Soar Design Dogma

Andrew Nuxoll and John Laird

version 0.6 – 07 July 2003

Introduction

This document contains a collection of Soar wisdom gathered during a series of conversations between John Laird and myself as I mounted the Soar learning curve over the course of my first year as a graduate student at the University of Michigan. My hope was that by writing these guidelines down I might ease the curve for future Soar users. To get the most value from this document, I recommend you read it once at the beginning of your Soar experience and then read it again once you've started using Soar in earnest.

Golden Rule: Beware the devil of "character efficiency"

Focus on creating short productions with only a few conditions and actions. As a rule of thumb, a production should have less than a handful (i.e., five) of each. If you have a production that's too big, it's a sign that you should examine that production and see if it should be broken up into multiple smaller productions.

A production with too many conditions often has a subset of conditions that define a separate "concept" that should be identified with a separate elaboration. When counting conditions, you should only count those that actually test a WME for its value not to reach other WMEs. For example, the condition `(<s> ^io.input-link <il>)` likely doesn't count because it's probably being used to reach and test another WME on the input link.

If a rule has too many actions usually some of those actions are not truly related. Consider separating it into multiple productions with the same or similar conditions. When counting actions, you should only count those that actually add or remove a WME in working memory. Exception: Initialization rules can often have a very large number of actions and this is ok.

Example: The rule shown at left (below) was taken from TankSoar. It has a total of six conditions. This is not necessarily too big. However, if you study this production (and you are familiar with TankSoar), you'll see that it is really testing three high level conditions:

1. Am I in the state `tanksoar`?
2. Am I low on health (less than 300)?
3. Am I in danger?

Adding an elaboration to detect "in-danger" not only makes the rule easier to read but also provides a potentially useful WME for use by other productions. The revised production and its companion elaboration are shown at right.

```
sp {propose*recharge*health-BAD
   (state <s> ^name tanksoar
              ^io.input-link <il>)
   (<il> ^radar.tank.distance > 0
         ^health < 300
         -^smell.distance < 4
         -^sound
         -^incoming)
-->
   (<s> ^operator <o> +)
   (<o> ^name recharge-health)
}
```

```
sp {elaborate*in-danger
   (state <s> ^name tanksoar
              ^io.input-link <il>)
   (<il> ^radar.tank.distance > 0
         -^smell.distance < 4
         -^sound
         -^incoming)
-->
   (<s> ^in-danger yes)
}

sp {propose*recharge*health
   (state <s> ^name tanksoar
              ^in-danger yes
              ^io.input-link <il>)
   (<il> ^health < 300)
-->
   (<s> ^operator <o> +)
   (<o> ^name recharge-health)
}
```

Justification: Smaller productions are easier to re-use and easier to understand when you return to them later. Smaller rules also lead to more general chunks. Instead of watching for a specific combination of data, the chunk will learn to watch for only one matching augmentation on the state. Productions with lots of conditions often produce many partial matches on the RETE network (see the `memories` command in the Soar Manual) that will noticeably slow down performance. Productions with lots of actions can lead to unwanted or unexpected side effects as the program evolves.

Only include the conditions you need

When proposing an operator, don't compute information that won't be needed until application. Exception: It's ok to attach matched values that were already required for the conditions of the proposal rule.

Example: Your TankSoar tank needs to turn on its radar whenever it turns. You've cleverly decided to set the range of the radar based upon the tank's distance from the wall it is facing in order to save energy. Rather than calculate this range in the proposal rule, use a separate elaboration to add the range data once the operator has been selected.

Justification: Clearly this practice reduces rule size and provides a small savings in execution time. It also prevents operators from being rejected and re-proposed when the extraneous data changes.

Consider all your options

Operator proposal rules should fire whenever an operator is legitimate, not just when it is appropriate. Exception: If limiting the agent's options will significantly improve its performance you may consider violating this guideline.

Example: You decide to modify your Eater so that it never moves back to the square it just came from. Rather than only proposing moves to new squares, propose moves to all adjacent squares and use selection operators to assign a less or least preference to the undesired direction.

Justification: You never know when your worst option is also your best. Deciding what to do is the job of operator preference rules. You should not short-circuit this mechanism without good reason. Also, as stated above, additional conditions for operator proposal may cause unnecessary operator retraction and re-proposal.

One action per operator

The name of the operator should indicate specifically what actions you are taking. In particular, avoid "multi-use" operators that perform similar actions for significantly different reasons depending upon what augmentations they have.

Example: Your agent needs to be able to navigate to a waypoint or a given x,y,z position. The move-to-waypoint operators could be implemented as a special case of the move-to-xyz operator. However, this is probably poor practice.

