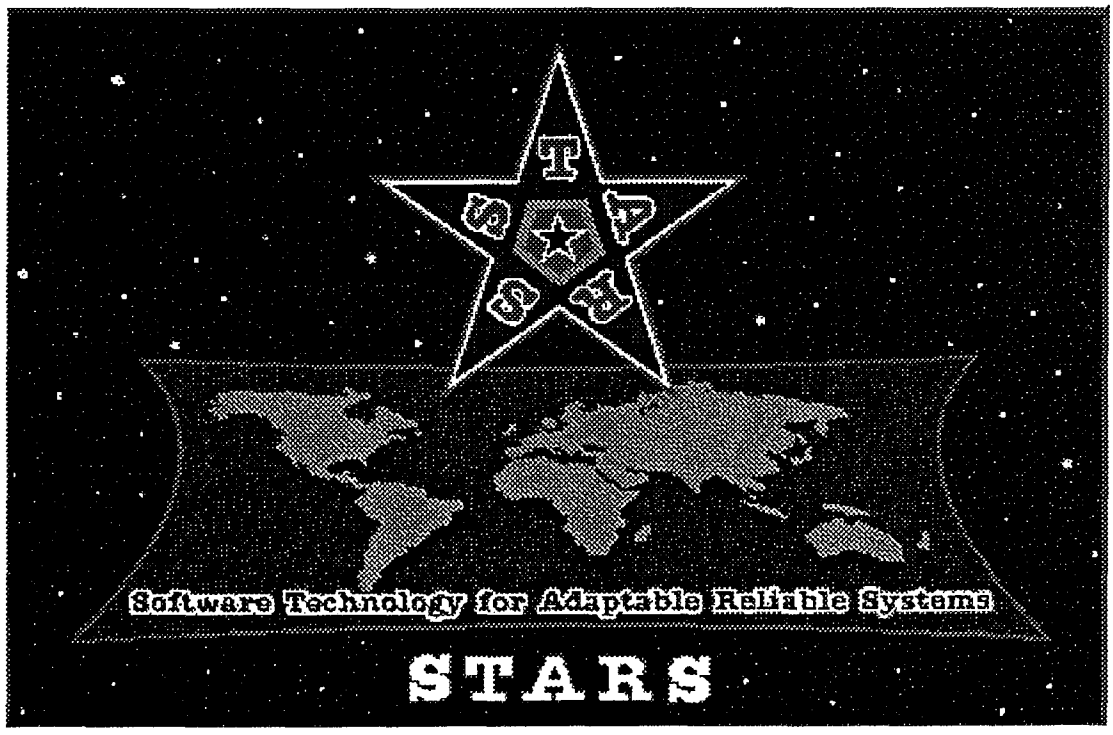
TASK: PA18
CDRL: C006
02 September 1994

AdaWise User's Manual
Alsys RISC Ada, Sun OS 4.1.2 Implementation

Informal Technical Data



STARS-AC-C006/001/01

REPORT DOCUMENTION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	02 September 1994	Informal Technical Report

4. TITLE AND SUBTITLE
AdaWise User's Manual

5. FUNDING NUMBERS
F19628-93-C-0130

6. AUTHOR(S)
C. A. Barbasch

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Unisys Corporation
12010 Sunrise Valley Drive
Reston, VA 22091-3499

8. PERFORMING ORGANIZATION REPORT NUMBER
CDRL NBR
STARS-AC-C006/001/01

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Department of the Air Force
ESC/ENS
Hanscom AFB, MA 01731-2816

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
C006

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution "A"

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Ada is a high-level language with features to support program reliability and maintenance. An Ada compiler will detect many potential programming errors or non-portabilities that other languages would require a separate analysis tool to detect. However, the Ada language definition includes rules that Ada programs are required to obey, but that compilers are not required to enforce, either at compile-time or at execution-time. An Ada program that violates one or more of these rules can have unpredictable behavior, or can have different effects with different compilers or in different execution environments.

14. SUBJECT TERMS

15. NUMBER OF PAGES
36

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

TASK: PA18
CDRL: C006
02 September 1994

INFORMAL TECHNICAL REPORT

For

SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS (STARS)

AdaWise User's Manual
Alsys RISCAda, Sun OS 4.1.2 Implementation

STARS-AC-C006/001/01
02 September 1994

CONTRACT NO. F19628-93-C-0130

Prepared for:

Electronic Systems Center
Air Force Materiel Command, USAF
Hanscom AFB, MA 01731-2816

Prepared by:

Odyssey Research Associates
under contract to
Unisys Corporation
12010 Sunrise Valley Drive
Reston, VA 22091

TASK: PA18
CDRL: C006
02 September 1994

Data Reference: STARS-AC-C006/001/01
INFORMAL TECHNICAL REPORT
AdaWise User's Manual
Alsys RISCAda, Sun OS 4.1.2 Implementation

Distribution Statement "A"
per DoD Directive 5230.24
Authorized for public release; Distribution is unlimited.

This document, developed under the Software Technology for Adaptable, Reliable Systems (STARS) program, is approved for release under Distribution "A" of the Scientific and Technical Information Program Classification Scheme (DoD Directive 5230.24) unless otherwise indicated. Sponsored by the U.S. Advanced Research Projects Agency (ARPA) under contract F19628-93-C-0130, the STARS program is supported by the military services, SEI, and MITRE, with the U.S. Air Force as the executive contracting agent. The information identified herein is subject to change. For further information, contact the authors at the following mailer address: delivery@stars.reston.paramax.com

TASK: PA18
CDRL: C006
02 September 1994

Data Reference: STARS-AC-C006/001/01
INFORMAL TECHNICAL REPORT
AdaWise User's Manual
Alsys RISCAda, Sun OS 4.1.2 Implementation

Principal Author(s):

Cheryl Barbasch *Date*

Dana Hartman *Date*

Approvals:

Teri F. Payton 9/6/94

Program Manager *Teri F. Payton* *Date*

(Signatures on File)

Preface

This manual describes how to use **AdaWise**, a set of program analysis tools for Ada code. Using **AdaWise**, a user can automatically check Ada code to detect potential errors related to compiler-dependent behavior and definedness.

The **AdaWise** tool set runs on Sun SPARCstations with the SunOS UNIX operating system. You will need the Alsys (TeleSoft) RISCAda/SPARC Development System^{TM}, Release 2, SunOS 4.1.2. We will refer to this version of the Ada compiler and associated files as *RISCAda* in the rest of the document.

AdaWise is written in Ada, using ASIS (Ada Semantic Interface Specification) for front-end static semantic analysis. A user of the tools must be able to compile the input source to be analyzed with a compiler that supports ASIS (as does the Alsys RISCAda compiler). However, while the tool set is built using ASIS, a user of the tools is *not* required to have the ASIS product itself to run the tools.

Whether you are a novice or an experienced Ada programmer, you can use the guidelines in this manual to check code. We assume that you have some practical experience with the kinds of errors that arise in programming and a working familiarity with Ada. We also assume that you are familiar with using the Alsys RISCAda compiler.

Organization of This Manual

Chapter 1 presents an overview of **AdaWise** and describes how to set up to use the tools, the user interface to the tools, and the command line arguments. Chapters 2, 3, and 4 describe the three **AdaWise** tools. In each of these chapters, we describe an **AdaWise** tool, show steps to run the tool on sample code provided for demonstration purposes, and show the tool's output for the sample code. Appendix A documents error messages produced by the **AdaWise** tools.

Conventions

In this manual, we use the following conventions.

