SPARCworks/Ada Tutorial

SunSoft

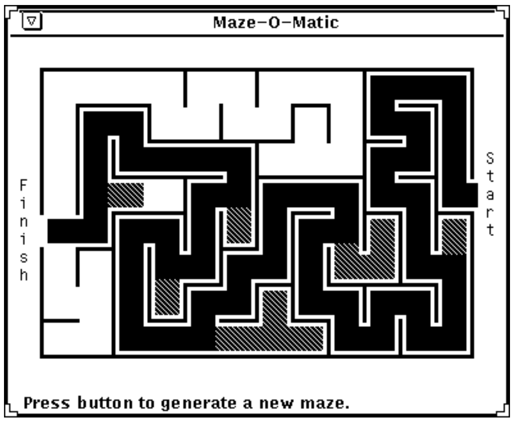
A Sun Microsystems, Inc. Business

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: 801-6174-11
Revision B, August 1994

Contents

Preface



SPARCworks/Ada Tutorial

This tutorial demonstrates the use of SPARCworks/Ada features and facilities as they might be used when working with an actual program. The SPARCompiler Ada demonstration program is called `solve`.

In this tutorial, you follow the bug-fixing efforts of a fictional SPARCworks/Ada developer, named Chris Holmes, who uses the tools to examine and fix a version of the `solve` program that does not run correctly.

Where `solve` Is Located

The `solve` program has two versions. One runs correctly; the other contains a bug. Both versions install automatically when you install SPARCworks/Ada on your system. The two versions of the `solve` program are located in the following two subdirectories, where *swada_location* is a variable name that depends on where you install SPARCworks/Ada in your file system:

swada_location`/examples/maze`
swada_location`/examples/broken_maze`

Chris Holmes, the Fictional Developer

Much of this tutorial is written in the third person, using a fictitious developer, for several reasons. First, it makes it easier to separate the sequence of actions you might use to fix the demo program from general statements about how SPARCworks/Ada works. General statements and pointers about how to use each tool are placed between thick bars. This way, you can choose to follow the tutorial steps without interruptions or you can read the extra material.

Also, it is obvious that developers might use many strategies and tactics to solve the kinds of problems this program has. The SPARCworks/Ada tools themselves provide multiple ways of doing things. Chris Holmes's approach in this tutorial is just one of many ways of going about it.

To make it easier to follow Chris's steps, the key steps Chris takes are numbered and placed to the left of a glyph, like this:

❖ Type `adavision &`

If you prefer to follow just the debugging steps, see Appendix A. If you perform the same operations as Chris does in this tutorial, you should reach the same conclusions. This tutorial does not, however, attempt to tell you how to organize your windows. That's up to you.

When a window is displayed, clicking SELECT on the pushpin in the title bar forces the pushpin to hold the window open. When the pushpin is removed, it closes automatically.

When you click SELECT on any AdaVision object, you *select* it. For objects represented with icons, AdaVision draws a box around the selected object(s). A *selected object* becomes the *target* of whichever item you then choose from a menu.

Though we have tried to introduce a bug that makes some sense, this tutorial is contrived primarily to exhibit SPARCworks/Ada features and facilities. Chris does a few things you might not ordinarily do, to illustrate some things about SPARCworks/Ada. Chris does *not* perform some actions for the same reason.

Sometimes Chris may appear to be very clever, at other times, an amateur. These differences are fictional. Any resemblances to real-life characters or situations are unintentional.

Note – In this tutorial, Chris uses local execution.

How This Tutorial Is Organized

This tutorial consists of two chapters:

Appendix A, "List of Steps for AdaDebug Tutorial," is a step-by-step summary of the AdaDebug tutorial so that you can restart it and advance quickly to some particular step in that part of the tutorial.

Solving the Maze (Part 1): AdaVision 1

Chris is assigned to take over the broken version of the maze from its original author, who has left the company. Chris is a veteran Ada programmer, but is fairly new to SPARCworks/Ada. The `solve` program is also something of a mystery. It is a short demo program, with only a few library units containing a small amount of C code, which must be compiled separately and linked with Ada. Chris also knows that after installing a new compiler upgrade the program must then be recompiled and relinked.

1.1 Listing Objectives and Tasks

Assume that you are Chris; a list of your objectives and the tasks required to accomplish each of them might read like the following:

1. Copy the files in the `broken_maze` directory to your own workspace, renaming it `my_broken_maze` to avoid confusion with the original.
2. Build the SPARCompiler Ada library `my_broken_maze`.
   a. Compile the C code in the `my_broken_maze` directory.
   b. Start AdaVision in the `my_broken_maze` directory.
   c. Use the New Library facility to create the new SPARCompiler Ada library.
   d. Set link directive to connect the compiled C code.
   e. Create a new SPARCompiler Ada library named `my_broken_maze`.

3. Run `solve` to see what the problem looks like.
4. Examine the structure of `solve` in AdaVision.
5. Start AdaDebug and try to fix the program.

1.2 Copying `broken_maze`

The first thing to do is copy the `broken_maze` files to your working directory.

`Broken_maze` is write-protected. Before you start the tutorial, copy the complete contents of the *swada_location*`/examples/broken_maze` directory to somewhere else in your file system. In this tutorial, Chris copies the files to a directory called `my_broken_maze`.

---

1. ❖ Chris uses the following commands to copy all of the `maze` tutorial files. (The directory contains some `.`*xxxx* files, so a simple `cp *` does not copy all of them. Add the option `-r`, as follows.)

```
% cd $SWADAHOME/examples/broken_maze
% cp -r . destination_directory/my_broken_maze
```

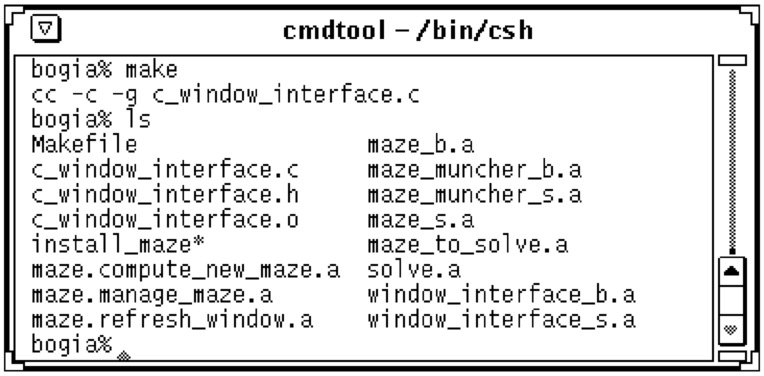
where *destination_directory* is the location in your workspace where you want to copy the `maze` tutorial files.

1.3 Building `my_broken_maze`

Chris changes the directory to where the copy of `broken_maze` resides, then lists the contents of the directory as follows:

2. ❖ `hostname% cd /`*destination_directory*`/my_broken_maze`
3. ❖ `hostname% ls`

The following figure shows the commands as entered and the system output following the `ls` command.



```
cmdtool - /bin/csh
bogia% make
cc -c -g c_window_interface.c
bogia% ls
Makefile                    maze_b.a
c_window_interface.c        maze_muncher_b.a
c_window_interface.h        maze_muncher_s.a
c_window_interface.o        maze_s.a
install_maze*               maze_to_solve.a
maze.compute_new_maze.a     solve.a
maze.manage_maze.a          window_interface_b.a
maze.refresh_window.a       window_interface_s.a
bogia%
```

At the top of the list of files, Chris notes a Makefile, a C language source header file, object file, an install script, and 10 Ada files. It looks as if a former coworker has already removed the old `ada.lib` file and other Ada library files.

If the old Ada library had been present, Chris could have removed the `ada.lib` file and all of the old compilation information from the library from within AdaVision, choosing Cleanlib from the Edit menu in AdaVision library mode.

Chris decides to compile the C code. The former colleague included a Makefile for this step.

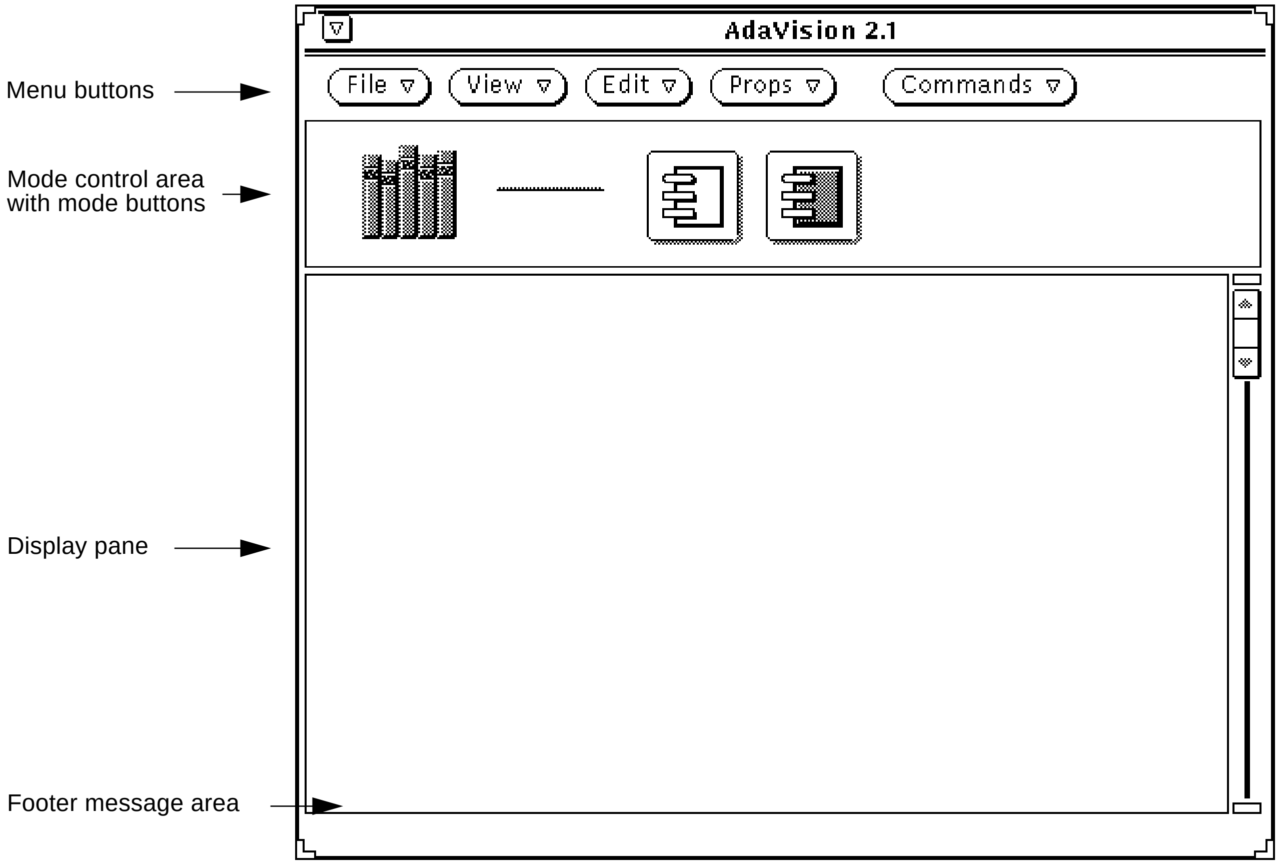
4. ❖ `hostname%` **`make`**

1.3.1 AdaVision Startup

5. ❖ Chris starts AdaVision by typing:

    `adavision &`

at a Command Tool prompt. In a few seconds, the AdaVision main window appears on the screen.



Chris sees an AdaVision main window featuring a row of menu buttons and, beneath them, three large icons. These "icons" are really *mode control buttons.* They control the three modes in AdaVision: *library* mode, *library unit* (LU) mode, and *compilation unit* (CU) mode. Each mode offers the features appropriate to the type of objects you find at each level: libraries, LUs, and CUs.

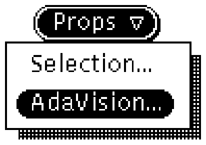
The first of the three mode buttons—the one that looks like a set of library books—is dark. This is the library mode button. AdaVision always starts up in library mode and highlights the button corresponding to the mode you are in.

Although the example window shows some libraries loaded, Chris sees a large, empty display pane beneath the mode buttons. No libraries are listed, because, although AdaVision was opened in the `my_broken_maze` directory, this directory is not yet an Ada library directory. (It has no `ada.lib` file, for starters.)

In just a minute, Chris will choose New Library from the File menu to create a new Ada library and import its name to the library mode list, along with the names of the libraries on its ADAPATH.

1.3.2 AdaVision Properties

Before creating the new library, Chris checks to see if the AdaVision properties are set the way they should be for compiling.



To set AdaVision properties, open the AdaVision Properties window from the Props menu button:

6. ❖ Choose AdaVision from the Props menu.

The first item on the menu, Selection, is inactive. AdaVision dims (grays out) menu items that you can't use in any given circumstance. In this case, the Selection item on the Props menu refers to the Properties window for the selected library. Because no library is selected, the item is inactive.

