

SNiFF+TM

Version 3.2 for Unix and Windows

Ada Tutorial



TakeFive Software, Inc.

Cupertino, CA

E-mail: info@takefive.com

TakeFive Software GmbH

5020 Salzburg, Austria

E-mail: info@takefive.co.at

Copyright

Copyright © 1992–1999 TakeFive Software Inc.

All rights reserved. TakeFive products contain trade secrets and confidential and proprietary information of TakeFive Software Inc. Use of this copyright notice is precautionary and does not imply publication or disclosure.

Parts of SNIFF+:

Copyright 1991, 1992, 1993, 1994 by Stichting Mathematisch Centrum,
Amsterdam, The Netherlands.

Portions copyright 1991-1997 Compuware Corporation.

Trademarks

SNiFF+ is a trademark of TakeFive Software Inc.

Other brand or product names are trademarks or registered trademarks of their respective holders.

Credits

The first version of Sniff was developed at the Informatics Laboratory of the Union Bank of Switzerland. Its development was considerably facilitated by the public domain application framework ET++.

Authors of the first version:

Walter Bischofberger (Sniff)

Erich Gamma (Sniffgdb)

Erich Gamma and André Weinand (ET++)

Table of Contents

Creating a Single-User Project	5
Preparing the Environment	5
Creating a Private Working Environment	5
Creating a new project.	6
Viewing the results	8
Conclusions.	8
Browsing Symbols	9
Using the Symbol Browser	9
Ada entries in the Symbol Type drop-down.	11
Follow Cross References	14
Cross referencing	14
Abbreviations used in Cross Referencer.	18
Browsing examples	19
Browsing Includes	19
Hierarchy Browsing	21
Using the Retriever	22
Parser Options	24
Environment Parser Options	24
Project-specific Parser Options	24
Building the Project's Executable	27
Setting up Make Support.	27
Building the project target	28

Ada Tutorial

Creating a Single-User Project

In this chapter, you will learn how to create a single-user project for the example code. After Preparing the Environment you will create a Private Working Environment and then create a single-user project.

Preparing the Environment

- Copy the directory

`<your_sniff_installation_dir>/example/ada/diners`, including subdirectories, to a place where you have write permissions.

In the rest of this tutorial, we will use `<diners>` to refer to the complete path to this directory.

- Start SNIFF+

The Launch Pad appears.

Creating a Private Working Environment

In this part, you will create a Private Working Environment (PWE) for using it with the example code. As we do not want to share the example files with others, the PWE will not be based on any Repository Working Environment (RWE). For the same reason, no CVMC tool will be used.

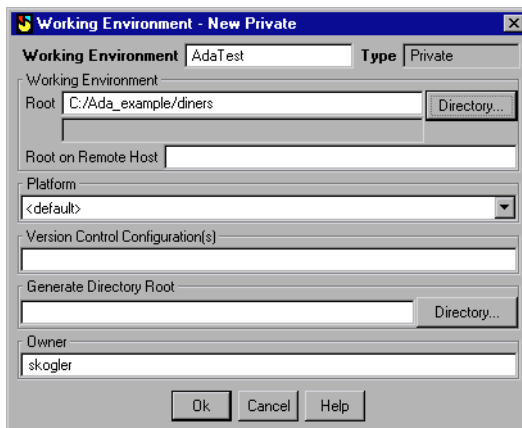
- Choose **Tools > Working Environments**, (on UNIX, the **Tools** menu is represented by an icon). The Working Environments Tool appears.

In the Working Environments Tool

1. Select the root (*) of the Working Environments Tree.

2. Choose **Context menu > New Private**.

A dialog appears.



3. Enter a name for your PWE - in the **Working Environment** field, for example **AdaTest**.
4. To select the **Root** of your PWE:
Press the **Directory...** button to navigate to <diners>, double-click on it and Press **Select**.
5. Press **OK**.
The new PWE is created and highlighted.

Creating a new project

In this part you will create projects from the example code in your PWE.

1. In the Launch Pad choose **Project > New Project... > With Defaults...**
2. A Directory Dialog appears.
Navigate to <diners> and press **Select**.
The "Attributes of New Project" dialog appears.

In the "Attributes of New Project" dialog

1. Select the **Generate Subproject Tree** checkbox (if it isn't already selected).
Using this checkbox, any subdirectories located in your project directory will be recursively processed, and for all of them a sub-project will be generated.
2. Select the **Build Options** node.
3. Select the **Use SNIFF+ Make Support** checkbox.

4. Select the **File Types** node.

The following file types should be included in our example project:

Ada9X

Make

Project Description

To accomplish this:

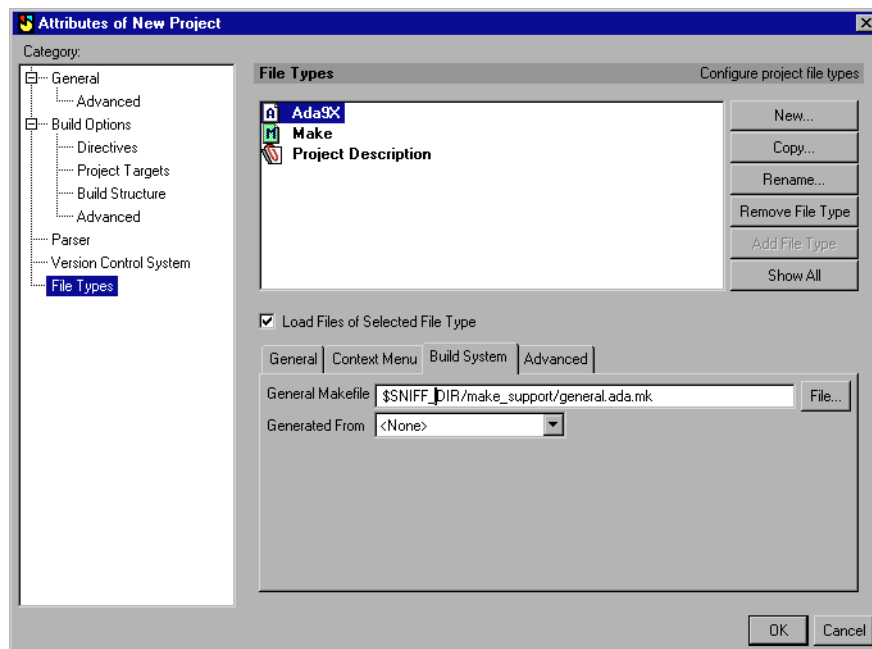
- From the File Types list select **Ada9X** (if there aren't all File Types visible press the **Show All** button).
- Press the **Add File Type** button (note that Ada9X changed from italics to bold).
- Press the **Hide Unused** button.
- Select **Header** from the File Types list and press the **Remove File Type** button.
- Select **Implementation** from the File Types list and press the **Remove File Type** button.

