Beyond Simple Decision Making:

Metacognition, Hierarchical Subgoals, and Planning.

Impasses and Substates in Soar

Soar Tutorial

May 6-7, 2019

Simple Decision Making

- A single state with operators gives only "flat" reasoning.
  - No subgoals for task decomposition
  - Only a single problem space
  - No planning
    - No simulation of external actions
  - No reasoning about reasoning (metacognition)
  - No reasoning about other agents
    - No simulation of other entities

Hierarchical Reasoning in Rosie

- In learning Tower of Hanoi:
  - Process language to create semantic representation of task
  - Construct task representation
- In executing Tower of Hanoi:
  - Try to determine which actions are possible
    - Interpret task representation in working memory
  - Try to decide which disk to move
    - Look-ahead search
  - Attempt to move a disk
    - Pick up disk
    - Put down disk
- These require *multiple problem spaces and goals*
  - Require state information in addition to the task
  - Require addition internal actions in addition to task actions

Hypotheses in Soar

- Metacognition arises from insufficient or conflicting knowledge
  - If have sufficient knowledge, *just do it*.
  - "Effort is felt only where there is a conflict of interests in the mind."

    ...

    "The stream of our thought is like a river. On the whole easy simple flowing predominates in it, the drift of things is with the pull of gravity, and effortless attention is the rule. But at intervals an obstruction, a set-back, a log-jam occurs, stops the current, creates an eddy, and makes things temporarily move the other way."

    -William James, 1890, The Principles of Psychology

- Metacognition involves *"stepping back"* and have a separate state from which to reason (without disrupting original reasoning)

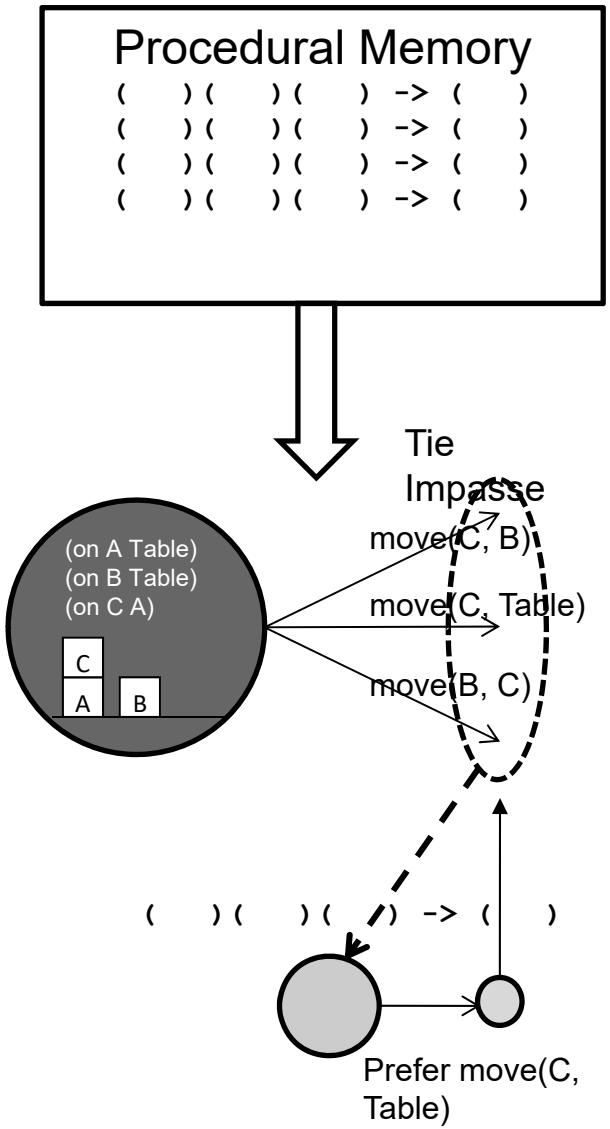
- Learning *compiles* metacognition into direct knowledge for future situations.

Processing Overview: Three Levels

- System 0: Architecture retrieves relevant knowledge
  - Searches memories to find knowledge using fixed algorithms.
    - Use meta-data to optimize retrievals (i.e., recency and frequency)
  - Task independent: doesn't change with task learning.
  - Operation is not open to introspection.
  - Doesn't compete with other task processing.

- System 1: Reactive decision making
  - Uses retrieved knowledge to guide behavior.
  - Learning can directly change and improve behavior.
  - Limited introspection, but limited access to meta-data.

- System 2: Subgoal processing
  - Arises if impasse in decision making: no choice, multiple choices.
  - Compositional processing: multiple operators, memory retrieval, mental imagery, planning, …
  - Creates results that resolve impasse.
  - Chunking compiles subgoal processing into rules:
    - converting System 2 to System 1 processing.



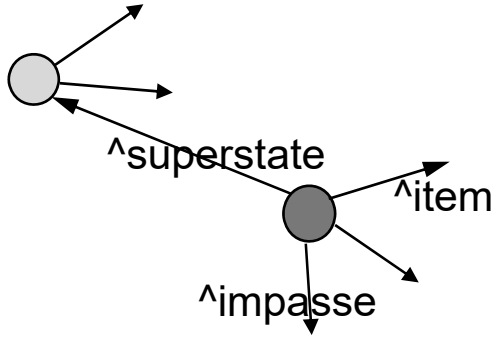
In Soar: Impasses Lead to Substates

An *impasse* arises if there is insufficient/conflicting procedural knowledge to select an operator

- when no operator is proposed.
  - **[state no-change]**
- when multiple operators are proposed by insufficient preferences to select between them?
  - **[tie]**
- when an operator is selected, but it can't be applied by a single rule?
  - **[operator no-change]**