All SPARCworks/Ada tools have default items for menu buttons. To see what the defaults are, press SELECT over each menu in succession; the default item is outlined. (Move the pointer from the menu button before releasing the mouse button to avoid actually choosing the item.)

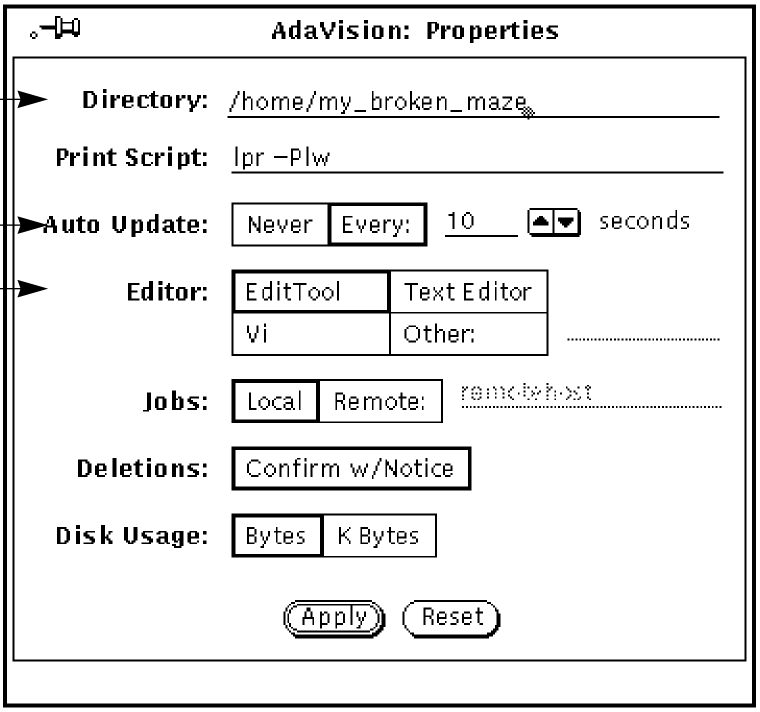
When Chris chooses AdaVision, the AdaVision Properties window pops up. Many of the AdaVision features and facilities are handled through pop-up windows.

Three dots trailing a menu button or item is the OPEN LOOK convention for signaling that the button or item opens a pop-up window.

AdaVision’s location in your file system. You can move it elsewhere with this control.

AutoRead control for updating the display pane as the library changes.

Choice of editors. EditTool includes compiler error-reporting and positioning features.



Directory: /home/my_broken_maze

Print Script: lpr -Plw

Auto Update: Every: 10 seconds

Editor: EditTool

Jobs: Local

Deletions: Confirm w/Notice

Disk Usage: Bytes

Chris accepts most of the control settings as they are:

- AdaVision’s location

  AdaVision was started in the my_`broken_maze` directory but it could be placed elsewhere in this file system with the Directory control.

- The Auto Update setting

  Auto Update controls the AutoRead facility, which updates the AdaVision display pane at regular intervals (here, every 10 seconds, which is the default).

- The Deletions control setting

  The Deletions control is set to Confirm w/Notice, which means you cannot delete files or directories without AdaVision’s query.

- Disk Usage control

  Chris leaves the Disk Usage control set to Bytes (the default). Sometimes it's wise to enter the exact figures, to be sure that the unit is still intact and is the correct one to work with.

- EditTool settings

  Normally, Chris works in `vi`. But in order to compile someone else's code, the error-reporting and cursor-positioning features of EditTool might be handy. So Chris accepts the Editor control as set to the default AdaVision editor, EditTool. EditTool is the SPARCworks/Ada enhanced version of the OpenWindows Text Editor.

  When Chris opens a unit for editing, AdaVision starts an EditTool session in a separate window and loads the file containing the selected unit.

1.3.2.1 EditTool

You can change the default AdaVision editor by means of the Editor control. Note, however, that EditTool supports a special compiler-error correction feature. When a Compile, Make, or Link job fails, AdaVision job status windows contain an EditTool button that starts EditTool and positions the cursor at or near the line where the compiler reported finding an error. Also, this EditTool window has a special Error menu button that lets you cycle through errors.

1.3.2.2 Control Option Settings

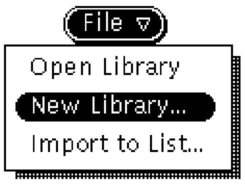
The options for the Editor control are all in boxes that touch each other. These are called exclusive settings; only one of the choices can be set at a time. Other AdaVision controls present options in boxes that don't touch. These are called nonexclusive settings; you can set as many of them as you want.

You can wait to press the Apply button until you have made all of the changes within a control window that you want to make. The Reset button returns the settings to how they were after the last time you pressed Apply. If you close the window without pressing Apply, the settings automatically revert to how they were when you opened the window, just as if you had pressed Reset.

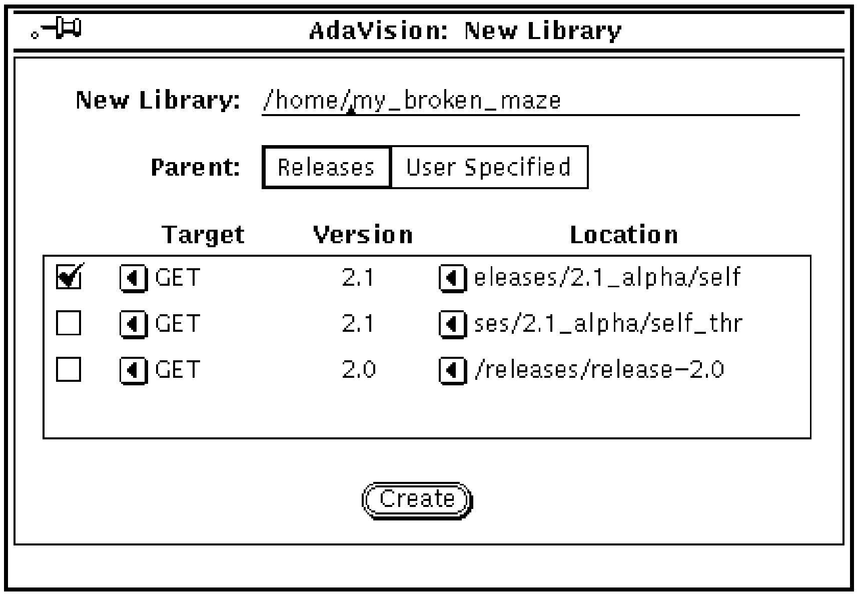
1.3.3 New Library Creation

Chris's first substantive task in AdaVision is to create a new library for the `broken_maze` program.

To create a new Ada library in the `broken_maze` directory:



7. ❖ Choose New Library from the File menu (library mode) to open the New Library pop-up window.


AdaVision: New Library

New Library: /home/my_broken_maze

Parent: [✓ Releases] User Specified

	Target	Version	Location
✓	GET	2.1	eleases/2.1_alpha/self
☐	GET	2.1	ses/2.1_alpha/self_thr
☐	GET	2.0	/releases/release-2.0

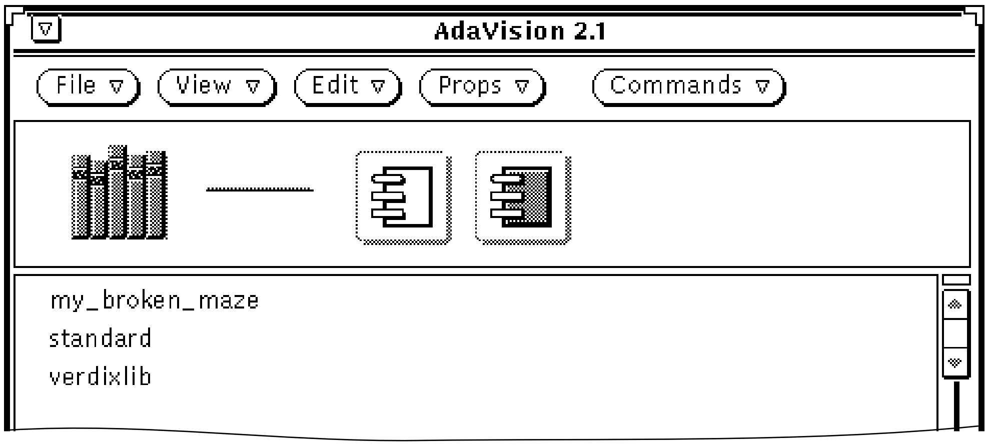
AdaVision enters the name of the directory as a default Ada library name. Chris could edit this text field to supply a different name, but normally prefers the Ada library and directory names to be the same.

Note – The target release(s) as shown on the display pane of this window depend on the entry(ies) in your `/etc/VADS` file. Parent marks the release pointed to by the first line in `/etc/VADS` and designates it the default library.

8. ❖ The New Library window has a control for setting the new library *parent* library. The default setting is Releases. By default, AdaVision sets the parent library control to the `standard` and `verdixlib` libraries in the latest version of SPARCompiler Ada installed on the system. A check mark in the box next to the first entry in the list of installed releases of SPARCompiler Ada, indicates that it is the default, so:
Press the Create button to create a new SPARCompiler Ada library named `my_broken_maze`, with a parent library named:

/*sunada_location*/`self/standard`

Almost immediately, the three library names appear in the AdaVision display pane: `my_broken_maze`, `standard`, and `verdixlib` (also required by all SPARCompiler Ada programs).

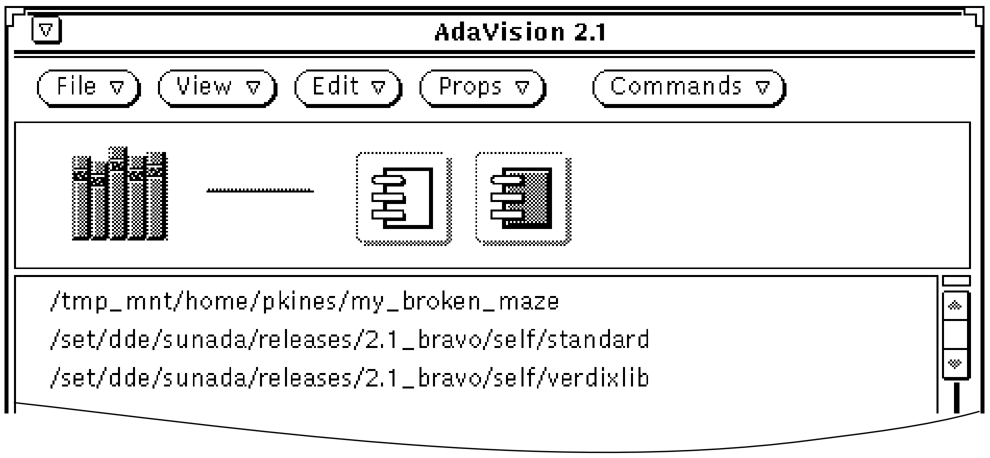


```
my_broken_maze
standard
verdixlib
```

When you create a new library, AdaVision displays the new library plus all of the libraries on its ADAPATH.

By default, AdaVision lists the *simple* Ada library name only. Because viewing full path names is useful:

9. ❖ Choose Full Pathnames from the View menu.



```
/tmp_mnt/home/pkines/my_broken_maze
/set/dde/sunada/releases/2.1_bravo/self/standard
/set/dde/sunada/releases/2.1_bravo/self/verdixlib
```

1.3.4 Unit Import into AdaVision

Chris is now ready to *import* the `my_broken_maze` LUs into AdaVision.

AdaVision is primarily an Ada unit-based system rather than a file-based system. When you import units, you instruct AdaVision to compile a list of Ada source files and display an object for each of the resulting units in the LU mode display pane. The unit objects are named and graphically-coded icons.

You import units into AdaVision from LU mode. AdaVision does not allow you to switch to either unit mode unless you first select a library.

To switch to LU mode from library mode:

10.❖ SELECT the path name for `my_broken_maze` from the list, then:

11.❖ Press the LU mode button.

Because Chris has not yet imported any units, the LU mode display pane is empty. Chris notices that the LU mode control button (the middle one) is now highlighted, and that the library mode button, now un-highlighted, shows the first part of the name of the selected library.

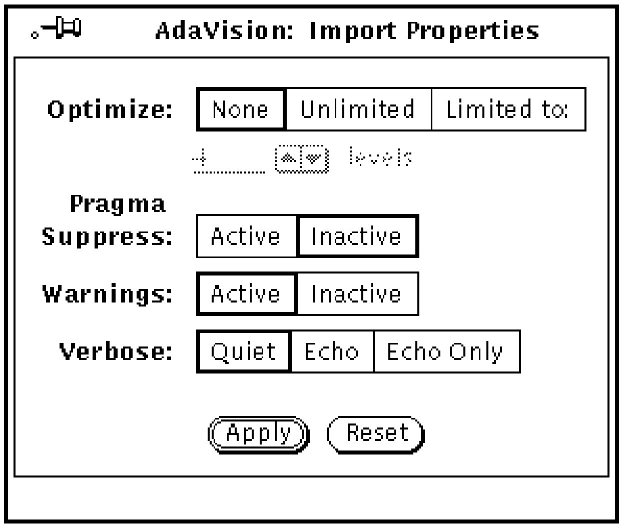
1.3.4.1 Mode-Sensitive AdaVision Menus

Notice that the items on the File menu in LU mode are different from the items on the File menu in library mode. In AdaVision, features and functionality are mode-sensitive. (As mentioned, SPARCworks/Ada menu items and controls are context-sensitive, that is, the tools dim the items on menus and options that are not applicable in a specific situation.)