If you do not have Ada9X file type, you can create it at user level, or have the Workspace Administrator create it at site level.

5. Select the **Build System** tab.

6. Press the **File...** button to navigate to

`<your_sniff_installation_dir>/make_support/general.ada.mk.`



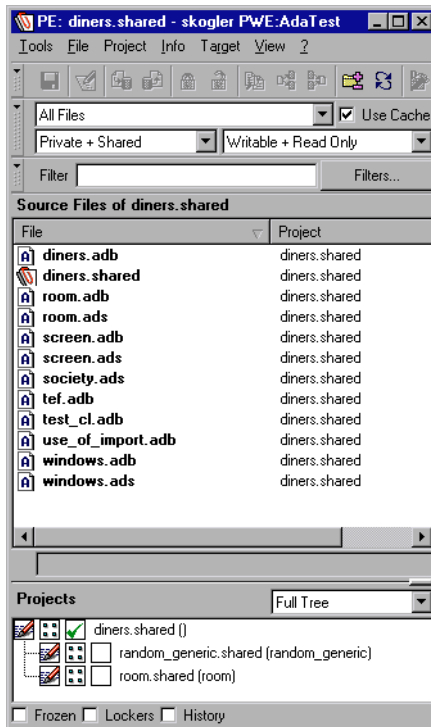
7. Press **OK**.

8. A dialog box appears—asking if the Cross Reference Info should be generated. Press the **Yes** button.

Your project `diners.shared` with two subprojects - `room.shared` and `random_generic.shared` - has just been created.

Viewing the results

The Project Editor on your screen should look like this:



Conclusions

You have just created a single-user project for browsing the Ada example project. Starting with the next chapter, you will learn how to use the various browsing tools available in SNIFF+. In the last chapter in this tutorial, you will set up SNIFF+'s Make Support for the project and then build the project's executable.

2

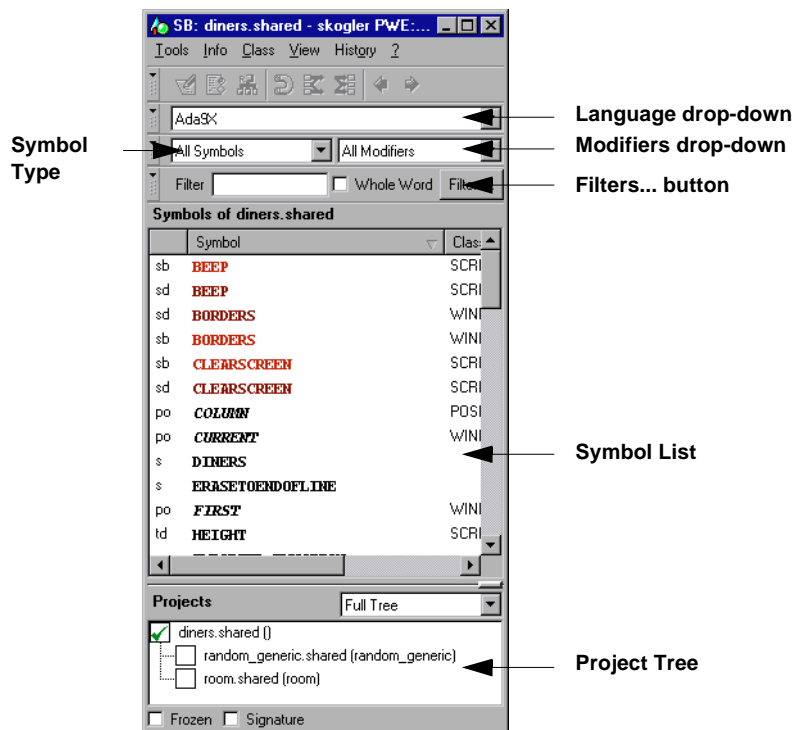
Browsing Symbols

In this chapter, you will learn how to use the Symbol Browser as a starting point for browsing your project's code. The Symbol Browser allows you to browse all global symbols and symbol members of a set of projects. It offers a wide range of possibilities for filtering information.

The Symbol Browser consists of a list of symbols whose content is determined by the Symbol Type and Modifiers drop-downs, the Project Tree settings, and the Filter field. The Project Tree shows the project structure and makes it possible to select the projects whose symbols are to be displayed. For detailed information about the Symbol Browser, please refer to your SNIFF+ online documentation or the *Reference Guide*.

Using the Symbol Browser

1. In the Project Editor, choose **Tools > Symbol Browser**. The Symbol Browser appears.



2. Take a look at the **Language** drop-down. The SNIFF+ Ada Parser is actually a Ada 95 parser that understands the Ada 83 subset of the language. For all Ada projects, the language string displayed in the drop-down is **Ada9X**.
3. Since we are interested in browsing all three projects in the Project Tree, let's checkmark them at this time. To checkmark all three projects quickly, right click anywhere in the Project Tree and choose **Context menu > Select from All Projects**.
4. Look at the entries in the **Symbol Type** drop-down:

The following symbols are listed:

- const
 - enum
 - enum item
 - object
 - package
 - pkg object
 - pkg subprog (body and def)
 - record
 - subprog (for functions, procedures and entries)
 - typedef
5. Choose the various entries in the **Symbol Type** drop-down and see what happens. By selecting the **Signature** check-box, you can see in which files the symbols appear. Also, information about object types, subprogram parameters and return types will be displayed.

