Cadence SKILL IDE User Guide

Product Version IC23.1
August 2023

Contents

3
Program Execution Control 47

B
SKILL Lint

C
SKILL Lint Rules

1

SKILL Integrated Development Environment

Virtuoso® SKILL IDE is a tool that helps you develop, test, and refine SKILL programs. It is an extension of the basic SKILL Debugger and provides an interactive interface to it.

The core of SKILL IDE is a multi-file that provides common editing and debugging capabilities. These capabilities include, auto indenting, auto completion of function names, syntax highlighting, single stepping through SKILL programs (that is, executing the program statements one by one), setting up and stopping at breakpoints, saving and loading the debugging information, tracing and editing the values of variables during program execution, and displaying variable and stack trace.

You can load and debug only SKILL and SKILL++ programs in SKILL IDE. Listed below are some of the features of SKILL IDE.

■ Source-level debugging of SKILL programs

The code that you debug through SKILL IDE is reusable across all Virtuoso tools. The basic debugging capabilities of SKILL IDE include:

❑ **Configurable Breakpoints**: This feature enables you to set conditional breakpoints, which are triggered when the specified conditions are met. By setting these breakpoints, you can decide where the program execution pauses, so that you can examine the value of your program variables.

❑ **Saving and Reusing the Debugging Information**: SKILL IDE supports exporting of debugging information to external SKILL files. The exported debug information includes current line and function breakpoints, their conditions, traced functions, variables, and properties. You can later reuse the debug settings by loading the debug information from a previously saved file.

❑ **Variable and Function Tracing**: Use this feature to trace the values of functions and variables as they change during program execution. You can also track function calls and view functions that are on the top of the calls stack. Once selected for tracing, these functions, variables, their values, and scope are listed in the Trace assistant. If the value of a function or variable changes during program execution, it

can be traced quickly using the Trace assistant. You can directly change the value of the traced variables either in the Trace assistant or in the source code pane. The

❑ **Stack Tracing**: Use the stack tracing feature to display a list of functions and their arguments, in the order in which they are executed in the code. The *Stack* assistant displays the list of functions that have been called up to the current execution point. By looking at the stack trace, you can determine the function that was called when the error occurred and jump to the exact location in the code.

❑ **Code Browsing**: Use the *Class Browser*, *Method Browser*, and *Code Browser* assistants of SKILL IDE to browse the classes, generic functions, and user-defined functions of your SKILL code.

❑ **Lint Integration**: The SKILL Lint tool is fully integrated into SKILL IDE, giving you the option to set up the files and directories for the Lint checker, set the Lint parameter values before running the Lint tool, and run the Lint tool on the selected files or directories from within the SKILL IDE interface.

■ Editing of multiple files in one window

The basic editing capabilities of SKILL IDE include:

❑ **Auto-Indentation of Code**: The SKILL IDE automatically indents SKILL code so that a new line of code is indented to the same level as the previous one.

❑ **Auto-Completion of Function Names**: SKILL IDE provides you options for auto-completion of function names by suggesting valid function names based on your initial keystrokes.

❑ **Code Folding**: SKILL IDE's code folding feature allows you to collapse or expand parts of your code, letting you quickly navigate and focus on specific sections. You can fold any code section as long as it is contained within a set of parenthesis ( ) that are not on the same line number. It is also possible to fold nested code blocks

❑ **Code Inspection with Show Matching and Go To Matching Parenthesis Options**: SKILL IDE provides robust code inspection features. The show matching option is enabled by default. As a result, the cursor highlights the matching opening and closing pairs of parentheses and comment delimiters.

Using the *Edit – Go to Matching Parenthesis* menu option, you can direct the cursor to move to the matching opening or closing parenthesis or comment delimiter. As a result, you can quickly spot any missing parenthesis or comment delimiters. This is especially useful when you are reviewing code that contains nested parentheses.

❑ **Syntax Highlighting**: This feature helps you to visually inspect your SKILL code by highlighting different parts of your code in different colors. You can also set the font size and tab stop settings for better code inspection.

■ Documentation support

The document generation and display capabilities of SKILL IDE include:

❑ **Support for Inline Documentation Strings**: SKILL IDE supports embedding of inline documentation strings within SKILL/SKILL++ code. These documentation strings describe the attributes of the code elements with which they are associated. For example, the documentation string for a SKILL function may contain the description, parameters, and return values for the function.

❑ **Document Generation using Finder Manager**: SKILL IDE has a document generation utility called Finder Manager, which inspects the code containing inline documentation and generates finder-compatible documentation from it. You can then view this documentation in Finder.

❑ **Viewing Detailed Descriptions of SKILL APIs in Finder**: The complete documentation of a SKILL API is available at a single click. The new *More Info* button in Cadence SKILL API Finder displays the detailed information for an API including the syntax, arguments, description, return values, and example in a Cadence Help window.

Licensing Requirements

SKILL uses the **Cadence Design Framework II** license (License Number 111), which is checked out at the launch of the `skill` executable or the workbench. However, when you launch the SKILL IDE interface, Virtuoso also checks out the **Cadence SKILL Development Environment** license (License Number 900).

Related Topics

How SKILL IDE Works

Starting SKILL IDE

SKILL IDE User Interface

How SKILL IDE Works

When you open a file in SKILL IDE, it automatically opens in the SKILL IDE editor window. If you open multiple files, they are displayed on different tabs in the tab bar. To switch between files, click the required tab.

When you open a file, only the editing features are available; debugging operations are available only after you load the file. SKILL IDE does not start debugging until you run a function belonging to the file that has been loaded. You can also open a file in read-only mode. However, none of the editing or debugging features are available in this mode.

When you run a function, SKILL IDE checks for the presence of the function in the files that have been loaded and executes it. Before executing the function, you can make use of various debugging facilities, such as breakpoints, to identify the source of errors in the code.

You can use the `load()` function to load a SKILL file directly from the CIW. If you do this while SKILL IDE is open, the functions contained in that file are also available for debugging.

SKILL IDE saves a backup copy each time you edit a SKILL file. This backup copy is saved in the same directory as the original file and has `.skillide` prefixed to the name of the original file. For example, when you open a file named `demo.il`, a backup file with the name `.skillide.demo.il` gets created. This backup file can be used to recover any unsaved edits if the Virtuoso session terminates unexpectedly.

When you open a file in SKILL IDE, Virtuoso searches the directory where the file being opened resides for any corresponding backup file. If a backup file exists and it is different from the original file, a question dialog box appears seeking input whether to open the original file or restore its backup file. Consider a case where the system crashed while you were modifying the `demo.il` file. When you restart Virtuoso and reopen this file in SKILL IDE, Virtuoso checks the contents of the `.skillide.demo.il` backup file that had got created for it. If the two files have different content, you get the question dialog box to specify your choice. However, if both files have the same content, the `demo.il` file opens instantly.

Also, when you open a new file `Document_<N>`, SKILL IDE checks if a temporary backup file (`.skillide.Document_<N>`) for `Document_<N>` exists. If such a file exists, SKILL IDE names the new backup file as `.skillide.Document_<N+1>`. For example, if a temporary backup file `skillide.Document_6` exists, to prevent overwriting of `Document_6`, SKILL IDE opens the new file as `Document_7` and names its backup file as `skillide.Document_7`.

Related Topics

SKILL Integrated Development Environment

Starting SKILL IDE

You can start SKILL IDE either from a terminal window or from the Virtuoso Studio design environment.

Starting SKILL IDE from a Terminal Window

To start SKILL IDE from a terminal window, type the following command:

```
virtuoso -skillide
```

You can type the following command to specify the files that should open by default on starting SKILL IDE. If the files are not found, a warning appears.

```
virtuoso -skillide <file-name1> <file-name2> <file-name3>
```

Starting SKILL IDE from the Virtuoso Command Interpreter Window (CIW)

To start SKILL IDE from the Virtuoso Command Interpreter Window (CIW), choose *Tools – SKILL IDE*. The SKILL IDE window appears with *SKILLIDE* as the default workspace.
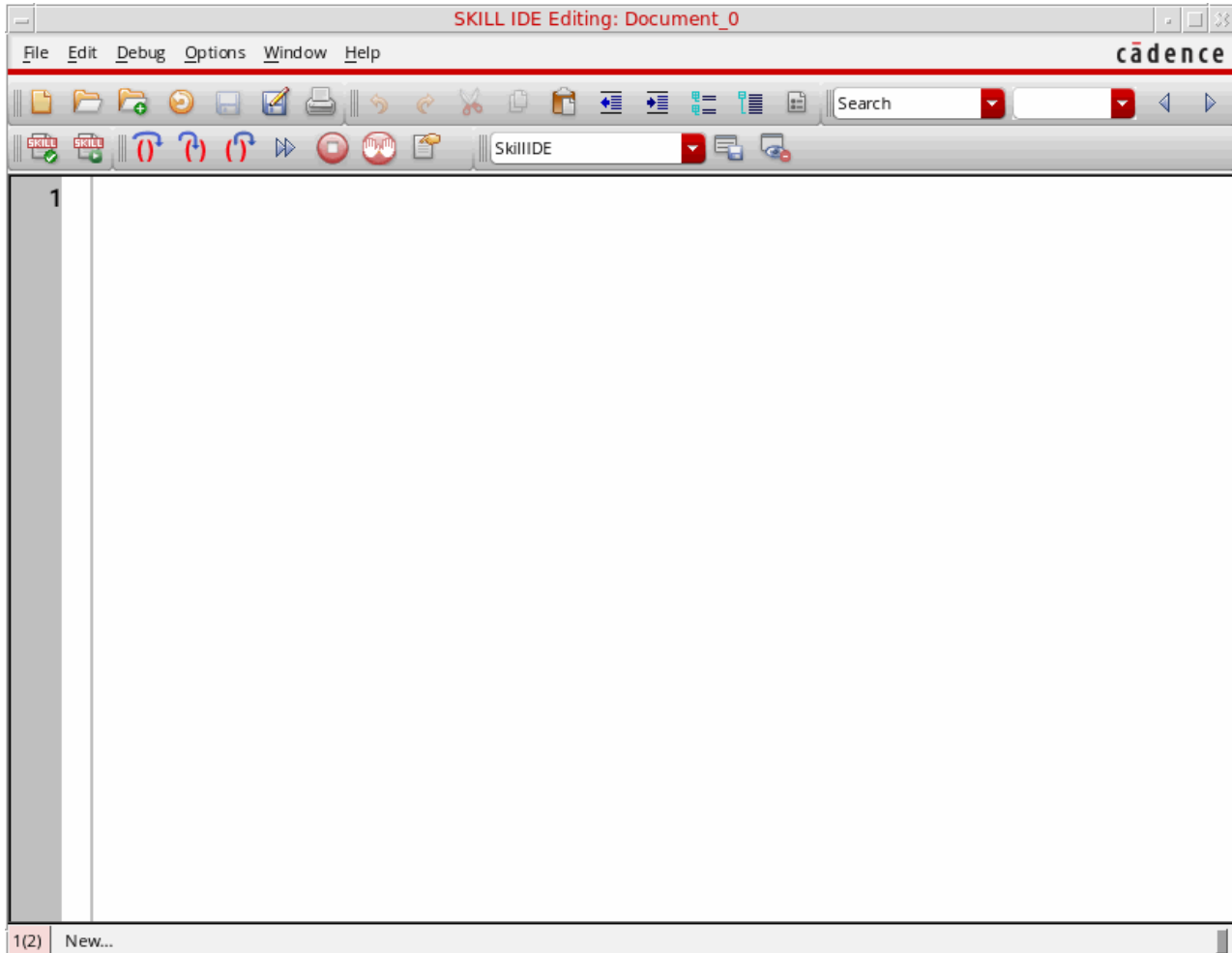
Related Topics

SKILL Integrated Development Environment

How SKILL IDE Works

SKILL IDE User Interface

SKILL IDE User Interface

The SKILL IDE interface is user-friendly and intuitive. It consists of menus, toolbars, dialog boxes, and windows that help you control your debugging operations. Therefore, you no longer need to learn complex debugging commands.

Menu Bar   Tab Bar   Toolbar   Assistants Pane

Status Bar   Source Code Pane
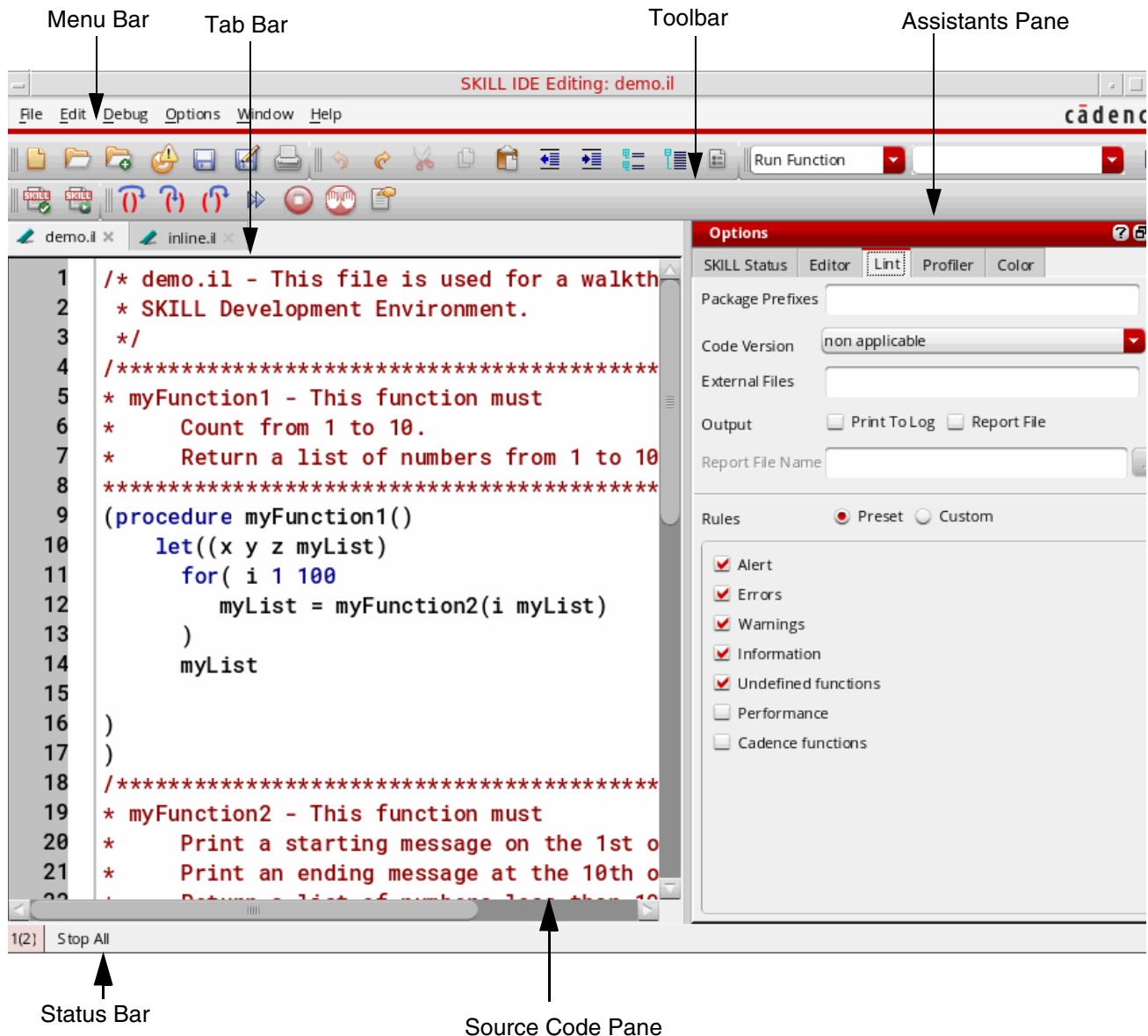
The following table describes the elements of the SKILL IDE interface:

Table: SKILL IDE User Interface Components

Menu Bar	Provides menus for basic file operations as well as editing, debugging, and tool control.
	The menu bar contains commands that perform an operation, display a cascade menu, or open a dialog box.
Toolbars	Contains buttons that provide quick access to the commonly used commands in the menu bar.
Tab bar	Displays the currently open files in tabs. It also contains buttons to help manage your tabs.
Source Code Pane	Displays the file being edited or debugged. If line numbering is enabled, line numbers are displayed to the left of the source code pane against each line of code. Syntax highlighting is enabled for the code in this pane for convenient code inspection.
Assistants Pane	Displays the currently enabled window assistants. You can display the *Breakpoints*, *Stack*, Trace, *Method Browser*, *Class Browser*, *Lint Manager*, *Finder*, *Code Browser*, *Profiler*, or *Step Result* assistants. You can also use the *Show All* and *Hide All* options to simultaneously display or hide all window assistants.
Status Bar	Displays the window number for each file tab at the bottom of the SKILL IDE window. For example, `1(2)`, where `1` is the session window number for the SKILL IDE, and `2` displayed (inside the parenthesis) is the window number for the open file.
	It also displays the line and column number of the current cursor position.

Related Topics

How SKILL IDE Works

Starting SKILL IDE

SKILL IDE Menu Commands

SKILL IDE Assistants

Managing SKILL IDE Toolbars

Managing SKILL IDE Tool Assistants

SKILL IDE Menu Commands

The SKILL IDE interface is user-friendly and intuitive. It consists of menus, toolbars, dialog boxes, and windows that help you control your debugging operations.

The following table lists the SKILL IDE menu commands along with their associated bindkeys. You can also view the SKILL IDE bindkeys in the *Bindkey Editor*.

Menu	Icon	Bindkey	Description
File			Contains commands that help you manage your SKILL files.
New		Ctrl+N	Creates a new file for editing.
Open		Ctrl+O	Opens an existing SKILL file for editing.
Open for Read			Opens SKILL file in read-only mode.
Open and Load			Opens SKILL file for editing and loads it.
Load		Ctrl+L	Loads the current file.
Close		Ctrl+W	Closes the SKILL file in the currently active tab.
Save		Ctrl+S	Saves the current version of the SKILL file.
Save As			Saves the SKILL file with a new name.
Discard Edits			Undoes all the edits you made in the currently open SKILL file since the last save operation.
Make Editable/ Make Read Only			Switches between edit and read-only mode.
Print		Ctrl+P	Prints the currently open SKILL file.
Close All			Closes all SKILL files and exits SKILL IDE. This command is replaced by the *Exit* command when the SKILL IDE is launched in a standalone mode.
Edit			Contains commands for editing SKILL files.
Undo		Ctrl+Z	Cancels the last edit operation.

Menu	Icon	Bindkey	Description
Redo		`Ctrl+Y`	Restores the last undo operation.
Cut		`Ctrl+X`	Moves the selected text to the clipboard.
Copy		`Ctrl+C`	Copies the selected text to the clipboard.
Paste		`Ctrl+V`	Pastes the copied text from the clipboard.
Select All		`Ctrl+A`	Selects all the text on the active tab.
Go to Matching Parenthesis			When you place the cursor before an opening parenthesis, the cursor moves to be after the matching close parenthesis. When you place the cursor after a closing parenthesis, the cursor moves to be before the matching open parenthesis. Otherwise the cursor is moved to the closest open parenthesis.
Go To Line Number		`Ctrl+G`	Moves the cursor to the specified line number.
Find/Replace		`Ctrl+F`	Searches and replaces text in the currently active tab.
Find Next		`Ctrl+Shift +F`	Searches and replaces text in the currently active tab.
Find Previous		`Ctrl+Shift +B`	Searches and replaces text in the currently active tab. See Finding and Replacing Text.
Debug			Contains commands for debugging SKILL programs.
Next		`Ctrl+E`	Executes the current SKILL statement. If the SKILL statement contains a function call, the debugger stops at the statement just after the call, without descending into the called function.
Step		`Ctrl+T`	Executes the current SKILL statement. If the SKILL statement contains a function call, the debugger descends into the called function.

Menu	Icon	Bindkey	Description
Step Out		Ctrl+J	Executes the remaining statements of the current function and returns control to the calling function.
Continue		Ctrl+U	Continues execution from a breakpoint until completion or until another breakpoint is encountered.
Stop Current Top-Level		Ctrl+Q	Terminates the function under execution.
Stop All Debugging			Stops execution of all functions on the execution stack.
Go To Current Line			Returns the cursor back to the file or tab that contains the current breakpoint line.
Dump Local Variables			Dumps the current values of all the local variables on the stack to the CIW.
Remove All Breakpoints			Removes all breakpoints from all SKILL files.
Save Settings			Exports the debug settings to a SKILL file.
Load Settings			Loads the debug settings from a previously saved file.
Options			Contains commands for customizing the status, editor, lint, profiler, and color settings.
SKILL Status			Opens the *Options* assistant and displays *SKILL Status* as the active tab.
			The *SKILL Status* tab lets you customize the status settings for the SKILL debugger, compiler, parser, print, stack, and trace.
Editor			Opens the *Options* assistant and displays *Editor* as the active tab.
			The *Editor* tab lets you customize the settings for the SKILL IDE editor.

Menu	Icon	Bindkey	Description
Lint			Opens the *Options* assistant and displays *Lint* as the active tab.
			The *Lint* tab lets you customize the settings for the lint checker.
Profiler			Opens the *Options* assistant and displays *Profiler* as the active tab.
			The *Profiler* tab lets you customize the settings for the SKILL profiler.
Color Settings			Opens the *Options* assistant and displays *Color* as the active tab.
			The *Color* tab lets you customize the color settings for the various elements of the SKILL code. For example, you can customize the colors for the background, foreground, comments, keywords, and other such elements of the code.
Window			Contains commands that enable you to hide or display tool components.
Assistants			Provides options for displaying the *Breakpoints*, *Stack*, *Step Result*, *Trace*, *Lint Manager, Code Browser*, *Class Browser*, *Method Browser*, *Finder*, or *Profiler* assistants. You can also use the *Show All* and *Hide All* options to simultaneously display or hide the assistants.
Toolbars			Provides options for displaying the *File*, *Edit*, *Debug*, *Search*, *Lint*, and *Workspace* toolbars.
Workspaces			Provides options for displaying the *Checking*, *Coding*, *Debugging*, *SKILL++*, *SKILL IDE*, and *User* workspaces. In addition, provides options to *save*, *delete*, *load*, and *set default* workspace. You can also *Revert to Saved* workspace, and *Show/Hide Assistants* displayed in specific workspace.

Menu	Icon	Bindkey	Description
Tabs			Displays the list of files that are currently open in the SKILL IDE tabs. It also provides the *Close Current Tab* and *Close Other Tabs* options.
Help			Gives you access to the Cadence documentation.

Related Topics

How SKILL IDE Works

Starting SKILL IDE

SKILL IDE User Interface

SKILL IDE Assistants

SKILL IDE Assistants

The SKILL IDE offers several development tools to help you develop, test, and refine SKILL programs. These are briefly described below.

Assistant	Description
Breakpoints	Displays breakpoints currently set in your code and lets you change their state or condition.See Program Execution Control.
Trace	Continually displays the values of variables as you step through the program.See Examining and Modifying Variable Values.
Stack	Displays functions and their arguments are listed in the order in which they were called. See Examining the Call Stack.
Class Browser	Displays the class inheritance hierarchy of the classes used in your SKILL code. See Class Inheritance Relationships.
Method Browser	Displays the method trees of generic functions. See Using the Method Browser Assistant.
Lint Manager	Helps you examine SKILL code for possible errors that go undetected during normal testing. See Efficiency Improvement of SKILL Code.

Assistant	Description
Finder	Displays the abstract and syntax statements for all SKILL/SKILL++ elements like classes, functions, and methods. See <u>Using the Finder Assistant</u>.
Code Browser	Displays the calling trees and definitions of user-defined functions. See <u>Using the Code Browser Assistant</u>.
Profiler	Helps you check the time and the memory consumption of your SKILL programs. See <u>SKILL Profiler Assistant</u>.
Step Result	Helps you examine your code, including evaluated expressions, as you step through the program statements. See <u>Using the Step Result Assistant</u>.

Related Topics

<u>How SKILL IDE Works</u>

<u>Starting SKILL IDE</u>

<u>SKILL IDE User Interface</u>

<u>SKILL IDE Menu Commands</u>

<u>Managing SKILL IDE Tool Assistants</u>

2

SKILL IDE Work Environment Configuration

You can rearrange the assistants and toolbars to suit your individual work preferences. A customized configuration of toolbars and assistants is called a workspace.

You can define the following properties of your SKILL IDE work environment:

■ The assistant panes that should be docked, floating, or hidden.

■ The toolbars that should be docked or hidden.

■ Position of each assistant pane and toolbar in your SKILL IDE window.

When you exit the SKILL IDE window, the changes you make to the SkillIDE workspace layout get saved automatically and are loaded the next time you start SKILL IDE. This behavior is true only if the default workspace is SkillIDE. If you selected a different default workspace, then that one is loaded on starting SKILL IDE.

Related Topics

Managing SKILL IDE Toolbars

Managing SKILL IDE Tool Assistants

Setting SKILL IDE Options

Basic Editing Features

Advanced Editing Features

Debugging in SKILL IDE

Managing SKILL IDE Toolbars

To manage your debugging space more efficiently, you can choose to control and arrange display of the SKILL IDE toolbars. SKILL IDE toolbars are also available in the Virtuoso Toolbar Manager.

Showing and Hiding Toolbars

To show or hide the SKILL IDE toolbars:

Choose *Window – Toolbars – <Toolbar name>*. The selected toolbar displays below the SKILL IDE menu bar. Alternatively, you can right-click anywhere in the menu bar or a toolbar and choose the required toolbar name from the context-menu to display it.

Choosing this option again hides the selected toolbar.

SKILL IDE toolbars are also available in the Virtuoso Toolbar Manager.

Moving Toolbars

SKILL IDE toolbars are dockable. Therefore, you can move the toolbars to different locations in the workspace. To move a toolbar:

1. Point to the move handle ▯ of the toolbar.

2. When the mouse pointer changes to a crosshair (⊕), drag the toolbar to a new location

   ❑ If you drag the toolbar close to the edges of the SKILL IDE window, you see an outline of the area where the toolbar will be docked to the window frame. The toolbar snaps to this area when you release the mouse button.

   ❑ If you drag the toolbar to an area away from the IDE window frame—inside or outside the window—the toolbar becomes a floating toolbar.

Related Topics

Using Toolbar Manager

Managing SKILL IDE Tool Assistants

SKILL IDE User Interface

Managing SKILL IDE Tool Assistants

SKILL IDE provides you with options to help you manage your assistants. You can also move and reposition the tool assistants to change your workspace layout.

Displaying Tool Assistants

To display or hide a tool assistant, choose *Window – Assistants – <Assistant name>*. To display or hide all tool assistants at once, choose *Window – Assistants – Show All* or *Window – Assistants – Hide All*.

Alternatively, to display or hide an assistant, right click the menu bar and choose *Assistants – <Assistant name>*. The assistants you choose for display have a selected check box in front as illustrated in the image below:
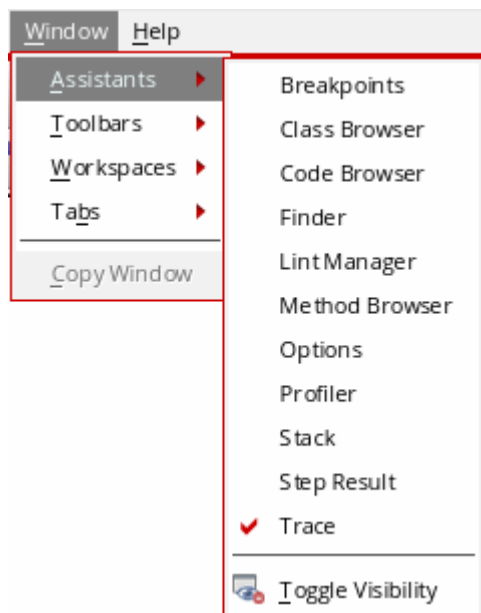
Docking or Floating Tool Assistants

To dock a floating assistant pane:

- Drag the assistant pane by its title bar and move it close to the edges of the SKILL IDE window frame. When an outlined area appears, release the mouse button; the assistant pane snaps and fits in the outlined area.

- Double-click the title bar of the floating assistant pane.

■ Click the Float/Dock 🗗 button on the title bar of the docked pane.

To make a docked assistant pane float:

■ Drag a docked pane by its title bar away from the edges of the SKILL IDE window frame and release the mouse button. The assistant pane becomes a floating assistant pane.

■ Double-click the title bar of the docked assistant pane.

■ Click the Float/Dock 🗗 button on the title bar of the floating pane.

Note: Hold down the Ctrl key while you drag an assistant pane to prevent it from docking.

Displaying Tool Assistants as Tabs

You can display assistant panes as tabs by dragging and dropping one assistant on top of another.

To display an assistant in a tabbed format:

➡ Drag the title bar of one assistant and drop it on the body area of another assistant.

You can click the Float/Dock 🗗 button to detach one assistant from the other tabbed assistant.

Related Topics

Managing SKILL IDE Toolbars

SKILL IDE User Interface

Setting SKILL IDE Options

SKILL IDE provides a number of options for customizing the behavior of SKILL IDE elements. The following options are accessible through the *Options* menu:

■ *SKILL Status*

Lets you customize the status settings for the SKILL debugger, compiler, parser, printer, stack, and trace. To set the *SKILL Status* options:

Choose *Options – SKILL Status*.

The Options assistant displays with *SKILL Status* as the default tab.
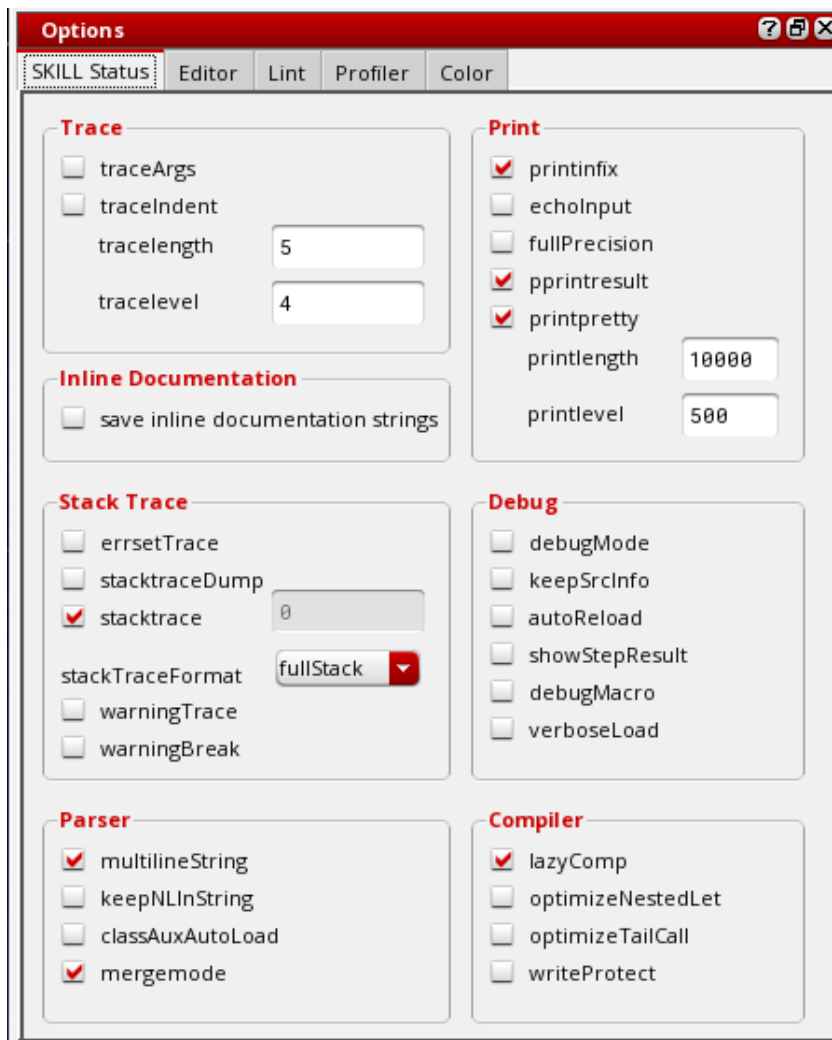
■ *Editor*

The *Editor* command lets you customize the settings for the SKILL IDE editor. To customize the behavior of the SKILL IDE editor:

Choose *Options – Editor*

The Options assistant displays with *Editor* as the default tab.

■ *Lint*

The *Lint* command lets you set the Lint parameter values before running the Lint tool. To set the Lint parameters:

Select *Options – Lint* or click ▣ (*Lint Options*) in the in the Lint Manager assistant window or ▣ (*Options*) in the *Edit* toolbar.

The Options assistant displays with *Lint* as the active tab.
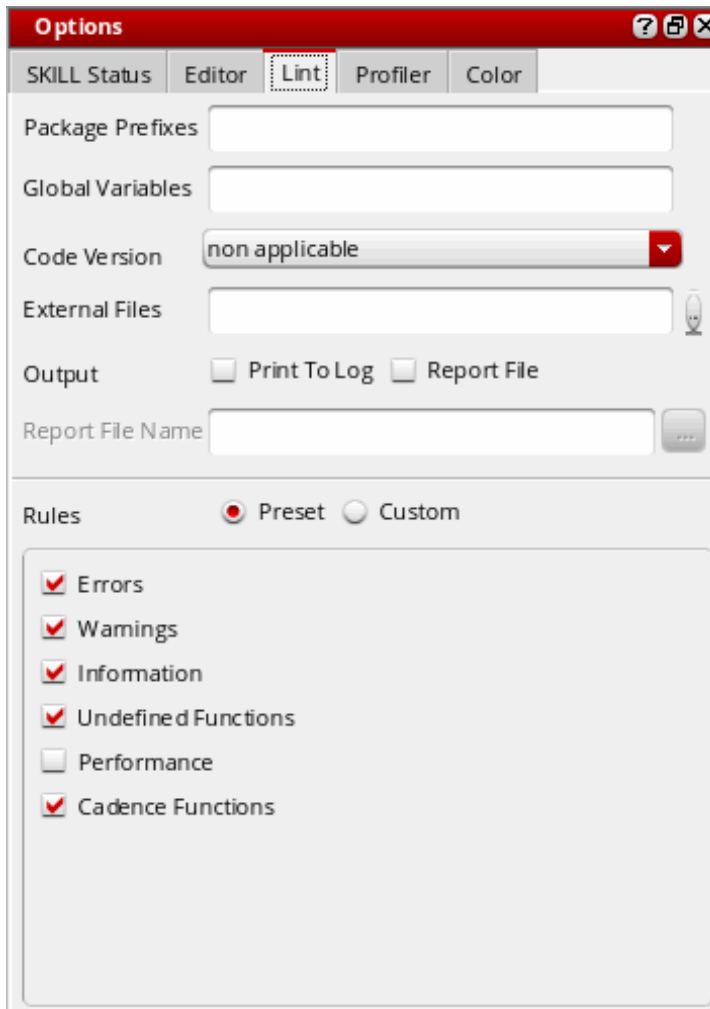
■ *Profiler*

The *Profiler* command lets you define the type of data you want the Profiler to collect and display. To set the Profiler Options for collecting data, before running the Profiler:

Choose *Options – Profiler*. You can also click ▦ (*Profiler Options*) in the toolbar displayed in the Profiler assistant window.

