Visual Soar User's Manual

Andrew Nuxoll

April 2023

version 1.03

Center for Integrated Cognition • IQMRI

Errors may be reported to John E. Laird <john.laird@cic.iqmri.org>

Contents

List of Figures

Chapter 1: Introduction

Overview

Visual Soar is an interactive development environment (IDE) created to support the creation of Soar agents. It was created 1999-2001 by John Bauman and Brad Jones, then undergraduate students at the University of Michigan. Extensions were made by Brian Harleton, and most recently Andrew Nuxoll.

Visual Soar contains tools that directly support Soar-related programming tasks, especially editing production source code and organizing source code into an operator hierarchy. Furthermore, you can use Visual Soar to define a datamap: a model for the structure of the working-memory elements. Visual Soar can verify that your productions are consistent with that model and use it to auto-complete attributes and values for you as you type. Visual Soar does not contain a built-in debugger but can cooperate with the Soar Java Debugger to help you debug your productions.

This manual assumes the reader has at least a passing familiarity with the syntax of Soar productions, the mechanics of Soar operators and the structure of Soar's working memory. The Soar Tutorial is a good introduction to this material. This tutorial, along with Soar-related downloads, documentation, FAQs, and announcements, as well as links to information about specific Soar research projects and researchers can be found on the Soar website:

http://soar.eecs.umich.edu

Installing and Running Visual Soar

Visual Soar is a Java application. As such, you will first need a Java interpreter installed on your computer to use Visual Soar. If you are uncertain whether Java is present, a quick test is to run this command in a Windows command prompt or Unix terminal (OS/X or Linux):

```
java -version
```

If present, you will see message with this format but the details are likely to be different:

```
java version "1.8.0_292”

Java(TM) SE Runtime Environment (build 1.8.0_292-b10)

Java HotSpot(TM) 64-Bit Server VM (build 25.292-b10, mixed mode)
```

In the above example, java is version 1.8. Visual Soar is targeted to run versions 1.7 or 1.8 but should work on any subsequent version as well.

If Java is not installed on your computer, you can find instructions here:

https://www.java.com/en/download/help/index_installing.xml

Visual Soar itself is provided as part of the Soar Suite package on the Soar website (see the URL above). Up to date instructions for downloading and installing Soar are provided on the site. Following these instructions will also download and install Visual Soar.

Once the Soar Suite is installed, the root folder of the installation will contain two files for running Visual Soar:

VisualSoar.bat ← For Windows users

VisualSoar.sh ← For Linux and OS/X users

Troubleshooting

When Visual Soar does not launch, the most frequent cause is an inability to find the needed .jar files. In this case you will typically see an error message that begins with one of the following:

Error: Unable to access jarfile

Error: Unable to initialize main class edu.umich.soar.visualsoar.VisualSoar

To resolve this, first verify the following:

- VisualSoar.jar is present in the bin subfolder of your root SoarSuite folder
- sml.jar is present in the bin/java subfolder of your root SoarSuite folder

If you are running Visual Soar in a command interpreter or terminal, make sure that your current working directory is the root SoarSuite folder.

If problems persist, join the soar-help mailing list and ask for help there:

https://sourceforge.net/projects/soar/lists/soar-help

Bugs and Feature Requests

Please do not hesitate to file bugs or request new features on our issue tracker:

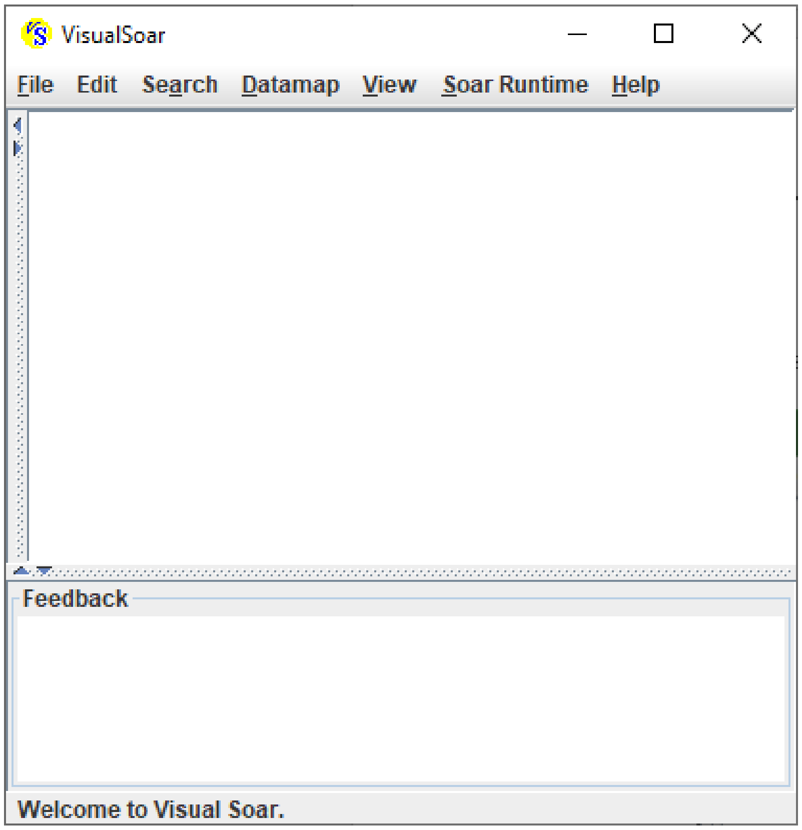
https://github.com/SoarGroup/Soar/issues

To avoid redundant entries, please search for duplicate issues first.

Chapter 2: User Interface Fundamentals

New Project

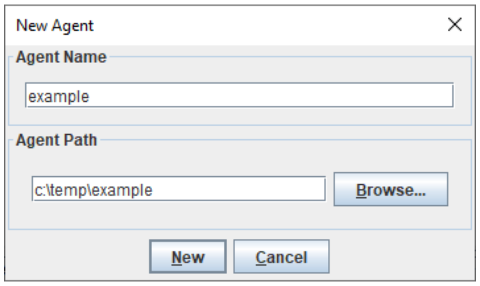
When first launched, Visual Soar's initial appearance is as shown in Figure 1.



Welcome to Visual Soar.

Figure : Visual Soar at startup with no project loaded

A Visual Soar project contains all the Soar source code for a single agent along with several support files. It occupies a particular folder and its sub-folders as well as the files those folders contain.