Note that as Ada is not case sensitive, all symbol names appear converted to upper case. Also note that (non-package) objects and constants are not present in this project, resulting in an empty Symbol List when you select them. Named number `TABLE_SIZE` of package `ROOM` can be found among package objects, having the type `_NamedNumber_`.

Choose **subprog def**. All package subprograms defined inside package specifications in the project are listed in the Symbol List.

Choose **All Symbols**.

6. Scroll down the list till you get to `sd REPORT_STATE` and double-click on it.

A Source Editor appears and the file `room.ads` is loaded into it. As you have just found out, double-clicking a symbol in the Symbol List opens a Source Editor and loads the file in which the symbol appears. The cursor is automatically positioned to the symbol in the file.

In the next chapter, we will use `REPORT_STATE` to show how you can perform cross referencing in SNIFF+.

Ada entries in the Symbol Type drop-down

package

When you select this symbol type, packages defined in the project are listed in the Symbol Browser's Symbol List. Also, task, task types and protected types are shown here. Enable **Signature** button will list project and file names in which these entities are defined and will prefix records as **record** and all other symbols as **package**.

Note

There are some different kind of Ada symbols, which appear under the same category in SNIFF+. For example, SNIFF+'s package type corresponds to packages, records, tasks, task types and protected types. Similarly, subprog type corresponds to subprograms and exceptions. In this tutorial, when we refer to one of these collective categories, we always refer to all Ada language constructs which are mapped to the same collective category. For example, SNIFF+'s various capabilities that apply to packages also apply to task types and to all other constructs that you can find in **package** of the **Symbol Type** drop-down.

enum

When you select this symbol type, enumeration types defined in the project are listed in the Symbol List. In case of enumeration types defined in a package, enabling the **Signature** checkbox will prefix those enums with the name of the enclosing package in the form **PACKAGE::ENUM**. In the example project, for example, you can see **PHIL::STATES** enumeration type. The project and the file name in which the definition occur, is also shown. Double-clicking on the symbol will start a Source Editor and jump to the enumeration type definition.

enum item

When you select this symbol type, all enumeration items defined in the project are listed in the Symbol List. Enabling the **Signature** checkbox will show the enumeration type to which this enumeration item belongs, in the form described above. In this case, the enumeration item name appears in brackets. For example, You can see **PHIL::STATES {EATING}**.

subprog

When you select this symbol type, a list of all global procedures and functions defined in your project is displayed in the Symbol List. Global here is used to refer subprograms which are not defined inside package specifications, that is, subprograms declared outside packages and subprograms that are defined inside package bodies, with no corresponding specification in the package specification. These subprograms are local to the package body, and as they are not visible from the package specification, are considered not to be a part of the

package. Subprograms defined in package specifications are not listed here (you can find them when you select **pkg subprog def** (or **pkg subprog body**) from the **Symbol Type** drop-down).

To see the parameter and return types and of the subprograms, enable the **Signature** button. You can see the return values in front of the function names. In case of procedures, the return type is empty. Parameter types can be seen after the subprogram names enclosed in parenthesis and separated by commas. If no parameters exist, you can see an empty opening and closing parenthesis, for example, **DINERS()**.

object

When you select this symbol type, all global objects and named numbers are displayed. Enabling the **Signature** checkbox will show the object's type in front of its name. For named numbers, type **_NamedNumber_** is shown.

In our example project, no global objects are defined, so the symbol list is empty.

pkg subprog

When you select this symbol type, all package subprograms are listed. With **Signature** disabled, you can see the package name in which the subprogram is specified after the symbol name, for example, **OPEN** is followed by **WINDOWS**, meaning that **OPEN** is a subprogram of package **WINDOWS**. Enabling **Signature** will display more detailed information. First the return type is shown, followed by the symbol name prefixed with its package name, and parameter types in parenthesis. Subprogram **OPEN** is now displayed as **WINDOW WINDOWS::OPEN (Position, Height, Width)**. As for global subprograms, return type and/or parameter types may be empty.

You can see special subprograms in the Symbol List, which are not actually defined in the project. Have a look at item **~RANDOM_GENERIC**. There is no subprogram with this name in any of the projects. This "subprogram" is generated by the parser for each package. It always has the name of its package, prefixed with a tilde; it has no parameters and return type. This generated subprogram corresponds to the package initialization statements in the package body, which can be viewed as a kind of subprogram.

Double-click symbol **~RANDOM_GENERIC** in the Symbol Browser. A Source Editor comes up, file `random_generic.ads` is loaded and the keyword `PACKAGE` is highlighted. This is the synthesized position of the generated initialization procedure specification.

In the Source Editor, choose **Show > Implementation of ~RANDOM_GENERIC**. File `random_generic.adb` is loaded and the keyword `BEGIN` is highlighted. This is the synthesized position of the body of the generated procedure.

All packages have the specification of the generated procedure. However, body of this procedure exist only for those packages, which have package initialization. For example, if you select and double-click **~SCREEN** in the Symbol Browser, `screen.ads` will be loaded and the keyword `PACKAGE` will be highlighted. But, as there is no package initialization for package `SCREEN`, the **Implementation of...** option is disabled.

pkg object

When you select this symbol type, package objects are displayed. Symbols appear in the list similarly to package subprograms, together with their defining package.

typedef

When you select this symbol type, type definitions, subtypes and derived types are shown. Enumeration types are not displayed here (see `enum`). Enabling **Signature** lets you see packages containing type definitions, and signatures can also be seen in parenthesis. For example, you can see **SCREEN::HEIGHT (_sub_INTEGER)**. This means that **HEIGHT** is defined in package **SCREEN** and it is a subtype of **INTEGER**.