1.3.4.2 Level of Optimization

Chris wants AdaVision to import the new units with all code optimization turned off. The default setting is level 4, so the setting must be changed before starting the import. These are the steps for an import:

12.❖ Choose Import from the Props menu to open the Import Properties window.


AdaVision: Import Properties
Optimize: [✓ None] Unlimited Limited to:
4 levels
Pragma Suppress: Active [✓ Inactive]
Warnings: [✓ Active] Inactive
Verbose: [✓ Quiet] Echo Echo Only
Apply
Reset

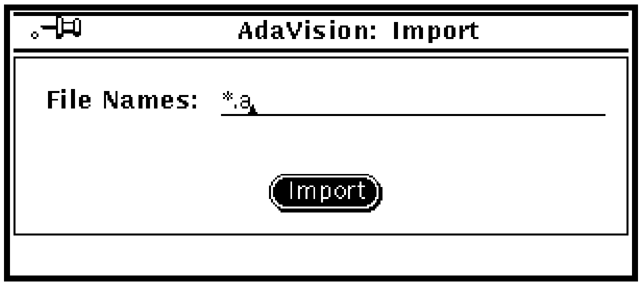
13.❖ Set the Optimize control to None (by clicking SELECT in the box labeled None); then press the Apply button.

1.3.4.3 Unit Import

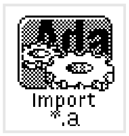
To import the LUs into AdaVision:

14.❖ Choose Import from the File menu to open the Import pop-up window.

15.❖ Type `*.a` in the File Names text field and press the Import button.


AdaVision: Import

File Names: *.a

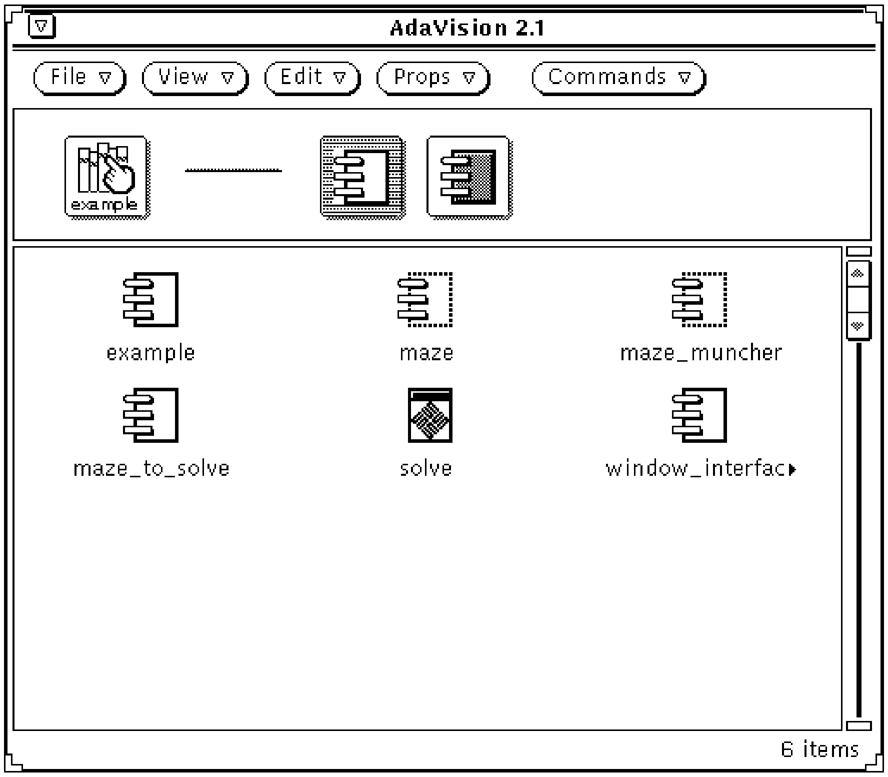


Import Status window icon

When Chris presses Import, AdaVision starts an import job in the background, which is handy for large libraries. AdaVision displays a gray icon (as shown in the margin) for an Import Status window. This window reports the outcome of the import job. If the job is successful, the icon becomes clear. You don't have to open the window.

Since `my_broken_maze` is a small library of only five LUs, the import should not take long. Chris watches for the gauge in the icon to turn clear, which will indicate that the import job is complete.

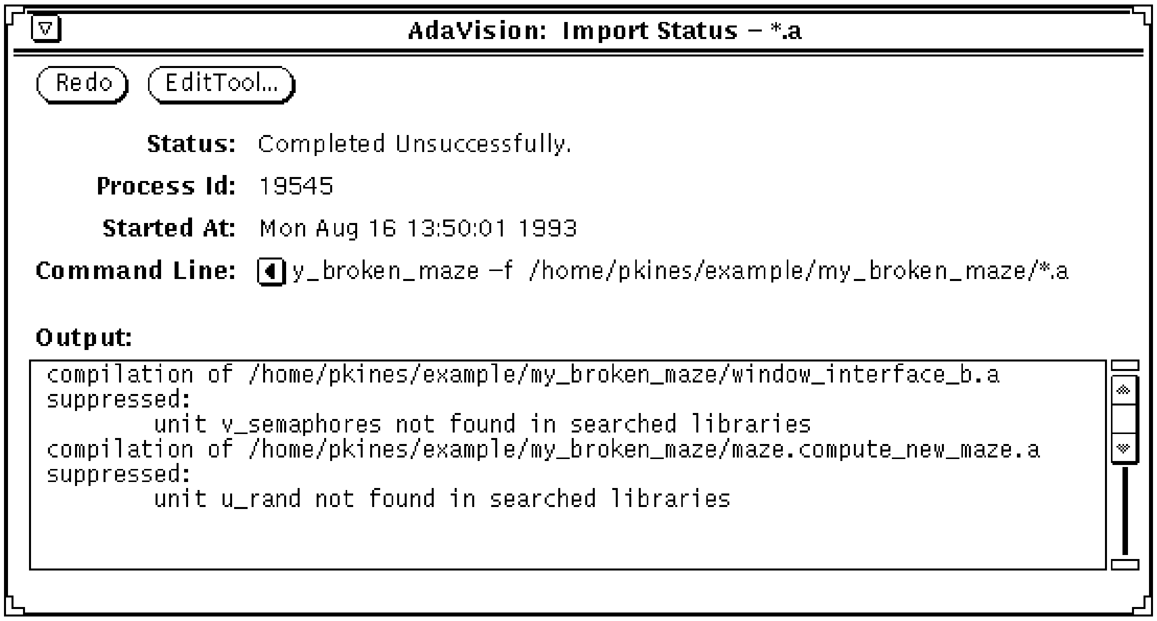
With AutoRead turned on, Chris can watch the display pane fill up with icons representing the compiled LUs in the program: one item appears, then the second through the sixth:



Unexpectedly, instead of completing a satisfactory compile, the gauge in the import Status icon breaks, indicating that the job terminated unsuccessfully. To discover what has happened, follow the steps below:

16.❖ Double-click SELECT on the icon to open the Import Status window.

Read the status and the message in the Output window:



```
Status:        Completed Unsuccessfully.
Process Id:    19545
Started At:    Mon Aug 16 13:50:01 1993
Command Line:  y_broken_maze -f /home/pkines/example/my_broken_maze/*.a

Output:
compilation of /home/pkines/example/my_broken_maze/window_interface_b.a
suppressed:
        unit v_semaphores not found in searched libraries
compilation of /home/pkines/example/my_broken_maze/maze.compute_new_maze.a
suppressed:
        unit u_rand not found in searched libraries
```

Chris reads the message and suspects referenced units in a library that is not on the `my_broken_maze` ADAPATH.

In a haste to proceed, Chris forgot to assign `publiclib` as the parent library when creating the new library. This particular program requires `publiclib` to be the parent of `my_broken_maze`. The `standard` library is too high up on the ADAPATH.

Chris has a choice of starting over or going to the `my_broken_maze` Library Properties window and changing the ADAPATH. Chris decides to change the ADAPATH.

1.3.4.4 Changing an ADAPATH

Changing the ADAPATH is a library-level operation, so AdaVision handles it in library mode.

17.❖ Chris switches back to library mode by pressing the library mode control button. The selected library is still `my_broken_maze`.

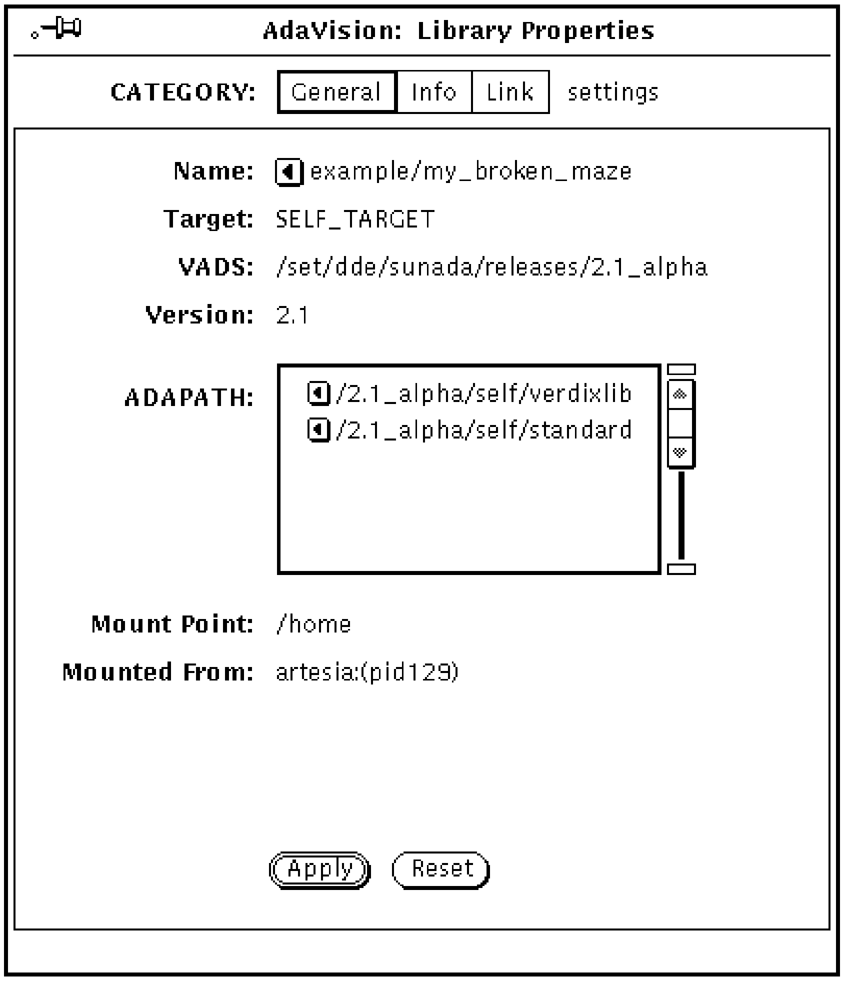
18.❖ Choosing the selection from the Props menu opens the Library Properties pop-up window. (The Selection menu item is now active because a library is selected.)

In addition to changing the ADAPATH, Chris might as well write the link directives for the C module while the Library Properties window is open. The link library directive controls are also on the Library Properties window.

Library property controls are separated into three categories:

- General
- Info
- Link

The ADAPATH control is on the General properties sheet.



AdaVision: Library Properties

CATEGORY: General | Info | Link settings

Name: example/my_broken_maze

Target: SELF_TARGET

VADS: /set/dde/sunada/releases/2.1_alpha

Version: 2.1

ADAPATH:
/2.1_alpha/self/verdixlib
/2.1_alpha/self/standard

Mount Point: /home

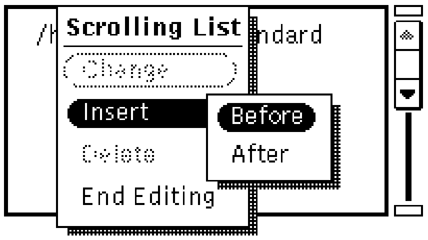
Mounted From: artesia:(pid129)

Chris sees that the ADAPATH control shows the `standard` and `verdixlib` libraries only. To add `vads_exec`, the ADAPATH list needs to be edited.

19.❖ Chris pins the property window and moves the pointer into the scrolling list box, pressing MENU to open the Scrolling List menu.

To insert `vads_exec` higher up on the list:

20.❖ Chris chooses Insert ⟹ Before from the Scrolling List menu.

