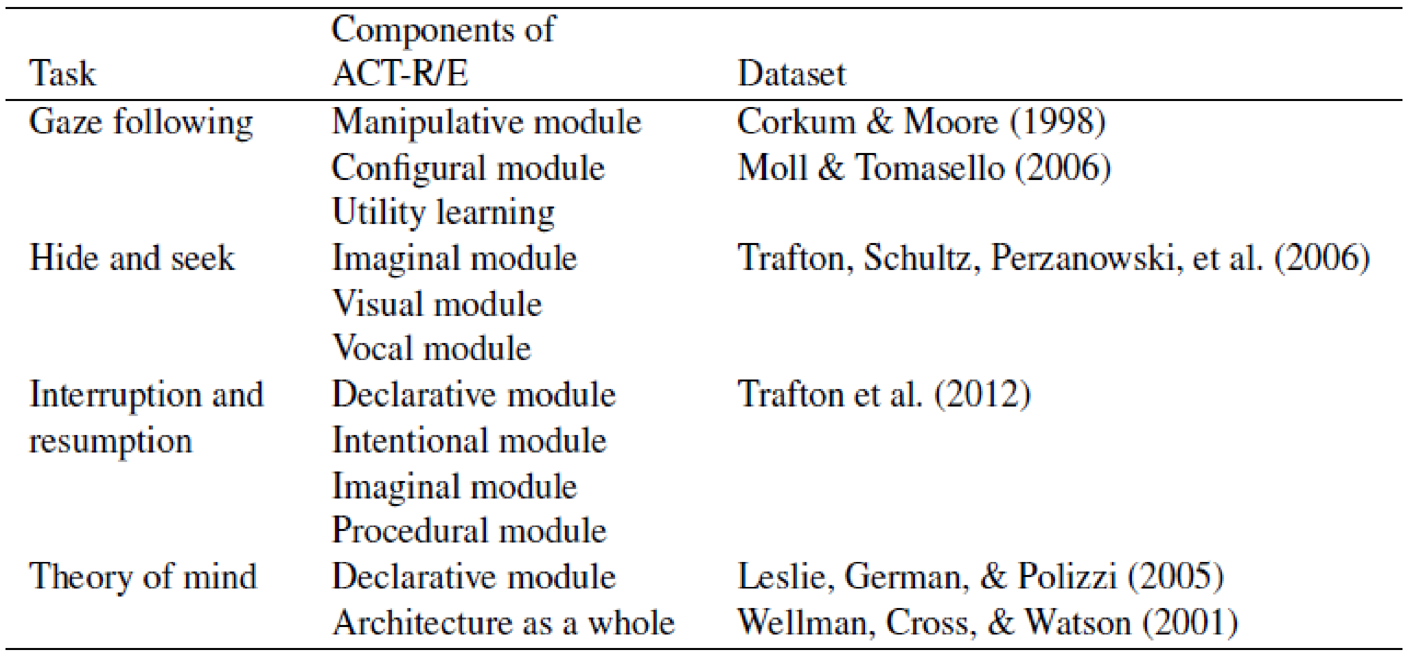
ACT-R/E

- Embodied: spatial reasoning



ACT-R/E

- HRI tasks

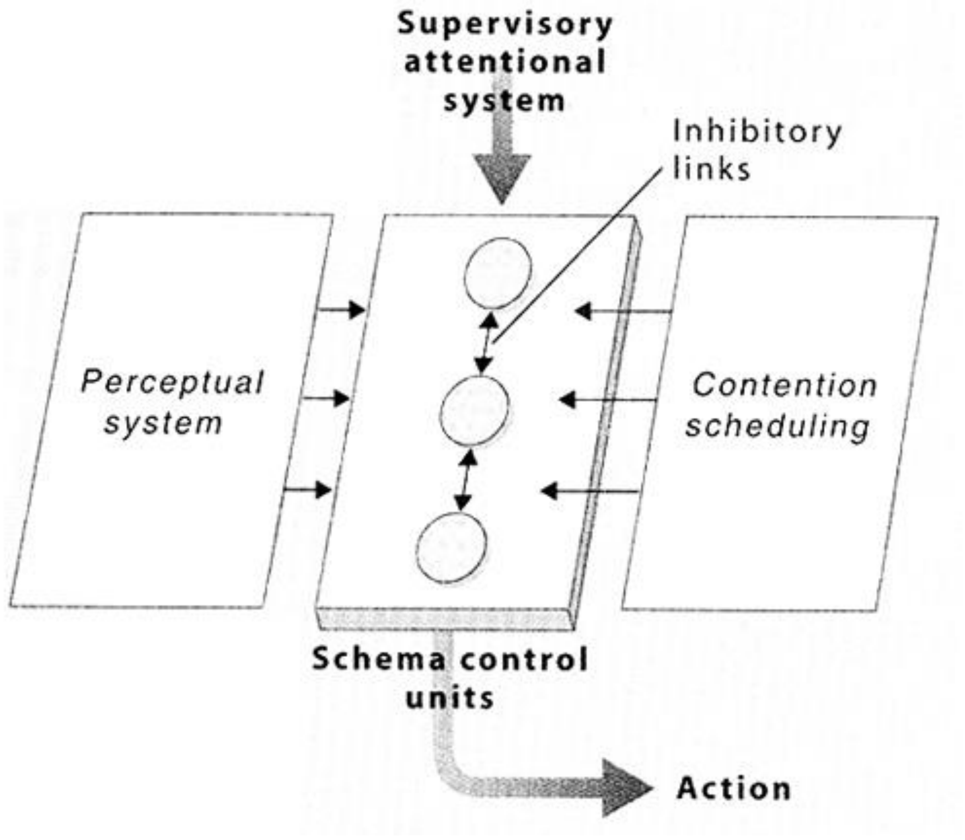


Task	Components of ACT-R/E	Dataset
Gaze following	Manipulative module	Corkum & Moore (1998)
	Configural module	Moll & Tomasello (2006)
	Utility learning	
Hide and seek	Imaginal module	Trafton, Schultz, Perzanowski, et al. (2006)
	Visual module	
	Vocal module	
Interruption and resumption	Declarative module	Trafton et al. (2012)
	Intentional module	
	Imaginal module	
	Procedural module	
Theory of mind	Declarative module	Leslie, German, & Polizzi (2005)
	Architecture as a whole	Wellman, Cross, & Watson (2001)

(1) test and evaluate each component separately, to validate it against human subject data;
(2) test different sets of the components as they interact;
