

# Soar Tutorial: Building Intelligent Agents Using Soar

John E. Laird

University of Michigan

30<sup>th</sup> Soar Workshop, 2010

laird@umich.edu



# Tutorial Outline

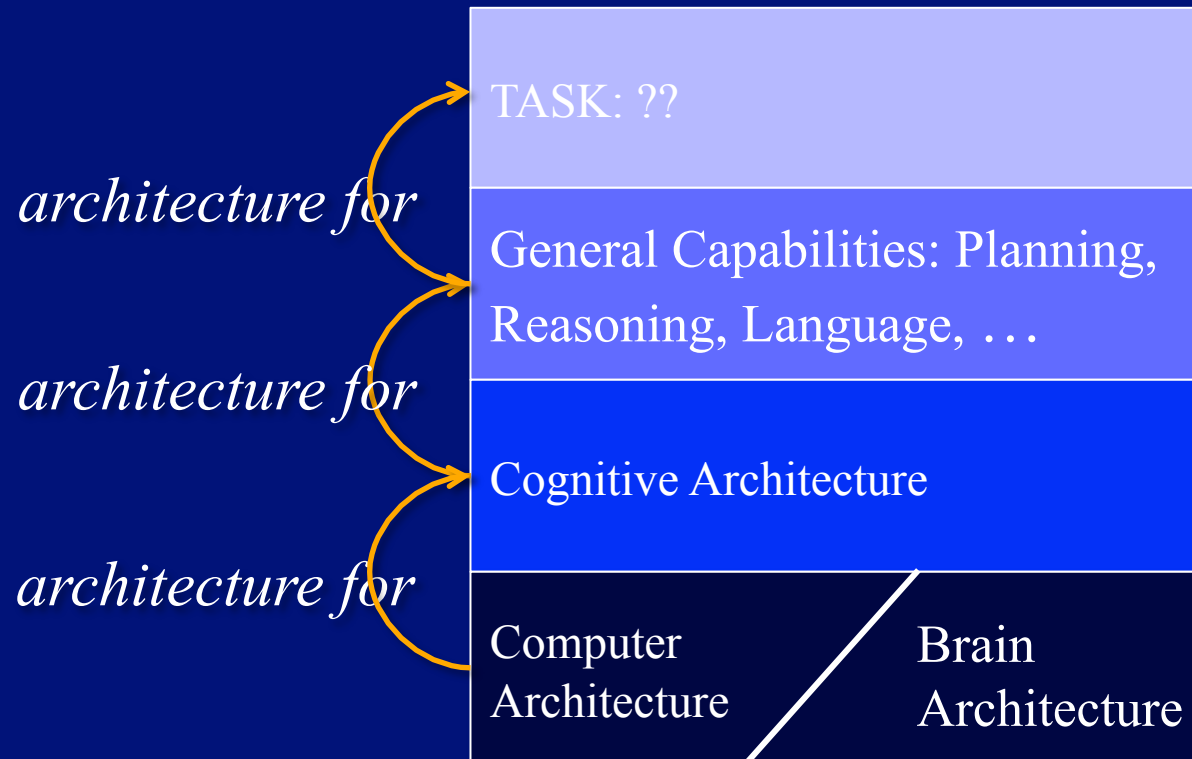
1. Cognitive Architecture
2. Soar History
3. Hello-World (simple syntax)
4. Water Jug (internal problem solving)
5. Eaters (simple external interaction)
6. Tank-Soar (more external interaction & subgoals)
7. Water Jug with subgoals and learning

My goal is for you to understand Soar enough to learn the rest on your own from the Tutorial and Manual.

# Research Goals and Approach

- Goal:
  - General, human-level behavior
  - Human capabilities across a broad range of tasks
- Approach:
  - Cognitive Architecture = fixed structures, mechanisms, and representations
  - Emphasized functionality & higher level cognition
    - Effective and efficient end-to-end performance
    - Scale to very large knowledge bases
    - Make use of whatever forms of knowledge available
    - Meta-reasoning, episodic memory, appraisals

# Role of Cognitive Architecture



# What is Cognitive Architecture?

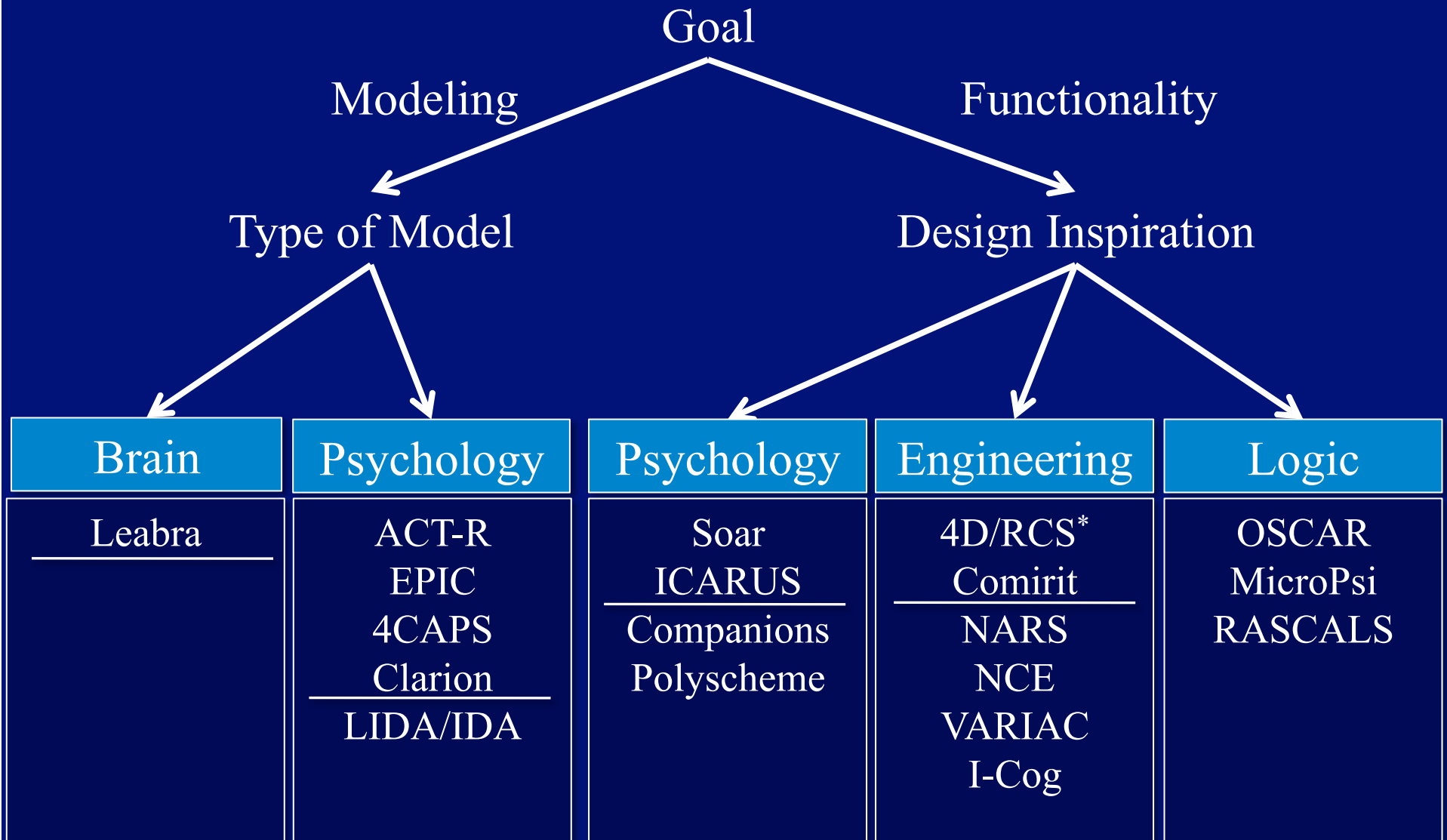
Fixed mechanisms and structures that underlie cognition

- Processors that manipulate data
  - Memories that hold knowledge
  - Representations of knowledge
  - Interfaces with an environment
- 
- Sharp distinction between
    - task-independent architecture and
    - task-dependent knowledge

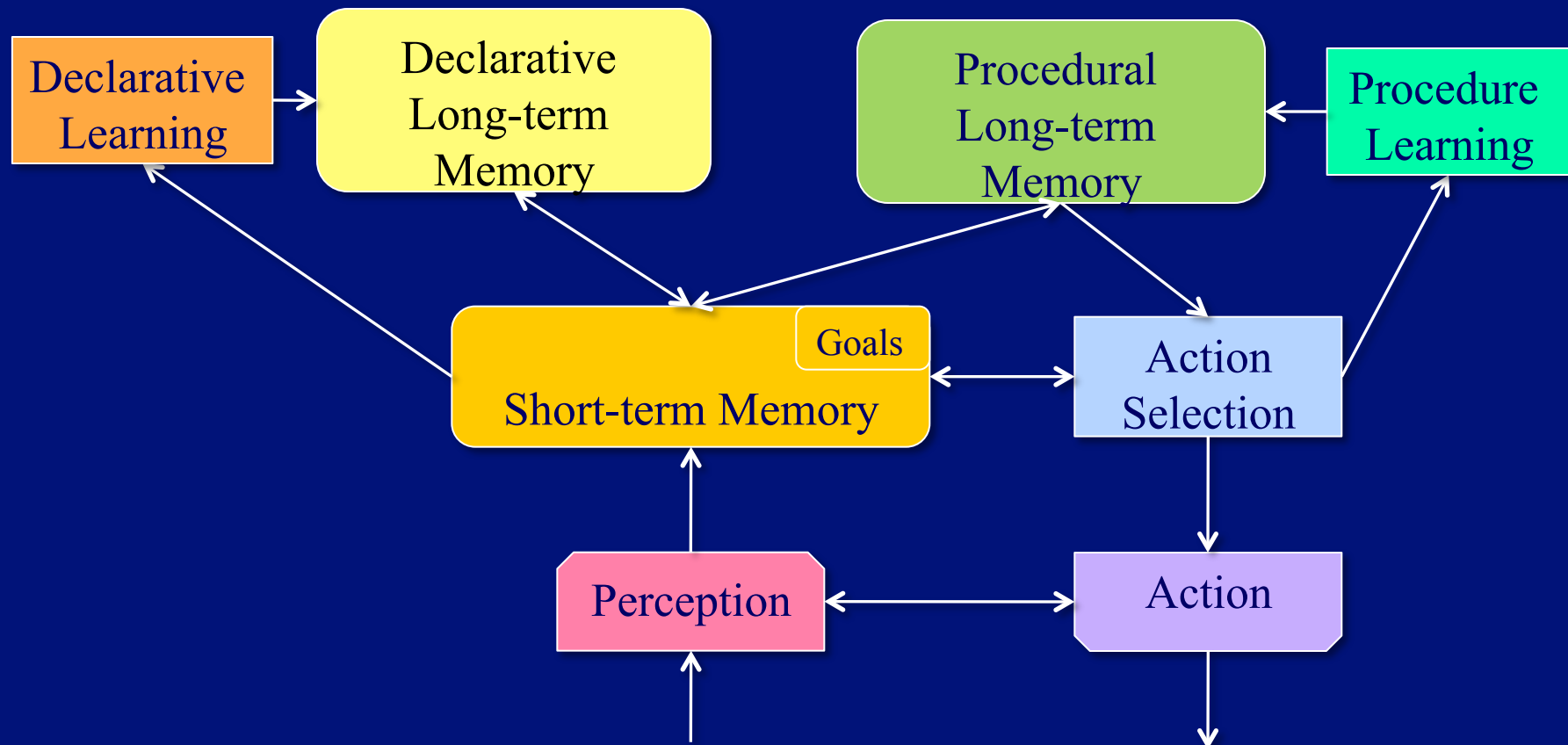
# Different Goals of Cognitive Architecture Research

- Biological modeling:
  - Does the architecture correspond to what we know about the brain?
- Psychological modeling:
  - Does the architecture capture the details of human performance in a wide range of cognitive tasks?
- Functionality:
  - Does the architecture explain how humans achieve their high level of intellectual function?
  - Does the architecture support the creation of useful systems?

# Classification of Current Architectures



# Common Structures of many Cognitive Architectures





# Common Processing Across Architectures

- Complex behavior arises from sequence of simple decisions over internal and external actions controlled by knowledge
  - Significant internal parallelism, limited external parallelism
  - For cognitive modeling, ~50msec is basic cycle time of cognition
- Knowledge access must be bounded for reactivity
- Learning is incremental & on-line

# Examples of Cognitive Architectures

- ACTE through ACT-R (Anderson, 1976; Anderson, 1993)
- Soar (Laird, Rosenbloom, & Newell, 1984)
- Prodigy (Minton & Carbonell., 1986; Veloso et al., 1995)
- PRS (Georgeff & Lansky, 1987)
- CIRCA (Munsliner & Atkins, 1993)
- 3T (Gat, 1991; Bonasso et al., 1997)
- EPIC (Kieras & Meyer, 1997)
- APEX (Freed et al., 1998)
- 4D/RCS (Albus)
- Clarion (Sun)
- Polyscheme (Cassimatis 2004)
- ICARAUS (Langley & Shapiro, 2003)

These systems cover only a small region of the space of possible architectures.

