Ada Bug Finder

Alan Marriott and Urs Maurer
White Elephant GmbH, Postfach 327, CH-8450 Andelfingen, Switzerland; email: ada@white-elephant.ch

Abstract

In the context of this paper, we consider bug patterns to be sections of code that whilst syntactically correct are unlikely to be what the author intended.

Everyone, even the most erudite programmers, make dumb mistakes, often as a result of a particularly inept piece of cut and paste editing or sometimes simply by typing the exact opposite of what was meant.

Our experience has shown that even the most blatantly incorrect code can make its way into production code!

In many situations compilers could have detected the bug patterns. However it seems that the current generation of Ada compilers is content if the programmer writes legal Ada syntax. Determining whether this code is meaningful or not seems to beyond their remit.

As a consequence we have written a bug finder tool which, using static code analysis, attempts to detect code that is either obviously incorrect, is in some way questionable or is so badly written that the tool itself cannot make sense of it and is therefore worthy of further analysis.

This paper describes the tool, the bug patterns it employs and an evaluation of the results of applying the tool over several large Ada code bases.

Keywords: utility, bug finder, Ada.

1 Introduction

In the autumn of 2004, we were fortunate to attend the Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2004 conference in Vancouver, BC, Canada. As part of the OOPSLA Onward! Track, David Hovemeyer and William Pugh presented their paper entitled "Finding Bugs is Easy [1]"

Their paper and the presentation they made at the conference have been the basis and inspiration for the work described in this paper. Hovemeyer & Pugh's work concentrated exclusively on Java. We merely extended and adapted the idea for Ada.

Their basic premise is that many simple and obvious bugs slip through testing and end up in production code and that with a little bit of effort these bugs can be automatically found.

It is their belief that bugs in production code are not normally found because either the user does not notice the symptom of the bug, has no means to report the bug to the developers or cannot reproduce the situation that caused the bug.

Their idea is that if you detected a bug in some code you are working on, you should examine how one could look for other occurrences of the same bug and then try to determine whether this search could be somehow automated.

If a pattern can be established by which the bug can be automatically discovered then this mechanism should be incorporated into some form of tool and the tool used to search actively for the bug in as much source code as possible.

This is to say not just in the current module or project, but also in other projects, libraries and as much open source that is available.

Because their paper concentrated on Java and Java specific problems most of the bug patterns described by Hovemeyer and Pugh were not applicable. Therefore part of our work has been to develop bug patterns that are specific to Ada.

2 The Ada Bug Finder Utility

Our Ada Bug Finder tool is an interactive Windows® based program written exclusively in Ada95.

Although not open source, the executable is available for download from our web site www.white-elephant.ch We would like actively to encourage everyone to try it out on all available Ada source code and to report back to us with statistics regarding how many bugs the tool found and how many of these were serious.

We would also be interested in any feedback concerning how the utility might be enhanced either by suggesting new bug patterns or citing occasions where an unnecessarily large number of false positives were incurred.

2.1 Overview

The Ada Bug Finder utility takes the name of a directory as its only input. When commanded the utility searches all the Ada files contained in the specified directory and all its subdirectories.

Ada package specifications and implementations are assumed to be in pairs of files, either

- Both files have the same filename but different file extensions. The package specification having the file