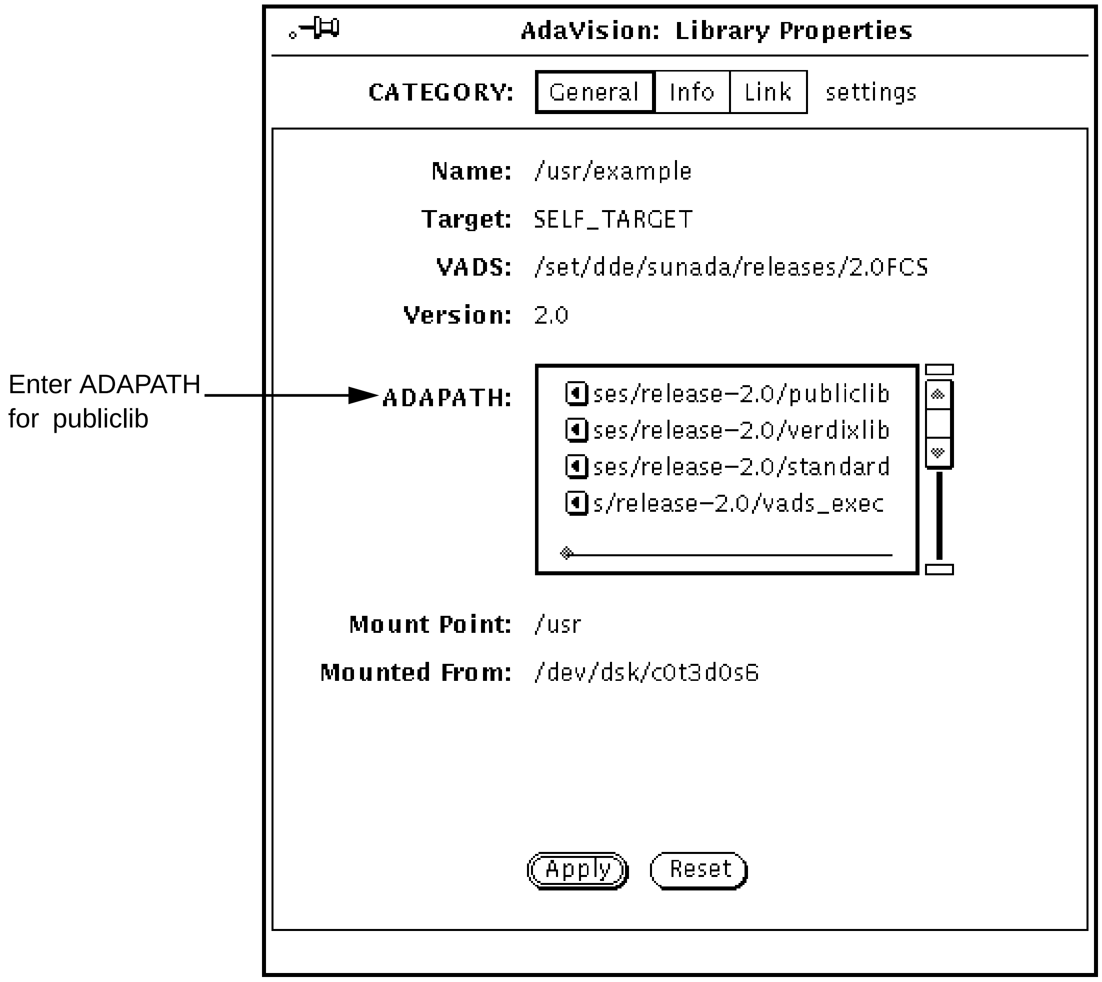


21.❖ A text field line is displayed.

Chris types in the ADAPATH for `publiclib`, where *sunada_location* is the location of SPARCompiler Ada installed on your system:

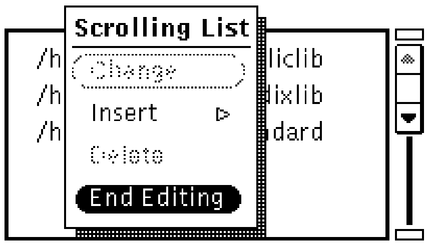
sunada_location/**`publiclib`**

(The value of the `$SUNADAHOME` environment variable is *sunada_location*.)



Name: /usr/example
Target: SELF_TARGET
VADS: /set/dde/sunada/releases/2.0FCS
Version: 2.0
ADAPATH:
ses/release-2.0/publiclib
ses/release-2.0/verdixlib
ses/release-2.0/standard
s/release-2.0/vads_exec
Mount Point: /usr
Mounted From: /dev/dsk/c0t3d0s6

22.❖ Chris opens the Scrolling List menu again and chooses End Editing. (To use an accelerator to exit the editing mode, press Return *twice* after completing an edit.)



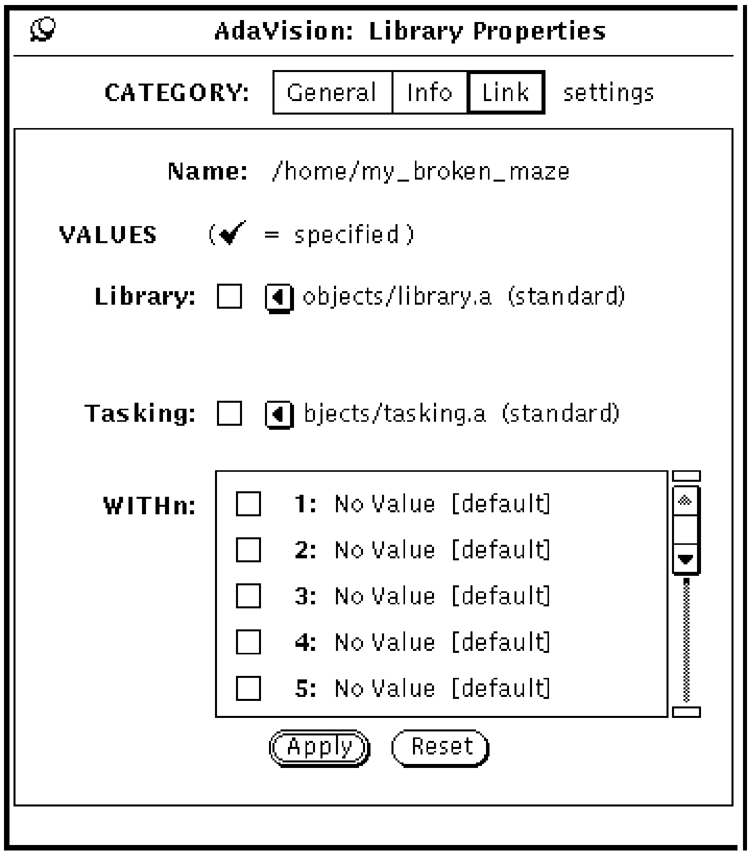
23.❖ Check the new ADAPATH list, then press Apply.

When Chris presses Apply, AdaVision analyzes the changes to the ADAPATH and alters the library list accordingly. In this case, it adds `publiclib` to the library mode list of path names.

1.3.5 LINK Directives

Now Chris is ready to write the link directives for the C module compiled earlier. Still working in the Library Properties window,

24.❖ Chris clicks SELECT on the LINK setting of the CATEGORY control to display the link directive controls.


AdaVision: Library Properties

CATEGORY: General Info Link settings

Name: /home/my_broken_maze

VALUES (✓ = specified)

Library: ☐ objects/library.a (standard)

Tasking: ☐ bjects/tasking.a (standard)

WITHn:
☐ 1: No Value [default]
☐ 2: No Value [default]
☐ 3: No Value [default]
☐ 4: No Value [default]
☐ 5: No Value [default]

Apply Reset

Chris sees that no WITHn directives are set in `my_broken_maze`; that is, none of the boxes is checked.

Also, both the Library and Tasking directives have been set, but not here in the current directory—or the boxes would be checked on this properties sheet. These two directives were set elsewhere on the ADAPATH, in the `standard` library, as indicated in parentheses to the right of each directive.

Chris wants to set link directives in the current library, `my_broken_maze`. These directives tell AdaVisionto link the `c_window_interface.o` file compiled earlier and the X11 library upon which it depends.

To set WITHn link directives in the current library:

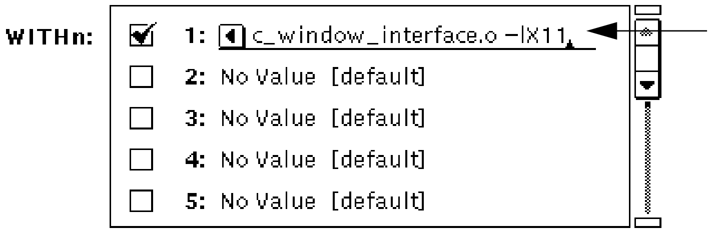
25.❖ Click SELECT in the check box next to WITHn 1.

AdaVision converts the default field, No Value, to a text field.

26.❖ Type in the name of the C object file compiled earlier:

```
c_window_interface.o -lX11
```

Note – In the command above, be sure you type a hyphen followed by a lowercase L and not a one (1).



WITHn:

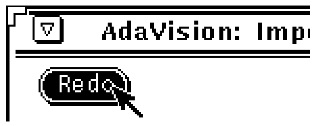
✓	1:	c_window_interface.o -lX11
☐	2:	No Value [default]
☐	3:	No Value [default]
☐	4:	No Value [default]
☐	5:	No Value [default]

27.❖ Chris presses the Apply button.

In the window footer, AdaVision reports that it has successfully updated the link directive for this library.

Now Chris is ready to rerun the Import job in LU mode.

28. ❖ Chris presses the Redo button in the Import Status window, which was left open on the Desktop.

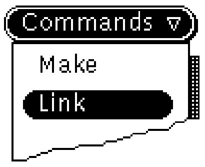


This time, the LU import is successful.

Chris is tempted to start looking at the code in the LUs, but is even more tempted to run the program. All that remains before running `solve` is to link the executable.

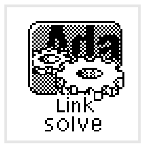
1.3.6 LU Linking

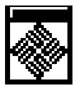
29.❖ Chris switches to LU mode and selects the unit `solve` by clicking SELECT on its icon.



30.❖ Next, Chris chooses Link from the Commands menu.

AdaVision starts a link job in the background and displays a Link Status icon. Chris watches the gauge. The link succeeds and the solve unit now has a Sun logo inside to indicate it is executable.




An executable unit

Chris throws away the Link Status icon by pressing MENU to open its Window menu and choosing Quit.

1.3.7 Program Execution from AdaVision

To start running `solve` from within AdaVision, Chris must

31.❖ *Drag* the `solve` unit icon off the AdaVision display pane and *drop* it onto the workspace background.[1]

The solve program, called Maze-O-Matic for end-users, is displayed on the screen (Figure A below). Chris tries out maze a few times to see how it works.

Chris moves the pointer inside the maze display pane and presses a mouse button, as the message in the maze footer instructs users to do. The first maze completes successfully, as shown in Part B of Figure 1-1 on page 1-24.

1. This interface operation is called *drag-and-drop*: you place the pointer over the `solve` icon, then press SELECT; while still holding down the SELECT button, move the drag-box outline of the icon to anywhere on your Desktop, then release the mouse button

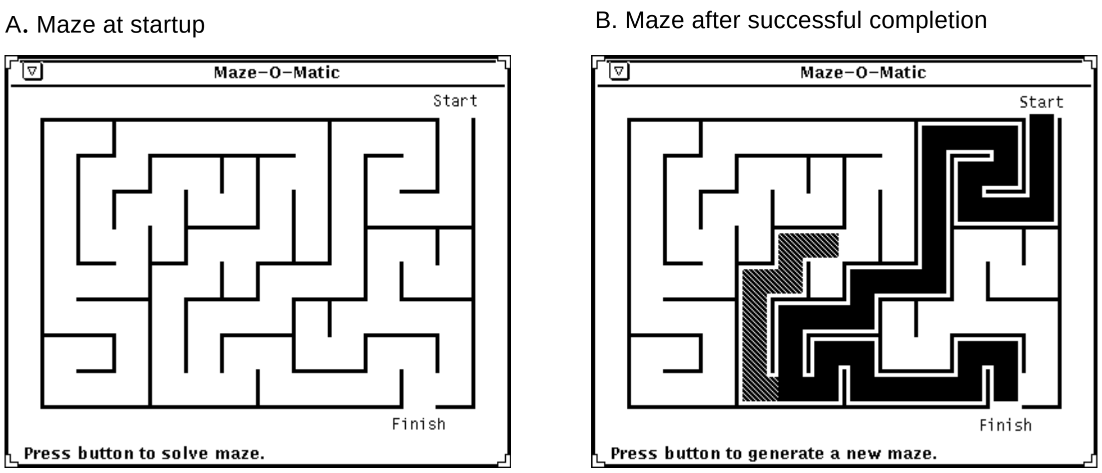

A. Maze at startup
B. Maze after successful completion
Maze–O–Matic
Start
Finish
Press button to solve maze.
Press button to generate a new maze.

Figure 1-1 Maze at Start and at Completion

Chris tries it again, forewarned that it misbehaves the second time through.

