How to Create a Soar Simulation Environment

Round 1

19th North American Soar Workshop

May 20-23, 1999

Outline

- Tcl vs C code
- A few pointers on Tcl and Tk
- Using the RHS tcl function for simple graphical displays
- Soar I/O
- The TSI: Tcl-Soar Interface
- Overview of TankSoar

Using Tcl vs. C code

Tcl

+ No recompilation
+ Portable
+ Fast prototyping
+ Build on TSI
+ High-level language for creating graphics
- “close enough” might have to be good enough
- low level details can be difficult to add

C

+ Faster performance
+ Program exactly what you want
+ Interface to any graphics package
- requires recompiling
- not as easy to prototype
- longer development time
- less portable

Tips for Tcl/Tk

- Variables used by more than one *proc* need to be declared global wherever used.
- Use arrays to manage related variables.
- Tcl 8 includes [incr Tcl], oop extension.
- Remember to use update.
- Learn how to use default params and variable *args*.
- Steal what you can.
- For large projects, consider buying TclPro, a development/debugging suite from Scriptics.

Tcl /Tk References

- http://www.scriptics.com
- Tcl and Tk Demo programs in distribution.
- *Practical Programming in Tcl and Tk, 2nd edition*, Brent B. Welch. isbn 0-13-616830-2
- *Effective Tcl/Tk Programming*, Mark Harrison & Michael McLennan. isbn 0-201-63474-0
- *Graphical Applications with Tcl & Tk, 2nd edition*, Eric Foster-Johnson. isbn 1-55851-569-0

Soar's RHS tcl Function

- Provides a hook to invoke ANY Tcl proc from the RHS of productions.
- Recommended for simple graphical displays to monitor what Soar is thinking.
- Examples:
  - Soar 8-puzzle GUI demo (file eight-puzzle.tcl)
  - monitor.soar file in handouts

Sample code for RHS tcl function

```
sp {
    ……
    -->
    (tcl |MakeANote | <o> 1)
}
```

If <o> is bound to x, then the above produces the string “MakeANote x1” which will be executed in the Tcl interpreter.

```
sp {
    ……
    -->
    (write |The log of | <x> | is | (tcl |expr log(|<x>|)| ))
}
```

If <x> == 10, the above will write to output-strings-destination:
“The log of 10 is: 2.30258509299

How to create a simple display using the RHS tcl function

1. Write Tcl procs to create and manipulate widgets
   - add graphical objects to canvas widgets or top-level windows
   - assign a Tcl variable to the widget **-variable** *attribute*
2. Use the RHS **tcl** function in productions
   - invoke the procs for manipulating widgets using Soar variables as the arguments to the Tcl procs
   - set new values for variables that are bound to widgets
3. Be sure to call **update** to update the display
   - can include at end of each Tcl proc
   - or set a Soar **monitor** to update display at given point in cycle

Sample production using RHS tcl function

```
echo "\nLoading quake/monitor.soar"
sp {elaborate*quake*monitor
    (state <s> ^name quake)
    -->
    (<s> ^monitor yes)}

sp {quake*monitor*init-map
  (state <s> ^name quake
             ^monitor yes)
  -->
  (tcl init-map | | -3000 | | 3000 | | -3000 | | 3000 | | 3)}

sp {quake*monitor*fov-angle
  (state <s> ^name quake
            ^operator.name explore)
  -->
  (tcl fov-angle | | 90)}

sp {quake*monitor*fov-radius
  (state <s> ^name quake
             ^operator.name explore)
  -->
  (tcl fov-radius | | 2000)}

## Current Position
sp {quake*monitor*position
  (state <s> ^name quake
             ^monitor yes
             ^io.input-link.agent.origin <or>)
  (<or> ^x <x>  ^y <y>)
  -->
  (tcl fov-pos | | <x> | | <y>)
}
```