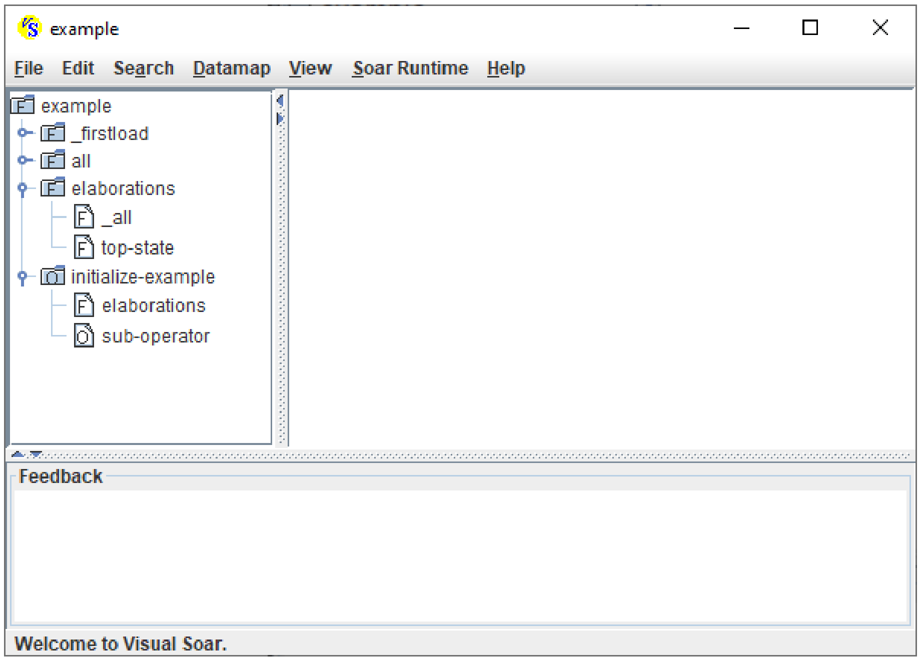
Agent Name: example

Agent Path: c:\temp\example

To demonstrate this, select File→New Project. In the dialog that appears (see Figure 2), provide a name for your project, then select to the folder where you want to store your project. Click the New button to create the project. Now that a project is loaded, the operator pane appears on the left (see Figure 3).

Operator Pane

The operator pane allows you to visualize and organize the operator hierarchy of your agent.

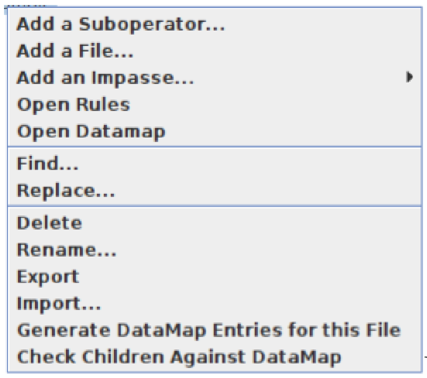


Welcome to Visual Soar.

Nodes in the file operator pane's tree are annotated with an icon so you can recognize their contents:

F	A file containing Soar elaboration productions and commands. These files can also be used to store large explanatory comments. In the file system, this is a single .soar file.
O	A low-level operator: a Soar operator that has no sub-operators. In other words, this is a "leaf" in the operator hierarchy tree. In the file system, this is a single .soar file.

📄	A file containing Soar elaboration productions and commands. These files can also be used to store large explanatory comments. In the file system, this is a single .soar file.
📁	A sub-folder that contains operator files, elaboration files and sub-folders that comprise the agent. In the file system, this is a folder.
📁	A high-level operator: a Soar operator that has sub-operators and its own local datamap. In the file system, this consists of a .soar file and a sub-folder with the same name.
📁	An impasse folder: a collection of Soar operators and/or elaborations designed to respond to an impasse at this level in the hierarchy. There are not separate operator no-change impasse folders. They are part of the high-level operator folders described above.



You can right click on any node in the operator pane to reveal a context menu for that node (see Figure 4).

The context menu allows you to edit the nodes in the hierarchy as detailed below. Not all options are available for all node types.

Add a Suboperator	Adds a new low-level operator to a node. If the node is a low-level operator, it becomes a high-level operator and a subfolder is created.
Add a File	Adds a new .soar file to the node.

Add a Suboperator	Adds a new low-level operator to a node. If the node is a low-level operator, it becomes a high-level operator and a subfolder is created.
Add an Impasse	Adds a new impasse folder to the hierarchy targeted at one of the Soar impasses: tie, conflict, constraint failure and state no change.
Open Rules	Creates a new rule editor window to edit the rules for this node. You can also just double click on the node to do this.
Open Datamap	Opens a datamap editor window for a high-level operator or the project root. (This option is disabled for other node types.)
Find/Replace	Find/replace any particular text in the subtree associated with this node.
Delete	Deletes this node and any sub-nodes from the hierarchy. The associated files are also removed.
Rename	Renames the node. Note: The code associated with the node is not modified and productions may have names that are now inconsistent with the node's name.
Export	Collects all the files associated with an operator (including any sub-operators) into a single file. This file is placed in the root of your project folder and has a .vse extension. The filename will match the node name. This file can then be imported into another Visual Soar project.
Import	Imports a .vse file into this project. The exported hierarchy is recreated as a suboperator of the current node.
Generate Datamap Entries for this File	Scans the productions in the file associated with this node for any attributes or values that are missing from the datamap. Any missing entries are added to the datamap as new unvalidated entries. See Chapter 3 for more about datamaps in Visual Soar.
Check Children Against Datamap	Scans the productions in each file in this folder or high-level operator for any attributes or values that are inconsistent with the datamap. See Chapter 3 for more about datamaps in Visual Soar.

Visual Soar Project File Structure

Figure 5 depicts how the operator pane in Figure 3 is reflected in the file and folder structure of the project in the file system. The files and folders **highlighted in boldface** map to the entries in operator pane's tree. When you double click on a node in the operator pane, the rule editor window that appears is for editing the associated .soar file in the project. Note that a high-level operator (e.g., initialize-operator in this example) has both an associated .soar file and a sub-folder as depicted by the dashed line.

A Visual Soar project contains other files that are not depicted in the operator pane:

- The .vsa file is the main configuration file for this project.
- The .soar file in the root that has the same name as the project (e.g., example.soar in the figure) is the main .soar file for the agent. When you source this file, you will source all other .soar files that comprise the agent. The files ending in _source.soar are in place to ensure this occurs seamlessly.
- The .dm files describe the project's datamap (discussed in Chapter 3). The comment.dm file is for datamap comments. The other .dm file has the same name as the project and describes all the WMEs in the datamap.

Certain .soar files and folders are always in a project by default:

- _firstload.soar is for source code you want to execute before any productions are loaded. This is typically used for configuration commands like: learn --on or chunk always.
- The all folder is for operators that don't belong in any particular place in the operator hierarchy because they can be proposed in multiple states.
- The elaborations folder is for elaborations that relate to the top-state, fire in multiple states, or otherwise don't belong in any particular place in the operator hierarchy.