(3) show how our models increase the ability, breadth, and parsimony of cognitive models.

Supervisory Attentional System

- Cognitive Control Model



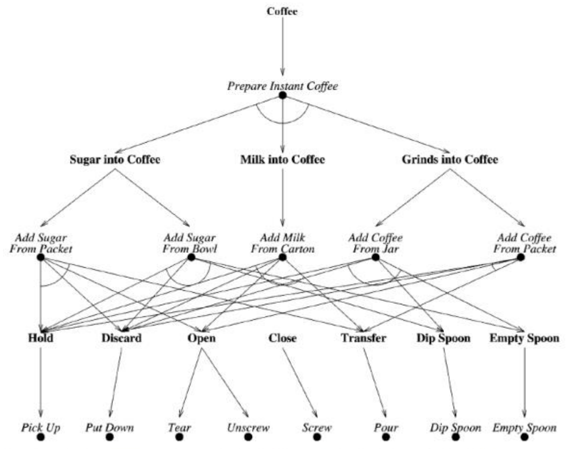


Fig. 2. *Schema/goal organisation in the coffee preparation domain. Schemas are indicated by italic type and goals by bold type.*

Architetture cognitive e Robotica

Le architetture cognitive vengono utilizzate prevalentemente in contesti di interazione uomo-macchina.

Nel contesto della robotica cognitiva, però, si riscontrano ulteriori problemi:

- La gestione e l'elaborazione di grandi quantità di informazioni spesso derivanti da fonti eterogenee.
- La necessità di eseguire diverse operazioni (sia motorie che deliberative) contemporaneamente.
- L'obbligo di operare in tempo reale ed in maniera human-aware