The Options form displays with *Profiler* as the default tab.
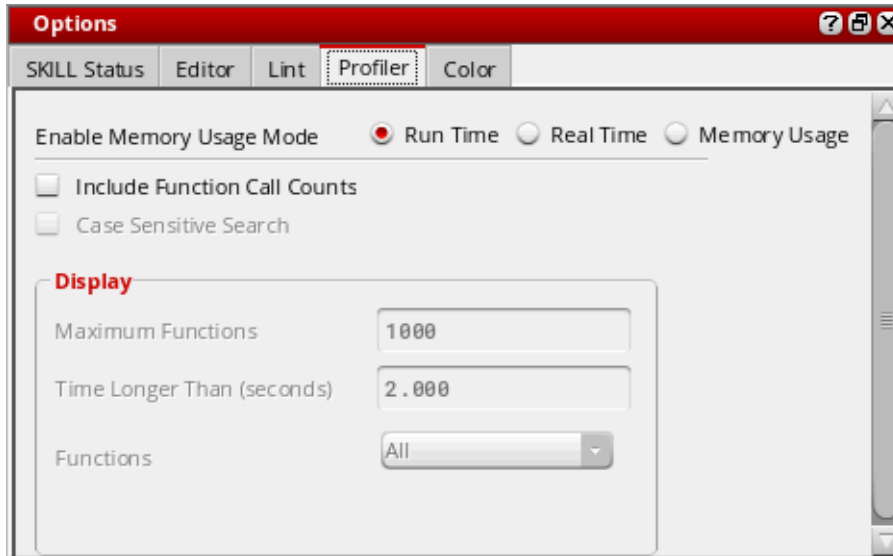
■ *Color Settings*

The *Color Settings* command lets you customize the color settings for the SKILL IDE editor window. To set the color preferences for various elements of the SKILL IDE editor window:

Choose *Options — Color Settings*.

The Options assistant displays with *Color* as the default tab.
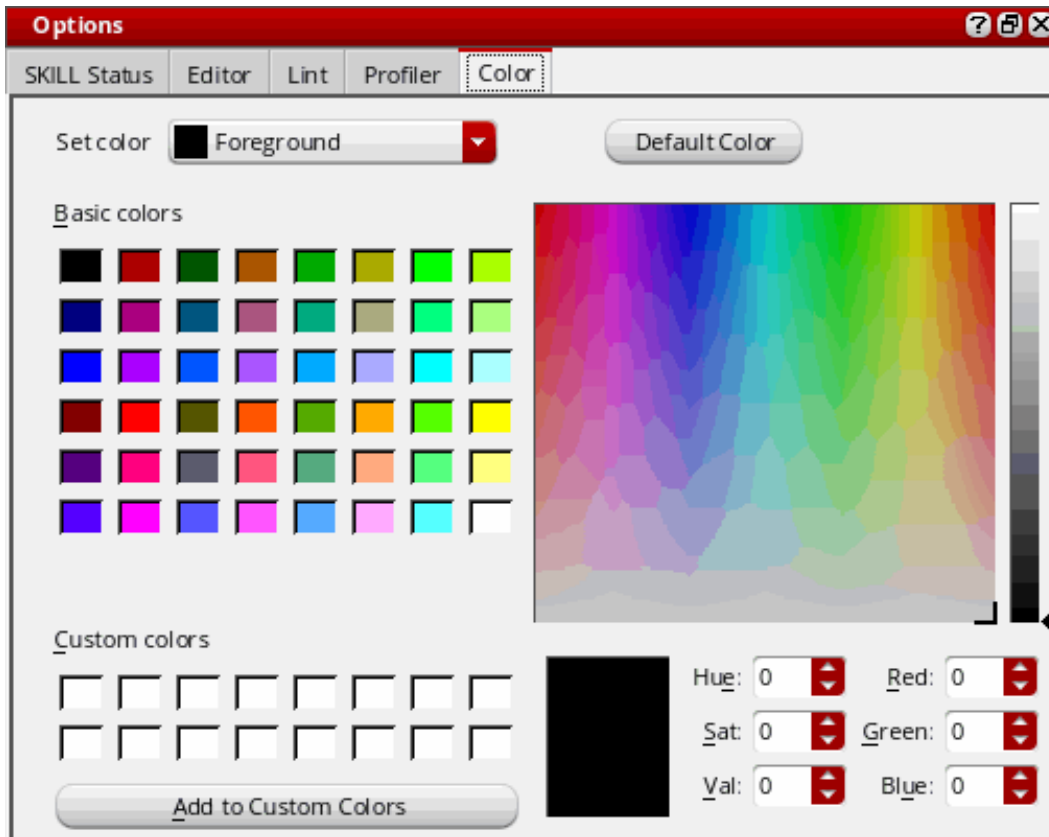
Related Topics

Setting Lint Options

Setting Profiler Options

Customizing the Color Settings

SKILL IDE Options Form

Customizing the Color Settings

To set the color preferences for various elements of the SKILL IDE editor window:

1. Choose *Options — Color Settings*.

The Options assistant displays with *Color* as the default tab.
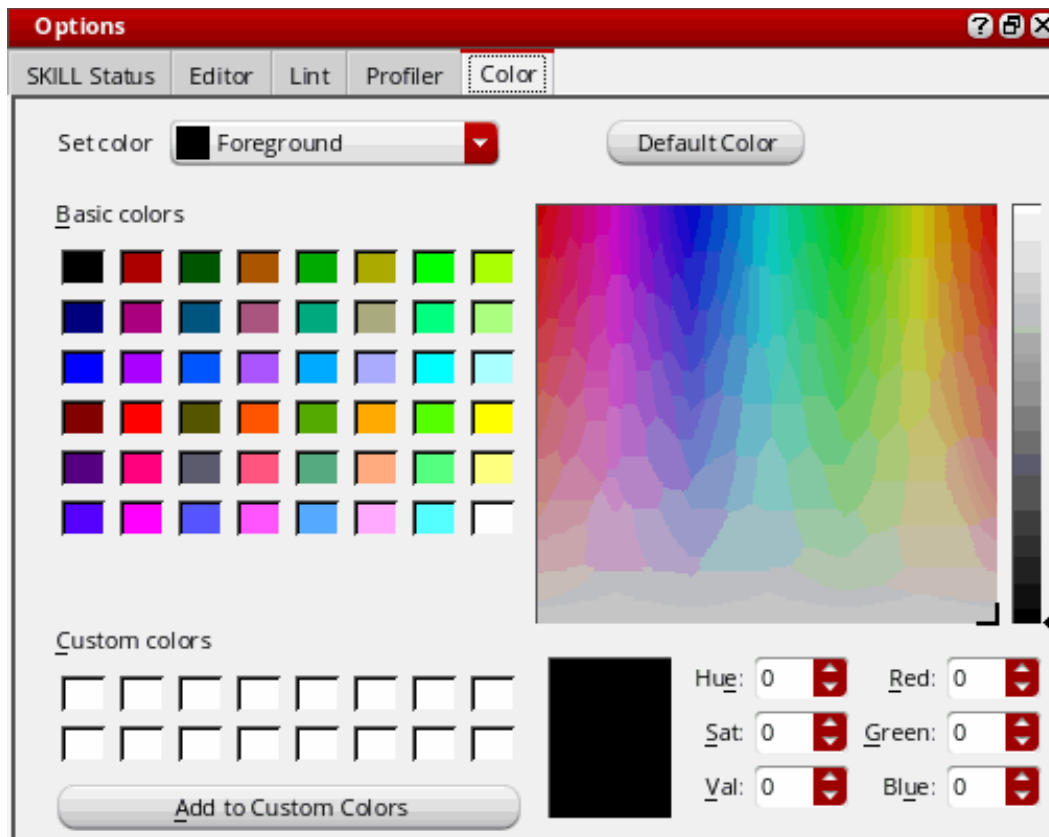
2. Use the *Set color* drop-down list box to change the color preferences for the code elements. The available options are: *Foreground, Background, Selected Text, Matching Text, Matching Parenthesis, Mismatching Parenthesis, Error, Step, and Cross-Highlight.*

   When you select a message in the *Lint Output* window, its corresponding code is highlighted in the cross highlight color in the editor window. This option also changes the cross highlight color between objects and their Pcell source code.

3. Click a color in the color swatch to change the color of the selected element. Click *Default Color* to switch it back to the default color.

4. Create custom colors by dragging the mouse pointer inside the color palette and adjusting the contrast. You can then click *Add to Custom Colors* to save the new color as a custom color.

Related Topics

Setting SKILL IDE Options

Basic Editing Features

SKILL IDE lets you edit and debug SKILL code in a graphical user interface (GUI). When you open a SKILL file or create a new one, the SKILL IDE editor gets invoked.

Creating New Files

1. To create a new file, do one of the following:

   ❑ Choose *File – New*

   ❑ Click 🗋 on the File toolbar

   A file with a default name (*Document_n*) opens in the current directory.

2. To save this file, click 💾 on the File toolbar or choose *File – Save As*. The *Save As* dialog box displays.

   The first time you display the *Save As* dialog box by either selecting *File – Open* or *File – Save As*, the current working directory displays as the default working directory. On subsequent calls, the last used directory is used as the default working directory.

3. Specify a name for the file in the *File name* field and select an appropriate file format from the *Files of type* drop-down list.

   Note: The default file formats available in the *Files of type* drop-down list are `.il` and `.ils`.

4. Click *Save*.

   Note: When saving a file, if the name you enter in the *File name* field matches an already open file, a warning message appears. You are then prompted to either rewrite the open file or cancel the save operation.

Opening Existing Files for Reading/Editing

1. To open an existing file for editing, do one of the following:

   ❑ Choose *File – Open*

   ❑ Click a file name from the list of recently opened files in the *File* menu.

   ❑ Click 📁 on the File toolbar

To open an existing file for viewing only, choose *File – Open for Read*.

The *Choose a File* dialog box displays, listing all the available files.

Note: By default, *All Files (*)* is selected in the *Files of type* list box. To refine the criterion for displaying only a specific type of files, select *SKILL files (*.il), Scheme files (*.ils *.scm)* or *SKILL/Scheme files (*.il *.ils *.scm)* from the list box.

2. Browse to select the file you want to read or edit and click *Open*. The selected file opens in the editor.

   If you open the file in read-only mode, the title bar of the SKILL IDE window displays `SKILL IDE Reading: <name_of_the_opened_file>`. If you open the file in edit mode, the title bar displays `SKILL IDE Editing: <name_of_the_opened_file>`.

   Note: If you attempt to open an already open file that has unsaved changes, a warning message appears. You are then prompted to reload the file from the disk.

   If the file currently open in the SKILL IDE editor has been edited using some other editor, a warning message appears. You are then prompted to either reopen the updated file without loading, reload the original file from the disk, or ignore the changes made using the other editor.
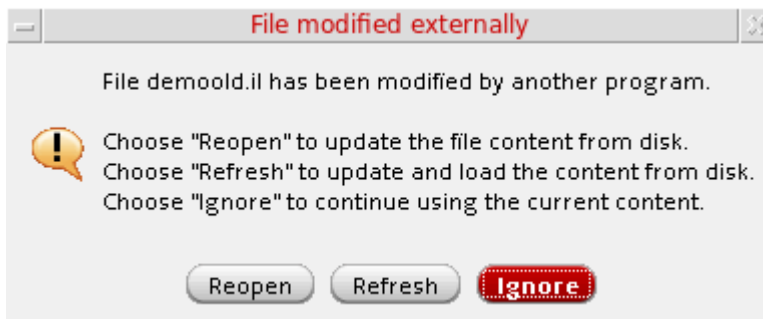
   If you choose to reopen the updated file without loading or reload the original file from the disk, you are prompted to save the current contents of the file to a different file before the reopen or refresh operation.
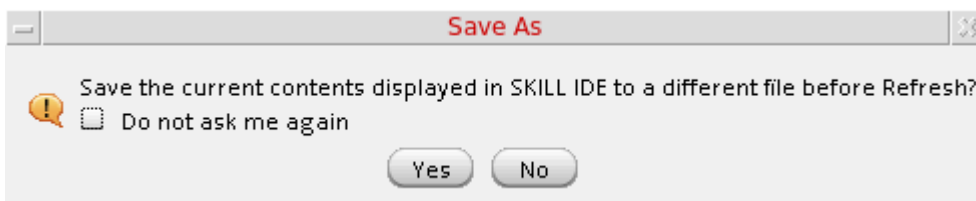
   If you do not wish to be prompted again, select the *Do not ask me again* option in this dialog box.

Switching Between Read-Only and Edit Modes

To switch from edit mode to read-only mode, choose *File – Make Read Only*.

When you edit a file in edit mode, the icon becomes enabled, indicating the need to save the file.

To switch back to edit mode, choose *File – Make Editable*.

When you edit a file in read-only mode, the icon becomes enabled, indicating the need to save the file.

Discarding Edits

To undo your edits:

1. Choose *File – Discard Edits*. A warning message that prompts you to confirm the action appears.

2. Click *Yes* to confirm. Once discarded, the edits can not be restored.

Finding and Replacing Text

To find a text string in your file:

1. Choose *Edit – Find* (Ctrl+F). The *Find and Replace* dialog box displays.
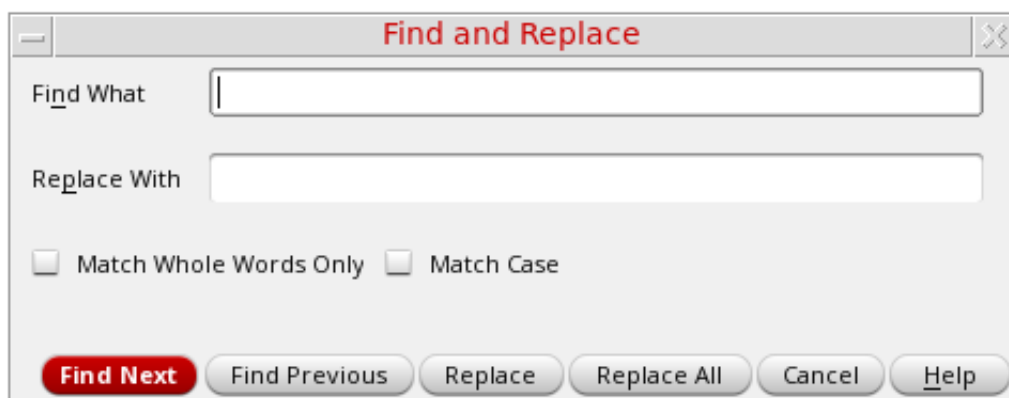
2. In the *Find What* field, type the search string and then, set the following options:

   ❑ *Match Whole Words Only*: Select this check box to match the exact string. For example, if you type "list" as the search string, both "list" and "mylist" will be highlighted, unless you select this check box.

❑ *Match Case:* Select this check box to perform a case-sensitive search.

3. Click *Find Next* (Ctrl+Shift+F) or *Find Previous* (Ctrl+Shift+B) to find the next or previous occurrence of the search string in the file.

Note: If the search string is not found in the file, an error message displays.

You can also access the search functionality from the *Search* toolbar, as described below:

In the *Search* toolbar, choose *Search* from the first drop-down list box. In the second drop-down list box, type the search string or click the drop-down arrow to choose a previously used search string and then, press Enter. After the search string is found, you can click the Find Previous ◁ and Find Next ▷ icons to find the previous or next occurrence of the search string.

Note: Instead of pressing Enter, you can click the Find Previous ◁ *or* Find Next ▷ icons to initiate the search operation after specifying the search criterion. When the *Highlight Matching text* option is enabled, all occurrences of the search string are highlighted in cyan.

To replace a text string in your file:

1. Choose *Edit – Find* (Ctrl+F). The *Find and Replace* dialog box displays.

2. In the *Find What* field, type the search string.

3. In the *Replace With* field, type the text string you want to replace the search string with.

4. Set the following options:

❑ *Match Whole Words Only*: Select this check box to match the exact string.

❑ *Match Case:* Select this check box to perform a case-sensitive search.

5. Click *Find Next* (`Ctrl+Shift+F`) to find the search string. If the search string is found, it is highlighted in the source code pane.

6. Click *Replace* to replace the search string with the new string or *Replace All* to replace all occurrences of the search string with the new string.

Printing Files

SKILL IDE provides the basic print features to help you print your SKILL files. To print a file:

1. Click 🖨 on the File toolbar or choose *File – Print*. The *Print Dialog* box displays.

2. Click *Options* to specify the print options for the current print job. You can specify the printer name, print range, output settings, and the color mode for the print job.

3. Click *Print* to send the file to the selected printer for printing.

Closing Files

If you have opened multiple files in the editor, to close each file separately do one of the following:

❑ Choose *File – Close*.

❑ Click 📂 on the File toolbar.

❑ Right-click an existing <file-name> tab in the tab bar and choose *Close Tab* from the context-menu.

To close all files simultaneously, choose *File – Close All.*

When you click *Close All*, if any of the open files has unsaved changes, the *Save Changes* dialog box displays. This dialog box prompts you to save the open files before exiting the editor. Select the check box adjacent to the files you want to save and click *OK*.
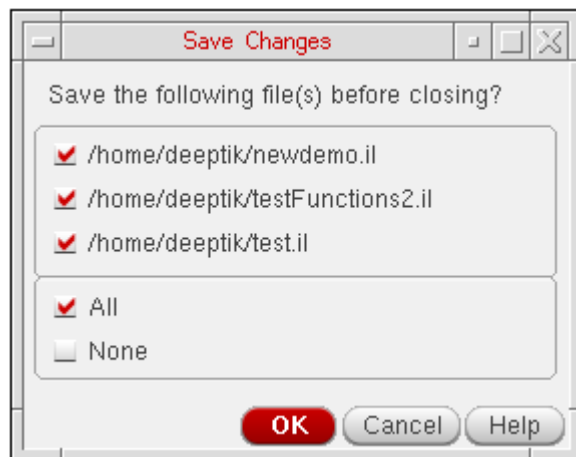
In addition, you get the *Save Changes* dialog box when trying to close the Virtuoso CIW directly without exiting the SKILL IDE editor. If you click *Cancel*, the exit from Virtuoso CIW too gets aborted.

Exiting the Editor

To exit SKILL IDE:

❏ Choose *File – Close All*.

❏ Click ⊠ on the SKILL IDE window.

If your open files have unsaved changes, you are prompted to save each file individually before exiting the editor.

Related Topics

Advanced Editing Features

Advanced Editing Features

Viewing Function Definitions

To view the definition of a function in the source code pane:

➡ Right-click the function name and choose *Go To Source* (⬚) from the context-menu. The first and the last line of the function definition is highlighted in gray.

Browsing Function Tree

To browse the calling tree of user-defined functions directly from the source code pane:

➡ Right-click the function name and choose *Go To Code Browser* from the context-menu.

The Code Browser assistant gets displayed with the name of the selected function in the *Function* drop-down list and its expanded tree in the results pane.

Matching Parentheses

For better code inspection, you can use the *Go To Matching Parenthesis* command.

➡ Place the cursor on an opening or closing parenthesis and choose *Edit – Go To Matching Parenthesis*.

The cursor moves to the matching parenthesis of the opening or closing parenthesis.

For better usability, the *Go To Matching Parenthesis* command can also be accessed by using the `Ctrl + M` bindkey. The two other bindkeys that can be used for this feature are:

■ `Ctrl + [` Moves the cursor to the next inner open parenthesis.

■ `Ctrl + ]` Moves the cursor to the next outer open parenthesis.

Going to a Line

Use the *Go To Line Number* command to go to a specific line in a file.

1. Choose *Edit – Go To Line Number* and enter a line number. The cursor moves to the specific line.

2. Having located a line, click *OK* close the dialog box or *Apply* to keep it open and continue to search for other lines.

The *Go To line Number* command can also be accessed by using the `Ctrl + G` bindkey. The two other bindkeys that can be used for this feature are:

`Ctrl + Home` Moves the cursor to the beginning of the file.

`Ctrl + End` Moves the cursor to the end of the file.

Selecting Text Between Matching Parentheses

To select the text between the matching parenthesis:

➡ Place the cursor on an opening or closing parenthesis and press `Ctrl+Shift+M`. The text between the two matching parenthesis gets selected.

Autocompleting Function Names

The autocomplete feature enables automatic completion of keywords or function names based on your keystrokes. To enable this option:

1. Choose *Options – Editor*.

2. Select the *Enable name completion* check box.

   When name completion is enabled, a list of valid function names displays in the tooltip on entering three or more keystrokes, or on pressing `Ctrl+Space`.

```
procedure(simple (a b)
let ((x y "abc")
hiCre|
```

| hiCreate2DMenu |
| hiCreateAction |
| hiCreateAppForm |
| hiCreateBBoxField |
| hiCreateBooleanButton |
| hiCreateButton |
| hiCreateButtonBoxField |
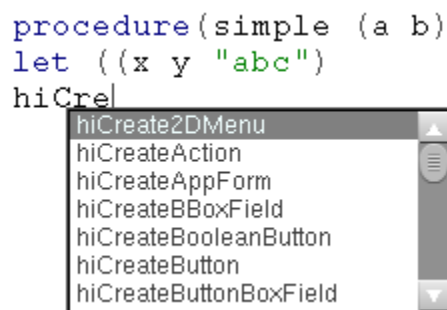
If the *Finder* assistant is open, it is updated with the finder documentation of the selected function.
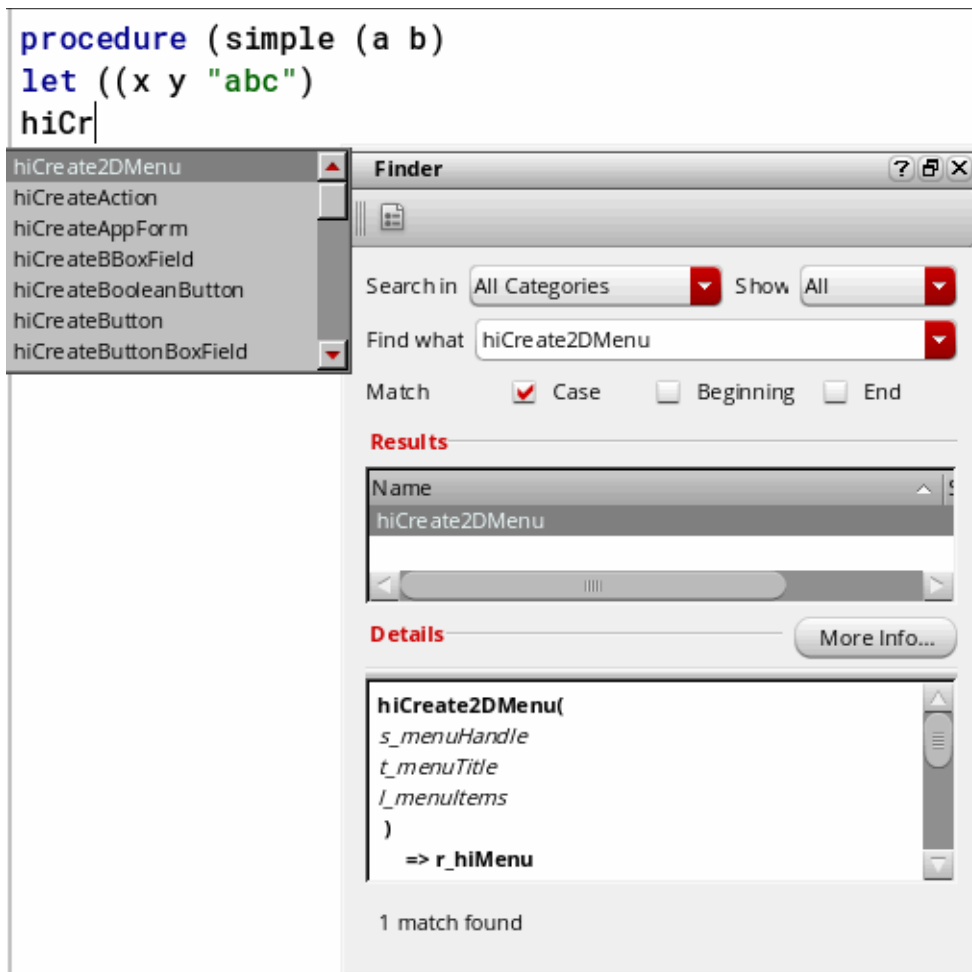
3. Click an appropriate function name in the pop-up list to add it to your code.

    The list of valid function names is picked up from the finder (`.fnd`) files found in:

    ```
    your_install_dir/doc/finder/language/functionArea/*.fnd
    ```

You can add your own internal functions to the autocomplete list by copying your function information to a finder database file under the `finder` directory. The name-completion option can also show matches with variable names and other function names that do not have finder files.

Indenting Code

➡ Use the *Shift left* (⬛) or *Shift right* (⬛) icons from the *Edit* toolbar to indent your code to the left or right by a predefined number of spaces.

Clicking the *Shift left* icon removes one level of indent from the current line (that is, the line on which the mouse pointer is placed) or selected block of code (including a partially selected line). *Shift left* stops indenting to left when one of the source lines has reached its limit, which is less than the tab stop value. If tab stop is set to 4, and there are only one-three spaces to the left of the line, then that line cannot indent to the left any more.

Clicking the *Shift right* icon adds one level of indent to the current line or selected block of code.

Folding and Expanding Code

You can use SKILL IDE's code folding feature to collapse or expand parts of your code, helping you navigate and focus on specific sections. You can fold any code section as long as it is contained within a set of parenthesis ( ) that are not on the same line number. It is also possible to fold nested code blocks.

➡ Click the *Outline View*(⬛) icon in the *Edit* toolbar to fold the code in the SKILL IDE editor window. Alternatively, right-click in the left margin of the source code pane and choose *Outline Mode* from the context menu.

When the code is folded, a ⊞ icon appears to the left of the code blocks.

On folding or collapsing the code, the lines containing the opening and closing parenthesis form an outline of the folded code, while the lines within the parenthesis are hidden. For

example in the image below, line number 1 and 6 contain the opening and closing parenthesis, and thus form the outline of the folded code.

➡ Click the *Expand All* ( ) icon in the *Edit* toolbar to unfold or expand the code in the SKILL IDE editor window. Alternatively, right-click in the left margin of the source code pane and choose *Expand All* from the context menu. When the code is expanded, a icon and a vertical bar appear to the left of the code blocks.

The icon and the vertical bar also appear every time you place the pointer before an opening parenthesis or after a closing parenthesis of a code block.

To unfold or expand only a specific code block, right-click in the left margin of the code block that needs to be expanded and choose *Unfold Line* from the context menu. Unfolding a specific code block does not unfold the nested code blocks within the top-level code block.

Important Points to Note

■ Folding or unfolding the code does not change the flow of code in any way.

■   Undo and redo operations have no impact on code folding.

■   Code folding changes are not persistent across sessions. After you close a file containing folded code, the file will be restored to its previous state.

Related Topics

Basic Editing Features

Customer Data

SKILL API Finder

Debugging in SKILL IDE

This section provides information on preparing a file for debugging in SKILL IDE. For information about the features that help you control the flow of your program during a debugging session, see Program Execution Control.

The SKILL IDE does not support debugging macros and other manually created code.

Loading a File for Debugging

To debug a SKILL file, you need to first load it. To do so:

1. Click ▣ on the *File* toolbar or choose *File – Open*. The Open File dialog box displays, listing all the available files.

2. Browse to locate the file you want to load and click *Open*. The selected file opens in the source code pane.

3. Click ✋ on the *File* toolbar or right-click in the source code pane and choose *Load* from the context-menu to load the file for debugging.

    After the file is loaded, the load icon changes from ✋ to ↺.

If an error occurs while loading the SKILL file, the line on which the error occurred gets highlighted in the source code pane. An error icon ⊖ also appears in the left margin of the line containing the error. In addition, related messages listing the error details can be viewed in the CIW as well as the tooltip of the error icon ⊖ . If you fix the cause of the error and save or load the file, the error icon disappears from the left margin.

To simultaneously open and load a file for debugging, click on the *File* toolbar or choose *File – Open and Load*.

If you attempt to debug a file that you edited after loading, a warning message displays. It states that the version of the file previously loaded does not match the file currently displayed in the editor and prompts you to take an appropriate action.

Executing a Function

To start debugging, you need to execute a function defined in the file that has been loaded. There are two ways to do this:

Executing the function from the Search Toolbar

1. In the *Search* toolbar, choose *Run Function* from the first drop-down list box.

2. Specify the name of the function you want to run (with the required argument values) in the second drop-down list box and press Enter.

Executing the function from the CIW

In the CIW, type the function name (with the required argument values) and press Enter. This method is used if you want to run debugging sessions concurrently.

Related Topics

Saving and Reusing Debug Information

Saving and Reusing Debug Information

You can export the debug settings of your current session to a SKILL file for future use. The saved debug settings includes current line and function breakpoints, their conditions, traced functions, variables, and properties.

If you accidentally delete all breakpoints from your code, you can restore them by loading the debug information from a previously saved file.

Saving the Debug Settings

To save the debug settings of your current session:

1. Choose *Debug – Save Settings*. The *Choose file name to save debug data* dialog box displays.

2. Specify the name of the output file in the *File Name* field and click *Save*.

Loading the Debug Settings

To load the debug settings:

1. Choose *Debug – Load Settings*. The *Load debug information* dialog box displays.
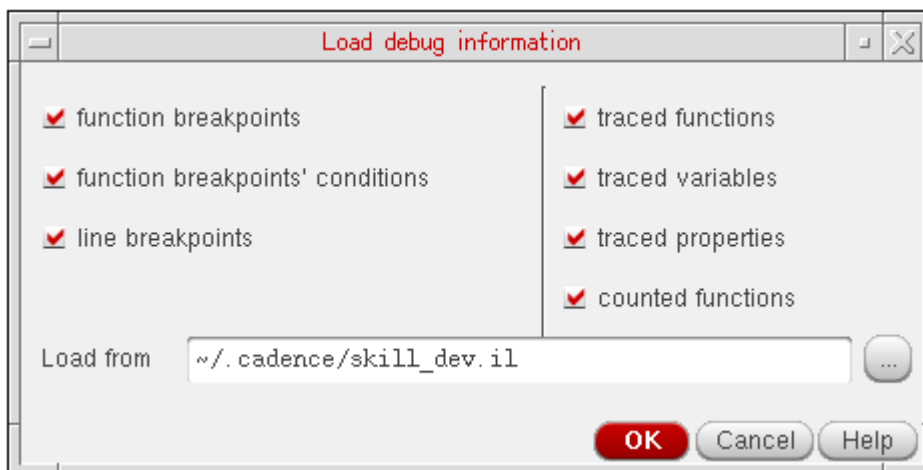
2. Specify the file from which you want to load your saved debug settings in the *Load from* field.

3. Click *OK*.

Related Topics

Debugging in SKILL IDE

3

Program Execution Control

Using the features that help you control the flow of your program, you can determine where the program execution pauses. You can then inspect your code, start, stop, and step through lines of code, and examine or modify the values of the various variables as required.

What are Breakpoints?

Breakpoints direct the debugger to pause the execution of a program at a certain point in the code or on the occurrence of a certain condition. During this suspended state, the program is said to be in break mode. Entering the break mode does not terminate the execution of your program. You can resume the execution at any time by single stepping through the lines of code or running the program from breakpoint to breakpoint using the *Continue* command.

In the break mode all variables and functions remain in the memory. This enables you to examine their values to determine the possible errors in the program.

You can set the following types of breakpoints in SKILL IDE:

- Line breakpoints

  A line breakpoint is triggered when the line it is set on is reached.

- Function breakpoints

  A function breakpoint is triggered when the function it is applied to is entered or exited, depending on how you configure the breakpoint. Program execution in this case pauses when the function is called or when it returns control.

Unconditional and Conditional Breakpoints

Unconditional breakpoints cause the debugger to pause the program execution at a given line of code.

To add flexibility, you can set conditions on breakpoints, such that the program execution pauses only when the breakpoint condition is satisfied. Such breakpoints are called conditional breakpoints.

By using conditional breakpoints, you can stop the code execution on specific lines of code, at the entry points of functions, on call returns, or on both. In conditional breakpoints, you define a condition—an expression that returns a logical value, for example, `i==10`. This condition is evaluated every time that breakpoint is reached. If the condition is satisfied, program execution pauses.

The following operators can be used to construct conditional expressions:

<, <=, ==, >=, >, and !=.

Related Topics

Configuration of Conditional Breakpoints

Setting Unconditional Breakpoints

Setting Conditional Breakpoints

Clearing Breakpoints

Managing Breakpoints

Configuration of Conditional Breakpoints

You can configure breakpoint conditions to suit your debugging needs. For example, you can temporarily disable a breakpoint without deleting it or change the entry or exit criteria for a function breakpoint.

■   Enabling and Disabling Conditional Breakpoints

■   Specifying Breakpoint Entry and Exit Criteria

Enabling and Disabling Conditional Breakpoints

The SKILL IDE debugger provides the following conditional breakpoint options to help you enable or disable conditional breakpoints:

■   **Conditional:** The debugger stops the program execution at the breakpoint only if the breakpoint condition is true.

■ **Disabled**: The debugger never pauses the program execution at the breakpoint, whether or not the breakpoint condition is true.

This variation of conditional breakpoint disables a breakpoint while preserving the location and condition of the breakpoint. You can use this option instead of deleting the conditional breakpoint. The advantage of using this option is that you do not have to find the location of the breakpoint in the source to set it again.

■ **Enabled**: The debugger always pauses the program execution at the breakpoint, whether or not the breakpoint condition is true.

This variation of conditional breakpoint is useful in cases where you want to test the state of the program under all conditions. For example, you define a variable $i=i+2$ in your program and set the breakpoint condition such that the breakpoint is triggered when $i==8$. Now, if the value of i starts at 1, the breakpoint is never triggered since i will always have an odd value. In such a scenario, to see how the value of i progresses through each iteration, even when $i==8$ is not true, set the conditional breakpoint to *Always*.

Specifying Breakpoint Entry and Exit Criteria

Function breakpoints pause the program execution when control enters or exits the function to which the breakpoint is applied. You can specify an entry or an exit criteria for a function such that the breakpoint is triggered only when the given criteria is met. For example, if you have a function `myTest(object)` you can set a breakpoint that activates only when the argument `object` has a specific value (say `object==4`) on entry to the function. You can also specify an exit criteria, which is tested before the function returns the control back to the program.

You have the following choices for setting up an entry and exit criteria:

■ **Break on Both Entry and Exit when**: The same condition is defined for both function entry and exit. When you use this option, the execution is suspended twice, both when the control enters and leaves a function. This option is useful when you want to test how a particular condition affects a function.

■ **Separate Entry/Exit Criteria**: Different conditions are defined for function entry and exit. Use this option when you want to test the return value of the function. For example, if you are aware that the value of `object` changes during the course of execution of the `myTest` function, but are only interested in knowing when it becomes `0`, you can set the exit criteria to `object==0`.

Related Topics

What are Breakpoints?

Setting Unconditional Breakpoints

Setting Conditional Breakpoints

Clearing Breakpoints

Managing Breakpoints

Setting Unconditional Breakpoints

When you execute a program on which breakpoints have been set, the debugger stops just before executing the line of code that contains the breakpoint and highlights it in yellow. At this point, you can evaluate variables, set more breakpoints, or use other debugging features.

Setting Unconditional Line Breakpoints

To insert breakpoints on a particular line in your code:

➡ In the source code pane, click the line of code where you want to set a breakpoint and do one of the following:

❑ Click in the left margin of the program statement.

❑ Right-click the program statement and choose *Set/Unset Breakpoint* from the context-menu. The *Line breakpoint* dialog box displays. Click *Enabled* and then *OK* to set an unconditional line breakpoint.

A 🛑 sign displays in the left margin of the source code pane, next to the selected line of code, indicating that an unconditional breakpoint has been set for that line.

Note: You can set breakpoints only on executable SKILL statements. You can not set a breakpoint on a comment line, a blank line, or a line containing only parenthesis.

Note: Setting breakpoints on statements containing the following SKILL functions is not allowed: `append`, `append1`, `car`, `cons`, `copy`, `list`, `listp`, `arrayref`, `defstructp`, `strcmp`, `strlen`, `strncmp`, `substring`, `difference`, `fixp`, `eq`, `equal`, `memq`, `nequal`, `null`, `mapcan`, `mapcar`, `return`, `boundp`, and `apply`.

