Augmenting Soar with Non-Symbolic Processing via the IO-Link

29th Soar Workshop

Samuel Wintermute
University of Michigan

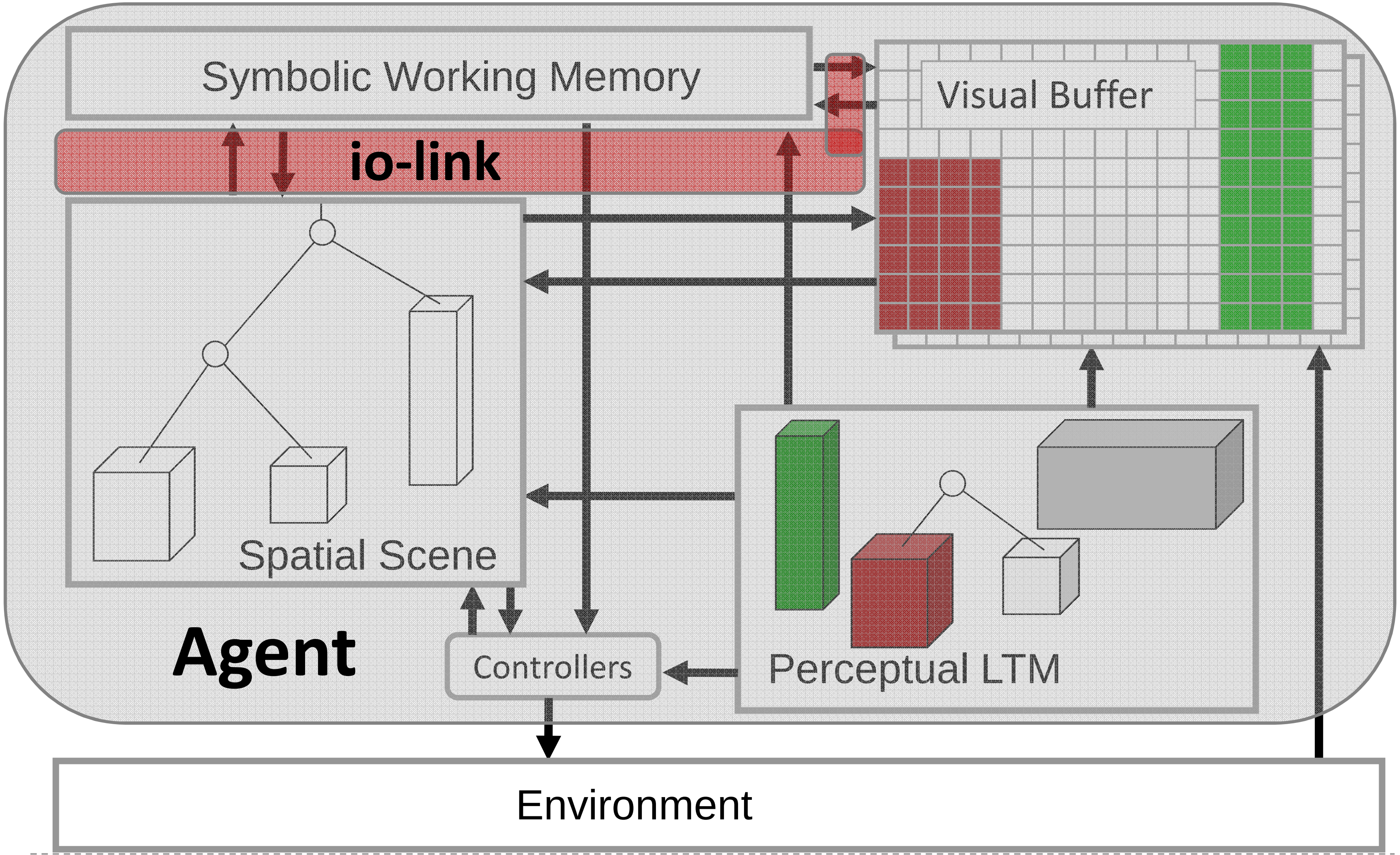
Soar and SVS

- SVS adds spatial and visual processing to Soar
- This involves new non-symbolic memories
  - Memories cannot be “retrieved” into working memory

- Conceptual problem:
  - How should these memories be integrated with symbolic working memory?

