Knowledge Representation and Reasoning

University "Politehnica" of Bucharest
Department of Computer Science

Fall 2012

Adina Magda Florea

Lecture 12

Soar: an architecture for human cognition (Part 2)

Lecture outline

- Soar representation
- Solving a goal
- Sub-goaling and Chunking in Soar
- Extended Soar architecture

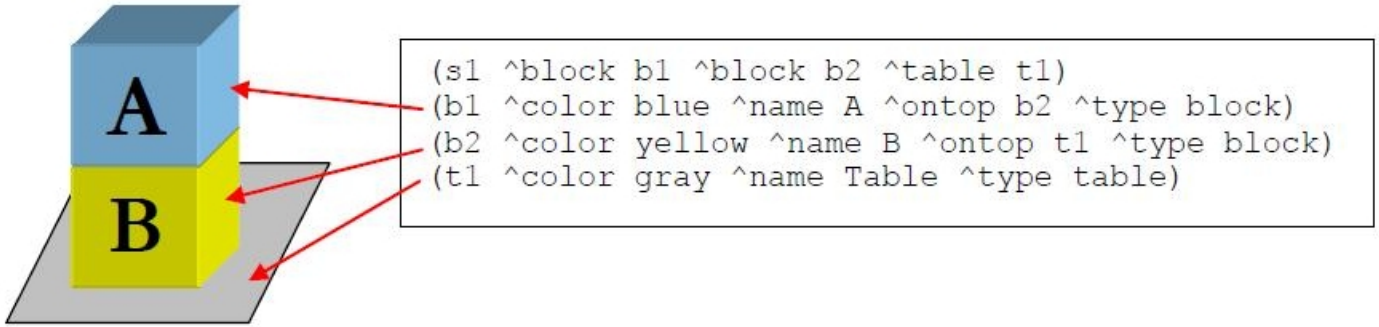
1. Soar representation

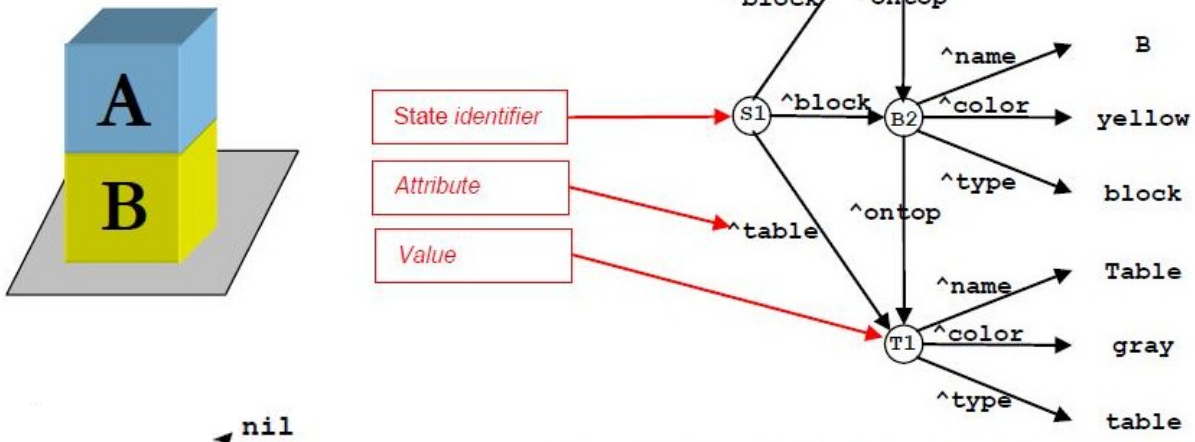
- Problem space (ps): set of states, set of operators
- A goal has 3 slots: problem space, state, operator
- Context = ps, state, ops, goal
- Goals can have sub-goals (and assoc contexts) => goal- subgoal hierarchy