32.❖ Clicking SELECT once, as instructed, generates a new maze.

33.❖ Clicking SELECT again attempts a solve of the new maze.

As predicted, Maze-O-Matic fails to work this time, dead-ending in a blind maze pathway.

1.4 Examining `solve` in AdaVision

The main thing Chris knows about `solve` is that it is a multitasking program that spawns tasks at each intersection in the maze, and that these tasks then follow all the pathways simultaneously to solve the maze.

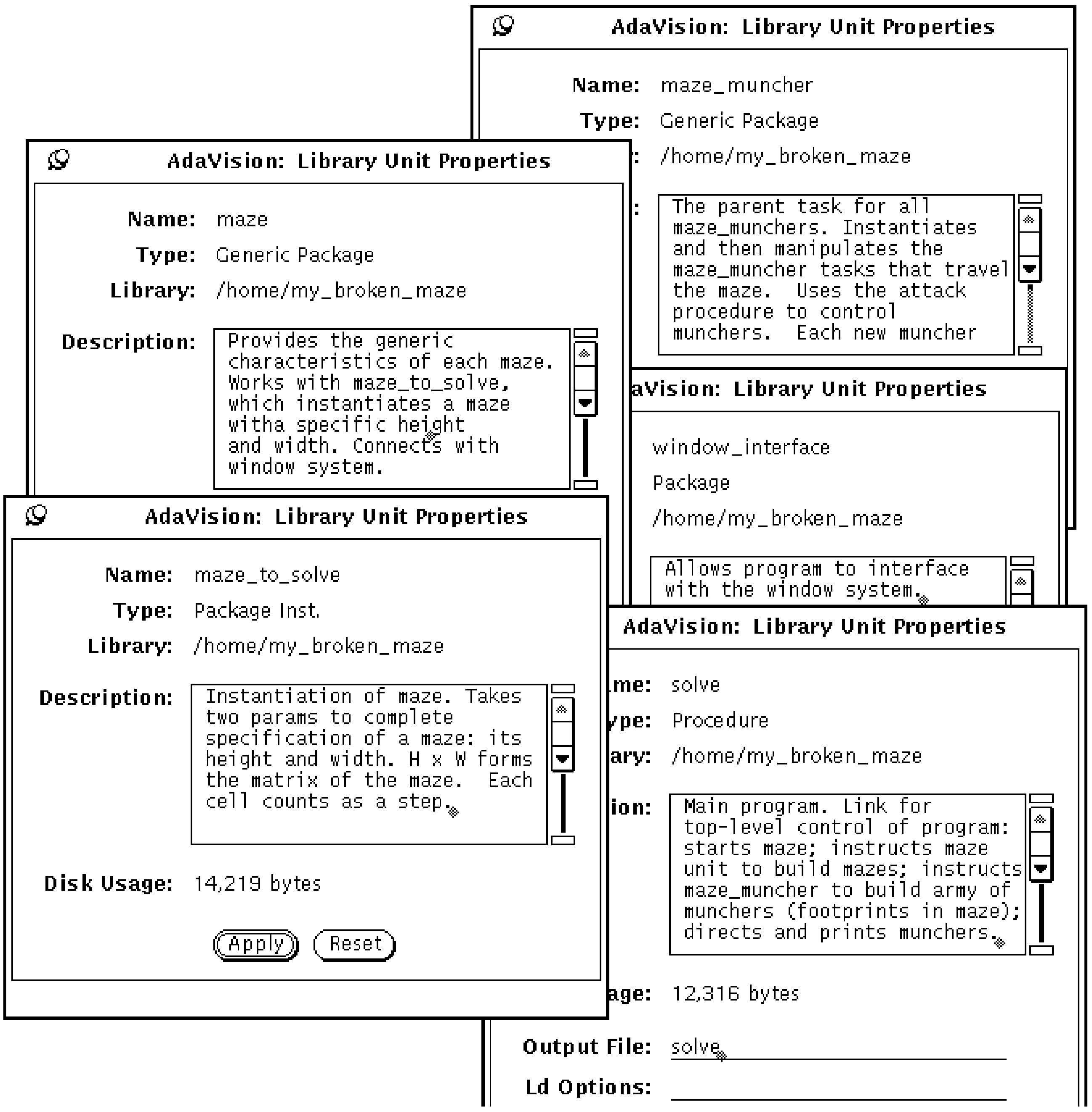
While watching the program fail, Chris noticed that it:

- Dead-ends
- Fails to make footprints on other pathways in each maze
- Seems to have trouble spawning tasks and sending them in each direction at an intersection

But this is all speculation. Before proceeding, Chris needs to know more about the program—more about how it is assembled. What little documentation is available is either in comments in the source files or in the brief descriptions of each LU that a friend wrote up early in the project. The first thing Chris does is look quickly at the description provided for each library unit.

34.❖ Working in LU mode, Chris selects each LU in turn and chooses Selection from the Props menu to display the Properties window for each LU.

The following figure is a composite of the Properties windows for the five units. The description of each unit is entered in the edit pane beside the Description control. Notice that the text panes have scrollbars. Several descriptions run longer than the few lines you see in each text pane without scrolling.



AdaVision: Library Unit Properties

Name: maze_muncher

Type: Generic Package

: /home/my_broken_maze

:
```
The parent task for all
maze_munchers. Instantiates
and then manipulates the
maze_muncher tasks that travel
the maze.  Uses the attack
procedure to control
munchers.  Each new muncher
```

AdaVision: Library Unit Properties

Name: maze

Type: Generic Package

Library: /home/my_broken_maze

Description:
```
Provides the generic
characteristics of each maze.
Works with maze_to_solve,
which instantiates a maze
witha specific height
and width. Connects with
window system.
```

aVision: Library Unit Properties

window_interface

Package

/home/my_broken_maze

```
Allows program to interface
with the window system.
```

AdaVision: Library Unit Properties

Name: maze_to_solve

Type: Package Inst.

Library: /home/my_broken_maze

Description:
```
Instantiation of maze. Takes
two params to complete
specification of a maze: its
height and width. H x W forms
the matrix of the maze.  Each
cell counts as a step.
```

Disk Usage: 14,219 bytes

AdaVision: Library Unit Properties

me: solve

ype: Procedure

ary: /home/my_broken_maze

ion:
```
Main program. Link for
top-level control of program:
starts maze; instructs maze
unit to build mazes; instructs
maze_muncher to build army of
munchers (footprints in maze);
directs and prints munchers.
```

age: 12,316 bytes

Output File: solve

Ld Options:

1.4.1 Views of the Library

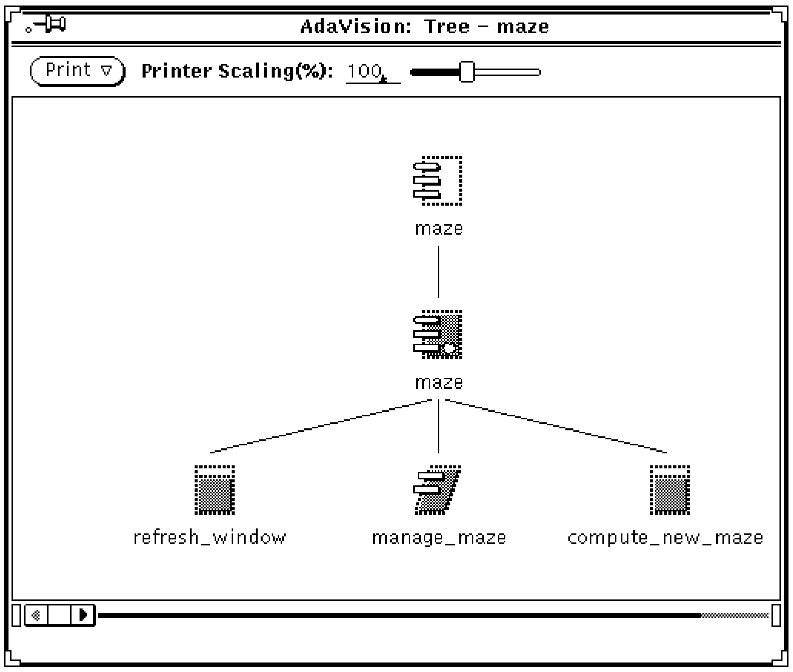
Now that Chris has a fair idea of how the program is assembled, it becomes important to see how the Ada LU objects are themselves structured into specs, bodies, and subunits.

1.4.1.1 Tree Graphs of Library Units

35.❖ Still working in LU mode, Chris selects the `maze` LU, then chooses Tree from the View menu.

AdaVision opens a base window containing a graph of the spec/body/subunit components of the `maze` LU object. (See the following figure.) Chris looks at the tree graphs for the other LUs. Since the `maze` program is small, the graphs for these other LUs are themselves small, in a few cases consisting of a single unit.

One of the separate tree graphs that exists for each LU in the library



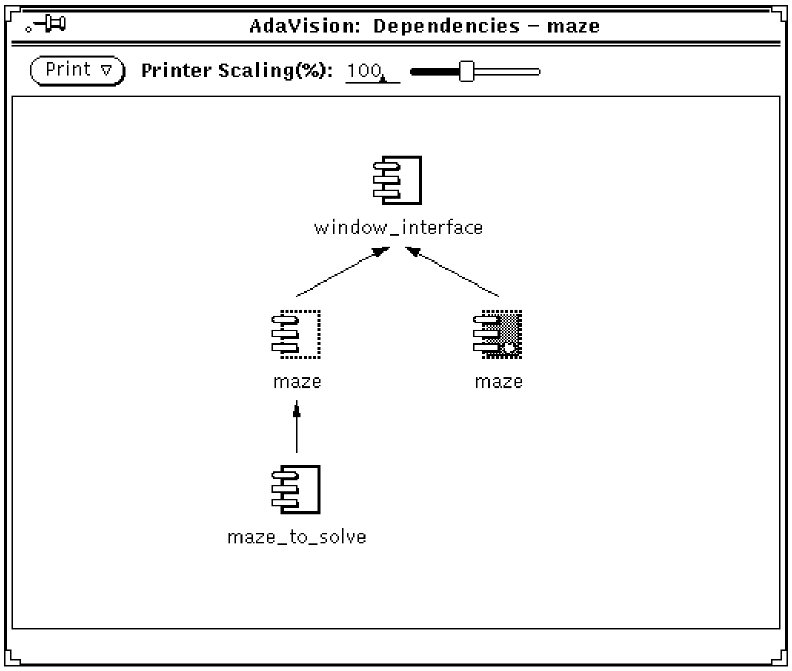
1.4.1.2 Dependency Graphs

Chris then decides to take a look at *dependencies* among the CUs. Since the library is small, all of the dependencies appear at one time. To view dependencies:

36.❖ Choose Select All from the Edit menu.

37.❖ Choose Dependencies from the View menu.

AdaVision opens a base window that graphs the interdependencies among the selected CUs that constitute the selected LU objects (in this case, all of the units in the `maze` program).



1.4.2 Two Hypotheses

Chris runs through the `maze` program again, looking at it with more informed eyes. (You can look at Figures A and B on page 1-24 to see what Chris sees, or run the `maze` program again yourself.)

Chris sees that the program is not spawning tasks in all possible directions at each intersection, as it is supposed to do. All that white space should be gray because task munchers should have gone down each path, flushing the entire maze. Also, in the second maze, a task did not head out to the left at the first intersection. After doing some thinking, Chris arrives at two hypotheses:

- Maybe only one task is running; the others are not being spawned successfully.
- Maybe tasks are spawned successfully, but they are dying for some reason before they get anywhere.

You can use the debugger tool to further examine the program.

This is the end of the first chapter of the tutorial. Chapter 2 of the tutorial concentrates on AdaDebug. You start again at (1), following the numbered steps Chris takes.

Because of the way the AdaDebug portion of the tutorial is constructed, it is important that you follow the steps precisely. A concise list of steps follows the end of the chapter. If you want to take the tutorial in more than one sitting (or if you miss a step that breaks the correct sequence), you may want to use the concise list of steps to return the program quickly to any particular place in the tutorial.