# Why Architecture Matters

- A commitment to common computational primitives which determine:
  - The complexity profile of an agent's computations
  - The building blocks for creating a complete agents
  - The primitive unit of reasoning/deliberation/learning
  - The primitive units of knowledge
  - What is fixed and unchanging vs. what is programmed/learned
- Major achievements: integration
  - Reaction, deliberation, planning, meta-reasoning, learning
    - Lots of knowledge that is really used
  - Integrated theory of wide range of human behavior
    - 50ms is a magic number
  - Taskable performance/embedded systems

# Historical Perspective: Soar

1960

1970

1980

1990

2000

2010

**Human Problem Solving**  
**Goal-directed search**  
**Rule-based systems**  
**Newell & Simon**



# Historical Perspective

1960

1970

1980

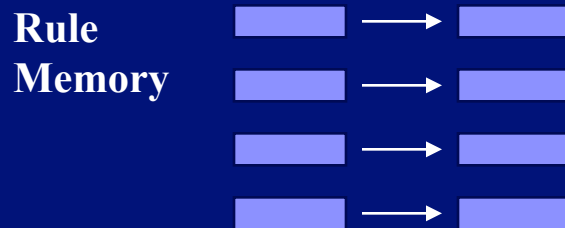
1990

2000

2010

**Efficient rule-based systems**

**Expert Systems**



# Historical Perspective

1960

1970

1980

1990

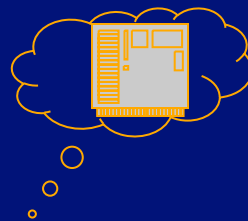
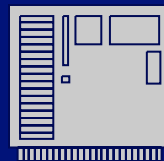
2000

2010

## Soar

Multi-method problem solving  
Knowledge-based, hierarchical  
reasoning, search, meta-level  
reasoning, and learning

“Inside the head” problems  
R1-Soar: Computer Configuration



# Historical Perspective

1960

1970

1980

1990

2000

2010

## External environments

Extreme efficiency

Mobile robot control

Stick control of simulated plane

## Model human behavior

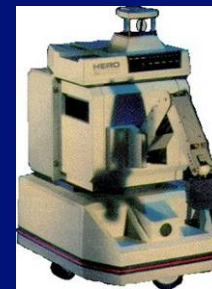
Natural language

Human-computer interaction

Many forms of learning



Air-Soar



Hero-Soar

# Historical Perspective

1960

1970

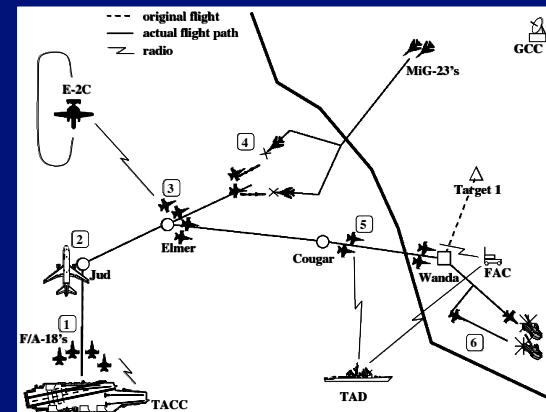
1980

1990

2000

2010

**Intelligent Forces for Training**  
WISSARD/IFOR (DARPA)  
Fixed-wing aircraft (UM)  
Rotary-wing aircraft (USC/ISI)  
STOW-E  
STOW-97 – 700 sorties, 100 in air



**TacAir-Soar**



# Historical Perspective

1960

1970

1980

1990

2000

2010

**Soar Technology, Inc.**  
Develop and deploy IFORs  
**Computer Game AIs**  
**USC/ICT**  
Teamwork  
Integrated Virtual Humans

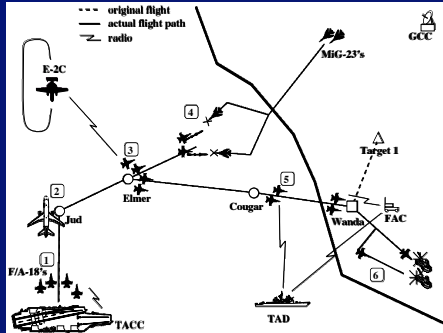


**Soar Quakebot**



**Haunt 2**

# Example Applications



**TacAir-Soar**  
*Complex Doctrine & Tactics Execution*



**Urban Combat**  
*Transfer Learning*



**Soar Quakebot**  
*Anticipation of Enemy Actions*



**Haunt**  
*Actors and Automated Direction*



**Soar MOUTbot**  
*Team Tactics and Unpredictable Behavior*



**SORTS**  
*Spatial Reasoning & Real-time Strategy*



**Scout Domain**  
*Mental Imagery*



**Splinter-Soar**  
*Robot Control*

# Historical Perspective

1960

1970

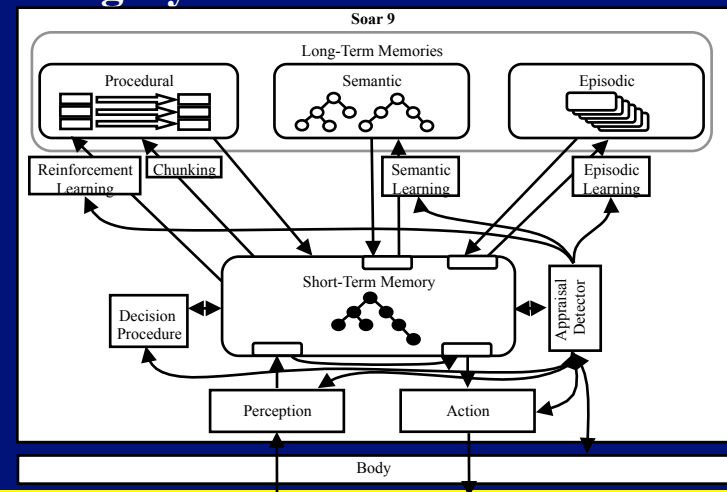
1980

1990

2000

2010

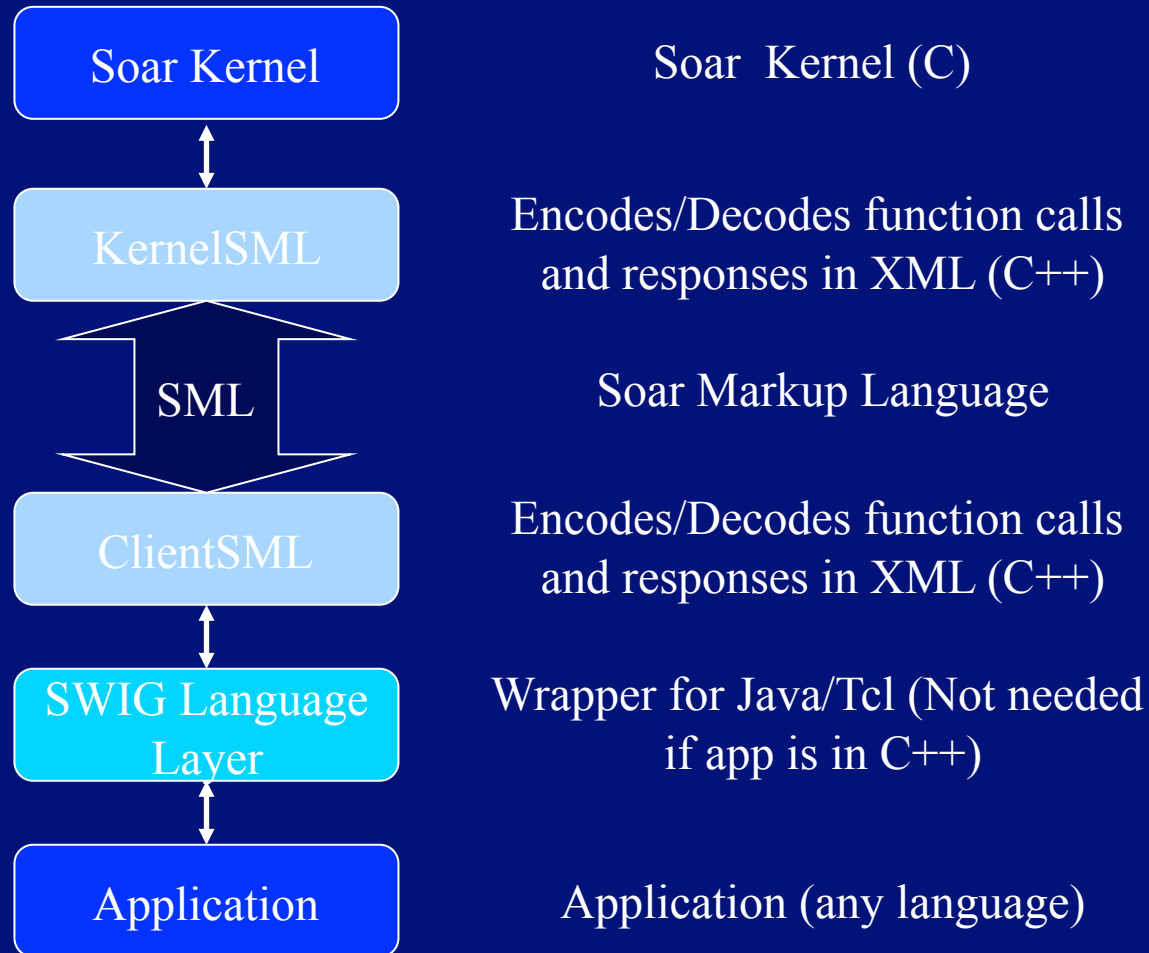
**Architectural Extensions**  
**Episodic & Semantic Learning**  
**Reinforcement Learning**  
**Emotions**  
**Imagery**



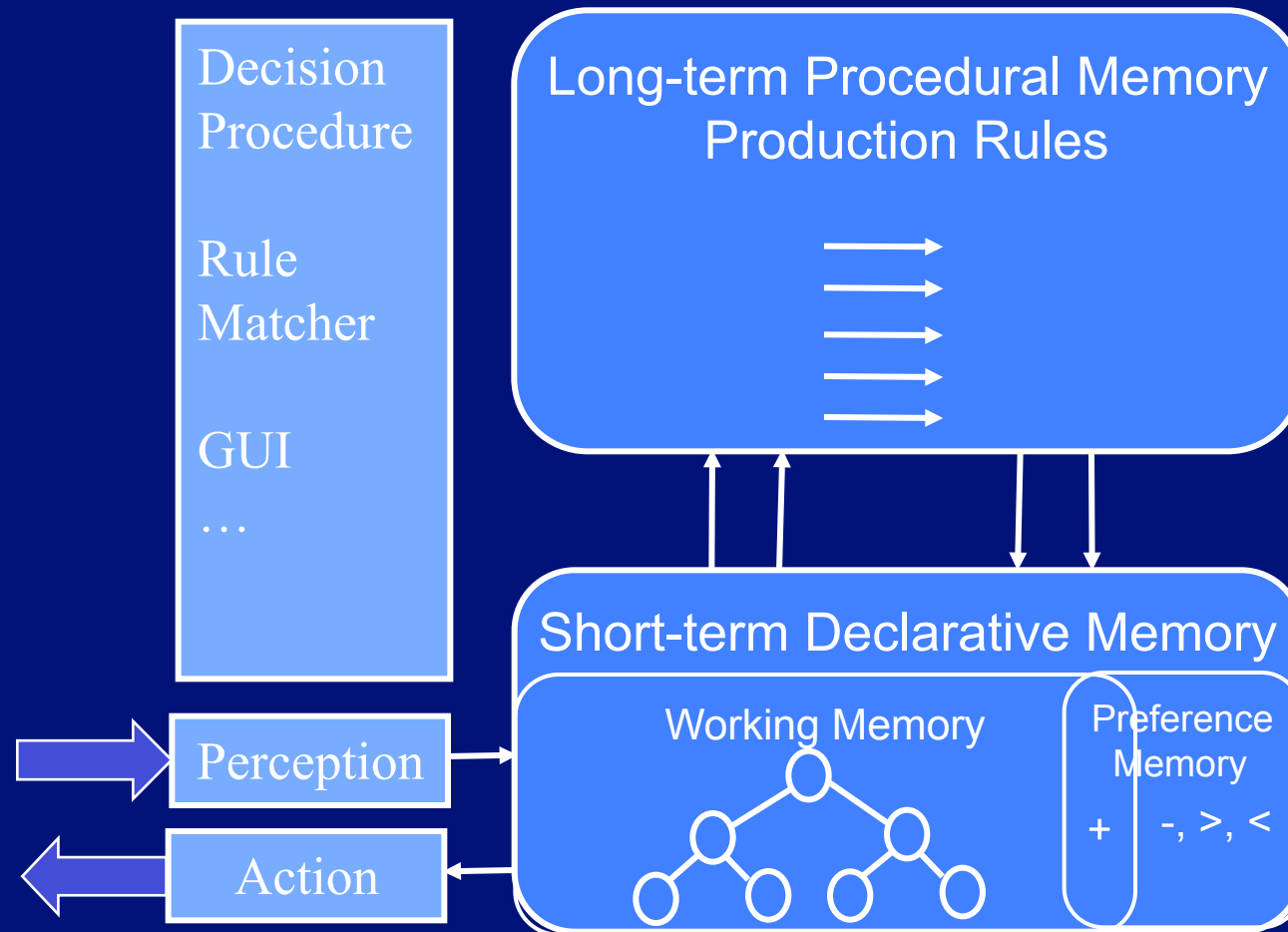
# Distinctive Features of Soar

- High performance
  - Can build very large systems that run for a long time
- Integrates reaction, deliberation, meta-reasoning
  - Dynamically switching between them
- Integrated learning
  - Adding reinforcement learning, episodic & semantic
- Useful in cognitive modeling
  - Expanding this is emphasis of many current projects
- Easy to integrate with other systems & environments
  - SML efficiently supports many languages, inter-process
- Many tools to aid development
  - Visual-Soar, Debugger, ...

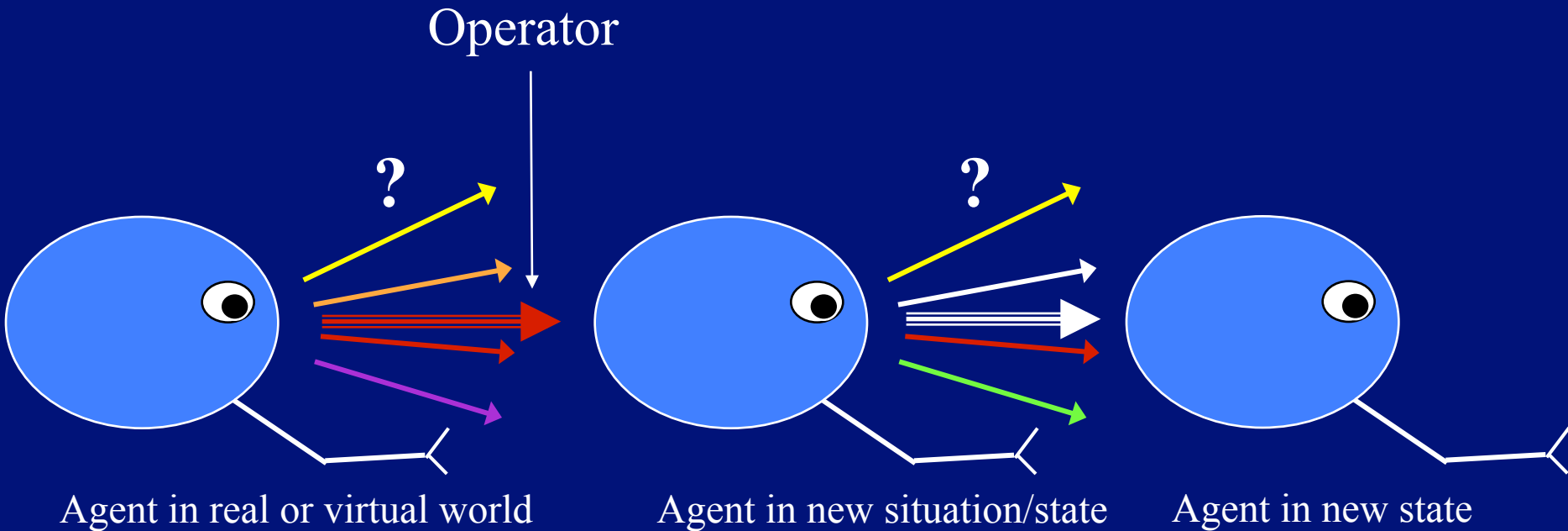
# System Architecture



# Basic Soar Structure



# Core Soar



# Core Soar

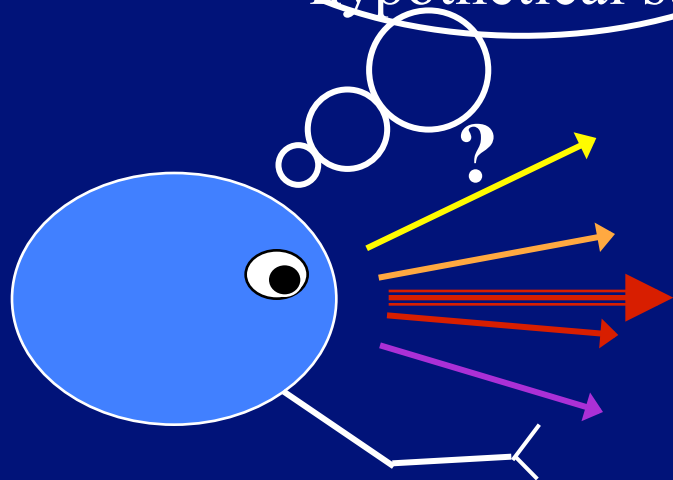
Long-term memory

Possible actions, effects of actions, facts,  
episodes, expected rewards, preferences,

...