Using **monitor** cmd to update display

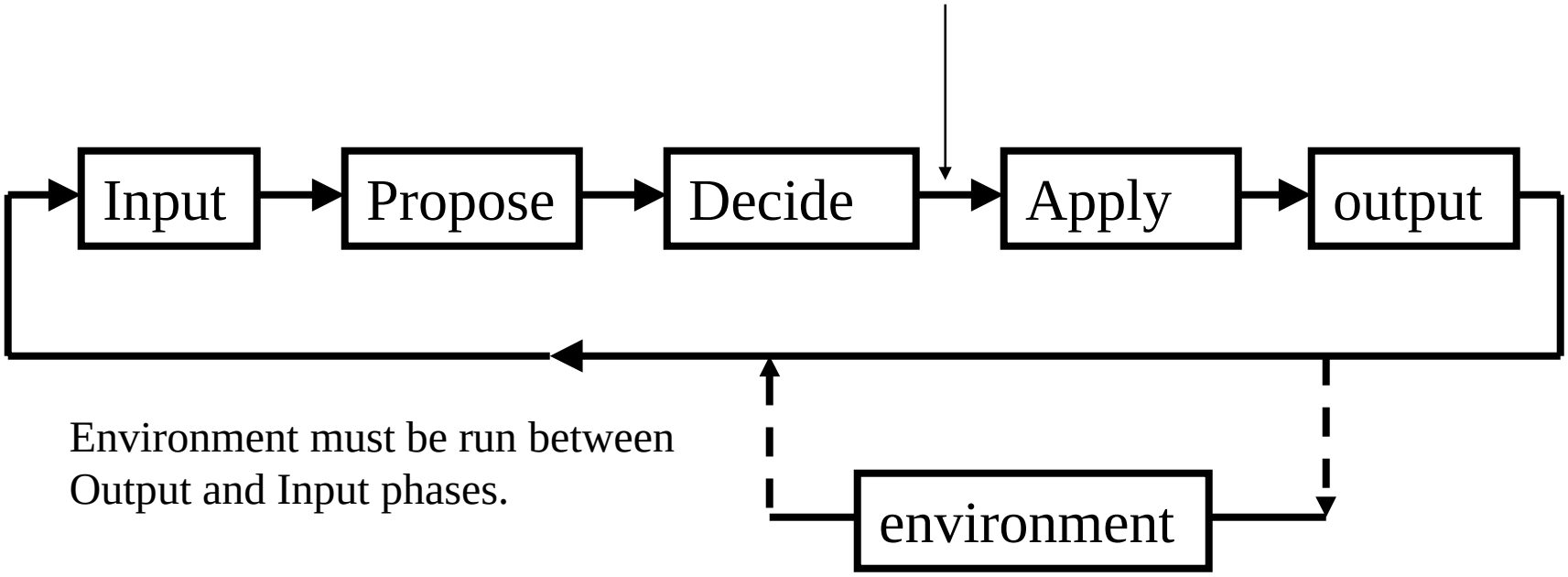
- **monitor** is a Soar command which allows users to register callbacks at various points in Soar's execution cycle. (help **monitor**)
- It's a command entered at the Soar prompt or included in a **source**'d file:

```
monitor -add after-decision-phase-cycle update
```

Soar's Decision Cycle

Can pause for user interface
by using a monitor, so can
inspect Soar's state and WM.

monitor -add after-decision-cycle-phase "stop-soar -self"



Environment must be run between
Output and Input phases.

Soar I/O

- Soar has specific structure for interacting with external environments.
- Created by architecture on top state.
  - `(S1 ^io I1)`
  - `(I1 ^input-link I2)`
  - `(I1 ^output-link I3)`
- User-defined input and output functions provide interface.

Input and Output Functions

- Input functions add working elements to `^io.input-link` structure using add-wme cmd. Also call remove-wme to remove old info.
- Output functions take data off the `^io.output-link` and process them for the environment.

Registering I/O Functions

- Input and output functions must be *registered* with Soar using the io command.

```
io -add -input  my-input-function.tcl  in1
io -add -output my-output-functions.tcl out1
```

- Input functions invoked during Input Phase
- Output functions invoked during Output Phase *only if output-link changes*.

Input Functions

- Input functions modify Soar's `^io.input-link` by calling `add-wme`.
- Input functions are passed one arg: `mode`
- First need to get identifiers used by Soar: It's UGLY code. Never write it. Always copy it from another application!