The following abbreviations are used in type signatures:

- **_sub_subtype**
- **_der_derived-type**
- **_acc_access-type**
- **_IntegerType_integral-type (RANGE)**
- **_RealType_real-type (DIGITS)**
- **_RecordType_record-type**

Array types has the signature **ARRAY(#, ...) OF ...**, where the number of hashmarks corresponds to the dimension of the array. In case of array of arrays, only the first set of dimension is displayed. For example:

- `ARRAY (1..5) OF BOOL` is displayed as **ARRAY (#) OF BOOL**,
- `ARRAY (1..5) OF STRING (1..20)` is displayed as **ARRAY (#) OF STRING**.

Note that as records in Ada always belong to a type declaration, they appear twice, both among typedefs and packages.

label

When you select this symbol type, all labels defined in the project are displayed in the Symbol List window.

Note

Labels are only shown in the **Symbol Type** drop-down if they exist in the project.

3

Follow Cross References

In this chapter, you will learn how to use the Cross Referencer to follow references in your source code.

With the Cross Referencer, you can answer questions like “What subprograms are called from this function” or “Where this package is referred to by” or “Where this object’s value is reassigned?”. The received cross-reference information can be filtered and refined in many ways.

Note

- To avoid overwhelming the user with unimportant reference information, local references are not displayed in SNIFF+. That is, if a symbol is referenced in the same scope it was defined, it will not be shown in the Cross Referencer.
- As package subprograms and objects are considered as bearing the highest importance, references to them are always displayed (even if local).
- References displayed in SNIFF+ can appear only in subprogram bodies. References in any other places are discarded.

Cross referencing

1. In the Source Editor, make sure that entry `REPORT_STATE` of task `MAITRE_D` is selected in either the Symbol List or in the Text View. (Remember that as tasks are mapped to packages and entries to package subprograms, it can be found under the **pkg**

subprog def in the Symbol Browser's **Symbol Type** drop-down). Then, choose **Info > REPORT_STATE Refers-To**.

The Cross Referencer appears. Entry **REPORT_STATE** is shown in it with all the symbols it refers to (19 nodes). See also [Abbreviations used in Cross Referencer — page 16](#).

The screenshot shows the 'CR: diners.shared - skogler PWE:AdaTest' window. The 'Language' is set to 'Ada9K'. The 'Filters...' button and 'Depth 1' are visible. The main pane displays a tree of symbols, with 'ps ROOM-MAITRE_D::REPORT_STATE' selected. The right pane shows the 'Root Symbol' list, including 'ROOM-MAITRE_D::REPORT_STA'. The bottom pane shows the code for the 'select' block, with 'accept Report State' highlighted. The status bar at the bottom indicates 'Frozen Nodes: 19', 'Matches: 12', and 'Cached Files: 3'.

Language: Ada9K Filters... Depth 1

Root Symbol

ROOM-MAITRE_D::REPORT_STA

DONE_EATING (ei) PHIL

DYING (ei) PHIL

EATING (ei) PHIL

GOT_ONE_STICK (ei) PHIL

GOT_OTHER_STICK (ei) PHIL

NAME_REGISTER (po) SOCIETY

NEW_LINE (ps) WINDOWS

PHIL (pk)

PUT (ps) WINDOWS

ROOM-MAITRE_D::REPORT_S

SOCIETY (pk)

THINKING (ei) PHIL

TITLE (ps) WINDOWS

UNIQUE_DNA_CODES (td) SOCIETY

WINDOWS (pk)

Projects Full Tree

diners.shared ()

random_generic.shared (r)

room.shared (room)

File: room.adb - C:/Ada_example/diners

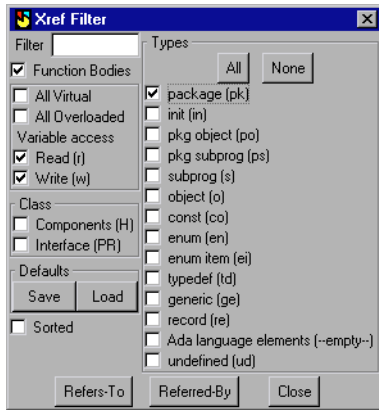
```

select
  accept Report State (Which_Phil : in Society.Unique_DNA_Codes;
    State : in Phil.States;
    How_Long : in Natural := 0;
    Which_Meal : in Natural := 0) do
    ...
  end accept;

```

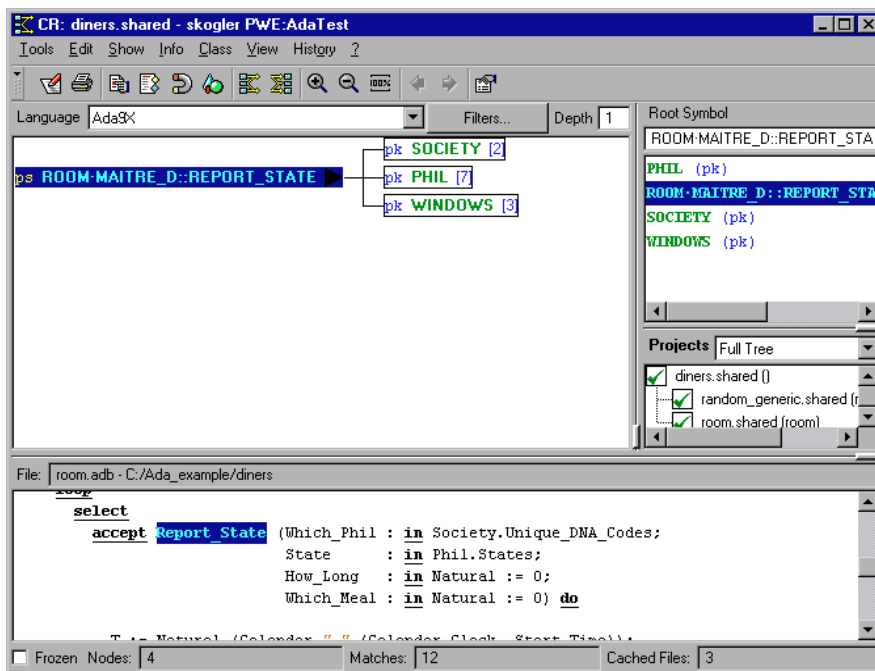
Frozen Nodes: 19 Matches: 12 Cached Files: 3

- Limit the scope of the next query to packages. Press the **Filters...** button. Press the **None** button under **Types** and then select the **package (pk)** checkbox.