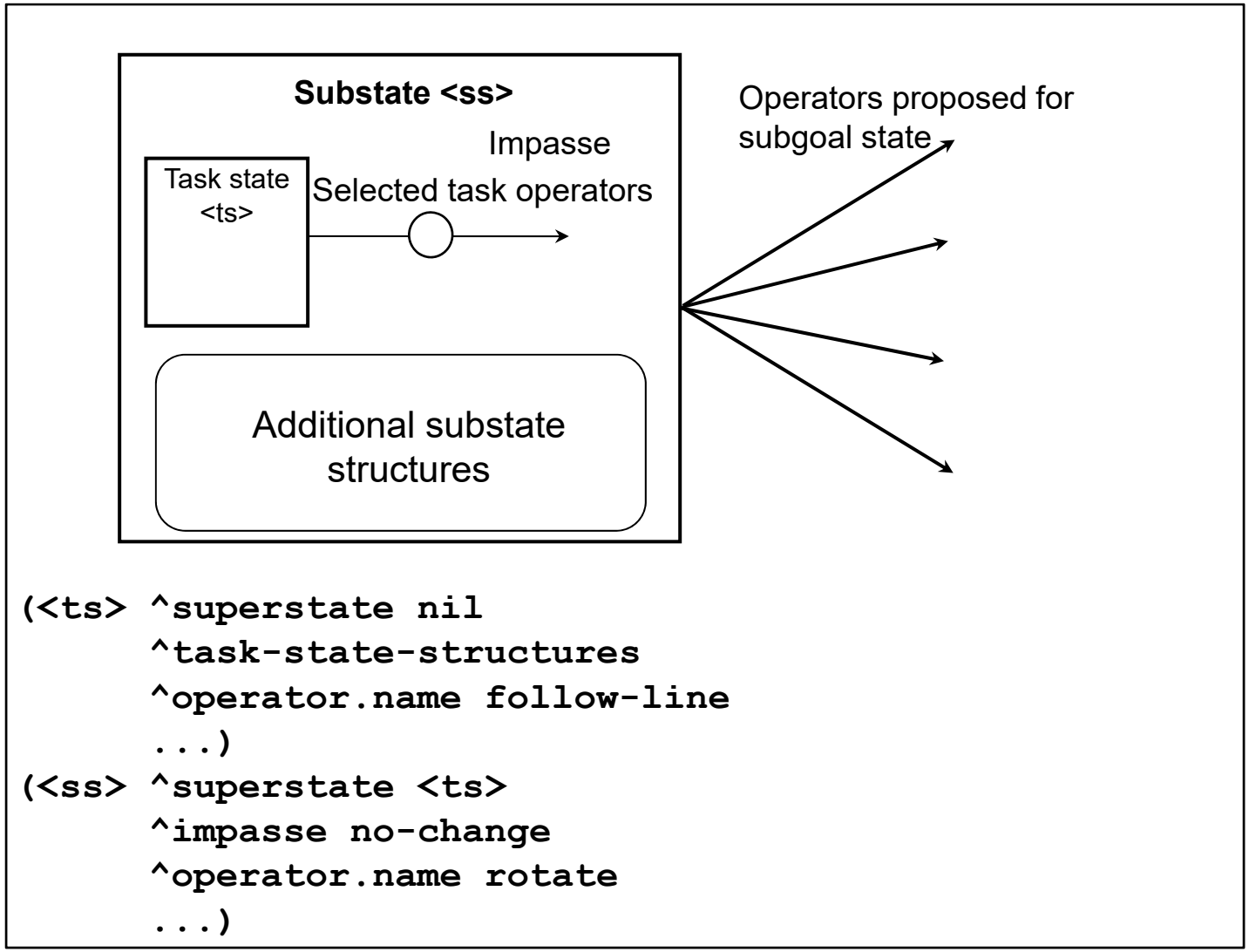
Substates

- Substate is created whenever there is an impasse
- Substate has augmentations that define impasse
  - **^superstate**
  - **^impasse – no-change, tie, conflict, …**
  - **^item – tied or conflicted operators**
  - …
- In a substate: Recursively select and apply operators
  - Access superstate information through
  - **(<s> ^superstate <ss>)**



- Substate is context for deliberate reasoning and accessing additional knowledge sources to resolve the impasse
  - Long-term memories
  - External environment
  - Internal reasoning (planning)
- Substate results:
  - Structures created in substate but linked to the superstate.
- Impasse is resolved when results lead to a decision
- Hierarchy of substates arise through recursive impasses

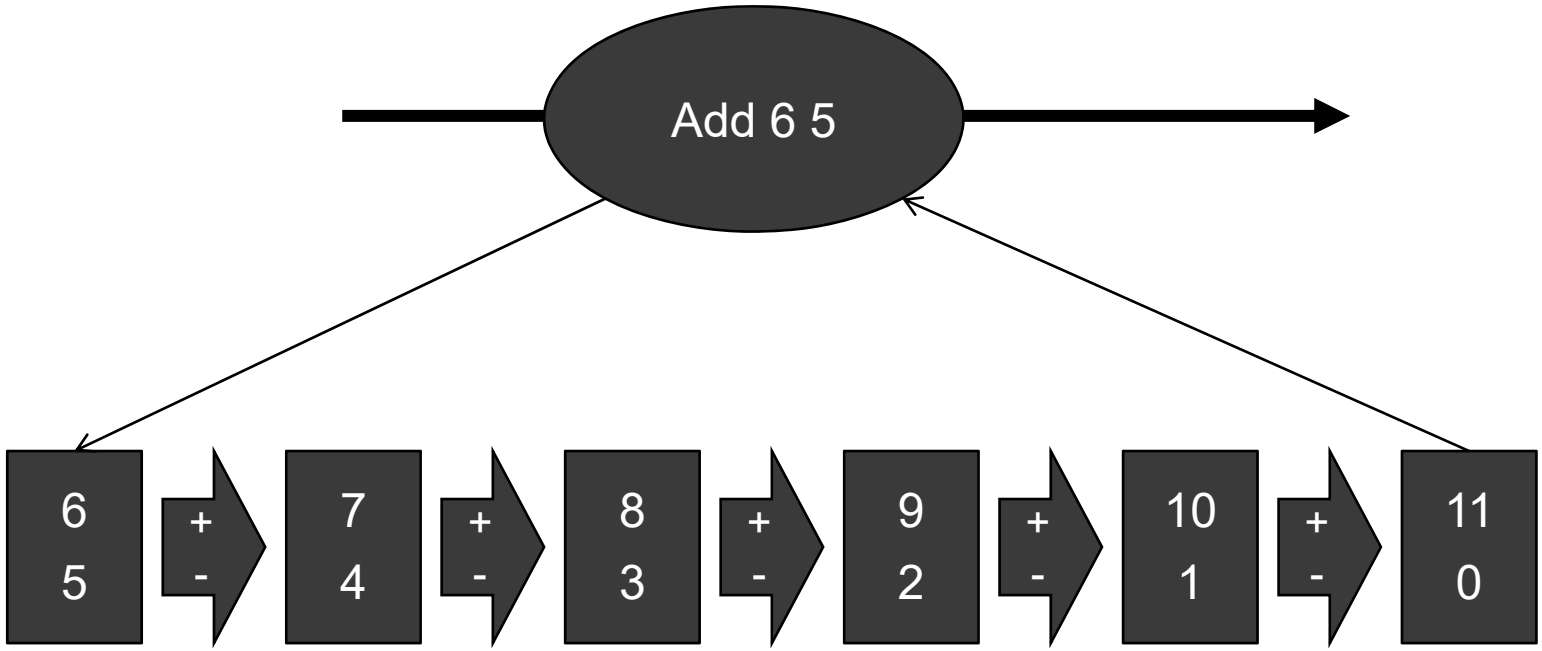
Impasses and Substates




```
(<ts> ^superstate nil
      ^task-state-structures
      ^operator.name follow-line
      ...)
(<ss> ^superstate <ts>
      ^impasse no-change
      ^operator.name rotate
      ...)
```

Operator Implementation

- Only operators count up and down.
- Add two numbers by counting up and down.



Show Running in Debugger

Top State Initialization

```
sp {propose*init-compute-sums
   (state <s> ^superstate nil)
  -(<s> ^name compute-sums)
   -->
   (<s> ^operator <o> +)
   (<o> ^name init-compute-sums)}

sp {apply*init-compute-sums
   (state <s> ^operator.name init-compute-sums)
-->
   (<s> ^name compute-sums
        ^add-pair <ap1> <ap2> <ap3>)
   (<ap1> ^adden1 6 ^adden2 5)
   (<ap2> ^adden1 3 ^adden2 3)
   (<ap3> ^adden1 7 ^adden2 4)}
```

Add Operator

```
# If an add-pair does not have a sum,
#  propose adding that add-pair

sp {propose*add
   (state <s> ^superstate nil
              ^add-pair <ap>)
  -(<ap> ^sum)
   -->
   (<s> ^operator <o> + =)
   (<o> ^name add
        ^add-pair <ap>)}
```

State Elaborations in Substate

```
sp {elaborate*add*add
   (state <s> ^superstate.operator <o>)
   (<o> ^name add)
   -->
   (<s> ^name substate)}

sp {elaborate*add*add-pair
   (state <s> ^superstate.operator <o>)
   (<o> ^name add
        ^add-pair <ap>)
   (<ap> ^adden1 <a1> ^adden2 <a2>)
   -->
   (write (crlf) <a1> | + | <a2> | = ?|)
   (<s> ^add-pair <ap>)}
```