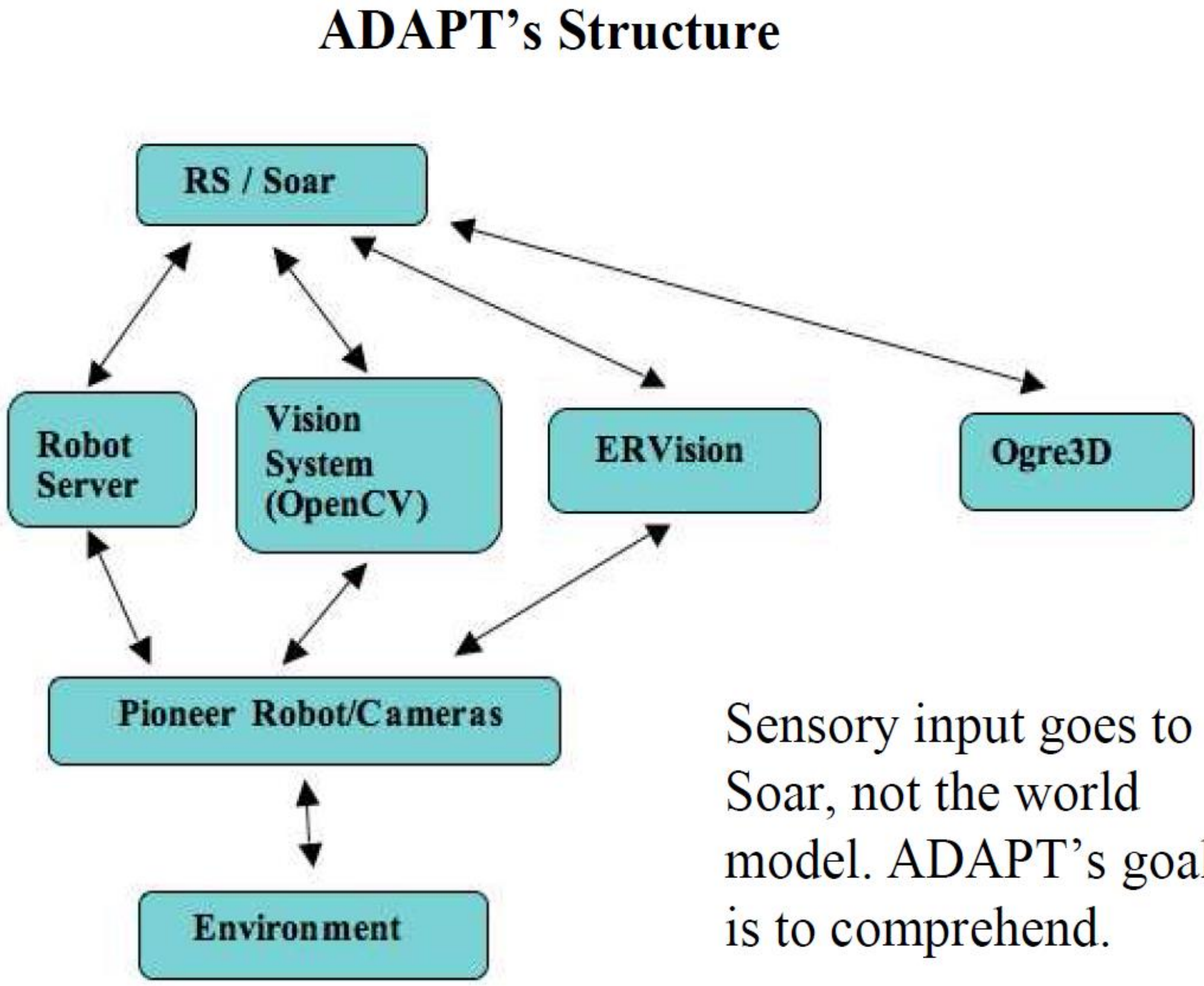
ADAPT

ADAPT è una architettura cognitiva sviluppata appositamente per la robotica e strutturata a partire dai sistemi SOAR ed ACT-R.

Il sistema ha lo scopo di implementare nei robot comportamenti sofisticati in diversi campi (visione, elaborazione del linguaggio, problem solving, learning). Esso si basa su due principi:

- La percezione è un processo attivo sensibile sia al goal che al contesto, quindi, vengono percepite più attentamente le porzioni di input inerenti allo scopo.
- Il robot deve poter ragionare in parallelo ed in tempo reale tra azioni concorrenti.

ADAPT



Sensory input goes to Soar, not the world model. ADAPT's goal is to comprehend.

ADAPT vs. Soar

SOAR possiede un singolo buffer per ogni goal, ciò consente, per ogni goal, la selezione di un solo operatore alla volta.

ACT-R consente il *firing* di una sola regola di produzione alla volta basandosi sulla utilità della stessa nel contesto attuale.

In entrambi i sistemi il parallelismo è difficilmente implementabile (se non esplicitamente vietato). Con ADAPT si cerca principalmente di risolvere tale limitazione.

ADAPT

ADAPT, analogamente a SOAR, confronta continuamente le regole con il contenuto della memoria di lavoro, aggiornandola con i risultati ottenuti, inoltre il sistema possiede una memoria a lungo termine come in ACT-R dove vengono mantenuti gli *schemi*.

Uno **schema** è l'equivalente dell'operatore di SOAR o del *chunk* di ACT-R.

- è basato sullo *schema theory* e possiede quindi schema percettivo e motorio.
- combina conoscenza procedurale e dichiarativa, consentendo di utilizzare tali informazioni per risolvere i goal o per ragionare sui comportamenti stessi.

ADAPT

Ad ogni step il sistema può effettuare una delle seguenti operazioni:

- scomporre uno schema in sottoschemi
- instanziare variabili in uno schema (unificandole eventualmente con altri schemi)
- eseguire uno schema
- sospendere l'esecuzione di uno schema
- terminare e rimuovere uno schema

La scelta delle azioni da compiere puo essere effettuata tramite ricerca nello spazio del problema (SOAR-like) oppure mediante metodi bayesiani (ACT-R-like).

ADAPT

The RS (Robot Schemas) language is the basis of the robotics capabilities of ADAPT. RS is precise and mature.

RS is a CSP-type programming language for robotics, that controls a hierarchy of concurrently executing schemas.

$$\mathrm{Joint}_i(s)() = [\mathrm{Jpos}_i()(x),\ \mathrm{Jset}_i(s, x)(u),\ \mathrm{Jmot}_i(u)()\ ]^{c0}$$

c0: $(\mathrm{Jpos}_i, x)\ (\mathrm{Jset}, x)$ $(\mathrm{Jset}, u)\ (\mathrm{Jmot}_i, u)$

$\mathrm{Jpos}_i()(x)$ continuously reports the position of joint i on port x
$\mathrm{Jmot}_i(u)()$ accepts a signal on port u and applies it to the actuator of joint i
$\mathrm{Jset}_i(s, x)(u)$ accepts a setpoint on port s and iteratively inputs a joint position on port x and outputs a motor signal on port u to drive the joint position to the setpoint