Setting Unconditional Function Breakpoints

You can set a breakpoint on a function so that the breakpoint is triggered every time the function is called. Such breakpoints are also called entry breakpoints because these are set on the function entry points. As with line breakpoints, you can use the *Set/Unset Breakpoint* command to insert function entry breakpoints.

➡ In the source code pane, click the line of code containing the function on which you want to set a breakpoint and do one of the following:

❑ Click in the left margin of the program statement.

❑ Right-click the program statement and choose *Set/Unset Breakpoint* from the context-menu. The *Function breakpoint* dialog box displays. Click *Enabled* and then *OK* to set an unconditional function breakpoint.

Note: You can also set an unconditional function breakpoint using the Search toolbar. In the Search toolbar, choose *Set Breakpoint* from the first drop-down list box. Then, specify the name of the function you want to set the breakpoint on in the second drop-down list box, and press Enter.

A sign displays in the left margin of the source code pane, indicating that an unconditional breakpoint has been set for the selected function.

Related Topics

What are Breakpoints?

Configuration of Conditional Breakpoints

Setting Conditional Breakpoints

Clearing Breakpoints

Setting Conditional Breakpoints

When you specify conditions for breakpoints, the debugger stops only when the breakpoint is triggered and its associated condition is met. If the condition evaluates to false, the program continues to run.

Note: You can set break conditions on lines as well as functions.

Setting Conditional Breakpoints on Lines

To set a conditional breakpoint on a line:

1. In the source code pane, click the line of code where you want to set a conditional breakpoint and do one of the following:

2. Right-click in the left margin of the source code pane and choose *Set/Modify Conditional Breakpoint* from the context menu.

   The *Line breakpoint* dialog box displays.
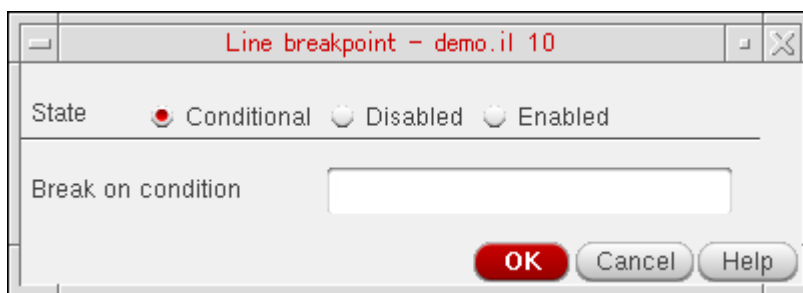
3. Select an appropriate breakpoint state. The available options are: *Conditional*, *Disabled*, and *Enabled*.

4. In the *Break on condition* field, specify the condition that you want to evaluate when the breakpoint is reached.

5. Click *OK*.

The (conditional) sign displays in the left margin of the source code pane next to the line of code, indicating a conditional breakpoint.

Setting Conditional Breakpoints on Functions

If the function on which you set the breakpoint is a regular function, the breakpoint is set only on that function; if it is a SKILL++ method (with a `defmethod` declaration), then the breakpoint is set only on that particular method's declaration; and if it is a generic function (with a `defgeneric` declaration) then the breakpoint is set on all methods that belong to that function.

If you do not specify a valid entry or exit criteria in the *Function breakpoint* dialog box, a warning message displays in the CIW, indicating an incomplete breakpoint criteria. Specify the criteria again.

To set a conditional breakpoint on a function:

1. In the source code pane, click the line containing a program function.

2. Right-click in the left margin of the source code pane and choose *Set/Modify Conditional Breakpoint* from the context-menu.

   The *Function breakpoint* dialog box displays.
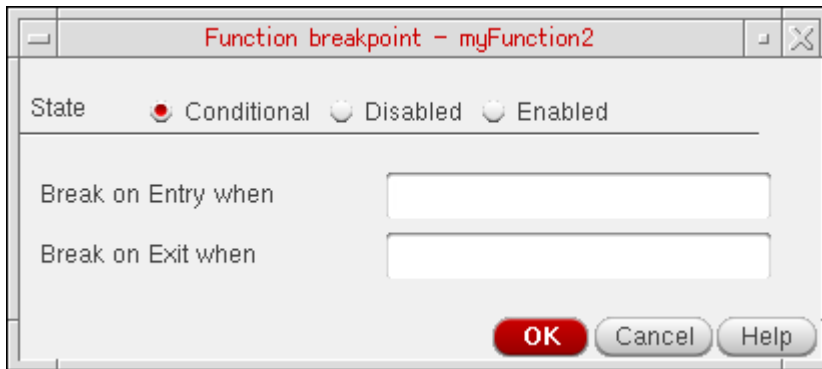
3. Select an appropriate breakpoint criteria. The available options are: *Conditional*, Disabled, and *Enabled*.

4. Specify when the breakpoint condition needs to be evaluated. You have the following options:

   ❑ In the *Break on Entry when* field specify the condition that is to be evaluated when the function is called.

   ❑ In the *Break on Exit when* field specify the condition that is to be evaluated when the function returns control to the statement following the call.

5. Click *OK*. When finished, a conditional breakpoint icon displays in the left margin of the source code pane.

Related Topics

What are Breakpoints?

Configuration of Conditional Breakpoints

Setting Unconditional Breakpoints

Clearing Breakpoints

Managing Breakpoints

Enabling and Disabling Conditional Breakpoints

Clearing Breakpoints

If you no longer need a breakpoint, you can remove it. Once you remove the breakpoint, the debugger no longer stops the execution at that point.

Clearing Unconditional Breakpoints

To remove an unconditional breakpoint from a specified line or function:

➡ In the source code pane, click in the line of code which contains the breakpoint:

❑ Click the left-margin of the line containing the breakpoint.

❑ Select the breakpoint in the *Breakpoints* assistant and click *Delete*.

The toggle action removes the breakpoint from the code and as a result, the sign disappears.

Clearing Conditional Breakpoints

To clear a conditional breakpoint, in the source code pane, click in the line of code which contains the breakpoint. Then, do one of the following:

❑ Right-click in the left-margin of the line containing the breakpoint and choose *Remove Conditional Breakpoint* from the context menu.

❑ Select the breakpoint in the *Breakpoints* assistant and click *Delete*.

The toggle action removes the breakpoint from the code causing the conditional breakpoint icon to disappear.

Clearing all Breakpoints

To clear all breakpoints, choose *Debug – Remove All Breakpoints* or click *Delete All* in the *Breakpoints* assistant.

Related Topics

What are Breakpoints?

Configuration of Conditional Breakpoints

Setting Unconditional Breakpoints

Setting Conditional Breakpoints
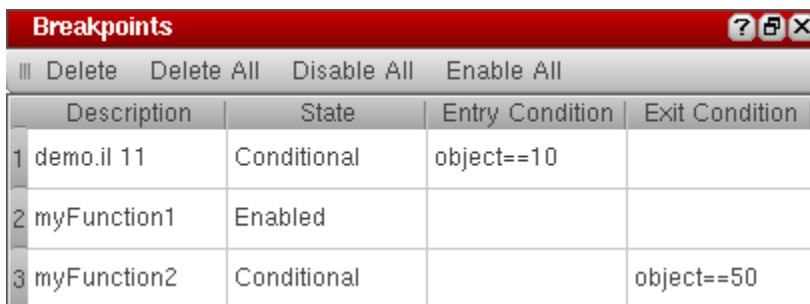
Managing Breakpoints

Managing Breakpoints

Use the Breakpoints assistant to view the list of breakpoints currently set in your code, or change their state or condition. The Breakpoints assistant lists both line and function breakpoints.

To view or edit the list of breakpoints in the Breakpoints assistant:

1. Choose *Window – Assistants – Breakpoints*. The Breakpoints assistant displays.

Breakpoints assistant lists all current breakpoints even if the code file containing the breakpoints is not currently open in the SKILL IDE.

2. To edit a breakpoint, click the line containing the breakpoint in the Breakpoints assistant.

   You can only change the *State*, *Entry Condition*, and *Exit Condition* for a breakpoint.

3. To change the *State* of the breakpoint, double-click the current state and select a different option from the drop-down list-box. The available options are: *Disabled, Enabled*, and *Conditional*.
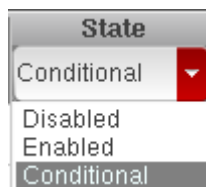
4. To change the entry condition for the breakpoint, double-click the current *Entry Condition* and specify a new condition. The debugger will halt the program execution when the entry condition evaluates to true.

5. To change the exit condition for the breakpoint, double-click the current *Exit Condition* and specify a new condition. The debugger will halt the program execution when the exit condition evaluates to true.

   Exit Conditions are not applicable for line breakpoints.

Breakpoints Assistant

The following table describes the Breakpoints assistant buttons for deleting, disabling, or enabling breakpoints.

Column	Description
Delete	Deletes the selected breakpoint from the code.
	Hold the CTRL key to select multiple breakpoints or hold the SHIFT key to select a contiguous range of breakpoints for deletion.
Delete All	Deletes all breakpoints from the code.
Disable All	Disables all existing breakpoints in the code.
Enable All	Enables all existing breakpoints in the code.

Related Topics

What are Breakpoints?

Configuration of Conditional Breakpoints

Setting Conditional Breakpoints

Setting Unconditional Breakpoints

Clearing Breakpoints

Stepping through Code

After the debugger encounters a breakpoint and pauses program execution, you can step through the rest of the program statements, one statement at a time. You can use the *Next*

command to advance to the next executable statement or you can use the *Step* and *Step out* commands for stepping through the code.

Choose *Debug – Step* to instruct the debugger to execute the next line of code. If it contains a function call, the debugger executes the function call and stops at the first line in the function. Choose *Debug – Step Out* if the control is with a function and you need to return to the calling function. The *Step Out* command causes the function code to be executed until the function returns the control.

The debugger must be in the break mode to use the *Step* and *Step Out* commands. If you step through the program statements when `debugMode` is off, some warning messages might appear in the CIW. To avoid the warning messages, set `<function-name>.debugMode=t`.

Choose *Debug – Continue* to resume the program execution after a breakpoint is encountered.

Related Topics

Managing Breakpoints

Terminate Program Execution

While debugging your code, you might need to terminate the execution after reaching a certain point in the code. SKILL IDE enables you to do that by either terminating the entire debugging session (using the *Stop All Debugging* menu command) or terminating only the program under execution (using the *Stop Current Top-Level* menu command).

For example, if SKILL IDE has nested top-levels, you can not continue debugging of parent top-level until the nested top-level returns control. In such cases, you can choose *Debug – Stop Current Top-Level* to exit the currently executing function, so that the control returns to the calling function.

In addition, if you have multiple programs loaded, using *Stop Current Top-Level* causes the debugger to stop the program at the top of the execution stack. For example, if you have the following SKILL programs:

The first file, `prog1.il` has the following contents:

```
(procedure Loop1Function()
    for( i 1 100
        printf("in function1, loopctr is %d...\n" i)
    )
)
```

The second file, `prog2.il` has the following contents:

```
(procedure Loop2Function()
    for( j 1 20
        printf("in function2, loopctr is %d...\n" j)
    )
)
```

If you load both the files one after the other, set breakpoints on the `printf` statements in both the programs, and execute functions `Loop1Function` and `Loop2Function`, the debugger pauses the program execution at the respective breakpoint conditions. If you then choose *Debug – Stop Current Top-Level* the debugger exits `Loop2Function` and resumes execution of `Loop1Function`.

If you want to exit the debugging session, choose *Debug – Stop All Debugging*. The *Stop All Debugging* command terminates all programs on the execution stack.

SKILL IDE separates one execution session from another and displays the session hierarchy in the order of execution. In the *Stack Window* assistant, the most recent session is expanded and is displayed at the top of the execution stack.

Related Topics

Managing Breakpoints

4

Program Data Analysis

Use the SKILL IDE debugger to inspect the call stack, examine or change the values of variables, and analyze the state of your program.

Examining and Modifying Variable Values

When the program encounters a breakpoint, you can evaluate the program variables or change their values to examine "what if" scenarios using the debugger.

1. To view the values of program variables and function description within the current scope, select *Options – Show Value*.

2. Place the mouse cursor over a variable or function in the source code pane displays the corresponding value in a tooltip as illustrated in the following image.
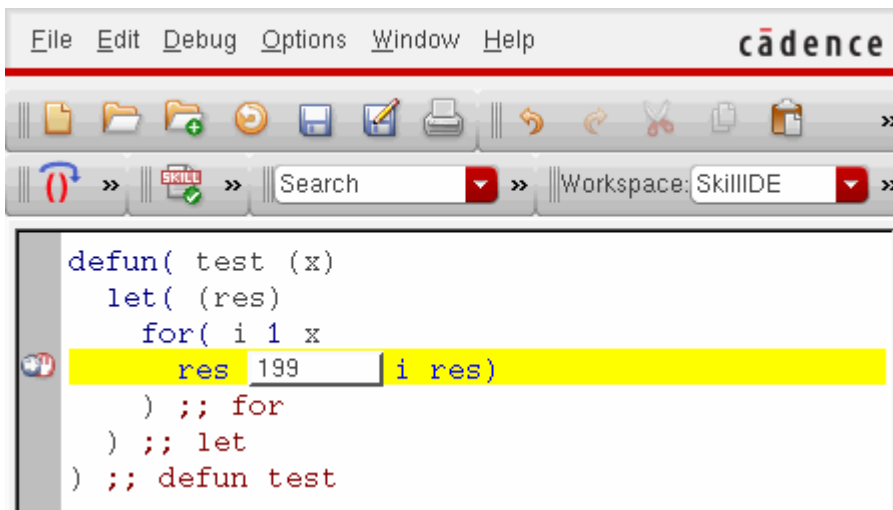
The following image illustrates a program variable that has multiple values, which are displayed as a list with scroll bars:
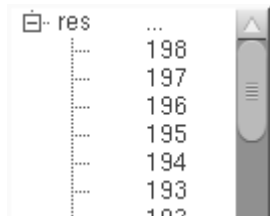
In tooltips, values of string variables display within double quotation marks and values of undefined variables display as unbound. Also, when you hover the mouse pointer over a function name, the tooltip prints the function syntax (from the SKILL Finder database, if available) or the function arguments list (for the user-defined functions).

Related Topics

Functions and Variables Tracing

Functions and Variables Tracing

The values of functions and variables in your code might change as you run the code. For such functions and variables, rather than watching the values in tooltips, you can use the *Trace* assistant to view the changing values. Using the *Trace* assistant, you can continually inspect the values of functions and variables as you step through the program, and view their scope. You can also view the functions that are on top of the call stack, along with their arguments and local variables.

As the program progresses, variables go in and out of scope. The scope information is useful to trace variables that are defined with same name within different scopes in a program. For such variables, *Trace* assistant displays the values of the variable in different scopes.

Tracking Function Calls

When your program stops at a breakpoint, the information about the function on the top of the call stack is displayed in the *Trace* assistant window. This information includes the arguments of the function in the stack frame as well as the local variables. When a new function is called, it gets added to the top of the call stack and the current function returns to the caller, and thus gets removed from the top of the call stack. To track function calls:

1. Choose *Window – Assistants – Trace*. The Trace assistant displays to the right of the source code pane and is initially empty.
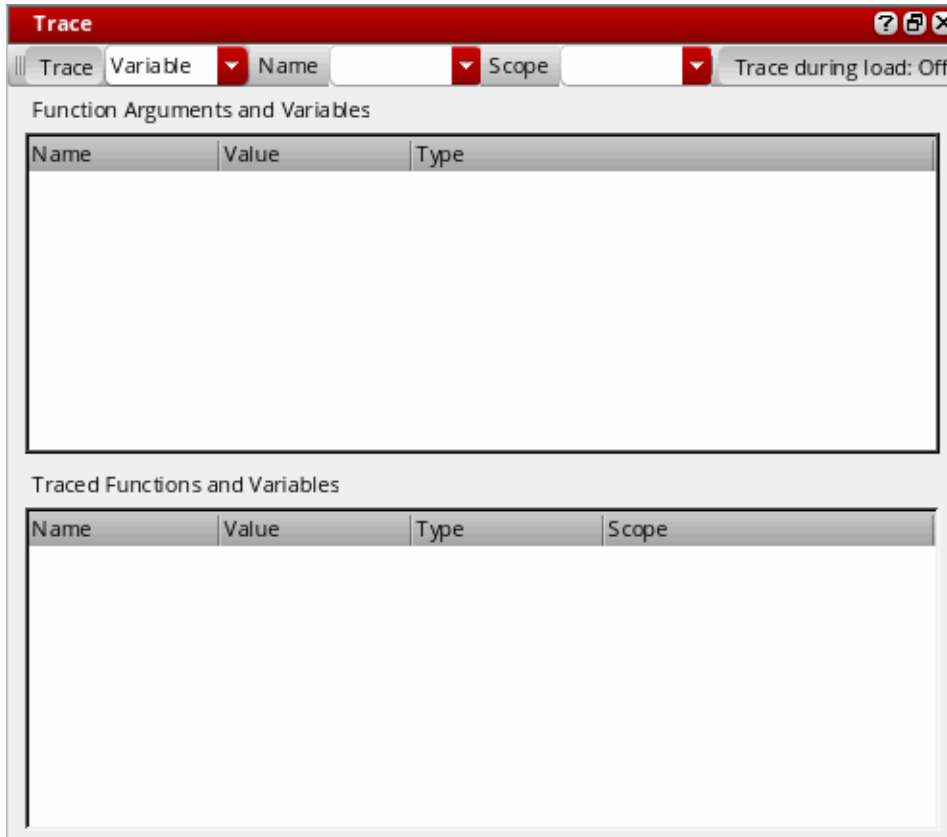
By default, tracing is disabled during file load operations. Consequently, the trace functionality will be disabled if you reload a file that is currently open in the SKILL IDE editor but has been edited using some other editor. To enable tracing during file load, click *Trace during load: Off*

2. Execute a function in your code. The *Function Arguments and Variables* section of the Trace assistant window gets updated with the name of the function on top of the call stack, along with its arguments, local variables, and their values.
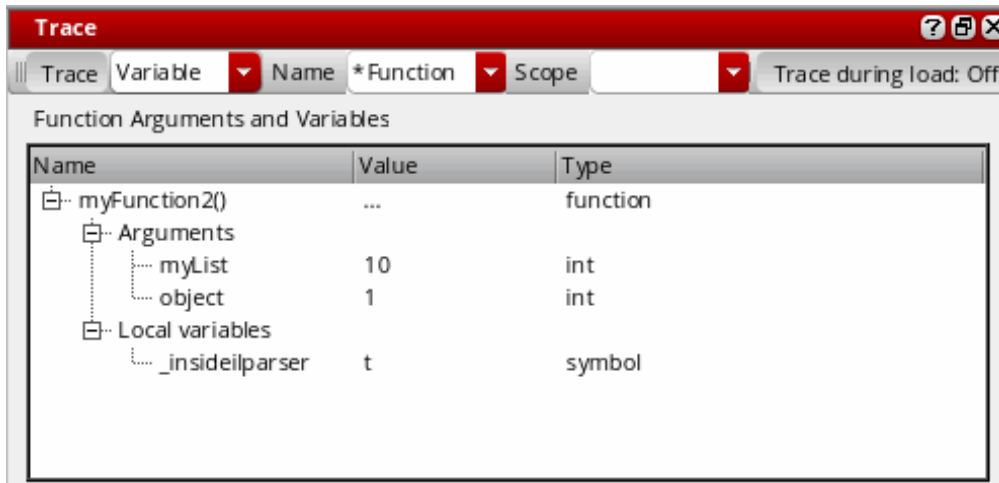
The values in the *Function Arguments and Variables* section will change as you step through the rest of the program statements.

To stop tracking the function, select the function and choose *Clear* from the context menu.

Tracking Changes in Program Variables

To monitor the changes in your program variables as your program executes, you can select the variables from the source code pane and add them to the Trace assistant. To trace a variable:

1. Choose *Window – Assistants – Trace*. The Trace assistant displays to the right of the source code pane and is initially empty because you are yet to add a trace variable or function.
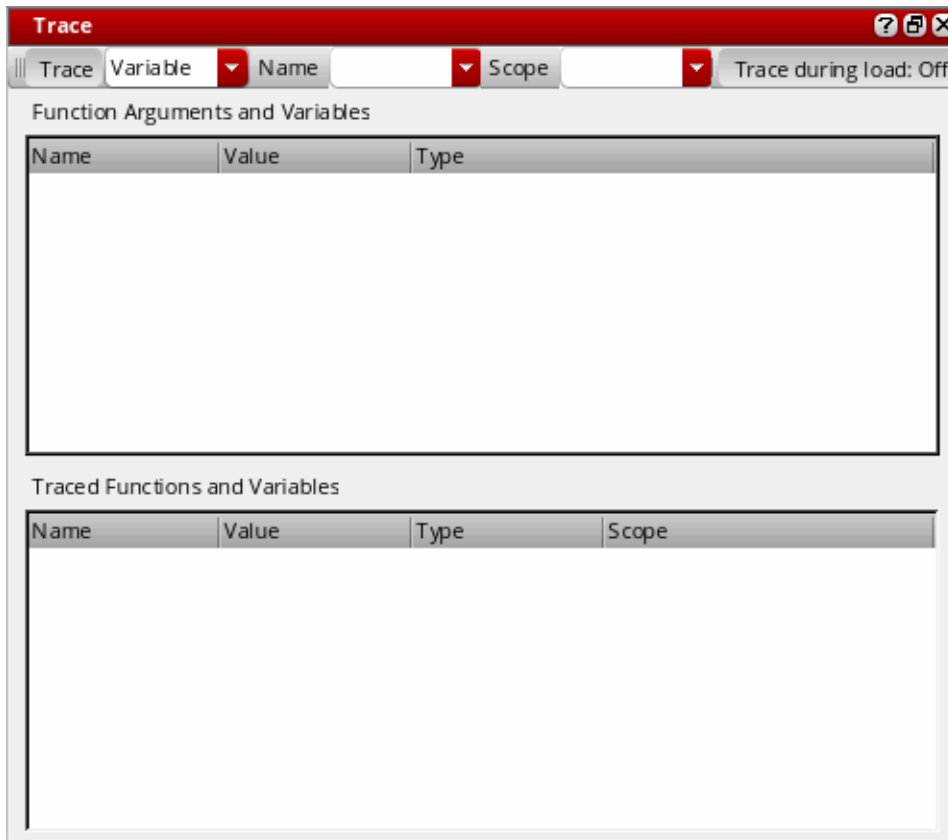
2. Select *Variable* or *Function* from the *Trace* drop-down list box and then, specify the name of the variable or function to be traced in the *Name* drop-down list box. If you are tracing a variable, you can specify the scope of the traced variable in the *Scope* drop-down list box.

Alternatively, select the variable or function that you want to trace in the source code pane and right-click and choose *Trace* from the context-menu.

You can use regular expressions in the *Name* drop-down list box, so that all functions or variables that match that expression are traced.

The name, value, data type, and scope of the selected variable or function display in the *Traced Functions and Variables* section of the Trace assistant window. The values of these variables are updated as you step through your program.
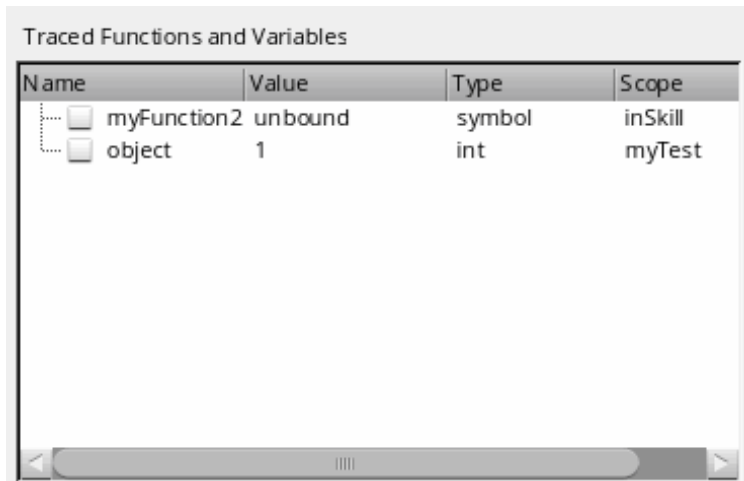
3. Select the check box next to the name of a variable to add it to watch list, so that you can track when the value of the variable changes.

4. To remove a variable from the Trace assistant, select the variable in the Trace assistant and choose *Untrace Selected* or *Untrace Selected (All Scopes)* from the context menu.

   Untrace All removes all functions and variables from the Trace assistant.

Changing Variable Values

To edit the values of variables at runtime by using the Trace assistant:

1. While the program is still in the debug mode, choose *Window – Assistants – Trace* to display the Trace assistant.

2. Double-click a variable value to edit it. Type a new value for the variable and press `Enter`.

   You can also place the pointer over a variable in the source code pane and click the tooltip to change its value. Press `Esc` to discard the edits.

Related Topics

Examining the Call Stack

SKILL IDE Scope Information

SKILL IDE uses scope information to determine the value of a variable. Scope defines the visibility of a variable within a code block. As the program progresses from one code block to another, the visibility of the variables defined in the code blocks changes. If a variable is defined inside a function or procedure block, its scope is said to be local to that function or procedure block. If however, the variable is defined at the program level, it has a global scope.

When the debugger hits a function breakpoint, the values of the traced variables within the current scope display. When the execution control exits the function block, the variables local to that function block go out of scope.

Because the debugger uses the scope information to determine the value of a variable, it is possible to have both global and local variables with the same name. For example, in the sample program given below, variable `x` is used at two places, both in function `testBreakpoints()` and `testBreak()`.
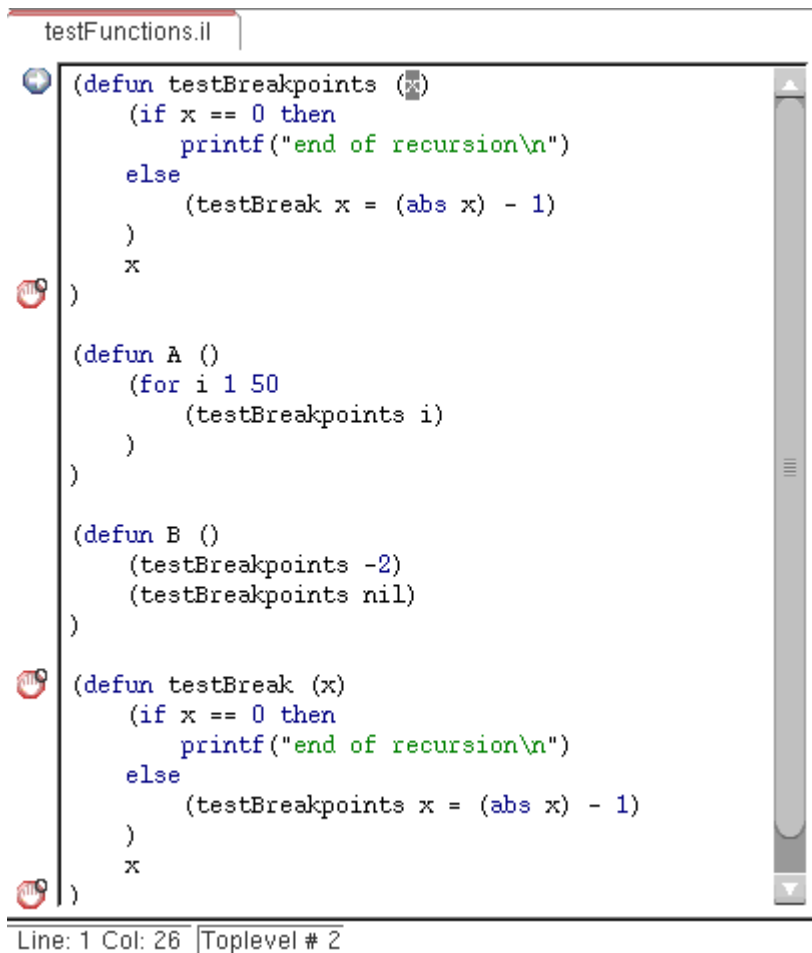
```
* Sample program- Understanding the scope of variables
*************************************************/
(defun testBreakpoints (x)
    (if x==0 then
        printf("end of recursion\n")
    else
    (testBreak x = (abs x) - 1)
    )
    x
)
(defun A()
    (for i 1 50
        (testBreakpoints i)
    )
)
(defun B()
    (testBreakpoints -2)
    (testBreakpoints nil)
)
(defun testBreak (x)
    (if x==0 then
        printf("end of recursion\n")
    else
    (testBreakpoints x = (abs x) - 1)
    )
    x
 )
```

To see how the scope of a variable changes with respect to the function currently being executed, set the entry and exit condition for both the functions (`testBreakpoints()` and

testBreak()`)` to `t`. Call testBreakpoints() with the value `9` and then trace the variable `x`.

Observe the value and scope of x as it changes with respect to the function currently being executed.
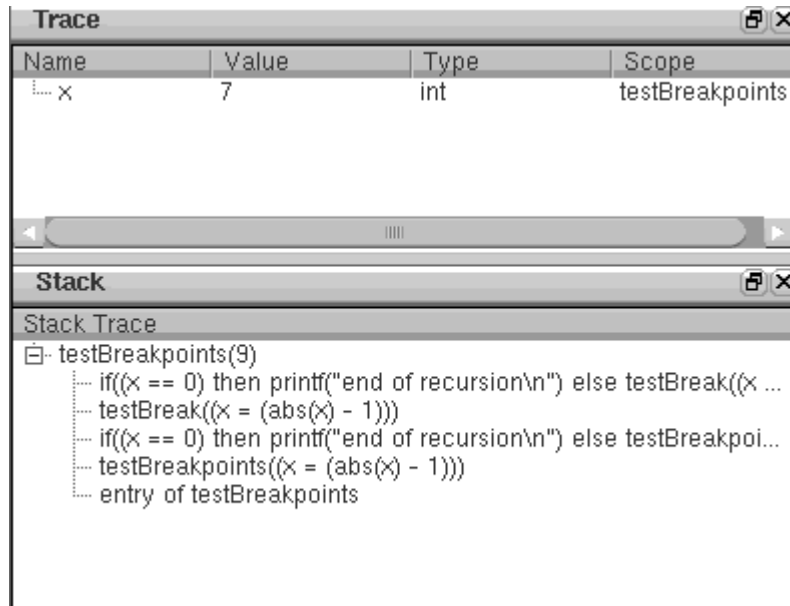
Note: If you are debugging multiple files with same variable names, *Trace* assistant displays the debug information of the traced variables of the program under execution.

Related Topics

Efficiency Improvement of SKILL Code

Examining and Modifying Variable Values

Functions and Variables Tracing

Examining the Call Stack

The call stack represents function calls that are currently active in the program being debugged. In the call stack, functions and their arguments are listed in the order in which they were called. Every time a function call is made, a new stack frame is pushed on the call stack. The most recently called function is at the top of the call stack.

By examining the call stack, you can trace the flow of execution, identifying the function calls that resulted in errors.

Displaying the Call Stack

To view the current call stack:

➡   choose *Window – Assistants – Stack*. The *Stack* assistant displays.
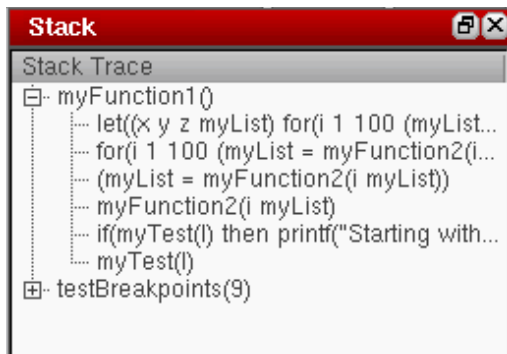
Moving Through the Call Stack

You can move up and down the call stack by clicking individual function names. SKILL IDE updates the source code pane to display the definition of the selected function. For example, if you click the `testBreakpoints(9)` in the call stack shown above, the source code pane displays the code containing the definition of `testBreakpoints()`. If the file containing the definition of the selected function is not already open, then it gets opened.

Related Topics

Efficiency Improvement of SKILL Code

Examining and Modifying Variable Values

SKILL IDE Scope Information

Class Inheritance Relationships

You can use the Class Browser to understand and follow the class inheritance hierarchy of the classes used in your SKILL code. You have the option to view the subclasses, slot definitions, and superclasses of a class.

Any `@reader`, `@writer`, `@initarg`, and `@initform` slot options you used for initializing the slots can also be viewed in the *Class Browser* assistant. All instances of a given class will have the same slots. If a subclass is inherited from a superclass, it also inherits the slots of the superclass. For more information on class inheritance concepts, see <u>SKILL++ Object System</u>.

Displaying the Class Hierarchy

To view the class hierarchy:

1. Choose *Window – Assistants – Class Browser*. The *Class Browser* assistant displays.
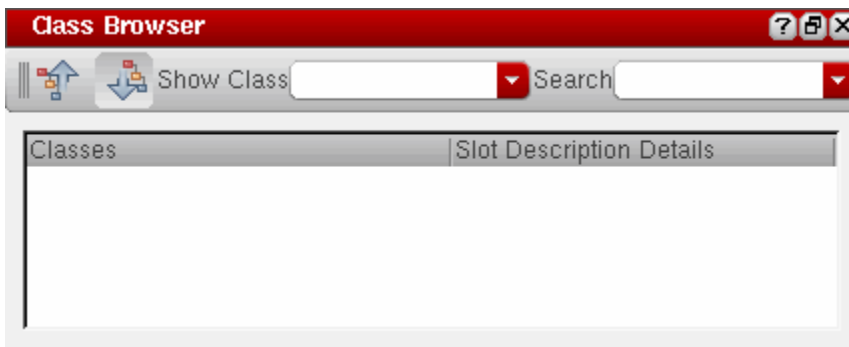
2. In the *Show Class* drop-down list box, type the class name for which you want to view the class hierarchy and press Enter. If the specified class does not exists, an error message is displayed.

3. Click [icon] to view the superclasses or [icon] to view the subclasses and slot definitions of the given class. If the superclass or subclass information exists, it is displayed as an inheritance tree in the *Class Browser* window.
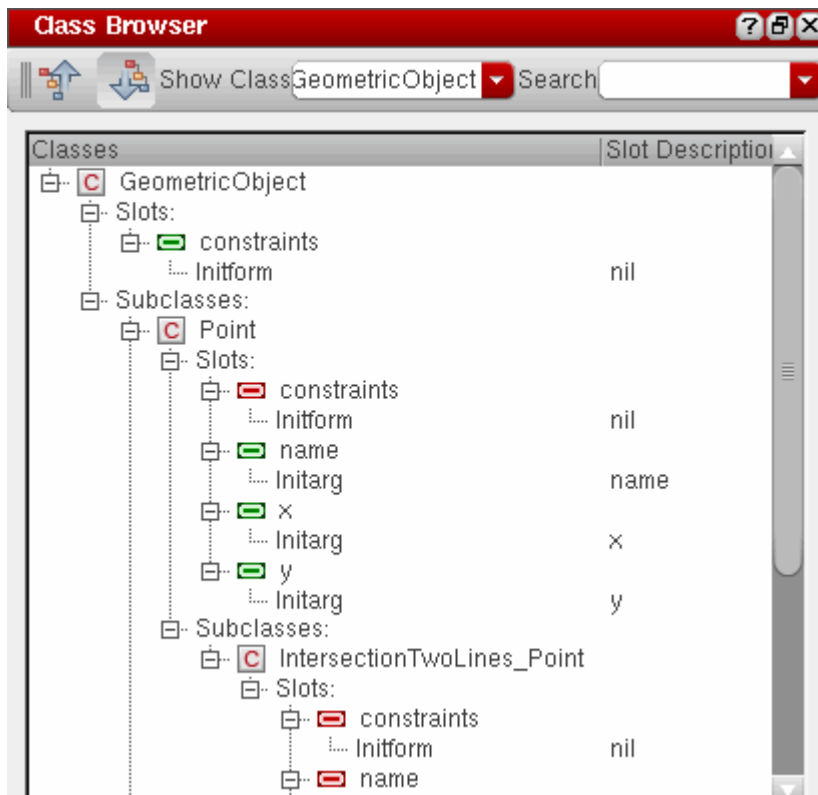
Different object types are identified by different icons in the class tree. For example, classes are identified by the [C] icon, slots defined within a given class are identified by the [□] icon, and slots inherited from a superclass are identified by the [□] icon.

   a. Right-click a class name in the current tree view and choose *Go To Source* from the context-menu to view its definition in the code. If the code file for the given class is not loaded, a warning message displays instead.

   b. Right-click a class name and choose *Go To Finder* from the context-menu to view its syntax and description in Finder.