Top State and Substate

```
(s1 ^superstate nil
    ^type state
    ^add-pair a1 ...
    ^operator o1)
(o1 ^name add
    ^add-pair a1)
(a1 ^adden1 6 ^adden2 5)

(s2 ^attribute operator
    ^impasse no-change
    ^superstate s1
    ^name add
    ^add-pair a1
...)
```

Substate operator

```
sp {propose*count-up-count-down
   (state <s> ^name add
              ^add-pair <ap>)
   (<ap> ^adden1 <count1>
         ^adden2 <count2>)
   -->
   (<s> ^operator <o> +)
   (<o> ^name count-up-count-down)}

sp {apply*count-up-count-down
   (state <s> ^operator.name count-up-count-down
              ^add-pair <ap>)
   (<ap> ^adden1 <count1> ^adden2 <count2>)
   -->
   (<ap> ^adden1 (+ <count1> 1)
                 <count1> -
         ^adden2 (- <count2> 1)
                 <count2> -)}
```

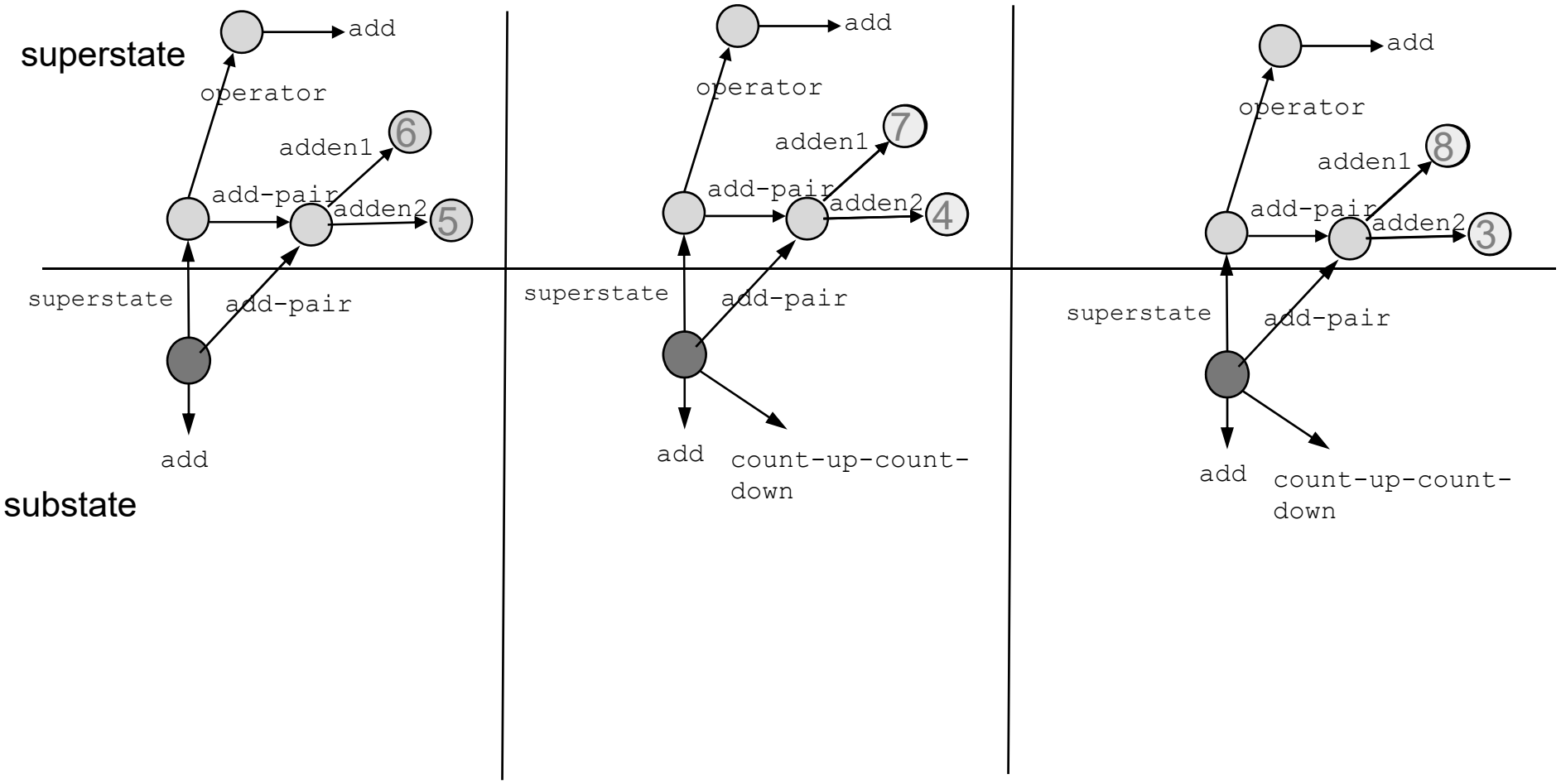
Elaboration Terminates Substate

```
sp {elaborate*add-pair*sum
   (state <s> ^add-pair <ap>)
   (<ap> ^adden1 <a1>
         ^adden2 0)
   -->
   (write (crlf) |Sum = | <a1>)
   (<ap> ^sum <a1>)}
```

Terminate Agent

```
#If there are no add-pairs that don't have
# sums, then halt.

sp {compute-sums*finished
    (state <s> ^name compute-sums)
   -{(<s> ^add-pair <ap>)
     (<ap> -^sum)}
    -->
    (write (crlf) |Finished.|)
    (halt)
}
```