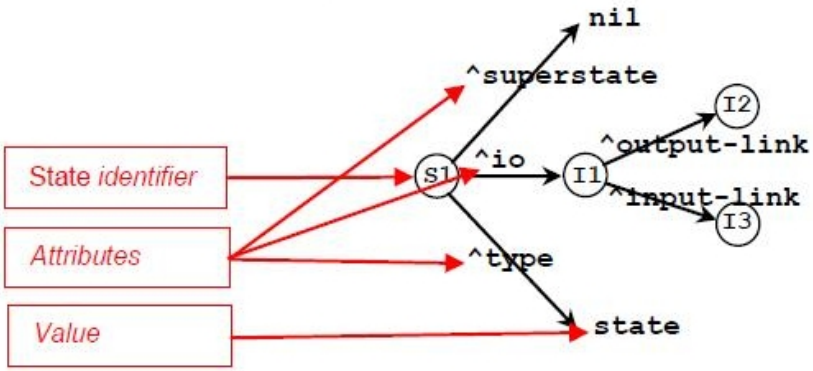
Soar representation

Objects

- have a unique **identifier** generated at the time the object was created
- have **augmentations** – further descriptions of the object
- Augmentation = triple (identifier, attribute, value)
- All objects are connected via augmentations in a ***context*** =>
- Identifiers of **objects** act as **nodes** of a semantic network where **augmentations** are the **links**


A
B
(s1 ^block b1 ^block b2 ^table t1)
(b1 ^color blue ^name A ^ontop b2 ^type block)
(b2 ^color yellow ^name B ^ontop t1 ^type block)
(t1 ^color gray ^name Table ^type table)

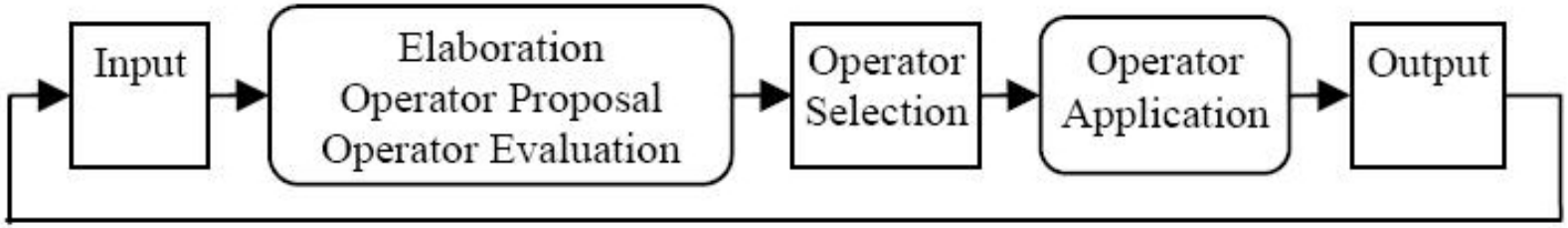

A
B
State identifier
Attribute
Value
S1
^block
^table
B1
^color
blue
^type
block
^ontop
B2
^name
B
^color
yellow
^type
block
T1
^name
Table
^color
gray
^type
table


State identifier
Attributes
Value
S1
nil
^superstate
^io
I1
I2
^output-link
^input-link
I3
^type
state

Remember that

- Rules that *propose* operators
- Rules that *evaluate* operators
- Rules that *apply* the operator

- Soar processing cycle



2. Solving a goal

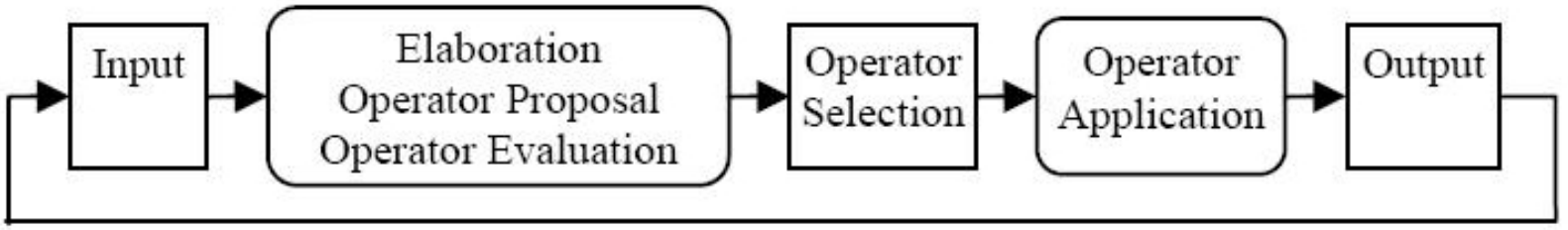
- Select a ***problem space*** (ps) for the goal
- Select an ***initial state***
- Select an ***operator*** to apply to the state
- Enter ***elaboration phase*** = collects all available knowledge relevant to the current situation => explicit search control knowledge

Info added during elaboration:

- existing objects have their description elaborated via augmentations (e.g., addition of an evaluation to a sate)
- create data structures - preferences

Solving a goal

- Repeat elaboration phase until ***quiescence***
- Use a fixed ***decision procedure*** to interpret preferences
- ***Apply*** operator



Solving a goal by sub-goaling

- During decision an **impasse** may occur (see previous lecture)
- Soar respond to an impasse by creating a sub-goal and an associated context
- Again select ps, init state, op, elaboration, etc
- If an impasse is reached, another sub-goal is created to resolve it – hierarchy of sub-goals
- **Universal sub-goaling in Soar** = the capability to generate automatically sub-goals in response to impasses and to create impasses for all aspects of problem solving behavior

Sub-goaling

- Because all goals are generated as responses to impasses, and each local goal has at most one impasse at a time, the goals (and associated contexts) in WM are structured as a stack -> **context stack**
- A sub-goal terminates when its impasse is resolved
  - pops the context stack
  - removes from WM all augmentations created in that sub-goal which are not connected to a prior context, either directly or indirectly (by a chain of augmentations)
  - removes from WM preferences for objects not matched in prior contexts
- The **augmentations** and **preferences** that are **NOT removed** are the **result** of the sub-goal

Sub-goaling

Remember that there are **4 types of impasses** in SOAR

- (1) **A state no-change impasse** - no operators are proposed for the current state
- (2) **An operator tie impasse** - there are multiple operators proposed, but insufficient preferences to select between them
- (3) **An operator conflict impasse** - there are multiple operators proposed, and the preferences conflict
- (4) **An operator no-change impasse** – an operator has been selected but there are no rules to apply it

3. Sub-goaling and Chunking

- Problem solving in Soar can support learning:
  - **When new knowledge is needed**: when an impasse occurs
  - **What to learn**: information discovered / obtained when an impasse is resolved = problem solving within a subgoal
  - **When to acquire new kowledge**: when a subgoal was achieved because its impasse has been resolved

3.1 An example of problem solving

Eight Puzzle

- A **single general op** to move adjacent tiles in the empty cell
- For a given cell, an instance of this op is created for each cell adjacent to the empty cell: up, down, left, right
- To encode this, we need productions that:
  - propose a problem space (ps)
  - create the initial state of the ps
  - implement the ops for the ps
  - detect the desired state (final state)

An example of problem solving

- If no knowledge is available, a depth-first search occurs as a result of the default processing of tie impasses
- Op selection -> tie impasses -> sub-goals for this impasses -> another tie impasse – another sub-goal -> search deepens
- If search control knowledge is added to the ops -> informed search

Another way to control search

- Decompose goal into sub-goals = **means-end analysis**
- Question?
- **Are sub-goals non-serializable?** = tasks for which there exists no ordering of sub-goals such that successive sub-goals can be achieved without undoing what was accomplished earlier
- **Non-serializable goals** can be tracktable if they are **serially decomposable** = if there is an ordering p_1, .., p_k, .. such that the solving of p_k depends only on p_1, …, p_{k-1}
- See also linear vs non-linear planning

Another way to control search

- We can do this for Eight Puzzle
- Operators:
  - (1) have the blank in its correct position in Sf
  - (2) have the blank and the first tile in their correct position in Sf
  - (3) have the blank and the first 2 tiles in their correct position in Sf
  - …
  - Obs about undone and redone
- This is not direct search control knowledge but knowledge about how to structure and decompose the puzzle

Another way to control search

We can use 2 problem spaces

- **eight-puzzle** – the set of 4 ops
- **eight-puzzle-sd** – the set of nine ops corresponding to the nine sub-goals

- The ordering of the sub-goals is encoded as search control knowledge that creates preferences for the operators

1 G1 solve eight-puzzle
2 P1 eight-puzzle-sd
3 S1

2	3	1
	8	4
7	6	5

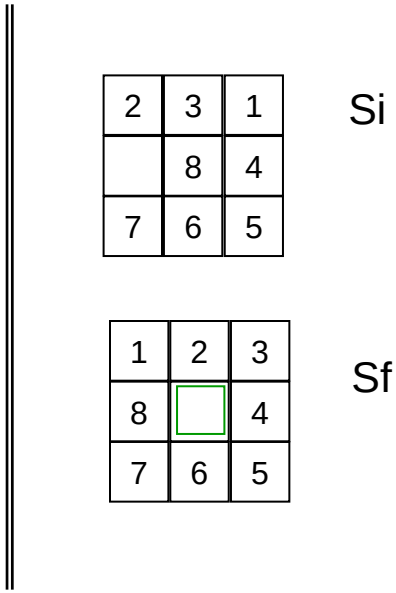
4 O1 place-blank
5 ==> G2 (resolve-no-change)
6 P2 eight-puzzle
7 S1
8 ==> G3 (resolve-tie operator)
9 P3 tie
10 S2 (left, up, down)
11 O5 evaluate-object (O2(left))
12 ==> G4 (resolve-no-change)
13 P2 eight-puzzle
14 S1
15 O2 left
16 S3