4. Use the *Search* drop-down list box to search for class or slot names within the current tree view. Alternatively, view the class hierarchy using the *Search* toolbar, as described below:

   c. In the *Search* toolbar, choose *Show Class* from the first drop-down list box.

    d. In the second drop-down list box, type the class name for which you want to view the class hierarchy and press Enter or click the ▷ icon.

    If the class information exists, it is displayed as an inheritance tree in the *Class Browser* window.

Related Topics

Using the Method Browser Assistant

Setting Up Files/Directories for the Lint Checker

SKILL Profiler Assistant

Using the Step Result Assistant

Using the Code Browser Assistant

Using the Method Browser Assistant

Use the Method Browser assistant to view the method trees of generic functions. You have the option to view all existing methods for a generic function, all applicable methods for a generic function, or only the methods that would be called next in the current function call.

To browse the method tree for a generic function:

  1. Choose *Window – Assistants – Method Browser*. The *Method Browser* assistant displays.

  2. Select one of the following options from the *Show methods* drop-down list:

    ❑   Select *next* to view the methods that will be called next in the current function call.

❑ Select *applicable* to view all applicable methods for a given generic function. In the *for* field, type the function name for which you want to view the applicable methods and in the *args* field, type the variable that takes on the value of the method's argument at runtime.

❑ Select *all* to view all existing methods for a given generic function. In the *for* field, type the function name for which you want to view the methods.

The *next* and *applicable* options work only in the context of a debug run.

3. Depending on the option you select in the *Show methods* drop-down list, the results pane displays the method tree for the given generic function.
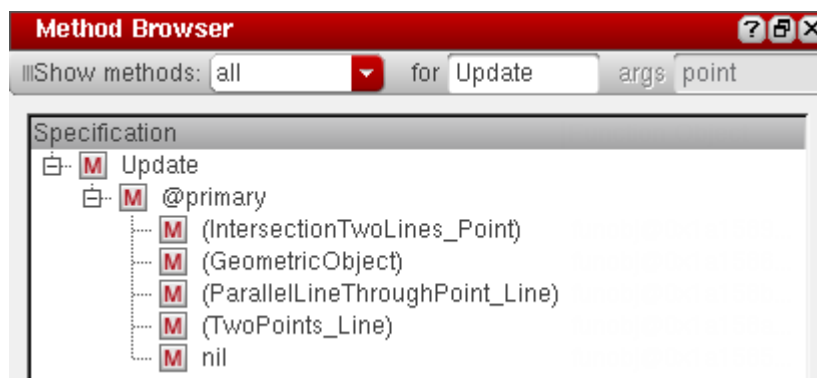
Different method types are identified by different icons in the method tree. For example, primary and top-level methods are identified by the ☐ icon, @around methods are identified by the ☐ icon, @before methods are identified by the ☐ icon, and @after methods are identified by the ☐ icon.

4. View the method tree using the *Search* toolbar, as described below:

a. In the *Search* toolbar, choose *Show Methods* from the first drop-down list box.

b. In the second drop-down list box, type the function name for which you want to view the methods and press Enter or click the ▷ icon.

The method tree for the given generic function displays in the *Method Browser* window.

Method Tree Context-sensitive menu

After the method tree is populated, you can right-click a method name and choose one of the following options:
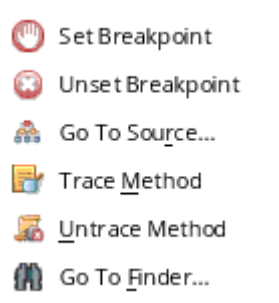
■ *Set Breakpoint*: To set breakpoints on the selected method.

■ *Unset Breakpoint*: To unset the breakpoint set on the selected method.

■ *Go To Source*: To view the method definition in the source file. If the source file for the method is not loaded in debug mode, an error message displays instead.

■ *Trace Method:* To trace the selected method using the Trace Assistant.

■ *Untrace Method*: To untrace the method selected in the *Method Browser*.

■ *Go To Finder*: To check the method definition in the Finder Assistant.

Related Topics

Using the Step Result Assistant

SKILL Profiler Assistant

Using the Code Browser Assistant

Efficiency Improvement of SKILL Code

Use SKILL Lint to examine SKILL code for possible errors that go undetected during normal testing. In particular, SKILL Lint is useful in finding unused local variables, global variables that should be declared locally, and functions that have been passed the wrong number of arguments. It also gives tips to improve the efficiency of your SKILL programs. For details, see SKILL Lint.

You can use the Lint Manager assistant to set up the files and directories for the Lint checker, set the Lint parameter values before running the Lint tool, and run the Lint tool on the selected files or directors.

Related Topics

Setting Up Files/Directories for the Lint Checker

Setting Lint Options

Running the SKILL Lint Tool

Setting Up Files/Directories for the Lint Checker

The Lint Manager assistant provides options for setting up the files and directories for the Lint checker. To add files and directories to the Lint Manager assistant:

1. Choose *Window – Assistants – Lint Manager*. The Lint Manager assistant displays.
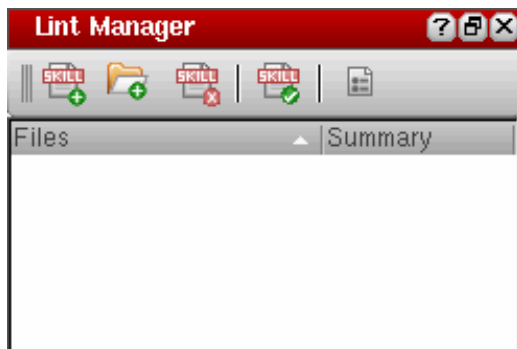
2. Click (*Add Files*) or (*Add Directory*) to specify a file or directory on which the Lint check needs to be run. Depending on your choice, the SKILL Lint File Select Dialog or SKILL Lint Directory Select Dialog dialog box displays.

   Only files with the extension `.il`, `ils`, or `.scm` should be selected for lint check in the SKILL Lint File Select Dialog*.*

3. Browse to select the file or directory you want to run the Lint checker on and click *Choose*. The specified files and directories are added to the Lint Manager assistant.

   If you add to the Lint Manager assistant first a file, by clicking (*Add Files*), and then its parent directory, by clicking (*Add Directory*), the file is automatically listed under the parent directory in the Lint Manager assistant tree structure.

   Select a file or directory listed under *Files* and click (*Remove Files/Directories*) to remove it from the *Lint Manager* assistant. Hold the CTRL key to select multiple files/directories or hold the SHIFT key to select a contiguous range of files/directories for deletion.

Related Topics

Setting Lint Options

Running the SKILL Lint Tool

Setting Lint Options

The *Lint* command lets you set the Lint parameter values before running the Lint tool. To set the Lint parameters:

1. Select *Options – Lint* or click 🔳 (*Lint Options*) in the in the *Lint Manager* assistant window or 🔳 (*Options*) in the Edit toolbar. The *Options* assistant displays with *Lint* as the active tab.
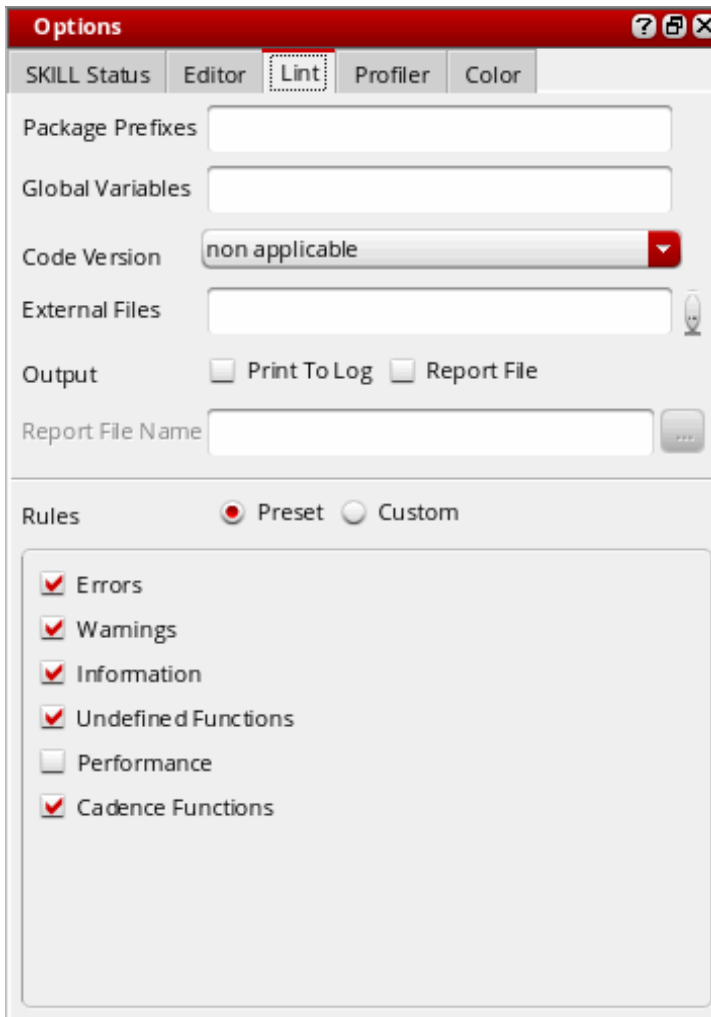
2. In the *Package Prefixes* field, type a list of acceptable package prefixes for functions and global variables (for example, `tr`). SKILL Lint notes any variables that do not have one of the prefixes you typed.

As prefixes are not normally used on local variables, you can find variables that you meant to declare as local but they have a prefix. You can also use this field to determine whether your SKILL program uses a global from some other program. See Check Function and Global Variable Prefixes for more information.

3. In the *Global Variables* field, type the list of global variables in the code that SKILL Lint should check. SKILL Lint suppresses `VAR8` warnings for the global variables specified in this field.

4. From the *Code Version* drop-down list, select the release version of the code you want SKILL Lint to check.

The checks for Cadence SKILL functions available in a particular release are applicable only when the *Cadence Functions* option is selected in the *Rules: Preset* section.

5. In the *External Files* field, browse or type a list of contexts or files that contain the macro definitions on which the code under analysis depends. This is used for loading external definitions files for functions and macros.

6. In the *Output* section, select one or both of the check boxes to specify where you want to direct the SKILL Lint output.

If you select both check boxes, output appears in the output window after SKILL Lint writes it to the `CDS.log` file and the CIW. If you want SKILL Lint to write its output report to a file, use the *Report File Name* field (below).

7. In the *Report File Name* field, browse or type the name of the file to which you want to write the SKILL Lint output report.

errors	Indicates the number of errors
general warnings	Indicates the number of general warnings.
top level forms	Indicates the number of expressions in the input file.
IQ score	= 100 - [25*(number of short list errors) + 20*(number of long list errors) / (number of top level forms)] See SKILL Lint PASS/FAIL and IQ Algorithms.

A line in the output report contains the following information:

❑   Message group priority, usually abbreviated and capitalized (for example, `INFO`).

❑   Built-in message name, in parentheses and capitalized (for example, `(REP110)`).

❑   Message text (for example, `Total information: 0.`).

8. In the *Rules* section, click *Preset* to apply system-defined rules or *Custom* to apply user-defined rules to the SKILL Lint report.

If you click *Preset*, select one or more of the following check boxes to specify the message groups you want SKILL Lint to report: *Errors*, *Warnings*, *Information*, *Undefined functions*, *Performance*, and *Cadence Functions*.

Cadence public functions should start with a lowercase character; the custom functions should start with an uppercase character.

The *Cadence Functions* check box turns the `ALERT` message group in the *Custom* section on or off. If turned on, SKILL Lint reports code that might result in an alert when the code is run.

If you click *Custom*, a hierarchical tree of the available types of error reporting mechanisms displays in the area within the *Rules* section. Select the check boxes adjacent to the different <u>classes of messages</u> you want SKILL Lint to report.
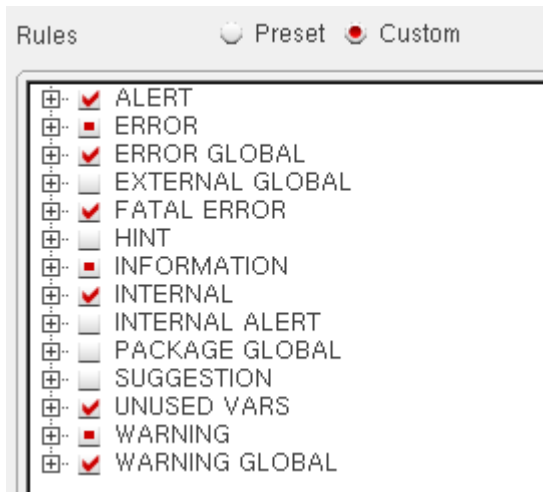
9. Click the + sign to expand the corresponding message group as shown in the figure below.
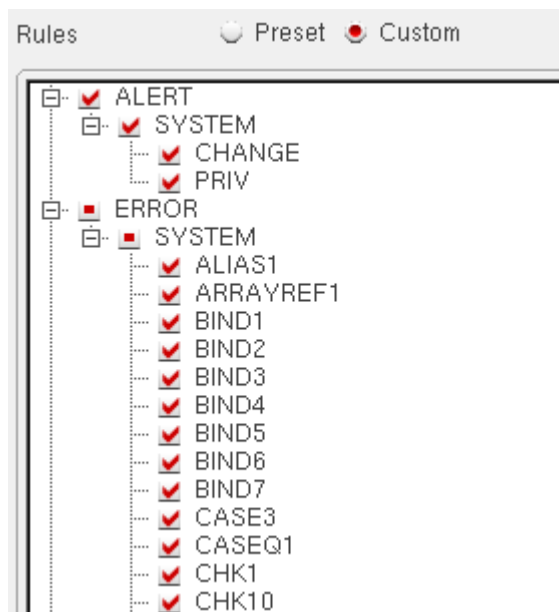
Selecting the check box corresponding to a <u>message group</u> selects all <u>built-in messages</u> under it. However, selecting a specific built-in message does not select the entire message group.

Related Topics

<u>Setting Up Files/Directories for the Lint Checker</u>

<u>Running the SKILL Lint Tool</u>

Running the SKILL Lint Tool

You can run the Lint checker tool either on the files and directories set up using the *Lint Manager* assistant, or the currently open file in the source code pane.

The Lint checker tool does not run on blank files, that is, files containing only spaces, tabs, or return characters.

Running Lint Checker on Multiple Files and Directories

After you have set up the files and directories for the Lint checker, you can run the Lint tool on the selected files or directories. To do so:

1. Click ![icon](Run Lint Tool) in the Lint Manager assistant or ![icon](*Lint all tabs*) in the Lint toolbar. The SKILL Lint Progress Box displays.
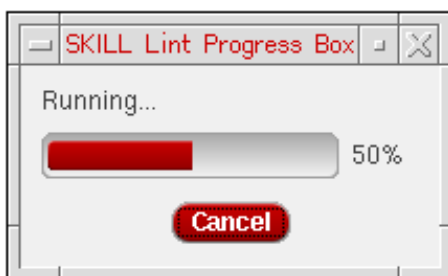
2. After the Lint tool has run, the Lint Manager assistant is updated with the summary of the run result. You can then view the Lint output report in the Lint Output window or the CIW.

To see the Lint report for a particular SKILL file, select the file name from the Lint Manager assistant. The Lint Output window is updated with the Lint report of the selected file.
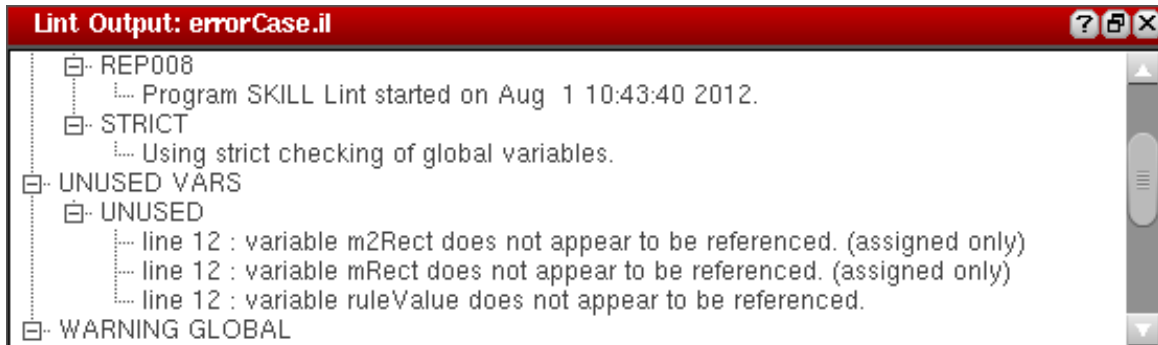
The Lint output report is organized into a tree structure with nodes representing errors, warnings, suggestions, hints, information, and unused variables. You can expand the nodes in the Lint Output window and click a message to view the corresponding code in the Source code pane. For example, if the Lint Output window reports that a variable is unused at line 12, you can click the message under UNUSED VARS and view the corresponding code in the source code pane.

Running Lint Checker on the Currently Open File

To run the Lint tool for the currently open file:

1. Click (*Lint current tab*) in the Lint toolbar.

2. After the Lint tool has run, the *Lint Output* window is updated with the run result.
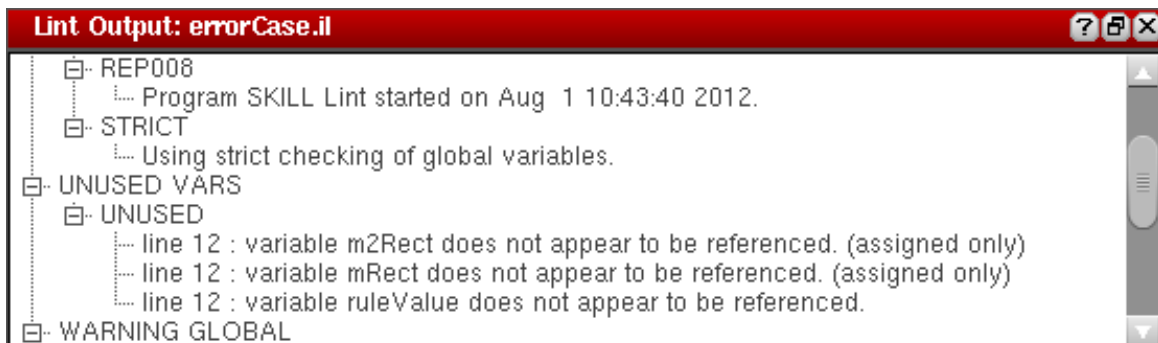
Lint tool can be run on the file currently open in the editor, even if the file has not been saved.

Related Topics

Setting Up Files/Directories for the Lint Checker

Setting Lint Options

Using the Finder Assistant

Use the Finder assistant to view the abstract and syntax statements for all SKILL/SKILL++ elements like classes, functions, and methods.

The graphical user interface and functionality of the Finder assistant is similar to the *Cadence SKILL API Finder*, which can be accessed from the CIW by choosing *Tools – SKILL API Finder*. For detailed information, see SKILL API Finder.

To view the syntax and description of a particular SKILL/SKILL++ object, use one of the following options:

■ Choose *Window – Assistants – Finder*.

■ Right-click the function name in the source code pane and choose *Finder* from the context-menu.

■ Right-click a class name in the Class Browser assistant and choose *Go To Finder.*

■ Right-click a method name in the Method Browser assistant and choose *Go To Finder.*

■ In the Search toolbar, choose *View Finder Doc* from the first drop-down list box. Then, specify the name of a SKILL/SKILL++ function in the second drop-down list box and press Enter.

The *Finder* assistant displays.
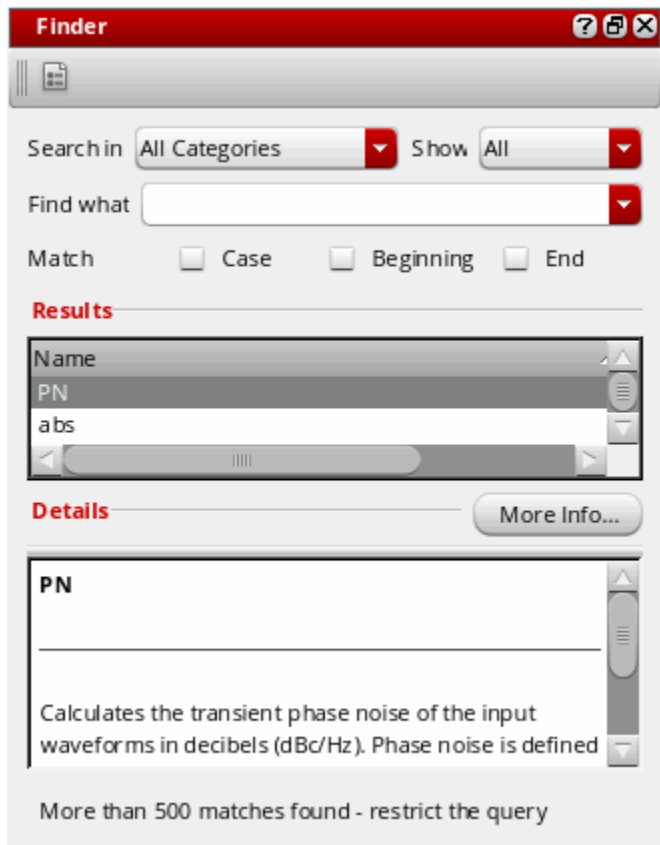
Use the *Show* drop-down list to select a SKILL/SKILL++ object for viewing. You can choose to view *All*, *Classes*, *Functions*, or *Methods*. The default is *All*.

Related Topics

Starting SKILL API Finder

Searching in SKILL API Finder

Types of Searches in SKILL API Finder

Viewing and Saving Syntax and Description of Matches Found

Problem Troubleshooting in SKILL API Finder

Using the Code Browser Assistant

Use the Code Browser assistant to browse the calling trees of user-defined functions. It helps you determine what child functions are called by the parent functions. You can expand the entire tree or one node at a time. You can also view the function definition for a user-defined function.

📽 Video

> For a video demonstration on Code Browser assistant, see <u>Browsing Your Code Using SKILL Code Browser</u> on Cadence Online Support.

To view the function tree of a user-defined function:

1. Access the Code Browser assistant using one of the following options:

   ❑ Choose *Window – Assistants – Code Browser*.

   ❑ Right-click the function name in the source code pane and choose *Go To Code Browser* (🖳) from the context-menu.

   The *Go To Code Browser* (🖳) context-menu item gets enabled for selection only after you load the open file.

2. In the *Function* drop-down list, type the name of the function you want to display. The expanded function appears in the results pane.
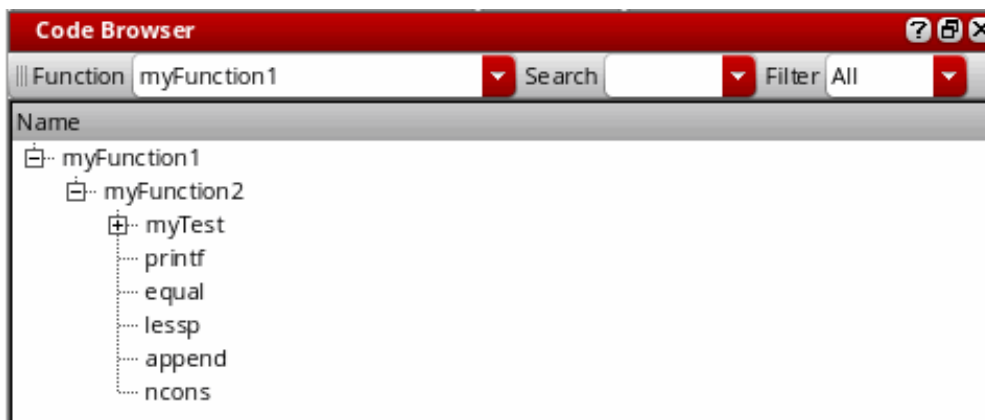
> You can use the *Filter* drop-down list to filter the results by *User* (functions that are neither binary nor write-protected), *System* (non-user binary SKILL functions), or *All*.

You can right-click a function name in the results pane and select one of the following options:

- *Go To Source*: To view the definition of the function in the source code pane.

- *Expand Deep*: To display all user-defined functions recursively until the entire calling tree is expanded.

- *Collapse*: To collapse the tree and remove all functions called by the selected function from the results pane.

- *Remove*: To remove the selected function from the results pane.

Related Topics

Using the Method Browser Assistant

SKILL Profiler Assistant

SKILL Profiler Assistant

Use the Profiler assistant to check the time and the memory consumption of your SKILL programs. You can use the Profiler assistant to accomplish the following:

■ Measure the time spent in each function.

■ Show how much SKILL memory is allocated in each function.

■ Measure performance without having to modify function definitions.

■ Display a function call tree of all functions executed and the time or memory spent in those functions.

■ Filter functions so you can see only specific functions.

To open the Profiler Assistant, choose *Window – Assistants – Profiler*. The Profiler assistant displays.
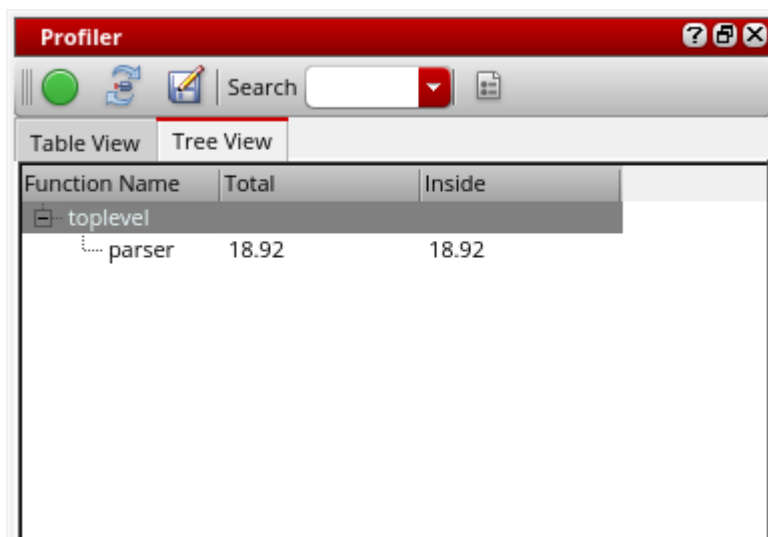
Video

For a video demonstration on Profiler assistant, see <u>Analyzing Your Code Using SKILL Profiler Assistant</u> on Cadence Online Support.

For more information of the profiler functions, see <u>Profiler Functions</u>.

Related Topics

<u>Command Line Interface for SKILL Profiler</u>

Setting Profiler Options

Use the options in the Profiler tab to define the type of data you want the Profiler to collect and display.

Profiler Options for Data Collection

To set the profiler options for collecting data, before running the Profiler, do the following:

1. Choose *Options – Profiler*. You can also click 📄 (*Profiler Options*) in the toolbar displayed in the *Profiler* assistant window. The Options assistant displays with *Profiler* as the default tab.
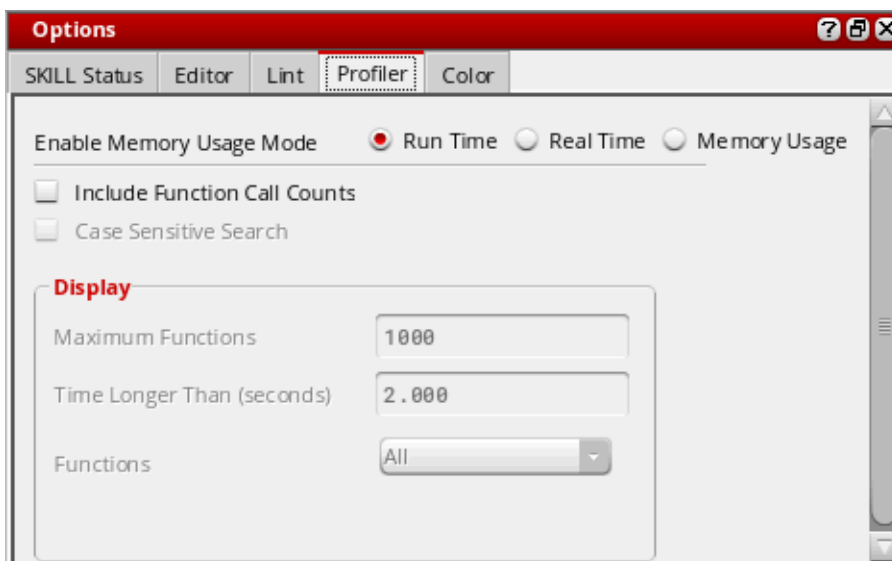
2. In the *Enable Memory Usage Mode* section, select one of the following options to indicate the mode in which the time should be measured.

❑ *Run Time*: To run the profiler in time mode with the time being measured as CPU time.

❑ *Real Time*: To run the profiler in time mode with the time being measured as clock time.

❑ *Memory Usage*: To run the profiler in memory mode.

3. Select the *Include Function Call Counts* check box to view the number of times a function is called in your SKILL program.

Profiler Options for Data Viewing

After you have finished running the Profiler on your SKILL code, the fields in the *Display* area become enabled as shown in the following figure. These fields let you set the display properties for viewing the collected data.
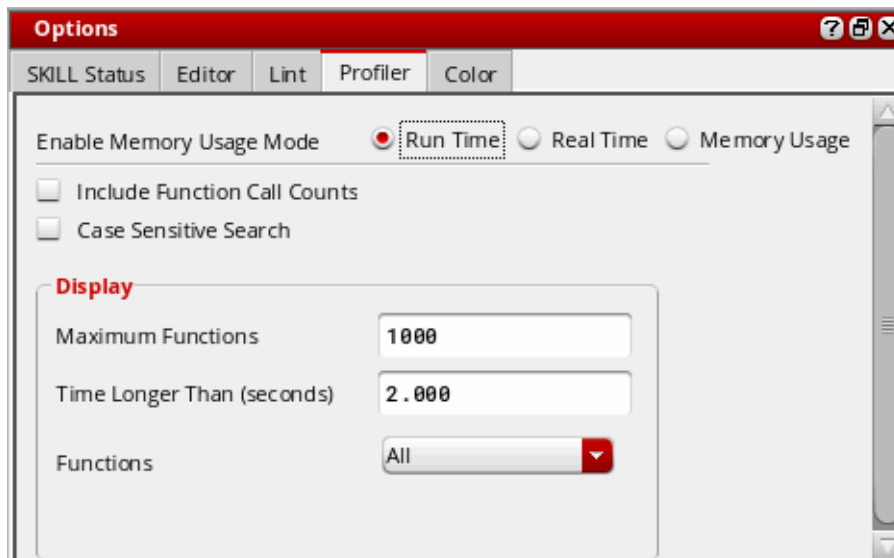
To set the Profiler Options for viewing data, set the following options in the *Display* area:

1. Select the *Case Sensitive Search* check box to match the case of the searched function in the profiler summary report.

2. In the *Maximum Functions* field, type the maximum number of functions you want to see. This field is disabled when you select the *Tree View* tab in the Profiler assistant pane.

3. Depending on the profiler mode currently selected, one of the following fields is displayed:

❑ *Time Longer Than (seconds)*: This field is displayed when you run the profiler in *time mode*. Type the minimum number of seconds that should be spent in a function for its information to appear in the summary.

❏ *Memory Larger Than (bytes)*: This field is displayed when you run the profiler in *memory mode*. Type the minimum number of bytes of memory a function has to have allocated in order for its information to appear in the summary.

These fields are disabled when you select the *Tree View* tab in the Profiler assistant pane.

4. Select one of the following options from the *Functions* drop-down:
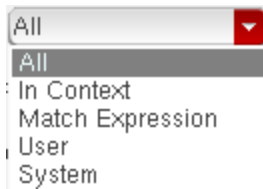
❏ *All* to display all functions in the profiler window.

❏ *In context* to display only the functions in the given context.

❏ *Match Expression* to display only those functions that match the given regular expression.

❏ *Match Prefix* to display only those functions who prefix match the specified.

❏ *User* to display user functions, that is, functions that are neither binary nor write-protected.

❏ *System* to display non-user binary SKILL functions.

If you select the *In context*, *Match Expression*, or *Match Prefix* options, a combo box appears below the *Functions* drop-down. Specify an appropriate context name, regular expression, or prefix name in the box and press Enter.

Note: If the specified context name, regular expression, or prefix name are not found, the profiler clears the summary results.

Related Topics

SKILL Profiler Assistant

Running the Profiler

Controls Available on the Profiler Summary Report

Saving the Profiler Summary

Running the Profiler

The profiler can be run on a SKILL code in two modes – *time mode* and *memory mode*. By default, when you access the profiler for the first time in a Virtuoso session, it is started in *time mode*. To change the mode any time during the run, use the Profiler Options window. For more information, see Profiler Options for Data Collection.

Running the Profiler in Time Mode

To run the profiler in time mode:

1. Choose *Window – Assistants – Profiler*. The Profiler assistant displays.
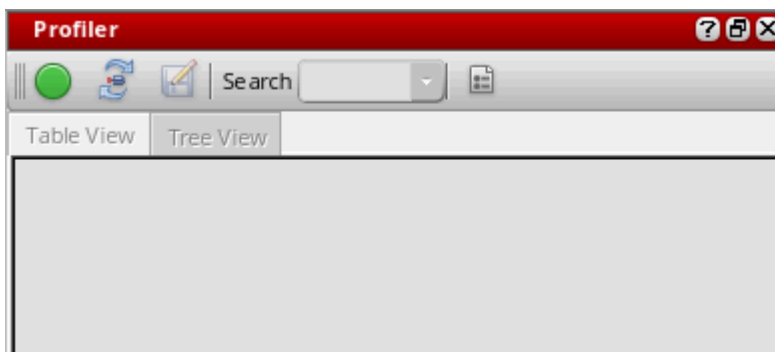
2. Click ⬤ to start profiling.

    When the profiler is running, the *Debug* menu items and the corresponding toolbar get deactivated. The debug capabilities become available for use only after you stop the profiler.
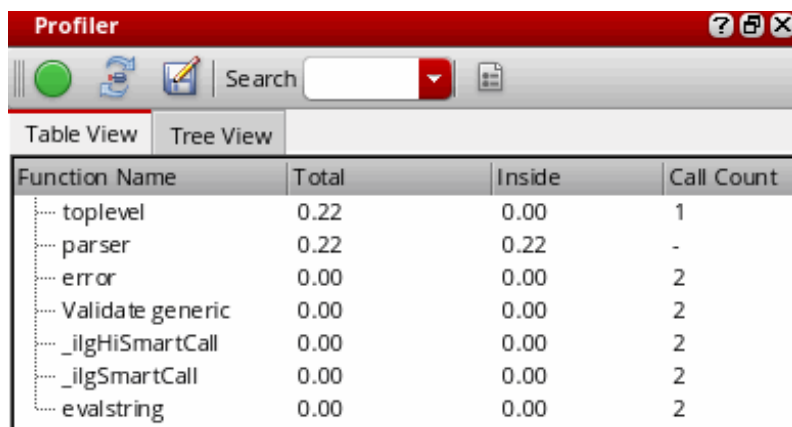
3. Execute the SKILL function you want to profile.

