Limited Parallel Operators for Soar

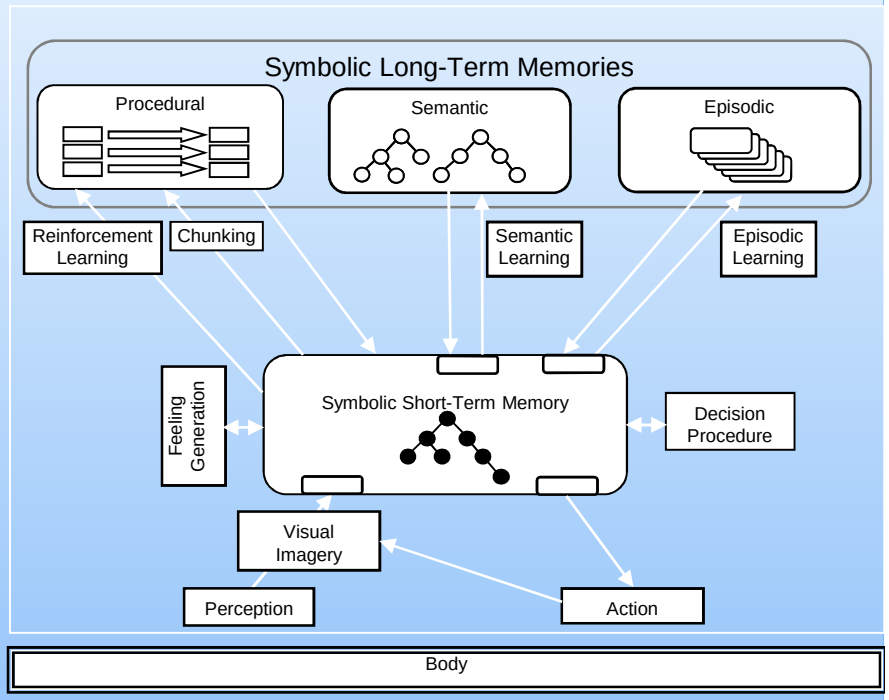
John Laird, University of Michigan

29th Soar Workshop

June 2009

Possible Levels for Parallelism in Soar

- Within Module Implementation:
  - Production match (Gupta et al., 1986; …)
  - Parallel episodic/semantic memory search
- Within Module Execution
  - Parallel production firing (XAPS; Soar 1-9)
  - Parallel retrievals from episodic/semantic
- Multi-Module Execution
  - Modules run asynchronously (ACT-R)
- Multi-Module Implementation
  - Modules on separate cores (Ray, 2010)
- Deliberation
  - Parallel operators (Newell & Rosenbloom)
- Task Level
  - Multiple goals (Chong, 1997)
- Multi-agent



Advantages of Operator-level Parallelism

- Increase module-level parallelism
  - Could eventually lead to faster implementation
- Eliminates need to control sequential processing
  - Currently must ensure that there is no operator "starvation"
  - Improve reactivity
- Example situation
  - Agent ready to act in world, retrieve for episodic memory, and do internal reasoning
  - Currently must select one to go first (possibly at random)
  - Must make sure it at some point initiates others

Prior Approach/Idea

"To enable further research on task-level parallelism we have added the experimental ability to simultaneously select multiple problem space operators within a single problem solving context. Each of these operators can then proceed to execute in parallel, yielding parallel subgoals, and ultimately an entire tree of problem solving contexts in which all of the branches are being processed in parallel. We do not yet have enough experience with this capability to evaluate its scope and limits."

- Rosenbloom, Laird, Newell, & McCarl, 1991

Potential Issues with Prior Proposal

1. Conflicting actions between parallel operators.
   - Operator applications from different operators are where the problem can arise
   - Different than parallel rules that are part of same operator which are designed/learned to cooperate.
2. No architectural constraint on parallelism
   - Processing in substates can grow exponentially