- Engineering problem:
  - What should the actual software look like?

Soar/SVS Architecture

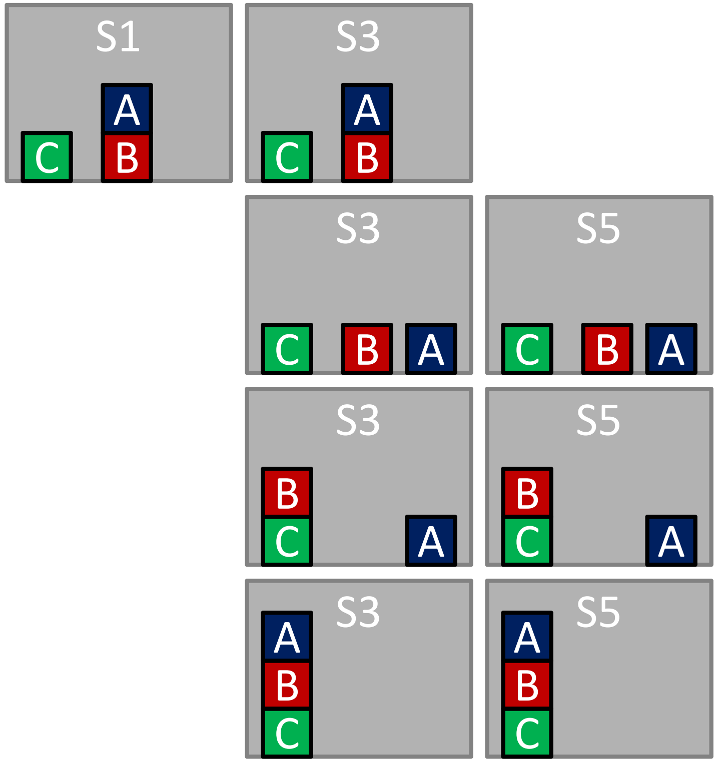


Goal of Integration

```
 1: O: O1 (initialize-blocks-world-look-ahead)
 2: ==>S: S2 (operator tie)
 3:    O: O8 (evaluate-operator)
 4:    ==>S: S3 (operator no-change)
 5:       O: C1 (move-block)
 6:       ==>S: S4 (operator tie)
 7:          O: O22 (evaluate-operator)
 8:          ==>S: S5 (operator no-change)
 9:             O: C2 (move-block)
10:             O: O28 (move-block)
11:       O: O19 (move-block)
12:       O: O18 (move-block)
13: O: O5 (move-block)
14: ==>S: S6 (operator tie)
15:    O: O41 (evaluate-operator)
16:    ==>S: S7 (operator no-change)
17:       O: C3 (move-block)
18:       O: O46 (move-block)
19: O: O38 (move-block)
20: O: O35 (move-block)
blocks-world achieved
```