Output Functions

- Output functions process the data on Soar's `^io.output-link`
- Output functions are passed two args: `mode` and `outputs`.
- Again need to get identifiers for `output-link`; steal this code from examples.
- `Outputs` is a list of lists: list of wmes, which are lists themselves. `(id ^attr value)`

Use existing I/O code

- soar-io-using-tcl.tcl from Bob Wray
- Eaters
- TankSoar
- ask on soar-group mailing list

Soar and Tcl

- Soar is an extension for Tcl. It adds Soar capabilities to any Tcl interpreter into which it is *loaded*.
- Tcl is the User Interface to Soar. It parses commands and passes arguments to the Soar kernel, the C code that manages working memory, production memory, the decision procedure, etc. Soar commands *are* Tcl commands.

Tcl Interpreters

- Tcl is an interpreted scripting language.
- Tcl an “interpreter” or Tcl shell can create slave interpreters. Each interpreter has its own namespace.
- A new slave interpreter is created for each Soar agent.

The Tcl-Soar Interface, TSI

- The TSI is loaded into the “main” Tcl interpreter, defining procedures for creating slave interpreters and adding the Soar extension to those slave interps.
- Each Soar agent is in its own slave interp. The TSI can send commands to any or all Soar agents.
- The TSI creates an interaction window for each Soar interp, which is just a bunch Tk widgets and procs for user interaction.

Soar Command Input and Output

- Some Soar commands *print* their output and some commands *return* a value.
- Printing is usually done to an I/O channel: a pipe, or file, or widget procedure.
- The Soar command output-strings-destination redirects printed output to the appropriate widget, file or pipe. It is also used to *return* printed results as a value, which can be stored in a Tcl variable for later use.

Renaming Soar Commands

- When add an external environment, need to process the environment at the end of every Soar decision cycle.
- Need to intercept Soar commands: run, stop-soar, (others?) but don’t want a separate set of commands for each application.
- Rename:
  - run --> soar-run;   alias environmentRun -->run
  - stop-soar --> soar-stop-soar; environmentStop --> stop-soar

TSI Code examples

- CreateNewAgent
- ManageInputLink
- ManageOutputLink
- EnvironmentRun, EnvironmentStop

Creating a New Agent

```
proc createTank {x y agentPath agentFile newname} {
#>-=>
# Sets up an agent
# Input: x,y grid coordinates, file name to source and
the name of the agent (color)
#>-=>
global map tank current tankList sensor agentsetup \
         agentsetup2 soarTimeUnit actionTaken powerUpList
         sensorList sensorInfo ETCPConfig soarTimeUnit


# CODE EXCISED HERE: Do a bunch of error checking to
# make sure it's cool to create an agent with those
# parameters

createNewAgent $newname $agentPath "$agentFile.soar"
if {![interp exists $newname]} {
   return 0
}

#>-=>
# Create aliases for the new slave interpretor
#>-=>
$newname alias setUpInputLink setUpInputLink $newname
$newname alias updateInputLink updateInputLink $newname
$newname alias removeInputLink removeInputLink $newname
$newname alias setActionRequest setActionRequest $newname

$newname eval rename run run-soar
$newname alias run environmentRun
$newname alias step environmentStep
$newname alias stop environmentStop
#$newname alias stop-soar environmentStop
if $ETCPConfig(afterDecision) {
   $newname eval monitor -add \
     after-decision-phase-cycle \
     \"stop-soar -self \{\}\" dp1
}
$newname eval {.tsw.frame.step configure \
       -command {tsiDisplayAndSendCommand step}}
#>-=>
# Source the agent.soar code & specific behavior code
#>-=>
if [catch {$newname eval \
     [list uplevel #0 source agent.tcl]} msg] {
   tk_dialog .error Warning \
   "Couldn't load interface code: $msg" warning 0 Ok
}

#>-=>
# CODE EXCISED HERE Set up all your environment-side
# stuff
#>-=>

drawTank $newname
refreshWorld

$newname eval tsiAddProductionMenu .tsw
}
```

Setting up the Input and Output Link Callbacks

```
foreach i [io -list -input] {
   io -delete -input $i
}
io -add -input manageInputLink

foreach i [io -list -output] {
   io -delete -output $i
}
io -add -output manageOutputLink output-link
```