Short-term memory

Current situation, goals, intentions,  
hypothetical states, ...





# Soar Basics

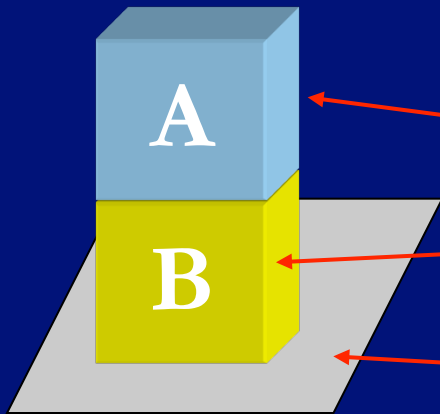
- *Operators*: Deliberate changes to internal/external state
- Activity is a series of operators controlled by knowledge:
  1. Input from environment
  2. Elaborate current situation: *parallel rules*
  3. Propose and evaluate operators via preferences: *parallel rules*
  4. Select operator
  5. Apply operator: Modify internal data structures: *parallel rules*
  6. Output to motor system

# Operator Selection

- Current operator only changes when decision changes.
- Reasons for new decision:
  - proposal instantiation no longer matches and retracts proposal
  - or other operators dominate selection through preferences



# Example Working Memory

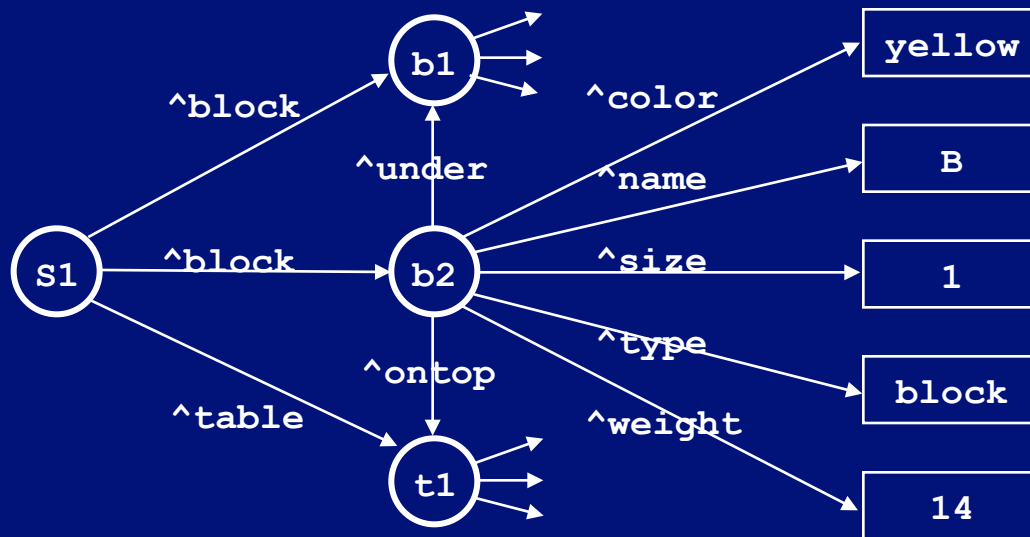


(s1 ^block b1 ^block b2 ^table t1)

(b1 ^color blue ^name A ^ontop b2 ^size 1  
^type block ^weight 14)

(b2 ^color yellow ^name B ^ontop t1 ^size 1  
^type block ^under b1 ^weight 14)

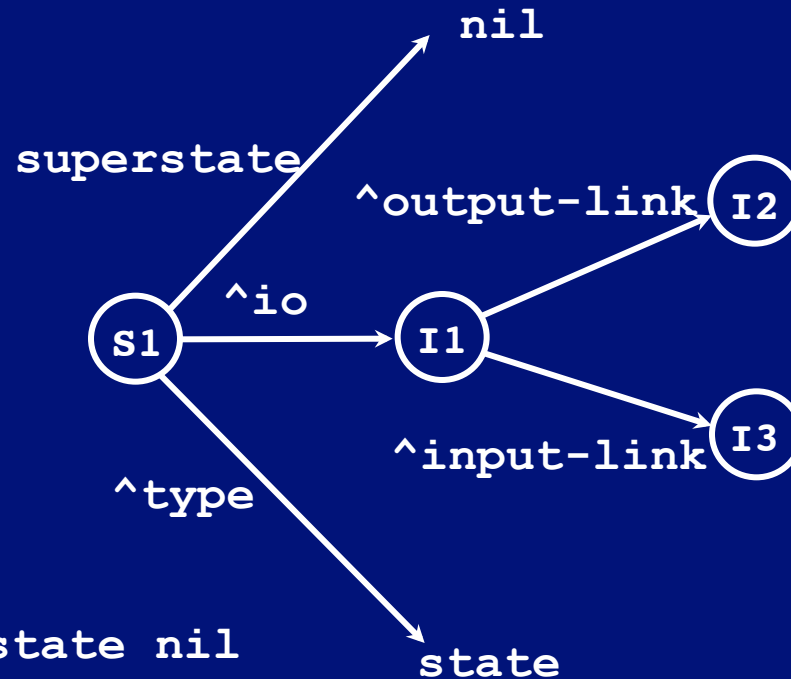
(t1 ^color gray ^shape square  
^type table ^under b2)



Working memory is a graph.

All working memory elements must be “linked” directly or indirectly to a state.

# Initial Working Memory



```
S1 ^superstate nil
```

```
S1 ^io I1
```

```
S1 ^type state
```

```
I1 ^output-link I2
```

```
I1 ^input-link I3
```

```
(S1 ^io I1 ^superstate nil ^type state)
```

```
(I1 ^input-link I3 ^output-link I2)
```

# Simple Soar Syntax

## Hello World Rule

If I exist,  
then write |Hello World| and halt.

```
sp {hello-world
   (state <s> ^type state)
-->
   (write |Hello World|)
   (halt)}
```

# Hello World Operator

**Propose\*hello-world:**

If I exist, propose the hello-world operator.

**Apply\*hello-world:**

If the hello-world operator is selected, write "Hello World" and halt.

```
sp {propose*hello-world
  (state <s> ^type state)
-->
  (<s> ^operator <o> +)
  (<o> ^name hello-world)}
```

Creating acceptable  
preference for  
operator

```
sp {apply*hello-world
  (state <s> ^operator <o>)
  (<o> ^name hello-world)
-->
  (write |Hello World|)
  (halt)}
```

Testing selected  
operator



# Soar 101

## Internal Problem Solving

Elaborate  
State

Propose  
Operator

Compare  
Operators

Select  
Operator

Apply  
Operator

Decision  
Procedure

```
sp {propose*hello-world  
  (<s> ^type state)  
-->  
  (<s> ^operator <o> +)  
  (<o> ^name hello-world)}
```

```
sp {apply*hello-world  
  (<s> ^operator <o>)  
  (<o> ^name hello-world)  
-->  
  (write |Hello World|)  
  (halt)}
```

Production  
Memory

```
(s1 ^type state  
  ^superstate nil  
  ^io i1  
  ...)  
(s1 ^operator o1 +)  
(o1 ^name hello-world)  
(s1 ^operator o1)
```

Working  
Memory

Hello World

# Operators in Working Memory

- To be considered for selection, an operator must have an acceptable preference on the state.  
`(s1 ^operator o1 +)`
- Operators must have a declarative representation in working memory (something rules can test, such as name).  
`(o1 ^name hello-world)`
- Rules can test for an acceptable operator preference  
`(<s> ^operator <o> +)` and create more preferences.
- When an operator is *selected*, there is a working memory element in the state (different than the preference)  
`(s1 ^operator o1)`
- Rules that test for a selected operator  
`(<s> ^operator <o>)` apply the operator by modifying the state.



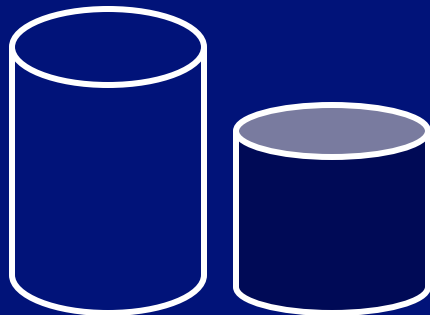
# Water Jug Problem

You are given two empty jugs. One holds five gallons of water and the other holds three gallons.

There is a well that has unlimited water that you can use to completely fill the jugs. You can also empty a jug or pour water from one jug to another.

There are no marks for intermediate levels on the jugs.

The goal is to fill the three-gallon jug with one gallon of water.



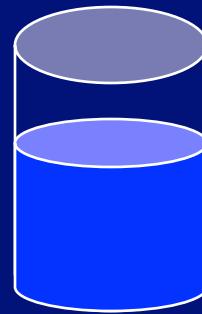
# Operators and States

- Operators:
  - Fill a jug from the well.
  - Empty a jug into the well.
  - Pour water from a jug to a jug.

- States

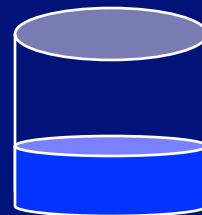
- Jug-a

- Volume: 5 gallons
    - Contents: X gallons
    - Empty: Y gallons



- Jug-b

- Volume: 3 gallons
    - Contents: M gallons
    - Empty: N gallons



# Water Jug State Structure

- name water-jug
    - (`<s> ^name water-jug`)
  - jug-a
    - Volume: 5 gallons
      - (`<j1> ^volume 5`)
    - Contents: x gallons
      - (`<j1> ^contents 0`)
    - Empty: y gallons
      - (`<j1> ^empty 5`)
  - jug-b
    - Volume: 3 gallons
      - (`<j2> ^volume 3`)
    - Contents: m gallons
      - (`<j2> ^contents 0`)
    - Empty: n gallons
      - (`<j2> ^empty 3`)
- (`<s> ^jug <j1>`) multi-valued attribute  
(`<s> ^jug <j2>`) attribute

# Soar 102

## Internal Problem Solving



<p>If jug &lt;j&gt; has content &lt;c&gt;, volume &lt;v&gt;, --&gt; ^empty &lt;v&gt; - &lt;c&gt;</p>	<p><b>If jug &lt;j&gt; is empty</b>          ^contents 0, 0,          propose operator          propose operator          to fill jug &lt;j&gt;</p>	<p>If operator &lt;o&gt; empties a jug --&gt; operator &lt;o&gt; &lt;</p>	<p>If selected operator is initialize state --&gt; &lt;j1&gt; ^contents 0 ^volume 5 &lt;j2&gt; ^contents 0 ^volume 3</p>	<p><b>Production Memory</b></p>
--	---	---	--	---------------------------------

j1                  j2

Operator: initialize state  
 Operator proposal: fill j1, fill j2

j1 ^volume 5 ^contents 0 *Persistent*  
   ^empty 0 *Not-persistent*

j2 ^volume 3 ^contents 0 *Persistent*  
   ^empty 3 *Not-persistent*

**Short term Memory (working memory & preference memory)**

# Water Jug Operators

- Initialize-water-jug
- Fill a jug from the well
- Empty a jug into the well
- Pour water from a jug to a jug
  
- For every operator, must define at least two rules:
  1. Proposal creates operator structure in working memory
    - Usually includes name and parameters (^fill-jug, ^empty-jug)
  2. Application tests for selected operator
    - Makes changes to the state (based on parameters)
  
- Can also create evaluation rules, but not always necessary
  - These rules create preferences

# Initialize-water-jug

- Proposal

If no task is selected,  
then *propose* the initialize-water-jug operator.

- Application

If the initialize-water-jug operator is *selected*,  
then create an empty 5 gallon jug and an empty 3 gallon jug.

```
sp {propose*initialize-water-jug
  (state <s> ^superstate nil
    -^name)
-->
(<s> ^operator <o> +)
(<o> ^name initialize-water-jug)}
```

Test that name  
doesn't exist

```
sp {apply*initialize-water-jug
  (state <s> ^operator <o>)
  (<o> ^name initialize-water-jug)
-->
(<s> ^name water-jug
  ^jug <j1>
  ^jug <j2>)
(<j1> ^volume 5
  ^contents 0)
(<j2> ^volume 3
  ^contents 0)}
```

# Elaboration of $\hat{\text{empty}}$

If a jug has volume  $v$  and contents  $c$ , then it has empty  $v - c$ .

```
sp {water-jug*elaborate*empty
  (state <s> ^name water-jug
    ^jug <j>)
  (<j> ^volume <v>
    ^contents <c>)
  -->
  (<j> ^empty (- <v> <c>)) }
```

Subtraction of  $\langle c \rangle$  from  $\langle v \rangle$



$\hat{\text{empty}}$  is *instantiation-supported* = *i-support*

When this instantiation retracts, the working memory element is removed.

The rule may match new values and produce a new working memory element.

# Instantiations

```
sp {water-jug*elaborate*empty
  (state <s> ^name water-jug
    ^jug <j>)
  (<j> ^volume <v>
    ^contents <c>)
  -->
  (<j> ^empty (- <v> <c>)) }
```

For each set of WMEs that successfully match the rule, an *instantiation* is created.

```
(s1 ^name water-jug)    (s1 ^name water-jug)
(s1 ^jug j1)            (s1 ^jug j2)
(j1 ^volume 5)         (j2 ^volume 3)
(j1 ^contents 0)       (j2 ^contents 0)
```

Both instantiations fire in parallel, creating two new WMEs:

```
(j1 ^empty 5)          (j2 ^empty 3)
```

If one of the matched WMEs in an instantiation is removed, the WME it created is removed.