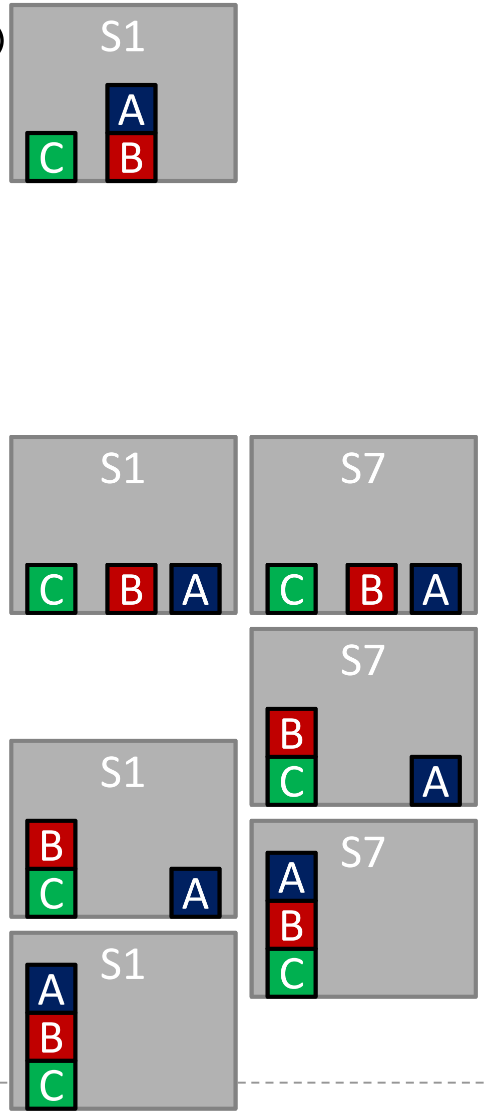
Goal of Integration

```
 1: O: O1 (initialize-blocks-world-look-ahead)
 2: ==>S: S2 (operator tie)
 3:    O: O8 (evaluate-operator)
 4:    ==>S: S3 (operator no-change)
 5:       O: C1 (move-block)
 6:       ==>S: S4 (operator tie)
 7:          O: O22 (evaluate-operator)
 8:          ==>S: S5 (operator no-change)
 9:             O: C2 (move-block)
10:             O: O28 (move-block)
11:       O: O19 (move-block)
12:       O: O18 (move-block)
```



Goal of Integration

```
 1: O: O1 (initialize-blocks-world-look-ahead)
 2: ==>S: S2 (operator tie)
 3:    O: O8 (evaluate-operator)
 4:    ==>S: S3 (operator no-change)
 5:       O: C1 (move-block)
 6:       ==>S: S4 (operator tie)
 7:          O: O22 (evaluate-operator)
 8:          ==>S: S5 (operator no-change)
 9:             O: C2 (move-block)
10:             O: O28 (move-block)
11:       O: O19 (move-block)
12:       O: O18 (move-block)
13: O: O5 (move-block)
14: ==>S: S6 (operator tie)
15:    O: O41 (evaluate-operator)
16:    ==>S: S7 (operator no-change)
17:       O: C3 (move-block)
18:       O: O46 (move-block)
19: O: O38 (move-block)
20: O: O35 (move-block)
blocks-world achieved
```

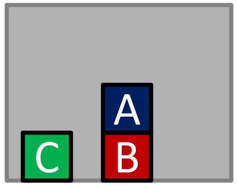


Imagery Rules

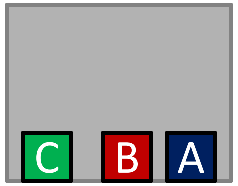
```
sp {apply*move-a-to-table
   (state <s> ^block-a <a>
     ^operator.name move-a-to-table)
   (<a> ^on <on>)
-->
   (<a> ^on <on> -
        ^on table)
}
```

Imagery Rules

```
sp {apply*move-a-to-table
   (state <s>
```




```
      ^operator.name move-a-to-table)
-->
   (<s>
```



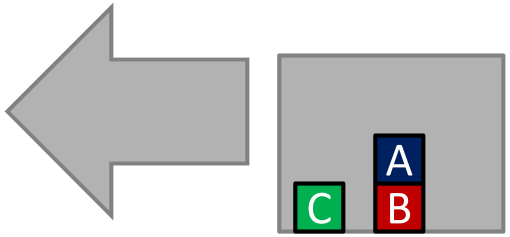

```
       )
 }
```

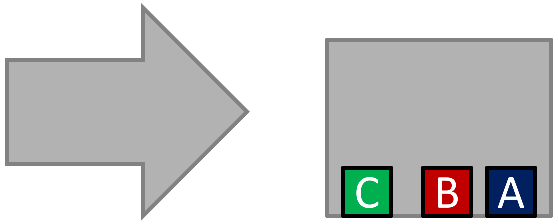
- In what scenes should the rule match?
- How should the scene be modified by the rule?
- Define and name these properties
  - Create a qualitative symbolic interface

Imagery Rules

```
sp {apply*move-a-to-table
   (state <s> ^property <x>
              ^property <y>
              ^property <z>

     ^operator.name move-a-to-table)
-->
   (<s> ^property <x> -
                  <x2>
        ^property <y> -
                  <y2>)
 }
```





- Imagery objects aren’t in WM, but are tightly related to it
- How can this relationship be implemented?