ADAPT

$P = (Q, L, X\ \delta, \beta, \tau)$ where

Q is the set of states

L is the set of ports

$X = (X_i \mid i \in L)$ is the event alphabet for each port

$XL = \{ (i, X_i) \mid i \in L \}$ i.e., a disjoint union of L and X

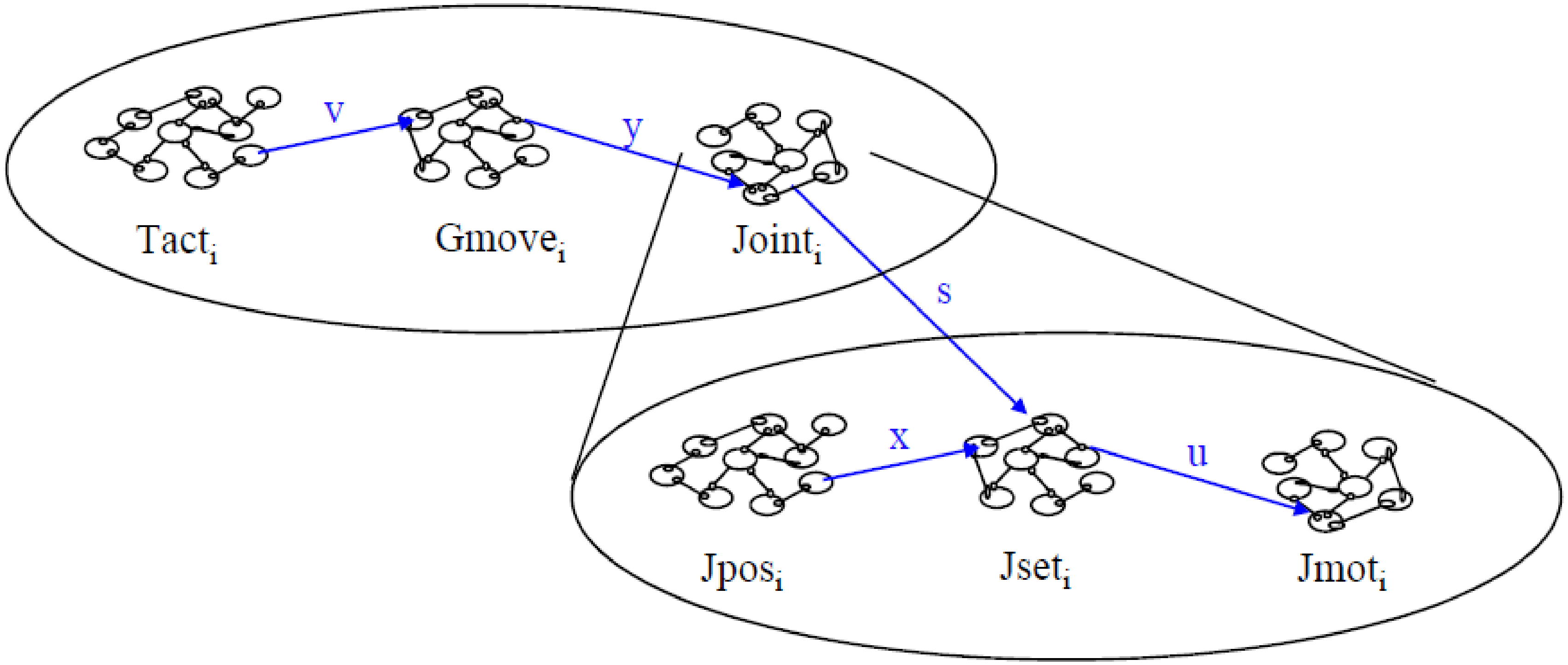
$\delta : Q \times XL \rightarrow 2^Q$ is the transition function

$\beta = (\beta_i \mid i \in L)$ $\beta_i : Q \rightarrow X_i$ is the output map for port i

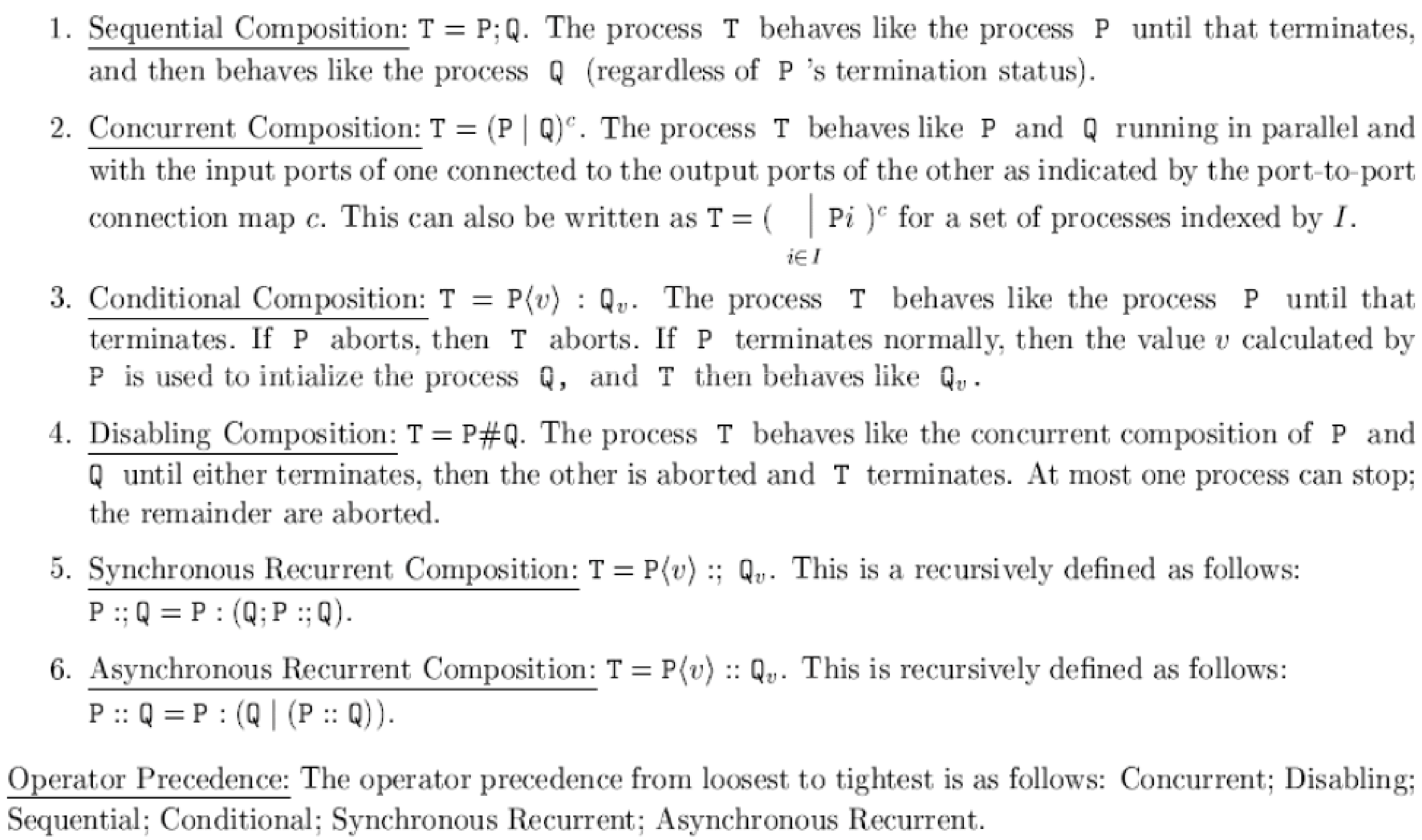
$\tau \in 2^Q$ is the set of start states