Returns pref to select left in S1/G2

2	3	1
8		4
7	6	5

17 O2 left
18 S4
19 S4
20 O8 place-1

A part of a problem-solving trace for the Eight Puzzle



2	3	1
	8	4
7	6	5

Si

1	2	3
8		4
7	6	5

Sf

Explanations

- (1) G1 – id to represent goal in the WM

  init current goal, start problem solving
- (2) problem solving begins in the eight-puzzle-sd ps P1
- (3) creates first sate S1 in P1
- (4) preferences are generated that order the ps so that place-blank is selected (O1)
- (5) A no-change-impasse leads to a sub-goal (G2) to implement place-blank (impasse will be resolved when the blank is in the desired position)
- (6) place-blank – implemented as a search in the eight-puzzle ps (P2 for a state with the blank in correct position
- (7) init state selected in P2
- (8) tie-impasse to select among left, up, down
- (resolve-tie-impasse subgoal G3 is automatically generated
- (9) tie ps is selected (P3)

Explanations

- (10) For the tie ps (P3) the states are set of objects to be evaluated and the set of ops evaluate objects so that preferences can be created
- (11) Suppose one of these evaluate-object operators (O5) is selected to evaluate the op that moves 8 to the left
- (12) A n-change impasse and sub-goal G4 is generated because there are no productions to compute an evaluation of the left operator
- Default search control knowledge attempts to implement the evaluate-object op (O5) by applying the left op to S1
- (13)-(16) – goes to S3

- Each tile is moved into place by one operator in the eight-puzzle-sd problem space
- The sub-goals that implement these operators are generated in the eight-puzzle ps

3.2 Chunking in Soar

- **Basic idea**
  - Task: [ont(a), ont(b), free(b)] → [on(a,b)]
  - Result: { pick(a), stack(a,b) }
- **Learn Chunk C1:**

  if the task is "Do a plan for [ont(a), ont(b), free(b)] → [on(a,b)]"

  then the result is { pick(a), stack(a,b) }
- **Generalize**
  - Task: [ont(X), ont(Y), free(Y)] → [on(X,Y)]
  - Result: { pick(X), stack(X,Y) }
- **Store generalized Chunk C1:**

  if the task is "Do a plan for [ont(X), ont(X), free(Y)] → [on(X,Y)]"

  then the result is { pick(X), stack(X,Y) }

How are chunks created

- Chunking creates rules that summarize the processing of a sub-goal
- In the future, this process can be replaced by a rule application
- When a *sub-goal is generated*, a learning episode begins which can lead to the creation of a chunk.
- When a *sub-goal terminates*, a chunk can be created
- A **chunk** is **a rule** or **a set of rules**