- For command lines that you type verbatim, we use **boldface** type. Unless otherwise specified, all command lines are followed by a carriage return. We also use **boldface** type for program names, filenames, and pathnames.
- For command line variables that you choose, we use *italic* type.
- Brackets indicate an option in a command line.
- For prompts, output from the tools, and sample code, we use `typewriter` type.

When we refer to details of the Ada language, we cite the relevant section of the Ada Reference Manual [1], which we call the ARM.

Contents

Chapter 1

Introduction to AdaWise

Ada is a high-level language with features to support program reliability and maintenance. An Ada compiler will detect many potential programming errors or non-portabilities that other languages would require a separate analysis tool to detect. However, the Ada language definition includes rules that Ada programs are required to obey, but that compilers are not required to enforce, either at compile-time or at execution-time. An Ada program that violates one or more of these rules can have unpredictable behavior, or can have different effects with different compilers or in different execution environments.

AdaWise is a set of tools that will check Ada programs at compile-time for the absence of some of these errors. Input to the tools can be any legal Ada program; users are not restricted to a subset of Ada. The AdaWise tools perform the checks automatically, without requiring the user to write formal specifications or carry out proofs.

Although running the tools is automated, using them must be combined with human analysis. The warnings that the tools issue are *conservative*—that is, the absence of a warning guarantees the absence of a problem. If a tool issues a warning, it points out a specific place in the program in which a problem may occur. You then have a localized region in which to look for a potential violation.

The **AdaWise** tools currently under development check for improper aliasing and incorrect order dependences, non-portabilities in the order of elaborating compilation units, and use of undefined variables. You can apply the tools to programs of arbitrary size.

Currently, there are three tools in the tool set:

- **aiod** is a tool to check for potential erroneous execution as a result of aliasing of subprogram parameters and for potential incorrect order dependences.
- **elab** is a tool to check for potential errors caused by incorrect order dependence during elaboration of Ada units.
- **def** is a tool to check that variables are not read before they are written and that subprograms are not called before their bodies have been elaborated.

AdaWise consists of a single executable, **adawise**, for all of the tools. There are two ways to invoke **adawise**:

- Without command-line arguments, the **adawise** command displays a basic menu. Using the menu, you can enter an Ada unit to analyze, change defaults for **AdaWise** options, and run an **AdaWise** tool. The menu is provided to help new users and is primarily useful for small test programs.
- With command-line arguments to the **adawise** command, you can change defaults for **AdaWise** options and run multiple **AdaWise** tools on multiple Ada units in one execution in batch mode. Using **adawise** with command-line arguments is efficient when analyzing the units of a large software project.

1.1 Where AdaWise Fits In

In this section, we describe the steps for using **AdaWise**. In later sections, we explain how to set up the environment for using **AdaWise**, how to create an **AdaWise** configuration file (**aw.cfg**), and how to run the tools.

The **AdaWise** tools use information stored in a compiler's library to perform the analysis of source code. The first step to run **AdaWise** is to compile the source code to be checked using a compiler that allows **AdaWise** to access that information. You create a library following the compiler vendor's instructions, and you compile the source to be checked into that library. In order to provide all of the information in the library that **AdaWise** requires, you may have to compile with special options. For example, using the Alsys RISCAda compiler you must compile with the **-k** or **-d** option.

Your source code does not have to be linked into an executable and does not have to be able to execute. It needs only to compile cleanly.

After compiling the source code, the next step is to set up and run the **AdaWise** tools. You create an **AdaWise** configuration file (**aw.cfg**), which tells the tools to ignore sublibraries that you do not want to check (**aw.cfg** is also described in section 1.2). You can then run **AdaWise** on the Ada units that you want to test. **AdaWise**, using static semantic information produced by the compiler, performs its checks and reports potential errors in the source code. (Note: **AdaWise** uses ASIS for the static semantic analysis; while you do not need to have the ASIS product, the compiler that you use must support ASIS.) Since the tools are conservative, you should inspect the source to determine if there is a real error.

1.2 Getting Started with AdaWise

We assume that your system administrator has installed the delivered tape according to the installation instructions. Your system administrator should tell you where the **AdaWise** executables are located on your system so that you can add the **AdaWise bin** directory to your path. The **AdaWise** installation directory also contains subdirectories with documentation and sample Ada source files.

You also need the Alsys TeleSoft RISCAda/SPARC Development System^{TM}, Release 2, SunOS 4.1.2. (We will refer to this version of the Ada compiler and associated files as RISCAda in the rest of this document.)

Create the RISCAda library and environment as described in the RISCAda documentation. Once you have your path and environment set, change to the directory in which you plan to run **AdaWise**. In this directory, you need a RISCAda library listing file (by default called the **liblst.alb** file) that contains the names of the sublibraries that you are using, and you need to create your sublibrary. The following example creates a **liblst.alb** and a working sublibrary:

```
% cat > liblst.alb
name: mylib.sub
name: $TELEGEN2/lib/rt.sub
<crtl-d>
% acr mylib.sub
```

Note the order of the sublibraries: the working sublibrary is first; the RISCAda run-time sublibrary (**rt.sub**) must be the last one in the library file.

Next you compile the Ada source using the **-k** or **-d** option with the compiler's **ada** command. The option is necessary to provide the internal information needed by the **AdaWise** tools to do the analysis.

Before running the **AdaWise** tools, you should create a configuration file, called **aw.cfg**, in the directory in which you are running the tools. This file lists the sublibraries that contain the Ada units you want **AdaWise** to ignore.

We recommend that you always provide a configuration file that contains at least the compiler run-time units, since the compiler's internal Ada units typically will not contain the additional library information that we need to do the analysis. You may also have additional sublibraries that you do not want to analyze.

To create a default **aw.cfg** configuration file, run the **AdaWise** setup script by typing

setup-tools

If you use a different name for the library (**.alb**) file, you can use the **-l** argument to the **setup-tool** command to specify that file:

setup-tools [-l *your-lib-name*.**alb**]

The setup script assumes that the last sublibrary in the library file is the name of the RISCAda run-time sublibrary. For example, if the library file contains the sublibraries in the example **liblst.alb** above, the setup script creates the following **aw.cfg** file:

```
--:  AdaWise Configuration File for library: liblst.alb
--:      Strings beginning with '--' are comments

--:  Automatically generated configuration options below.

--: AdaWise tools will ignore TeleSoft internal Ada units
IgnoreSublib:  $TELEGEN2/lib/rt.sub
```

This default configuration file instructs **AdaWise** to ignore the units in the RISCAda **rt.sub** sublibrary when checking your compiled Ada source. If **AdaWise** is run without ignoring the units in **rt.sub**, the tools may exit abnormally, or they may execute but with misleading results.

Lines in the **aw.cfg** file consist of the string "IgnoreSublib: " followed by the pathname of the sublibrary to ignore. Each sublibrary name must be one of the sublibraries following "name: " in the library file, and it must appear exactly as it does in the library file (for example, if you use `$TELEGEN2` in your library file, you must use it, not the full path name, in **aw.cfg**). The configuration file can contain comments (strings beginning with ''--'') and whitespace (which is ignored). Also, the case of the letters in the string "ignoresublib" and the order in which the sublibraries appear in the file are not important.