ADAPT

The behavior of every RS schema is defined using port automata. This provides precision to the semantics and also a constructive means of reasoning about the behavior and meaning of schemas.



ADAPT



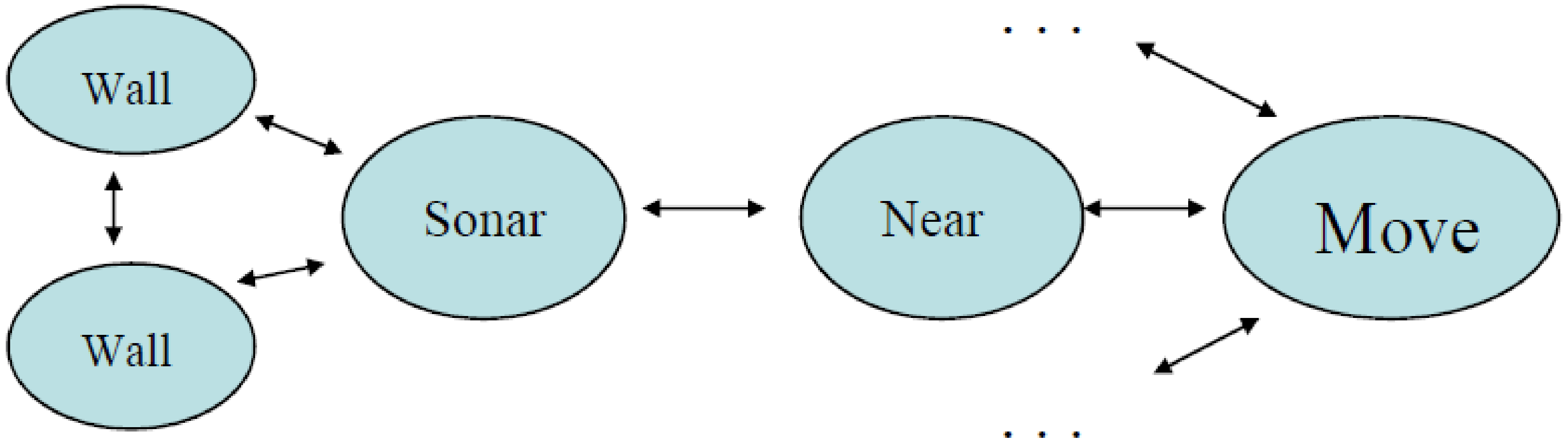
1. Sequential Composition: $\mathtt{T} = \mathtt{P};\mathtt{Q}$. The process $\mathtt{T}$ behaves like the process $\mathtt{P}$ until that terminates, and then behaves like the process $\mathtt{Q}$ (regardless of $\mathtt{P}$'s termination status).
2. Concurrent Composition: $\mathtt{T} = (\mathtt{P} \mid \mathtt{Q})^c$. The process $\mathtt{T}$ behaves like $\mathtt{P}$ and $\mathtt{Q}$ running in parallel and with the input ports of one connected to the output ports of the other as indicated by the port-to-port connection map c. This can also be written as $\mathtt{T} = (\mathop{\Big|}_{i \in I} \mathtt{P}i \;)^c$ for a set of processes indexed by I.
3. Conditional Composition: $\mathtt{T} = \mathtt{P}\langle v \rangle : \mathtt{Q}_v$. The process $\mathtt{T}$ behaves like the process $\mathtt{P}$ until that terminates. If $\mathtt{P}$ aborts, then $\mathtt{T}$ aborts. If $\mathtt{P}$ terminates normally, then the value v calculated by $\mathtt{P}$ is used to intialize the process $\mathtt{Q}$, and $\mathtt{T}$ then behaves like $\mathtt{Q}_v$.
4. Disabling Composition: $\mathtt{T} = \mathtt{P}\#\mathtt{Q}$. The process $\mathtt{T}$ behaves like the concurrent composition of $\mathtt{P}$ and $\mathtt{Q}$ until either terminates, then the other is aborted and $\mathtt{T}$ terminates. At most one process can stop; the remainder are aborted.
5. Synchronous Recurrent Composition: $\mathtt{T} = \mathtt{P}\langle v \rangle :; \mathtt{Q}_v$. This is a recursively defined as follows: $\mathtt{P} :; \mathtt{Q} = \mathtt{P} : (\mathtt{Q}; \mathtt{P} :; \mathtt{Q})$.
6. Asynchronous Recurrent Composition: $\mathtt{T} = \mathtt{P}\langle v \rangle :: \mathtt{Q}_v$. This is recursively defined as follows: $\mathtt{P} :: \mathtt{Q} = \mathtt{P} : (\mathtt{Q} \mid (\mathtt{P} :: \mathtt{Q}))$.