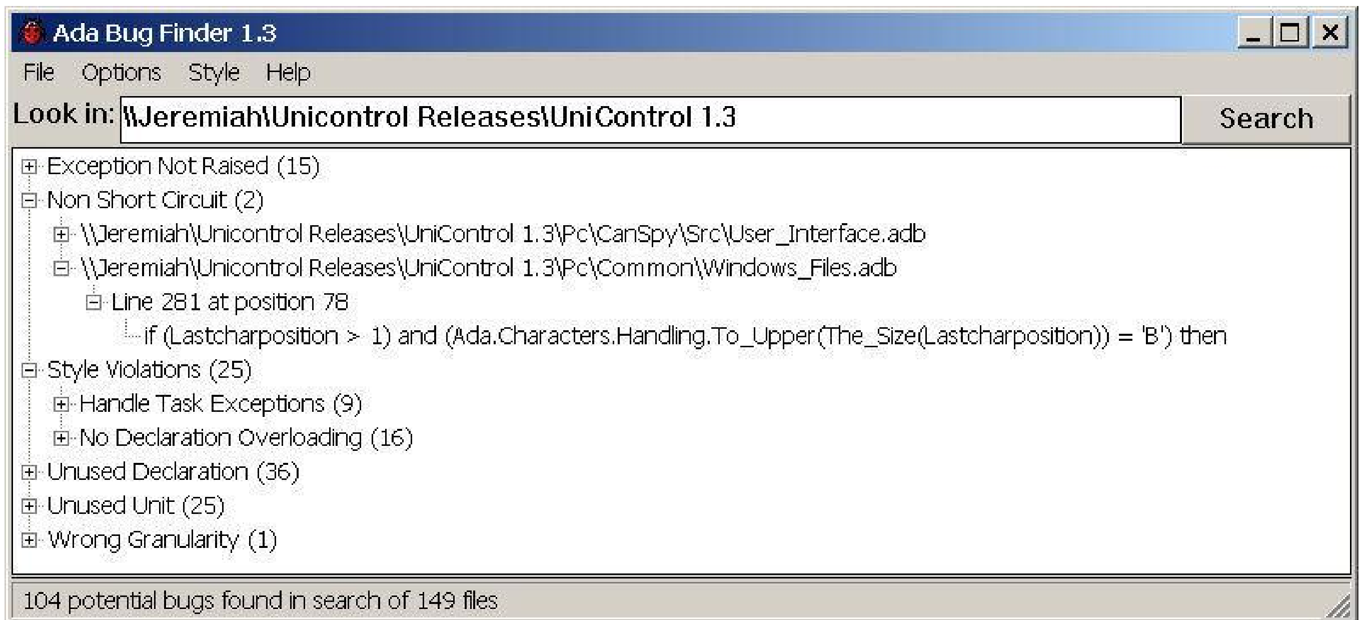

Figure 1 Screenshot of the Ada Bug Finder

extension ads and the implementation the extension adb

For example: Test.ads and Test.adb

- Both files have the file extension ada and share the same filename up until the final character. An additional underscore signifies that the file contains the package specification.

  For example: Test_.ada and Test.ada

The program supports two options.

- Gnat extensions

  If enabled instructs the syntax parser to accept the Gnat implementation defined attributes.

- Preparation phase.

  If enabled causes the utility to process the files twice. The first pass gathers additional information that can be used to reduce the number of false positives at the expense of speed.

Results of the search are displayed in a tree view. These may be saved as either text or as a comma delimited file suitable for further processing by utilities such as Microsoft Excel.

2.2 Bugs vs. Style

Hovemeyer and Pugh consider the primary purpose of style rules is to make it easier for developers to understand each other's code and consequently they should not be included in a bug-finding tool.

Whilst we agree that this is true with respect to many style rules, we believe that some style rules have been introduced with the specific intention of prohibiting language features that are considered to encourage unsafe programming practices. Other style rules have been introduced to facilitate easier debugging.

In both these cases the enforcement of style rules could directly affect the software reliability. For this reason our utility also offers the optional detection of various style rule violations.

2.3 False Positives

Unfortunately, the utility doesn't always get it right! Occasionally the utility will highlight a segment of code as being a bug when, in fact, it is perfectly correct.

A goal of the utility is to reduce the number of these false positives to a minimum without making the pattern unnecessarily complex and unduly expensive to implement. There is a trade off, therefore, between complexity and the number of false positives the pattern might generate.

Within reason, we would rather have a few false positives than the utility missing an actual bug or making the pattern so complex that it becomes no longer reasonable to implement.

2.4 Code Marking

Given that there will be the occasional false positive, we decided that there should be a mechanism whereby the utility could be instructed to ignore a specific pattern on a particular line.

The mechanism we chose is to place a special comment at the end of the line on which the utility detects the bug. The special comment starts with a *greater than* symbol (>) followed by a two or three character bug abbreviation code terminated by a colon.

Example:

C_False : **constant** Integer := 0; --> UD: Completeness

In the above example the line is flagged against unused declarations (UD). The Ada Bug Finder utility will not report any unused declarations declared on this line.

3 Ada Bug Patterns

Version 1.3 of the Ada Bug Finder utility recognises eight Ada bug patterns.

3.1 Illogical Operator Rename (IOR)

In Ada83, where there is no *use type* clause, operators are often renamed to avoid the use of prefixed notation in environments where the *use* clause is expressly forbidden.

Clumsy cut and paste editing might result in renaming an operator to be something totally different. The compiler allows this, although it is highly unlikely to be what the author intended.

Example:

```
function "<"(Left, Right : Xt.Widget)
            return Boolean renames Xt."=";
```

3.2 Code Not Reachable (CNR)

Statements after an unconditional raise, return or exit will never be executed.

Note: The Gnat compiler 3.15p checks for this pattern however both the Aonix and HP compilers do not.

Example:

```
procedure Cnr is
begin
  loop
    exit;
    Io.Put_Line ("Never written!");
  end loop;
  return;
  Io.Put_Line ("Will never get written!");
end Cnr;
```

3.3 Null Pointer (NP)

This pattern looks for occasions of a pointer being dereferenced whilst it is known to be null. Typically, this occurs in the body of an *if* statement that has previously tested the pointer explicitly for null.