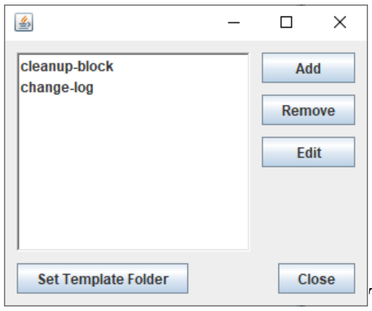
While is it generally safe to edit the .soar files in a Visual Soar project directly (i.e., via a text editor), it is not recommended that you modify other files or folders, rename any files or folders or rearrange the file locations.

Rule Editor

When you double click on a node in operator pane, Visual Soar creates a new rule editor window that contains the contents of the associated .soar file (see Figure 5). The rule editor provides appropriate syntax highlighting and auto-justifying of Soar source code.

Each rule editor window provides a basic set of code editing functions via its menu and associated hotkeys. Many of these are canonical and need no particular explanation here. The remainder are discussed below:

Edit → Comment Out	This function comments out the currently highlighted code as you expect. It is also a toggle. If the currently highlighted code is already commented out, then it is uncommented instead.
Soar → Check Productions Against Datamap	Scans all the productions in this file for expressions that are inconsistent with the datamap. See Chapter 3 for an explanation of datamaps and how they are used.
Soar → Soar Complete	This function finds all attributes/values that could be typed at the cursor's current position. This is done by consulting the datamap. If there is only one completion possible, it is immediately inserted for you. If there are multiple possible completions they are listed on the status bar. This function is typically accessed via its hotkey: Ctrl+Enter. Notably, Soar Complete runs automatically when you type a period (dot) character in a Soar production. However, in this circumstance it will not auto-insert a sole completion.
Insert Template	The first set of options in this menu will insert a partially complete production for a different general function as described. Each template supplies an appropriate name and some typical conditions. If you have any custom templates (see below) they are listed in the bottom of this menu.

Edit → Comment Out	This function comments out the currently highlighted code as you expect. It is also a toggle. If the currently highlighted code is already commented out, then it is uncommented instead.
Insert Template → Edit Custom Templates…	 cleanup-block change-log Add Remove Edit Set Template Folder Close This option allows you to create your own templates to use in Visual Soar. When selected, the dialog shown at right appears. The buttons function as follows: • Add – creates a new custom template which is immediately opened for you to edit. A default template content is provided. • Remove – deletes the selected template. • Edit – opens the template to be edited in a rule editor window. • Set Template Folder – By default, Visual Soar searches in the .java subfolder of your user folder for custom templates. You can change the folder via this button. The content of a template can be any text you wish including Soar code, comments and other items. You can also place macros in the code that will be automatically substituted with a context specific text (e.g., the current date or the name of the operator file that this template was inserted into). See the default template content for more details.
Runtime	This menu allows you to send commands to an agent running in the Soar Debugger. In cases where the command is associated with a specific production, a function uses the production nearest to the typing cursor. See Chapter 4 for more information about Visual Soar's runtime interface to the Soar Debugger.

Main Menu

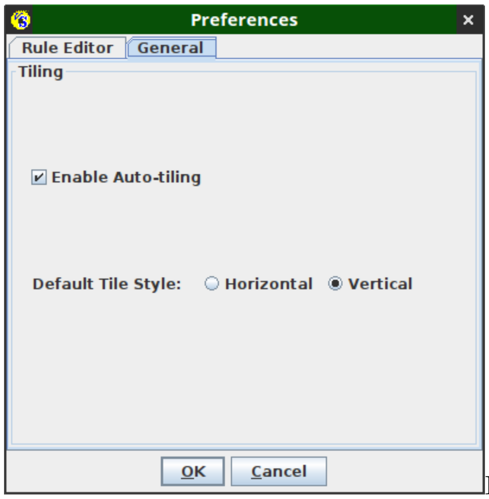
The main menu appears at the top of the Visual Soar window as depicted in Figure 1. The functions provided by the menu items are discussed in the tables below

File → New Project	This function creates a new project. See beginning of this chapter for an example.
File → Open Project	This function is used to load an existing Visual Soar project. You will be prompted to select the .vsa file associated with the project. If you open a project while another project is open, the latter will be closed. You cannot have more than one project open at once in Visual Soar, although you can have multiple instances of Visual Soar running at once.
File → Open File	Allows you to open any text file in a Visual Soar rule editor. This file need not be part of the project and does not need to have a .soar extension and need not contain Soar code.
File → Save	Saves all modified files in the project. The project's .vsa file is updated as well.
File → Save Project As	Creates a copy of the currently open project in a new folder. The original project is closed and the new copy is now open. Subsequent edits to any open files will apply to the copy.
File → Exit	Close Visual Soar

Edit->Preferences	See the Preferences Dialog section below for details on this dialog. Note that typical edit functionality (e.g., copy, paste, undo) is in the Edit menu of each individual rule editor or datamap editor window.
Search	This menu provides find and replace functionality for all the source code in the project. You can also ask Visual Soar to list all productions in the project.
Datamap	The datamap functionality is detailed in Chapter 3
View	Allows you to select a particular editor window as the current active window. There are also options for tiling or cascading the open editor windows.
Soar Runtime	The debugging functionality is detailed in Chapter 4
Help	This menu provides information about Visual Soar's creators and links to online resources for assistance with using Visual Soar.

Preferences Dialog

Selecting Edit→ Preferences from Visual Soar's main menu displays its preferences dialog. Use this dialog to configure Visual Soar's appearance and behavior.



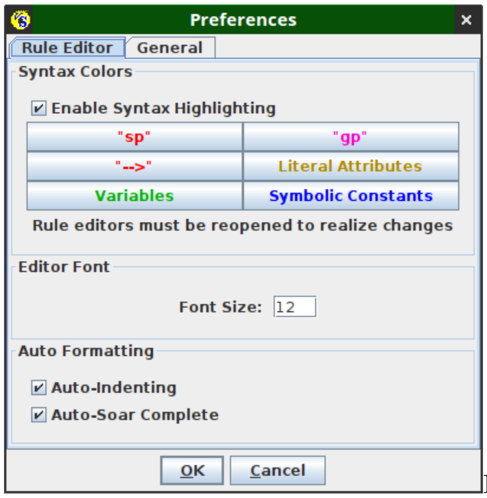
Tiling

☑ Enable Auto-tiling

Default Tile Style: ○ Horizontal ● Vertical

Figure 6 depicts the General tab on the preferences dialog. The contents of this tab are described in the table below:

Enable Auto-tiling	If this box is selected (default) then each time a new editor window is added or an existing editor window is removed, all editor windows are re-tiled to use a fair share of all existing space.
Default Tile Style	A vertical tile style (default) causes Visual Soar to prefer to divide the window area vertically before horizontally. The horizontal option does the opposite.