Managing the Input Link

```
proc manageInputLink {mode} {
   global ioID

   switch $mode {
      top-state-just-created {
          output-strings-destination -push -append-to-result
          set topStateID [lindex [wmes {(* ^superstate nil)}] 1]

          set ioID [lindex [wmes "($topStateID ^io *)"] 3]
          set ioID [string trimright $ioID ")"]

          set inID [lindex [wmes "($ioID ^input-link *)"] 3]
          set inID [string trimright $inID ")"]

          set outID [lindex [wmes "($ioID ^output-link *)"] 3]
          set outID [string trimright $outID ")"]

          output-strings-destination -pop
          setUpInputLink $ioID $inID $outID
      }
      normal-input-cycle {
         updateInputLink
      }
      top-state-just-removed {
         removeInputLink
      }
   }
}
```

Managing the Output Link

```
proc manageOutputLink {mode outputs} {
   global ioID outputLinkID sensorList sensorInfo outputList

   set outputList $outputs
   switch $mode {
      added-output-command {
         set outputLinkID [getOutputValue $ioID "output-link"]

         foreach currSensor $sensorList {
            set sensorDone 0
            while {!$sensorDone} {
               set outInfo($currSensor,ID) [getOutputValue $outputLinkID $currSensor]
               if {$outInfo($currSensor,ID) != ""} {
                  set outInfo($currSensor,value)   \
                     [getOutputValue $outInfo($currSensor,ID) $sensorInfo($currSensor,parameterName)]
                  set isComplete [getOutputValue $outInfo($currSensor,ID) status]
                  if {$outInfo($currSensor,value) == "" || $isComplete == "complete"} {
                  } else {
                     setActionRequest $sensorInfo($currSensor,actionName) \
                        $outInfo($currSensor,value) $outInfo($currSensor,ID)
                  }
               } else {
                  set sensorDone 1
               }
            }
         }
      }
      modified-output-command {

         SAME STUFF AS ABOVE ADDED-OUTPUT-COMMAND

      }
   }
}
```

```
set sensorList {move rotate radar fire radar-power shields}
set sensorInfo(move,actionName) moveDir
set sensorInfo(rotate,actionName) rotateDir
set sensorInfo(fire,actionName) fireWeapon
set sensorInfo(radar,actionName) activateRadar
set sensorInfo(radar-power,actionName) changeRadar
set sensorInfo(shields,actionName) activateShields
set sensorInfo(move,parameterName) direction
set sensorInfo(rotate,parameterName) direction
set sensorInfo(fire,parameterName) weapon
set sensorInfo(radar,parameterName) switch
set sensorInfo(radar-power,parameterName) setting
set sensorInfo(shields,parameterName) switch
```

The Run/Step/Stop Functions

```
proc environmentRun {args} {
global runningSimulation

   tsiOnEnvironmentRun
   if {$args != ""} {
      set n 1
      scan $args %d n
      for {set i 1} {$i <= $n} {incr i} {
         step
      }
      return
   }
   set runningSimulation 1
   runSimulation .wGlobal.map
}

proc environmentStep {} {
global runningSimulation

   tsiOnEnvironmentStep
   set runningSimulation 0
   runSimulation .wGlobal.map
}

proc environmentStop {} {
global runningSimulation

   if { $runningSimulation } {
      set runningSimulation 0
   }
   tsiOnEnvironmentStop
}
```

A Useful Utility for IO

```
proc getOutputValue {id attr} {
global outputList

   #>-=>
   # This function parses the items on the output link to determine
   # whether they match for a given command
   #>-=>

   foreach wme $outputList {
      if {(($id == "") || [string match $id [lindex $wme 0]]) && \
          (($attr == "") || [string match $attr [lindex $wme 1]])} {
         set outputList [ldelete $outputList $wme]
         return [lindex $wme 2]
      }
   }

   return ""

}
```