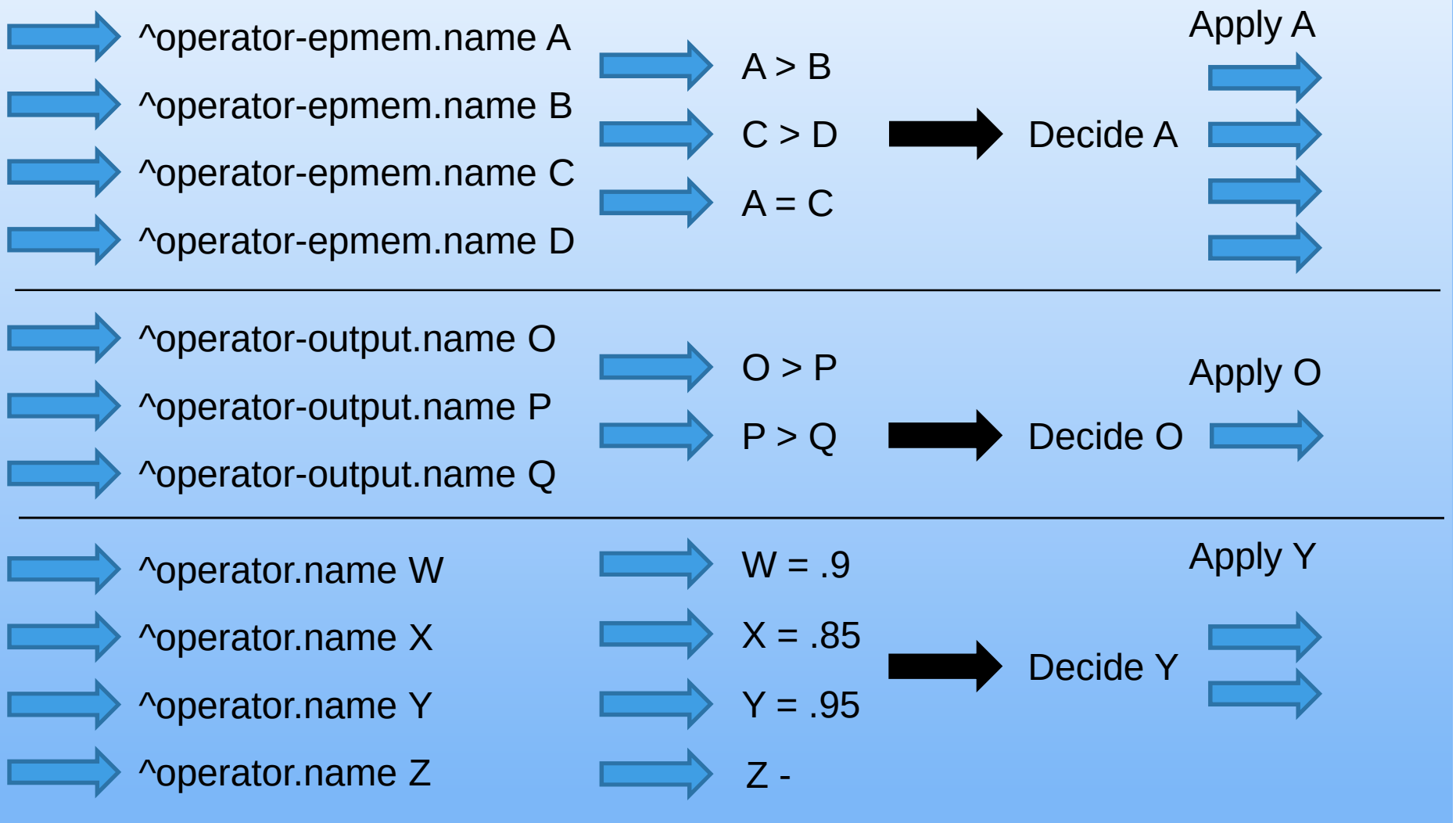
Proposal: More Operator Slots

- A small number of fixed slots for operators based on where they can create/modify WMEs
  - ^operator: modify non-buffer working memory
    - ^operator.type cognitive: “Cognitive operators”
  - ^operator.type output: modify output-link
    - Multiple output slots for independent motor subsystems?
  - ^operator.type epmem: modify ep. memory cue
  - ^operator.type smem: modify sem. memory cue
  - ^operator.type SVS: modify SVS cue

More Details

- Rules propose, evaluate, apply operators in parallel
  - Apply rules can test all of working memory except other operator slots
  - Apply rules can change only their own buffer
    - Can enforce at load time.
  - Data modularization eliminates potential action conflicts
- Operators are synchronized to decision cycle

Parallel Operator Picture



Impasses: General Ideas

- Assume cognitive operator is the only one where progress is necessary.
  - Only operator where state no-change can arise.
  - Other operators can “stall” without impasse
- Can have multiple tie/conflict impasses at the same time.
- Want to avoid explosion in impasses/substates.

Impasse/Substate Proposal

- Multiple active impasses
- Operator slots without impasses continue select/apply
  - Maintains reactivity better than current Soar!