# Fill Jug

- Proposal

If there is a jug that is not full, then *propose* the fill operator.

- Application

If the fill operator is *selected* for a jug, then change the contents of that jug to its volume.

```
sp {water-jug*propose*fill-water-jug
  (state <s> ^name water-jug
    ^jug <j>)
  (<j> ^empty > 0)
-->
(<s> ^operator <o> + =)
(<o> ^name fill
  ^fill-jug <j>)
```

Only match if  
value > 0

= means indifferent  
(a random selection will be made)

```
sp {water-jug*apply*fill-water-jug
  (state <s> ^name water-jug
    ^operator <o>)
  (<o> ^name fill
    ^fill-jug <j>)
  (<j> ^volume <v>
    ^contents <c>)
-->
(<j> ^contents <v>
  ^contents <c> -) }
```

Causes WME to  
be removed

# Multiple Instantiations

```
sp {water-jug*propose*fill-water-jug
  (state <s> ^name water-jug
    ^jug <j>)
  (<j> ^empty > 0)
  -->
  (<s> ^operator <o> + =)
  (<o> ^name fill
    ^fill-jug <j>)}

```

For each set of working memory elements that successfully match the rule, an *instantiation* is created.

```
(s1 ^jug j1)           (s1 ^jug j2)
(j1 ^empty 5)         (j2 ^empty 3)

```

Both instantiations fire, creating two new operators and preferences:

## Working Memory Elements:

```
(s1 ^operator o1 +)    (s1 ^operator o2 +)
(o1 ^name fill)       (o2 ^name fill)
(o1 ^fill-jug j1)     (o2 ^fill-jug j2)

```

## Preferences:

```
(s1 ^operator o1 +)    (s1 ^operator o2 +)
(s1 ^operator o1 =)    (s1 ^operator o2 =)

```

The decision procedure will pick only one (randomly because they are indifferent).

# Persistence!

- Actions of non-operator application rules *retract* when rule no longer matches
  - No longer relevant to current situation
  - Operator proposals and state elaboration
  - Instantiation-support = i-support
  - *Rule doesn't test operator and modify state.*
    - *Elaborate state*
    - *Propose operator*
    - *Create operator preferences*
- Actions of operator application rules *persists* indefinitely
  - Otherwise actions retract as soon as operator isn't selected
  - Operators perform non-monotonic changes to state
  - Operator-support = o-support
  - *Rule tests a selected operator and modifies the state*
    - *Operator application*

# Empty Jug

## Proposal

If there is a jug that is not empty,  
then *propose* the empty operator.

```
sp {water-jug*propose*empty
  (state <s> ^name water-jug
    ^jug <j>)
  (<j> ^contents <> 0)
  -->
  (<s> ^operator <o> + =)
  (<o> ^name empty
    ^empty-jug <j>)} }
```

## Application

If the empty operator is *selected* for a jug,  
then change the contents of that jug to 0.

```
sp {water-jug*apply*empty
  (state <s> ^name water-jug
    ^operator <o>)
  (<o> ^name empty
    ^empty-jug <j>)
  (<j> ^contents <c>)
  -->
  (<j> ^contents 0
    ^contents <c> -)} }
```

# What you don't do in Soar

1. Must explicitly add and remove structures
  - No replace command
2. Cannot match variables in actions
3. Can't do math in conditions
  - Conditions can only test existence or absence of WME's
  - Equality or inequality of identifiers and constants
    - Simple inequality of numbers ( $>$ ,  $<$ ,  $>=$ ,  $<=$ ,  $<>$ )
4. Only simple calculations in actions
  - Allow simple math
  - Can do call outs (exec) to other languages/systems
    - Not encouraged
5. Complex calculations should be done via I/O
  - External computational aids (calculators, ...)

# Goal Detection

If there is a jug with volume three and contents one, then write that the problem has been solved and halt.

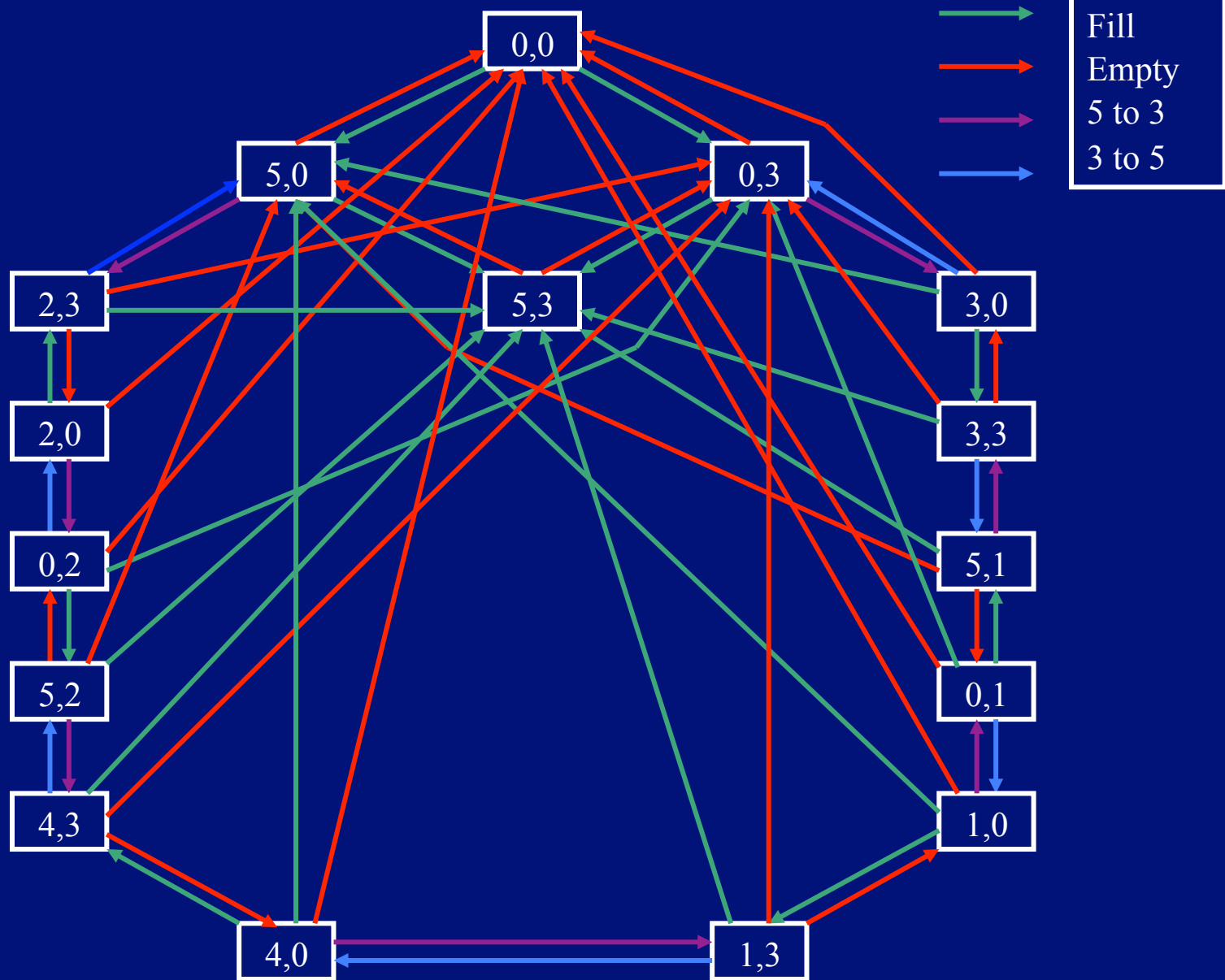
```
sp {water-jug*detect*goal*achieved
    (state <s> ^name water-jug
        ^jug <j>)
    (<j> ^volume 3
        ^contents 1)
```



-->

```
(write (crlf) |The problem has been solved.|)
(halt)}
```

# Water Jug Problem Space



# Simple Control Knowledge

Rules that influence operator selection using preferences

```
sp {water-jug*select*empty*worst
    (state <s> ^name water-jug
        ^operator <o> +)
    (<o> ^name empty)
-->
    (<s> ^operator <o> < ) }
```

```
sp {water-jug*select*empty*low
    (state <s> ^name water-jug
        ^operator <o> +)
    (<o> ^name empty)
-->
    (<s> ^operator <o> = 30) }
```



# Summary of Preferences

Acceptable:  $\langle o1 \rangle +$

Reject:  $\langle o1 \rangle -$

Better:  $\langle o1 \rangle > \langle o2 \rangle$

Worse:  $\langle o1 \rangle < \langle o2 \rangle$

Best:  $\langle o1 \rangle >$

Worst:  $\langle o1 \rangle <$

Indifferent:  $\langle o1 \rangle = \langle o2 \rangle$

Indifferent:  $\langle o1 \rangle =$

Indifferent:  $\langle o1 \rangle = 0-100$

# Maintain History of Last Operator

- What if want to avoid empty jug just filled?
- How do it? As part of operator application!
- If an operator is selected, then record the type of operator.
- If an operator is selected that differs from the recorded operator, remove the recorded operator.

```
## If an operator is selected
## record its name in last-operator
```

```
sp {water-jug*record*operator
    (state <s> ^name water-jug
        ^operator.name <name>)
-->
(<s> ^last-operator <name>)}

```

New syntax!

```
## If an operator is selected
## and its name is different than
## last-operator,
## remove last-operator
```

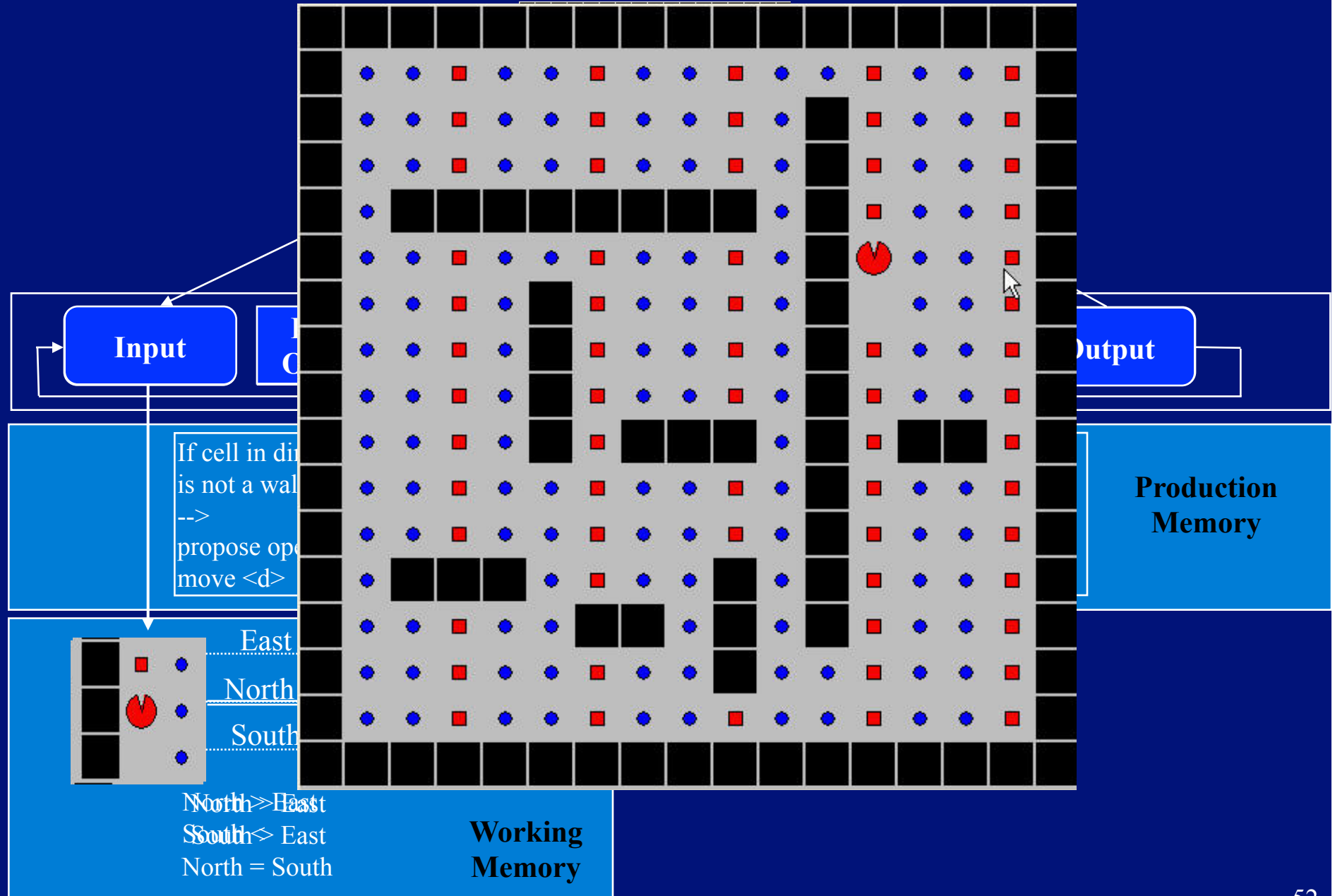
```
sp {water-jug*remove*last-operator
    (state <s> ^name water-jug
        ^last-operator <name>
        ^operator <o>)
    (<o> ^name <> <name>)
-->
(<s> ^last-operator <name> -)}
```

# Control Knowledge

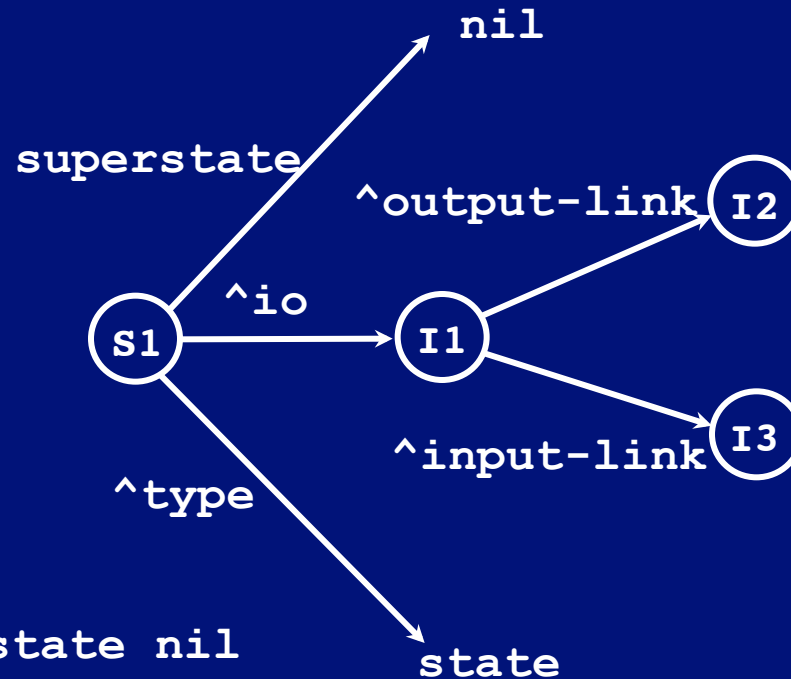
```
## If just applied fill, don't apply empty
sp {water-jug*select*fill*empty*worst
    (state <s> ^name water-jug
        ^last-operator fill
        ^operator <o> +)
    (<o> ^name empty)
-->
(<s> ^operator <o> < > ) }
```

```
## If just applied empty, don't apply fill
sp {water-jug*select*empty*fill*worst
    (state <s> ^name water-jug
        ^last-operator empty
        ^operator <o> +)
    (<o> ^name fill)
-->
(<s> ^operator <o> < > ) }
```