Solving the Maze (Part 2): AdaDebug 2

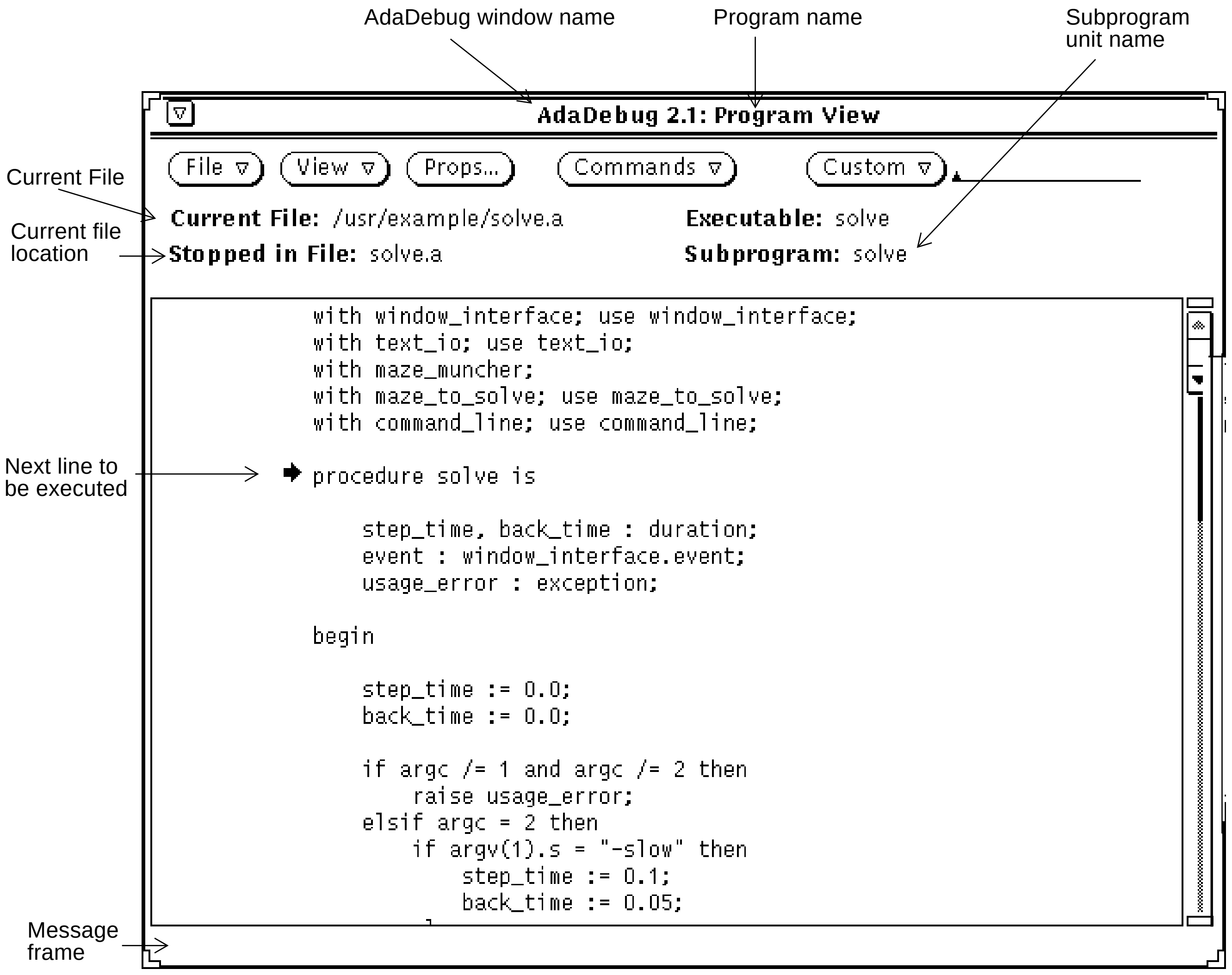
Chris is now ready to view and edit the source code. Although viewing and editing is possible from within AdaVision, by double-clicking SELECT on a unit icon, using the SPARCworks/Ada debugger, AdaDebug, is probably the fastest and most efficient method.

2.1 Starting AdaDebug From AdaVision

To start AdaDebug from within AdaVision:

1. ❖ Select the executable, `solve`, then
2. ❖ Choose Debug from the Commands menu.

If AdaDebug is already running on your Desktop, you can start a debugging session by dragging and dropping the `solve` icon onto the AdaDebug display pane. If neither AdaVision nor the debugger is running on your Desktop, you can type `AdaDebug solve &` in a Command or Shell Tool to start the debugger.



Current File: /usr/example/solve.a   Executable: solve
Stopped in File: solve.a   Subprogram: solve

```
        with window_interface; use window_interface;
        with text_io; use text_io;
        with maze_muncher;
        with maze_to_solve; use maze_to_solve;
        with command_line; use command_line;

      ➡ procedure solve is

            step_time, back_time : duration;
            event : window_interface.event;
            usage_error : exception;

        begin

            step_time := 0.0;
            back_time := 0.0;

            if argc /= 1 and argc /= 2 then
                raise usage_error;
            elsif argc = 2 then
                if argv(1).s = "-slow" then
                    step_time := 0.1;
                    back_time := 0.05;
```

At startup, AdaDebug displays the *Program View* window (shown above). AdaDebug displays the main program source code in the display pane. A small arrow marks the next line to be executed by the debugger—that is, the current position in the program.

The Program View window looks much like the AdaVision window. The commands controlling AdaDebug are arranged on four menu buttons. A fifth menu button, Props, contains a set of user option controls.

2.1.1 First Breakpoint Setting

Chris knows that the program solves the first maze so decides to find a place in the code where it passes the first maze but does not start the second one.

2.1.1.1 Line Numbers

To make orientation within the code easier, Chris turns on line numbering. To do this:

3. ❖ Choose Show Line Numbers from the View menu.

4. ❖ Next, scroll down halfway through the file to find the interesting parts of the code.

2.1.1.2 Breakpoint Setting

Line 56 reads:

```
generate_new_maze
```

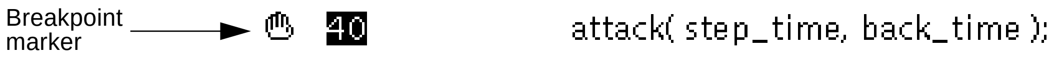
At first, this sounds like a good place to put a breakpoint. But on second thought, Chris figures that this statement probably comes too late to catch any of the action.

Chris reads the code that comes before this call to generate a new maze. At line 40, is a call to the `attack()` procedure. From the description of `solve`, it is fairly obvious that the program uses `attack()` to launch the task munchers, which solve the maze. A breakpoint at line 40 sounds like a good idea. If there is something wrong with the task munchers, this is a better place to start than earlier.

5. ❖ Chris double-clicks SELECT in line 40. The cursor highlights the word.

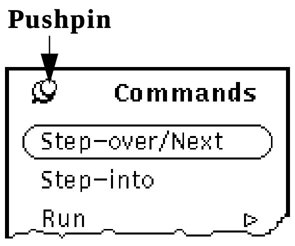
6. ❖ Then, Chris chooses Set Break from the Commands menu. (The default choice is Set Break => Line. To set a breakpoint at a subprogram, choose Subprogram from the Set Break pull-right menu.)

A glyph representing an open-palm hand appears to the left of line 40, indicating that a breakpoint is set at that line.



```
40            attack( step_time, back_time );
```

2.1.1.3 Commands Menu Pinning



Because the debugging commands will be used many times, Chris decides to pin the Commands menu to his Desktop.

7. ❖ Click MENU with the third mouse button, to open the Commands menu.

8. ❖ Pin the menu to the screen by clicking SELECT on the pushpin.

   Pinning prevents the menu from disappearing after you choose an item. Chris then drags the menu away from the display pane to read more of the code.

2.1.1.4 Program Execution

9. ❖ Having set a breakpoint, Chris now chooses Run from the Commands menu to start the program running from the beginning. Run always starts a program over again from the beginning. (Run => No Arguments is the default choice. To run a program with arguments, choose With Arguments from the Run pull-right menu.)

   The Maze-O-Matic window opens. The message in the Maze-O-Matic window prompts the user to click the mouse button to solve the maze.

10. ❖ Chris clicks SELECT in the maze window to start the program solving the first maze.

    The program runs until it hits the breakpoint at the call to `attack()`. The arrow indicating the next line to be executed points at line 40 where the breakpoint is set. Nothing of interest has happened yet in the Maze-O-Matic window. The message in the footer changes to say `Solving maze...`

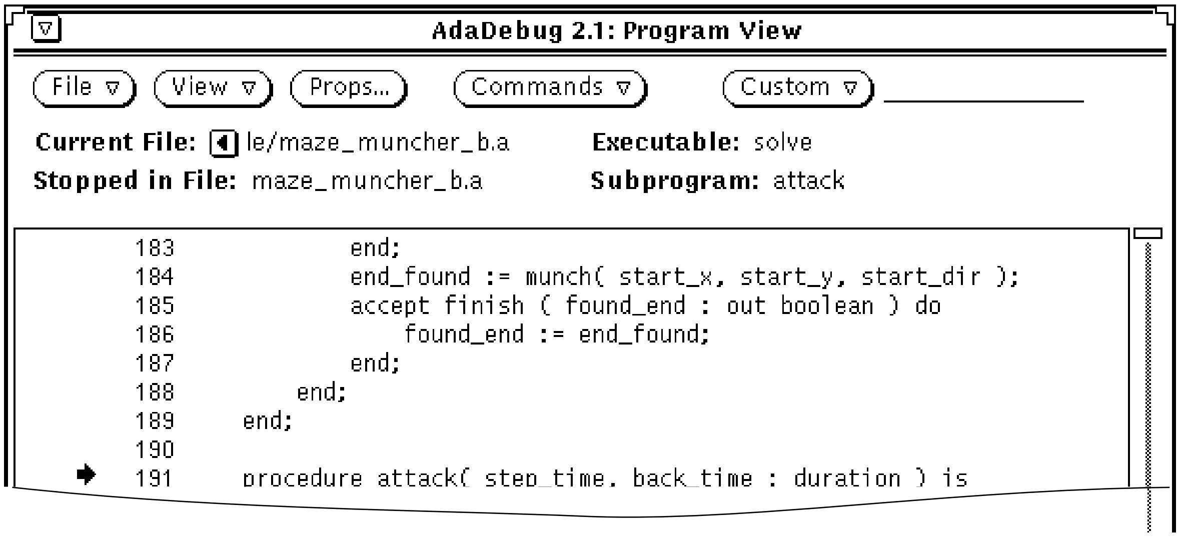
Chris recalls that the program always solves the first maze, even though it does not work exactly as it is supposed to work. (It doesn't send out munchers to explore the opposite path when it reaches an intersection.) Chris decides to work with the first maze for awhile.

2.1.1.5 *Step-into Facility*

To investigate what is happening when the program calls the `attack()` procedure, Chris decides to step into this procedure. To do this:

11.❖ Choose Step-into from the pinned Commands menu to step into the `attack()` procedure.

The Program View window now displays the `attack()` procedure. AdaDebug positions the current line arrow at the first line of `attack()` (line 191). The AdaDebug updated information in the Stopped in File and Subprogram message fields located above the display pane.



Current File: le/maze_muncher_b.a   Executable: solve
Stopped in File: maze_muncher_b.a   Subprogram: attack

```
 183            end;
 184            end_found := munch( start_x, start_y, start_dir );
 185            accept finish ( found_end : out boolean ) do
 186                found_end := end_found;
 187            end;
 188        end;
 189    end;
 190
➡191    procedure attack( step_time, back_time : duration ) is
```

At this point, Chris assumes that the program will behave properly so starts stepping through `attack()` line by line:

12.❖ Choose Step-over / Next from the Commands menu.

Step-Over / Next is the default item on the Program View window Commands menu. If the Commands menu is not pinned open, you can click SELECT on the Commands menu button to choose Step_over / Next.

13.❖ After repeating Step-over / Next 8 more times, advancing to line 206 and reading ahead a line or two to analyze what is happening before each step, Chris also observes what happens in the Maze-O-Matic window. With the current line arrow pointing at line 206, nothing has happened.

Line 206 reads:

```
206       first_muncher:=
          start_muncher(start_x,\
                 start_y,start_dir);
```

Chris looks ahead to the next line:

```
207       first_muncher.finish(found_finish);
```

Realizing that the munchers' progress results in footprints on the maze, Chris doesn't want to allow the program to step over line 206, the action involved in drawing the first footprint will be lost. So Chris decides to Step Into `start_muncher`.

14.❖ Chris chooses Step-Into at the `start_muncher` procedure.

AdaDebug scrolls the display to point at line 21, where `solve` declares the `start_muncher()` function. This is a short but important routine.

15.❖ Chris chooses Step-Over / Next until reaching line 26.