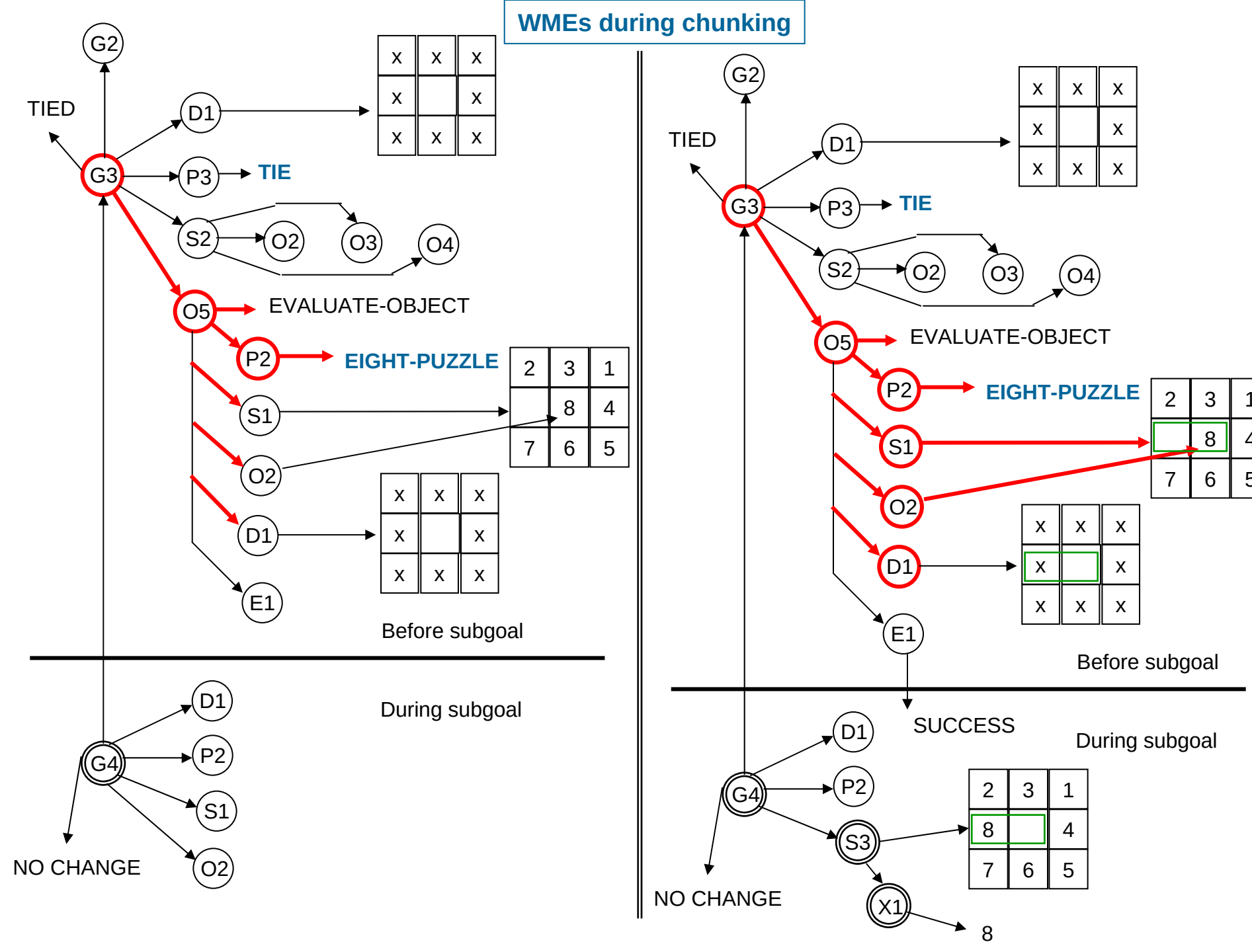
How are chunks created

- **Chunk conditions** – aspects of the situation that existed prior to the goal + were examined during the processing of the goal
- **Chunk action** – the result of the goal
- When a goal terminates, the WMEs are converted into the conditions and action of one or more production rules

Collecting conditions and actions

- **Conditions** = WME that were matched by productions that fired in the goal (or one of its sub-goals) and existed before the goal was created
- This collection of WMEs are maintained for each goal in a **reference-list**
  - (implies for any rule application to look for which goal in the stack is a rule associated)
- **Action** = based on the results of the goal

WMEs during chunking



Collecting conditions and actions

- **Chunk learned:** conditions + action
- Conditions test if there is an evaluate-object operator which can evaluate the **op** that moves the blank into its desired location + all tiles are either in the desired position or irrelevant for the current eight-puzzle-sd operator
- Action marks the evaluation as successful (the op being evaluated achieves the goal)
- The chunk implements the evaluate-object operator (avoiding thus the no-change impasse and sub-goaling)

Identifier variabilization

- All identifiers are replaced by problem variables
- Constants are left unchanged (e.g., evaluate-object, eight-puzzle)
- All occurrences of a single identifier are replaced with the same variable and all occurrences of different identifiers are replaced with different variables
- Conditions to ensure that two different variables can not match the same identifier
- Maybe over-specialization

Chunk optimization

- Remove conditions from chunk that provide no constraint on the match process
  - A condition is removed if it has a variable in the value field of the augmentation that is not bound elsewhere in the rule (condition or action)
- Apply a condition ordering algorithm to take advantage of the Rete-network matcher used in Soar

4. Extended Soar architecture