Example:

```
if The_String = null then
  Io.Put_Line (The_String.all);
end if;
```

3.4 Non Short Circuit (NSC)

Essentially, testing for a condition and then, in the same expression, using the result of that condition normally requires that the programmer use the "and then" or the "or else" construct rather than simply "and" or "or".

Example:

```
Result := (The_String = null) or
          (The_String.all = "Hello");
```

```
Result := (Index <= Numbers'last) and
          (Numbers (Index) = 42);
```

3.5 Wrong Granularity (WG)

Ada's *'Size* attribute returns the size of the object in bits whereas storage allocation and most interfaces expect object sizes to be supplied in bytes.

Consequently it is very unusual for *'Size* to be used outside of an expression. These occurrences are likely to be bugs and therefore warrant further scrutiny.

Example:

```
Read (Buffer      => Buffer'address
      Max_Size    => Buffer'size
      Amount_Read => The_Size);
```

3.6 Unused Declarations

If something is declared but never used, it might simply be because it is not required. Its presence might cause the compiler more work, it might make the program bigger and it might possibly make the code less understandable.

Whilst all these symptoms are undoubtedly undesirable they are not actually bugs.

However another reason that a declared object may never be referenced is because something else is being referenced in place of it. These occurrences are bugs and it the aim of the next three patterns is to identify them.

3.6.1 Unused Declaration (UD)

A constant or variable is declared but never used. Note, however, that this might be deliberate. The initialization of controlled objects or the default initialization may have an effect that is actually required.

3.6.2 Exception Not Raised (ENR)

An exception is declared, perhaps even handled, but is never raised.

3.6.3 Unused Unit (UU)

A package is imported but never used, or a procedure, function or package is defined but neither exported nor used locally.

3.7 Syntax Error (SE)

This isn't really a bug pattern per se. Unfortunately, some of the code placed in open source libraries doesn't actually compile! Our utility reports syntax error if the code it is analysing appears to be invalid Ada.

4 Style Rule Checking

Our utility optionally checks six style rules. Each style rule can be individually enabled or disabled.

The paragraph references within parentheses refer to the Ada 95 Quality and Style Guide [2].

4.1 HTE - Handle Task Exceptions (6.3.4)

A task will terminate if an exception is raised within it, for which there is no exception handler. In such cases, the exception is not propagated outside of the task (unless it occurs during a rendezvous). The task simply dies with no notification to other tasks in the program. This makes debugging these tasks especially difficult and so we have implemented a style rule that checks that every task has an exception handler at its outermost level that includes a *when others* statement.

4.2 NDO - No Declaration Overloading

Prohibits declarations that have the same name as a declaration currently in scope. We believe that it is poor programming style to occlude a declaration deliberately.

4.3 NGS - No Goto Statements (5.6.7)

Prohibits the use of the *goto* statement as this is considered an unstructured change in the control flow. In Ada, the label does not require an indicator of where the corresponding *goto* statements are. Many believe that this renders the code unreadable.

4.4 NPUC - No Package Use Clause (5.7.1)

Prohibits the use of the *use* clause and thereby forces external names to be always fully qualified. To provide visibility to operators use the *use type* clause.

4.5 NVIS - No Variable in Specification

Prohibits the declaration of variables in package specifications.

4.6 CNP – Code Not Portable

In Ada83 identifiers may only contain ASCII alphanumeric characters. However some compilers fail to enforce this restriction. Although Ada-95 allows identifiers to be constructed from any alphanumeric from row 00 of the ISO 10646 BMP, effectively ISO 8859-1 (Latin-1), using characters outside of the ASCII character range may lead to portability problems.

4.7 Superfluous Code Mark

If an Ada Bug Finder code mark (>xx:) is used to suppress the reporting of a particular bug but the line in question doesn't actually produce the bug in question then something is probably wrong. It is bad style to suppress warnings unnecessarily.

5 Other Patterns (to be implemented)

5.1 Division by Zero

This pattern looks for the situation when an identifier is explicitly compared with zero and then used as the right operand of one of the operators /, *rem* and *mod*

Example:

```
if Index = 0 then
  Result := (42 / Index) > 10;
end if;
```

5.2 Raise after Assignment

Leaving a procedure abnormally nullifies any assignment to in-out or out parameters.

Example:

```
procedure Raa (The_Number : in out Natural) is
begin
  The_Number := The_Number + 1;
  raise Failed;
end Raa;
```

5.3 Redundant Comparison to null

If a null pointer check is made after code has already dereferenced the pointer, the comparison is redundant.

Either the comparison is made too late or is superfluous because the condition is known never to arise.