Operator Precedence: The operator precedence from loosest to tightest is as follows: Concurrent; Disabling; Sequential; Conditional; Synchronous Recurrent; Asynchronous Recurrent.

ADAPT

Schemas, facts, and hypotheses are nodes in a graph.
Links implement the composition operations, as well as otl
relations, including deductive and evidential inference.

Automata that implement a schema are built as needed.



ADAPT

The basic loop of ADAPT is:

1 - check Soar's output link to see if there are any commands, which may be either motion commands for the robot or modeling commands for the World Model,

2 - blend the motion commands that are to be sent to the robot,

3 - send all robot commands both to the robot and to the virtual robot in the World Model,

4 - send all other commands to the World Model,

5 – periodically (every tenth of a second) fetch data from the robot to be put into Soar's working memory,

6 - periodically fetch data from the Vision System, compare it to visual data from the World Model, and put any significant differences into Soar's working memory.

ADAPT

Il principale metodo di apprendimento è basato sul *chunking learning* del SOAR e consiste nella generazione di nuovi *chunk*.

I *chunk* generati possono essere di due tipi:

1. sfrutta una stima bayesiana per decidere l'azione da compiere. Questo permette di effettuare la stessa scelta che farebbe ACT-R in un singolo step.
2. riassume il risultato di una ricerca effettuando la stessa scelta di SOAR in un singolo step.

Il risultato dell'apprendimento comporta la modifica degli schemi esistenti.

ICARUS

Icarus designed as an integrated acrchitecture for controlling an agent that exists in a complicated physical environment.

The agent was to be able to navigate and manipulate objects in the environement.

The design is modular, using separate modules for planning, perception, execution and long-term memory.

ICARUS

Designs for ICARUS have been guided by six principles:

1. Cognitive reality of physical objects
2. Cognitive separation of categories and skills
3. Primacy of categorization and skill execution
4. Hierarchical organization of long-term memory
5. Correspondence of long-term/short-term structures
6. Modulation of symbolic structures with utility functions

These ideas distinguish ICARUS from most other architectures.

ICARUS

Designs for ICARUS

1. ARGUS – perception

   an attention mechanism to determine which of these changes is worthy of attention

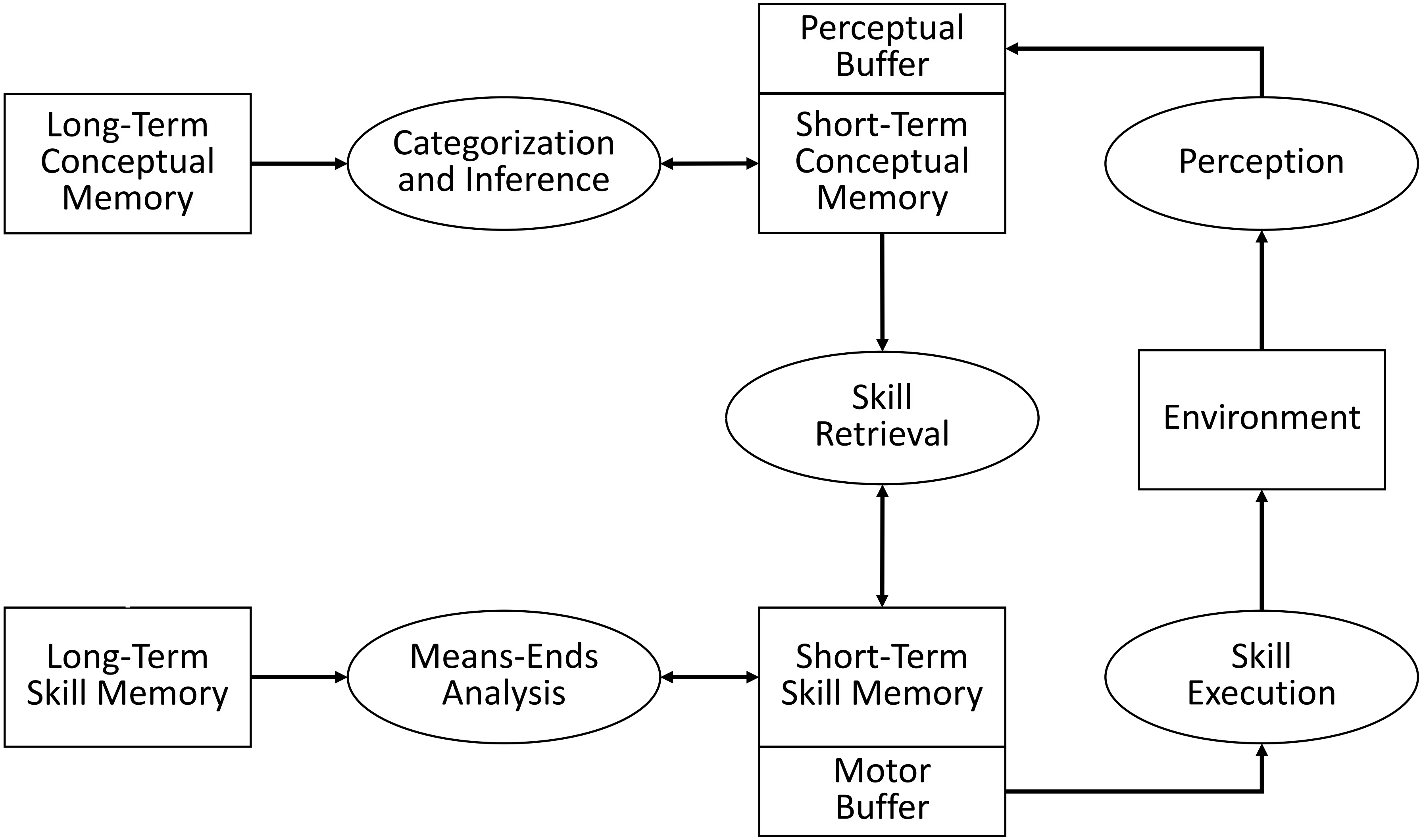
2. DAEDALUS – planning

   heuristic best-first search through the problem space.