Syntax Colors

☑ Enable Syntax Highlighting

Rule editors must be reopened to realize changes

Editor Font

Font Size: 12

Auto Formatting

☑ Auto-Indenting

☑ Auto-Soar Complete

Figure 7 depicts the Rule Editor tab on the preferences dialog. The controls on this tab affect the behavior of the rule editor windows. Details are in the table below:

Enable Syntax Highlighting	Use this check box to disable all syntax highlighting.
Syntax Highlighting Buttons	The dialog presents six buttons for selecting the colors used for syntax highlighting. To change a particular color, click a button and select a new color.
Editor Font	Increase or decrease the font size used in rule editor windows.
Auto-Formatting	Use this section to disable auto-indenting and auto-Soar Complete functionality on all rule editor windows.

Note: Not all preference choices made in this dialog are saved; they must be reconfigured each time you run Visual Soar. A planned future enhancement is to improve Visual Soar's ability to remember preferences across sessions.

Visual Soar Command Line

Typically, you will run Visual Soar using the VisualSoar.bat or VisualSoar.sh file in your Soar Suite folder where Soar has been installed. However, in certain contexts you may wish to run Visual Soar directly in a command window (Windows) or terminal window (Unix or OS/X). The base command is:

```
java -jar VisualSoar.jar
```

If this command does not work as is, some troubleshooting may be necessary:

- For the above command to work, VisualSoar.jar should be in the current working directory. If it is not, you may need to specify full path to the file. If you are running Linux or OS/X you may need to specify a relative path (e.g., ./VisualSoar.jar)
- Visual Soar uses sml.jar and will not run without it. This file is provided with your SoarSuite distribution in the bin/java sub-folder. It should be in your CLASSPATH. See https://docs.oracle.com/javase/tutorial/essential/environment/paths.html
- Visual Soar uses swt.jar for runtime communication with the Soar Java Debugger. If you wish to use this functionality, swt.jar must also be in the bin/java sub-folder.

When running Visual Soar, you can specify the path to a .vsa file on the command line to automatically open the associated project.

Chapter 3: Datamaps

Visual Soar has a built-in parser to scan your agent's productions for syntax errors. This same parser is used to provide syntax highlighting for your Soar source code. However, syntactically perfect Soar source code is no guarantee of its semantic fitness. In particular, due to programmer error, Soar productions may test for attributes and values that can't possibly occur in working memory when the agent operates in its intended environment. Visual Soar uses a tool called a datamap to help catch these errors early and avoid (some) debugging headaches.

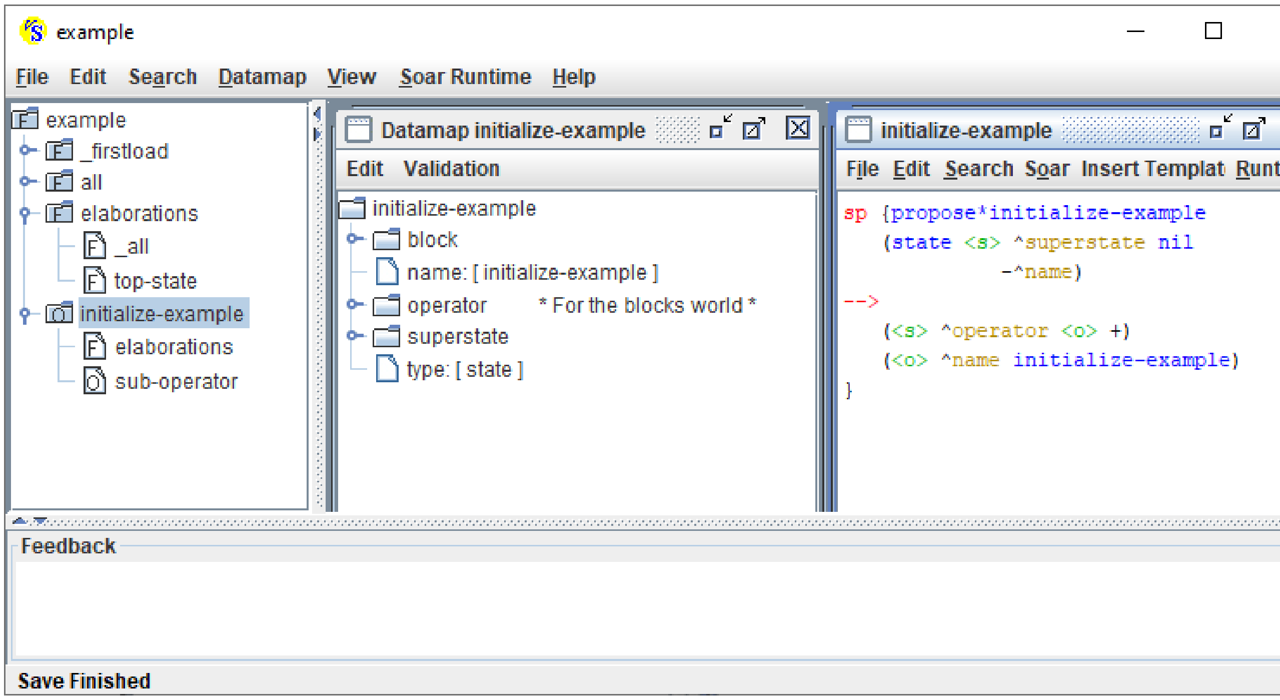
Loading a Datamap Editor Window

Every high-level operator in a Visual Soar project has a datamap associated with the sub-state that it creates. A datamap describes the superset of all elements that can be in working memory in that state. As you write productions designed to fire in this sub-state you should create additional datamap entries to reflect the WMEs that will be created by those productions in the sub-state.

When a high-level operator is first created – by adding a sub-operator to an existing operator – a datamap for the new associated sub-state is automatically created by Visual Soar. Visual Soar will also automatically insert some new entries into the datamap that reflect WMEs that are created by Soar. Specifically, the ˆname, ˆsuperstate and ˆoperator.name WMEs are added. In addition, Visual Soar will add an entry for the ˆtop-state elaboration that is created by most agents. See Figure 8 for an illustration of this.

Note that, due to the ˆsuperstate attributes in Soar, any sub-state's datamap also contains its parent state's datamap.

To view a datamap for a particular state, right click on the associated operator node in the operator pane and select Open Datamap. This displays a datamap editor window for that datamap. Figure 8 depicts a simple operator's source code and the associated datamap editor window for the sub-state that operator creates.