# Soar 103: Eaters



# Initial Working Memory



```
S1 ^superstate nil
```

```
S1 ^io I1
```

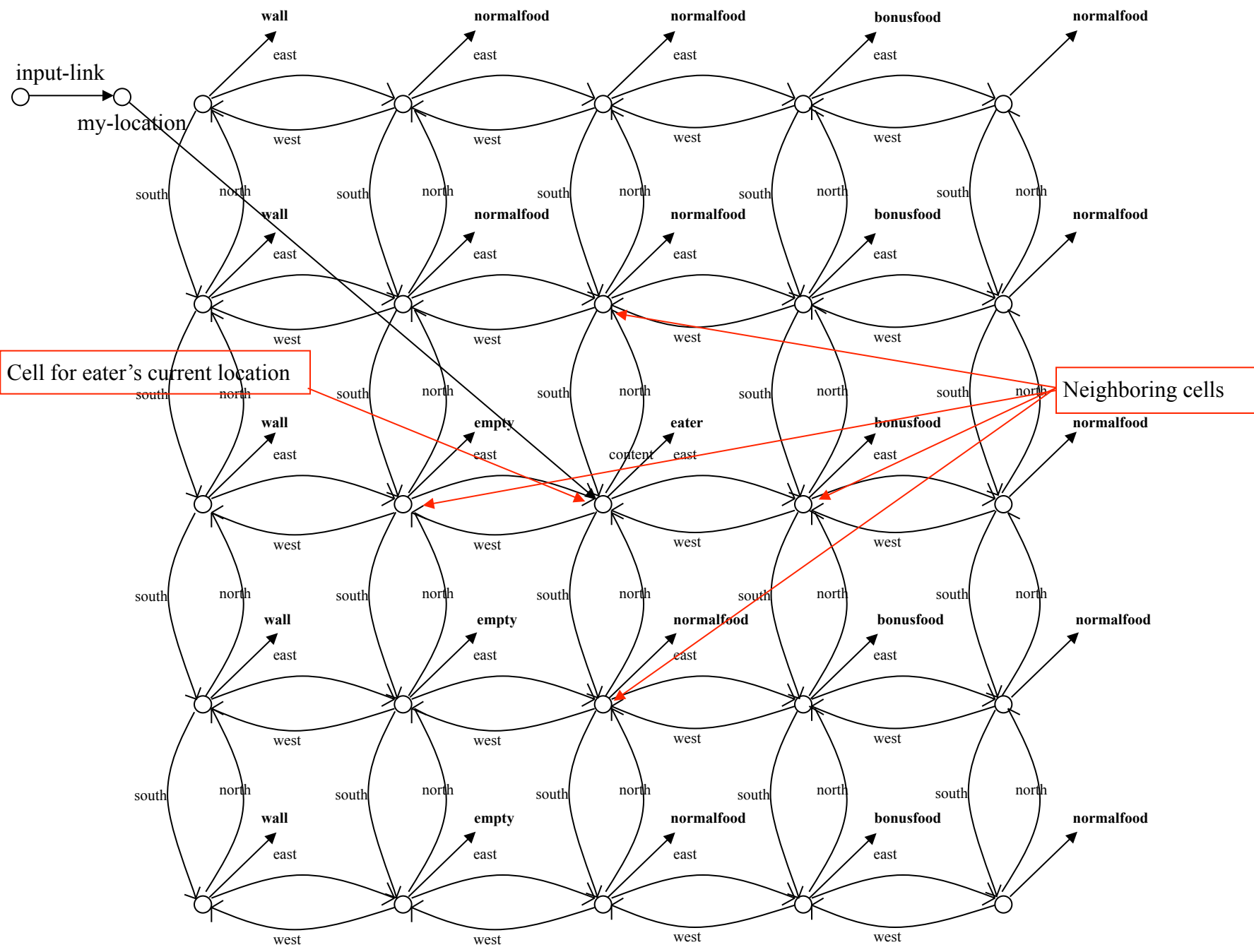
```
S1 ^type state
```

```
I1 ^output-link I2
```

```
I1 ^input-link I3
```

```
(S1 ^io I1 ^superstate nil ^type state)
```

```
(I1 ^input-link I3 ^output-link I2)
```



# Move to Food

```
# Propose*move
# If there is normalfood or bonusfood in an adjacent cell,
#   propose move in the direction of that cell
#   and indicate that this operator can be selected randomly.
#
```

# Initial Move Proposals

```
sp {propose*move*normalfood
  (state <s> ^io <io>)
  (<io> ^input-link <input-link>)
  (<input-link> ^my-location <my-loc>)
  (<my-loc> ^<dir> <cell>)
  (<cell> ^content normalfood)
-->
  (<s> ^operator <o> +)
  (<s> ^operator <o> =)
  (<o> ^name move
    ^direction <dir>)}

```

```
sp {propose*move*normalfood
  (state <s> ^io.input-link.my-location.<dir>.content normalfood)
-->
  (<s> ^operator <o> + =)
  (<o> ^name move
    ^direction <dir>)}

```



# Short Cut: << >>

```
sp {propose*move
    (state <s> ^io.input-link.my-location.<dir>.content
        << normalfood bonusfood >>)
-->
(<s> ^operator <o> + =)
(<o> ^name move
    ^direction <dir>)}

```

# General Move Operator

```
# Propose*move:
# If there is normalfood, bonusfood, eater, or empty in an adjacent cell,
#   propose move in the direction of that cell, with the cell's content,
#   and indicate that this operator can be selected randomly.

sp {propose*move
  (state <s> ^io.input-link.my-location.<dir>.content
    { <content> << empty normalfood bonusfood eater >> })
-->
(<s> ^operator <o> + =)
(<o> ^name move
  ^direction <dir>
  ^content <content>)}

```

```
sp {propose*move
  (state <s> ^io.input-link.my-location.<dir>.content
    { <content> <> wall })
-->
(<s> ^operator <o> + =)
(<o> ^name move
  ^direction <dir>
  ^content <content>)}

```

# Move apply

```
# Apply*move
# If the move operator for a direction is selected,
#   generate an output command to move in that direction.
#
sp {apply*move
    (state <s> ^io.output-link <out>
        ^operator <o>)
    (<o> ^name move
        ^direction <dir>)
-->
    (<out> ^move.direction <dir>)}

# Apply*move*remove-move:
# If the move operator is selected,
#   and there is a completed move command on the output link,
#   then remove that command.
sp {apply*move*remove-move
    (state <s> ^io.output-link <out>
        ^operator.name move)
    (<out> ^move <move>)
    (<move> ^status complete)
-->
    (<out> ^move <move> -)}
```

# Move Selection

```
sp {select*move*food-better-than-empty-eater
  (state <s> ^operator <o1> +
    ^operator <o2> +)
  (<o1> ^name move
    ^content << bonusfood normalfood >>)
  (<o2> ^name move
    ^content << empty eater >>)
```

```
-->
  (<s> ^operator <o1> > <o2>)} }
```

```
sp {select*move*prefer*bonusfood
  (state <s> ^operator <o1> +)
  (<o1> ^name move
    ^content bonusfood
```

```
-->
  (<s> ^operator <o1> >)} }
```

# Jump

- Eaters allows a jump action – move two spaces in a single direction, jumping over a cell, but costing 5 points.
- What would be a proposal for that?

```
sp {propose*jump
  (state <s> ^io.input-link.my-location.<dir>.<dir>.content
    <> wall)
-->
(<s> ^operator <o> + =)
(<o> ^name jump
  ^direction <dir>)
```

- How should we write control knowledge to select between moving and jumping to different objects?

# Jump/Move Selection

```
sp {init*elaborate*name-content-value
  (state <s> ^type state)
-->
  (<s> ^name-content-value <c1> <c2> <c3> <c4>
    <c5> <c6> <c7> <c8>)
  (<c1> ^name move ^content empty ^value 0)
  (<c2> ^name move ^content eater ^value 0)
  (<c3> ^name move ^content normalfood ^value 5)
  (<c4> ^name move ^content bonusfood ^value 10)
  (<c5> ^name jump ^content empty ^value -5)
  (<c6> ^name jump ^content eater ^value -5)
  (<c7> ^name jump ^content normalfood ^value 0)
  (<c8> ^name jump ^content bonusfood ^value 5)}
```

# Jump/Move Selection

```
sp {elaborate*operator*value
  (state <s> ^operator <o> +
    ^name-content-value <ccv>)
  (<o> ^name <name> ^content <content>)
  (<ccv> ^name <name> ^content <content> ^value <value>)
-->
  (<o> ^value <value>)}

```

```
sp {select*compare*best*value
  (state <s> ^operator <o1> +
    ^operator <o2> +)
  (<o1> ^value <v>)
  (<o2> ^value < <v>)
-->
  (<s> ^operator <o1> > <o2>)}

```

# TankSoar

Red Tank's  
Shield

Borders  
(stone)

Walls  
(trees)

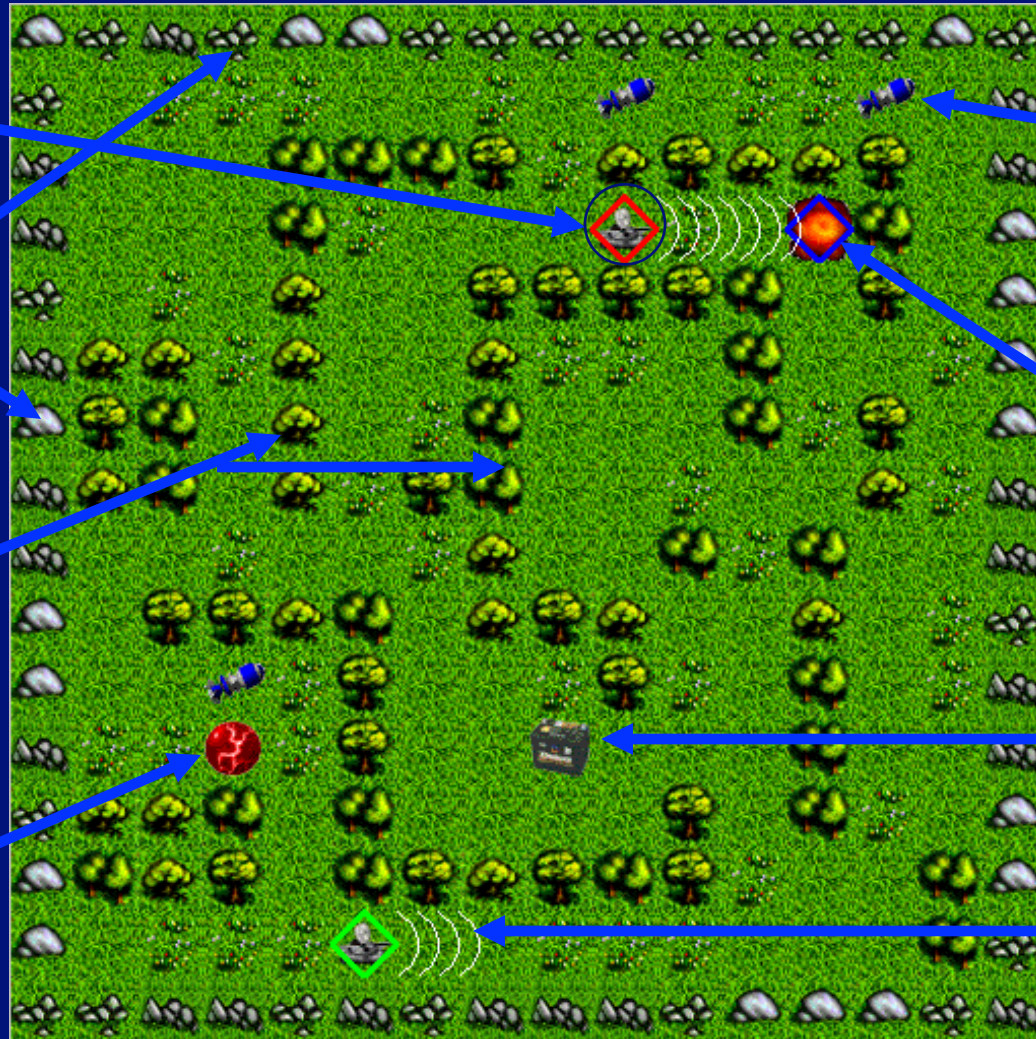
Health  
charger

Missile  
pack

Blue tank  
(Ouch!)