- Now select **ps MAITRE_D::REPORT_STATE** in the Cross Referencer's **Graph** view and press the **Refers-To** button in the Filter dialog.

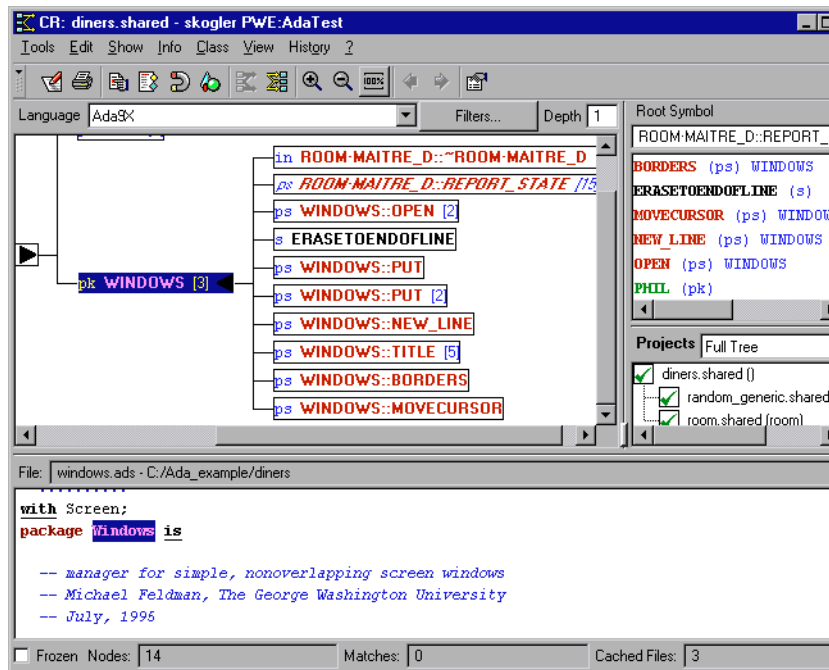
You should now see a call tree similar to the following:



- Find package **WINDOWS** in the referenced symbols list.

Number 3 in brackets following the name means that there are 3 references in this entry to package **WINDOWS**.

- Click on **pk WINDOWS** to highlight it. Choose **Context menu > Show Reference**. The Source Editor highlights the first reference of package **WINDOWS**: it is an implicit reference through its subprogram **TITLE**.
- In the Source Editor choose **Show > Next Match** to jump the position of the next reference.
- Switch back to the Cross Referencer. Let's see what subprograms (both global and package) package **WINDOWS** is referred to by. The Filter dialog should still be open. Press the **None** button and then select the **subprog (s)** and **pkg subprog (ps)** checkboxes. Press the **Referred-By** button.



There are 10 subprograms that refer to **WINDOWS**. One of them, **MAITRE_D::REPORT_STATE** is in italic. This means that this reference is already displayed in the Cross Referencer.

- Close the Cross Referencer, the Symbol Browser and Source Editor tools.

Abbreviations used in Cross Referencer

Symbol types that can be referred by a subprogram are listed in the following table. The table also contains abbreviations for the symbol types used in the Cross Referencer.

Symbol	Abbreviation
package	pk
package object	pv
package subprogram	ps
object	v
subprogram	s
enumeration type declaration	en
type declaration	td
record	re
undefined reference	ud

Abbreviations for symbol types are also displayed in the Filter dialog.

Cross Referencing undefined symbols

Only SNIFF+ symbols (symbols residing in the Symbol Table and shown in the Symbol Browser) can be cross referenced. These symbols must be defined in files in the Project Tree. If a symbol is outside of the files of a given project and its subprojects, it is inaccessible to SNIFF+, and will be displayed as undefined. Undefined references are always shown in the Cross Referencer. An example is CALENDAR, which is, though WITH-ed and may be available to the compiler, not included in the project.

4

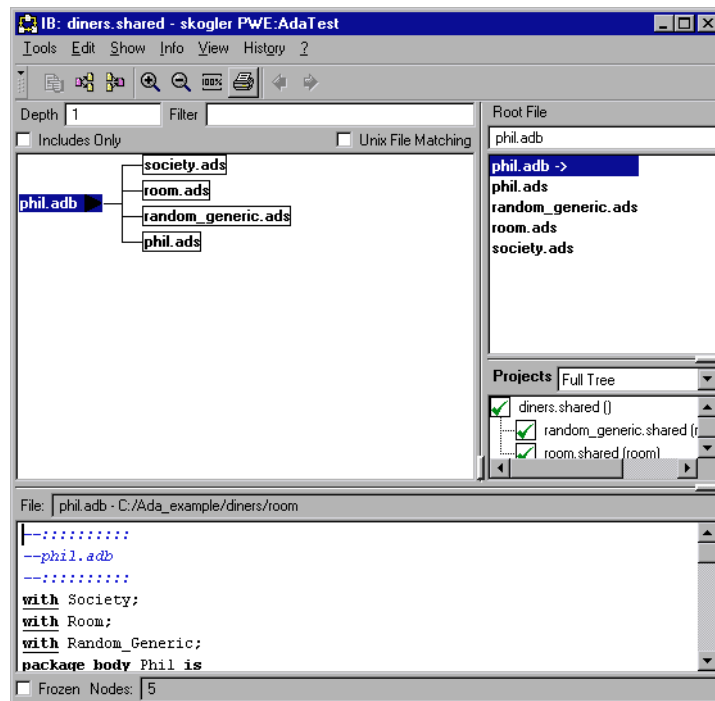
Browsing examples

In this chapter, you will learn how to browse a variety of Ada symbols, as well as follow include statements in your source files.