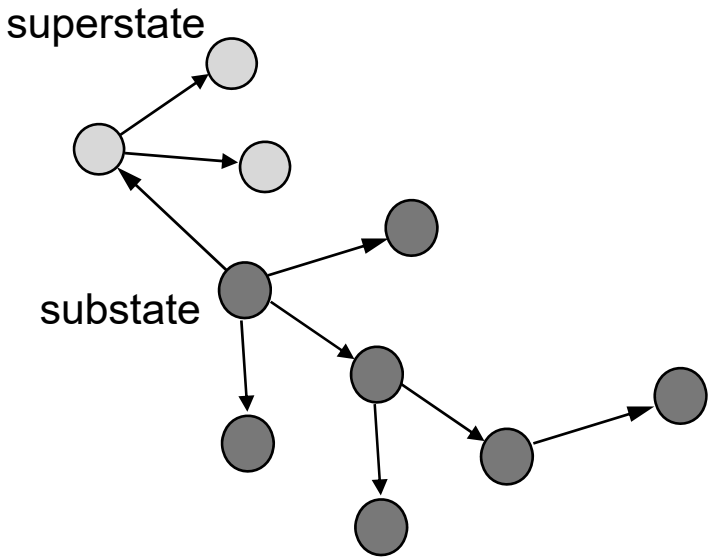
Problem Solving in Substate

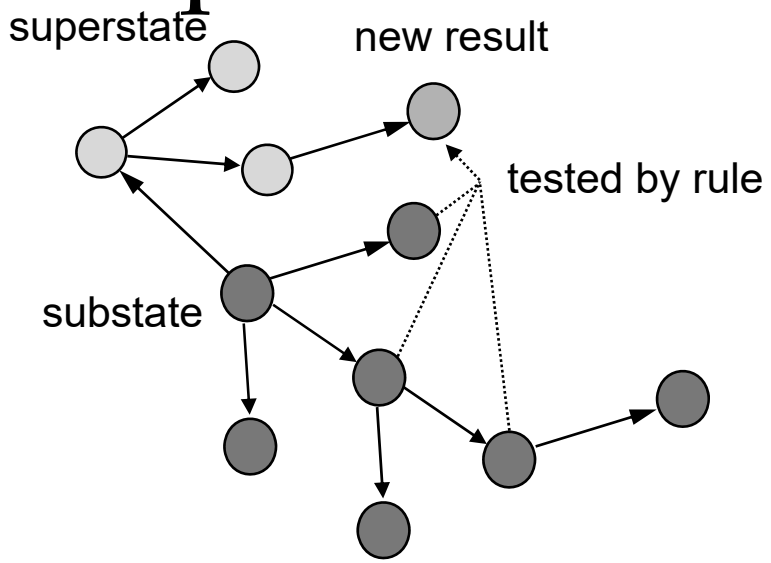


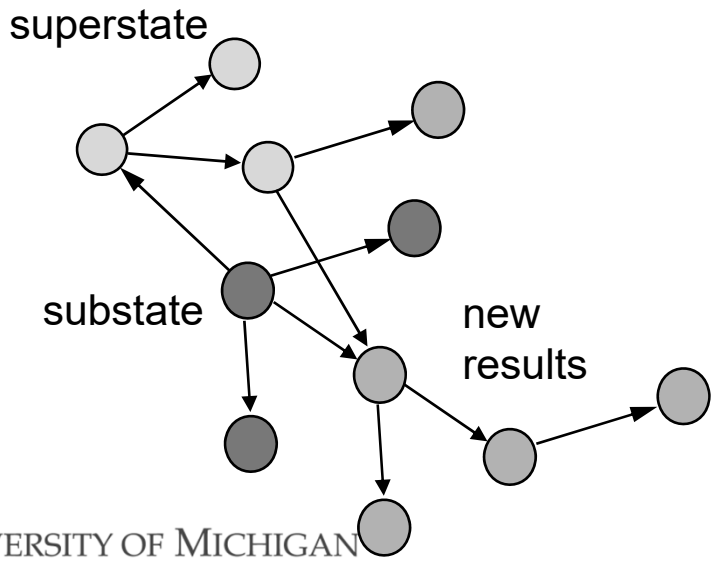
Substate Results

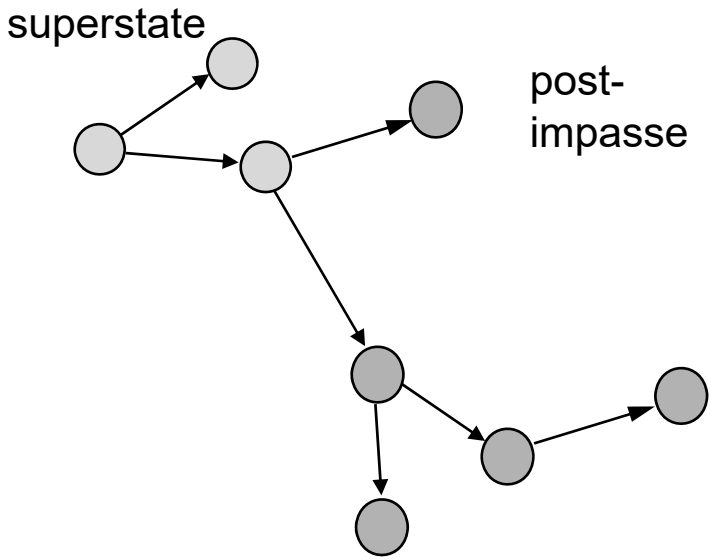
- Problem
  - What are the results of substates/subgoals?
  - Don't want to have programmer determine via special syntax
  - Results should be side-effect of processing
- Approach
  - Results determined by structure of working memory
  - Structure is maintained based on connectivity to state stack
  - 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









What if copy add-pair structure to substate and only return final sum?

- Show Running in Debugger

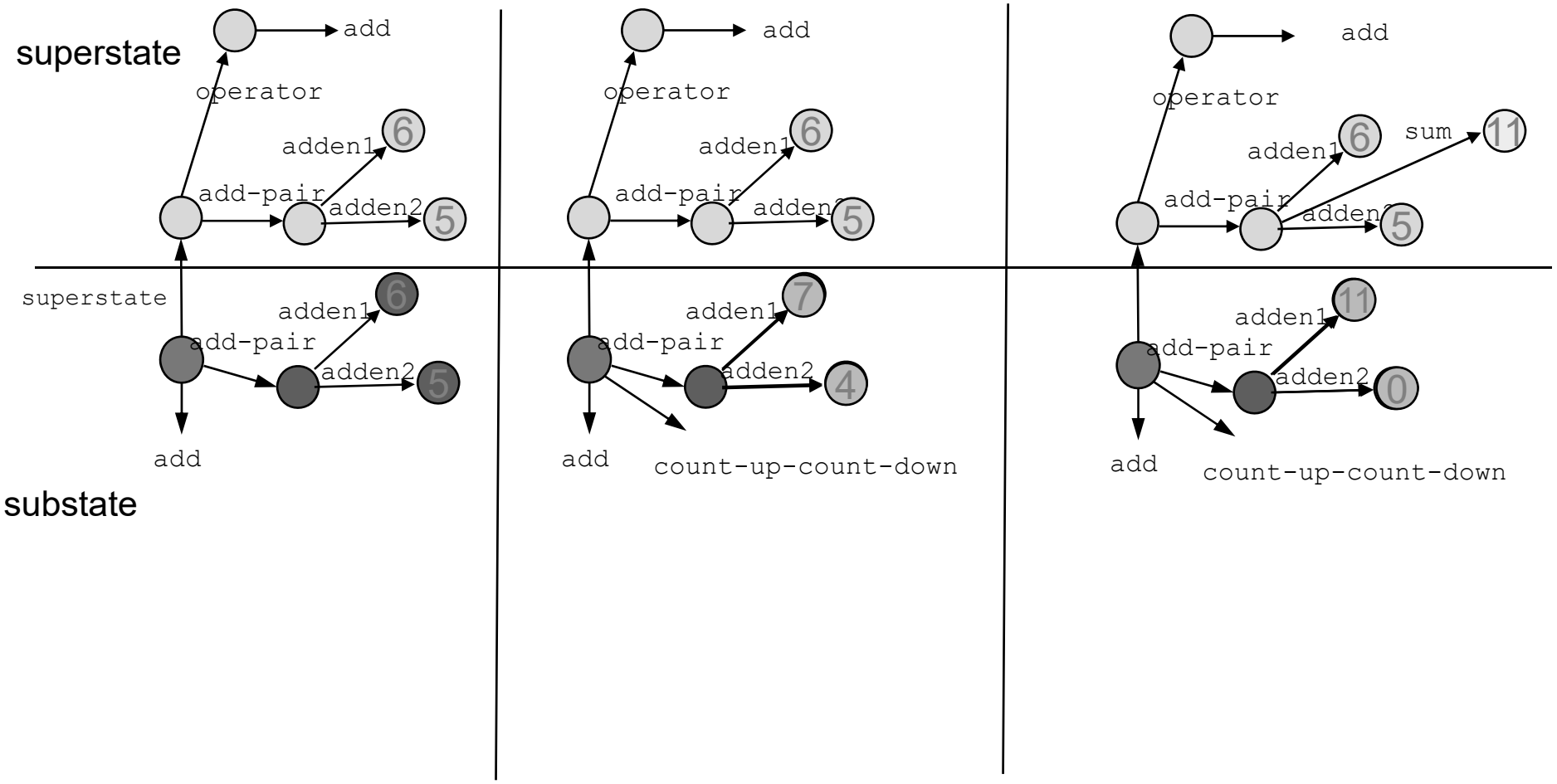
State Elaboration - copy

```
sp {elaborate*add*add
   (state <s> ^superstate.operator <o>)
   (<o> ^name add)
   -->
   (<s> ^name add)}

sp {elaborate*add*add-pair
   (state <s> ^superstate.operator <o>)
   (<o> ^name add
        ^add-pair <ap>)
   (<ap> ^adden1 <a1>
         ^adden2 <a2>)
   -->
  (write (crlf) <a1> | + | <a2> | = ?|)
  (<s> ^add-pair <apx>)
  (<apx> ^adden1 <a1>
         ^adden2 <a2>)}
```