Setting up the Input Link Structure

```
proc setUpInputLink {whichAgent io_id inID outID} {
global tank tankList sensor topstate_id healthcharge energycharge turn wmeRecord agentsetup2 cycle

if {[info exists tank($whichAgent,inputlink)]} {
   return
}

# Record input and output ID for this agent
set tank($whichAgent,inputlink)  $inID
set tank($whichAgent,outputlink) $outID

# Set up the hierarchy for multi-level sensor objects

set incominginput [$whichAgent eval "add-wme $tank($whichAgent,inputlink) ^incoming *"]
set tank($whichAgent,incominglink) [lindex $incominginput 3]
set radarinput [$whichAgent eval "add-wme $tank($whichAgent,inputlink) ^radar *"]
set tank($whichAgent,radarLink) [lindex $radarinput 3]

#Initialize sensors directly on input link
initSensor $whichAgent $tank($whichAgent,energy) energy energyval inputlink
initSensor $whichAgent $tank($whichAgent,missiles) missiles missilesval inputlink
initSensor $whichAgent $tank($whichAgent,x) x xval inputlink
initSensor $whichAgent $tank($whichAgent,y) y yval inputlink
initSensor $whichAgent $tank($whichAgent,dir) direction dirval inputlink

#Initialize sensors directly on other links
initSensor $whichAgent $tank($whichAgent,Incomingforward) forward incomingupval incominglink
initSensor $whichAgent $tank($whichAgent,Incomingbackward) backward incomingdownval incominglink
initSensor $whichAgent $tank($whichAgent,Incomingleft) left incomingleftval incominglink
initSensor $whichAgent $tank($whichAgent,Incomingright) right incomingrightval incominglink

# Initialize the radar (vision)
radarToInputLink $whichAgent
}
```

Managing Percepts

```
proc initSensor {whichAgent sensorValue soarObjectName sensorField link} {
# This function adds a wme with attribute name ^soarObjectName and value sensorValue
# - It adds this information to the inputlink specified for a particular agent by its link attribute in
#   the tank array
# - for nested augmentations the link will be the parent soar object
# - It then adds entries to the sensor array, recording value and the wme ID
global tank sensor wmeRecord

scan [$whichAgent eval "add-wme $tank($whichAgent,$link) $soarObjectName $sensorValue"] "%d" wmeadded
set sensor($whichAgent,$sensorField) $sensorValue
set wmeRecord($whichAgent,$sensorField) $wmeadded
}

proc checkSensor {whichAgent sensorValue soarObjectName sensorField link} {
# The sensor array keeps track of the last io values set in soar
# - This function minimizes changing of wmes by changing them only when they are different than before
global sensor tank wmeRecord

if {$sensor($whichAgent,$sensorField) != $sensorValue} {
   $whichAgent eval "remove-wme $wmeRecord($whichAgent,$sensorField)"
   scan [$whichAgent eval "add-wme $tank($whichAgent,$link) $soarObjectName $sensorValue"] "%d" wmeadded
   set sensor($whichAgent,$sensorField) $sensorValue
   set wmeRecord($whichAgent,$sensorField) $wmeadded
}
}
proc replaceSensor {whichAgent sensorValue soarObjectName sensorField link} {
# This function replaces a wme w/o checking whether the sensor
# array has changed
global sensor tank wmeRecord

$whichAgent eval "remove-wme $wmeRecord($whichAgent,$sensorField)"
scan [$whichAgent eval "add-wme $tank($whichAgent,$link) $soarObjectName $sensorValue"] "%d" wmeadded
set wmeRecord($whichAgent,$sensorField) $wmeadded
}
```

Updating the Input Link

```
proc sensorsToInputLink {whichAgent} {
global tank agentsetup sensor healthcharge energycharge turn sensor wmeRecord actionTaken cycle

checkSensor $whichAgent $tank($whichAgent,energy) energy energyval inputlink
checkSensor $whichAgent $tank($whichAgent,missiles) missiles missilesval inputlink
checkSensor $whichAgent $tank($whichAgent,x) x xval inputlink
checkSensor $whichAgent $tank($whichAgent,y) y yval inputlink
checkSensor $whichAgent $tank($whichAgent,dir) direction dirval inputlink

checkSensor2 $whichAgent $tank($whichAgent,Incomingforward) forward incomingupval incominglink
checkSensor2 $whichAgent $tank($whichAgent,Incomingbackward) backward incomingdownval incominglink
checkSensor2 $whichAgent $tank($whichAgent,Incomingleft) left incomingleftval incominglink
checkSensor2 $whichAgent $tank($whichAgent,Incomingright) right incomingrightval incominglink

if {$sensor($whichAgent,radarval) != $tank($whichAgent,radarInfo)} {
   set templist [lsort -decreasing -integer $wmeRecord($whichAgent,radarwme)]
   foreach i $templist {
      $whichAgent eval "remove-wme $i"
   }
   radarToInputLink $whichAgent
}

}
```