Option: Interfacing via Action Operators

```
2: ==>S: S2 (operator tie)
3:    O: O8 (evaluate-operator)
4:    ==>S: S3 (operator no-change)
          O: XX (add-imagery-structure) * N
          O: XX (extract-property) * N
5:        O: C1 (move-block)
          O: XX (remove-imagery-structure)
          O: XX (add-imagery-structure)
          O: XX (extract-property) * N
   O: XX (remove-imagery-structure) * N
6: O: O9 (move-block)
```

- Treat imagery as an external environment modified via actions
- Problems
  - Slow
  - Requires lots of knowledge
  - No truth maintenance
    - When are images added and removed?
    - When are properties valid?
- Not a tight integration with working memory

Option: Use EpMem/SMem Interface

- Separate interfaces at every state
  - Some truth maintenance benefit
- Parallel access, not necessarily operator-based

- Episodes and semantic structures are symbolic
  - Storage and retrieval are interesting, actually using the data is just regular Soar processing

- Imagery objects are not symbolic
  - “Storage” (WM → imagery) is different
    - Imagery structures are temporary, not long-term
  - “Retrieval” (imagery → WM) is different
    - Imagery objects can not be copied to WM, properties of objects are retrieved instead
    - Truth maintenance is a problem

- More dynamic interface is needed

Working Memory Integration

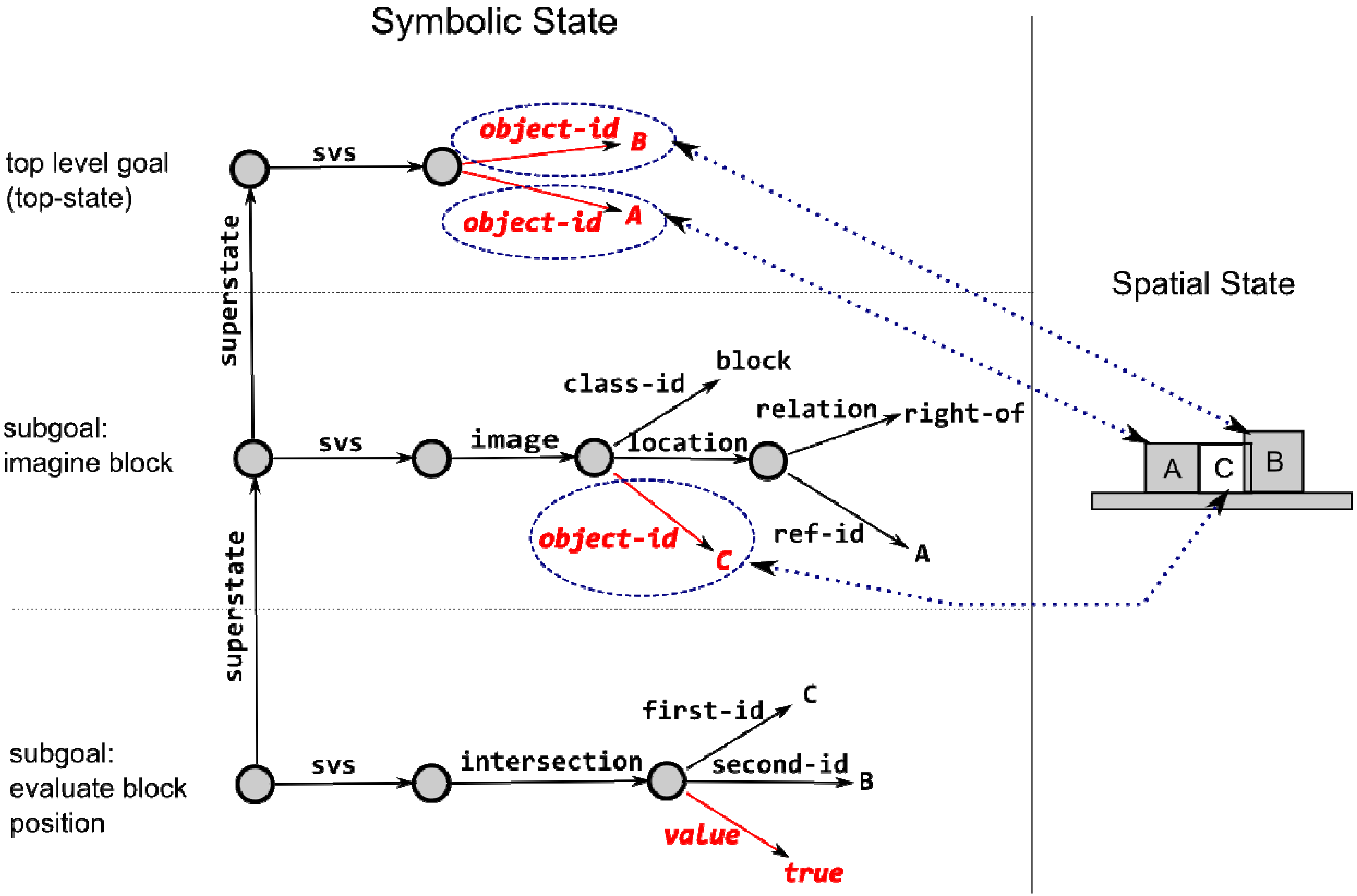
- State-local **svs** structure connects to imagery system
- Most imagery operations are equivalent to a hidden set of elaboration productions

```
svs {create*image
      (state <s> ^svs <svs>)
      (<svs> ^image <im>)
      (<im> ^property foo
            ^property bar)
-->
      (<svs> ^object-id image22)
}
```

```
svs {evaluate*intersect-query
      (state <s> ^svs <svs>)
      (<svs> ^intersection <i>)
      (<i> ^first-object image22
           ^second-object image37)
-->
      (<i> ^value true)
}
```