Substate Result

```
sp {return*add-pair*sum
    (state <s> ^name add
               ^add-pair <apx>
               ^superstate.operator <o>)
    (<o> ^name add
         ^add-pair <ap>)
    (<apx> ^adden1 <a1>
           ^adden2 0)
    -->
    (<ap> ^sum <a1>)}
```

Problem Solving in Substate



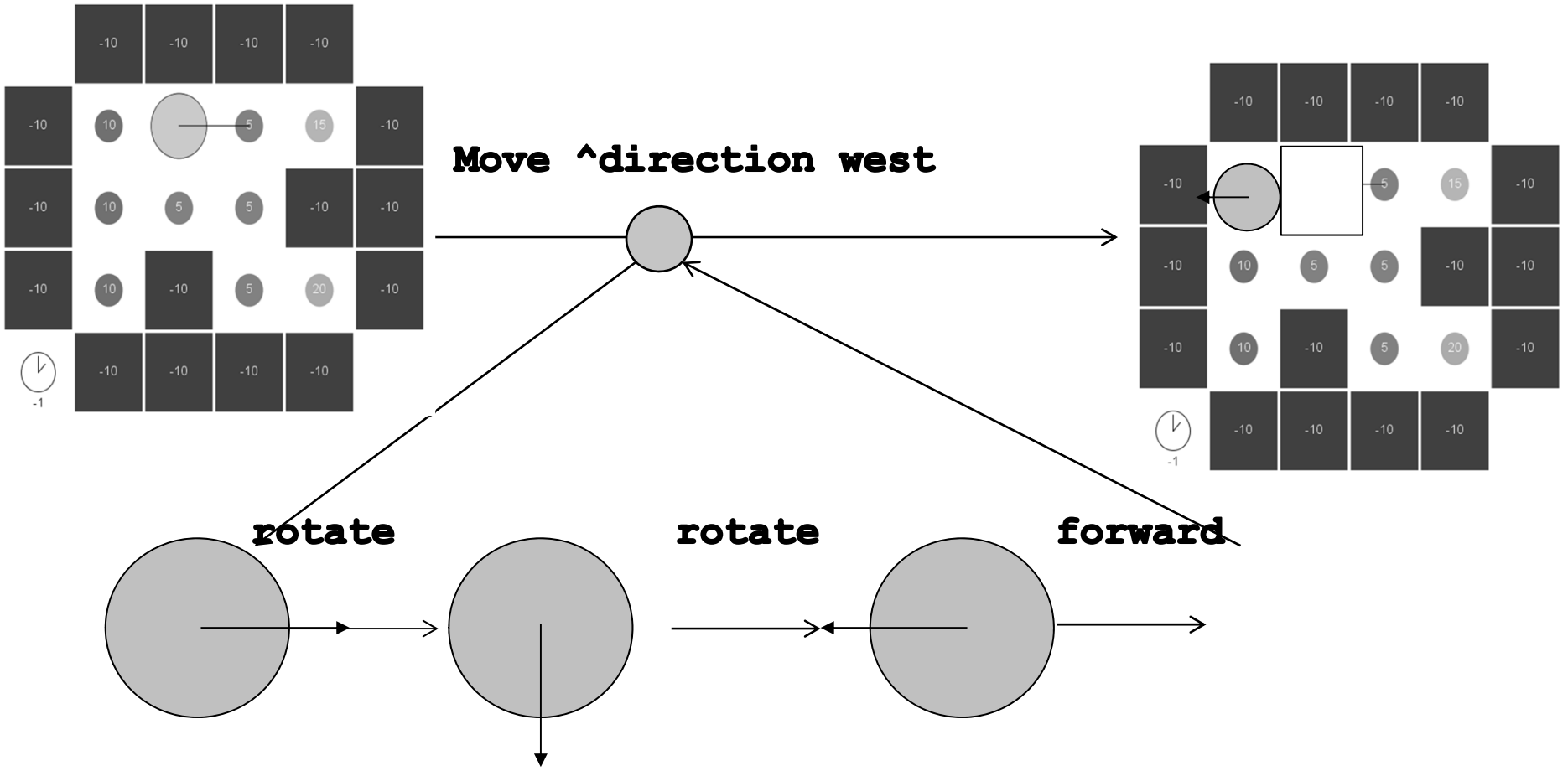
Switch to Eaters

- Create abstract **Move** operators that combine turn and forward to move in a given cardinal direction.

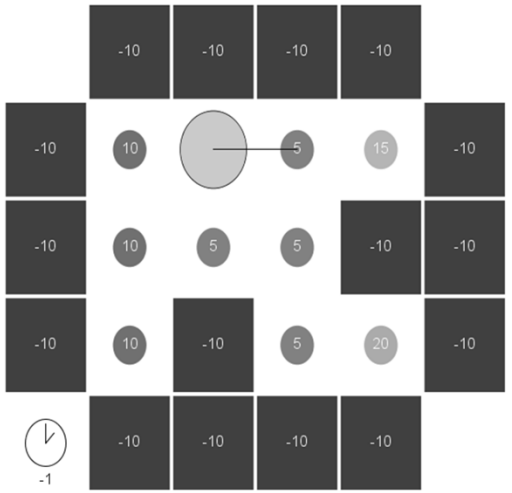
Operator Implementation

- Add operator to move in a cardinal direction:

```
(<o> ^name move ^direction << north south east west >> )
```



Move Operator

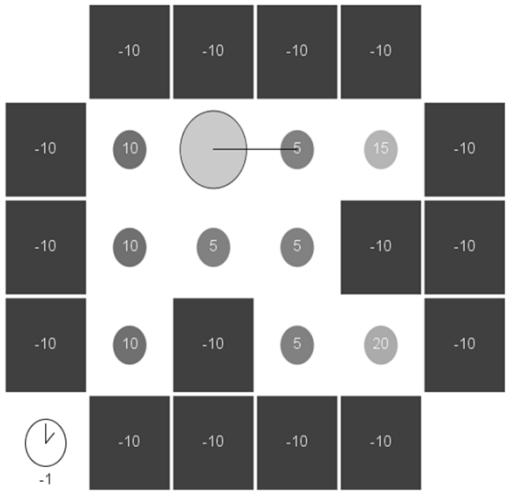


- Need only a proposal.
- Apply will be in substate.

```
If name eater and
    in direction <dir> there is a non-wall
then
    propose move in direction <dir>
```

Cardinal directions don't "blink" during rotate, only during forward

Propose Move



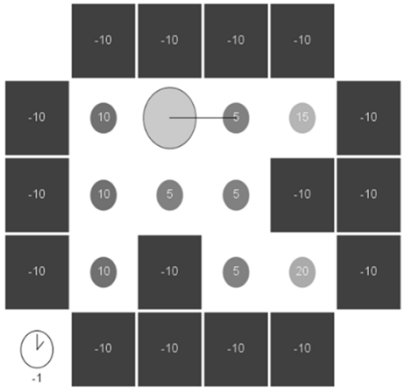

```
If in the hierarchical-eater and
    in direction <dir> there is a non-wall
then
    propose move in direction <dir>

sp {eater*propose*move
   (state <s> ^name eater
              ^io.input-link <input>)
   (<input> ^{ << west east north south >> <dir> } <> wall)
-->
   (<s> ^operator <op> + =)
   (<op> ^name move
         ^direction <dir>)}
```