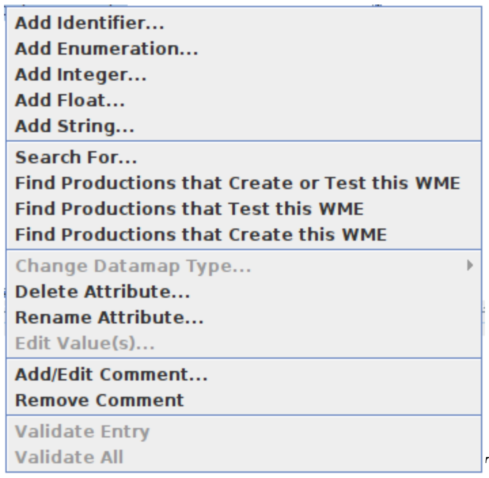


```
sp {propose*initialize-example
   (state <s> ^superstate nil
             -^name)
-->
   (<s> ^operator <o> +)
   (<o> ^name initialize-example)
}
```

Figure : A simple operator and the datamap for the substate it creates

You can access the datamap for the agent's top-state via the root note of the operator pane. Alternatively, select Datamap → Display Top-State Datamap from the main menu.

Datamap Tree

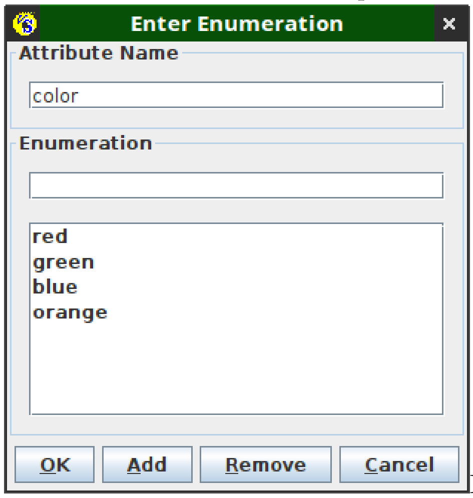


The datamap editor uses a tree to depict the possible contents of working memory. Since Soar's working memory is a graph, not a tree, this can create complications. Particularly, it is possible to expand this tree indefinitely via certain paths as the same datamap entries repeat themselves over and over. For example, the top-state attribute that is initialized by default has this behavior.

Editing a Datamap

Datamap editing functions are accessible via a context menu. Right click on any node in the datamap tree to access this menu (see Figure 9). These functions are divided into five sections. The first, third and fourth sections deal with editing the datamap.

The functions in the first section are used to add new attributes (nodes) to the datamap. They are described in the table below:

Add Identifier	This function adds a new attribute to the selected node whose value will be an identifier in Soar. Example: (<s> ^block <b1>)
Add Enumeration	This function adds a new attribute to the selected node whose value must be one of a set of specific strings (see Figure 10).  Enter Enumeration Attribute Name color Enumeration red green blue orange Example: (<b1> ^color red)
Add Integer	This function adds a new attribute to the selected node whose value is an integer. The range of valid values for the integer can also be specified. Example: (<b1> ^size 17)
Add Float	This function adds a new attribute to the selected node whose value is a decimal number. Similar to integers, you can optionally specify a range of valid values for a float. Example: (<b1> ^offset 0.113)

Add Identifier	This function adds a new attribute to the selected node whose value will be an identifier in Soar. Example: (<s> ˆblock <b1>)
Add String	This function adds a new attribute to the selected node whose value is a string with an unknown or unlimited set of possible values. Example: (<b1> ˆcreator George)

The third section of the datamap editor context menu contains functions for editing existing nodes in the datamap. These are self-explanatory and need no further explanation here.

The fourth section of the datamap editor context menu allows you to add, edit and delete comments on datamap nodes. Just as with source code comments, these allow you to leave notes for humans to help them understand the contents of the datamap. Comments appear in the tree next to their associated node and are delimited by asterisks.

Additional Notes on Datamap Editing:

- In addition to the context menu functions, you can move particular entries in the datamap around via drag and drop.
- Sometimes attributes have mixed-type values (e.g., either integer or string). You can simulate that by creating more than one attribute-value with the same attribute name but different value types.

Datamap Links

To support working memory's graph structure, the value of a datamap identifier can be a link to another section of the datamap. This is best illustrated by the ˆsuperstate link that Visual Soar adds for you. Expanding that link will show you all the entries in the parent state. Modifying these entries will modify the parent state's datamap (i.e., they are the same entries).

To create your own datamap links, hold down the Ctrl and Shift keys while dragging an existing datamap entry to a new location. See the *Example Datamap* section below for an example of this.

While you can also copy a datamap entry, creating links in this manner is almost always preferable to a copy. With a copy any change you make must be made to both copies to avoid datamap inconsistencies.

Searching the Datamap

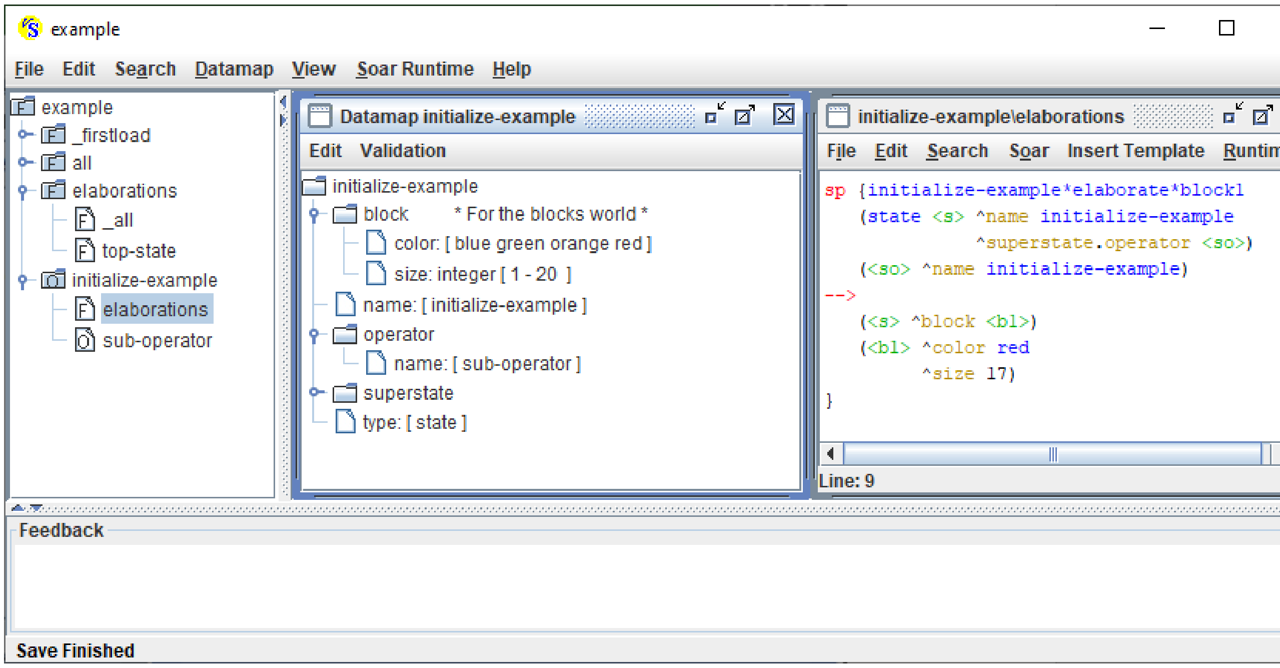
The second section of the context menu allows you to search for particular nodes in the datamap:

Search	Allows you to search all descendants of the current node for an attribute with a particular name. You must specify the full name rather than a partial match or regular expression; the search functionality does not perform partial matching, Furthermore, you cannot use this function to search for values (e.g., “red” as the value of the “color” enumeration).
Find Production That. . .	The three “Find Productions that” functions can be invaluable for keeping your datamap up to date by locating and excising orphan nodes.