Also the Hierarchy Browser and the Retriever are used in this chapter.

Browsing Includes

1. In the Project Editor, select the file **phil.adb**.
2. Choose **Info > phil.adb Includes**.
3. The Include Browser appears. You can see which files are explicitly or implicitly included in this file.



4. Select **society.ads**. Choose **Context menu > Show Include Statement**.

The Source Editor appears, and the statement `WITH SOCIETY` is highlighted. As you can see, the parser kept track of in which source file was the package `SOCIETY` defined. As the source explicitly named the package through a `WITH` clause, this is an explicit include.

5. Back in the Include Browser, select **phil.ads** and right-click to show its include statement.

In the Source Editor the statement `PACKAGE BODY PHIL` is highlighted. For semantically correct information, at this point the package specification must be read in, though it was not specified with a `WITH` clause. This is called implicit include.

6. Back in the Include Browser, select **random_generic.ads**. Choose **Context menu > Included By** to see which other source files include this one.

You can see that two files include **random_generic.ads**: **phil.adb** and **random_generic.adb**. **phil.adb** is displayed in italic, showing that this include relationship is already displayed.

7. Our interest is now on **random_generic.adb**. Select it and choose **Context menu > Start from random_generic.adb**.

random_generic.adb becomes the root. Currently, only this file is displayed in the Include Browser.

8. Let's see which files are included by **random_generic.adb**.

Two additional nodes are displayed: **random_generic.ads** and **ADA.NUMERICS.DISCRETE_RANDOM**.

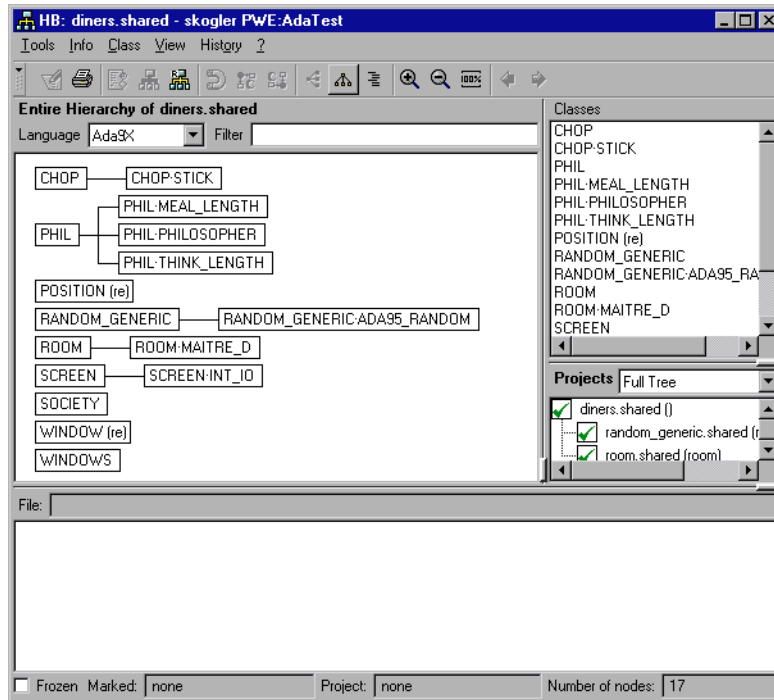
As the package `ADA.NUMERICS.DISCRETE_RANDOM` is not included in the project, the parser was unable to locate the file in which it is defined. For this reason, instead of the file name, the referenced package name was displayed in the Include Browser. Note that the name of the package is in normal typeface, while file names that were found by the parser, are bold.

9. Close the Source Editor and Include Browser tools.

Hierarchy Browsing

1. In the Source Editor choose **Tools > Hierarchy Browser**.

All packages, records, protected types, tasks and task types are displayed in a tree.



Language constructs defined within another one, are displayed as a branch of their surrounding construct. Double-click on **CHOP.STICK** to see that the protected type **STICK** is defined within package **CHOP**.

2. You can restrict displayed symbols to particular projects. In the Project Tree window of the Hierarchy Browser select **room.shared**. Choose **Context menu > Select From room.shared only**.

Now the display shows information only from that project.

3. Close the Hierarchy Browser and Source Editor tools

Note

See [Parser Options — page 22](#) to see how hierarchy relationship display can be altered by parser switches.

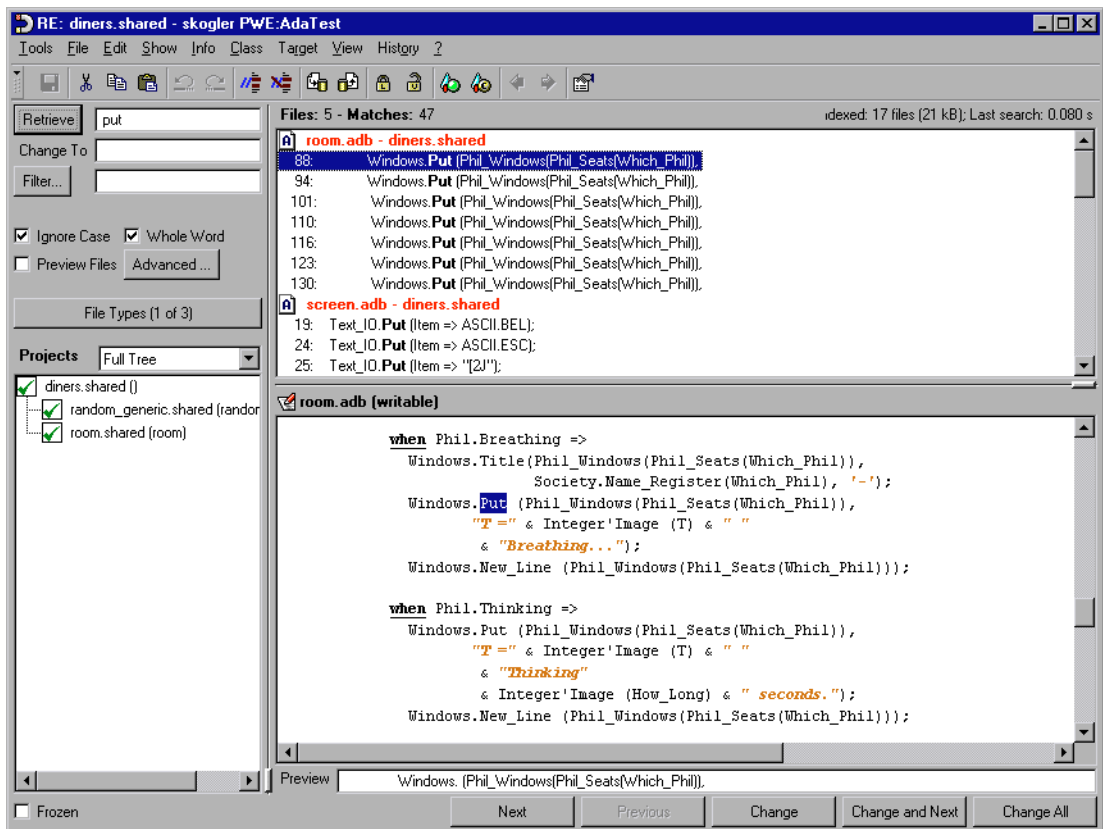
For embedded packages the parser generates unique symbol names for the contained element. For example, in the above figure you can see protected type **STICK** appearing as **CHOP.STICK**.