3. MAENDER – execution
   executes all the primitive actions

4. LABYRINTH – memory
   probabilistic hierarchy to store the knowledge

Overview of the ICARUS Architecture



Some Concepts from the Blocks World

```
(on  (?block1 ?block2)
 :percepts     ((block ?block1 xpos ?x1 ypos ?y1)
                (block ?block2 xpos ?x2 ypos ?y2 height ?h2))
 :tests        ((equal ?x1 ?x2)
                (>= ?y1 ?y2)
                (<= ?y1 (+ ?y2 ?h2))) )

(clear  (?block)
 :percepts     ((block ?block))
 :negatives    ((on ?other ?block)) )

(unstackable  (?block ?from)
 :percepts     ((block ?block) (block ?from))
 :positives    ((on ?block ?from)
                (clear ?block)
                (hand-empty)) )
```

Primitive Skills from the Blocks World

```
(pickup  (?block ?from)
 :percepts     ((block ?block xpos ?x)
                (table ?from height ?h))
 :start        ((pickupable ?block ?from))
 :requires     ( )
 :actions      ((∗ move ?block ?x (+ ?h 10)))
 :effects      ((holding ?block))
 :value        1.0 )

(stack  (?block ?to)
 :percepts     ((block ?block)
                (block ?to xpos ?x ypos ?y height ?h))
 :start        ((stackable ?block ?to))
 :requires     ( )
 :actions      ((∗ move ?block ?x (+ ?y ?h)))
 :effects      ((on ?block ?to)
                (hand-empty))
 :value        1.0 )
```

A Nonprimitive Skill from the Blocks World

```
(puton  (?block ?from ?to)
 :percepts     ((block ?block) (block ?from) (table ?to))
 :start        ((ontable ?block ?from) (clear ?block)
                (hand-empty) (clear ?to))
 :requires     ( )
 :ordered      ((pickup ?block ?from) (stack ?block ?to))
 :effects      ((on ?block ?to))
 :value        1.0 )

(puton  (?block ?from ?to)
 :percepts     ((block ?block) (block ?from) (block ?to))
 :start        ((on ?block ?from) (clear ?block)
                (hand-empty) (clear ?to))
 :requires     ( )
 :ordered      ((unstack ?block ?from) (stack ?block ?to))
 :effects      ((on ?block ?to))
 :value        1.0 )
```

Hierarchical Organization of Memory

ICARUS' long-term memories are organized into hierarchies:

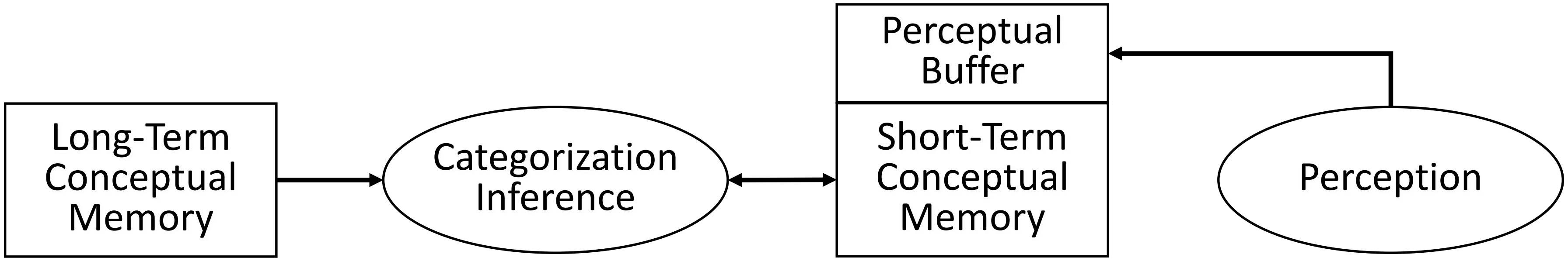
- concepts can refer to percepts and to other concepts;
- skills refer to percepts, to concepts, and to other skills.

Conceptual memory is similar to a network, but each node represents a meaningful category.

Different expansions for skills and concepts also make them similar to Horn clause programs.

These hierarchies are encoded by direct reference, rather than through working-memory elements, as in ACT and Soar.

Categorization and Inference



For each cycle, perception puts object descriptions into the perceptual buffer.

The system matches its concepts against the contents of this buffer.

Categorization proceeds in an automatic, bottom-up manner.

Monotonic inference that adds concept instances to short-term memory.