- Imagery structures persist with WM structures, and update when WM changes
- WM structures queried from imagery persist with query structure, and update when WM or imagery state changes

Working Memory Integration



SVS Implementation

- All communication happens at top-state over the io-link
- Why?
  - Project integration is simple
  - Software evolved from originally using an operator-action interface
  - It works well
- Remainder of talk will cover implementation

Commands on SML Side

- Problem: If commands aren't associated with operators, many can be present simultaneously
- Solution: Allow this, but carefully consider monotonicity

- Most SVS commands are monotonic
  - Can be processed in (pseudo) parallel
  - Even "persistent" structures are monotonic: these are like elaborations of O-supported WM structures
  - These commands must be *reversible*, and retractions must be monitored
  - Intra-command interactions are still possible
    - Current SVS solution is to process commands in one carefully-ordered wave
    - Must guarantee working memory is consistent with commands by input phase

- Some commands are non-monotonic
  - Actual actions passed to the external world
  - Internal commands with global effects

Soar Implementation Basics

- Default productions are defined to
  - build **svs** structures on each state
  - copy commands to output-link from **svs** structures
  - associate commands with responses from input-link
  - fill in decisions requiring multiple i/o phases
  - keep output-link consistent with subgoal **svs** structures

Decision Cycle Integration

- Problem: imagery structures can be i-supported, requiring multiple waves to make a decision, but i/o happens between decisions
- Careful ordering on SML side handles some of this
- Some cases require interleaved imagery and rule-matching, and can't be done in one decision
  - Partial solution: propose filler operator if any commands are present without responses