Justification: When an operator is vaguely named or has multiple behaviors your Soar program will be difficult to debug because you aren't certain what the agent is doing. You can get the same effect by having additional augmentations on the operator that trigger the general actions. For example, you could have ^type move on the above operators and there can be selection and application rules that test for the type (but not the name).

Don't force operator proposals

If you find yourself adding operators that put o-supported data on the top state for the sole purpose of causing another rule to fire and not be retracted when the original data changes you're probably doing something wrong.

Example: In a real time system, you want the agent to turn toward an object. First, you write your proposal operator to match the heading (angle off) to the object and then turn to it. Unfortunately, since you must turn to reach the new heading, the angle off to the object changes and the operator retracts before you complete. You decide to have on operator record the angle off on the top state and a second operator turn to that angle off as recorded.

A better solution: Create an I-supported augmentation that records that the object is to the agent's right, left or directly in front. Have your agent turn in the specified direction until the object is in front of the agent.

Justification: Having operators depend on transient, yet o-supported top state augmentations can cause problems if the operator that removes that augmentation is interrupted.

Avoid assumptions

Whenever you remove a WME in the actions of a rule, test that WME exists in the conditions of a rule.

Example:

```
sp {apply*remove-foo
   (state <s> ^name my-state
              ^operator <o>)
   (<o> ^name remove-foo)
-->
   (<s> ^foo bar -)  ←BAD!
}
```

```
sp {apply*remove-foo
   (state <s> ^name my-state
              ^operator <o>
              ^foo bar)  ← Correct
   (<o> ^name remove-foo)
-->
   (<s> ^foo bar -)
}
```

Justification: The purpose of a rule is to minimize implicit assumptions.

Establish a proper context for o-support

When the operator application rule of a substate modifies a superstate, the rule should always include an attribute of that superstate in its conditions. The best way to do this is to reference the state(s) that will be modified by name if possible. (See the next section.)

Example: You create a generic rule that adds a "last-action" WME to the top state (i.e., the last action taken by the agent). To create this generic rule you only match on actions on the output-link.

Justification: O-support means "permanent" only in the context of the state(s) that were tested to create it. Once those state(s) cease to exist, the o-supported WMEs are removed even if they are used to augment a state that has not been removed. Specifically, Soar establishes the context of an o-supported attribute by examining the states that are tested in the operator proposal rule for the operator that created the attribute. Therefore, if an operator proposal rules does not test any part of the state it intends to modify, then o-supported attributes that are created by applying that operator will vanish as soon as you leave the current state.

Use state names

If possible, refer to a state or states by name in the LHS of operator proposal rules.

Example:

```
sp {my-rule
    (state <s> ^name mystate)
  etc…
```

Justification: By being specific about the context of a rule, you prevent it from firing when least expected! This also often prevents more subtle problems like vanishing o-supported WMEs (see previous section).

An agent has only one state

When possible, avoid rules that test multiple states.

Example:

```
sp {my-rule
    (state <s> ^name state1)
    (state <s> ^name state2)
  etc…
```

Justification: This creates multiple matches on the RETE and may, as a result, create performance bottlenecks.

Use VisualSoar

Write your Soar code using VisualSoar and maintain and use your VisualSoar data map!

Example: N/A

Justification: Much like comments on code, VisualSoar's enforced structure and data map is exceptionally helpful for people who are examining code they have not themselves written. VisualSoar provides helpful functionality like syntax highlighting and identifier completion. It also helps you catch bugs in your code by watching for rules that match unspecified WMEs. Such bugs (particularly typos involving nearly identical letters like l and 1) can sometimes require significant effort to discover.

Don't Sacrifice Semantics for Syntax

Only write rules that perform standard problem space functions: state elaboration, operator proposal, operator comparison, operator elaboration, and operator application. Do not write rules that:

1. Test a selected operator in a rule that creates a preference for another operator.
2. Test a for a currently proposed operator in the condition of another operator proposal rule.
3. Test a proposed operator in the conditions of a rule that modify a state (outside of the operator).

Example:

```
sp {xxyyzz
   (state <s> ^att <v>
              ^operator <o>)
   (<o> ^name xxyyzz
-->
   (<s> ^operator <o> +)
   (<o> ^name dance)
}
```

Justification: By violating the intent of the Soar language you will likely achieve undesirable results. There is almost certainly a better way of doing what you are trying to do.

One thing at a time

Don't mix different problem space functions in the same rule except operator proposal and selection. It is ok to propose an operator and create a unary preference at the same time.

Example:

```
sp {xxyyzz
   (state <s> ^att <v>
              ^operator <o>)
   (<o> ^name xxyyzz
-->
   (<s> ^operator <o> +
        ^att <v> -)
   (<o> ^name dance)
```

Justification: By violating the intent of the Soar language you will likely achieve undesirable results. There is almost certainly a better way of doing what you are trying to do.