Notably, the Soar Complete functionality in Visual Soar is also a form of datamap search.

Example Datamap

Figure 11 shows a new production added to the initialize-example sub-state from Figure 8. This production creates a new Soar object of type block that has a color and a size. The datamap on the right-hand side of the figure matches the production. Specifically, the block identifier has been added to the tree. It has an enumeration for color and an integer for the size which has a ranged of [1..20] defined. The block identifier also has a comment attached to it (“For the blocks world”).


example
File Edit Search Datamap View Soar Runtime Help
example
_firstload
all
elaborations
_all
top-state
initialize-example
elaborations
sub-operator
Datamap initialize-example
Edit Validation
initialize-example\elaborations
File Edit Search Soar Insert Template Runtim
Feedback
Save Finished

```
initialize-example
    block        * For the blocks world *
        color: [ blue green orange red ]
        size: integer [ 1 - 20 ]
    name: [ initialize-example ]
    operator
        name: [ sub-operator ]
    superstate
    type: [ state ]
```

```
sp {initialize-example*elaborate*blockl
   (state <s> ^name initialize-example
              ^superstate.operator <so>)
   (<so> ^name initialize-example)
-->
   (<s> ^block <bl>)
   (<bl> ^color red
         ^size 17)
}
```

Line: 9

Figure : Example datamap modification

Checking Production Against the Datamap

The most frequent use of a datamap is to check for places where a production does not match the datamap content. These errors take the following forms:

- The production creates or tests an attribute that does not exist in the datamap.
- The production creates an integer or float value that is out of range.
- The production creates a string or enumeration value that is not included in the corresponding enumeration.

To check for datamap errors in a particular file, use the Soar→ Check Productions Against Datamap menu option in the rule editor window for that file (or press F7).

To check for datamap errors in a particular subtree of the operator hierarchy, right click on the root node in the operator pane and select Check Children Against Datamap in the context menu.

To check all the files in your project, use the Datamap→ Check All Productions Against the Datamap option in the main menu.

Notes:

- Before performing any datamap checks, Visual Soar will verify there are no syntax errors in the code and abort if any are found. You can also manually check your entire project for syntax errors via the Datamap→ Check All Productions for Syntax Errors function in the main menu.
- While Soar will accept conditions in a production in any order, Visual Soar relies upon a WME's id being created before it is used. For example, this production is syntactically legal but will result in a datamap error in Visual Soar:

When performing these checks, any errors found will appear in the feedback pane. You can double click on any error to be taken directly to the corresponding line of code.

Aside from production errors, other datamap inconsistencies can appear. These are not necessarily errors. The Datamap section of the main menu has an array of five "Search the Datamap for WMEs that..." options to search for these issues so you can address them.

Creating a New Datamap

Ideally, you would add entries to a datamap before you write any production that test corresponding WMEs. In reality, this is not always practical or convenient. As a result, Visual Soar has built-in support for situations where new productions exist, but they have no datamap support.

To automatically add new datamap entries for the existing productions in a file, right click on the corresponding node in the operator pane and select Generate Datamap Entries for this File. Alternatively, you can do this for the entire project via the main menu: Datamap → Generate the Datamap from the Current Operator Hierarchy.

Generated datamap entries are considered unverified. They appear in green font in the datamap. The expectation is that a human should review these entries and correct them. For example, if a datamap entry is generated from (<b1> ^creator John) it would be an enumeration when, instead, it should be a String. Similarly, integer and float value may have a limited range that needs to be specified. Another common issue is that an identifier should be a link to an existing node in the datamap so that they share the same underlying datamap. Visual Soar is not able to recognize this and will create independent structures.

Once you are ready to verify a generated datamap entry, right click on it and select Validate Entry from the context menu.

You can also validate all the unvalidated entries in a datamap. This function can be accessed in two ways:

- selecting Validate All in the datamap's context menu. (Use with caution!)
- selecting Validation→ Validate Datamap from the datamap editor's menu
- editing the entry in any way (e.g., renaming or specifying a range) also caused to be validated

In some cases, you may accidentally add unvalidated entries for your code. In this case, it may be convenient to select Validation→ Remove NonValidated from the datamap editor's menu.

Example: Datamap Generation and Validation

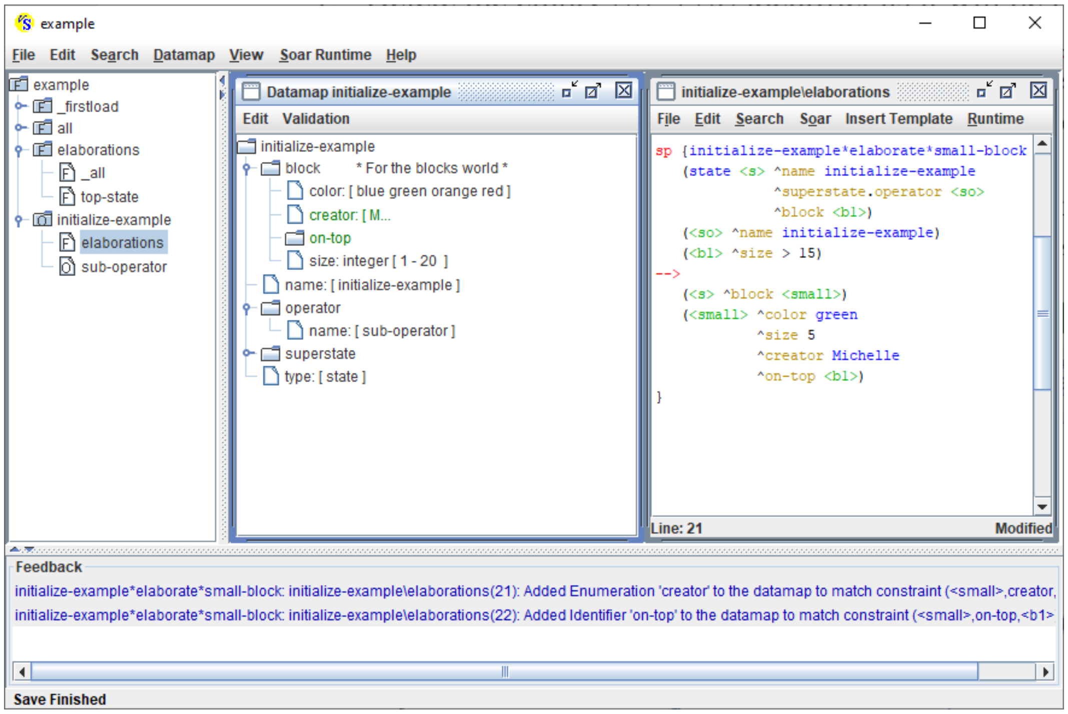
Figure 12 presents a small example of generating datamap entries. The second production in the rule editor (initialize-example*elaborate*small-block) creates a new block object on its right-hand-side. The block attribute and two of its constituent WMEs (color and size) are already consistent with the datamap as was shown in Figure. But the creator and on-top attributes are new. The user has asked Visual Soar to generate new entries for these WMEs and they now appear in the datamap in green font.