The second line in `start_muncher()` (25) allocates a *task* of type `maze_muncher`. In stepping over line 25, Chris realizes that `solve` has just created a new task. Chris also recognizes that the next line (26) is a *task rendezvous* call:

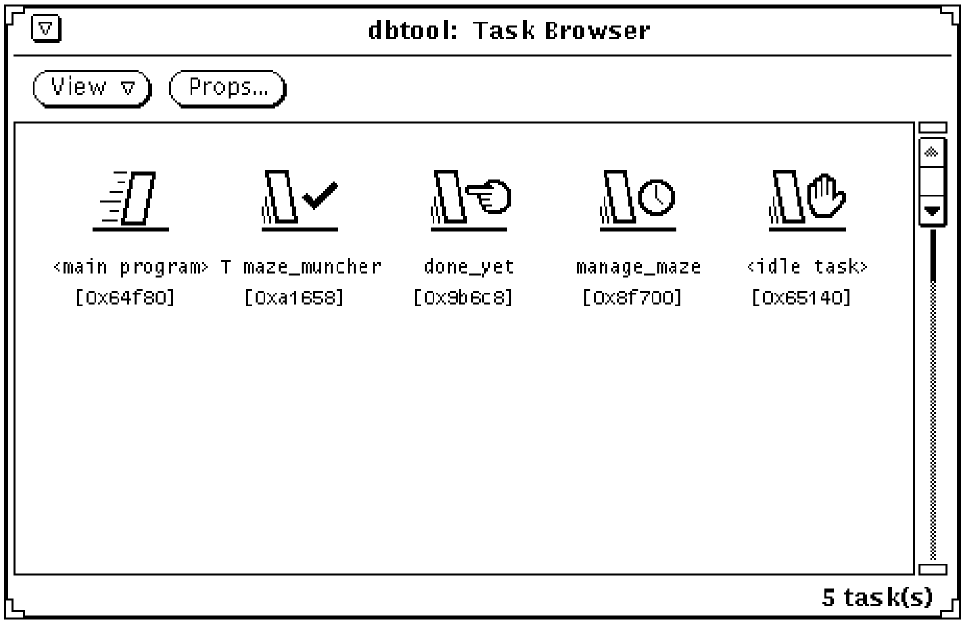
```
26     new_muncher.start(x, y, dir);
```

Things are becoming more concrete now. Chris recalls that the `maze_muncher` unit uses `attack()` to control the munchers that travel the maze. Now it becomes obvious that each muncher is a task.

2.1.2 Task Examination Using the Task Browser

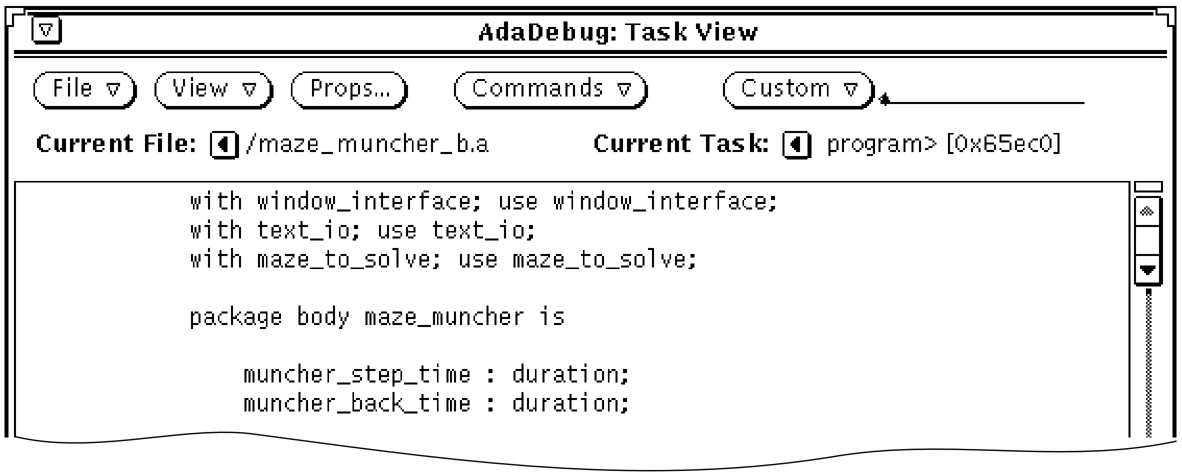
Chris decides to examine the task muncher just created, using the Task Browser. From the Task Browser, the task opens into a Task View window; then a *task-specific breakpoint* can be set at the `accept` statement for this muncher. Chris wants to set a breakpoint at the line containing the `accept start` statement for this new muncher. By setting a breakpoint in the Task View window, the program will reach the breakpoint only if this particular task attempts to execute this line of code.

16.❖ Chris chooses Task Browser from the View menu to open the AdaDebug Task Browser window, which displays an object for each activated task.



As expected, the icon for the new muncher task that `start_muncher` just created is visible. It is named `T maze_muncher`.

17.❖ Chris double-clicks SELECT on the `T maze_muncher` task icon to open the task in a Task View window.


AdaDebug: Task View
File
View
Props...
Commands
Custom
Current File: /maze_muncher_b.a
Current Task: program> [0x65ec0]

```
with window_interface; use window_interface;
with text_io; use text_io;
with maze_to_solve; use maze_to_solve;

package body maze_muncher is

    muncher_step_time : duration;
    muncher_back_time : duration;
```

2.1.2.1 Task Views

The Task View window looks much the same as the Program View window. The important difference is functional: when you choose a menu item from within a Task View window, it applies only to the task identified in the Task View title bar.

18.❖ First, Chris chooses Show Line Numbers from the View menu.

2.1.2.2 Command Line

To locate the `accept start` statement, Chris uses the `a.db` search (/) command from the AdaDebug command line.

19.❖ Chris types `/accept start` on the command line in the Task View window and presses the Return key.

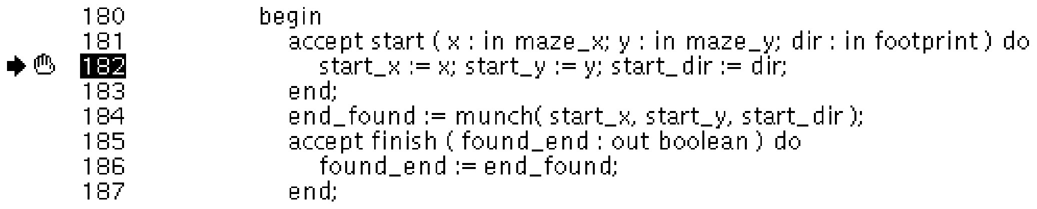
The code in the Task View pane scrolls to bring the `accept start` statement into view (line 181).

Chris realizes that a breakpoint at line 181 is too early; the program should execute the `accept start` rendezvous statement, then stop. So the task-specific breakpoint goes on line 182.

20.❖ Chris double-clicks SELECT on line number 182 and then chooses Commands => Set Break, *being careful to choose Set Break* from the *Task View* window Commands menu to set this breakpoint for this task, not the *Program View* window Commands menu. (Use the Program View Commands menu to set a breakpoint.)

21.v Chris chooses Continue from the Program View Commands menu.

The program runs until the task named `T Maze_Muncher` hits this task-specific breakpoint.



```
180          begin
181             accept start ( x : in maze_x; y : in maze_y; dir : in footprint ) do
182                start_x := x; start_y := y; start_dir := dir;
183             end;
184             end_found := munch( start_x, start_y, start_dir );
185             accept finish ( found_end : out boolean ) do
186                found_end := end_found;
187             end;
```

Still, nothing has happened in the Maze-O-Matic window.

Chris wants some workspace back and doesn't need the Task View window any more, having set the task-specific breakpoint in it:

22.❖ Choose Quit from the Task View Window menu at the title bar.

Clear the breakpoint at line 182, which has already served its purpose.

23.❖ Double-click SELECT on line 182 and choose Clear Break from the Commands menu.

(Chris clears the task-specific breakpoint from the Program View window, where the glyph indicating its presence is displayed.)

Chris looks over the code in the `accept start()` statement and sees that by stepping into the `munch()` procedure (line 184), the entire process by which `solve` creates task munchers can be stepped through.

24.❖ Choose Step-over / Next *twice* to advance the current line pointer to line 184.

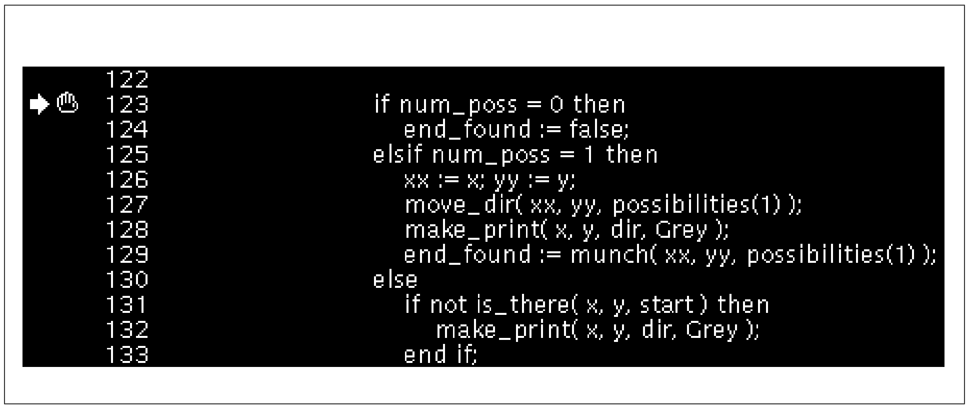
25.❖ Choose Step Into at line 184, to go to the start of the `munch()` procedure (line 70).

2.1.3 Tutorial Fast-Forward

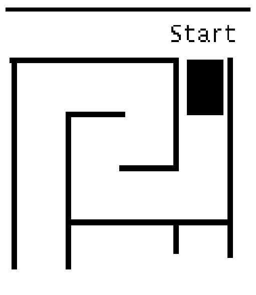
At this place in the tutorial, we are skipping the steps that Chris takes to discover that a new breakpoint needs to be set at line 123. So also do we skip the many Step-overs that cycle through the loop in the code where the program calculates how many directions there are. The program cycles through lines 131-137 four times—one for each direction—then leaves the loop at line 123.

2.1.4 Footprints in the Maze

26.❖ Chris chooses Set Break at line 123 and then chooses Continue. The program advances to the new breakpoint:

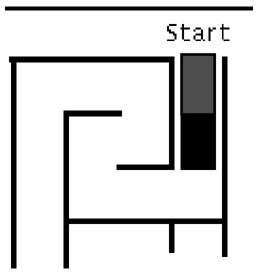


```
122
123        if num_poss = 0 then
124           end_found := false;
125        elsif num_poss = 1 then
126           xx := x; yy := y;
127           move_dir( xx, yy, possibilities(1) );
128           make_print( x, y, dir, Grey );
129           end_found := munch( xx, yy, possibilities(1) );
130        else
131           if not is_there( x, y, start ) then
132              make_print( x, y, dir, Grey );
133           end if;
```



In the Maze-O-Matic window, the program draws a single, heavy black footprint in the maze. At last, some real action, and the kind Chris hoped for: a single footprint in the maze.

Chris decides to step through `munch()` one line at a time and see what happens next.



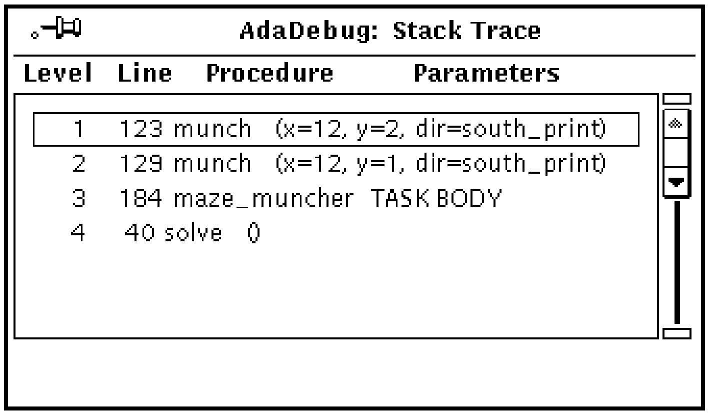
27.❖ Chris chooses Step-over / Next 6 times. The fifth step executed the `make_print(x, y, dir Grey)` procedure (line 128), causing the first footprint to turn gray.

When Chris takes the sixth step-over, to line 129, something unexpected happens: the program returns to line 123, even though the program is not in a loop, and `solve` draws another footprint in the maze.

Chris figures that `munch()` must be calling itself recursively. To check this hypothesis, Chris brings up the Stack Trace window.

2.1.5 *Stack Tracing*

28.❖ Choose Stack Trace from near the bottom of the Commands menu:



```
Level  Line  Procedure        Parameters
  1     123 munch   (x=12, y=2, dir=south_print)
  2     129 munch   (x=12, y=1, dir=south_print)
  3     184 maze_muncher  TASK BODY
  4      40 solve   ()
```

The Stack Trace window display provides Chris with some key pieces of information. First, the top two entries on the stack are calls to `munch()`. Looking at the values of the parameters and comparing them to the location of the two footprints with respect to the maze—which, Chris recalls, is a 12 x 8 matrix—shows that the coordinates and the direction match perfectly. Therefore, `south_print` must mean “downward.”

29.❖ Chris steps through the recursive lines of code again (lines 123 through 129, returning to 123) just to see the third footprint appear and the new call show up on the Stack Trace. (You have an option to follow Chris here.)

2.1.6 Maze Footprints

Chris concentrates on the main problem with the maze: footprints do not explore each avenue (or “possibilities” as the code calls them) when one comes to an intersection. Chris looks ahead in the code to where the `start_muncher()` procedure begins (line 137) and decides to set the next breakpoint there.

30.❖ Select a word in line 123 and choose Clear Break to remove the breakpoint at line 123.

31.❖ Close the Stack Trace window. (Unpin it, or double-click SELECT on the pushpin.) It is no longer needed and it slows things down to have AdaDebug running in trace mode indefinitely.