You can edit the **aw.cfg** file to add other sublibraries that you want the tools to ignore. For example, if you call third party software that you do not want to check or that is inaccessible to **AdaWise**, and the software is in **sublib3.sub**, you can edit **aw.cfg** to add **sublib3.sub** to the list of sublibraries to ignore (using the same `IgnoreSublib:`*sublib-name* format):

```
IgnoreSublib:  $TELEGEN2/lib/rt.sub
```

```
IgnoreSublib:  sublib3.sub
```

You now have your working sublibrary, **.alb** file, and **aw.cfg** file. You can invoke **AdaWise** with or without command line arguments. If you do not use arguments, you get the **AdaWise** menu; using arguments, you bypass the menu.

1.2.1 Using the AdaWise Menu

The **AdaWise** menu is a very basic interface for the **AdaWise** tools. We expect that you may want to use this menu only to become familiar with the tools and with the defaults for the options. After you are familiar with **AdaWise** you will likely find it more efficient to use the command-line arguments.

To use the **AdaWise** menu, type **adawise**. The first output (as shown below) tells which sublibraries will be ignored based on your **aw.cfg** file.

```
Beginning AdaWise
Compiler Vendor: Alsys Inc.

-> *** The following TeleSoft sublibs will be ignored:
-> $TELEGEN2/lib/rt.sub

Input Library (CR defaults to liblst.alb):
```

You get a prompt for the Input Library. You can press the carriage return to select the default library (**liblst.alb**) or input the name of your **.alb** file.

AdaWise next shows the option defaults and current library, provides a menu of tools and control options, and gives a list of the options you can select:

```
Ada unit to check     :
Ada unit kind         : body
Verbose option        : no
Closure               : no
Output to             : standard out

-----------------------------------------------------------------
Current Open Library : liblst.alb
The list of AdaWise Tools:
    1    aiod:  Check Aliasing & Incorrect Order Dependence
    2    elab:  Check Elaboration Order
    3    def:   Check Definedness
  Control options:
    u    Ada unit to check
    k    Ada unit kind (body, spec)
    v    Set verbose or non-verbose
    c    Closure: run tool on all units in elaboration order
    r    Redirect output (file or standard out)
    q    Quit
-----------------------------------------------------------------

Choice? (u, k, v, c, r, q):
```

To enter a unit name, enter **u** at the `Choice?` prompt. You then see an `Input Ada unit:` prompt; at this prompt, enter the name of the Ada unit that you want to check. (You do not enter the name of the source file containing the unit.) The Ada unit must be one that was compiled into the working sublibrary listed in the **.alb** file. After you enter the unit name, you get a new menu with the unit name in the list at the top of the menu and with the tool selections in the `Choice?` prompt. Each time you change a control option, you get a new menu.

Using the **k** control option, you can change the unit kind, which is either specification or body. The default unit kind is body.

The **v** control option indicates whether the output is in verbose mode or non-verbose mode. The default is non-verbose mode. With non-verbose, you typically see a few notices (with a ** prefix) and warnings (with a >> `WARNING:` prefix). With the verbose mode, in addition to those notices and warnings, you see checkpoint notices (with a -- `Checking:` prefix) and additional information for some warnings (with a -> prefix). We describe the **AdaWise** messages in more detail in the chapters on the individual tools and in the appendix.

The **c** control option is the closure option and is for only the **aiod** tool. With this option, the **aiod** tool checks all units required by the context clauses of the input unit, and recursively all the units they require. If you use the closure option, you provide a single Ada unit name, and **aiod** will check each Ada unit in the with-closure. This option is provided as an easy

way for the user to input a potentially large number of compilation units to check with **aiod**. The option is ignored by the **def** and **elab** tools. The default is no closure.

If you enter **r**, you can choose to have output directed to your screen (the default) or to have the output directed to a file. You can keep the default by pressing the carriage return, or you can direct the output to a file by giving a filename. As the prompt message points out, **AdaWise** will overwrite a file if it already exists. If you redirect output to a file, the new menu shows the filename before the `Choice?` prompt:

```
Output will be redirected to file: output-filename
Choice? (1, 2, 3, u, k, v, c, r, q):
```

To run one of the tools, you enter the number for the tool at the `Choice?` prompt. By entering **1**, you start the **aiod** tool, which checks for aliasing errors and incorrect order dependencies. The **elab** tool (**2**) assumes that the input Ada unit is a valid main program and guarantees that all legal elaborations of that program have the same effect. The **def** tool (**3**) assumes that the input Ada unit has no aliasing errors and that all legal elaborations of the program have the same effect; therefore, to avoid misleading results from the **def** tools, you should run the **elab** and **aiod** tools on the Ada unit(s) before running the **def** tool. The **def** tool checks for undefined scalar variables.

After a tool has finished, **AdaWise** presents the menu again. You can then select **u** to enter another Ada unit name and select a tool to run on that unit. (See the appropriate chapter for the examples and output for each tool.)

The last choice in the **adawise** menu is the **q** control option to quit using **AdaWise**.

1.2.2 Using AdaWise Command Line Arguments

All of the arguments to the **adawise** command are optional (without arguments, you get the **AdaWise** menu). The **adawise** command with its arguments has the following format:

adawise [**tools**='*list of tools*' **units**='*list of Ada units*' **kind**=**body**|**spec**
verbose=**no**|**yes** **closure**=**no**|**yes** **lib**=*library-name*]

where

- for the **tools** argument, you can choose one of the three tools (**aiod**, **elab**, **def**), or you can list tools enclosed in either single or double quotation marks with white space separating the tool names; a single tool name does not require quotation marks; there is no default;

- for the **units** argument, you provide an Ada unit name, or you list the Ada unit names enclosed in either single or double quotation marks with white space separating the unit names; a single unit name does not require quotation marks; there is no default;
- for the **kind** argument, you choose either body or spec; this kind holds for all units specified with the **units** argument; the default is body;
- for the **verbose** argument, you choose yes or no, with no being the default;
- for the **closure** argument, you choose yes or no, with no being the default;
- for the **lib** argument, you give the **.alb** file, with the default being **liblst.alb**.

The arguments can be in any order. Also, you can abbreviate the arguments and variables as follows:

adawise [t='*list of tools*' u='*list of ada units*' k=b|s v=n|y c=n|y
l=*library-name*]

If you list multiple tools and multiple units on the command line, **AdaWise** calls each tool to analyze each unit, taking defaults if necessary for the other arguments.

If you specify a tool but no unit, **AdaWise** prompts for a unit name and executes the tool. If you specify multiple tools but no unit, **AdaWise** prompts for a unit, then runs each tool on that unit.

If you specify a unit or units but no tool, **AdaWise** prompts for the tool, taking defaults for the other arguments if necessary, and executes the tool on each of the units.

Chapters 2, 3, and 4 describe the **AdaWise** tools, give sample Ada code and **AdaWise** commands to check the code, and show the **AdaWise** output.

Chapter 2

The aiod Tool

2.1 Aliasing and Incorrect Order Dependence Errors in Ada

Two program variables are aliased if their storage overlaps, so that modifying one of the variables may affect the value of the other. Unintentional or improper aliasing is a well known source of programming errors. Two categories of errors that can result from improper aliasing are erroneous execution and incorrect order dependence (see ARM 1.6). A compiler is not required to recognize these errors.

Erroneous execution can arise because the mechanism for passing parameters is not completely specified by the language rules for all variable types. (See ARM, sections 1.6, 6.2(13), 6.4, and 12.3(17).) Array, record, and task types may be passed either by reference or by copy, as the compiler implementation chooses. When actual parameters to a subprogram or an entry call are aliased, the results produced by the subsequent processing can depend upon the parameter passing mechanism chosen by the compiler. If so, the execution is erroneous and its effect is undefined.