Subgoal Integration

- Problem: o-supported subgoal imagery must be removed when subgoal goes away
- Problem: i-supported subgoal imagery modifies **output-link**, usually creating o-supported results

- Solution: default cleanup production
- If a command is ever present on **output-link** without an equivalent command on an **svs** WME (on any state), it is removed
  - Production must be o-supported, but does not need its own operator

Subgoal Integration: Interaction Between States

- Problem: There is only one instance of each memory in SVS, but multiple subgoals may be present in Soar

- Solution: monotonic commands prevent most problems
  - Commands in substates cannot interfere with results of commands in superstates if all are monotonic
  - Non-monotonic commands must be issued at top-state

- Superstate processing can still access imagery objects created in substates
  - Could be fixed by notifying SVS of which state commands belong to

Conclusion

- Nuggets:
  - Rich, efficient interaction with non-symbolic memories over the io-link is possible without Soar kernel modification
  - Resulting interface is useful and intuitive
  - Task knowledge can be concisely represented
- Coal:
  - Multiple waves of i/o operation are impossible during the decision cycle, resulting in extra decisions
  - Lack of direct connections to subgoals can cause minor problems with chunking and GDS

Efficiently Processing Changes on SML Side

- Problem: If commands exists for many cycles and have deep structure, what if symbolic processing modifies them?

- Solution: Detect changes in SML WME structures
- Every new WME added via SML has a unique timetag, timetags always increase

- If commands are trees, hashing can be done O(# of WMEs)
  - Parse through WME tree, find highest timetag
  - Simultaneously count how many WMEs are in the tree
  - Hash is {WME count , highest timetag}
    - Adding a WME will result in a new count and new timetag
    - Modifying a WME will result in a new timetag
    - Deleting a WME will result in a lower count
  - Hash can detect changes quickly, but can't detect *what* changed
    - Currently no general-purpose solution to this..

Binding Output and Input

- Problem: **^status complete** is insufficient feedback for commands

- Solution: **request-id**s
  - In Soar, use **make-constant-symbol** to add an id whenever a command appears
  - Externally keep track of id, return result with it on **input-link**
  - In Soar, link the result structure to the command structure
    - **request-id** need not concern agent developers

Binding Input and Output: Example

```
1: (svs.command <c>
     <c> ^parameter one
         ^parameter two)
```

```
2: (svs.command <c>)
     -->
     (<c> ^request-id (make-constant-symbol))
```

```
3: (<c> ^request-id constant23)
```

```
4: (svs.command <c>)
   (<c> ^request-id <id>)
   -->
   (output-link ^command (deep-copy<c>))
```

```
5: (output-link.command <c-copy>
     <c-copy> ^parameter one
              ^parameter two
              ^request-id constant23)
```

6: process command, create response

```
7: (io.input-link.response <r>
     <r> ^request-id constant23
         ^property a
         ^property b)
```

```
8: (svs.command <c>)
    (input-link.response <r>)
    (<c> ^request-id <id>)
    (<r> ^request-id <id>)
    -->
    (<c> ^response <r>)
```

```
9: (<c> ^response <r>)
```

Binding Input and Output: Agent Developer View

```
1: (svs.command <c>
     <c> ^parameter one
         ^parameter two)
```

2: process command, create response

```
3: (<c> ^response <r>
     <r> ^property a
         ^property b)
```

Subgoal Integration: Chunking and GDS

- If an imagery structure is *supposed* to be a result, it can be created on a superstate's **svs** WME
  - Chunking automatically captures these

- If intermediate imagery steps are used during a subgoal, chunking cannot capture those
  - Chunks get created to build the structures on output-link, but the structures are immediately removed by the cleanup rule since there is no state for them
  - This could be fixed, but not without modifying chunking
    - Chunking should *never* completely remove non-symbolic steps from reasoning, though
  - Solution: disable chunking in these subgoals

- GDS also causes non-symbolic subgoal processing to be handled differently than symbolic
  - In some cases, local o-supported imagery structures created based on local non-symbolic reasoning can cause the GDS to remove the goal
  - No solution, but not a common occurrence