Using the Retriever

The Retriever is a text filtering tool, allowing you to find fragments of text using regular expressions in your source.

1. In the Source Editor choose **Tools > Retriever**.
2. Make sure that all projects are checkmarked in the Project Tree window of the Retriever.
3. Enter the word `put` right to the **Retrieve** button.
4. Checkmark **Ignore Case** and **Whole Word**.
5. Press **Retrieve**.

A list of source lines appears. The word `put` is shown in bold.



6. Double-click any of the lines.

A Source Editor appears and the cursor is positioned to the word `put`.

7. In the Source Editor choose **Show > Symbol(s) PUT...**

The Choose Symbol dialog appears. The dialog appears whenever SNIFF+ finds more than one symbol of the same name that matches a symbol request, or when multiple matches are found after **Show > Symbol(s) symbol...** is executed.

8. In the Choose Symbol dialog, switch on the **Show listing of files** button. This enables you to see the file names and projects in which the symbol appears.
9. Select the first entry and press the **Definition** button to jump to the procedure's definition.
10. By following the steps outlined above, you can quickly jump to any symbol without have to scan through your source code.
11. Close the Source Editor and Retriever tools.

5

Parser Options

In this chapter, you will learn how to configure a number of parser options for parsing Ada code.

The Ada Parser allows you configure certain parser options using environment variables, some of them by Project Attributes; some options are affected by both. These options are discussed in detail below.

Environment Parser Options

There are two environment variables that the Ada Parser reads. These environment variables has effect on all projects currently open in SNIFF+.

- `SNIFFADA_CACHE`

For improving performance of the parser, a symbol table cache is utilized. The number of files in the cache can be limited by setting `SNIFFADA_CACHE` environment variable. If not set, defaults to 1000.

Disabling the cache (setting `SNIFFADA_CACHE` to 0) will increase parsing time by up to 500%, depending on the project size.

- `SNIFFADA_NOBASE`

Ada package nesting is displayed in SNIFF+ as inheritance in the Hierarchy Browser by default. If this variable is set, the default behavior is not displaying nesting information.

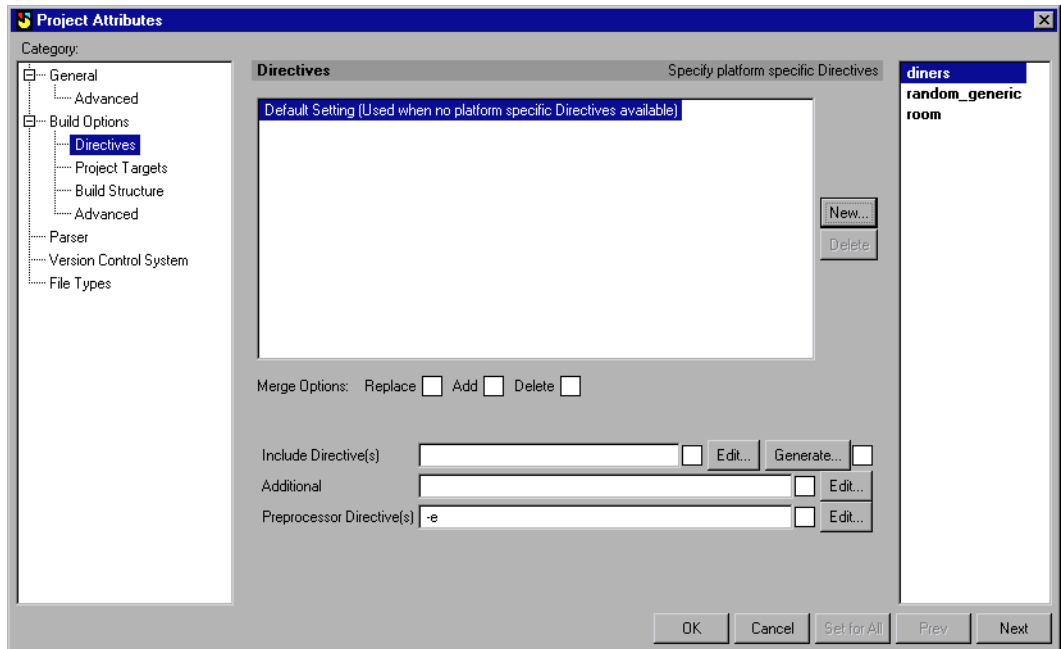
See project specific option `-b+` and `-b-` which override the default behavior below.

Project-specific Parser Options

In the Project Editor click anywhere in the Project Tree and choose **Context menu > Select from All Projects**. Then choose **Project > Attributes of Checkmarked Projects**.