Tie/Conflict Impasse Proposal

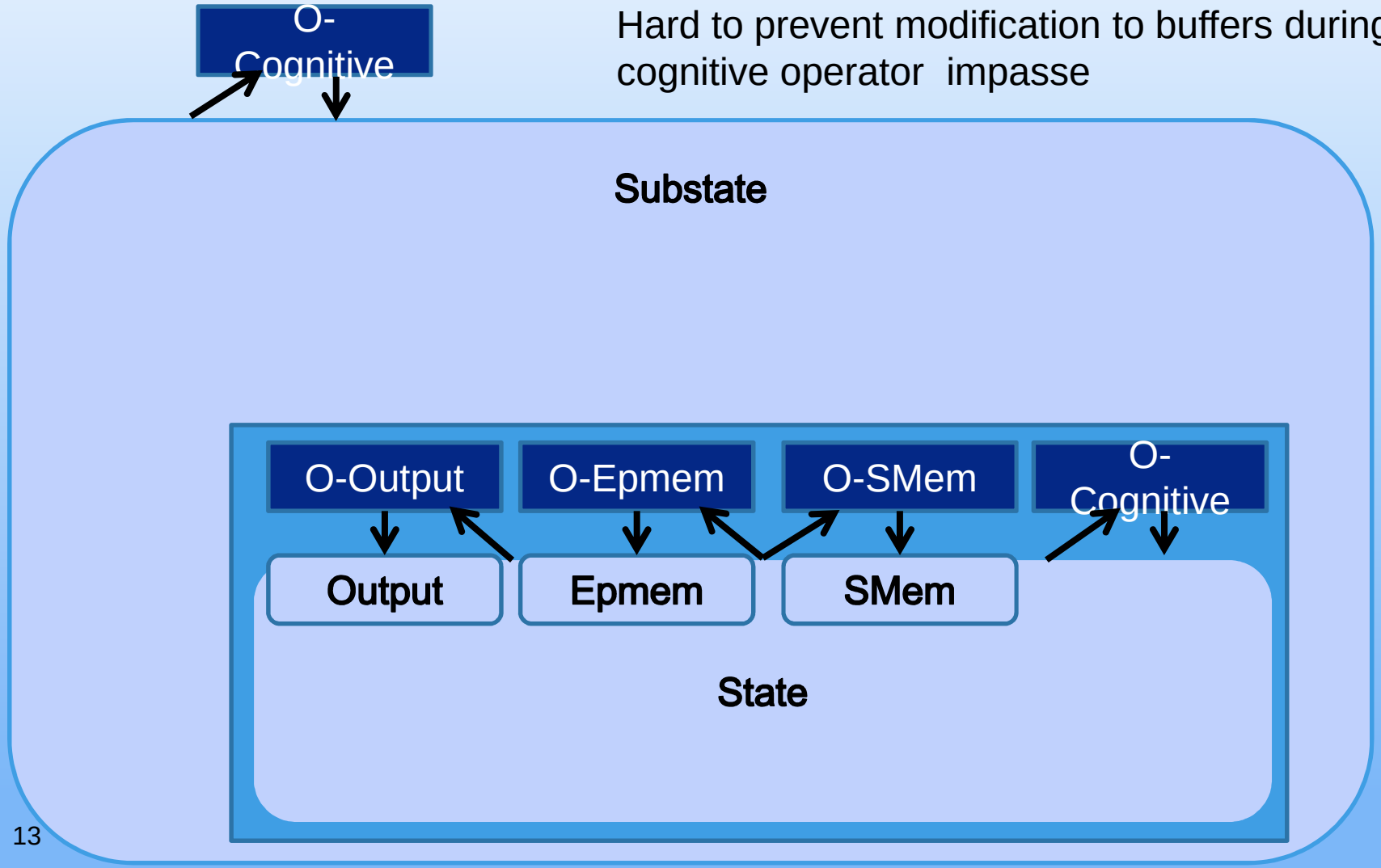
- Generate independent substate for each impassed operator slot
- Within those substates, only single cognitive operator selection is allowed
  - Results are preferences or state elaborations
  - Maintains compartmentalization
  - Avoids multiplicative growth in impasses/substates

Key issue for other impasses

- Need to maintain compartmentalization of operator application knowledge learned from substates and via chunking.
- If processing in a substate creates results that modify a different other WM buffers will learn “illegal” chunks.

Key Issues: Modification in Substates

Hard to prevent modification to buffers during cognitive operator impasse



Operator No-Change Impasses?

- Need to maintain compartmentalization of operator application knowledge even after chunking:
- Proposal 1:
  - Only cognitive operator can have no-change
  - All other operators must have complete rule sets for application
- Proposal 2:
  - Separate substates for each no-change.
  - Can have cognitive operators internal to the substate
  - All results must be restricted to operator's buffer
    - How can this be enforced?
- Proposal 3:
  - Ideas from the audience??

Conclusion

- Coal
  - Still unresolved issues in how to maintain closure with chunking.
- Nuggets
  - Relatively complete proposal to add limited operator parallelism
    - Improve reactivity
    - Eliminate unnecessary control sequencing
    - Eliminate some forms of substate juggling