Incorrect order dependence arises because the compiler implementation is allowed to execute different parts of some constructs in any given order. For example, the order of evaluation of actual subprogram or entry call parameters is not defined by the language rules (see ARM 6.4). In the case of two output parameters, if the actual parameters are aliased and the result of the subprogram call depends upon the order of copy back to the actual parameters, then there is an incorrect order dependence.

Other constructs in which an incorrect order dependence can occur include default initializations, subtype range evaluation, constrained array elaboration, indexed components, record aggregates, expression evaluation, assignment statements, and generic instantiations.

Roughly twenty sections in the ARM discuss and define the incorrect order dependences that can occur in a program.

References [2] and [3] enumerate and discuss the sections of the ARM that address incorrect

order dependences.

2.2 What aiod Checks

The **aiod** tool checks for potential aliasing that could result in erroneous execution or incorrect order dependences. To check for possible errors resulting from aliasing, the **aiod** tool checks the actual parameters of subprograms and generic instantiations (ARM 6.2(13), 6.4, and 12.3(17)). Specifically, **aiod** finds all subprogram calls and generic instantiations in a given compilation unit and checks the actual parameters (depending on mode and type) for potential aliasing errors.

The following three checks are performed on the actual parameters for each subprogram call:

- Check each non-scalar, non-access type actual parameter for aliasing with a global variable. If an IN parameter is aliased with a variable that is updated in the body of the subprogram, then the call is flagged for potential erroneous execution. If an output parameter is aliased with a variable that is referenced or updated in the body of the subprogram then the call is flagged for potential erroneous execution.
- Check pairwise aliasing, except both mode IN. If two actual parameters are aliased and are non-scalar, then the call is flagged for potential erroneous execution. If two output parameters are aliased (including scalar variables) then the call is flagged for potential incorrect order dependence (copy back).
- Check pairwise independence. **AdaWise** calculates the set of objects changed (Lvals) and the set of objects referenced(Rvals) in evaluating each actual expression. If the Lvals of either intersects the Rvals or the Lvals of the other, then the call is flagged for incorrect order dependence. If the actuals are both input parameters, there could be incorrect order of copy in. For output parameters, there could be incorrect order of evaluating the addresses of the output variables.

The **aiod** tool also checks for incorrect order dependence in the following constructs: assignment statements (ARM 5.2(3)), discriminants (3.2.1, 3.7.1), record types (3.2.1(15)), discriminant constraints (3.7.2(13)), ranges (3.5(5) or 3.6(10)), indexed components (4.1.1(4)), aggregates ((4.3.1(3), 4.3.2(10)), index constraints (3.6(10)), and operands of an operator (4.5(5)).

The **aiod** tool is conservative in its warning messages: If no warnings are issued by the tool, then neither the choice of parameter passing mechanism nor the order of parameter evaluation by the compiler can affect the visible behavior of the program during execution (assuming the call does not propagate an exception), and there are no incorrect order dependences in the constructs enumerated above. If warnings are issued, the flagged errors may or may not be actual errors (i.e., some correct programs may be marked as having errors).

2.3 The aliased Sample Unit

The following sample Ada code illustrates an error that results from aliasing.

This unit illustrates the cases of aliasing between pairs of parameters and between a parameter and a global variable. This aliasing can cause erroneous execution, resulting from the Ada parameter passing mechanism (ARM 6.2). That is, if the compiler passes non-scalar parameters by copy, the resulting executable will produce different results than it would if the compiler were to pass the parameters by reference.

2.4 Using aiod on the Sample Unit

To run **aiod** on the sample unit, you can use the **adawise** menu or command line arguments as described in Chapter 1.

Here is a command line for checking the sample code with the **aiod** tool:

adawise tools=aiod units=aliased

This command line takes the defaults for the other arguments (i.e., the unit is an Ada body, the mode is non-verbose, and the library file is **liblst.alb**).

This command produces the following output:

```
Beginning AdaWise
Compiler Vendor: Alsys Inc.

-> *** The following TeleSoft sublibs will be ignored:
-> $TELEGEN2/lib/rt.sub


**
** Executing Tool: aiod   Ada unit: aliased (body)
**


>> WARNING: potential parameter error in
>> aliased: line  35: P( B, B )

>>   Parameters  1 and  2 are potential ALIASES
>>  (potential ERRONEOUS EXECUTION)

>> WARNING: potential parameter error in
>> aliased: line  38: P( ALIASED.A, B )

>>   Parameter  1: is aliased with a global.
```

```
procedure ALIASED is

    type TABLE is array(1 .. 10) of integer;

    A: TABLE := (others => 0);

    function F  return TABLE is
    --------------------------
    begin
        A := (others => 1);
        return A;
    end F;

    procedure P (X: in TABLE;  Y: out TABLE) is
    -------------------------------------------
    Z: TABLE;
    begin
        Z := F;          -- note: call to F modifies global A
        Y := Z;          --       writes Y,
        Y(5) := X(5);    --       then reads X
    end P;

    procedure Ecall is
    ------------------
    A, B: TABLE := (others => 2);
    begin

        P( B, B );              -- if pass-by-copy, B(5)=2,
                                --    else by reference:  B(5) = 1

        P( ALIASED.A, B );      -- if pass-by-copy, B(5)=0,
                                --    else by reference:  B(5) = 1

        P( B, ALIASED.A );      -- no actual error, but conservative warning

        P( A, B);               -- no aliasing error

    end Ecall;

begin
        Ecall;
end ALIASED;
```

```
>>  (potential ERRONEOUS EXECUTION)

>> WARNING: potential parameter error in
>> aliased: line  41: P( B, ALIASED.A )

>>   Parameter  2: is aliased with a global.
>>  (potential ERRONEOUS EXECUTION)

**
** aiod Finished
**
```