- Under Build Options select the **Directives** node.

The **Preprocessor Directive(s)** field is available in this view. This field was originally designed for entering preprocessing directives for C/C++ projects. Limited capabilities of the C/C++ preprocessor is available in the Ada Parser (see -D options below). Also, this field can be used to set project-specific parser options. Any of these options affects only the project for which is set.



Note

The **Preprocessor Directive(s)** field (containing parser options) is always evaluated, regardless of whether **Preprocess before Source Code Parsing** checkbox (located in the Parser node) is selected or not.

Available options:

- -e

Log (syntax) errors in SNIFF+'s Log Window. It is recommended that this option is always set.

- `-nf`

Ignore forward declarations.

When both forward declaration and complete declaration is found, both is displayed in the Symbol Browser. Setting this option, only the complete declaration will be shown. If the complete declaration can not be found, the forward declaration will be displayed, regardless of this option.

- `-83`

Ada 83 mode. In this mode, the Ada Parser does not recognize Ada95-specific keywords as keywords, but treats them as identifiers. Using this option, you can parse Ada83 code, which contain Ada95 reserved words as identifiers (in the default Ada95 mode, it would be treated as syntax error). The keywords appearing only in Ada95 but not in Ada83 are:

```
ABSTRACT
ALIASED
PROTECTED
REQUEUE
TAGGED
UNTIL
```

- `-b+` and `-b-`

With these options, you can override the default behavior of displaying nesting in the Hierarchy Browser (see explanation for `SNIFFADA_NOBASE` in Environment Variables section above). `-b-` forces nesting information hiding, while `-b+` forces nesting information display.

- `-Dident`

The Ada Parser recognizes and processes `#ifdef`, `#ifndef`, `#else` and `#endif` C preprocessor directives. For correct preprocessing, symbolic names (macros) must be defined. `-D` directives specified here will be passed to the parser's preprocessor. Multiple `-D` directives are allowed. `#ifdef` nesting in the source code is allowed up to the nesting depth of 20.

To set the `-e` option:

- enter `-e` in the **Preprocessor Directive(s)** field.
- Select the checkbox right to the field.
- Press the **Set for All** button.
- Press **Ok**.

A dialog asking to update the makefiles appears. We will do this later so press **No**.

Save the whole project in the Launch Pad.

6

Building the Project's Executable

In this chapter, you will set up SNIFF+'s Make Support for the project and then build its executable.

Note

In order to complete the last section in this chapter you must have a GNAT Ada environment installed on your machine. SNIFF+'s Make Support creates search paths only for files which are actually in the project. Standard included files (the GNAT RTL) are usually not added to the SNIFF+ project, but must be available to GNAT when compiling and linking.

An environment variable `SNiFF_ADAMAKE` should be set to 1.

Setting up Make Support

Generating search paths

1. In the Project Tree of the Project Editor, make sure that all projects are checkmarked. Choose **Project > Attributes of Checkmarked Projects...** to open the Project Attributes dialog. In this dialog, you can look at and modify all the attributes of selected projects.
2. Select the Build Options - Directives node.
3. Select toggles left and right to the **Generate...** button.
You can use checkmarks to modify attributes of all projects simultaneously. With the checkmarks off, you can set attributes of the project which is highlighted in the Project List window on the right.
4. Press the **Generate...** button to generate the include paths for all projects. The project's include path information will then be displayed in the **Include Directive(s)** field (Ignore the warnings in the Log Window).
5. Press **Set for All** to apply changes to all projects (Again ignore the warnings in the Log window).
6. Press **Ok** to save the changes to the project attributes.
7. Two dialog boxes appear. Press **No** in both.

Setting up the main target for `diners.shared`

1. In the Project Tree of the Project Editor, double-click `diners.shared` to open its Project Attributes dialog.

2. In the **Executable** field of the Build Options-Project Targets node, enter a name for the project's executable. Using GNAT, this name must match the name of the source file which contains the project's main target (`diners`).
3. Please note that include path has already been generated.
4. Switch to the Build Options-Build Structure node.
5. Press the **Generate...** button next to the **Recursive Make Dirs** field at the bottom of the view.

The executable is built using recursive Make rules. By pressing the **Generate...** button, SNIFF+ generates the order of subprojects in which Make is executed.

Note

When using languages other than Ada, building an executable is usually done by specifying all the object files needed to build it. For this purpose, a project can have the **Passed to Superproject** attribute, which specifies which object files are to be included in the link.

In the GNAT Ada model, building the main target is performed by `gnatmake`, based on generated `.ali` files. Usage of `.ali` files obsoletes specifying object files to be passed to superproject. For all Ada projects, you can leave this field empty.

6. Press **Ok** to save the changes to the project attributes.
7. A dialog—asking to update the makefiles appears—press **Yes**.
8. In the Launch Pad, save the changes made to the Project Description Files of `diners.shared` and its subprojects.

Building the project target

You are now ready to build the executable. The steps outlined below are to be executed in the Project Editor.

1. Choose **Target -> Make... > diners** to build all the object files and the main target in the shared project.

A Shell opens, in which the `sniffmake diners` command is executed. Upon completion, you should have an executable named `diners` in `<diners>`.

2. Run the executable if you want. To do so, enter `diners` in the Shell, or choose **Target > Run diners**.

SNIFF+'s Shell does not have all the terminal capabilities which are required by `diners`.

3. Close the `diners.shared` project

Note

Due to differences between the C/C++ and GNAT library models the following specialities apply:

Within one project you can not mix Ada sources with other languages.

GNAT has very strict file naming conventions. The project's executable target must be named according to these rules; that is, if the main file is `'diners.adb'`, the executable must be named `'diners'`.

`gnatmake` supports executable targets only. Any other kind (relinkable object, library) must be made manually. As object sharing is not supported, the only helping target that applies is `'clean'`.

During `make` you will receive warnings about overwriting targets (e.g. Warning: overwriting rule `'diners'`), which is the normal behavior and can be ignored.

This concludes the tutorial on browsing Ada code.