4. Click ⬤ to stop profiling. The profile summary report appears in the Profiler window. The total CPU time (in seconds) taken by the profiler is displayed at the bottom of the summary page. You can check the CPU time taken by a particular function by specifying the function name in the *Search* drop-down list box. To repeat the last search, press Enter.

    You can also specify a prefix name in the *Search* drop-down list box to filter the profile summary report by function prefixes. The search results display both public and private functions that match the searched prefix.

The profile summary report is available in two formats – *Table View* or *Tree View.* Click the related tab to switch the view of the profiler summary report.

Table View of Profiler Summary (Time Mode)

On the *Table View* tab (refer to the image above), the profiler summary report displays the following information:

■ *Function Name*: The name of the calling function.

■ *Total*: The total execution time spent in the function, including the time spent in calls to other functions.

■ *Inside*: The execution time spent within the function.

■ *Call Count*: The number of times the function is called in your SKILL program.

To sort the columns in the profiler summary report, click the column header once.

The profiler summary report may not report the call count for SKILL functions that run quickly (quick functions). To display the call count for all functions, including quick functions, select both *Memory Usage* and *Include function call counts* options in the Profiler *Options* window before running the Profiler.

Tree View of Profiler Summary (Time Mode)

On the *Tree View* tab, the profiler summary report displays the following information in a hierarchical tree format (see the image below):

■ *Function Name*: The name of the calling function.

■ *Total*: The total execution time spent in the function, including the time spent in calls to other functions.

■ *Inside*: The execution time spent within the function.

Profiler summary report reports if a function is called recursively and displays a + sign against such functions. For example, in the image below, the function `fibb1` is called recursively:

Running the Profiler in Memory Mode

To run the profiler in memory mode:

1. Click ⟳ to reset the Profiler.

2. Click ▤ (*Profiler Options*) in the Profiler assistant window. The Profiler Options assistant displays.

3. Select the *Memory Usage* check box to switch to profiling in memory mode.
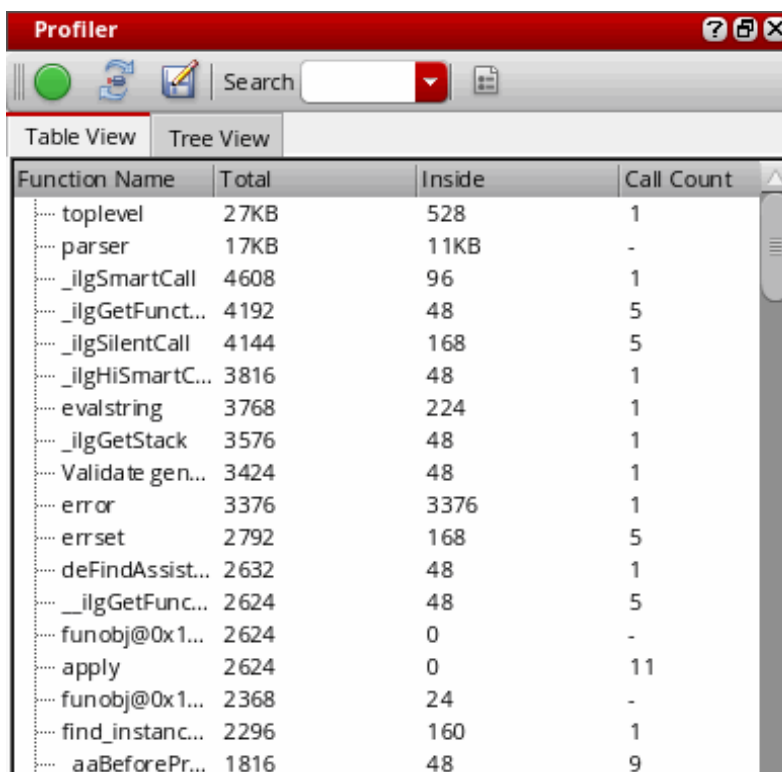
4. Click ⬤ to start profiling.

> **Note:** When the profiler is running, the *Debug* menu items and the corresponding toolbar get deactivated. The debug capabilities become available for use only after you stop the profiler.

5. Execute the SKILL function you want to profile.

6. Click ⬤ to stop profiling. The profile summary report appears in the Profiler window. The total memory allocated (in bytes) by the profiler displays at the bottom of the summary page. You can check the memory allocated to a particular function by specifying the function name in the *Search* drop-down list box.

Click the *Table View* or *Tree View* tab to switch the profile summary view.

Table View of Profiler Summary (Memory Mode)

On the *Table View* tab (refer to the image above), the profiler summary report displays the following information:

■ *Function Name*: The name of the calling function.

■ *Total*: The total bytes of memory consumed by the function, including the memory consumed by its child functions.

■ *Inside*: The memory spent within the function.

■ *Call Count*: The number of times the function is called in your SKILL program.

To sort the columns in the profiler summary report, click the column header once.

Tree View of Profiler Summary (Memory Mode)

On the *Tree View* tab, the profiler summary report displays the following information in a hierarchical tree format (see the image below):

■ *Function Name*: The name of the calling function.

■ *Total*: The total bytes of memory consumed by the function, including the memory consumed by its child functions.

■ *Inside*: The memory spent within the function.

Related Topics

SKILL Profiler Assistant

Setting Profiler Options

Controls Available on the Profiler Summary Report

Saving the Profiler Summary

Controls Available on the Profiler Summary Report

After the profiler displays the summary report, you can right-click the summary results and choose one of the following options:

■ In the *Table View*:

❑ *Go To Source*: To view the definition of the selected function in the source code pane.

❑ *Remove*: To remove the selected function from the results pane.

The *Go To Source* option is disabled for binary functions and function objects.

■ In the *Tree View*:

❑ *Go To Source*: To view the definition of the selected function in the source code pane.

❑ *Expand Deep*: To display all functions called by the selected function recursively until the entire calling tree is expanded.

❑ *Expand Critical*: To expand and highlight the function call sequence that consumes the maximum memory or takes the maximum execution time for the selected function.

❑ *Collapse*: To collapse the function tree.

❑ *Remove*: To remove the selected function from the calling tree.

You can view the definition of a function in the source code pane, only if, the function is not a binary or a read/write protected function. The *Go To Source* option is disabled for function objects as well.

Related Topics

SKILL Profiler Assistant

Setting Profiler Options

Running the Profiler

Saving the Profiler Summary

Saving the Profiler Summary

To save the profiler summary results for later reference, do the following:

1. In the Profiler assistant window, click 📝 . The *Choose a File* dialog box displays.

2. Specify a path and file name for your results and click *Save*.

The profiler summary results can be saved in both table and tree view. You can later open the saved summary report using the *File – Open* menu.

Related Topics

SKILL Profiler Assistant

Setting Profiler Options

Running the Profiler

Controls Available on the Profiler Summary Report

Using the Step Result Assistant

Use the Step Result assistant to examine your code as you step through the program statements. Every time you click *Step* ↱ to step through your code, an expression is executed, and the *Step Result* assistant is updated with the evaluated expression and its value.

To use the Step Result assistant:

1. Choose *Window – Assistants – Step Result*. The *Step Result* assistant displays.

2. Load your SKILL file and set breakpoints in your code.

3. Execute a function in your code to trigger the breakpoint.

4. Click *Step* ↱ to step through your code one statement at a time. The *Step Result* assistant is updated with the currently evaluated expression and its value.
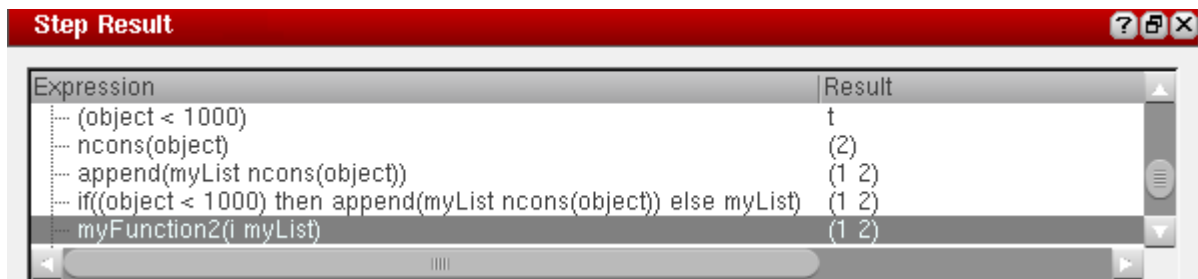
Related Topics

Managing Breakpoints

Executing a Function

5

Workspaces in SKILL IDE

SKILL IDE lets you configure user interface components to suit your individual work preferences. This customized configuration of toolbars and assistants is called a *workspace*.

Each workspace in SKILL IDE is designed to help you perform a set of related tasks, such as *checking*, *coding*, and *debugging*. The workspace can be tailored to a particular application, cellview, or work objective.

You can choose to either use the available workspaces or create your own workspace while working in the SKILL IDE window.

You can even customize SKILL IDE workspaces on requirement basis. You can use a workspace to change the way you view your data by choosing what assistant panes and toolbars appear in your window, where they appear, and whether each assistant pane is docked or floating. You cannot use a workspace to manipulate (filter or reduce) the underlying data itself.

Related Topics

Getting Started with Workspaces

Working with Workspaces

Types of Workspaces

Selecting a Workspace

Saving a Workspace

Loading a Workspace

Deleting a Workspace

Setting the Default Workspace

Types of Workspaces

SKILL IDE provides the following types of workspaces with default docked assistants, which you can modify on requirement basis:

■ **SkillIDE:** This workspace opens by default when a SKILL IDE session starts (see Setting the Default Workspace for more information). When you exit the SKILL IDE session, the current configuration is saved into the default workspace. For more information, see SKILL IDE Work Environment Configuration.

By default, the *SkillIDE* workspace does not display any docked assistants. However, you can customize the workspace to suit your requirements.

■ **Checking:** This workspace displays the following docked assistants:

❑ Lint Manager

Uses the Lint checker to examine the SKILL code for possible errors that go undetected during normal testing. For details, see Efficiency Improvement of SKILL Code.

❑ Profiler (time mode)

Checks the time and memory consumption of the displayed SKILL program. For details, see SKILL Profiler Assistant.

■ **Coding:** This workspace displays the Finder assistant that helps to view the abstract and syntax statements for the SKILL language elements, such as classes, functions, and methods. For details, see Using the Finder Assistant.

■ **Debugging:** This workspace displays the following docked assistants:

❑ Trace

Inspects the changes in values of variables as you step through the program. For details, see Examining and Modifying Variable Values.

❑ Stack

Examines the flow of execution of the function calls that are currently active in the program and are being debugged. For details, see Examining the Call Stack.

❑ Step Result

Examines the code as you step through the program statements. For details, see Using the Step Result Assistant.

■ **SKILL++:** This workspace displays the following docked assistants:

❑ Method Browser

Helps to browse the method tree of generic functions. For details, see <u>Using the Method Browser Assistant</u>.

❑ Class Browser

Helps to understand and follow the class inheritance hierarchy of the classes used in your SKILL code. For details, see <u>Class Inheritance Relationships</u>

Related Topics

<u>Workspaces in SKILL IDE</u>

<u>Selecting a Workspace</u>

<u>Saving a Workspace</u>

<u>Loading a Workspace</u>

<u>Deleting a Workspace</u>

<u>Setting the Default Workspace</u>

Selecting a Workspace

To select a workspace using the menu, do the following:

1. Choose *Window – Workspaces*.

A submenu of workspaces appears listing only those configurations available for use with the current cellview/application.
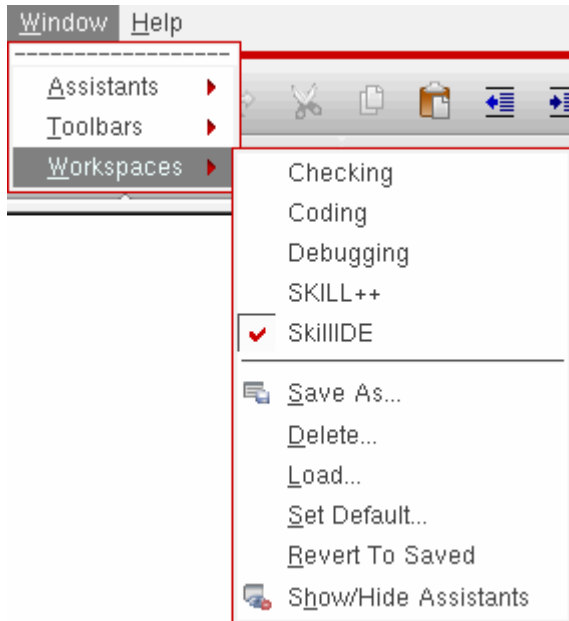
2. Select the workspace you want to apply to the current session window.

   The program applies the workspace you selected to the current session window.

Alternatively, you can select a workspace from the drop-down combo box on the Workspace toolbar.

The assistant panes that are part of a SKILL IDE workspace are initially docked. You can modify the arrangement of your session window and save it as a custom workspace.

Showing and Hiding Assistants in the Workspace

To show or hide the assistants in the workspace, do one of the following:

■ Choose *Windows – Workspaces – Show/Hide Assistants*

■ Press the F11 key to hide or show the assistants.

■ On the Workspace toolbar, select the option.

Related Topics

Workspaces in SKILL IDE

Types of Workspaces

Saving a Workspace

Loading a Workspace

Deleting a Workspace

Setting the Default Workspace

Saving a Workspace
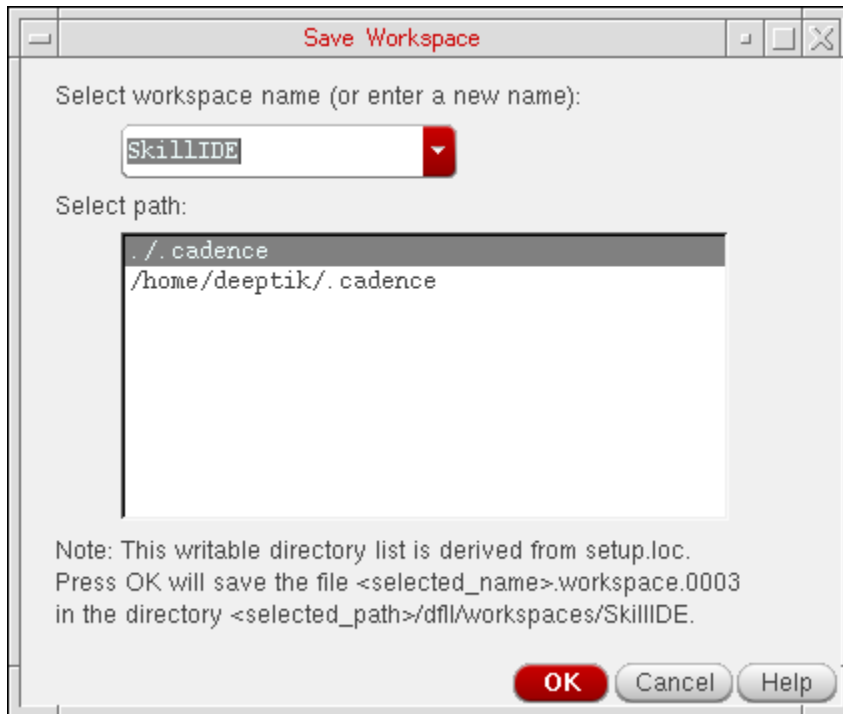
You can customize a workspace by selecting the assistants that you want to display from the *Window – Assistants* menu. You can then save the customized workspace by doing one of the following:

■ Choose *Window – Workspaces – Save As.*

■ On the Workspace toolbar, select the option.

The *Save Workspace* form appears.

In this form, specify the name with which you want to save the workspace and select the path where you want to save the workspace. You can specify a new name or make changes to an existing workspace. Click *OK* to save the changes you made.

If you do not want to save the changes you made to the existing workspace, choose *Windows – Workspaces – Revert to Saved* to revert to the factory settings.

Related Topics

Workspaces in SKILL IDE

Types of Workspaces

Selecting a Workspace

Loading a Workspace

Deleting a Workspace

Setting the Default Workspace

Loading a Workspace

To load a workspace,

1. Choose *Window – Workspaces – Load*.

   The *Load Workspace* form appears.
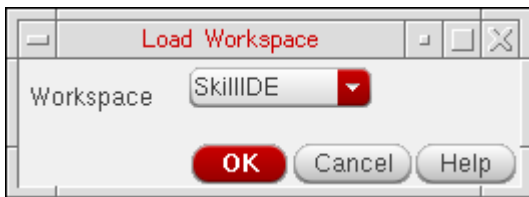
2. From the *Workspace* drop-down combo box, select a workspace.

3. Click *OK*. The program changes the layout of your session window using the workspace you selected.

Alternatively, select the required workspace from the *Workspace* drop-down list box on the Workspace toolbar.

Related Topics

Workspaces in SKILL IDE

Types of Workspaces

Saving a Workspace

Selecting a Workspace

Deleting a Workspace

Setting the Default Workspace

Deleting a Workspace

To delete a workspace,

1. Choose *Window – Workspaces – Delete*.

The *Delete Workspace* form appears.

2. In the *Workspace* drop-down combo box, select the workspace you want to delete.

3. Click *OK*.

The program deletes the workspace. If you delete the current workspace, the program displays the default workspace.

Related Topics

Workspaces in SKILL IDE

Types of Workspaces

Saving a Workspace

Loading a Workspace

Selecting a Workspace

Setting the Default Workspace

Setting the Default Workspace

To set a workspace as the default workspace,

1. Choose *Windows – Workspaces – Set Default.*

The *Set Default Workspace* form is displayed.
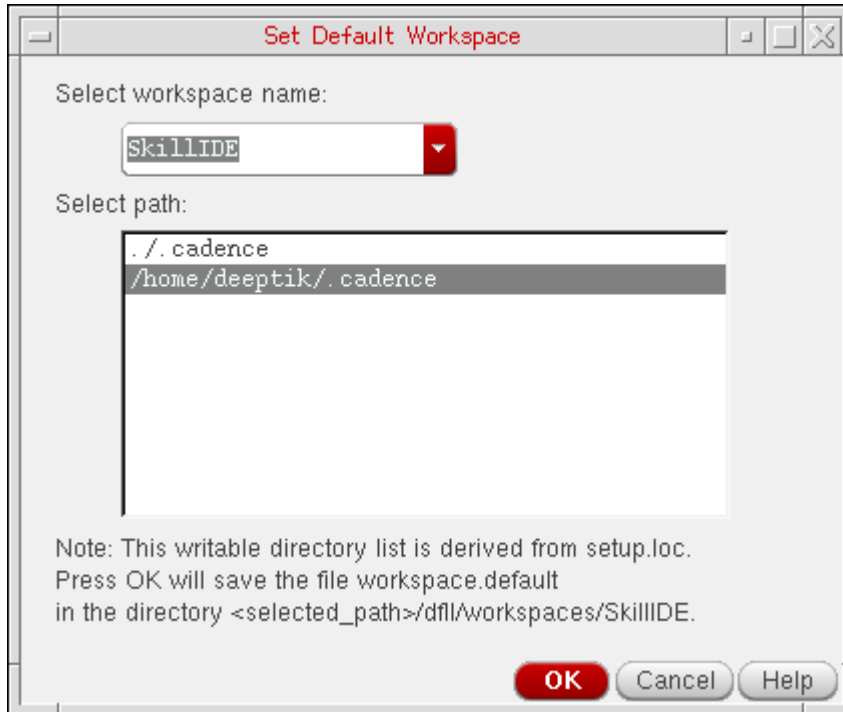
2. From the *Select workspace name* drop-down combo box, select the workspace you want to use as the new default.

   This workspace will appear for each subsequent invocation of the current application or view type.

3. Optionally, select the path where you want to save the default workspace specification to.

   All writable locations in your Cadence Search File (CSF) will be listed.

   Note: Assuming that your home directory has been set up as a member of the CSF, the workspace will be saved to $HOME by default. However, you may want to change this to ./.cadence, or another writable CSF location, so that the default is only applied to the current design.

4. Click *OK* to set the new default workspace for the current application.

Related Topics

Workspaces in SKILL IDE

Types of Workspaces

Saving a Workspace

Loading a Workspace

Deleting a Workspace

Selecting a Workspace

6

SKILL IDE Debugger Example File

The following sample program is designed to illustrate the various features of the SKILL IDE debugger. Using the sample program you learn how to use the SKILL IDE tool to analyze and debug a program to improve its performance.

The following is the sample program file that is used for an example.

<your_install_dir>/tools/dfII/samples/skill/demo.il

The sample program file, `demo.il` has the following contents:

```
/* demo.il - This file is used for a walkthrough of the
 * SKILL Development Environment.
 */
/*********************************************************
* myFunction1 - This function must
*     Count from 1 to 10000.
*     Return a list of numbers from 1 to 1000 in any order.
*********************************************************/
(procedure myFunction1()
    let((x y z myList)
      for( i 1 10000
         myList = myFunction2(i)
      )
      myList
    )
)
/*********************************************************
* myFunction2 - This function must
*     Print a starting message on the 1st object.
*     Print an ending message at the 1000th object.
*     Return a list of numbers less than 1000 in any order.
*********************************************************/
(procedure myFunction2(object myList)
    if(myTest(object)
      then printf("Starting with object %d...\n" object)
    )
    if(object == 1000
      then printf("Ending with object %d...\n" object)
    )
    if(object < 1000
      then append(myList ncons(object ))
      else myList
    )
)
/*********************************************************
```

```
* myTest - This function must
*     return t if object equals one.
******************************************************/
(procedure myTest(object)
    if(object == 10
      then t
      else nil
    )
)
```

Related Topics

Copying the Example File

Loading and Running the Example File

Tracing the Error

Correcting the Error

Using Breakpoints to Find and Correct a Functional Error

Copying the Example File

The following demo file is used in this example:

`<your_install_dir>/tools/dfII/samples/skill/demo.il`

To copy this demo file for the walkthrough, do the following:

1. Make a temporary directory in your current working directory:

   `mkdir tmp`

2. Change directory to your new temporary directory:

   `cd tmp`

3. Copy the demo file from the Cadence installation hierarchy:

   `cp <your_install_dir>/tools/dfII/samples/skill/demo.il .`

4. In your CIW, choose *Options – Log Filter*.

The Set Log File Display Filter form appears.
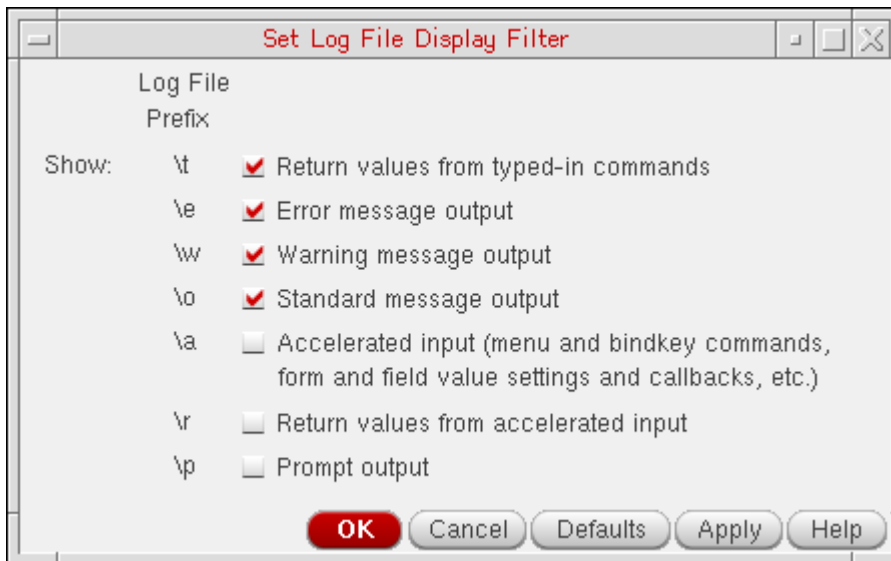
5. Verify the settings as shown above.

Related Topics

SKILL IDE Debugger Example File

Loading and Running the Example File

Tracing the Error

Correcting the Error

Using Breakpoints to Find and Correct a Functional Error

Loading and Running the Example File

To load and run the example file:

1. Click or choose *File – Open and Load* from the SKILL IDE menu bar. The *Open File* dialog box displays, listing all the available files.

Browse to locate the *tmp* directory and click *Open*. The Top-Level number, #1, displays in the status bar.

2. Type the following in the CIW:

```
myFunction1()
```

An error message displays in the message area of the CIW.

You can also run `myFunction1()` from the *Debug* menu or Edit toolbar. For more information, see Executing a Function.

Related Topics

SKILL IDE Debugger Example File

Copying the Example File

Tracing the Error

Correcting the Error

Using Breakpoints to Find and Correct a Functional Error

Tracing the Error

The error message that displays in CIW indicates that `myFunction2` expects two arguments and only one argument is passed:

```
***Error in routine myFunction2:
Message: *Error* myFunction2: too few arguments (2 expected, 1 given) - (1)
```
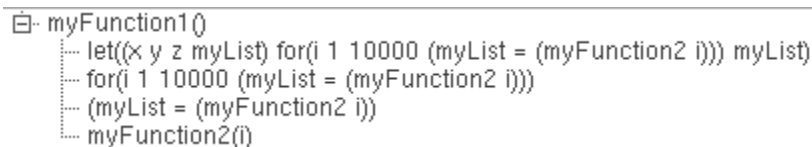
The debugger stops at line number 12 and highlights it in red:

```
myList = myFunction2(i)
```

Examining the Call Stack to Trace the Error

To find which function called `myFunction2` with the wrong number of arguments, do the following:

1. Choose *Window – Assistants – Stack*. The *Stack* assistant displays the current call stack.

The call to `errorHandler` is at the bottom of the call stack frame; the call to `myFunction1` is at the top.

2. Click ![icon] or choose *Debug – Stop All Debugging* to terminate debugging.

Related Topics

SKILL IDE Debugger Example File

Copying the Example File

Loading and Running the Example File

Correcting the Error

Using Breakpoints to Find and Correct a Functional Error

Correcting the Error

To correct the call to `myFunction2`, do the following:

1. Review the definition of `myFunction2`. Notice that it requires a second argument, `myList`:

```
(procedure myFunction2(object myList)
```

2. Change the `myFunction2` function call in the `for` loop in `myFunction1` accordingly:

Change	`myList = myFunction2(i)`
to	`myList = myFunction2(i myList)`

3. Click ![icon] to load the edited file.

The following messages display in the CIW:

```
function myFunction1 redefined
function myFunction2 redefined
function myTest redefined
```

Note: Every time you load your file, the functions are redefined with their new definitions.

Related Topics

SKILL IDE Debugger Example File

Using Breakpoints to Find and Correct a Functional Error

To find the next error, execute `myFunction1` that you modified to correct the error.

➤ In the CIW, type

```
myFunction1
```

The following output displays in the CIW:

```
Starting with object 10...
Ending with object 1000...
(1 2 3 4 5
    6 7 8 9 10
    ...
    996 997 998 999
)
```

The starting object is 10 instead of 1. This is a functional error.

Using Breakpoints to Identify the Source of the Error

Let us use breakpoints to identify the source of the error.

1. In the source code pane, click the following line:

   ```
   myList = myFunction2(i)
   ```

   Choose *Debug – Set/Modify Conditional Breakpoint*. The *Set Conditional Breakpoint (Line)* dialog box displays.

2. In the *Break on condition* field, type `i==10`.

   As a result, the program will pause when the value of `i` equals to 10.

3. Click *OK*.

Executing a Function Through Single-stepping

To execute `myFunction1` and single-step from the conditional breakpoint you have set (`i==10`), do the following:

1. In the CIW, type

```
myFunction1()
```

When `i` equals 10, a breakpoint message displays in the CIW.

```
myFunction1()

Stopped at file : /home/deeptik/tmp/demo.il line : 12
<<< Break >>> on explicit 'break' request
Entering new debug toplevel due to breakpoint:
```

2. In SKILL IDE, click [icon] or choose *Debug – Step* five times.

The following messages display in the CIW:

```
stopped before evaluating myFunction2(i myList)
<<< Entering myFunction2 >>>
stopped before evaluating if(myTest(object) then printf("Starting with object %d...\n" object)
stopped before evaluating myTest(object)
<<< Entering myTest >>>
stopped before evaluating if((object == 10) then t else nil)
stopped before evaluating (object == 10)
<<< Exiting myTest >>>
stopped before evaluating printf("Starting with object %d...\n" object)
```

Single-stepping through the program shows that `myFunction2` calls `myTest` with a value of 10. `myTest` returns `t` if the value passed to it is equal to 10. However, `myTest` should return `t` if the value is equal to 1.

3. In SKILL IDE, click [icon] or choose *Debug – Stop All Debugging*.

Correcting the Erroneous File

Make the following correction in `myTest`:

1. In the source code pane, change the erroneous line in `myTest` as follows:

Change   `if(object == 10`

to   `if(object == 1`

2. Click 👆 to load the edited file.

The following messages display in the CIW:

```
function myFunction1 redefined
function myFunction2 redefined
function myTest redefined
```

Verifying the Fixed Problem

To verify the problem has been fixed, do the following:

➤ In the CIW, type

```
myFunction1()
```

The following output displays in the CIW:

```
Starting with object 1...
Ending with object 1000...
(1 2 3 4 5
    6 7 8 9 10
    ...
    996 997 998 999
)
```

The starting object is 1. You now have a working program.

Related Topics

SKILL IDE Debugger Example File

Copying the Example File

Loading and Running the Example File

Tracing the Error

Correcting the Error

A

Command Line Interface for SKILL Profiler

The Virtuoso executable containing the Cadence® SKILL language accepts command line options to turn on SKILL Profiling. The SKILL Profiler can be turned on when you start an executable by passing it the following arguments.

```
virtuoso -ilProf {time | memory | realTime} [-ilProfFile filename]
```

Because you enter these commands at the shell level, you use the shell syntax, hence the dash options. For example:

```
virtuoso -ilProf memory -ilProfFile /tmp/profMem.out
```

ilProf

Turns on SKILL Profiling for `time` by default. If `memory` is specified as the argument, the profiler runs in memory mode. If `realTime` is specified as the argument, the the profiler runs in time mode, with the time being measured as clock time.

ilProfFile

Specifies the destination file for the SKILL Profiling results. The file name should follow this argument. The default is `ilProf.out` in the directory from which the executable was started.

When the executable is exited, the profile summary file is written out.

Related Topics

Command Line: Test Coverage

TCov Report Files

Command Line: Test Coverage

SKILL Test Coverage lets you determine which code was executed during a session. This information lets you increase the coverage of your test cases and thereby improve the quality of your SKILL code.

Test coverage only measures statements inside a function. Top-level statements or statements that are not within the function body are not covered.

When you start up SKILL test coverage, you must pass the executable command line arguments telling SKILL which files to measure. When those files are loaded, they automatically compile the functions to include `tCov` instructions. When the SKILL session ends, report files are written out.

➤ Run SKILL Test Coverage using the following command line arguments.

```
virtuoso {-ilTCov fileNames | -ilTCovList fileName} [-ilTCovDir directory] [-
ilTCovReportsOnly]
```

Because you enter these commands at the shell level, you use the shell syntax, hence the dash options and the single quotes around the file names, to submit them as one argument to the option.

For example:

```
virtuoso -ilTCov 'block.il tools.il' -ilTCovDir /tmp/test
```

ilTCov

`ilTCov` followed by a list of files is the only argument required to run SKILL Test Coverage. You can also pass the list of SKILL files in the current directory.

ilTCovDir

Takes the directory into which all report files are written as its argument. If this argument is not given, the report files are written to the same directory from which the SKILL files being measured were loaded.

ilTCovList

A text file containing the name of the source files to be instructed for code coverage. For example:

```
virtuoso -ilTCovList files.dat -ilTCovDir ./tcov
```

In the above example, `files.dat` is a text file with the SKILL source file names to be instructed. Each file name is in its own line without any path information.

ilTCovReportsOnly

Allows you to print only the summary report files and not the files that show the source code annotated with test coverage information. This option greatly reduces exit time during test coverage.

ilTCovUseFullFileName

The current `tcov` report file names are generated from the given source file names by replacing their file extension with `tcov` extension. For example `file1.api` would have its `tcov` report file as `file1.tcov`.

This becomes a problem if there are multiple source files with different extensions sharing the same base name. For example, `file1.api` and `file1.cb`. The coverage report for these files cannot be captured correctly because their `tcov` report filenames are the same.

Using `ilTCovUseFullFileName` instructs SKILL `tcov` to preserve the original source file extension to avoid the report file name collision issue. For the same example, the correct `tcov` reports are in `file1.api.tcov` and `file1.cb.tcov`.

Related Topics

Command Line Interface for SKILL Profiler

TCov Report Files

TCov Report Files

When the SKILL session is over and SKILL exits, three report files for SKILL Test Coverage listed below are written out.

ilTCovSummary

A summary report that shows the percentage of expressions and functions that were executed. After the results, untested functions or functions that were not called are listed. This

file is written either to the directory from which the SKILL executable was started or to a directory specified with `-ilTCovDir`.

<fileName>.tcov

A `tcov` file for each source file or source file in a context showing each function definition and the expressions that were executed. This file is placed in the directory containing the source code or in a directory specified with `-ilTCovDir`.

Note: `tCov` mode for context files is supported only for Cadence-generated context files, which are located in the `dfII` installation hierarchy. Each context source has a file named `startup.il`, which needs to be loaded in `-ilTCov` mode.

<fileName>.d

A temporary file used to collect data across multiple runs. This file is placed in the directory containing the source code or in a directory specified with `-ilTCovDir`.

Related Topics

Command Line Interface for SKILL Profiler

Command Line: Test Coverage

B

SKILL Lint

Cadence® SKILL Lint examines SKILL code for possible errors (that went undetected during normal testing) and inefficiencies. SKILL Lint also reports global variables that are not declared locally. SKILL Lint checks a SKILL file or context and reports potential errors and ways to clean up your code. In particular, SKILL Lint is useful for helping programmers find unused local variables, global variables that should be locals, functions that have been passed the wrong number of arguments, and hints about how to improve the efficiency of the user's SKILL code.

SKILL Lint is usually run over a file. If a context is specified and the file is `startup.il` or is not specified, all the files ending with *`.il` or *`.ils` in the directory `your_install_dir/pvt/etc/context/t_contextName` are checked. By default, the SKILL Lint output prints to the Command Interpreter Window but can be printed to an output log file as well or instead. SKILL Lint prints messages about the user's code starting with the file and function name to which the message pertains.

Related Topics

SKILL Lint Rules

SKILL Lint Features

SKILL Lint PASS/FAIL and IQ Algorithms

Built-In Messages

sklint

SKILL Lint Features

Listed below are some of the SKILL Lint features.

Check the Number of Function Arguments

SKILL Lint checks that the number of arguments passed to a function is correct by verifying it against a previously-known definition of the function from a previous SKILL Lint session or a previous declaration. SKILL Lint delays checking the number of arguments until it finds the procedure definition.

If a procedure is used in a file before it is defined in the same file and the number of arguments to the procedure changes, it may be necessary to run SKILL Lint twice to get accurate results because the first run will use the previous declaration of the procedure.

Check Function and Global Variable Prefixes

Functions and global variables used in SKILL code are expected to be prefixed with a suitable string. You type these strings in the *Package Prefixes* field on the <u>Lint Options form</u>.