Substate Structure

```
(s1 ^superstate nil
    ^type state
    ...)
(s9 ^attribute operator
    ^choices none
    ^impasse no-change
    ^superstate s1
    ^type state
    ^smem …
    ...)
```

No ^io structure in substate

Rotate in Substate




```
# If not facing direction of move (in superstate), propose rotate.
# When rotate, will retract this proposal (orientation blinks)
# Move operator will not retract (directions don't blink on rotate)

sp {move*propose*rotate
   (state <s> ^superstate <ss>)
   (<ss> ^operator <o>
         ^io.input-link.orientation <dir>)
   (<o> ^name move
        ^direction <> <dir>)
-->
   (<s> ^operator <op> + =)
   (<op> ^name rotate)}


sp {apply*rotate
   (state <s> ^operator.name rotate
              ^superstate.io.output-link <out>)
  -(<out> ^rotate)
-->
   (<out> ^rotate <r>)}
```

Goal Initialization Conventions

Default rules:

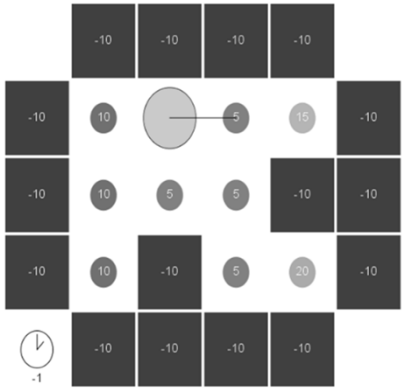
- Always copy down a pointer to the top-state
  - `(<s> ^top-state <ts>)`
- Always copy down a pointer to the io-links
  - `(<s> ^io <io>)`
- Usually name the state with the name of the superoperator
  - `(<s> ^name <operator-name>)`

- Often copy down parameters of operator
  - `(<s> ^direction <dir>)`
  - No default rules to help with this

Simplify Substate

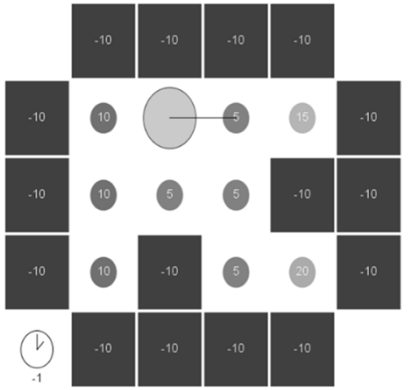
```
sp {eater*elaborate*state                  # included in base agent!
   (state <s> ^superstate.operator.name <name>)
-->
   (<s> ^name <name>)}

sp {eater*elaborate*state*io               # included in base agent!
   (state <s> ^superstate.io <io>)
-->
   (<s> ^io <io>)}

sp {move*propose*rotate
   (state <s> ^name move
              ^superstate.operator.direction <> <dir>
              ^io.input-link.orientation <dir>)
-->
   (<s> ^operator <op> + =)
   (<op> ^name rotate)}


sp {apply*rotate
   (state <s> ^operator.name rotate
              ^io.output-link <out>)
  -(<out> ^rotate)
 -->
   (<out> ^rotate <r>)}
```

Forward in Substate




```
# If facing direction of move (in superstate), propose forward.

sp {move*propose*forward
   (state <s> ^name move
              ^superstate.operator.direction <dir>
              ^io.input-link.orientation <dir>)
 -->
   (<s> ^operator <op> + =)
   (<op> ^name forward)}


sp {apply*forward
   (state <s> ^operator.name forward
              ^io.output-link <out>)
  -(<out> ^forward)
-->
   (<out> ^forward <r>)}
```

Other Rules to include

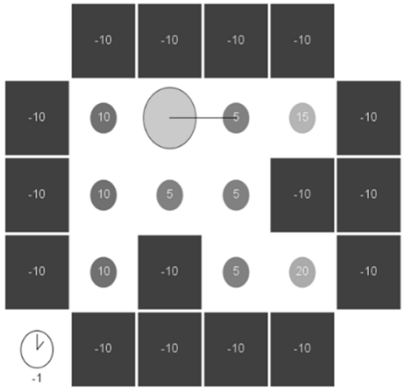



```
sp {eater*monitor*move
    (state <s> ^name eater
               ^operator <op>)
    (<op> ^name move
          ^direction <dir>)
-->
    (write (crlf) |Move direction: | <dir>)}


sp {task*complete
    (state <s> ^name eater
               ^io.input-link.food-remaining 0)
-->
    (halt)
}
```

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 substate 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

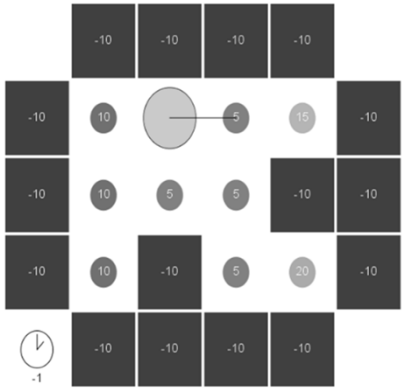
Adding Selection Knowledge



Select operators based on value of food they will consume.

1. Maintain in working memory information on what each color is worth.
2. Propose operators with value they will get
3. Use that values to select operators.

Values in working memory




```
sp {hierarchical*elaborate*map-object*reward
   (state <s> ^name eater)
-->
   (<s> ^color-values <r>)
   (<r> ^wall -10
        ^empty 0
        ^red 5
        ^purple 10
        ^green 15
        ^blue 20)}
```

Numeric Indifferent Rule

```
sp {eater*propose*move
   (state <s> ^name eater
              ^io.input-link <input>)
   (<input> ^{ << west east north south >> <dir> }
                                    { <> wall <color> })
-->
   (<s> ^operator <o> + =)
   (<o> ^name move
        ^direction <dir>
        ^color <color>)}

sp {eater*select*move*operator*indifferent
   (state <s> ^operator <o> +
              ^color-values.<color> <value>)
   (<o> ^name move
        ^color <color>)
-->
   (<s> ^operator <o> = <value>)}
```

Comparison Rule

```
sp {eater*select*move*operator
   (state <s> ^name eater
              ^operator <o1> +
              ^operator { <> <o1> <o2> } +
              ^color-values <cv>)
(<cv> ^<color1> <value1>
      ^<color2> < <value1>)
(<o1> ^color <color1>)
(<o2> ^color <color2>)
-->
(<s> ^operator <o1> > <o2>)}
```