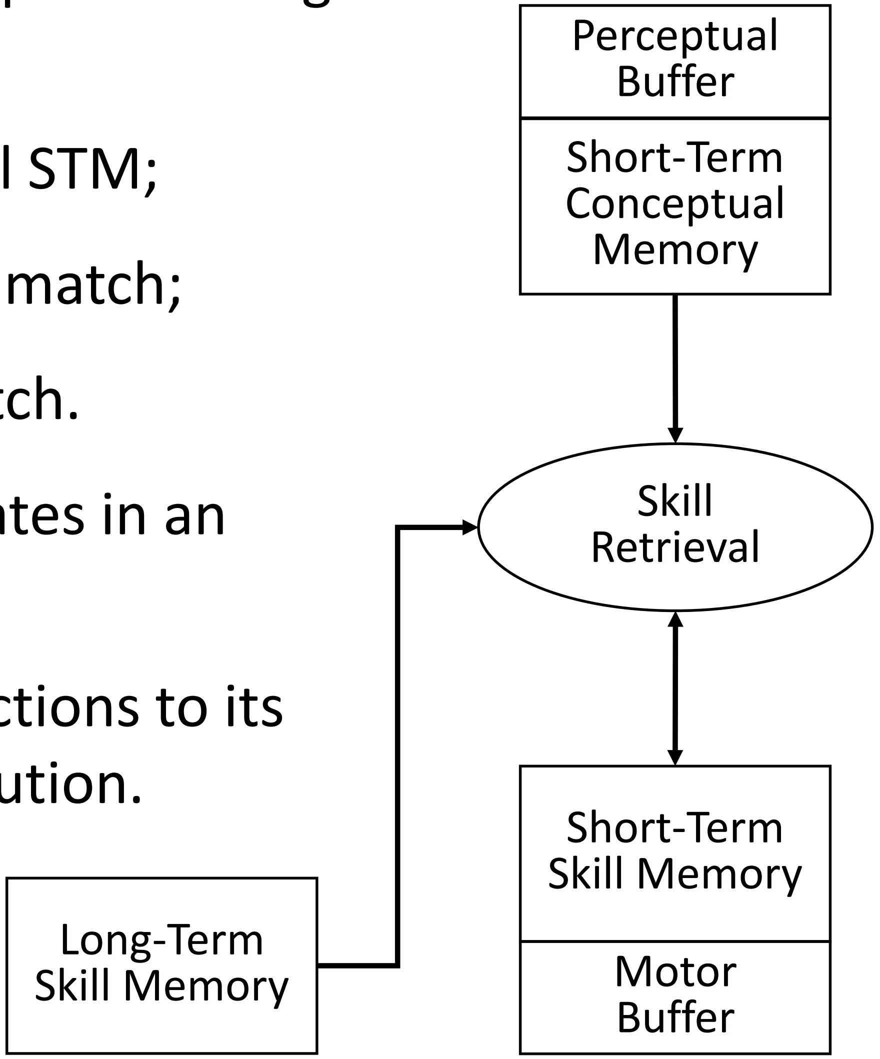
Retrieving and Matching Skill Paths

For each cycle, ICARUS finds all paths through its skill hierarchy:

- begin with an instance in skill STM;
- start and requires fields that match;
- effects fields that do not match.

Each instantiated path terminates in an executable action.

ICARUS adds these candidate actions to its motor buffer for possible execution.



Evaluating and Executing Skills

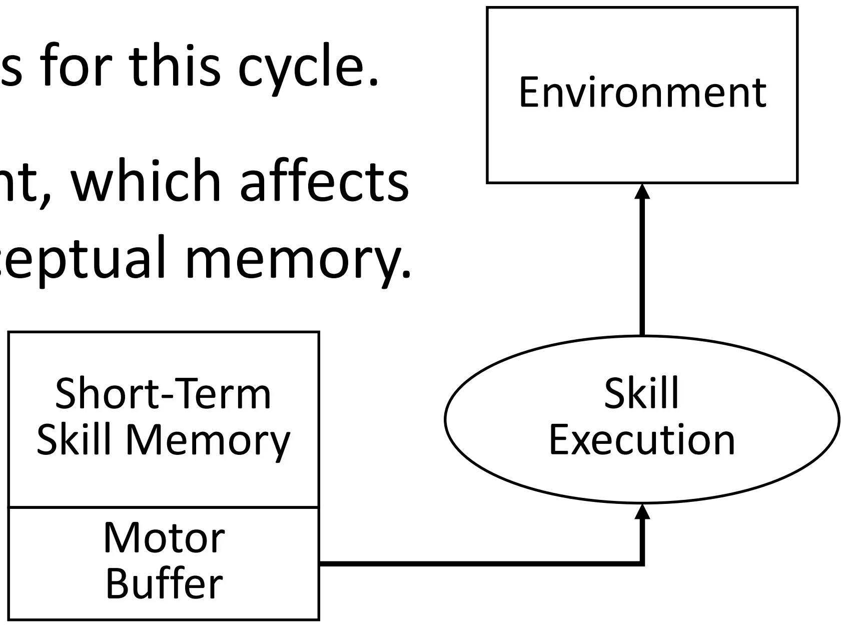
For each selected path, ICARUS computes a utility by summing the values of each skill along that path.

For each path, with decreasing utility:

• If required resources are available, execute actions;

• If executed, commit the resources for this cycle.

These actions alter the environment, which affects the perceptual buffer and the conceptual memory.



Cognitive Systems and Robotics

- Since 2001: Cognitive Systems intensely funded by the EU "Robots need to be more robust, context-aware and easy-to-use. Endowing them with advanced learning, cognitive and reasoning capabilities will help them adapt to changing situations, and to carry out tasks intelligently with people"

- More than 100 projects on Cognitive Systems funded; many have an attention module (e.g. MACS, Paco-plus, CogX)