If we use the verbose mode, we get the following output:

```
Beginning AdaWise
Compiler Vendor: Alsys Inc.

-> *** The following TeleSoft sublibs will be ignored:
-> $TELEGEN2/lib/rt.sub

ASIS version:   1.1.0.2
Vendor version: 1.1.0.2
Vendor info:    1.1.0.2

-> Verbose Option set

-> Opening library: liblst.alb

-> Unit: aliased (body)

**
** Executing Tool: aiod   Ada unit: aliased (body)
**
5/6/1994 12:22: 5.26719E+01

-- Checking: in range (3.5(5))
-> aliased: line  10: 1 .. 10

-- Checking: in aggregate (4.3.1(3), 4.3.2(10))
-> aliased: line  12: (others => 0)

-- Checking: in assignment statement (5.2(3))
-> aliased: line  17: A := (others => 1)

-- Checking: in aggregate (4.3.1(3), 4.3.2(10))
-> aliased: line  17: (others => 1)

-- Checking: in assignment statement (5.2(3))
-> aliased: line  25: Z := F

-- Checking: FUNCTION CALL
-> aliased: line  25: F
```

```
->   No parameters to check

-- Checking: in assignment statement (5.2(3))
-> aliased: line  26: Y := Z

-- Checking: in assignment statement (5.2(3))
-> aliased: line  27: Y(5) := X(5)

-- Checking: in indexed component (4.1.1(4))
-> aliased: line  27: Y(5)

-- Checking: in indexed component (4.1.1(4))
-> aliased: line  27: X(5)

-- Checking: in aggregate (4.3.1(3), 4.3.2(10))
-> aliased: line  32: (others => 2)

-- Checking: in aggregate (4.3.1(3), 4.3.2(10))
-> aliased: line  32: (others => 2)

-- Checking: PROCEDURE CALL
-> aliased: line  35: P( B, B )

-- Checking: Each parameter for global aliasing
->   Parameter  1: not aliased with a global
->   Parameter  2: not aliased with a global
-- Checking: Parameter pairs for independence
->   Parameters  1 and  2 are independent.
-- Checking: Parameter pairs for aliasing (except both IN)

>> WARNING: potential parameter error in
>> aliased: line  35: P( B, B )

>>   Parameters  1 and  2 are potential ALIASES
>>  (potential ERRONEOUS EXECUTION)

-- Checking: PROCEDURE CALL
-> aliased: line  38: P( ALIASED.A, B )

-- Checking: Each parameter for global aliasing

>> WARNING: potential parameter error in
>> aliased: line  38: P( ALIASED.A, B )

>>   Parameter  1: is aliased with a global.
>>  (potential ERRONEOUS EXECUTION)
->   Parameter  2: not aliased with a global
-- Checking: Parameter pairs for independence
->   Parameters  1 and  2 are independent.
-- Checking: Parameter pairs for aliasing (except both IN)
->   Parameters  1 and  2 are not aliases.

-- Checking: PROCEDURE CALL
```

```
-> aliased: line  41: P( B, ALIASED.A )

-- Checking: Each parameter for global aliasing
->   Parameter  1: not aliased with a global

>> WARNING: potential parameter error in
>> aliased: line  41: P( B, ALIASED.A )

>>   Parameter  2: is aliased with a global.
>>  (potential ERRONEOUS EXECUTION)
-- Checking: Parameter pairs for independence
->   Parameters  1 and  2 are independent.
-- Checking: Parameter pairs for aliasing (except both IN)
->   Parameters  1 and  2 are not aliases.

-- Checking: PROCEDURE CALL
-> aliased: line  43: P( A, B)

-- Checking: Each parameter for global aliasing
->   Parameter  1: not aliased with a global
->   Parameter  2: not aliased with a global
-- Checking: Parameter pairs for independence
->   Parameters  1 and  2 are independent.
-- Checking: Parameter pairs for aliasing (except both IN)
->   Parameters  1 and  2 are not aliases.

-- Checking: PROCEDURE CALL
-> aliased: line  48: Ecall

->   No parameters to check

**
** aiod Finished
**
5/6/1994 12:22: 5.91914E+01
```

The verbose option shows each call or parameter that is being checked and its line in the unit file, each check being performed, and the results of each check. Checks that do not result in error detection are prefixed by "->", and checks that result in an error being detected are prefixed by ">>".

As described in Chapter 1, if we list multiple units on the command line, **AdaWise** calls the tool to analyze each unit. For example, if we have the **aliased** unit and a unit called **copy_back**, we can use the command line

adawise u='aliased copy_back' t=aiod

to run **aiod** first on the **aliased** unit and then on the **copy_back** unit, which gives the

following output:

```
Beginning AdaWise
Compiler Vendor: Alsys Inc.

-> *** The following TeleSoft sublibs will be ignored:
-> $TELEGEN2/lib/rt.sub

**
** Executing Tool: aiod   Ada unit: aliased (body)
**

>> WARNING: potential parameter error in
>> aliased: line  35: P( B, B )

>>   Parameters  1 and  2 are potential ALIASES
>>  (potential ERRONEOUS EXECUTION)

>> WARNING: potential parameter error in
>> aliased: line  38: P( ALIASED.A, B )

>>   Parameter  1: is aliased with a global.
>>  (potential ERRONEOUS EXECUTION)

>> WARNING: potential parameter error in
>> aliased: line  41: P( B, ALIASED.A )

>>   Parameter  2: is aliased with a global.
>>  (potential ERRONEOUS EXECUTION)

**
** aiod Finished
**

**
** Executing Tool: aiod   Ada unit: copy_back (body)
**

>> WARNING: potential parameter error in
>> copy_back: line  35: SWAP ( i, i)

>>   Parameters  1 and  2 are potential ALIASES
>>  (potential ORDER of COPY OUT error)

>> WARNING: potential parameter error in
>> copy_back: line  36: PROC( i, i)

>>   Parameters  1 and  2 are potential ALIASES
>>  (potential ORDER of COPY OUT error)

**
** aiod Finished
**
```

Chapter 3

The elab Tool

3.1 Elaboration Order in Ada

The execution of an Ada program begins with the elaboration of all compilation units needed by the main program unit. A *compilation unit* is a declaration or body that can be compiled separately. It can be a subprogram declaration or body, a package declaration or body, a generic declaration or body, a generic instantiation, or a subunit (ARM 10.1).

In general, a program's compilation units may be elaborated in more than one order. Chapter 10 of the ARM constrains the possible orders; any order meeting these constraints is legal. If there are two different legal elaboration orders that have different observable effects, the program has an incorrect order dependence. If no order satisfies the constraints, the program is illegal. Run-time errors occur if an attempt is made to use an entity when a required body has not been elaborated (ARM 10.5(4) and 3.9(8)).

3.2 What elab Checks

The **elab** tool analyzes all the compilation units needed by an Ada main program for incorrect order dependence during elaboration. If two compilation units are not related in the partial order defined by the with clauses, then **elab** checks the pair for independence. If an object could be updated during elaboration of one of the units and could be read or updated by the other, then the units are flagged for potential incorrect order dependence. If a subprogram could be called during elaboration of one of the units, and the subprogram body is defined in the other, then the units are flagged potential incorrect order dependence.

If no potential errors are reported by the tool for the program, then any legal elaboration order can be chosen by the compiler (or by another program analysis tool) and be guaranteed to produce the same visible result according to the Ada semantics.

Other types of error will not be guaranteed to be absent, though other errors may be found

as a result of an incorrect order dependence during elaboration. For example, programs that raise PROGRAM_ERROR or cause erroneous execution in some elaboration order will be detected by **elab** only if there is another legal elaboration order that produces a different result.

3.3 The elab_prog_err Sample Unit

The following sample Ada code illustrates an error.

```
package elab_prog_err_A is
        I: integer;
        function F return integer;
end elab_prog_err_A;

package body elab_prog_err_A is
        function F return integer is begin
          return 1;
        end F;
end elab_prog_err_A;

with elab_prog_err_A;
package elab_prog_err_B is
        J: integer := elab_prog_err_A.F;
end elab_prog_err_B;

package body elab_prog_err_B is
begin
        elab_prog_err_A.I := 0;
end elab_prog_err_B;

with elab_prog_err_A;
package elab_prog_err_C is
        K: integer := elab_prog_err_A.I;
end elab_prog_err_C;

with elab_prog_err_B, elab_prog_err_C;
procedure elab_prog_err_MAIN is
x, y: integer;
begin
        x := elab_prog_err_B.J;
        y := elab_prog_err_C.K;
end elab_prog_err_MAIN;
```

In this unit, the incorrect order dependence produces a legal elaboration order that causes one of the other categories of error: PROGRAM_ERROR is raised if the body of **elab_prog_err_A** is not elaborated before package **elab_prog_err_B**. If **elab_prog_err_B** is not elaborated before **elab_prog_ err_C**, then variable **elab_prog_err_C.K** is undefined.

3.4 Using elab on the Sample Unit

To run **elab** on the sample unit, you can use the **adawise** menu or command line arguments as described in Chapter 1.

Here is a command line for checking the sample code with the **elab** tool:

```
adawise tools=elab units=elab_prog_err_main
```

This command line takes the defaults for the other arguments (i.e., the unit is an Ada body, the mode is non-verbose, and the library file is **liblst.alb**).