Hierarchical RL

- Just use RL for move operators

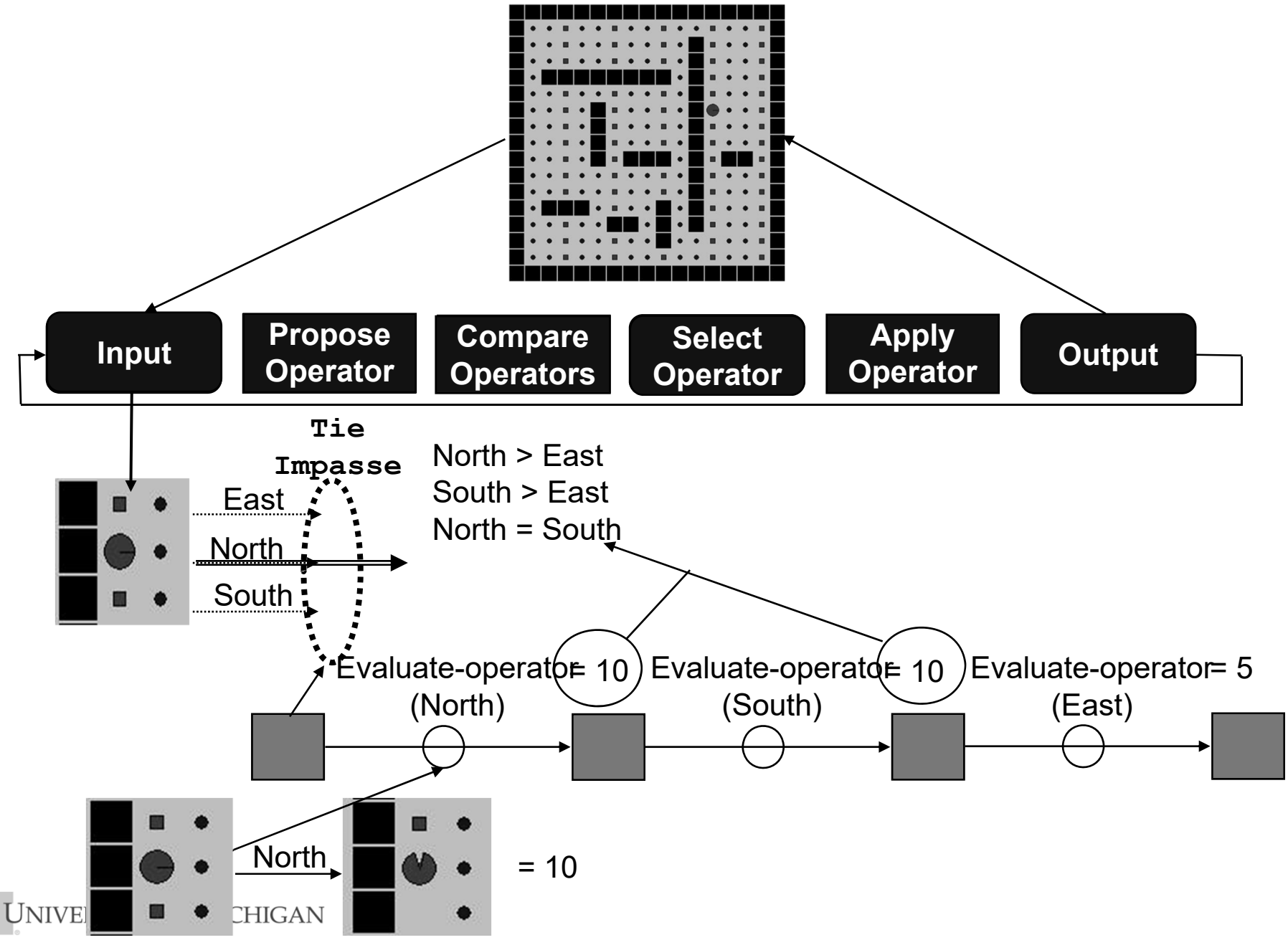
```
sp {eater*propose*move
   (state <s> ^name eater
              ^io.input-link <input>)
   (<input> ^{ << west east north south >> <dir>}
                  { <> wall <color>})
-->
   (<s> ^operator <o> +)
   (<o> ^name move
        ^direction <dir>
        ^color <color>)}

gp {eater*select*move
   (state <s> ^name eater
              ^operator <o> +)
   (<o> ^name move
        ^color [ purple red blue green empty ] )
-->
   (<s> ^operator <o> = 6)}
```

Hierarchical RL

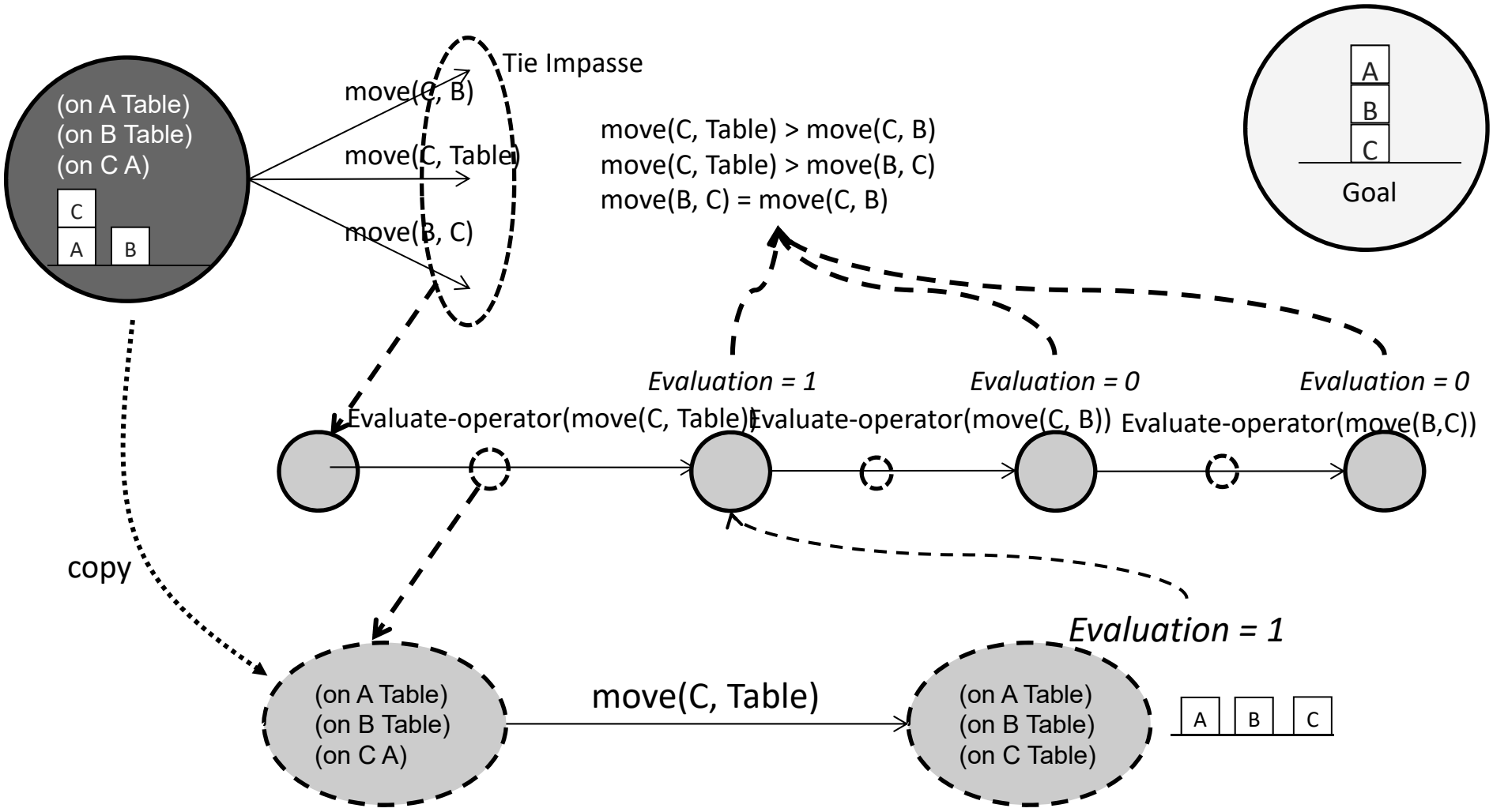
- Just use RL for move operators

```
sp {RL*elaborate*state                # same as in normal RL
   (state <s> ^name eater
              ^reward-link <rl>
              ^io.input-link.score-diff <d>)
-->
   (<rl> ^reward.value <d>)}
```