Example:

```
procedure Rcn is
begin
  Ada.Text_Io.Put_Line (The_String.all);
  if The_String /= null then
    Ada.Text_Io.Put_Line (The_String.all);
  end if;
end Rcn;
```

5.4 Symmetrical Comparison

If the left and right sides of a comparison are identical then this is probably a cut and paste error as it obviously makes no sense!

Example:

```
if Table (Index) = Table (Index) then
```

Sources	Files	CNR	ENR	IOR	NSC	NP	SE	UD	UU	WG	Style
UniControl 1.3	149		15		2			36	25	1	25
ILTIS 3622_12_36	4539	25	267	2	131	11		1672	317	23	2109
Aonix 7.2.2	828	2	18		4			196	23	5	1080
GCC 3.15p, Gps1.4	2976	1	55		4	1	8	255	236	3	14070
AI-302	147				1			1	1		240

Figure 2 – Bug Warnings

Sources	Total	CNP	HTE	NDO	NGS	NPUC	NVIS	SCM
UniControl 1.3	25		9	16				
ILTIS 3622_12_36	2109	24	22	1443		169	451	
Aonix 7.2.2	1080		16	20	2	475	567	
Gnat GCC 3.15p, Gps1.4	14070		17	314	462	11303	1974	
AI-302	240			2	3	233	2	

Figure 3 - Style Rule Violations

6 Evaluation

It was relatively easy to use our utility to search for bugs in the Ada source code we had available, however, evaluating the results is a time-consuming and subjective process.

Figures 2 and 3 summarise the results of our using the Ada Bug Finder version 1.4 on the following applications and libraries

- Soudronic AG, UniControl release 1.3
- Siemens AG, ILTIS PC release 3622_12_36
- Source code provided with the Aonix compiler version 7.2.2
- Gnat open source for GCC version 3.15p, Gps 1.4 and Xml
- Charles library & AI- 302

In Figure 2, the number of files that the utility analysed is provided in order to give some sort of idea as to the their relative sizes.

Code	Description
CNR	Code Not Reachable
ENR	Exception Not Raised
IOR	Illogical Operator Rename
NSC	Non Short Circuit
NP	Null Pointer
SE	Syntax Error
UD	Unused Declaration
UU	Unused Unit
WG	Wrong Granularity
Style	Style Rule Violation

Figure 4 - Bug Pattern Codes

Code	Description
CNP	Code Not Portable
HTE	Handle Task Exceptions
NDO	No Declaration Overloading
NGS	No Goto Statements
NPUC	No Package Use Clause
NVIS	No Variable In Specification
SCM	Superfluous Code Mark

Figure 5 - Style Rule Codes

Unfortunately we have had neither the time nor the resources to make anything other than a cursory evaluation of the results.

However, we have been able to make the following observations:

1. The ILTIS application was the only Ada83 code we analysed. This explains why it alone contained illogical operator renaming bugs.
2. We believe that the low number of CNR bugs within the Gnat code base can be attributed to it normally being compiled using the Gnat Ada Compiler which itself issues this type of warning.
3. The vast majority of reported bugs were harmless unused declarations of some sort. However, we believe that removing this clutter generally improved the readability of the code.

7 Conclusions

The utility has been instrumental in discovering several bugs that had made their way into production code.

Some of these bugs were so obscure that they would probably be very difficult to discover using traditional methods.

For example, the UniControl Wrong Granularity (WG) bug informed an API that a buffer was larger than it really was. The consequence of this was that occasionally, depending on what the function wanted to return, code would get overwritten and the application would crash.

Although written to search for bugs in existing code bases, we have discovered that the utility is also a useful development tool. Occasionally running the Bug Finder over newly developed code before it has been released or submitted into a library has detected several bugs that probably would have only been detected during testing.

8 An alternative method

From start to finish, the Ada Bug Finder project, including testing and presentation, took 140 Man-hours of effort.

We were able to develop the utility within these constraints by reusing an Ada text parser that we had developed for a previous project.

However using static code analysis has severe limitations. The utility simply does not know enough about the semantics of the code it is analysing for it to detect some of the bug patterns we had hoped to implement.

An alternative method could be to use the ASIS compiler interface [3]. This is an open, published callable interface that gives access to semantic and syntactic information from an Ada environment.

Acknowledgments

Siemens Schweiz AG sponsored the development of the Ada Bug Finder

References

[1] David Hovemeyer and William Pugh,*Finding Bugs Is Easy*. Department of Computer Science, University of Maryland, College Park, Maryland 20742 USA {daveho, pugh}@cs.umd.edu

[2] Ausnit-Hood, Johnson, Pettit & Opdahl, *Ada 95 Quality and Style*. LNCS 1344, Springer-Verlag

[3] Ada Semantic Interface Specification (ASIS) JTC1/SC22 ISO Standard ISO/IEC 15291:1999