This command produces the following output:

```
Beginning AdaWise
Compiler Vendor: Alsys Inc.

-> *** The following TeleSoft sublibs will be ignored:
-> $TELEGEN2/lib/rt.sub


**
** Executing Tool: elab   Ada unit: elab_prog_err_main (body)
**

*****************************************************************************
** CHECKING ELABORATION ORDER ERRORS FOR MAIN UNIT: elab_prog_err_main
**
**  Errors checked:
**    circular dependencies,
**    missing related units,
**    obsolete related units,
**    incorrect order dependence
*****************************************************************************
-> Number of compilation units to check:  6

>> WARNING:  Compilation Units are NOT independent:
>>        elab_prog_err_c(A_PACKAGE_DECLARATION)  and
>>        elab_prog_err_b(A_PACKAGE_BODY)
>>   Object: ELAB_PROG_ERR_A(A_PACKAGE_DECLARATION):i
>>   is UPDATED in elab_prog_err_b(A_PACKAGE_BODY)
>>   and READ in elab_prog_err_c(A_PACKAGE_DECLARATION)

>> WARNING:  Compilation Units are NOT independent:
>>        elab_prog_err_a(A_PACKAGE_BODY)  and
>>        elab_prog_err_b(A_PACKAGE_DECLARATION)
>>   Subprogram: f
>>   is DEFINED in elab_prog_err_a(A_PACKAGE_BODY)
>>   and CALLED in elab_prog_err_b(A_PACKAGE_DECLARATION)
```

```
**************************************************************************
>> WARNING: elab_prog_err_main
>>      has a potential INCORRECT ORDER DEPENDENCE during elaboration.
**************************************************************************


**
** elab Finished
**
```

As this output shows, the non-verbose output includes a list of the kinds of errors being checked, warnings for compilation units that are not independent, and a warning pointing out the possibility of an incorrect order dependence during the elaboration of the unit.

If we use the verbose mode, we get the following output:

```
Beginning AdaWise
Compiler Vendor: Alsys Inc.

-> *** The following TeleSoft sublibs will be ignored:
-> $TELEGEN2/lib/rt.sub

ASIS version:   1.1.0.2
Vendor version: 1.1.0.2
Vendor info:    1.1.0.2

-> Verbose Option set


-> Opening library: liblst.alb

-> Unit: elab_prog_err_main (body)

**
** Executing Tool: elab   Ada unit: elab_prog_err_main (body)
**
5/10/1994 14:58: 4.89414E+01

**************************************************************************
** CHECKING ELABORATION ORDER ERRORS FOR MAIN UNIT: elab_prog_err_main
**
** Error messages for compilation units X and Y will be of the form:
**  Object UPDATED in X is READ in Y
**  which means:
**      during elaboration of X, a variable is updated
**      which is potentially aliased with a variable
**      that is read during the elaboration of Y.
** OR
**  Subprogram DEFINED in X is CALLED in Y
**  which means:
**      the elaboration of X declares a subprogram body
```

```
**      that is called during elaboration of Y
**
**  Errors checked:
**    circular dependencies,
**    missing related units,
**    obsolete related units,
**    incorrect order dependence
**
**NOTE: if none of the above errors, then all legal elaboration orders
**      will produce the same visible results -
**      (including possibly some other type of error).
**      Other tools (e.g. definedness) can use any of the legal orders.
********************************************************************************
5/10/1994 14:58: 4.91094E+01

-- Checking: Inconsistent units
-- Checking: Missing units
-- Checking: Circularities


-- Checking: Independence of incomparable compilation unit pairs

-> Number of compilation units to check:  6
5/10/1994 14:58: 4.97109E+01
-> Number of explicit units to check:  5

-- Checking: Compilation unit pair:
elab_prog_err_a(A_PACKAGE_DECLARATION) <--> elab_prog_err_b(A_PACKAGE_BODY)

-- Checking: Compilation unit pair:
elab_prog_err_a(A_PACKAGE_BODY) <--> elab_prog_err_b(A_PACKAGE_BODY)

-> Pair is incomparable - checking independence
->  Compilation Units are independent.


-- Checking: Compilation unit pair:
elab_prog_err_c(A_PACKAGE_DECLARATION) <--> elab_prog_err_b(A_PACKAGE_BODY)

-> Pair is incomparable - checking independence

>> WARNING:  Compilation Units are NOT independent:
>>        elab_prog_err_c(A_PACKAGE_DECLARATION)  and
>>        elab_prog_err_b(A_PACKAGE_BODY)
>>   Object: ELAB_PROG_ERR_A(A_PACKAGE_DECLARATION):i
>>   is UPDATED in elab_prog_err_b(A_PACKAGE_BODY)
>>   and READ in elab_prog_err_c(A_PACKAGE_DECLARATION)


-- Checking: Compilation unit pair:
elab_prog_err_b(A_PACKAGE_DECLARATION) <--> elab_prog_err_b(A_PACKAGE_BODY)
-> Not checking own spec

-- Checking: Compilation unit pair:
```

```
elab_prog_err_a(A_PACKAGE_DECLARATION) <--> elab_prog_err_b(A_PACKAGE_DECLARATION)

-- Checking: Compilation unit pair:
elab_prog_err_a(A_PACKAGE_BODY) <--> elab_prog_err_b(A_PACKAGE_DECLARATION)

-> Pair is incomparable - checking independence

>> WARNING:  Compilation Units are NOT independent:
>>        elab_prog_err_a(A_PACKAGE_BODY)  and
>>        elab_prog_err_b(A_PACKAGE_DECLARATION)
>>   Subprogram: f
>>   is DEFINED in elab_prog_err_a(A_PACKAGE_BODY)
>>   and CALLED in elab_prog_err_b(A_PACKAGE_DECLARATION)


-- Checking: Compilation unit pair:
elab_prog_err_c(A_PACKAGE_DECLARATION) <--> elab_prog_err_b(A_PACKAGE_DECLARATION)

-> Pair is incomparable - checking independence
->  Compilation Units are independent.


-- Checking: Compilation unit pair:
elab_prog_err_a(A_PACKAGE_DECLARATION) <--> elab_prog_err_c(A_PACKAGE_DECLARATION)

-- Checking: Compilation unit pair:
elab_prog_err_a(A_PACKAGE_BODY) <--> elab_prog_err_c(A_PACKAGE_DECLARATION)

-> Pair is incomparable - checking independence
->  Compilation Units are independent.


-- Checking: Compilation unit pair:
elab_prog_err_a(A_PACKAGE_DECLARATION) <--> elab_prog_err_a(A_PACKAGE_BODY)
-> Not checking own spec

**********************************************************************
>> WARNING: elab_prog_err_main
>>      has a potential INCORRECT ORDER DEPENDENCE during elaboration.
**********************************************************************


END Check_Elaboration_Order
5/10/1994 14:58: 5.12813E+01

**
** elab Finished
**
5/10/1994 14:58: 5.12891E+01
```

The verbose option explains more about the elaboration order errors that it's checking for; gives checkpoint notices (with the `-- Checking:` prefix) as it performs each check; gives additional information (with the `->` prefix); shows each pair of user compilation units that is

being checked, whether the pair is incomparable, and whether the pair is independent (each pair being checked has a <--> separator); and gives warning messages.