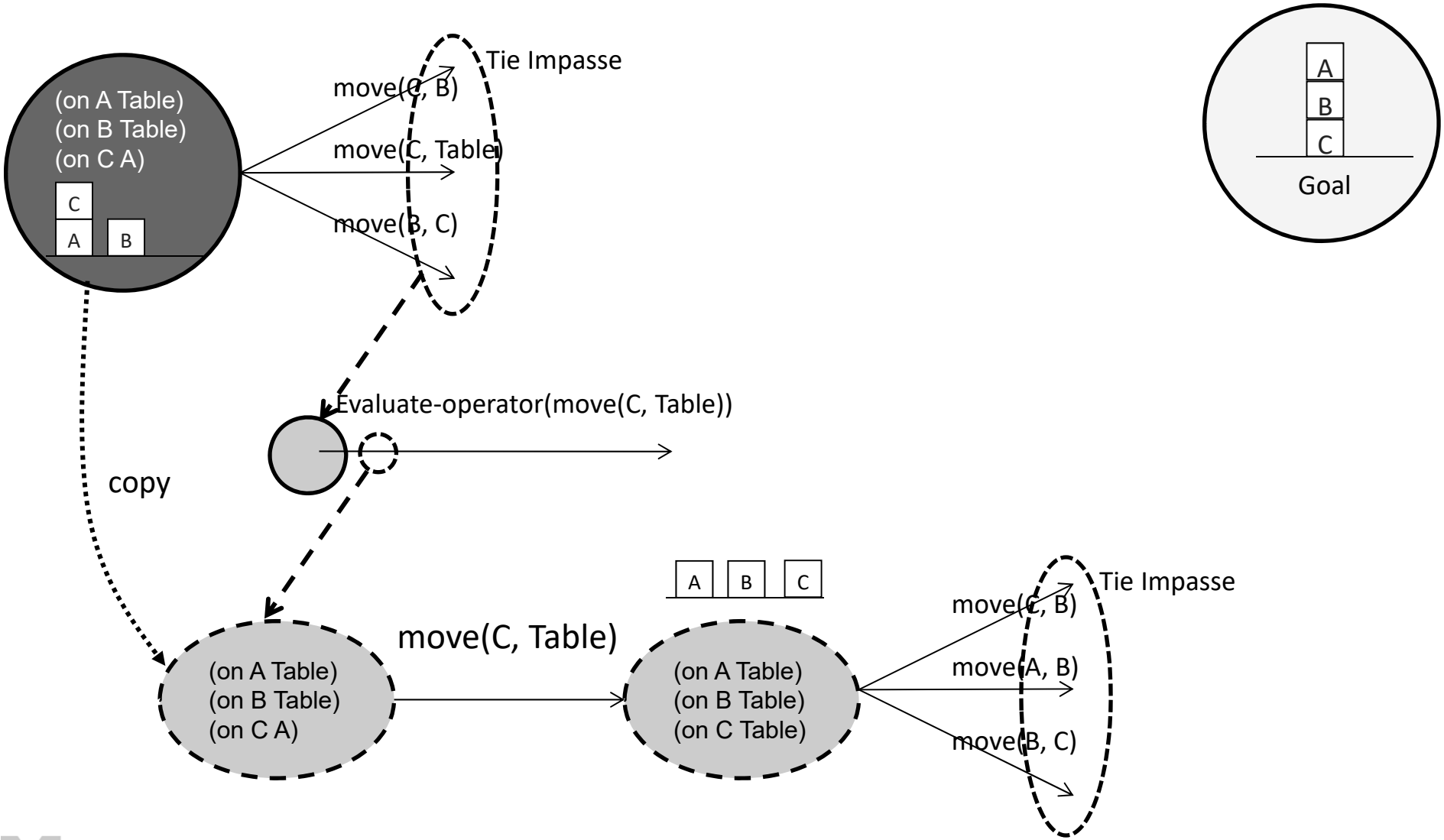
Tie Subgoals



Overview One-step Look-ahead Using Selection Problem Space



Overview One-step Look-ahead Using Selection Problem Space



Depth-First Search in Soar

- If no evaluation of the state, continues in substate
  - If sufficient knowledge, selects and applies operator
  - If insufficient knowledge, get a tie impasse and recursively get depth-first search.
- The state "open" list is represented as the stack of substates.
- Elaboration rules pass success up the stack to avoid extra search.
- No guarantee of finding shortest path.

Selection Space

- Important state structures created by Soar
  - ^impasse tie, ^item 01 02 …
- Evaluate-operator
  1. Instantiated with every item (every tied operator) that has not been evaluated

     ```
     (<s> ^operator <o>)
     (<o> ^name evaluate-operator
                  ^superoperator <so>)
     ```
  2. Usually randomly select between them (some exceptions)
  3. Create ^evaluation structure on selection state

Evaluate State Structure

- When evaluate-operator is selected, create:
  - (<s> ^evaluation <e>)
  - (<e> ^superoperator <i>)
  - (<o> ^evaluation <e> # on evaluate-operator
  - ^superstate <ss> # task state
  - ^superproblem space <ps>)
- Evaluate-operator terminates when a value is created on the associate evaluation
  - (<e> ^value true)

Evaluate-operator Substate

- Create a *copy* of the task state
  - Includes ^name, ^desired
  - ^problem-space determines how to create copy
    - Many flags to control what to copy and how deep
    - ^default-state-copy yes is default
- If don't create copy, original state will change

Evaluate-operator Processing

1. Force selection of a copy of the operator being evaluated
2. Operator application rule should fire and generate new state
   - Requires *action model:* operator application rule for simulating operator
   - If doesn't, will eventually get impasses that lead to a failed evaluation.
3. If there is state evaluation knowledge, it adds augmentation to state
   - ^numeric-value, ^symbolic-value, ^expected-value
   - Copied up to the evaluation structure in the selection space
   - Leads to evaluate-operator terminating

- By default, elaboration rules aggressively convert evaluations to preferences.
  - Evaluates only as many operators as necessary to generate preferences to break the tie.
- Chunks are learned for computing evaluations and preferences

Iterative Deepening

- Include an evaluation-depth in the selection space
- Evaluate all of the task operators to that depth
  - Start with depth = 1
  - In each recursive selection substate, decrement depth
- Terminate if achieve goal
- Increment depth when all task operators have been evaluated

Requirements to use Selection Space

- Source in selection.soar!
  - Explains the following requirements
- Have a ^problem-space structure on the state
- Have a ^desired structure on the state
- Include rules that compute failure/success/evaluation.
- Have rules that simulate action of operators
  - This is an *action model*
  - Only apply when in state with ^name evaluate-operator

Implications of Substates:

- Substate = goal to resolve impasse – get more knowledge
  - Generate operator
  - Select operator (deliberate control)
  - Apply operator (task decomposition)
- Knowledge can be in rules or embedded in substates
  - Complete coverage either way.
- All basic problem solving functions open to reflection
  - Operator creation, evaluation, application, state elaboration
- Substate is really meta-state that allows system to reflect

- Chunking converts deliberate reasoning into reactive knowledge
  - Covers all aspects of problem solving.

Task Goals vs. Architecture Goals

- Operator no-changes provide short-term task goals.
  - But are the same as a long-term declarative goal structure.
- Alternative