Note: By default, strict checking is applied only to the global variables, while functions and Cadence's prefixes are checked by specification.

The naming policy for functions and global variables is as follows:

■ Cadence-official SKILL functions and global variables must start with a lower-case character, while three characters and all lower-case are preferred. Cadence-official SKILL functions include those functions that are documented and supported.

■ Customer SKILL functions and global variables, and any functions that are not documented or supported, must start with an upper-case character.

■ Functions or global variables must start with the required prefix, or the prefix plus an optional lower-case character followed immediately by an upper-case character or an underscore, _. The optional lower-case character must be one of the following: b, c, e, f, i, m, or v. So, the two syntax forms for the functions or variables will be as follows:

   ❑ prefix + ([A-Z] | '_') + rest

   ❑ prefix + ('b' | 'c' | 'e' | 'f' | 'i' | 'm' | 'v') + ([A-Z], '_') + rest

   Examples:

   EV1var is flagged because the character after EV1v 'a' is not ([A-Z] | '_').

EV1vAr is not flagged because the letter after EV1v 'A' is ([A-Z] | '_').

EV1v_r is not flagged because the letter after EV1v '_' is in ([A-Z] | '_').

EV1d_r is flagged because the letter after EV1 'd' is not in the accepted set.

EV1_ar is not flagged because the letter after EV1 is '_'.

You can turn off strict checking by disabling the STRICT <u>message</u> such that SKILL Lint checks only global variables beginning with a specified prefix.

Check SKILL Operators

SKILL Lint reports the SKILL operators used in the SKILL code. The following operators are supported in SKILL:

■   &&, ||, !, <<, and >> (logical operators)

■   ~, &, ~&, ^, ~^, |, and ~| (bitwise logical operators)

■   <, <=, >, >=, ==, != (relational operators)

■   **, *, /, +, -, ++s, s++, --s, s--, - (arithmetic operators)

■   = (assignment operator)

■   : (range operator)

Check Redefinition of Write-protected Functions and Macros

SKILL Lint reports an error if the input name of an expression, defun, defmacro, mprocedure, nprocedure, or procedure is the name of a write-protected function or macro. For example, SKILL Lint will report an error for the following code because lindex is a write-protected function:

```
procedure( lindex(@optional (input1 "hello") (input2 "hi") )

    printf("\n%s %s" input1 input2)
)
```

Check Files Based on the Extension of the Input File (*.il/*.ils/*.scm)

SKILL Lint performs SKILL or SKILL++ language checks based on the extension, .il, .ils, or .scm of the input file for sklint. If the extension of the input file is .il, SKILL Lint applies SKILL language checks and if the extension of the input file is .ils, SKILL Lint applies

SKILL++ language checks. It also applies SKILL++ language checks for the files with the extension `.scm`.

For example, SKILL Lint will report errors if the Scheme code displayed below is included in a file with extension `.il`.

```
defun( f1 (a b)
    let( (f (lambda((x) x*x)) )
        (f a+b)
)
```

Execute Code Blocks Placed Inside `inSkill()` and `inScheme()`

Before describing the following feature, it is important to reiterate that in the SKILL programming environment, Scheme mode describes SKILL++ code that is saved in a file with the `.ils` extension or enclosed in the `inScheme()` function. Similarly, SKILL mode describes SKILL code saved in a file with `.il` extension or enclosed in the `inSkill()` function.

Starting from IC6.1.6, SKILL Lint executes source code placed in the `inScheme()` function as SKILL++ language code even if the code is placed in a file with extension `*.il`. For example, SKILL Lint executes the following Scheme code correctly even though it is placed inside a file with extension `.il`

```
(let (v1 v2)
    v1 = v2 = 0
    (inScheme vf = (lambda () 'some_value))
) ; "vf" global Scheme function is defined inside inScheme() block
```

SKILL Lint understands the code (in **Bold**) as SKILL++ or Scheme code. In this example, `vf` is considered a Scheme function and not as a symbol.

Consider the following example where the code is placed inside a file with extension `.ils`

```
(let (v1 v2)
    v1 = v2 = 0
    (inSkill vf = (lambda () 'some_value))
) ; "vf" global Scheme function is defined inside inScheme() block
```

SKILL Lint understands the code (in **Bold**) as SKILL code. In this example, SKILL Lint will report errors because the expression related to `vf` function is in basic SKILL, which does not support local functions.

Check For Matching Global and Local Variable Names

SKILL Lint allows the name of a local variable to match that of a global variable and does not report the variable as being used before definition in a `let` assignment. In the example below, the name `VAR` has been used for both local and global variables.

```
(defvar VAR)
(defun ListGen (n)
(let ( (VAR (cons n VAR)) )
(info "The value of local variable VAR: %L\n" VAR)
(setq VAR (if (plusp n) (ListGen (sub1 n)) (cdr VAR)))
)
)
(info "Global variable VAR before function call: %L\n" VAR)
(info "Function call result: %L\n" (ListGen 3))
(info "Global variable VAR after function call: %L\n" VAR)
```

After the lint run, the following output is printed in the CIW:

```
Global variable VAR before function call: nil
The value of local variable VAR: (3)
The value of local variable VAR: (2 3)
The value of local variable VAR: (1 2 3)
The value of local variable VAR: (0 1 2 3)
Function call result: (1 2 3)
Global variable VAR after function call: nil
```

SKILL Lint understands that the variable VAR has been used as both global and local variable and does not report a warning on line 3 for it being used before available in a `let` assignment.

Support Local Functions in the Scheme Mode

In the Scheme mode (files with extension .ils), SKILL Lint recognizes local functions to be defined inside `let`, `letseq`, `letrec`, `setq`, `flet`, and `labels` blocks. This means that SKILL Lint allows local functions to be added as a list of valid variables and does not report them as unused variables even if they are not referenced or used. In the example shown below, `f1` is a local function that is considered as a variable in the `let` block.

```
let( (
      ((f1 lambda(() println("f1()")))
      f1()

    )
```

SKILL Lint applies special checks for Scheme code (files with extension .ils/.scm), if a function is defined as global function/procedure or if a local function is not defined inside the `let`, `letseq`, or `letrec` block.

Support Local Arguments in the Scheme Mode

In the Scheme mode, SKILL Lint recognizes local arguments defined via @key, @optional, and @aux options. For example, the following constructs are valid in the Scheme mode:

```
procedure( test1( arg11 @key (arg12 arg11) )
      (progn
```

```
        (println "In function test1")
            (assert arg11 == arg12)
    )
)


procedure( test2( arg21 @optional (arg22 arg21) )
    (progn
        (println "In function test2")
        (assert arg21 == arg22)
    )
)


(defun test11 ( arg111 @key (arg112 arg111) )
    (progn
        (println "In function test11")
        (assert arg111 == arg112)
    )
)


(defun test21 ( arg211 @optional (arg212 arg211) )
    (progn
        (println "In function test21")
        (assert arg211 == arg212)
    )
)


(inScheme
(defun smartlessp2 (arg1 arg2 @aux fn)
(cond
((and (numberp arg1) (numberp arg2)) (setq fn lessp) )
((and (symstrp arg1) (symstrp arg2)) (setq fn alphalessp) )
)
(if fn (fn arg1 arg2) -1)
)
(defvar A (list 3.0 1.0 "as" "df" 7))
(defvar B (list 1.0 5.0 "asd" "ca" "gh"))
(info "Test AUX:\n")
(foreach (itA itB) A B (info "%L < %L: %L\n" itA itB (smartlessp2 itA itB)))
)
```

Support Assignment of Functions as Variables in Scheme Mode

In the Scheme mode, SKILL Lint recognizes the use of local functions as variables that are defined inside the `setq`, `getq`, `getd`, `putd`, and `defun` and `procedure` blocks. See the following examples.

Example 1

This example shows the assignment of a variable to a function in a `setq` block. SKILL Lint considers the function `smartless()` as a local function in the Scheme mode.

```
defun( smartless (arg1 arg2)
    let( ( (fn nil) (ret nil) )
        if( (and (numberp arg1) (numberp arg2))
            then
                setq( fn lessp)
            else
                (when (and (symstrp arg1) (symstrp arg2))
                    setq( fn alphalessp))
            )
            (when fn (setq ret (fn arg1 arg2)))
            ret
    )
)
```

Example 2

In this example, SKILL Lint considers the code `((getq obj1 funcObj) val)` as a function call.

```
inScheme(
    defun( funcInst (x) x*x)
    defstruct( myStruct funcObj)
    defclass( testClass () ((intVar @initarg intVar)))
    let( (obj classObj obj1 (val 3))
        setq( obj (make_myStruct ?funcObj funcInst))
        setq( classObj (makeInstance 'testClass ?intVar obj))
        setq( obj1 (getq classObj intVar))
        info( "%L\n" ((getq obj1 funcObj) val))
    )
)
```

Example 3

In the following example, the function myFunc1() is defined as a local function that has one argument. SKILL Lint considers `myFunc1` as a function and checks for the number of input arguments.

```
let( ()
    putd( 'myFunc1 (getd 'exp))
)
info( "Result: %L\n" (myFunc1 1 2))
```

Example 4

In this example, SKILL Lint considers `exp()` as a local function.

```
putd( 'myFunc2 exp)
```

Example 5

In the following example, `smartlessp()` is defined by using the `defun()` function and then assigned to a variable `newFn`. In the Scheme mode, SKILL Lint considers the variable, `newFn`, as a SKILL++ function. This new SKILL++ function, `newFn`, is further assigned to another variable `func1`. Therefore, SKILL Lint considers both `newFn` and `func1` as SKILL++ functions.

```
defun( smartlessp (arg1 arg2)
    let( ( fn )
        cond(
            ((and (numberp arg1) (numberp arg2)) (setq fn lessp) )
            ((and (symstrp arg1) (symstrp arg2)) (setq fn alphalessp) )
        )
        (if fn (fn arg1 arg2) -1)
    )
)
newFn = smartlessp
func1 = newFn
info( "A: %L\n" (newFn (list 1) (list 2)))
info( "B: %L\n" (newFn 1 2))
info( "C: %L\n" (newFn "str2" "str3"))
```

Support of SKILL++ Features in SKILL Lint

SKILL Lint supports the following SKILL++ features:

■ Recognizes multi-methods or methods specialized on more than one argument, such as the `@before`, `@after`, and `@around` methods.

- Considers an argument of the `defmethod` beginning with an underscore `` `_? `` as a valid character and generates only an `info` message. In the earlier releases, SKILL Lint treated such arguments as invalid and generated warnings.

- Displays warning messages if functions are incorrectly referenced as SKILL public functions and not as SKILL++ functions.

- Recognizes multiple inheritance of classes created using `defclass`.

Related Topics

SKILL Lint

Built-In Messages

Message Groups

SKILL Lint PASS/FAIL and IQ Algorithms

Message Groups

The message group priority appears as the first token on an output report line. Message group priorities and their associated message group names (which are listed as a hierarchical tree in the *Rules* section when you select to customize message rules) are as follows:

Message Group Priority	Message Group Name
ALERT	`alert` is the group of messages that are notifications.
ERROR	`error` is the group of messages that are considered errors.
ERROR GLOBAL	`error global` is the list of variables used as both globals and locals.
EXTERNAL GLOBAL	`external global` is the list of variables defined externally as globals.
FATAL ERROR	`fatal error` is the group of messages that prevent SKILL Lint from proceeding with analysis.
HINT	`hint` is the group of messages that tell you how to make your code more efficient.
INFORMATION	`information` is all general information messages.

Message Group Priority	Message Group Name
INTERNAL	`internal` is the group of messages about failures of the reporting mechanism.
INTERNAL ALERT	`internal alert` is a group of messages to flag SKILL code that will not work in the next release.
PACKAGE GLOBAL	`package global` is the list of global variables that begin with the package prefix.
SUGGESTION	`suggestion` is the group of messages that indicate possible ways you can increase the performance of your code.
UNUSED VAR**S**	`unused vars` is the list of local variables that do not appear to be referenced.
WARNING	`warning` is the group of messages that are potential errors.
WARNING GLOBAL	`warning global` is the list of global variables that do not begin with a package prefix.

Related Topics

SKILL Lint

SKILL Lint Features

Built-In Messages

SKILL Lint PASS/FAIL and IQ Algorithms

Built-In Messages

The built-in message name appears in parentheses and capitalized on the output report line. You can customize which messages SKILL Lint reports using the hierarchical tree displayed in the *Rules* section when you select *Custom* in the Lint Options form. SKILL core messages and their associated message groups are as follows:

Built-in Message Name	Message Group	Message Description
ALIAS1	`error`	Both arguments to `alias` must be symbols.
APPEND1	`suggestion`	Consider using `cons` rather than `append`.

Built-in Message Name	Message Group	Message Description
ARRAYREF1	error	First argument to `arrayref` must evaluate to an array.
ASSOC1	suggestion	Consider using `assq` rather than `assoc`.
BACKQUOTE1	suggestion	Possibly replace this backquote with a quote.
CASE1	warning	`case` can never be reached (after default `t`).
CASE2	warning	Symbol `t` used in `case` or `caseq` list.
CASE3	error	Duplicate value in `case` or `caseq`.
CASE5	hint	`case` can be replaced with `caseq`.
CASE6	warning	Quoted value in `case` or `caseq` (quote not required).
CASEQ1	error	You must use `case` rather than `caseq`.
CHK1	error	Type template string must be last argument.
CHK2	error	Redundant statement.
CHK3	error	Bad argument (must be a symbol).
CHK4	error	Redundant argument template.
CHK6	error	Macros cannot have `@key`, `@rest`, or `@optional`.
CHK7	error	Nlambda 1st argument must be a symbol.
CHK8	error	Entry after `@rest` not allowed.
CHK9	error	`@rest,` or `@key,` or `@optional` not followed by an argument.
CHK10	error	Argument duplicated.
CHK11	error	Nlambda 2nd argument should be a list.
CHK12	error	Nlambda maximum of two arguments.
CHK13	error	`@optional` and `@key` cannot appear in the same argument list.
CHK14	error	Bad argument, should be a list of length 2.
CHK15	error	Bad argument, should be a list.

Built-in Message Name	Message Group	Message Description
CHKARGS1	error	Function requires at least n arguments. See <u>Check the Number of Function Arguments</u>.
CHKARGS2	error	Function takes at most n arguments. See <u>Check the Number of Function Arguments</u>.
CHKARGS3	error	Key argument repeated.
CHKARGS4	error	Unknown key argument.
CHKARGS5	error	No argument following key.
CHKFORM1	error	Number of arguments mismatch.
CHKFORM2	error	Bad statement.
DBGET1	error	Second argument to ~> must be symbol or string.
DEADCODE1	warning	Unreachable code.
DECLARE1	error	Arguments to `declare` must be calls to `arrayref`, (e.g. `a[10]`).
DECODE1	error	You must use `case` or `caseq` rather than `decode`.
DEF1	error	Extra argument passed to `def`.
DEF2	error	Last argument to `def` is bad.
DEF3	hint	`nlambda`, `macro`, or `alias` should not be referenced before it is defined.
DEF4	hint	`nlambda`, `macro`, or `alias` might be referenced before it is defined.
DEF5	hint	Recursive call to an `nlambda` function or macro is inefficient, call `declareNLambda` first.
DEF6	error	Definition for function `def` cannot have more than 255 required or optional arguments.
DEFSTRUCT1	error	Arguments to `defstruct` must all be symbols.
EQUAL1	hint	You can replace `== nil` with `!`.
EQUAL2	hint	You can replace `== 1` with `onep`.
EQUAL3	hint	You can replace `== 0` with `zerop`.

Built-in Message Name	Message Group	Message Description
EVALSTRING1	suggestion	Consider using `stringToFunction` when `evalstring` is called multiple times with the same string.
ExtHead	information	Known/Unknown External functions called.
ExtKnown	information	Functions called that are defined outside of analyzed code.
External	information	Functions called that are not defined.
External0	information	Line numbers in which undefined functions were called.
FOR1	error	First argument to `for` must be a symbol.
FnsLocal	information	Name of a SKILL++ local function
FnsLocal0	information	Location of a SKILL++ local function
Flow	information	Reports the call flow for the code analyzed.
GET1	error	Second argument to -> must be either a symbol or a string.
GET2	error	Autoload symbol is no longer used, replace `get` with `isCallable`.
GETD1	error	`getd` no longer returns a list, use the function `isCallable`.
GO1	error	`go` must have only one argument, a symbol.
GO2	error	`go` must be called from within a `prog` containing a label.
IF4	error	`then` and `else` required in `if` construct.
IF5	error	`else` without corresponding `then`.
IF6	hint	Remove the `then nil` part and convert to `unless`.
IF7	hint	Remove the `else nil` part, and part convert to a `when`.
IF10	hint	Invert the test and replace with `unless`, as no `else` part.

Built-in Message Name	Message Group	Message Description
IQ	information	IQ score (best is 100).
IQ1	information	IQ score is based on messages * priority.
LABEL1	warning	Label not used within.
LABEL2	error	More than one declaration of label within.
LAMBDA1	error	Bad use of `lambda`.
LET1	error	Incorrect `let` variable definition.
LET2	hint	`let` statement has no local variables, so can be removed.
LET3	hint	Variable repeated in local variable list for `let`.
LET4	warning	Variable used before available in `let` assignment.
LET5	error	`let` statements will not accept more that 255 local variables in the next release.
LOAD1	warning	Can't evaluate to an `include/load` file.
LOOP1	error	First argument must be a symbol or list of symbols.
LoadFile	information	Loading file.
MEMBER1	suggestion	Consider use of `memq` rather than `member`.
MultiRead	information	Attempt to read file more than once.
NEQUAL1	hint	You may be able to replace with !=.
NEQUAL2	hint	You can replace with !=.
NTH1	hint	Can replace call to `nth` with call to `car`, `cadr`, and so on.
NoRead	error	Cannot read file.
PREFIXES	information	Using package prefixes.See <u>Check Function and Global Variable Prefixes</u>.
PREFIX1	warning	Prefixes must be all lower case or all upper case. See <u>Check Function and Global Variable Prefixes</u>.
PRINTF1	error	Incorrect number of format elements.
PRINTF2	error	Format argument is not a string.

Built-in Message Name	Message Group	Message Description
PROG1	`error`	Bad action statement.
PROG2	`hint`	`prog` construct may be removed.
PROG4	`hint`	Variable repeated in local variable list of `prog`.
PROG5	`hint`	`prog` may be replaced with `progn`.
PROG6	`hint`	Will need a `nil` at end if `prog` removed.
PROGN1	`hint`	`progn` with only one statement can be removed.
PUTPROP1	`information`	The autoload symbol is no longer used for functions in contexts.
REMOVE1	`suggestion`	Consider using `remq` rather than `remove`.
REP	`SKILL lint run message`	Short for report. REP is not based on the content of the program.
RETURN1	`warning`	Not within a `prog: return`.
RETURN2	`hint`	Replace `return(nil)` with `return()`.
SETQ1	`error`	First argument should be a symbol.
SETQ2	`suggestion`	Possible variable initialized to `nil`.
SETQ3	`suggestion`	Assignment to loop variable.
SKFATAL	`fatalerror`	Error found from which SKILL Lint cannot proceed.
STATUS1	`error`	Second argument must be `t` or `nil`.
STATUS2	`error`	Unknown status flag.
STATUS3	`warning`	Internal (s) status flag, do not use.
STRCMP1	`hint`	Inefficient use of `strcmp`. Change to `equal`.
STRICT	`information`	Applying strict checking of global variable prefixes.
		See <u>Check Function and Global Variable Prefixes</u>.
		Note: `sklint` calls the `skIgnoreMessage` or `skUnignoreMessage` SKILL API when you disable or enable strict checking. For more information on these APIs, see <u>Lint Functions</u>.
STRLEN1	`hint`	Inefficient use of `strlen`. Change to `equal ""`.

Built-in Message Name	**Message Group**	**Message Description**
Checks	information	Applying SKILL Lint checks.
Form	information	Form being read by SKILL Lint.
Read	information	This message is given for each file that is analyzed.
Unused	unused vars	Variable does not appear to be referenced.
VAR	information	Variable used or set in function/file.
VAR0	information	Variable used or set in function/file.
VAR1	error	Attempt to assign a value to t.
VAR4	information	Variables used as both global and local.
VAR5	information	Unrecognized global variables.
VAR6	information	Acceptable global variables.
VAR7	error global	Variable used as both a local and global.
VAR8	warning global	Global variable does not begin with package prefix.
VAR9	package global	Global variable begins with package prefix.
VAR12	warning	Argument does not appear to be referenced.

For VAR12:

Add an underscore (_) prefix to an argument in a function definition to indicate that it is intentionally unused within the body of the function. For example:

```
procedure(notused(a b)
  t
)
```

then you get a warning from SKILL Lint.

Built-in Message Name	Message Group	Message Description
VAR13	information	Internal global variable does not appear to be referenced.
		If the arguments are prefixed with an underscore, for example:

		`procedure(notused(_a _b)`
		`  t`
		`)`

		than, SKILL Lint displays an INFO message.
VAR14	information	Package global variable does not appear to be referenced.
VAR15	error	Variable cannot begin with keyword symbol (?).
VAR15	error	Variable cannot begin with keyword symbol (?) in the next release.
VAR16	information	Variable declaration hides a previous declaration.
WHEN1	hint	Invert test and convert to `when/unless`.

Related Topics

SKILL Lint

SKILL Lint Features

Message Groups

SKILL Lint PASS/FAIL and IQ Algorithms

SKILL Lint PASS/FAIL and IQ Algorithms

SKILL Lint uses a standard reporting mechanism that supports the following:

■ Allows any program to report messages in a consistent manner to the screen and log files.

■ Allows messages to be switched off.

■ Prints a summary at the end.

■ Gives a simple way of changing messages to a different language.

You can register different message classes—such as `information`, `warning`, and `error`—and specify whether generating a message of that class should cause overall failure.

In SKILL Lint, the following classes cause a failure (status `FAIL`):

■ `error global`

■ `error`

■ `fatal error`

■ `warning`

A case fails if it has a `warning`. Even if SKILL Lint does not issue the `warning` because the message has been switched off, it still appears in the summary scores and the status.

Note: A case may have an IQ score of `0`, but if there is nothing to cause a real failure, the overall status can still be pass.

The IQ score is specific to SKILL Lint and is based on the number of each message class issued, multiplied by a factor for each different class.

■ Most classes score zero.

■ The following classes score 1:

❑ `warning`

❑ `error`

❑ `error global`

❑ `warning global`

❑ `unused var`

❑ `authorization`

■ `Fatal error` scores 100.

The final score is the lower of the following two values:

■ Value One: The figures are totalled, divided by the number of top level forms (the number of `lineread` statements performed by SKILL Lint in parsing the files), and multiplied by 20. This figure is subtracted from 100 to give the score. The minimum score is zero.

■ Value Two: There is a class called `shortListErrors`, which consists only of the number of `error` class messages. This is multiplied by 25 if you run SKILL Lint on a

single file or by 10 if you run `sklint` across several files. The result is again subtracted from 100.

There is no cost to the IQ or pass/fail (with respect to the score) for undefined functions.

Note: If a particular message group is turned off using `skIgnoreMessage` or `?ignoresMessageList`, these messages are neither printed by SKILL Lint nor counted in the final score at the end of the run.

Related Topics

SKILL Lint Features

Message Groups

Built-In Messages

C

SKILL Lint Rules

SKILL IDE provides a mechanism for you to write your own rules to output SKILL Lint messages.

Rule Structure - SK_RULE Macro

The `SK_RULE` macro is the main entry point for writing a rule:

`SK_RULE(` *sl_functions g_test g_statement ...*`)`

The components of a `SK_RULE` call are as follows:

sl_functions	Name of the function to which the rule applies. Rules in SKILL Lint always apply to a particular function. For example, there is a rule associated with the `setq` function (the assignment operator) which says that the first argument must be a symbol. The first argument to `SK_RULE` may be a single function name or it may be a parenthesized list of function names if the same rule is to be applied to multiple functions.
g_test	SKILL statement known as the test statement. The rules work by applying a series of commands whenever a call to the function(s) named is found in the code under analysis. The test function is evaluated first, and the rest of the commands are carried out only if the test function evaluates to non-nil.
g_statement	One or more rules commands (SKILL statements) that are executed whenever a call to the named function(s) is found, providing that the test statement evaluates to non-`nil`.
	While the rule command statements are being evaluated, a number of macros are available for accessing the SKILL code being checked and for reporting any problems found. The simplest macro is `SK_ARGS` which takes no arguments and returns the list of arguments to the function call being tested.

The macros you can use to write rules begin with `SK_` and have all capital letters.

For example, the following rule applies to the `ggTestData` function which has two required arguments and an optional third. If, in the next release, the third argument becomes mandatory, this rule will find all calls with only two arguments:

```
SK_RULE( ggTestData
  length(SK_ARGS()) == 2
  warn("Found call to ggTestData with only 2 arguments.\n")
)
```

Related Topics

Rule Access Macros

Rule Reporting Macros

Adding a New Required Argument to a Function

Rule Access Macros

You can use the following macros in either the test statement or the rules commands of the `SK_RULE` macro:

Rule Access Macro	Description
`SK_ARGS()`	Returns the list of the arguments to the function call under test. This macro takes no arguments. The list values returned by this macro should never be destructively altered (using `rplaca` etc.) because that would produce unknown effects.
`SK_CUR_FILENAME()`	Returns the name of file currently being checked in a SKILL Lint rule. For example: ```SK_RULE(test` ` t` ` printf("Current file being checked is: '%s'\n"` ` SK_CUR_FILENAME())` `)```
`SK_NTH_ARG(`n`)`	Returns the specified argument number (n) in the function call. n is zero-based: 0 is the first argument to the function call; 1 is the second argument, etc. You must not destructively alter the list values returned by this macro (for example, using `rplaca`) because that would produce unknown effects.

Rule Access Macro	Description
`SK_FUNCTION()`	Returns the name of the function call under test. You might want to establish the function name where the same rule is being used for several different functions. You must not destructively alter the list values returned by this macro (for example, using `rplaca`) because that would produce unknown effects.
`SK_FORM([`*n*`])`	Returns the entire function call under test as a list. If you specify *n*, this macro returns the call *n* levels up the call stack. For example, if an `if` is called in a `foreach` that is in a `let`, `SK_FORM(2)` returns the call to `let`. **Note:** `SK_FORM(0)` is the same as `SK_FORM()`. `SK_ARGS` is effectively the same as `cdr(SK_FORM())` and `SK_FUNCTION` is effectively the same as `car(SK_FORM())`. You must not destructively alter the list values returned by this macro (for example, using `rplaca`) because that would produce unknown effects.

Related Topics

SKILL Lint Rules

Rule Reporting Macros

Advanced Rule Macros

Rule Reporting Macros

You can use the following macros in SK_RULE macros to report errors, warnings, hints, and information to the user in the same format that SKILL Lint uses when generating standard messages:

```
SK_ERROR( type format arg ...)
SK_WARNING( type format arg ...)
SK_HINT( type format arg ...)
SK_INFO( type format arg ...)
```

The arguments are as follows:

type	Message identifier
format	Format string (as used by `printf`)
arg ...	One or more printing arguments

For example:

```
SK_WARNING( GGTESTDATA "This function now requires 3 arguments: %L\n" SK_FORM())
```

This macro prints a message of the following form:

```
WARN (GGTESTDATA) myFile.il line 32 : This function now requires 3 arguments:
ggTestData(abc 78.6)
```

You should use these macros in rules commands to report messages to the user when problems are encountered.

To allow the user to control the reporting of these messages the way they can with other SKILL Lint messages, use the `SK_REGISTER` macro outside the `SK_RULE` macro as follows:

```
SK_RULE(…
    ruleReportingMacro( type … )
)
SK_REGISTER( type )
```

For example:

```
SK_RULE( ggTestData
    length(SK_ARGS()) == 2
    SK_ERROR( GGTESTDATA "This function now requires 3 arguments: %L\n" SK_FORM())
)
SK_REGISTER( GGTESTDATA )
```

Related Topics

SKILL Lint Rules

Rule Structure - SK_RULE Macro

Rule Access Macros

Advanced Rule Macros

Advanced Rule Macros

Cadence provides the following advanced rule macros for your convenience:

- SK_CHANGED_IN(t_release)

- SK_CHECK_STRINGFORM(t_stringForm)

- SK_RULE(SK_CONTROL ...)

- SK_CHECK_FORM(l_form)

- SK_PUSH_FORM(l_form) SK_POP_FORM()

- SK_PUSH_VAR(s_var)

- SK_POP_VAR(s_var [dont_check])

- SK_USE_VAR(s_var)

- SK_ALIAS(s_function s_alias)

SK_CHANGED_IN(t_release)

This macro is used to specify the release version (e.g. "500" for IC5.0.0) that a function is changed. The `SK_CHANGED_IN` macro must be embedded as the second argument of `SK_RULE`. For example:

```
SK_RULE( myFunc
   SK_CHANGED_IN("500")
   SK_INFO( myFunc
   . . .
)
```

`SK_CHANGED_IN` evaluates to non-nil if the code being checked, as specified with the `sklint` argument `?codeVersion`, is from an earlier release than the release specified through the argument of `SK_CHANGED_IN` and the SKILL Lint rules message that describes function change (only) will be reported. The argument must me a numeric string of the release version (for example, *500* for IC 5.0.0). If `?codeVersion` is not specified, `SK_CHANGED_IN` will always evaluate to non-nil and a function change rules message will be reported.

This macro is useful when the user wants to restrict reporting of function change rule messages which occurred after the release for which the code being checked was written. When users check the code in IC500 they will not be interesting in seeing the information about the change in IC 4.4.5, since that was before they wrote the code (or perhaps before it was migrated).

If the function changes more than once, then there should be a separate SKILL Lint rule for each change, each with a different `SK_CHANGED_IN` macro.

Note: `SK_CHANGED_IN` should only be used for filtering out function changed rule messages. Function deleted rule messages should always be reported.

SK_CHECK_STRINGFORM(t_stringForm)

This macro is similar to `SK_CHECK_FORM` but it is used to check SKILL form in strings (e.g. callback string). This macro is added to deal with the problem that when a string form is converted to a SKILL form, the line number of the string form will be messed up and causes an incorrect line number to be reported.

An example of usage:

```
procedure( test()
    let( (c)
      c = myFunc(
          "foreach(i '(1 2 3 4) a=i)"
      )
     c
    )
)
SK_RULE( myFunc
    t
    SK_CHECK_STRINGFORM( SK_ARGS() )
)
```

Note: The argument to SK_CHECK_STRINGFORM must be a string.

SK_RULE(SK_CONTROL ...)

The `SK_RULE` macro has an optional first argument which is the keyword `SK_CONTROL`. When this keyword is given, it means that this rule is a "controlling" rule. This means that the arguments to the function are not themselves checked by SKILL Lint. Usually, SKILL Lint will first apply checking to all the arguments of a function call and then to the call itself. However, if there is a controlling rule, then the arguments are not checked automatically. This type of rule is usually needed for `nlambda` expression (for example `nprocedures`) where only some of the arguments are evaluated.

SK_CHECK_FORM(l_form)

This macro can be used to apply checking to a statement. This is usually useful within a controlling rule. The argument is a list whose first element is the SKILL code to be checked.

For example, consider a rule to be written for the `if` function (ignoring for the moment that there are internal rules for `if`.) This function evaluates all its arguments at one time or another, except for the `then` and `else` keywords. Writing a rule for `if` would require a controlling rule, which would call this macro to check all the arguments except for the `then` and `else`. For example:

```
SK_RULE( SK_CONTROL if
   t
   foreach(map statement SK_ARGS()
      unless(memq(car(statement) '(then else)) SK_CHECK_FORM(statement))
   )
)
```

The `SK_CONTROL` keyword means that the arguments to `if` will not be checked automatically. The test in this case is `t`, which means that the rule will be applied to all calls to `if`. The rule command is a call to `foreach`, with `map` as the first argument. Each time through the loop the statement is a new `cdr` of the arguments. We check that this is not a `then` or `else`, and if not, then call `SK_CHECK_FORM` to check the argument.

Note: The argument to `SK_CHECK_FORM` must be a list whose first element is the statement to check, not the statement itself.

It is important to call the checker on all appropriate arguments to a function, even if they are just symbols, because the checker handles trapping of variables which are unused, or are illegal globals and so forth.

There should only be a single control rule for any function.

SK_PUSH_FORM(l_form)
SK_POP_FORM()

These two macros are used to indicate an extra level of evaluation, such as is introduced by the various branches of a `cond` or `case` function call. These macros should not be needed by most user rules. They are used in rare circumstances to indicate to the dead-code spotting routines where branches occur in the code.

SK_PUSH_VAR(s_var)

Declares a new variable. For example, the rules for `let`, `setof`, etc. declare the variables in their first argument using this function. The function should be called *before* calling `SK_CHECK_FORM` on the statements in the body of the routine.

SK_POP_VAR(s_var [dont_check])

Pops a variable that was previously declared by `SK_PUSH_VAR`. Unless the second argument is `t`, the variable is checked to see whether it was used by any of the statements which were checked between the calls to `SK_PUSH_VAR` and `SK_POP_VAR`.

For example, consider a new function called `realSetOf`. Assume this function works just like `setof`, except that it removes any duplicates from the list that is returned. The rule is a control rule which pushes the variable given as the first argument, checks the rest of the arguments, and then pops the variable, checking that it was used within the loop:

```
SK_RULE( SK_CONTROL realSetOf
   t
   SK_PUSH_VAR(car(SK_ARGS()))
   map('SK_CHECK_FORM cdr(SK_ARGS()) )
   SK_POP_VAR(car(SK_ARGS()))
)
```

SK_USE_VAR(s_var)

Marks the given variable as having been used. Usually a variable is marked as having been used if it is passed to a function. However, if a function has a controlling rule, and does not call `SK_CHECK_FORM` then it might wish to mark a variable as having been used. For example, the rule for `putprop` marks the first argument as having been used. The same rule ignores the third argument (the property name) and calls the checker on the second argument. If `putprop` did not have a controlling rule, then the symbol used for the property name would get marked as having been used and would probably be reported as an error global.

SK_ALIAS(s_function s_alias)

This macro can be used where one function should be checked with the same rules as another function. For example, it is fairly common to see functions replacing `printf`, which add a standard prefix to the function. For example:

```
procedure( ERROR(fmt @rest args)
   fmt = strcat("ERROR: " fmt)
   apply('printf cons(fmt args))
)
```

It would be nice to check calls to ERROR with the same rules as are used for `printf` (mainly to check that the number of arguments matches that expected by the format string.) This can be achieved using the following call:

```
SK_ALIAS( ERROR printf )
```

This macro, like `SK_REGISTER`, is used outside of any rule definitions.

Related Topics

SKILL Lint Rules

Rule Structure - SK_RULE Macro

Rule Access Macros

Rule Reporting Macros

Advanced Rule Macros

Rule Definition Locations

Rule definitions belong in `.il` files stored in one of the following locations in your Cadence installation hierarchy (`your_install_dir`):

Location	Description
`your_install_dir`/tools/local/sklint/rules	
	Recommended location for user SKILL Lint rule definitions. Files stored in this location are loaded each time you run SKILL Lint. This location is not likely to be overwritten when you install a new release of Cadence software.
	Tip While you are developing new rules, it is useful to have these rules loaded each time you run SKILL Lint.
`your_install_dir`/tools/sklint/rules	
	Files stored in this location are loaded and ready for SKILL Lint when the SKILL Development environment context is loaded.