As described in Chapter 1, if we list multiple units on the command line, **AdaWise** calls the tool to analyze each unit. For example, if we have the **elab_prog_err** unit and a unit called **elab_global**, we can use the command line

adawise u='elab_prog_err_main elab_global_main' t=elab

which runs **elab** first on the **elab_prog_err** unit (with the same output as shown previously) and then on the **elab_global** unit:

```
Beginning AdaWise
Compiler Vendor: Alsys Inc.

-> *** The following TeleSoft sublibs will be ignored:
-> $TELEGEN2/lib/rt.sub


**
** Executing Tool: elab   Ada unit: elab_prog_err_main (body)
**

*****************************************************************************
** CHECKING ELABORATION ORDER ERRORS FOR MAIN UNIT: elab_prog_err_main
**
**  Errors checked:
**    circular dependencies,
**    missing related units,
**    obsolete related units,
**    incorrect order dependence
*****************************************************************************
-> Number of compilation units to check:  6

>> WARNING:  Compilation Units are NOT independent:
>>        elab_prog_err_c(A_PACKAGE_DECLARATION)  and
>>        elab_prog_err_b(A_PACKAGE_BODY)
>>   Object: ELAB_PROG_ERR_A(A_PACKAGE_DECLARATION):i
>>   is UPDATED in elab_prog_err_b(A_PACKAGE_BODY)
>>   and READ in elab_prog_err_c(A_PACKAGE_DECLARATION)

>> WARNING:  Compilation Units are NOT independent:
>>        elab_prog_err_a(A_PACKAGE_BODY)  and
>>        elab_prog_err_b(A_PACKAGE_DECLARATION)
>>   Subprogram: f
>>   is DEFINED in elab_prog_err_a(A_PACKAGE_BODY)
>>   and CALLED in elab_prog_err_b(A_PACKAGE_DECLARATION)

*****************************************************************
```

```
>> WARNING: elab_prog_err_main
>>      has a potential INCORRECT ORDER DEPENDENCE during elaboration.
*****************************************************************


**
** elab Finished
**

**
** Executing Tool: elab   Ada unit: elab_global_main (body)
**

*******************************************************************************
** CHECKING ELABORATION ORDER ERRORS FOR MAIN UNIT: elab_global_main
**
**  Errors checked:
**    circular dependencies,
**    missing related units,
**    obsolete related units,
**    incorrect order dependence
*******************************************************************************
-> Number of compilation units to check:  4


>> WARNING:  Compilation Units are NOT independent:
>>        elab_global_a(A_PACKAGE_BODY)  and
>>        elab_global_b(A_PACKAGE_DECLARATION)
>>   Object: ELAB_GLOBAL_A(A_PACKAGE_DECLARATION):i
>>   is UPDATED in elab_global_a(A_PACKAGE_BODY)
>>   and READ in elab_global_b(A_PACKAGE_DECLARATION)

*****************************************************************
>> WARNING: elab_global_main
>>      has a potential INCORRECT ORDER DEPENDENCE during elaboration.
*****************************************************************


**
** elab Finished
**
```

Chapter 4

The def Tool

4.1 Undefined Objects in Ada

In compiler terminology, reading a variable is call a "use", and writing it, a "definition". We can also use the terms "write", "read", and "create". An Ada variable is *written* when it is assigned to, *read* when it is evaluated as part of some expression, and *created* when its declaration is elaborated. We have a "definedness" error if a variable is used (or read) before it is defined (or written). Definedness is the problem addressed by the **def** tool.

Definedness errors can precipitate two problems, one of which is erroneous execution. The execution of a program is erroneous if it attempts to evaluate a scalar variable with an undefined value or apply a predefined operator to a variable that has a scalar subcomponent with an undefined value (ARM section 3.2.1). In addition, defined variables may become undefined during the execution of the program—even when declared as a record type with default expressions for all scalar components. An unsuspecting programmer may be incorrectly assuming that an object of such a type is always guaranteed to be defined.

An Ada scalar can become (accessible and) undefined in the following situations:

- An Ada scalar is undefined when it is created, unless it is given a defined initial value. (Note that it can be initialized and still be undefined if, for example, the corresponding component of the initialization expression is undefined.)
- An Ada scalar is undefined after an assignment to a containing record.
- Likewise, an Ada scalar is undefined after a procedure call where the IN or IN-OUT actual parameter is a containing record and the component is undefined.
- A scalar OUT parameter is undefined after a call if it is not updated within the call.
- A component of an array or record variable may be undefined after an exception handled in the body of a subprogram if the component was undefined before the call

or the component of the formal parameter becomes undefined during the execution of the subprogram.

- A scalar at the beginning of an exception handler is undefined if it was an actual OUT parameter.
- A formal parameter becomes undefined after modifying a global alias (ARM 6.2). The **aiod** tool can been used to preclude potential errors due to aliasing.

Definedness errors can also cause program errors. This type of definedness error is a call to a subprogram whose body has not been elaborated.

4.2 What def Checks

The definedness checking tool analyzes an Ada program for two potential problems: erroneous executions resulting from reading undefined scalars and program errors resulting from calling a subprogram whose body has not been elaborated.

The problem of automatically detecting undefined variables is undecidable in general, but we use a decidable conservative approximation based on data flow analysis of the kind often used in optimizing compilers.

Often, the proof that a variable is written before being read is easy: simply follow the flow of control on each possible path until one or the other of these actions happens. It is not always so simple, though. The analysis gets more complicated if it is difficult to determine whether a control flow path is taken, if it is difficult to determine whether an expression actually reads or writes a particular variable at run-time, or if the control flow path is infinite and it is not clear whether the variable is read or written at all.

The definedness tool conservatively automates the analysis to handle the easy cases correctly and to identify the hard cases. The data flow traces can be extracted by following the program's control flow and neglecting use of state information to determine the flow of control at branch points. By analyzing the enlarged set of data flow traces, we can guarantee that no trace of the actual program will fail to be analyzed.

In the case of analyzing for program errors, our model treats the elaboration of the body of a subprogram as a "write", a call to the subprogram as a "read", and the declaration as a "create". Our algorithm assumes there are no aliasing errors and that there are no incorrect order dependences in the elaboration (i.e., that all legal elaboration orders produce the same results). The **aiod** and **elab** tools can be used to check for aliasing errors and incorrect order dependence during elaboration.

The **def** tool is only partially implemented at this time.

Appendix A

Error Messages from AdaWise

This appendix contains a partial list of some common messages that you may encounter when running the **AdaWise** tools, along with explanations of the causes and solutions for the messages.

There are three kinds of messages that can be issued by the tools: warnings that the code being checked has a potential error, error messages because the user entered incorrect input or did not set up the environment correctly, and errors messages because of internal bugs or deficiencies in the AdaWise executable itself.

A.1 Warnings

The messages issued by the tools to warn of potential incorrect order dependence, erroneous execution, or undefined variables are prefixed by ">>". Chapters 2, 3, and 4 illustrated some of these messages. In the following warning messages, names enclosed in {} are variables, depending on the source being analyzed.

A.1.1 Warnings from the aiod Tool

Subprogram parameter error:

```
>> WARNING: potential parameter error in
>> {compilation-unit}: line {line-number}: {subprogram-call-source}
```

The parameter error message is followed by an explanation of the kind of error, which depends on the mode and type of the formal parameters:

- global aliasing

```
>> Parameter  {parameter-position}: is aliased with a global.
>> (potential ERRONEOUS EXECUTION)
```

- pair-wise aliasing of parameters (except if mode of both is IN):

```
>> Parameters {parm-position1} and {parm-position2}: are potential ALIASES
>> (potential ERRONEOUS EXECUTION)
```

or,

```
>> Parameters {parm-position1} and {parm-position2}: are potential ALIASES
>> (potential ORDER of COPY OUT error)
```

or, if both types of error are possible:

```
>> Parameters {parm-position1} and {parm-position2}: are potential ALIASES
>> (potential ORDER of COPY OUT error) and  (potential ERRONEOUS EXECUTION)
```

- pair-wise independence of all parameters:

```
>> Parameters {parm-position1} and {parm-position2}: are NOT independent.
>>  (potential ORDER of COPY IN or ADDRESS EVALUATION error)
```

Incorrect order dependences (other than subprogram parameters):

```
>> WARNING: potential incorrect order dependence (IOD)
>> {compilation-unit}: line {line-number}: {source-expression}
```

The IOD error message will be followed by an explanation of the type of error and the section of the Ada Reference Manual that mentions the construct that could be evaluated in any order by the compiler implementation.