In this example, the user may decide that creator should be a string rather than an enumeration. So, she right clicks on that node and selects Change Datamap Type → to String. This change also auto-validates the entry.

The on-top attribute should be a reference to a block object. The computer can't infer that, so it's up to the human to adjust it. She follows these steps to create a recursive reference:

1. Delete the empty on-top identifier as it will be replaced.

2. Selects the block identifier and use Ctrl+Shift+drag to drag it to another (temporary) location in the datamap. This creates a link to the block in the new location. Since you cannot create a directly recursive link, it has to be temporarily placed in this new location.
3. Renames the new identifier link to on-top.
4. Drag the copy back inside the block and drops it there
5. Re-run the datamap check to verify that all is working.


example
File Edit Search Datamap View Soar Runtime Help
Datamap initialize-example
Edit Validation
initialize-example\elaborations
File Edit Search Soar Insert Template Runtime
Line: 21
Modified
Feedback
Save Finished

```
example
  _firstload
  all
  elaborations
    _all
    top-state
  initialize-example
    elaborations
    sub-operator
```

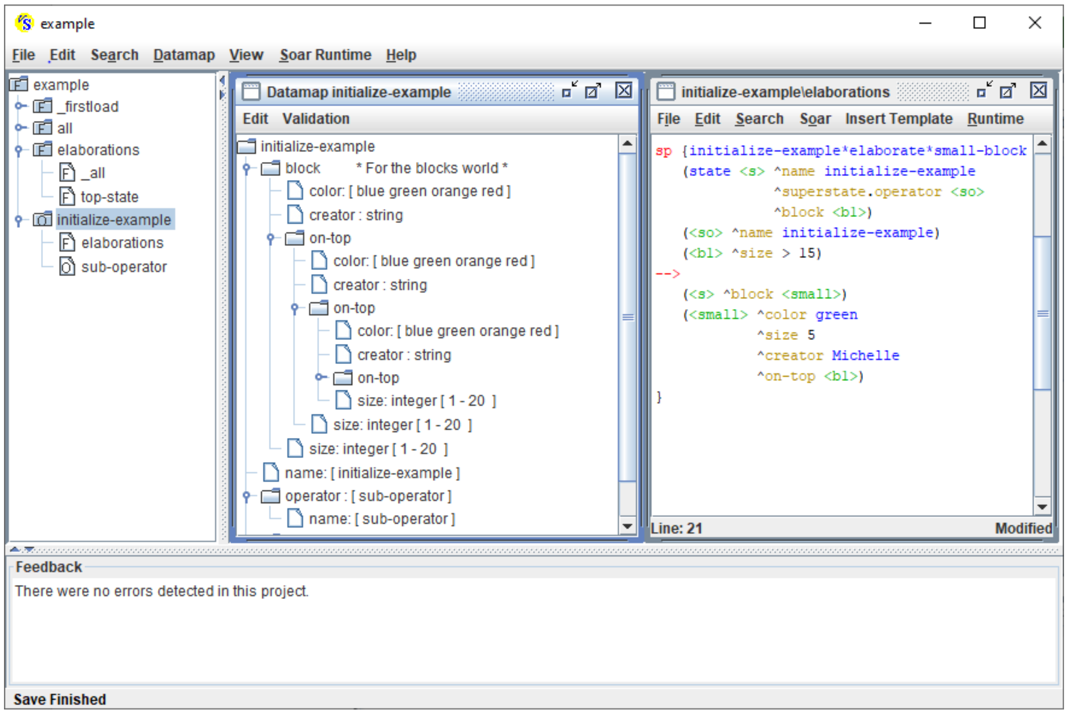
```
initialize-example
  block        * For the blocks world *
    color: [ blue green orange red ]
    creator: [ M...
    on-top
    size: integer [ 1 - 20 ]
  name: [ initialize-example ]
  operator
    name: [ sub-operator ]
  superstate
  type: [ state ]
```

```
sp {initialize-example*elaborate*small-block
   (state <s> ^name initialize-example
              ^superstate.operator <so>
              ^block <bl>)
   (<so> ^name initialize-example)
   (<bl> ^size > 15)
-->
   (<s> ^block <small>)
   (<small> ^color green
            ^size 5
            ^creator Michelle
            ^on-top <bl>)
}
```

```
initialize-example*elaborate*small-block: initialize-example\elaborations(21): Added Enumeration 'creator' to the datamap to match constraint (<small>,creator,
initialize-example*elaborate*small-block: initialize-example\elaborations(22): Added Identifier 'on-top' to the datamap to match constraint (<small>,on-top,<b1>
```

Figure : Example of generating datamap entries

Figure 13 shows the result of the modifications. In this figure, the recursive on-top link has been expanded multiple times to show its nature.


example
File Edit Search Datamap View Soar Runtime Help
example
_firstload
all
elaborations
_all
top-state
initialize-example
elaborations
sub-operator
Datamap initialize-example
Edit Validation
initialize-example
block * For the blocks world *
color: [blue green orange red]
creator : string
on-top
color: [blue green orange red]
creator : string
on-top
color: [blue green orange red]
creator : string
on-top
size: integer [1 - 20]
size: integer [1 - 20]
size: integer [1 - 20]
name: [initialize-example]
operator : [sub-operator]
name: [sub-operator]
initialize-example\elaborations
File Edit Search Soar Insert Template Runtime
Line: 21
Modified
Feedback
Save Finished

```
sp {initialize-example*elaborate*small-block
   (state <s> ^name initialize-example
              ^superstate.operator <so>
              ^block <bl>)
   (<so> ^name initialize-example)
   (<bl> ^size > 15)
-->
   (<s> ^block <small>)
   (<small> ^color green
            ^size 5
            ^creator Michelle
            ^on-top <bl>)
}
```

There were no errors detected in this project.