Energy  
charger

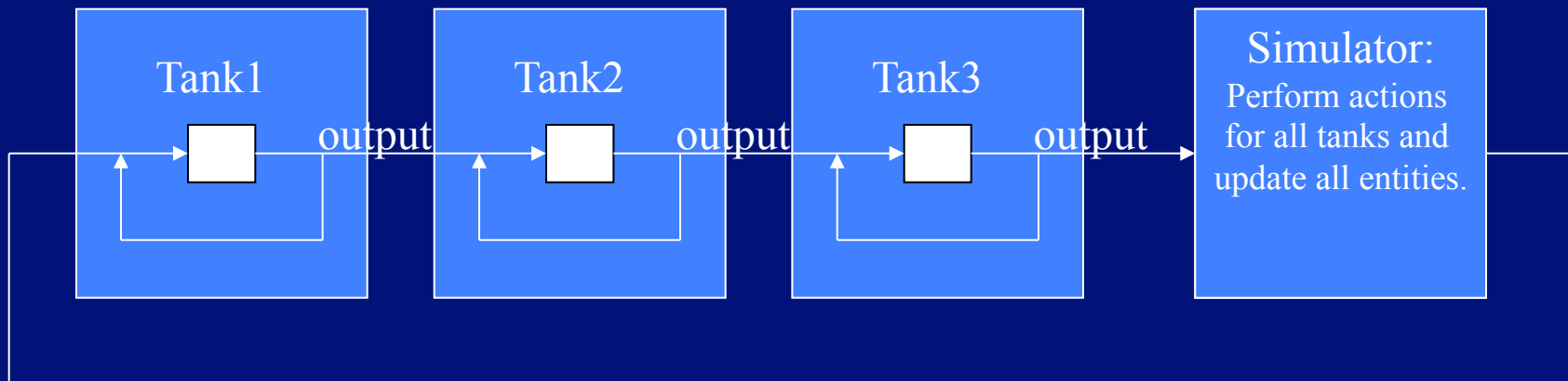
Green  
tank's radar





# TankSoar Scheduler

- Each tank runs until it has output
  - Possibly multiple decisions

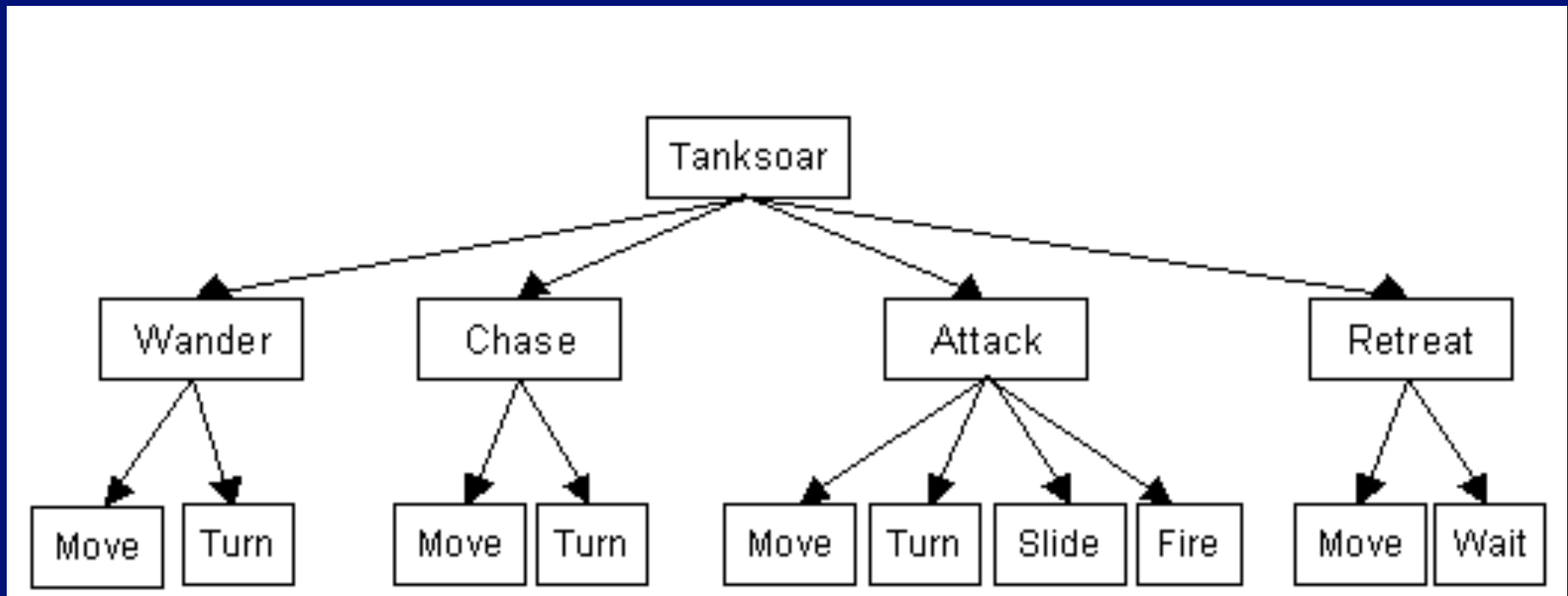


# Basic Operators for Wandering Tank

- If not blocked, move forward:
  - `^io.input-link.blocked.forward no`
  - `^move.direction forward`
- If blocked, rotate to clear direction and turn on radar to power 13
  - `^io.input-link.blocked.forward yes`
  - `^rotate.direction <direction>`
  - `^radar-power.setting 13`
- If radar is on and there are no objects, turn off radar
  - Can be an elaboration, not a separate operator
  - `^radar.switch off`

# TankSoar Hierarchy

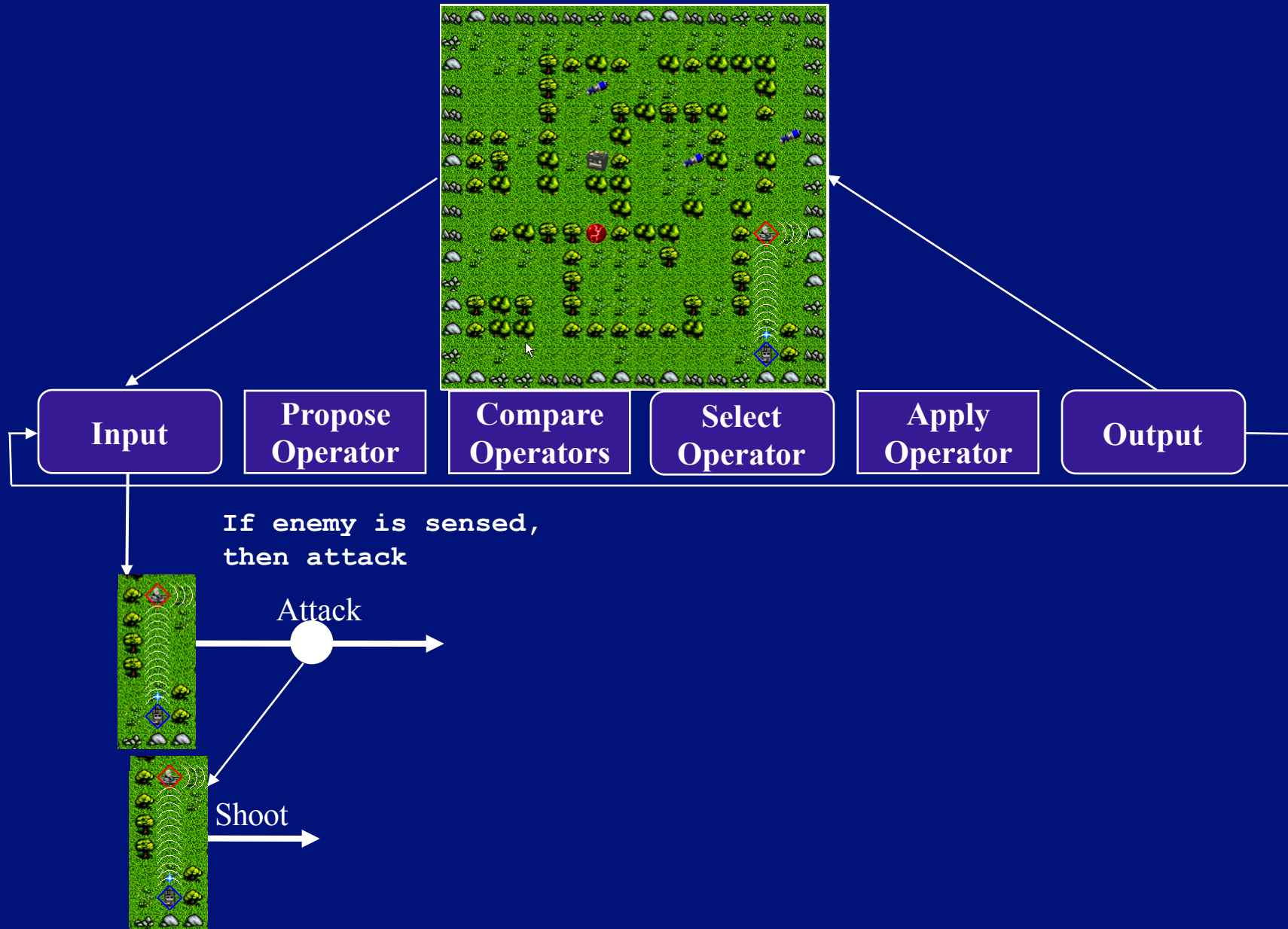
The Soar Tutorial's full Hierarchy for TankSoar:



# Soar 104: Substates

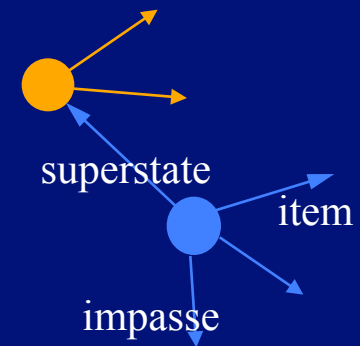


# Soar 104: Substates



# Impasses and Subgoals/Substates

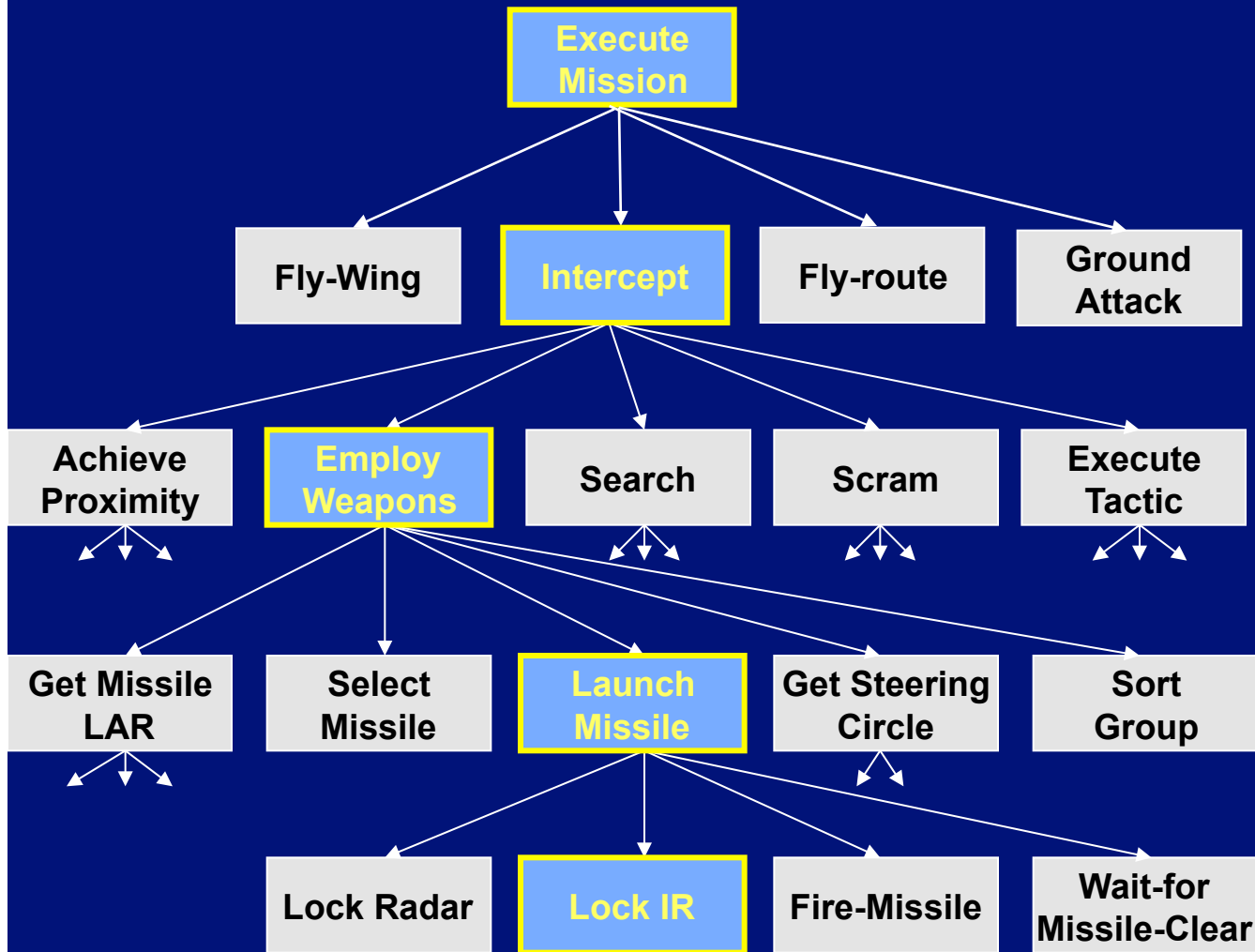
- Problem:
  - What to do when inconsistent or incomplete knowledge?
- Approach:
  - Detect impasses in decision procedure: tie, conflict, no-change
  - Create substate with augmentations that define impasse
    - Superstate
    - Impasse – no-change, tie, conflict, ...
    - Item – tied or conflicted operators
    - ...
  - Impasse resolved when decision can be made



# Implications:

- Substate is really meta-state that allows system to reflect
- Substate = goal to resolve impasse
  - Generate operator
  - Select operator (deliberate control)
  - Apply operator (task decomposition)
- All basic problem solving functions open to reflection
  - Operator creation, selection, application, state elaboration
- Substate is where knowledge to resolve impasse can be found
- Hierarchy of substate/subgoals arise through recursive impasses

# TacAir-Soar Task Decomposition



If instructed to intercept an enemy then propose intercept

If intercepting an enemy and the enemy is within range ROE are met then propose employ-weapons

If employing-weapons and missile has been selected and the enemy is in the steering circle and LAR has been achieved, then propose launch-missile

If launching a missile and it is an IR missile and there is currently no IR lock then propose lock-IR