Related Topics

SKILL Lint Rules

Rule Structure - SK_RULE Macro

Rule Access Macros

Rule Reporting Macros

Advanced Rule Macros

Examples Using Macros

The examples in the following sections show how you can use macros in rules:

Adding a New Required Argument to a Function

You can write a rule like the following to trap problems associated with a function (such as `ggTestData`) requiring one or more new arguments with a new release of code and notifying the user:

```
SK_RULE( ggTestData
   length(SK_ARGS()) == 2
   SK_WARNING( GGTESTDATA
   strcat( "This function will require 3 arguments in the next release: %L\n"
   "The extra argument will specify the width of the widget.\n") SK_FORM())
)
SK_REGISTER( GGTESTDATA )
```

Replacing One Function with Another

You can write a rule like the following to replace a standard function called `setof` with another function called `realSetof`. The standard `setof` function is not a true `setof` because it does not remove repeated elements; instead, it is more of a `bagof` function. The replacement `realSetof` function removes repeated elements and allows many statements in the body of the function call (whereas `setof` allows only one). The rule needs to handle the fact that the first argument is a loop variable.

```
SK_RULE( realSetof
   t
   let( ((args SK_ARGS())))
      when(symbolp(car(args))
         SK_PUSH_VAR(car(args))
      )
      map('SK_CHECK_FORM cdr(args))
      when(symbolp(car(args))
         SK_POP_VAR(car(args))
      )
   )
)
```

The rule above uses `let` to declare a local variable `args` to save calling `SK_ARGS` many times. You can define a second rule as follows to check that the loop variable is given as a symbol:

```
SK_RULE( realSetof
   !symbolp(car(SK_ARGS()))
   SK_ERROR( REALSETOF1 "First argument must be a symbol: %L\n" SK_FORM())
)
SK_REGISTER( REALSETOF1 )
```

Promoting Standard Format Messages

You can write a rule as follows to check that the format string for three new functions matches the given number of arguments. The three new functions (`ggInfo`, `ggWarn` and `ggError`) take the same arguments as `printf` and usually work the same, except that they change the format a little and also copy the messages to various log files. The `SK_ALIAS` macro lets you alias the three new functions to `printf` so that you can apply the same rule.

```
SK_ALIAS( (ggInfo ggWarn ggError) printf )
```

Preventing Heavily Nested Calls to Boolean Operators

You can write a rule that promotes nicer looking code by preventing too many nested calls to boolean operators (`null`, `or`, and `and`).

Consider the following example, which is difficult to understand:

```
!a && ((b || !c) && (!d || !b)
```

To improve code readability, it would be better to split this expression into several statements and add associated comments.

The following rule counts boolean operators and warns you when there are "too many":

```
SK_RULE( (null and or)
   ggCountBools(SK_FORM()) > 5
   SK_HINT( BOOLS "Lots of boolean calls found : %L\n" SK_FORM())
)
SK_REGISTER( BOOLS )
```

You might write the `ggCountBools` function as follows:

```
procedure( ggCountBools(args)
   let( ((i 0))
      foreach(arg args
         when(listp(arg) && memq(car(arg) '(null or and))
            i = i + 1 + ggCountBools(cdr(arg))
         )
      )
      i
   )
)
```

In case of deeply nested booleans, you can improve the rule by looking at the function call higher in the call stack and, if that call is a boolean function itself, not checking the current call (because it is unnecessary):

```
SK_RULE( (null and or)
   !memq(car(SK_FORM(1)) '(null and or)) && ggCountBools(SK_FORM()) > 5
      SK_HINT( BOOLS "Lots of boolean calls found : %L\n" SK_FORM())
)
```

Related Topics

SKILL Lint Rules

Rule Structure - SK_RULE Macro

Rule Access Macros

Rule Reporting Macros

Advanced Rule Macros

Rule Definition Locations

D

SKILL Lint Environment Variables

The topic provides a list of the *SKILL Lint* environment variables that you can use to set the default values on the Lint tab of the Options form. You can set the values of these environment variables in the `.cdsenv` file.

alert	codeVersion	depends
error	globals	hint
information	internal	prefixes
printToLog	reportFile	reportFileName
suggestion	viewMessageFilter	warning

alert

```
sklint.Messages alert boolean { t | nil }
```

Description

If set to `t`, SKILL Lint alert messages are shown in the SKILL Lint Output Assistant.

Default is `nil`.

GUI Equivalent

None

Examples

```
envGetVal("sklint.Messages" "alert")
envSetVal("sklint.Messages" "alert" 'boolean t)
```

Related Topics

SKILL Lint Environment Variables

codeVersion

```
sklint.Input codeVersion string "codeVersion"
```

Description

Specifies the value for input argument `?codeVersion`, which signifies the release version of the code SKILL Lint should check.

Default is `non-applicable`.

GUI Equivalent

Command:	*Options – Lint*
Field:	*Code Version*

Examples

```
envGetVal("sklint.Input" "codeVersion")
envSetVal("sklint.Input" "codeVersion" 'string "abc")
```

Related Topics

SKILL Lint Environment Variables

depends

```
sklint.Input depends string "fileName"
```

Description

Specifies the value for input argument `?depends`, which lists the files that contain macro definitions on which the code under analysis depends.

Default is empty string.

GUI Equivalent

None

Examples

```
envGetVal("sklint.Input" "depends")
envSetVal("sklint.Input" "depends" 'string "file1")
```

Related Topics

SKILL Lint Environment Variables

error

```
sklint.Messages error boolean { t | nil }
```

Description

If set to `t`, SKILL Lint error messages are shown in the SKILL Lint Output Assistant.

Default is `t`.

GUI Equivalent

Command:	*Options – Lint*
Field:	*Errors*

Examples

```
envGetVal("sklint.Messages" "error")
envSetVal("sklint.Messages" "error" 'boolean nil)
```

Related Topics

SKILL Lint Environment Variables

globals

```
sklint.Input depends string "globalVariables"
```

Description

Specifies the value for input argument `?globals`, which stores the list of global variables in the code that SKILL Lint should check.

GUI Equivalent

Command: *Options – Lint*

Field: *Global Variables*

Examples

```
envGetVal("sklint.Input" "globals")
envSetVal("sklint.Input" "globals" 'string "var1")
```

Related Topics

SKILL Lint Environment Variables

hint

```
sklint.Messages hint boolean { t | nil }
```

Description

If set to `t`, SKILL Lint hint messages are shown in the SKILL Lint Output Assistant.

Default is `nil`.

GUI Equivalent

None

Examples

```
envGetVal("sklint.Messages" "hint")
envSetVal("sklint.Messages" "hint" 'boolean t)
```

Related Topics

SKILL Lint Environment Variables

information

```
sklint.Messages information boolean { t | nil }
```

Description

If set to `t`, SKILL Lint information messages are shown in the SKILL Lint Output Assistant.

Default is `t`.

GUI Equivalent

Command:	*Options – Lint*
Field:	*Information*

Examples

```
envGetVal("sklint.Messages" "information")
envSetVal("sklint.Messages" "information" 'boolean nil)
```

Related Topics

SKILL Lint Environment Variables

internal

```
sklint.Messages internal boolean { t | nil }
```

Description

If set to t, SKILL Lint internal error messages are shown in the SKILL Lint Output Assistant.

Default is t.

GUI Equivalent

None

Examples

```
envGetVal("sklint.Messages" "internal")
envSetVal("sklint.Messages" "internal" 'boolean nil)
```

Related Topics

SKILL Lint Environment Variables

prefixes

```
sklint.Input prefixes string "prefixes"
```

Description

Specifies the value for input argument `?prefixes`, which lists the acceptable package prefixes for functions and global variables.

Default is empty string.

GUI Equivalent

Command:	*Options – Lint*
Field:	*Package Prefixes*

Examples

```
envGetVal("sklint.Input" "prefixes")
envSetVal("sklint.Input" "prefixes" 'string "pre_")
```

Related Topics

SKILL Lint Environment Variables

printToLog

```
sklint.Output printToLog boolean { t | nil }
```

Description

If set to `t`, SKILL Lint output is printed to the CIW.

Default is `nil`.

GUI Equivalent

Command:	*Options – Lint*
Field:	*Print To Log*

Examples

```
envGetVal("sklint.Output" "printToLog")
envSetVal("sklint.Output" "printToLog" 'boolean t)
```

Related Topics

SKILL Lint Environment Variables

reportFile

```
sklint.Output reportFile boolean { t | nil }
```

Description

This environment variable works together with the `reportFileName` variable. If set to `t` and the value of `reportFileName` is a valid path to a log file, SKILL Lint output is shown in a new window as a text file.

Default is `nil`.

GUI Equivalent

Command:	*Options – Lint*
Field:	*Report File*

Examples

```
envGetVal("sklint.Output" "reportFile")
envSetVal("sklint.Output" "reportFile" 'boolean t)
```

Related Topics

SKILL Lint Environment Variables

reportFileName

```
sklint.Output reportFileName string "FilePath"
```

Description

Represents the full path to a log file for SKILL Lint output.

Default is an empty string.

GUI Equivalent

Command: *Options – Lint*

Field: *Report File Name*

Examples

```
envGetVal("sklint.Output" "reportFileName")
envSetVal("sklint.Output" "reportFileName" 'string "/home/file1")
```

Related Topics

SKILL Lint Environment Variables

suggestion

```
sklint.Messages suggestion boolean { t | nil }
```

Description

If set to t, SKILL Lint suggestion messages are shown in the SKILL Lint Output Assistant.

Default is nil.

GUI Equivalent

None

Examples

```
envGetVal("sklint.Messages" "suggestion")
envSetVal("sklint.Messages" "suggestion" 'boolean t)
```

Related Topics

SKILL Lint Environment Variables

viewMessageFilter

```
sklint.Messages viewMessageFilter string "MessageID"
```

Description

The string specifies the SKILL Lint message IDs (separated by spaces) that should not be
shown in the SKILL Lint Output Assistant.

Default is `RedefMsgID Read Form Checks MultiRead LOAD1 ExtKnown`
`FnsDefined FnsLocal FnsLocal0 Flow VAR6 VAR9 VAR10 VAR11 DEF7 SETQ4.`

GUI Equivalent

None

Examples

```
envGetVal("sklint.Messages" "viewMessageFilter")
envSetVal("sklint.Output" "viewMessageFilter" 'string "Msg-001")
```

Related Topics

SKILL Lint Environment Variables

warning

```
sklint.Messages warning boolean { t | nil }
```

Description

If set to `t`, SKILL Lint warning messages are shown in the SKILL Lint Output Assistant.

Default is `t`.

GUI Equivalent

Command:	*Options – Lint*
Field:	*Warnings*

Examples

```
envGetVal("sklint.Messages" "warning")
envSetVal("sklint.Messages" "warning" 'boolean nil)
```

Related Topics

SKILL Lint Environment Variables

E

SKILL API Finder

The SKILL API Finder displays abstracts and syntax statements for language functions and APIs. It reads the language information that each product loads from your Cadence hierarchy.

Note: Your database will vary according to the products loaded on your system.

You can add your own functions locally for quick reference. The SKILL API Finder can display any information that is properly formatted and located.

Related Topics

Starting SKILL API Finder

Searching in SKILL API Finder

Types of Searches in SKILL API Finder

Cadence Data

Customer Data

Problem Troubleshooting in SKILL API Finder

Starting SKILL API Finder

To start the SKILL API Finder from the CIW:

➡ Choose *Tools – SKILL API Finder*. The Cadence SKILL API Finder window appears.
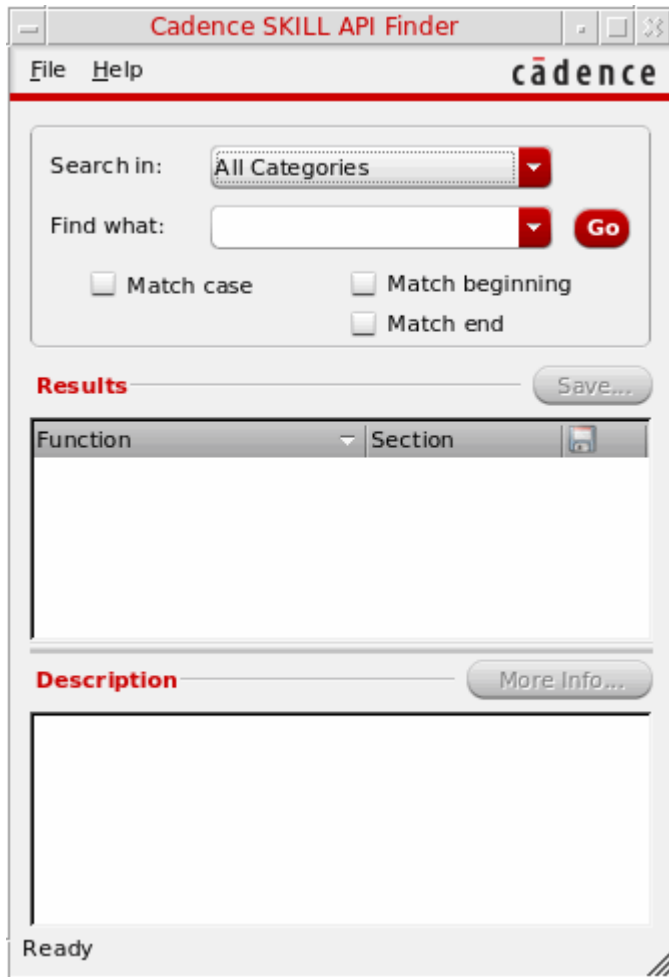
To start the SKILL API Finder from a UNIX command line, type the following:

```
cdsFinder
```

Starting the Finder in Test Mode

When you run in test mode, the Finder writes a report file called `/tmp/finder.tst` as well as writing to standard output. The Finder reports directories it finds in the database and the number of entries found in each file.

To start the Finder in test mode, run the `cdsFinder` command with the `-t` option as follows:

```
cdsFinder -t
```

To check for duplicate instances of each name in a particular directory's data files, add the directory name as follows:

```
cdsFinder -t checkDir
```

The *checkDir* you specify should include the *language* and *functionArea* subdirectories. For example:

```
cdsFinder -t doc/finder/SKILL/SKILL_Language
```

Related Topics

SKILL API Finder

Searching in SKILL API Finder

Types of Searches in SKILL API Finder

Problem Troubleshooting in SKILL API Finder

Searching in SKILL API Finder

To start a search, do the following:

1. Choose *Tools – SKILL API Finder*.

   The Cadence SKILL API Finder form is displayed.

2. Click *Go*.

Matches appear in the *Results* area (to a maximum of 500). If your search produces more than 500 results, a message displays in the status bar at the bottom of the SKILL API Finder window.

Related Topics

SKILL API Finder

Starting SKILL API Finder

Types of Searches in SKILL API Finder

Problem Troubleshooting in SKILL API Finder

Types of Searches in SKILL API Finder

You can perform the following types of searches:

■   Categories

■   Simple Strings

■   Combinations

■   Multiple Strings

Categories

By default, the SKILL API Finder searches all categories in the Cadence database and the optional Customer database. Therefore, *All Categories* appears in the *Search in* cyclic field. To confine a search to a particular category of SKILL API Finder data, do the following:

➤   Select the required category from the *Search in* drop-down list and click *Go*.
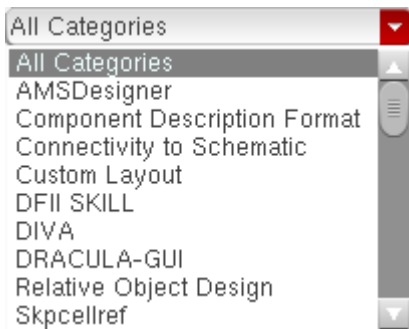
The set of categories that appears in your *Search in* cyclic field depends on the products installed on your system.

The SKILL API Finder confines all searches for functions in the selected category.

To see all items in the selected category, type `.*` in the *Find what* field and click *Go*.

Simple Strings

You can search for simple strings in the selected set of Finder data. Matches appear in the *Results* area. Search strings from all previous searches are saved as search history. This includes any searches carried out in previous sessions.

If you type a search string and do not mark any of the search string qualifier check boxes (*Match case*, *Match beginning*, or *Match end*), all items containing that string—without regard to casing—appear in the *Results* area:
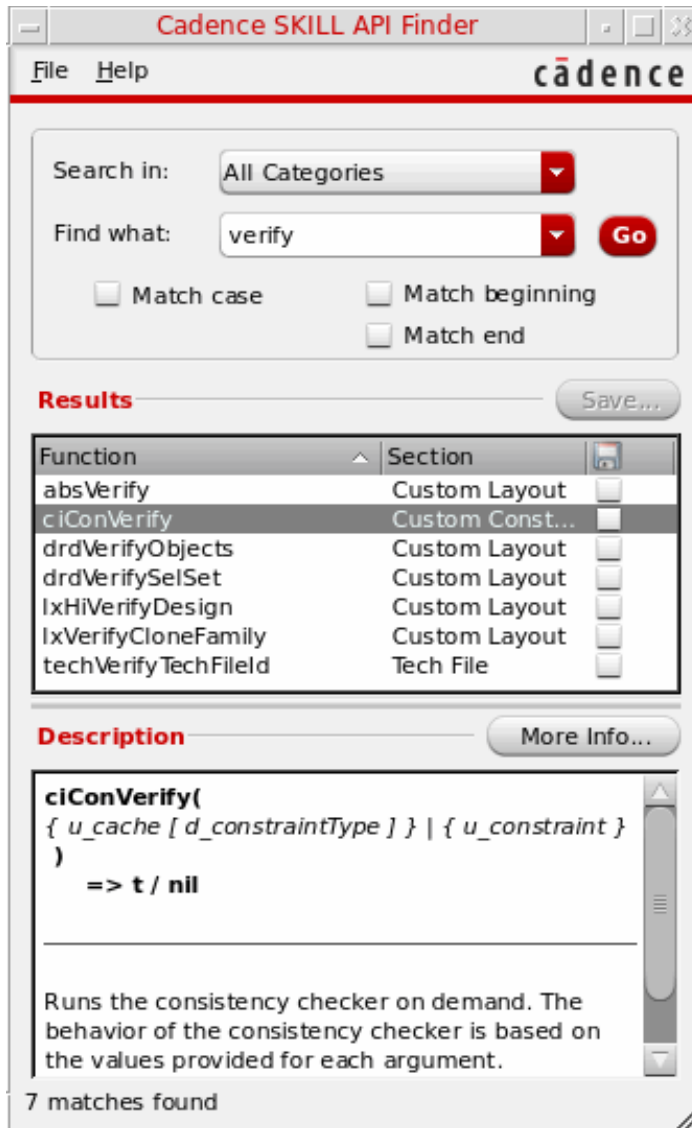
If you mark the *Match end* check box, only those matches where your search string occurs at the end of an item appear in the *Results* area:
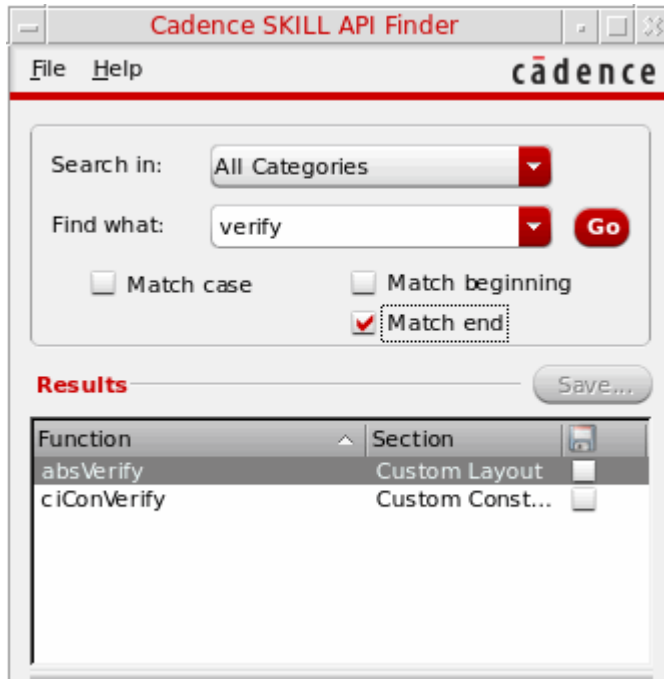
As an alternative to selecting the *Match end* check box, you can type $ at the end of your search string. For example:

```
string$
```

If you select the *Match beginning* check box, only those functions or APIs where your search string occurs at the beginning appear in the *Results* area (as illustrated in the image below).
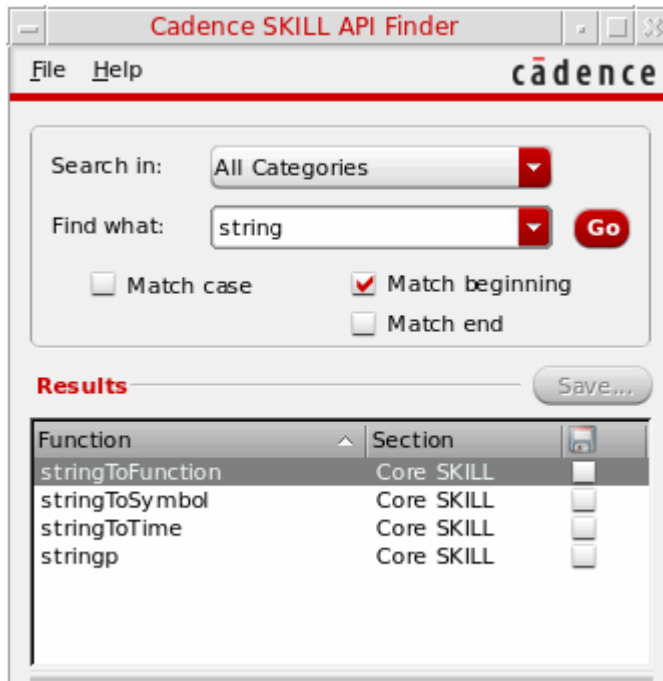
As an alternative to selecting the *Match beginning* check box, you can type ^ at the beginning of your search string. For example:

```
^string
```

You can further restrict your search by selecting the *Match case* check box.
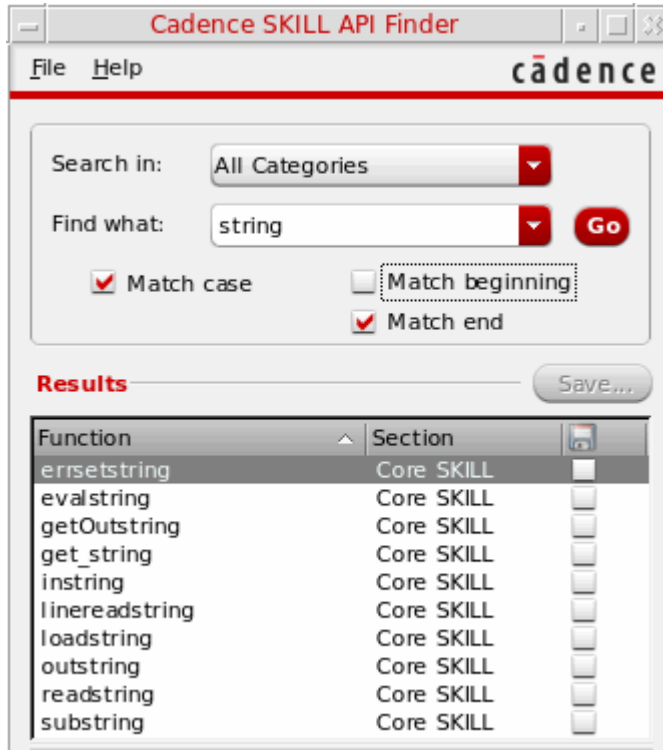
Combinations

You can use `.*` in your search string to create combinations, such as "any item containing some letter or string followed by some other letter or string with any number of other letters in between." You can combine the use of `.*` with the search string qualifier check boxes to refine your search.

Here is an example that uses the *Match beginning* and *Match case* check boxes in combination with `.*` to find all items containing a lowercase letter e followed by all-lowercase

`string` with zero or more of any other letters in between, where `string` must occur at the end of the item:
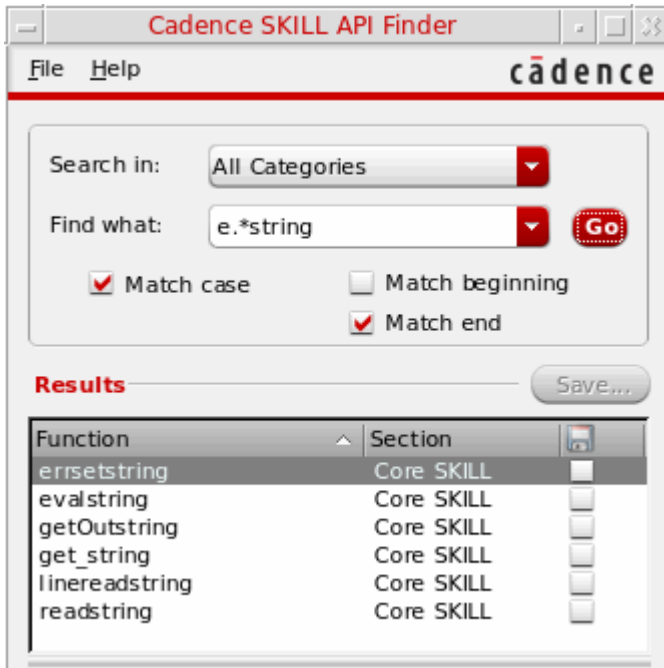
Here is an example using the *Match beginning* check box in combination with `.*` to find all items starting with the letter `e` (or `E`) followed by case-insensitive `String`:
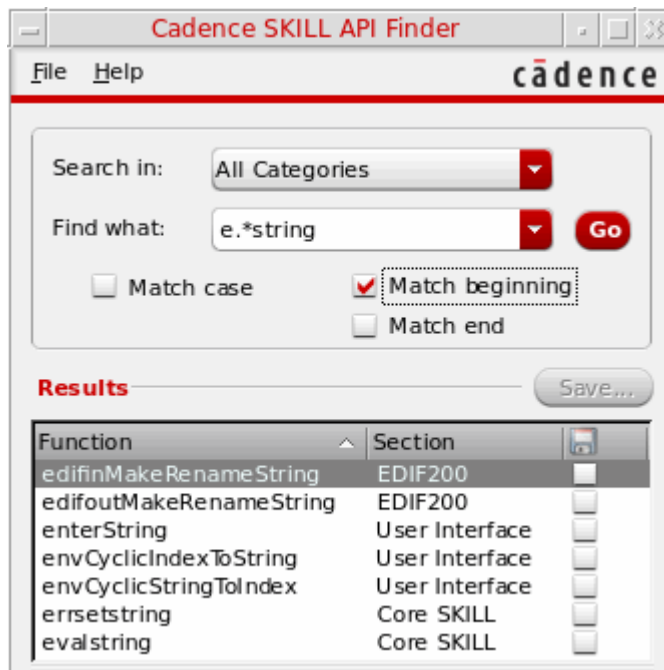
Multiple Strings

If you specify multiple search strings in the *Find What* field, all functions or APIs that have the searched strings, irrespective of the order, will appear in the *Results* area. The search results will ignore the *Match beginning* and *Match end* check boxes even if they are selected as the string qualifiers.

Here is an example of using multiple search strings to search the APIs that have the words db, save, and trigger in any order. The search results are case insensitive.
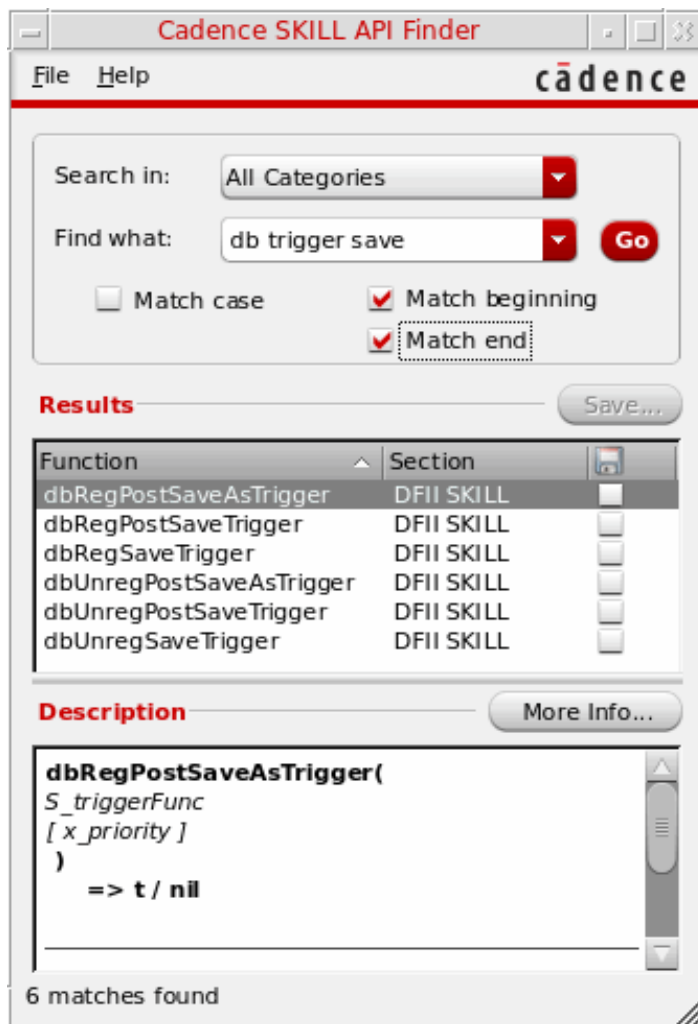
Related Topics

SKILL API Finder

Searching in SKILL API Finder

Starting SKILL API Finder

Problem Troubleshooting in SKILL API Finder

Accessing SKILL API Help

Quick reference information for SKILL APIs is available from the CIW and the SKILL API Finder. To access the reference information for a particular SKILL API, do one of the following:

- Type `help <function_name>` in the CIW.

- Type `startFinder ( [ ?funcName` _`t_functionName`_ `] )` in the CIW.

- Start the <u>SKILL API Finder</u> from the CIW by choosing _Tools – Finder_ or type `cdsFinder` on the UNIX command line.

  In the _Search in_ field of the displayed Cadence SKILL API Finder window, type the SKILL API name for which you want to display the help information and click _Go_.

  The matches for the searched SKILL API appear in the _Results_ area.

  To view the complete documentation of the searched SKILL API, select the API name in the _Results_ area and click the <u>_More Info_</u> button. The complete documentation of the selected SKILL API appears in a new Cadence Help window.

Viewing and Saving Syntax and Description of Matches Found

The SKILL functions and APIs that match the search string are populated in the *Results* area. Scroll bars appear as necessary. When you highlight a particular match by clicking it, the related syntax and text description appears in the *Description* area.
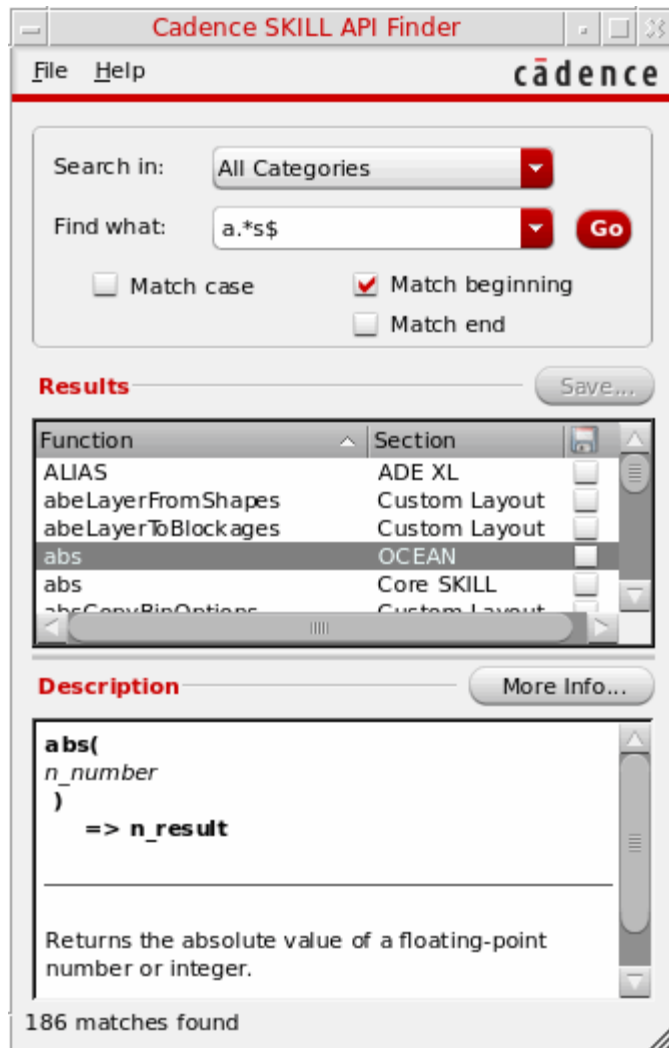
Viewing Detailed Descriptions of SKILL APIs

In addition to viewing the syntax and text descriptions that appear in the *Description* area, you can also view the complete API documentation including the arguments, return values, and examples. To do so:

➡ Select a match in the *Results* area and click *More Info.* Alternatively, double-click the selected API or press Enter.

The complete documentation of the selected API appears in the Doc Assistant window.
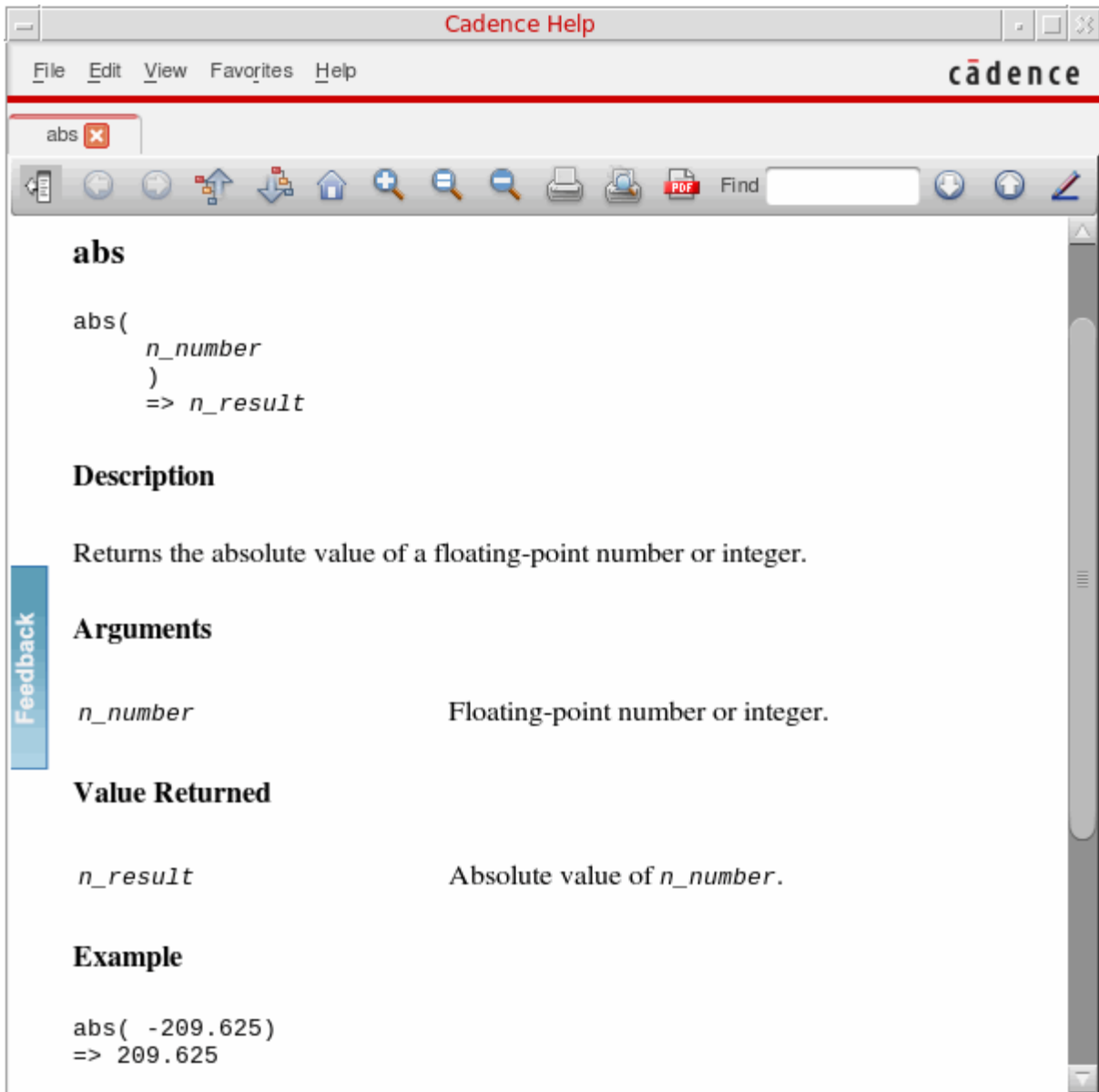
If you click the *More Info* button again, the documentation of the selected API appears in a new tab in the Doc Assistant window.