Figure : Result of validating datamap entries

Chapter 4: Debugger Interface

Visual Soar can connect to a running Soar Java Debugger process to send and receive data as the agent is running. This chapter discusses how to use this interface.

Connecting

First, Visual Soar must connect to the debugger. Follow these steps:

1. Both Visual Soar and the Soar Java Debugger must be running.
2. Select Soar Runtime → Connect from Visual Soar's main menu.

This connection succeeds silently. If it was successful, the only indication of success is that the remaining options in the Soar Runtime menu are no longer disabled (see Figure 14). If the connection fails you will typically receive an error message. See the Troubleshooting sections below.

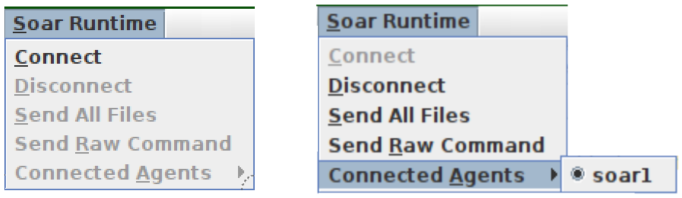
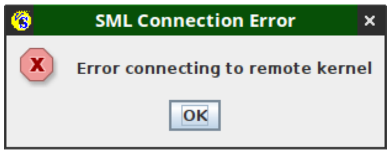


Figure : Change in Soar Runtime Menu after Connect

Troubleshooting: Error connecting to remove kernel

When connection fails, you may see this message:



SML Connection Error

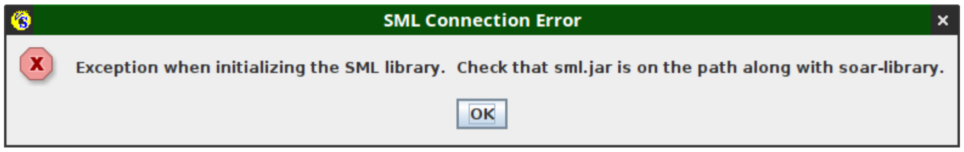
Error connecting to remote kernel

OK

This indicates that the Soar Java Debugger is not responding. Verify that your debugger is running. Then verify that any firewall software you have installed is not interfering with local connections between apps.

Troubleshooting: Exception when initializing the SML library

When connection fails, you may see this message:



SML Connection Error

Exception when initializing the SML library. Check that sml.jar is on the path along with soar-library.

OK

Certain files need to be present and found by Visual Soar for the connection to begin. This message occurs because Visual Soar cannot find one of more of these files. You should verify:

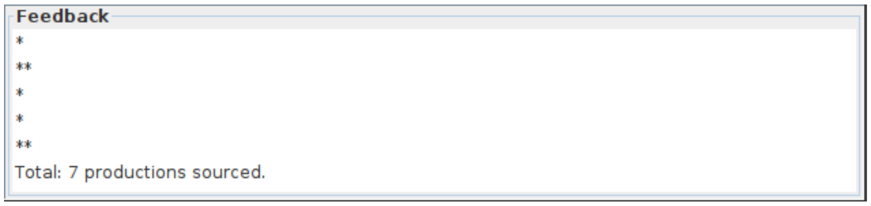
- The file sml.jar exists and is located in the same folder as swt.jar. This will typically be the bin/java sub-folder of your SoarSuite folder.
- Windows:
  - Your SoarSuite folder should contain the bin/win64 sub-folder. This folder should contain the following DLLs: Java_sml_ClientInterface.dll and Soar.dll
  - Your bin/win64 sub-folder should be in the PATH
- Linux and OS/X:

- Your SoarSuite folder should contain the bin sub-folder. This folder should contain the following shared libraries: libJava_sml_ClientInterface.so and libSoar.so.
- If the files are present and the error still occurs, consider how you are running Visual Soar. If you are using the VisualSoar.sh script, it should be providing a flag on the command line to the Java VM that looks like this: -Djava.library.path="$SOAR_HOME" Verify that $SOAR_HOME is properly set and then copy the .so files to this folder.

Using the Interface

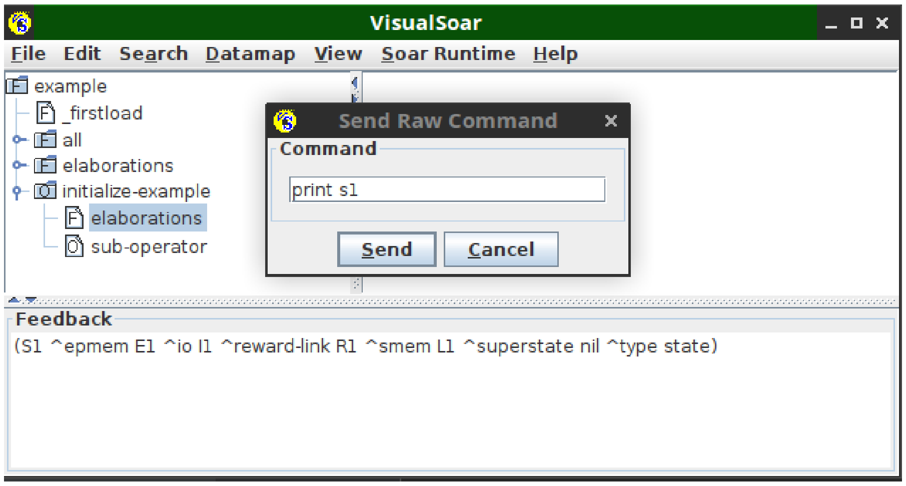
Once connected, all communication is via the Soar Runtime menu as described below:

- Soar Runtime → Disconnect closes the connection with the Soar Java Debugger
- Soar Runtime → Send All Files (re)loads all the productions for the agent in the current VisualSoar project into the debugger. Both the debugger's console and Visual Soar's feedback window will show the results of this.



```
*
**
*
*
**
Total: 7 productions sourced.
```

- Soar Runtime → Send Raw Command issues a command to the agent. Any command that can be issued in the debugger's console can be sent from Visual Soar via this option. The output of the commend is displayed in Visual Soar's feedback window.



```
(S1 ^epmem E1 ^io I1 ^reward-link R1 ^smem L1 ^superstate nil ^type state)
```

- Soar Runtime → Connected Agents switches which agent Visual Soar is connected to. The Soar Debugger may be monitoring multiple agents (see the Agents menu in the debugger's main menu). This option allows you to switch which agent Visual Soar is interacting with.

Index