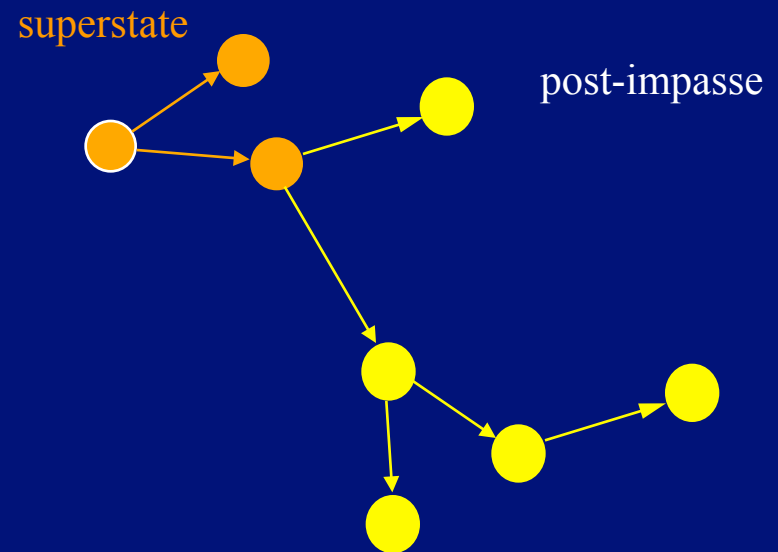
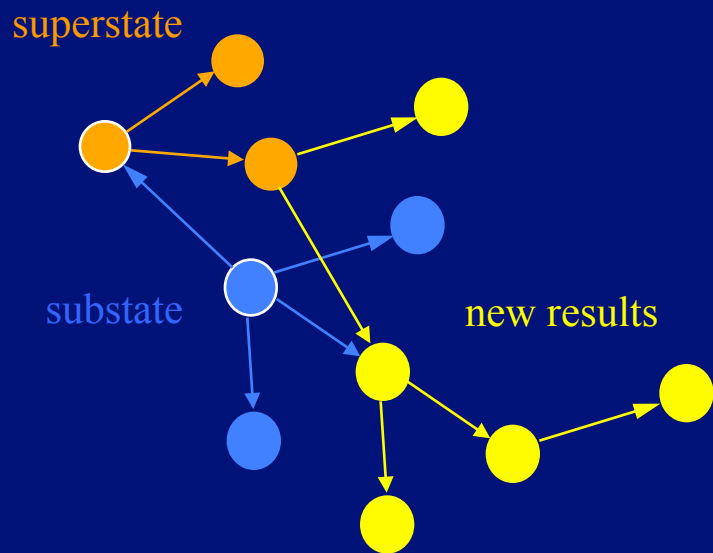
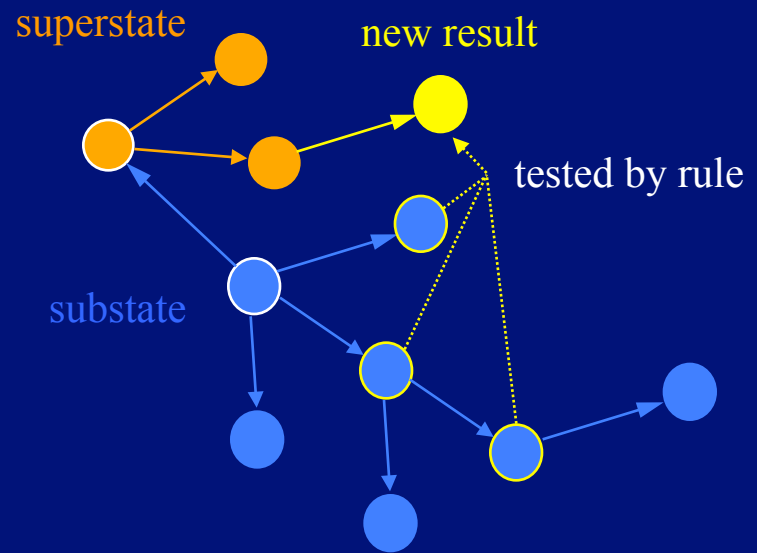
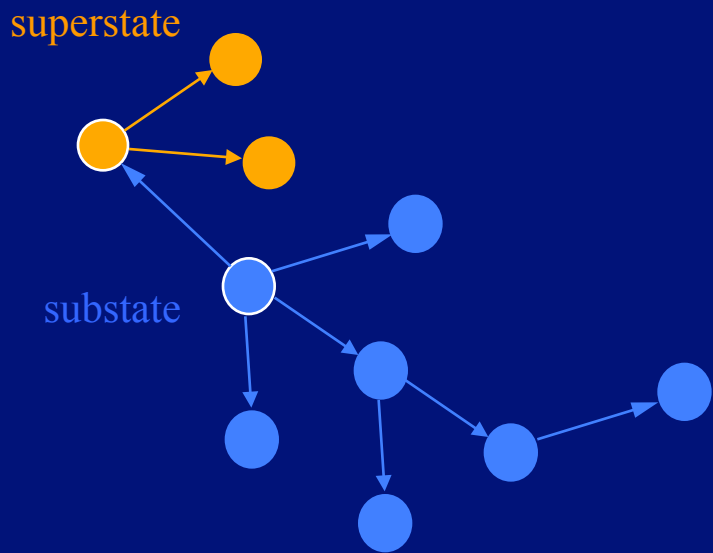
>250 goals, >600 operators, >8000 rules



# Substate Results

- Problem
  - What are the results of substates/subgoals?
  - Don't want to have programmer determine via special syntax
  - Results should be side-effect of processing
- Approach
  - Results determined by structure of working memory
  - Structure is maintained based on connectivity to state stack
  - Result is
    - Structure connected to superstate but created by rule that tests substate structure
    - Structure created in substate that becomes connected to superstate
  - Remove everything that isn't a result with impasse resolved
- Substate Approach Implications
  - Results do not always resolve impasses
  - One result can cause large substate structure to become result
  - Superstate cannot be augmented with substate – substate would be result

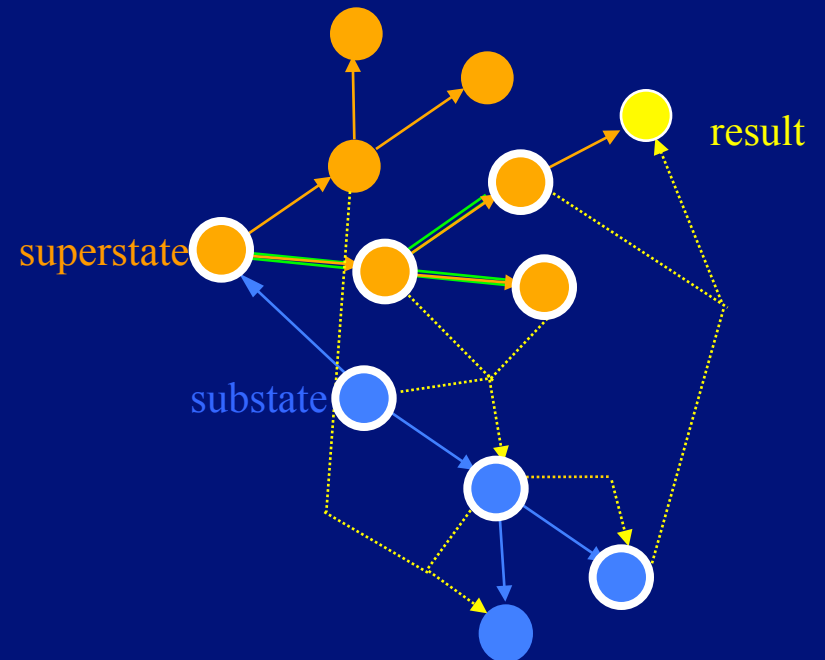
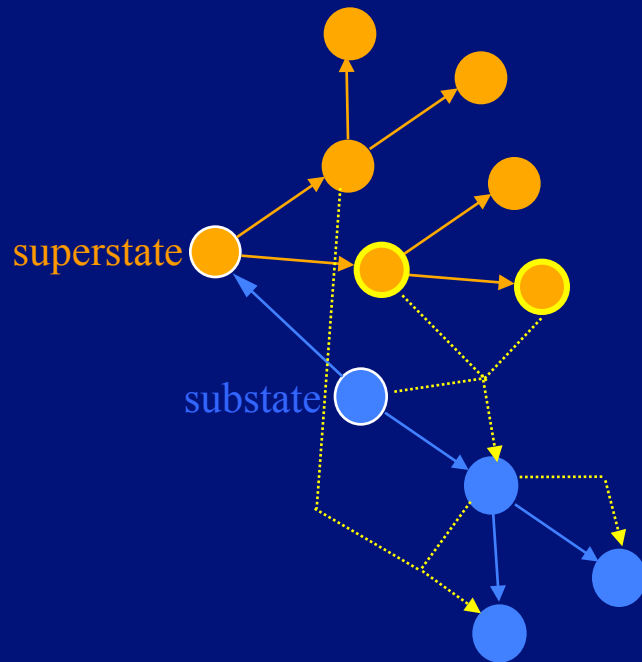
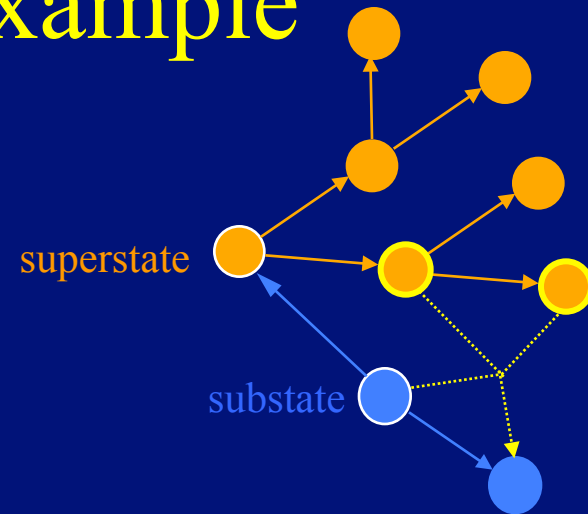
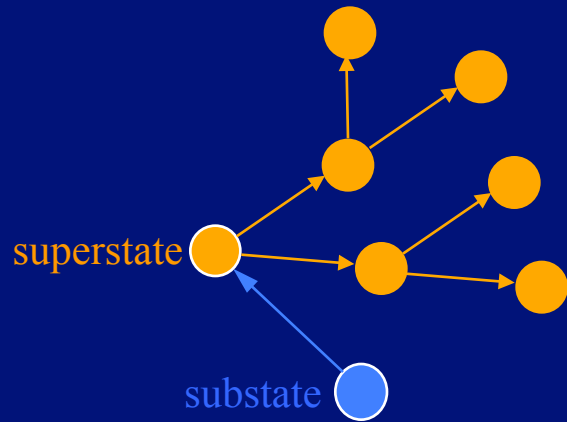
# Result Examples



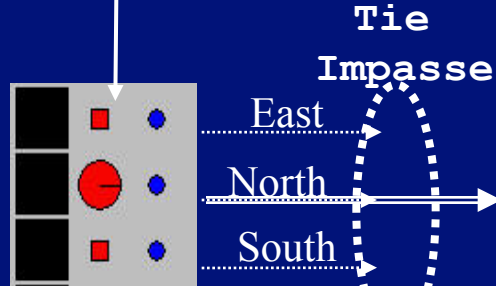
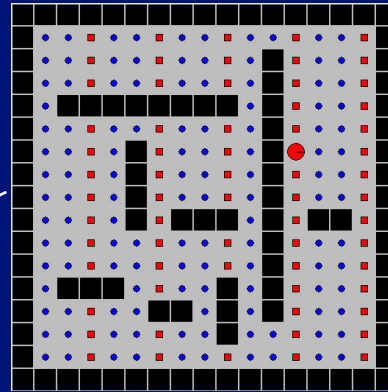
# Persistence of Results

- Problem:
  - What should be the persistence of results?
  - Based on persistence of structure in subgoal?
  - Could have different persistence before and after chunking
    - Operator in subgoal could create elaboration of superstate
  - How maintain i-support after substate removed?
- Approach:
  - Build justification that captures processing
  - Analyze justification
    - Elaborate, propose, select, apply
    - Assign o/i-support
  - Maintain justification for i-support until result removed

# Justification Example

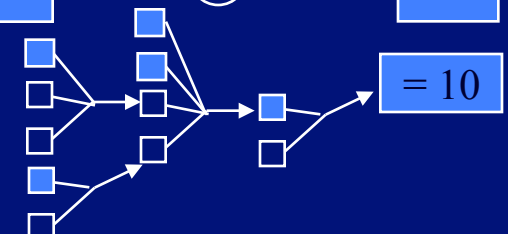
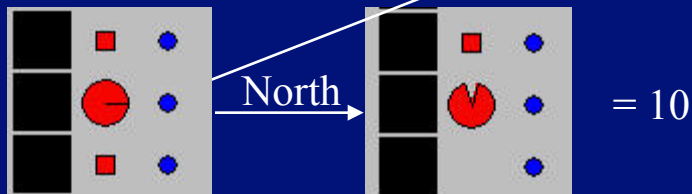
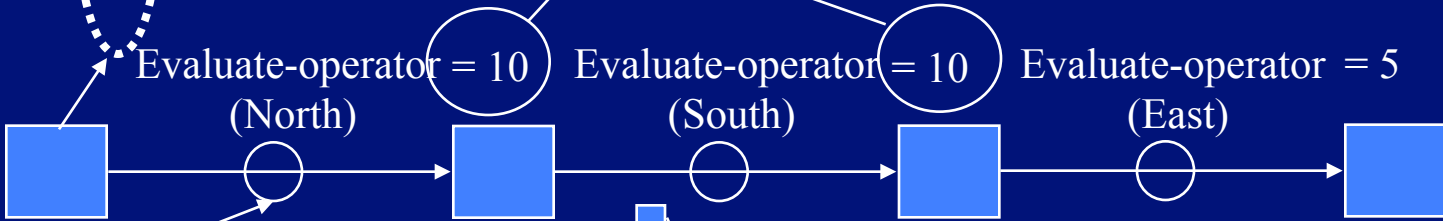