Video

> For a video demonstration on the use of the *More Info* button to quickly locate detailed description of SKILL APIs, see Viewing Detailed Documentation of SKILL APIs on Cadence Online Support.

Saving SKILL API Descriptions in a Text File

You can save the syntax and description of one or more matches in a text file. To do so, perform the following steps:

1. Select the check box corresponding to the required function displayed in the *Results* area.

2. Click *Save*. The Save text descriptions form appears.
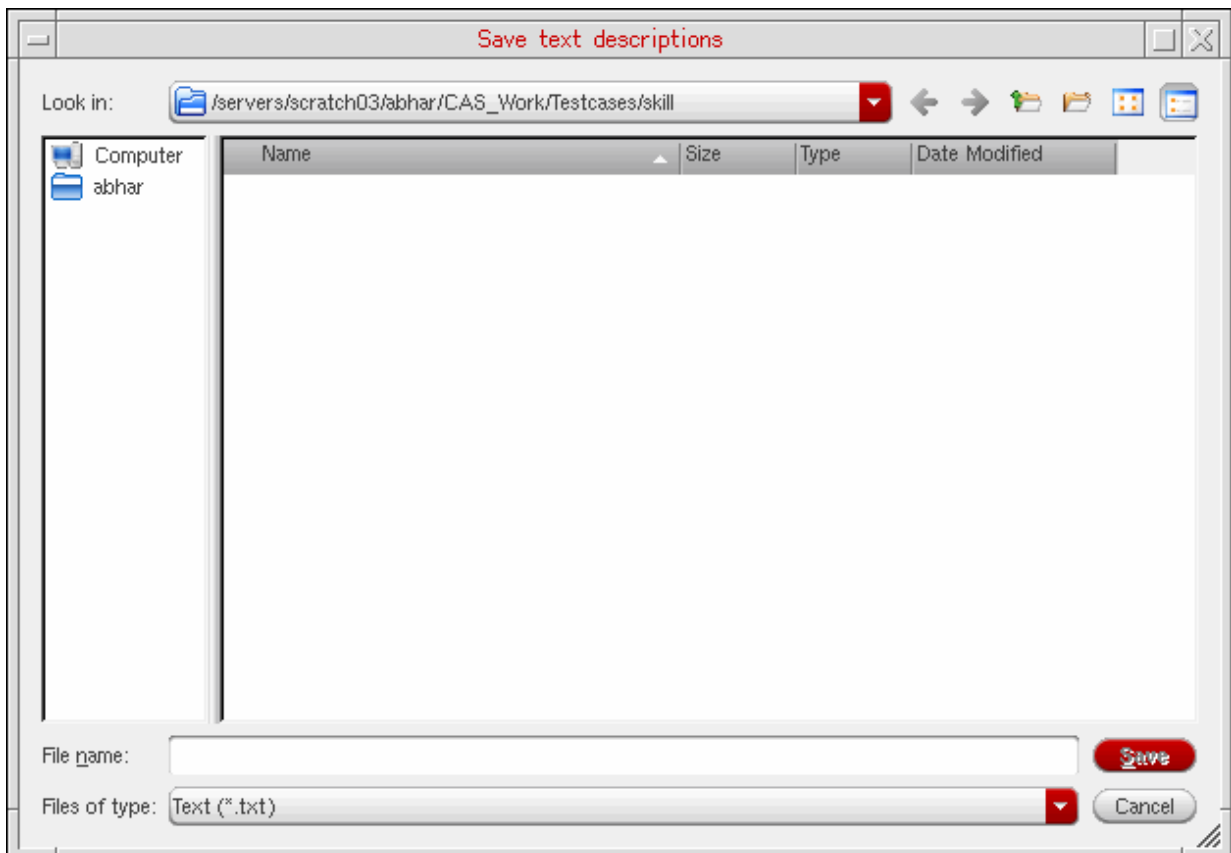
3. In the *File name* field, specify a new or existing filename.

   By default, the file gets saved in the current work directory. If you want to save the file in some other directory, you can choose the directory using the *Look in* drop-down list box.

Alternatively, you can use the two directory icons, one for accessing the Parent Directory and another for creating a New Folder.

4. Click *Save*.

If the specified file exists, the SKILL API Finder appends the *Description* of the selected function(s) to the end of that file. If the file does not exist, the SKILL API Finder creates it.

Related Topics

SKILL API Finder

Starting SKILL API Finder

Searching in SKILL API Finder

Types of Searches in SKILL API Finder

Problem Troubleshooting in SKILL API Finder

Cadence Data

You can find Cadence-supplied information here:

your_install_dir/doc/finder/*language*/*functionArea*/*.fnd

⚠ *Important*

Database files must have .fnd as the file extension, such as chap1.fnd. All files must contain information in the appropriate Data Format.

where

your_install_dir   is the directory where your Cadence software is installed.

💡 *Tip*

Type cdsGetInstPath and press *Return* on the command line in your Command Interpreter Window to find *your_install_dir*.

language   is the language type, such as SKILL.

functionArea   is a descriptive subdirectory name for the product information, such as `SKILL_Language` or `SKILL_Development`.

Note: Underscores used in subdirectory names to separate words appear as spaces in the *Searching* cyclic field. The word *Only* appears after each entry.

Related Topics

Customer Data

Customer Data

You can add your own internal functions to the database by copying your information to the directory where Cadence-supplied information is installed. Your subdirectory name will appear as a choice in the *Searching* cyclic field, in alphabetical order, with spaces in place of underscores. The word *Only* appears after the subdirectory name.

For example, if you add your functions to

your_install_dir/local/finder/SKILL/Your_APIs/your.fnd

the following string appears in the *Searching* cyclic field:

Your APIs Only

Specifying Additional Finder Data Directories

You can use the `CDS_FINDER_PATH` environment variable to specify additional directories for Finder database files. After the Finder loads data from

your_install_dir/doc/finder/*language*/*functionArea*/*.fnd

it then loads data from a colon-separated list of one or more directories you specify using this environment variable. The path or paths you specify can be a combination of absolute and relative paths.

For example:

setenv CDS_FINDER_PATH .:/home/myFinder/db:../../projectFinderFiles

The program will look for your Finder database files (`*.fnd`) two directory levels below the paths you specify (corresponding to *language*/*functionArea*/*.fnd). Cadence strongly recommends that the additional directories for Finder database files follow the naming conventions outlined for Cadence data.

For example, if you specify

```
setenv CDS_FINDER_PATH /home/myFinder/db
```

the Finder database files must be located in a directory path such as

```
/home/myFinder/db/SKILL/myFunctionArea/
```

Data Format

The Finder requires the following three-string format for each unique entry in the `*.fnd` files:

```
("functionName"
"syntaxString"
"AbstractText.")
```

For example:

```
("abs"
"abs( n_number ) => n_result"
"Returns the absolute value of a floating-point number or integer.")
```

If you have more than one function that performs the same task, you can put them together as follows:

```
("functionName1, functionName2, …"
"syntaxString1"
"syntaxString2"
…
"AbstractText.")
```

For example:

```
("sh, shell"
"sh( [t_command] ) => t/nil
shell( [t_command] ) => t/nil"
"Starts the UNIX Bourne shell sh as a child process to execute a command string."
```

Related Topics

Cadence Data

Problem Troubleshooting in SKILL API Finder

Too Many Matches

```
More than 500 matches have been found. Please use a more restrictive search string.
```

Change the search string to limit the number of matches.

Save File Is Not Writable

filename `is not a writable file. Please enter a new file name.`

This message appears if any aspect of specifying the file name results in an error. Click *OK*. The error message disappears leaving the File Name Entry form on the screen so you can type another name.

No Files Found

Look in the Cadence database directory to see if any files were loaded at installation:

your_install_dir`/doc/finder/`*language*`/`*functionArea*`/*.fnd`

Descriptions List Area Full

`WARNING: The display has reached its maximum capacity. Please save (if desired) and clear the window.`

If the number of characters in the *Descriptions* list area exceeds one megabyte, the current expansion operation aborts and this message appears. You can clear the list area by clicking *Clear* with or without saving your information first.

Related Topics

Starting SKILL API Finder

Searching in SKILL API Finder

Viewing and Saving Syntax and Description of Matches Found

F

SKILL IDE Document Generation

As a SKILL programmer, you may need to reuse the code written by other developers or provide your code to other developers for integration. When it comes to programming, documenting the details of the code is as important as the code itself. Since the code evolves over time, it is imperative that the documentation stays synchronized with the code. The best way to achieve this is to embed the documentation within the code.

SKILL IDE's inline documentation capability allows you to embed documentation strings within SKILL/SKILL++ functions, classes, and methods. These documentation strings describe the attributes of the code elements with which they are associated. For example, the documentation string for a SKILL function may contain the description, parameters, and return values for the function.

SKILL IDE also has a document generation utility called Finder Manager, which inspects the code containing inline documentation and generates finder-compatible documentation from it.

Video

For a video demonstration of SKILL IDE's inline documentation capabilities, see Generating Documentation from SKILL/SKILL++ Code using Finder Manager on Cadence Online Support.

Related Topics

Documentation-specific Code

Output Formatting

Extracting Documentation using Finder Manager

Viewing the Generated Documentation in Finder

Documentation-specific Code

This topic provides information on writing documentation-specific code in SKILL/SKILL++.

For the Finder Manager to extract documentation from the SKILL/SKILL++ source files, you need to insert inline documentation strings in the code blocks. Each inline documentation string in a code block is associated with a declaration.

The following section describes the elements of an inline documentation string.

Inline documentation strings are added right after the definition of a class, function, or method in the source file. Opening quotation " and closing quotation " mark the beginning and end of a documentation string.

Documentation strings may also contain some additional keywords that describe the attributes of classes, functions, or methods being defined. These keywords are identified by an @ symbol.

Keywords for Functions, Classes, and Methods in SKILL and SKILL++

The following table lists the keywords that are common for functions, classes, and methods in SKILL and SKILL++:

Keyword	Description
@brief *<text>*	A brief description of the function, method, or class
@param *<name>*	The parameter name
@return *<value>*	The function return value

Inline documentation strings are supported for the following SKILL functions and classes: `defun`, `procedure`, `nprocedure`, `defmethod`, `defgeneric`, `defglobalfun`, `globalProc`, `defclass`, and `ansiDefmethod`.

Inline documentation strings for functions and classes are treated differently. So, there's no conflict if you specify the same name for a function as well as a class.

Sample Inline Documentation Strings for Functions, Classes, and Methods in SKILL and SKILL++

```
defun(mytest1 (x y)
   "test function mytest1 <BR> for test use
    @brief a brief description ...
    @param x x parameter
    @param y Y parameter
    to compute x + y
    @return x_sumValue
    Returns the sum of x and y"
  x + y)

procedure(testp(x  y  "ng")
     "documentation of testp"
       <function body> )



nprocedure(testn(x  y  "ng")
     "documentation of testn"
       <function body> )


defglobalfun(testg (x y "ng")
"documentation of testg"
<function body>)


globalProc(testgp(x  y  "ng")
     "documentation of global proc"
       <function body> )
```

Keywords for Classes and Methods in SKILL++

The following table lists the keywords that are specific to SKILL++ classes and methods:

Keywords for SKILL++ Classes

Keyword	Description
`@brief <text>`	A brief description of the class
`@slot <description>`	Slot description of the SKILL++ class

Keywords for SKILL++ Methods

Keyword	Description
`@brief <text>`	A brief description of the method
`@spec <spec-list>`	The list of specializers
`@return <value>`	Return value for the method

Sample Inline Documentation Strings for Classes and Methods in SKILL++

In SKILL++ classes, the inline documentation strings are added after the slot list. For example:

```
defclass(device
    "class DEVICE description
    @slot type describes the type of device"
    (figure) ((type @initform `undef)))
```

In SKILL++ methods, the inline documentation strings are added after the specializer list. For example:

```
defmethod(draw  ((obj  device) (cv dbobject))
  "documentation of DRAW method . . .
      @spec (device dbobject)
      @return t/nil"
      <method body>)


ansiDefmethod(draw  ((obj  device) (cv dbobject))
  "documentation of DRAW method . . .
      @spec (device dbobject)
      @return t/nil"
      <method body>)


defgeneric(draw  (obj  cv)
  "documentation of DRAW generic function . . .
      @spec generic" )
```

Related Topics

SKILL IDE Document Generation

Output Formatting

Extracting Documentation using Finder Manager

Viewing the Generated Documentation in Finder

Output Formatting

SKILL IDE supports the following HTML tags for formatting the appearance of the documentation generated from the inline documentation strings:

■   `<B>text</B>`: To display the text in bold.

■   `<I>text</I>`: To display the text in italics.

■   `<U>text</U>`: To display underlined text.

■   `<BR>`: To insert a line break in text.

■   `<COLOR=green>text</COLOR>`: To specify the color of the text.

Related Topics

SKILL IDE Document Generation

Documentation-specific Code

Extracting Documentation using Finder Manager

Viewing the Generated Documentation in Finder

Extracting Documentation using Finder Manager

After you add inline documentation strings in your SKILL/SKILL++ source files, you are ready to generate finder-compatible documentation using Finder Manager. Finder Manager processes the code blocks containing inline documentation strings and saves the output in a finder (`.fnd`) file.

To create a finder file from your source file:

1. Start the *Finder* assistant by choosing *Window – Assistants – Finder.* The *Finder* assistant displays.
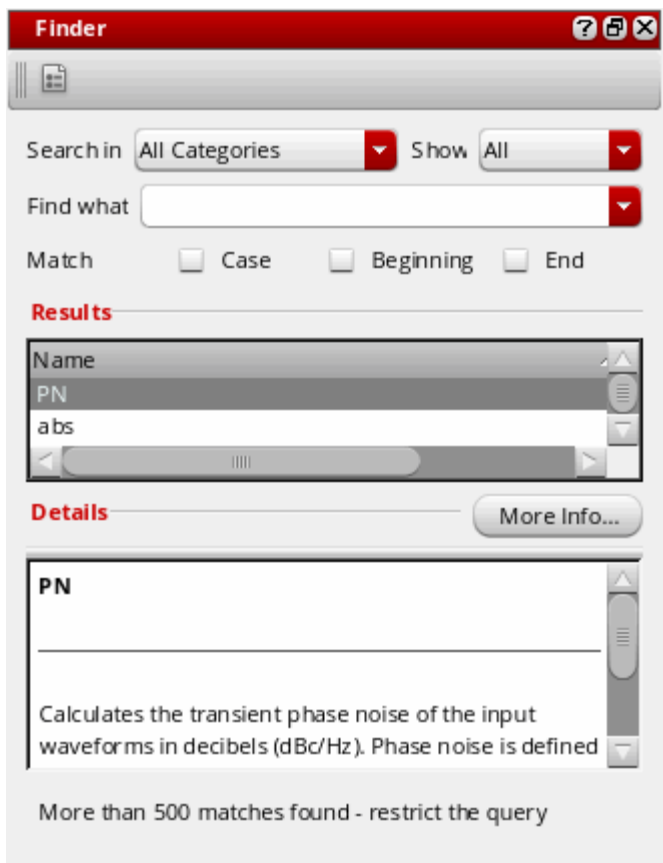
2. Click (*Finder Manager*) in the *Finder* assistant toolbar. The *Finder Manager* window displays.
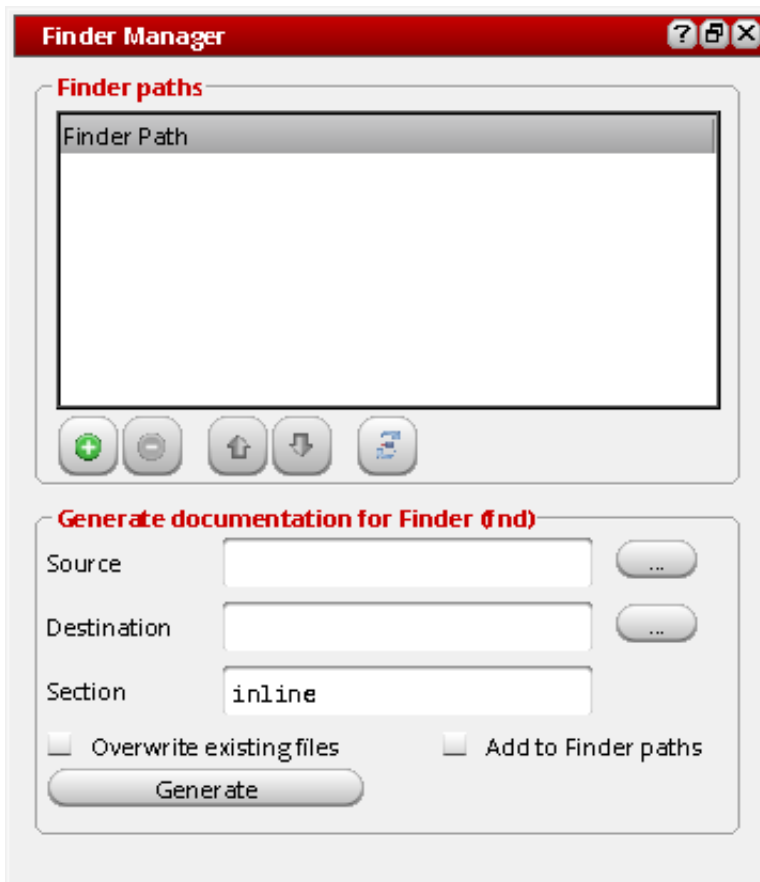
3. In the *Finder paths* section, specify additional directories for Finder database files. When you launch the *Finder* assistant, Finder will look for finder database files (`*.fnd`) under these directories.

❑ Click *Add Finder Path* () to add a new directory path in the *Finder paths* section. A directory path can be added only once.

❑ Click *Remove selected path from the path list* () to remove the selected directory path from the *Finder paths* section.

❑ Click *Move path up in the list* () or *Move path down in the list* () to move a directory path up or down the list. Moving a directory path up or down the list changes the order in which Finder will search the finder database files.

If a directory path listed in the *Finder paths* section is long and appears truncated, you can hover the mouse pointer over the path to view the full path in a tooltip.

The finder database reloads every time you add or remove a directory path, or move it up or down the list.

4. In the *Generate documentation for Finder (fnd)* section, specify the required source and destination information:

    a. In the *Source* field, browse or type the name of the directory that has the source files containing inline documentation strings.

    Finder Manager looks only for files with the extension `.il`/`.ils`. If the specified directory path does not contain any `.il`/`.ils` file, the documentation will not be generated.

    b. In the *Destination* field, specify the directory path under which you want to save the generated finder (`.fnd`) file.

    c. In the *Section* field, specify a section name for the generated finder (`.fnd`) file. The default name is `inline`, which means the finder (`.fnd`) file will be created under a directory named `inline` under your destination directory.

    Your finder (`.fnd`) file will be created two directory levels below the path you specify in the *Destination* field. For example, if your source file is named `inline.il` and you specify the following information in the *Destination* and *Section* fields:

    Destination: `/home/user/skillide`

    Section: `inline_doc`

    A file named `inline.fnd` will be created under:

    `/home/user/skillide/SKILL/inline_doc/`

5. Select the *Overwrite existing files* check box to overwrite any existing finder (`.fnd`) file in the destination path.

6. Select the *Add to Finder paths* check box to add the path of the generated finder (`.fnd`) file in the `$CDS_FINDER_PATH` environment variable. Finder loads its data from the paths specified in this environment variable.

7. Click *Generate* to generate the finder (`.fnd`) file. The *Generating fnd files* progress box displays.
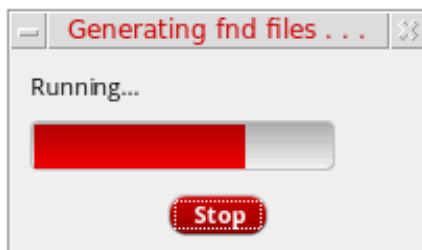
After the file generation process completes, you can check the generated finder (.fnd) file under the directory path specified in the *Generate documentation for finder (fnd)* section.

You can also use a corresponding SKILL script, `genFndFiles.il`, which is located at `tools/dfII/samples/skill/` in your Virtuoso installation directory.

The syntax to run it is as follows:

```
genFndFiles.il -d directory_with_SKILL_files -o output_directory -v log_file
```

For example:

```
genFndFiles.il -d .fnd/source -o .fnd/output -v .fnd/log.txt
```

Related Topics

SKILL IDE Document Generation

Documentation-specific Code

Output Formatting

Viewing the Generated Documentation in Finder

Specifying Additional Finder Data Directories

Viewing the Generated Documentation in Finder

After you have generated the finder (.fnd) file, you can view its contents using the *Cadence SKILL API Finder* tool or the SKILL IDE *Finder* assistant. If you have selected the *Add to Finder paths* check box while generating the documentation, the contents of your finder file will automatically be imported into the finder database.

To view the generated documentation in Finder:

1. Choose *Window – Assistants – Finder* to start the *Finder* assistant. Alternatively, choose *Tools – SKILL API Finder* to start *Cadence SKILL API Finder* from the CIW.

   The *Cadence SKILL API Finder* window displays. Finder loads data from the finder database files.

2. Because the contents of your finder file are automatically imported into the finder database, the section name you specify in the *Section* field of the *Finder Manager* window appear in the *Search in* drop-down list.
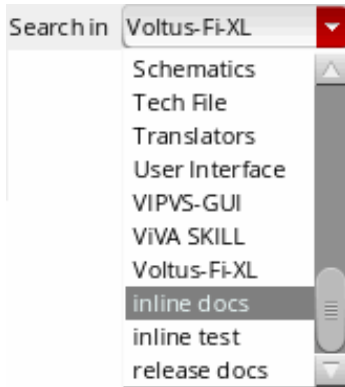
3. Select the appropriate category name from the *Search in* drop-down list. The data in your finder (`.fnd`) file appears in the *Results* area.
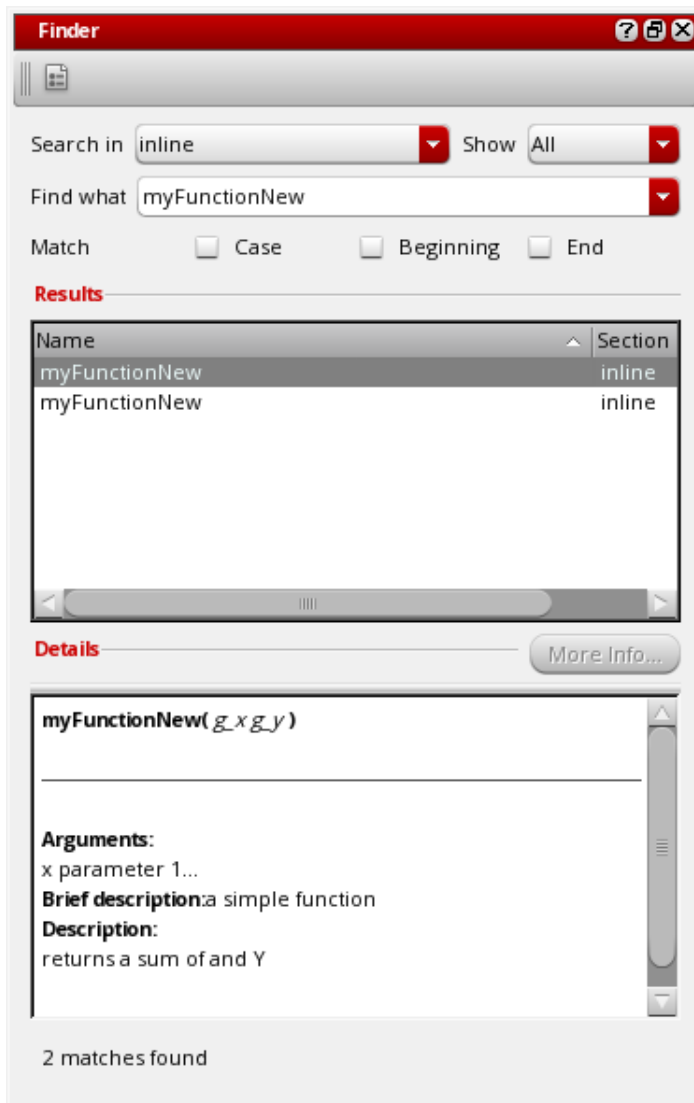
Related Topics

SKILL IDE Document Generation

Documentation-specific Code

Extracting Documentation using Finder Manager

Output Formatting

G

SKILL Integrated Development Environment Forms

This section lists the forms available in SKILL Integrated Development Environment.

SKILL IDE Options Form

The SKILL IDE Options form lets you specify the options to customize the behavior of SKILL IDE elements. The form contains the following tabs.

Tab	Description
SKILL Status	Lets you customize the status settings for the SKILL debugger, compiler, parser, printer, stack, and trace
Editor	Lets you customize the settings for the SKILL IDE editor.
Lint	Lets you set the Lint parameter values before running the Lint tool.
Profiler	Lets you define the type of data you want the Profiler to collect and display.
Color	Lets you customize the color settings for the SKILL IDE editor window.

SKILL Status

The following table describes the fields available on the *SKILL Status* tab of the SKILL IDE Options form.

Field	Description
Trace	This section provides options to specify the tracing options for the debugger.
traceArgs	Saves the evaluated arguments of function calls, which can then be displayed in the stacktrace
traceIndent	Sets the trace indentation format to numbers or '|'.
tracelength	Sets the length of the trace output
tracelevel	Sets the depth of the nested trace printing.
Inline Documentation	This section provides options to associate inline documentation strings with SKILL functions at compile time

Field	Description
save inline documentation strings	Saves the inline documentation strings in your source file. This option also preserves any existing newline characters '\n' in the inline documentation strings. This makes the documentation strings accessible through the `fdoc` API. You can also set this option by setting the `saveInlineDoc` switch to `t` in the `sstatus` SKILL API.
Print	This section provides options to set the printing options for the output.
printinfix	Prints the arithmetic expressions and function calls in infix notation.
echoInput	Prints all input to a log file.
fullPrecision	Prints floating point numbers in full precision (usually 16 digits); otherwise, prints using 7 digits precision.
pprintresult	Prints lists with alignment and indentation (for example, print 5 elements per line).
printpretty	Calls `printself` method when printing standard objects.
printlength	Sets the number of elements that can be printed in a list.
printlevel	Sets the number of print levels.
Stack Trace	This section provides options to set stack trace.
errsetTrace	Prints the errors and stacktrace information that is normally suppressed by `errset`.
stacktraceDump	Prints the local variables when an error occurs.
stacktrace	Prints the stack frames every time an error occurs. You can specify the depth of the stack that needs to be printed when an error occurs.
stackTraceFormat	Sets one of the following options for the level of detail of the stack frame. ■ *fullStack:* Prints the complete set of SKILL stack frames. ■ *onlyCalls*: Suppresses the printing of non-function frames in the output. ■ *onlyTop*: Suppresses the printing of non-function frames except for the top most function frame.
warningTrace	Prints the SKILL stack when a SKILL warning is issued.
warningBreak	Enables SKILL execution to break when a SKILL warning is issued.

Field	Description
Debug	This section provides options to set the SKILL debugger.
debugMode	Enables debug functions and allows you to redefine write-protected SKILL functions.
keepSrcInfo	Saves additional information for debugging. The source information (file/line information) is added to `funobject` during compilation.
autoReload	Debugger auto-reloads a file that is not loaded under `debugMode` when the user tries to single-step into the code defined by that file. It works only for Cadence context source files.
showStepResult	Displays expression evaluation results. If *showStepResult* is enabled, the step results are printed in the CIW every time you click *Step* or *Next* ..
debugMacro	Sets the lineNumber on the expanded macro code to the lineNumber of the original form.
verboseLoad	Displays the full path of the loaded file.
Parser	This section provides options to set the SKILL parser.
multilineString	Allows SKILL strings inside double quotes to be spanned on several lines.
keepNLInString	Saves the newline characters '\n' in strings.
classAuxAutoLoad	Allows class search in `.aux` files.
mergemode	Merges arithmetic expressions to minimize the number of function calls.
Compiler	This section provides options to set the SKILL compiler.
lazyComp	Disables compiling of top-level functions when they are defined. Instead, compiles them when they are called.
optimizeNestedLet	Instructs the SKILL compiler to parse the code for `let` constructions and expand/remove them by moving their local variables to the top-level function's local variables section. It reduces the multiple nested `let` declarations.
optimizeTailCall	Sets the tail call recursion on, which prevents runtime stack overflow when a function is called recursively. Meant only for Scheme functions.
writeProtect	Sets write-protection on. When functions being defined have write-protection enabled, they cannot be redefined.

Editor

The options available on the *Editor* tab of the SKILL IDE Options form lets you customize the settings for the SKILL IDE editor. The changes that you make in the editor options are persistent across SKILL IDE sessions.

Field	Description
Show Line Number	Displays line numbers to the left of the source code pane against each line of code.
Show Tooltips When Hover Over Name	Displays tooltips when you hover the mouse pointer over any object, such as a function, class, method, or variable. You can also press the F3 key to select or deselect this check box.
Highlight Matching Text:	Highlights all occurrences of a search string in cyan.
Enable Name Completion	Enables auto-completion of keywords or function names based on your keystrokes. You can also press the F4 key to select or deselect this check box.
Auto Indent:	Automatically indents each new line in the editor to the left or right by a predefined number of spaces.
Emulate Tabs	Uses spaces to emulate tabs, so that a fixed set of spaces (equal to the tab size) is inserted when the tab key is pressed.
Tab Size	Sets the number of spaces to be inserted for each tab character.
Font Size	Sets the font size of the code in the editor window.
Font Style	Sets the font style of the code in the editor window. The available options are *Normal* and *Bold*.

Lint

The options available on the *Lint* tab of the SKILL IDE Options form lets you set the Lint parameter values before running the Lint tool.

Field	Description
Package Prefixes	List of acceptable package prefixes for functions and global variables (for example, `tr`). Environment variable: prefixes

Field	Description
Global Variables	Specifies a list of global variables in the code that SKILL Lint should check.
	Environment variable: globals
Code Version	Lets you select the release version of the code you want SKILL Lint to check.
	Environment variable: codeVersion
External Files	
Output	Select one or both of the check boxes to specify where you want to direct the SKILL Lint output.
Print To Log	Lets SKILL Lint to send its output to the CDS.log file and the Command Interpreter Window (CIW).
	Environment variable: printToLog
Report File	Lets SKILL Lint to display the output in a separate Lint output window.
	Environment variable: reportFile
Report File Name	Specifies the name of the file to which you want to write the SKILL Lint output report. The *Report File Name* field is enabled only when you select the *Report File* check box.
	Environment variable: reportFileName
Rules	Applies rules to the SKILL Lint report.
Preset	Applies system-defined rules to the SKILL Lint report.
Custom	Applies user-defined rules to the SKILL Lint report.
Errors	Enables `error`, `error global`, and `fatal error` message groups. If selected, SKILL Lint reports code that might result in any of these errors when the code is run.
	Environment variable: error
Warnings	Enables `warning` and `warning global` message groups. If selected, SKILL Lint reports potential errors and areas where you might need to clean up your code.
	Environment variable: warning

Field	Description
Information	Enables `information` message groups.
	Environment variable: <u>information</u>
Undefined Functions	Enables message groups having undefined functions that cannot be run in the executable which started SKILL Lint.
Performance	Enables the `hint` and `suggestion` message group. When selected, SKILL Lint provides hints or suggestions to improve potential performance problems in your SKILL code.
Cadence Functions	Includes Cadence public functions or variables while checking prefixes. If you do not select this check box, SKILL Lint limits prefix checking to your custom functions and variables.
	Cadence public functions should start with a lowercase character; your custom functions should start with an uppercase character.
	The *Cadence Functions* check box turns the `ALERT` message group in the *Custom* section on or off. If turned on, SKILL Lint reports code that might result in an alert when the code is run.

Profiler

The options available on the *Profiler* tab of the SKILL IDE Options form lets you set the Lint parameter values before running the Lint tool.

Field	Description
Enable Memory Usage Mode	Specifies a list of acceptable package prefixes for functions and global variables (for example, `tr`).
	Environment variable: <u>prefixes</u>
Run Time	Runs the profiler in time mode with the time being measured as CPU time.
Real Time	Runs the profiler in time mode with the time being measured as clock time.
Memory Usage	Runs the profiler in memory mode.
Include Function Call Counts	The number of times a function is called in the SKILL program.

Field	Description
Case Sensitive Search	Matches the case of the searched function in the profiler summary report.
Display	Applies rules to the SKILL Lint report.
Maximum Functions	Specifies the maximum number of functions to be seen. This field is disabled when you select the *Tree View* tab in the Profiler assistant pane.
Time Longer Than (seconds)	Specifies the minimum number of seconds that should be spent in a function for its information to appear in the summary. This field is displayed when you run the profiler in *time mode*.
Functions	Lets you select one of the following options: ■ *All* to display all functions in the profiler window. ■ *In context* to display only the functions in the given context. ■ *Match Expression* to display only those functions that match the given regular expression. ■ *Match Prefix* to display only those functions who prefix match the specified. ■ *User* to display user functions, that is, functions that are neither binary nor write-protected. ■ *System* to display non-user binary SKILL functions.

Color

The options available on the *Color* tab of the SKILL IDE Options form lets you customize the color settings for the SKILL IDE editor window..

Field	Description
Set Color	Lets you change the color preferences for the following code elements: ■ *Foreground*: Changes the foreground color of the text. ■ *Background*: Changes the background color of the editor window. ■ *Selected Text*: Changes the highlight color of the text selection. ■ *Matching Text*: Changes the highlight color of the text matching the selected text. ■ *Matching Parenthesis*: Changes the highlight color for the matching brackets. ■ *Mismatching Parenthesis*: Changes the highlight color of the mismatched bracket. ■ *Error*: Changes the highlight color for the errors in the editor window. ■ *Step*: Changes the highlight color for *Step* and *Next* commands. ■ *Cross-Highlight*: Changes the cross highlight color of code in the editor window. ■ *Keyword*: Changes the color for SKILL language keywords. ■ *Comment*: Changes the color of comments in the SKILL code. ■ *Number*: Changes the color of numeric values in the SKILL code. ■ *String*: Changes the color of strings in the SKILL code ■ *Highlight1*, *Highlight2*, *Highlight3*, *Highlight4*, *Highlight5*: Changes the custom highlight colors.
Color Swatch	Lets you choose and change the color of the selected element.
Custom Colors	Displays custom colors that have been created.
Add to Custom Colors	Lets you save the new color as a custom color.

Related Topics

Setting SKILL IDE Options

Customizing the Color Settings

Setting Lint Options

Setting Profiler Options