32.❖ Then, double-click SELECT on a word in line 137, choose Set Break, wait for the breakpoint marker to be displayed, then choose Continue under Commands.

The maze takes off this time, drawing footprints one after another until it reaches the first two-way intersection in the maze. It stops at the intersection.

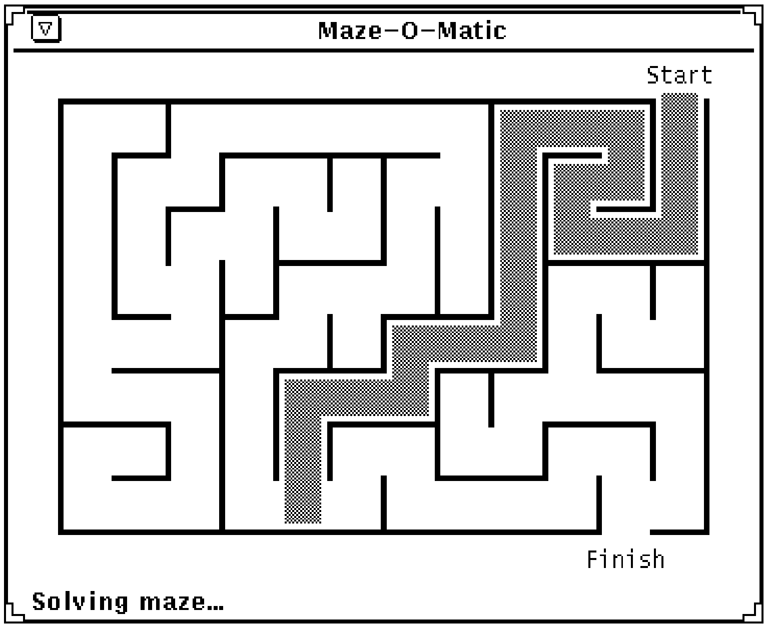


Figure 2-1 Maze Stops at an Intersection

Chris now realizes that `solve` must spawn two new maze muncher tasks here, sending one in each direction. The breakpoint set at the line containing `start_muncher()` has stopped the program just before it creates the new task munchers.

Chris also realizes that this is the place where the maze fails. It fails because, when it reaches intersections, it sends a muncher in one direction but not the other.

2.1.7 Conclusion

Chris decides to evaluate the parameters of the `start_muncher()` procedure: `xx`, `yy`, and `possibilities(i)`.

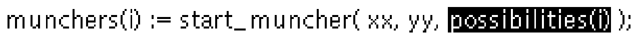
33.❖ Double-click SELECT to select the `xx` parameter in `start_muncher()`, then choose Evaluate from the bottom of the Commands menu.

AdaDebug opens the Value Display pop-up window; it shows `xx = 6`. Chris counts 6 cells from the left. The `xx` coordinate is correct but how is the `yy` coordinate?

34.❖ Select the `yy` parameter, then choose Evaluate.

The Value Display window shows: `yy = 8`, the bottom row in the matrix. This, too, is correct so what are the `possibilities(i)`?

35.❖ Select `possibilities`, then click ADJUST to adjust the selection and extend the selection to include the parameter `(i)`.



```
munchers(i) := start_muncher( xx, yy, possibilities(i) );
```

36. ❖Choose Evaluate again.

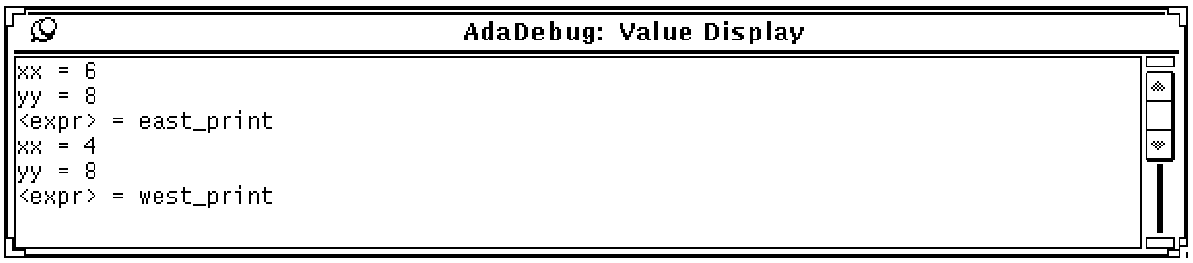
The Value Display window reports: `(1 = east_print)`.

This value is also correct. Things seem to be in order thus far.

37.❖ Choose Continue.

Choosing Continue here at an intersection does not advance the maze; instead, the program starts up a second maze muncher, as expected. A second `T maze_muncher` appears in the Task Browser. This is the muncher that should go in the other direction, west. Chris tests this out by evaluating the arguments to `start_muncher()` for this second task:

38.❖ Chris evaluates the `start_muncher()` parameters, `xx`, `yy`, and `possibilities(i)`. (Evaluate these three parameters in three separate steps, as described in steps 32-35). The results are shown in the following figure:



```
xx = 6
yy = 8
<expr> = east_print
xx = 4
yy = 8
<expr> = west_print
```

The direction is not correct: `east_print` coordinates are the same for the first muncher. It appears that the second muncher is preparing to follow the first one. Why?

Chris looks at the line above the `start_muncher()` call, to where the program calls `move_dir()`:

```
136        move_dir( xx, yy, possibilities(1) );
```

In an instant, Chris knows that this is the bug: the parameter should be `possiblities(i)`, not `possiblities(1)`! There are four possible directions, not one. No wonder the task munchers travel in one direction when they come to an intersection: the `possibilities` variable in the `mov_dir` call was coded mistakenly as a constant.

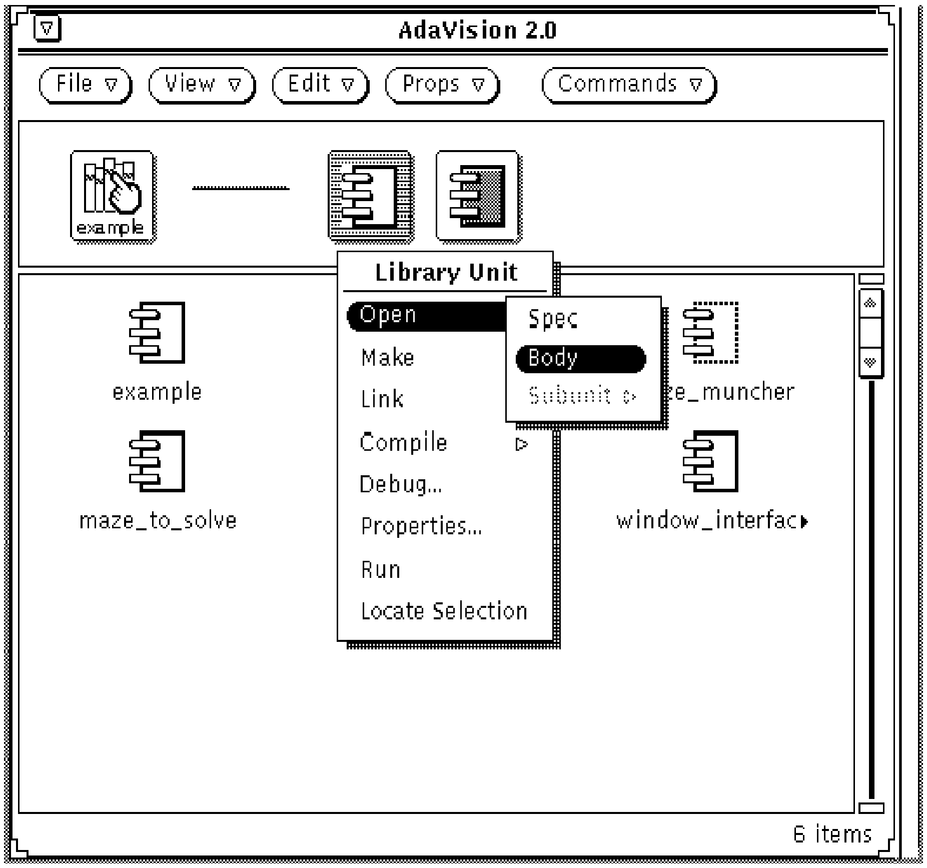
Chris is sure that by changing “`1`” to “`i`,” the program will behave correctly.

2.1.8 Bug Fixing in AdaVision

Chris shifts back to AdaVision to do the editing and then make the library again. AdaVision is still in LU mode.

39.❖ Chris selects the `maze_muncher` LU object.

40.❖ Then, Chris chooses Open => Body, from the selected file (`solve`), using the right mouse button. AdaVision opens the body of the `maze_muncher` LU in an EditTool window.



Remember that in LU mode, AdaVision displays LU objects. These LU objects are composites of a SPARCompiler Ada LU spec, body, and subunits (if there are any). When you want to open a unit from LU mode, you must indicate which unit in the LU object you want to open.

41.❖ Choose Select Line at Number from the EditTool View menu to open the Line Number pop-up window.

42.❖ Type in line 136, which contains the call to `mov_dir()`, then press the Select Line Number button. EditTool scrolls the code to line 136 and highlights it.

43.❖ Edit the variable, changing “`1`” to “`i`” in the `possibilities` parameter in the `mov_dir()` call.

44.❖ Choose Save then Quit from the File menu. Return to the AdaVision window and click SELECT in a blank area of the display pane to *deselect* the `maze_muncher` LU object.

Notice that, in LU mode, when an LU object is selected, the Make item is the top item on the Commands menu. If you choose Make, AdaVision makes, that is, compiles and relinks the units in the selected LU object only. When no LU units are selected, the top item on the Commands menu changes to Make Library. When you choose Make Library, AdaVision makes the entire library (including any units which might not be visible on the display).

---

45. ❖ Next, Chris chooses Make Library from the LU mode Commands menu.

AdaVision starts a Make Library job, which runs make on *all* of the units in the library. AdaVision reports on the progress and outcome of this background job in a Make Library Status icon/window. In a moment, the gauge on the icon tells Chris that the job has completed successfully.

46.❖ Chris drags and drops the `solve` executable onto the Desktop, then waits a moment for Maze-O-Matic to be displayed, along with a Program I/O window.

47.❖ Chris clicks SELECT in the Maze-O-Matic window.

48.❖ After the program solves the first maze handily, Chris clicks SELECT to generate a new maze, then clicks SELECT again to solve the second maze.

Voila! Chris has corrected the error and `solve` now solves each of the mazes correctly.

List of Steps for AdaDebug Tutorial A

This appendix provides a list of just the steps in the AdaDebug portion of the tutorial. Having the steps in this format should make it easier to restart the tutorial AdaDebug session, should you want to take the tutorial in more than one sitting, or if you stray from following in Chris's exact footsteps.

The steps here are presented in a more direct and procedural style than the text presents them and the diamond (❖) is omitted.

1 In AdaVision, LU mode, select `solve`
2 Choose Debug from Commands menu
3 Choose Line Numbers from View menu
4 Scroll down until line 40 is in view
5 Select a word or number in line 40
6 Choose Set Break from Commands menu
7 Open the Commands menu
8 Pin the Commands menu open
9 Choose Run from Commands menu
10 Click SELECT in the Maze-O-Matic window
11 Choose Step Into from Commands menu
12 Choose Step-over/Next from Commands menu
13 Choose Step-over/Next to get to line 206
14 Choose Step Into at line 206
15 Choose Step-over to line 26
16 Choose Task Browser from View menu
17 Open the `T maze_muncher` task
18 Choose Line Numbers from View menu
19 Type `/accept start` on the command line
20 Select a character in line 182
Choose Set Break from Task View window menu
21 Choose Continue from Program View Commands menu
22 Choose Quit from Task View title bar Window menu
23 Select a word in line 182; choose Clear Break from Commands menu
24 Choose Step-over/Next to get to line 184
25 Choose Step Into at line 184
26 Select a word in line 123, choose Set Break, then choose Continue
27 Choose Step-over/Next until reaching line 123; (choose the item 6 times)
28 Choose Stack Trace from Commands menu
29 Select a word in line 123; Choose Clear Break from Commands menu
30 Dismiss the Stack Trace pop-up window
31 Select a word in line 137
Choose Set Break
Choose Continue
32 Select `xx`
Choose Evaluate from Commands menu
33 Select `yy;`Choose Evaluate from Commands menu
34 Select `possibilities(i)` (select `possibilities`, then click ADJUST to adjust the selection to include the parameter `(i)`)
35 Choose Evaluate
36 Choose Continue
37 Select and Evaluate again each of the three `start_muncher()`parameters, as in steps 32-35
38 In AdaVision, select `maze_muncher` LU object
39 Choose Open => Body from floating pop-up menu
40 In EditTool, choose View => Select Line Number
41 Choose Select Line at Number from Edit Tool View menu.
42 Type `136`, press Select Line Number button.
43 Change `1` to `i` in `mov_dir()` procedure
44 Choose File => Save Current File
45 Choose Make Library from Commands menu
46 Drag and drop `solve` icon onto your workspace
47 Click SELECT in Maze-O-Matic window to test the program
48 Click SELECT again to solve the second maze