# Tie Subgoals and Chunking



North > East  
 South > East  
 North = South

Chunking creates rules that create preferences based on what was tested

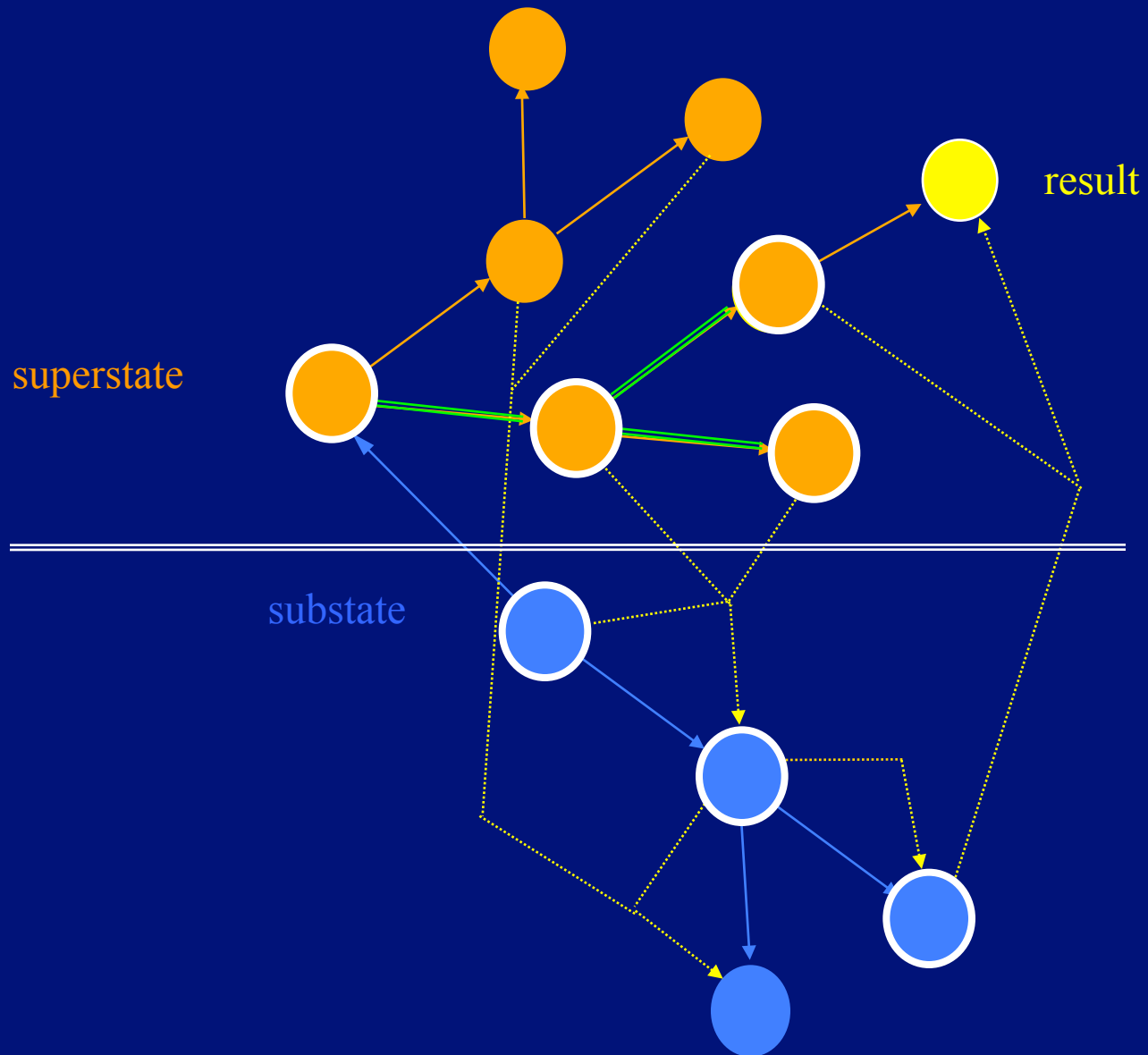


Chunking creates rule that applies evaluate-operator

# Learning/Chunking

- Problem:
  - Subgoals “discover” knowledge to resolve impasses but it is lost after each problem solving episode
- Approach
  - Automatically build rules that summarize processing
    - Variablize justifications = chunks
    - Variablizes identifiers – not constants: loses  $>$ ,  $<$ , ... tests between constants
    - Conditions include those tests required to produce result = implicit generalization
  - Chunks are built as soon as a result is produced
    - Immediate transfer is possible
  - One chunk for each result, where a result consists of connected WMEs that become results at the same time
    - Different results can lead to very different conditions
    - Improves generality of chunks
  - Only chunk over high-confidence decisions

# Chunk Example

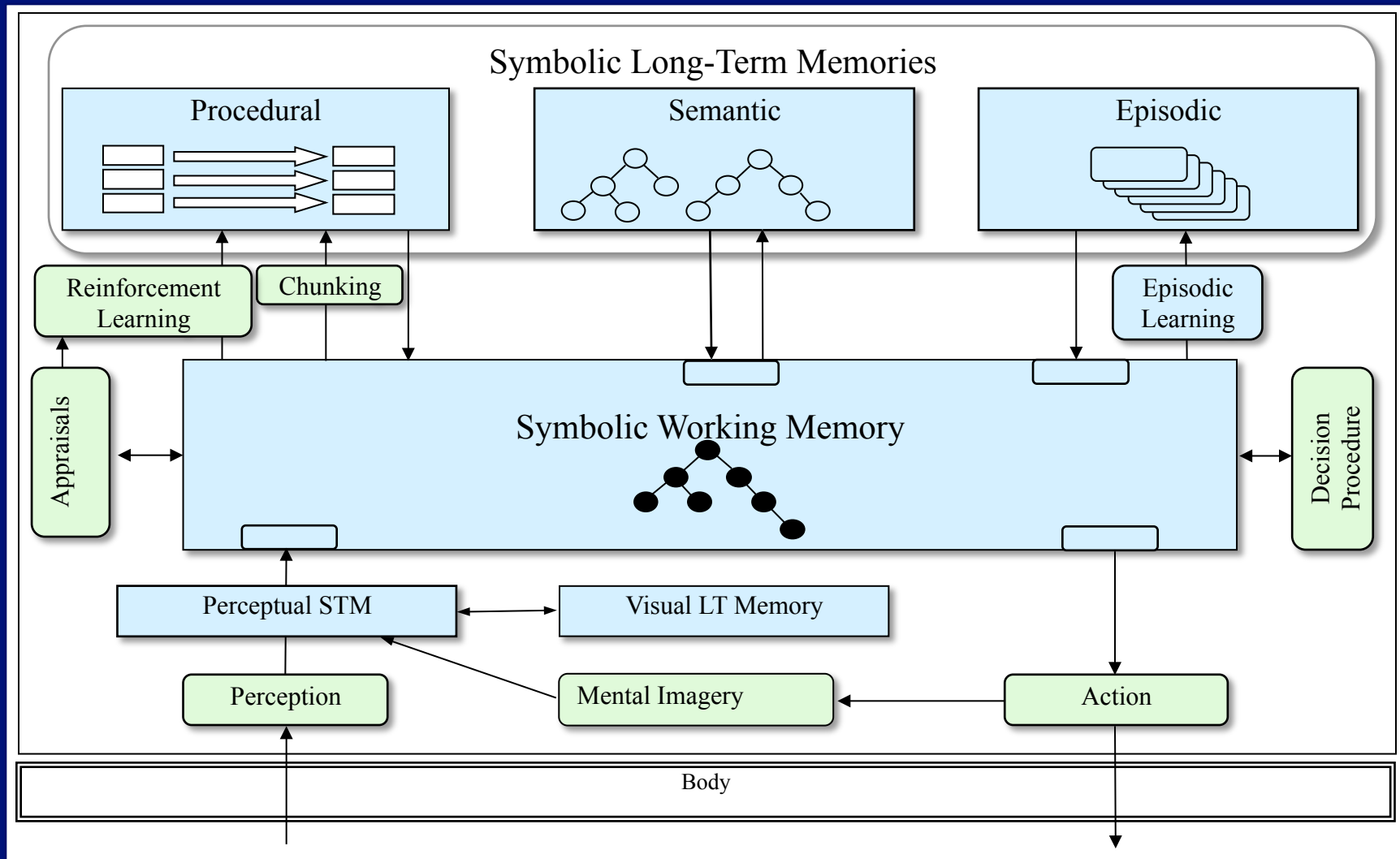


# Chunking Analysis

- Converts deliberate reasoning/planning to reaction
- Generality of learning based on generality of reasoning
  - Leads to many different types learning
  - If reasoning is inductive, so is learning
- Soar only learns what it thinks about
- All learning is impasse driven
  - Learning arises from a lack of knowledge



# Soar 9 Structural Diagram

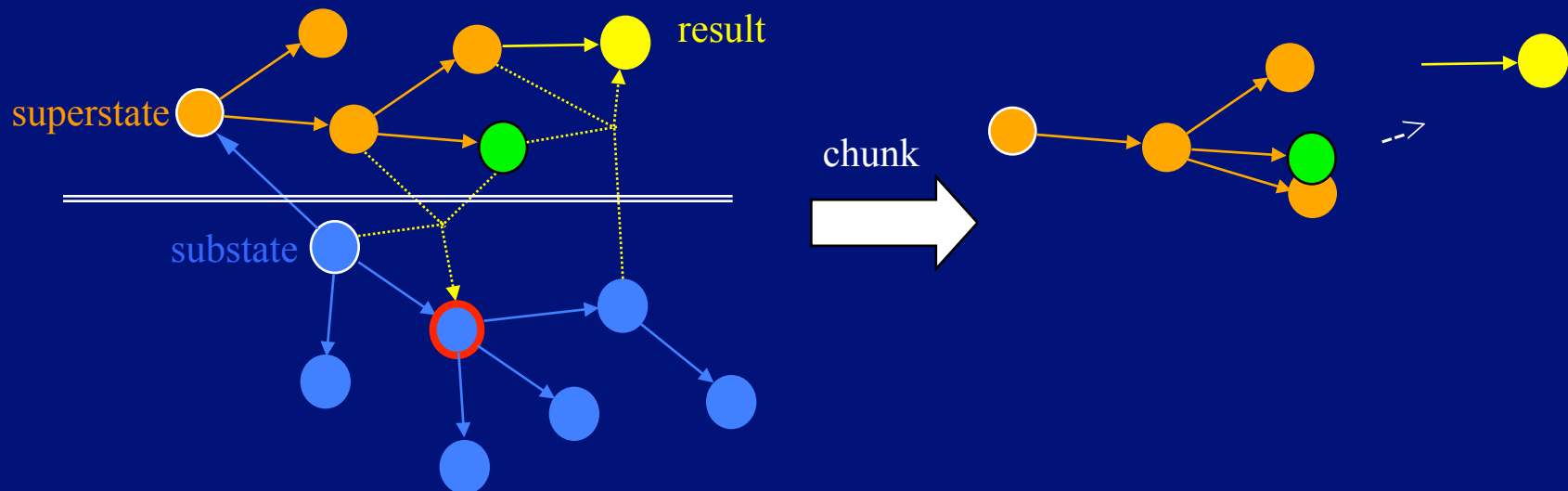


# Soar Community

- Soar Website
  - <http://sitemaker.umich.edu/soar>
- Soar-group
  - <http://lists.sourceforge.net/lists/listinfo/soar-group>
  - Low traffic

# Persistence of Substate Structures: Problem

- O-supported structure in subgoals can become *inconsistent*
  - Future behavior is no longer reactive to changes in the context
    - Non-reentrant – results would be different if reentered subgoal
  - Chunks have conditions that can never match
    - Test mutually exclusive values of same attribute
    - Non-contemporaneous



# Analysis

- Whenever the substate WMEs cannot be recreated from superstate WMEs using existing rules.
- Occurs from changes to input and returning results.
- Only a problem for o-supported structures and their entailments
  - Not a problem for i-supported structures

# Possible Approach

- Remove any substate WME that becomes inconsistent
  - One detail of Soar makes this very nasty
    - WMEs don't "blip" when there is a change in i-support
    - If an i-supported WME loses support, but at exact same time, same WME is created with new i-support, WME doesn't change

`(<s> ^sensor-a < 20) --> (<s> ^enemy near)`

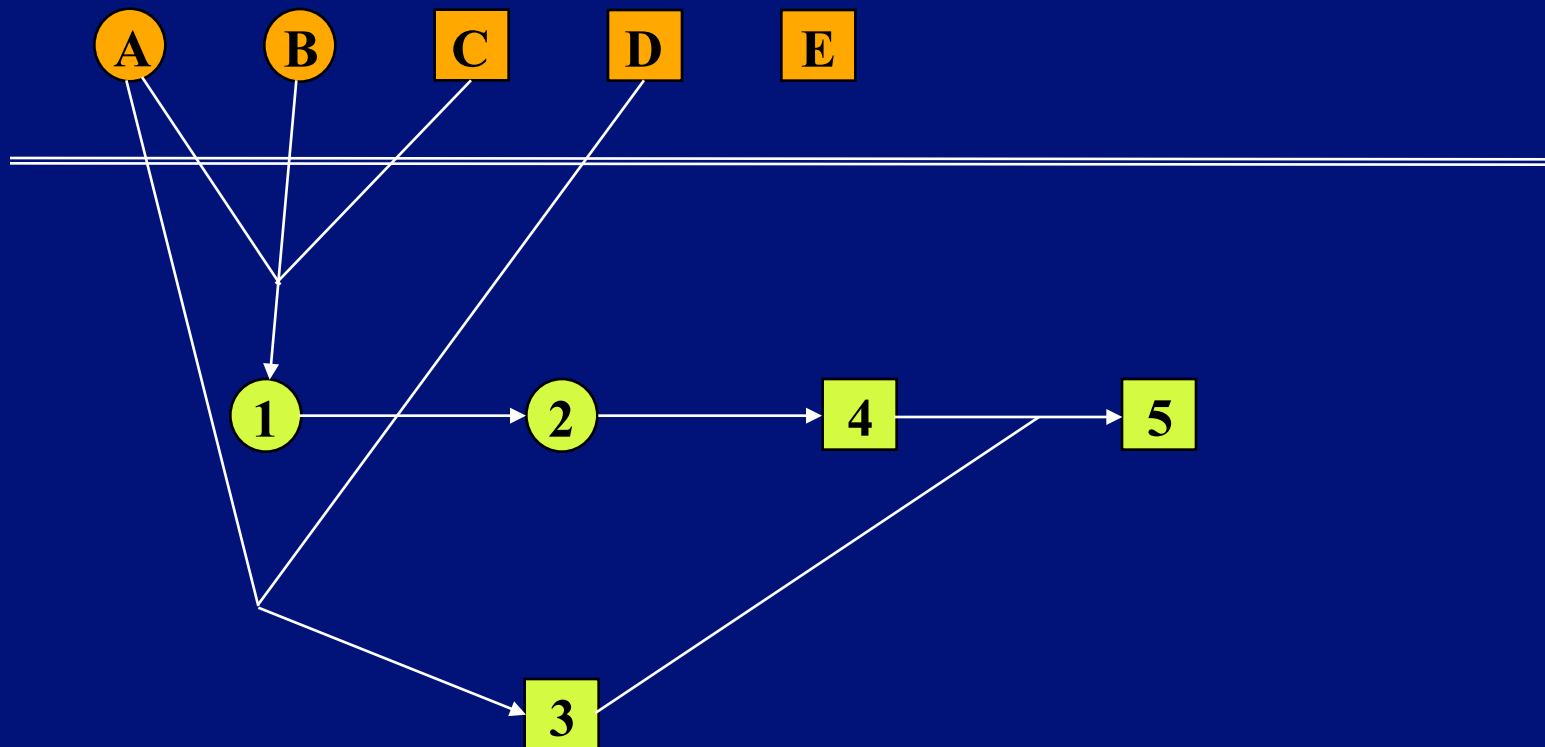
- Can't maintain derivation information with every WME
  - Because it can change
  - Must dynamically compute derivation information
- Very expensive to maintain and compute

# Approach

- A substate is regenerated whenever higher state WMEs become *inconsistent* with substate's internal processing
- Regenerated = all substate structure removed from WM and new substate created.
- Each substate maintains a *goal dependency set (GDS)*
  - All superstate WMEs tested in creating o-supported WMEs in substate
- If anything changes in GDS, substate is regenerated.


# GDS Example

superstate



substate

GDS= []    GDS= [A,D]    GDS= [A,B,C,D]    GDS= [A,B,C,D]

 = i-support

 = o-support

# Implications

- Only an issue for o-supported structures in substates.
- Can't create o-supported structures based on changing sensors.
  - Can't create counters of external events in substates
- O-supported structures in substates are steps in that problem space.
  - Look-ahead search
- Can avoid regeneration by maintaining “fragile” o-support structure on top-state.