- in assignment statement (5.2(3))
- in discrete range (3.6(10))
- in record component list (3.2.1(15))
- in range (3.5(5))
- in discriminant constraint (3.7.2(13))
- in indexed component (4.1.1(4))
- in aggregate (4.3.1(3), 4.3.2(10))
- between operands of an operator (4.5(5))
- in discriminant part (3.2.1, 3.7.1)
- in index constraint (3.6(10))

The object that causes the potential IOD is printed, followed by whether it is updated or read by the construct. The variables {expression1} and {expression2} are the two expressions that may be evaluated in any order from {source-expression}:

```
>> Object: {compilation-unit-name}(unit-kind):{simple-name}
>>   is updated by: {expression1}
>>   and read by:   {expression2}
```

A.1.2 Warnings from the elab Tool

The **elab** tool prints the number of compilation units being checked, and if there is a potential incorrect order dependence during elaboration, prints warning messages depending on the kind of error. There are three kinds of user errors that cause **elab** to stop processing and that will be described in the next section:

- circular dependencies
- inconsistent units
- missing units

For incorrect order dependence, the warning message is followed by the names of the compilation units that are not independent and either a subprogram that is called in one unit and defined in the other:

```
>> WARNING:  Compilation Units are NOT independent:
>>        {compilation-unit-name}({unit-kind})
>>        {compilation-unit-name}({unit-kind})
>>   Subprogram: {subprogram-name}
>>   is DEFINED in {compilation-unit-name}({unit-kind})
>>   and CALLED in {compilation-unit-name}({unit-kind})
```

or, an object that is read or written to cause the potential incorrect order dependence:

```
>> Object: {compilation-unit-name}(unit-kind):{simple-name}
>>   is updated in: {compilation-unit1}
>>   and read in:   {compilation-unit2}
```

At the end of the execution of **elab**, if there was a potential incorrect order dependence during elaboration, the following message will be printed:

```
*****************************************************************
>> WARNING: {main-unit-name}
>>      has a potential INCORRECT ORDER DEPENDENCE during elaboration.
*****************************************************************
```

A.1.3 Warnings from the def Tool

The definedness tool will warn of three cases: if there is an attempt to evaluate a scalar variable with a potentially undefined value, or an attempt to apply a predefined operator to a variable that has a scalar subcomponent with a potentially undefined value, or an attempt to call a subprogram before its body has been elaborated.

TBD

A.2 User Errors

- ERROR MESSAGE:

```
** ERROR: Compilation Unit Unknown: {unit-name}
** Was the unit compiled with options necessary for ASIS?
```

  - EXPLANATIONS:

    You may get this error message after entering an Ada unit in a command line or from the menu. This message means that the tool can't find the unit in the library. You may have misspelled a unit name or have entered a unit that was not compiled into the library.

    You may have compiled the program without the necessary **ada** option.

  - SOLUTIONS:

    If you are running the tools on one of the demonstration examples, check the documentation for the name of the unit to check. Use the compiler **als** command or list the **.obj** directory to see what units are in the library.

    Each file in the program must be compiled with the **-k** or **-d** option. Make sure to compile the program with one of those options to the RISCAda compiler, for example, **ada -k foo.aa**. If the **amake** program is used, the '**-N** option may be used to pass the option to the compiler, for example, **amake -N -k *.aa**.

- ERROR MESSAGE:

```
** ERROR: Unknown tool
```

  - EXPLANATION:

    You may get this error message after entering an incorrect name on the command line for the "**t=**" option.

  - SOLUTION:

    The name of the tool must be one of **elab, aiod, def.**

- ERROR MESSAGE:

```
** ERROR: NAME_ERROR: Error opening library
** ERROR: Open: Name Error. Library "{input-library}" not found
**   Does {input-library} exist?
**   Does {input-library} contain a sublib file that does not exist?
```

  - EXPLANATION:

    You will get this error if the name given in response to the library prompt does not exist, or one of the sublibs mentioned in the **.alb** file is incorrect.

– SOLUTION:

If you take the default library name by using command line arguments for tool and unit, or by hitting carriage return in response to the prompt, then check that **liblst.alb** exists in your current directory. If you use the "-l" option to the tool, or specify a library name in response to the prompt, then make sure you have spelled the filename correctly. If the library file exists, then check that each sublib specified in it by "name: {sublib}" is a sublib that has been created by using the command "**acr** {sublib}".

- ERROR MESSAGE (**elab** or **def** tool):

  ```
  ** ERROR: {main-unit-name} A CIRCULARITY EXISTS.
  ```

  - EXPLANATION: The **elab** tool discovered a circularity among the supporting units required for the elaboration of the input compilation unit.
  - SOLUTION: Run the **elab** tool with the verbose option. A list of pairs of units will be displayed, with each pair representing a unit followed by a supporter unit. A circularity is established when a unit's supporter unit is equal to the first unit in the set. Note that the compiler itself will usually report this error at link time.

- ERROR MESSAGE (**elab** or **def** tool):

  ```
  ** ERROR: {main-unit-name} has INCONSISTENT UNITS.
  ```

  - EXPLANATION: The units in the library required for the elaboration of the input compilation unit are inconsistent due to the recompilation of one or more supporting units.
  - SOLUTION: Run the **elab** tool with the verbose option. A list of pairs of units will be displayed, with each pair representing a unit followed by a unit causing the inconsistency. Note that the compiler itself will usually report this error at link time.

- ERROR MESSAGE (**elab** or **def** tool):

  ```
  ** ERROR: {main-unit-name} has MISSING RELATED UNITS.
  ```

  - EXPLANATION: Units are missing from the library that are required for the elaboration of the input compilation unit. Package bodies that are optional according to the rules of Ada, and that do not exist, are not considered missing.
  - SOLUTION: Run the **elab** tool with the verbose option. A list of pairs of units will be displayed, with each pair representing a unit followed by a missing related unit. Note that the compiler itself will usually report this error.

A.3 Errors within AdaWise

If **AdaWise** detects an internal inconsistency, it prints messages prefixed by "INTERNAL ERROR".

If STORAGE_ERROR or another predefined exception is raised running any of the tools, check the input arguments. If there were multiple units listed for the "**u=**" option or multiple tools listed in the "**t=**" option, then rerun the tools separately, specifying only one unit on the command line with each tool execution.

Bibliography

[1] ANSI. *Reference Manual for the Ada Programming Language*, 1983. ANSI/MIL-STD-1815A.

[2] LabTek Corporation. Cecom final report - catalogue of ada runtime implementation dependencies. Technical Report CIN C02-092JB-0001-00, U.S. Army HQ, Center for Software Engineering Advanced Software Technology, February 1989.

[3] B.A. Wichmann. Insecurities in the ada programming language. Technical Report ISSN 0262-5369, National Physical Laboratory, January 1989.