Updating the World

```
proc updateworld {} {
global tankList tank cycle decisions_per_update randomorder \

set randomizedTankList [randList [llength $tankList]]
set randnames [list]
foreach n $randomizedTankList {
            lappend randnames [lindex $tankList $n]
}

if {[expr $cycle%$decisions_per_update] == 0} {
   incr turn 1
   foreach t $randnames {
      if {$tank($t,status) == "active"} {
         set tank($t,lastmove) none
         changeState $t ;# Processes output commands
      }
   }

   # CODE EXCISED HERE:  Update environments items (powerups, projectiles, kills)

   clearAllAgentSensors
   foreach t $randnames {
      if {$tank($t,status) == "active"} {
         updateState $t ;# Updates sensor info
      }
   }
   foreach t $randnames {
      if {$tank($t,status) == "active"} {
         updateSensors $t ;# Sends sensor info to input link
      }
   }
}
incr cycle 1
}
```

Running a Simulation Event Cycle

```
proc runSimulation {w} {

set agents [interp slaves]
set agents [ldelete $agents siu]

if {$agents == ""} {
   tk_dialog .error Warning "There are currently no
                        tanks to run." warning 0 Ok
   set runningSimulation 0
   return
   # May need to call tsiOnEnvironmentStop
}

if {$worldCount >= $worldCountLimit} {
   set runningSimulation 0
   tk_dialog .info {Game Over} \
    "Time limit reached. Winner is $winTank" info 0 Ok
   return
}

if {$skipSimTick} {
   set skipSimTick False
} else {
   incr worldCount
   tickSimulation $w
   update
}
```

```
[lindex $agents 0] eval [list run-soar $soarTimePerTick
$soarTimeUnit]
   if {[info exists runningSimulation] &&
$runningSimulation} {
      after $tickDelay runSimulation $w
   } else {
      environmentStop
   }
}

proc tickSimulation {w} {
global globalTick ticksPerMove ticksPerTankCycle tankList
tank

if [info exists globalTick] {
   set globalTick [expr ($globalTick + 1) % $ticksPerMove]
} else {
   set globalTick 1
}
# Code excised here:
# Making sure that a tank exists.  If not return.

## If we have completed a global tick cycle, move the
## tanks, otherwise just do animation
if !$globalTick {
   updateworld
}
}
```

TankSoar File Summary

agent.tcl	**i/o routines sourced into each agent's interpreter**
constants*.tcl	**These files set up constants and images**
environment.tcl	**environment event loop code**
et-controlpanel.tcl	**code to create customized tsi control panel for tanksoar and eaters**
initWindows.tcl	**initializes various windows**
main.tcl	**launches tsi and tanksoar**
tank.tcl	**code that creates/registers agent, lots of game-specific code**
tankworld_graphics.tcl	**tcl/tk display code**
tankworld_search.tcl	**some search code for generating sensors**
tankworld_sensors.tcl	**code to generate and update percept info**
tankworld_update.tcl	**code to update world based on output and trigger agent sensor updates**
tsi_display.tcl	**various functions that are used in the modified TSI control panel**
utilities.tcl	**various tcl-related or game-related utilities**
windowAgentInfo.tcl	**sets up agent’s percept viewing window**
windowConstants.tcl	**sets up constants for window placement, size**
windowGlobal.tcl	**sets up main map window**
windowManualControl.tcl	**sets up human control window**

This code is found in TankSoar25 /tcl_code.
The TSI is found in TankSoar25/soar_lib/tsi30

Additional Resources

- Demo file: soar-io-using-tcl.soar
- C extensions demo file
  ftp://ftp.eecs.umich.edu/people/kcoulter/Soar/c_ext.tar
- *Soar Advanced Applications Manual*
  http://ai.eecs.umich.edu/soar/docs.html (fall 99)

All need to be updated for Soar 8, but still useful.