Re-engineering Global Variables in Ada

Ricky E. Sward
Department of Computer Science
U.S. Air Force Academy, CO
719-333-7664
ricky.sward@usafa.af.mil

A.T. Chamillard
Computer Science Department
University of Colorado at Colorado Springs
Colorado Springs, CO 80933
719-262-3150
chamillard@cs.uccs.edu

ABSTRACT
This paper describes one method for identifying *global variables* in Ada procedures and adding them to the parameter list of appropriate procedures. Global variables, ones that are visible to the entire program, are considered harmful to the maintainability of software systems because of the *side effects* that can result. A tool has been developed based on the Ada Semantic Interface Specification (ASIS) that can be used to identify global variables and re-engineer Ada procedures. By adding global variables to the parameter lists of appropriate procedures, the variables can be removed from the globally visible data space and introduced into the local declaration space.

Categories and Subject Descriptors
D.2.6 [**Software Engineering**]: Programming Environments – *programmer workbench.*

General Terms: Languages, Verification.

Keywords: Global variables, ASIS, SPARK.

1. INTRODUCTION
Software systems can often spend 70% of their life span in the maintenance phase of the Software Development Life Cycle. Certain software development practices have become recognized as harmful to the maintainability of software systems. One such practice is the use of *global variables*. This practice of making data visible to the entire software system is particularly harmful because of the *side effects* that can result when several different software modules are changing the same global variable.

This paper explains one method for identifying and removing global variables from Ada programs. By using a static analysis tool we have developed using the Ada Semantic Interface Specifi-cation (ASIS) [2,3,7], we are able to identify global variables in procedures and add them as parameters of the appropriate procedures. By changing these variables to parameters of the procedure, we are introducing them into the local data space of the procedure and removing them from the globally visible data space. This removes the possibility of side effects in the resulting code and improves the maintainability of the re-engineered Ada software system.

Similarly, the SPARK programming language [4] recognizes the harmful nature of global variables and, while not prohibiting global variables, requires the SPARK programmer to annotate any global variables being used. The tool we have developed can be used to identify and annotate these global variables automatically in SPARK programs.

As is desirable with any re-engineering project, our tool does not change the *semantics* of the legacy Ada code, i.e. the code produced from our tool is *functionally equivalent* to the legacy code. All the transformations performed on the legacy Ada code are semantics-preserving transformations.

The paper will first discuss some related work, define global variables in more detail, and then explain how we identify them in Ada procedure declarations and procedure calls. Our tool does not currently process functions or function calls. Next, we explain how the tool adds SPARK annotations to the Ada procedures, if desired. Then we discuss how global variables are added to the parameter list of the appropriate procedures. We conclude with future directions for this research.

SIGAda 2004, November 14-18, 2004, Atlanta, Georgia, USA.

2. RELATED WORK
The identification of global variables is a well defined process and is discussed in many programming language textbooks [5,11]. Many re-engineering tools have been developed that identify global variables in legacy programs [9,12]. From our examination of these tools it does not appear that any of these tools change the Ada code to remove global variables. Identifying global variables has also been useful in software metrics [6] and extracting objects from legacy code [8,10].

With the recent advent of the SPARK language, the identification and removal of global variables has also become more desirable. Global variables in SPARK are not prohibited, but they must be annotated properly in order for the information flow analysis of the SPARK Examiner to work properly. As legacy Ada code is re-engineered into SPARK code, our tool can be used to identify and remove any global variables from the software system. If an organization chooses not to remove the global variables, our tool can also be used to identify and annotate the code with the *global* annotation as required for the Examiner.

3. GLOBAL VARIABLES
In this section we define the term *global variable* in more depth as explained in fundamental programming language theory. Each variable in an Ada program must be declared at some point. Ada language rules define the *scope* of the variable [1] as the portion of

the program in which that variable may be referenced, i.e. in which the variable is *visible*. Sebesta [11] explains the difference between local, non-local, and global variables. *Local variables* are those variables declared within the scope of a program unit or block. *Non-local* variables are visible in the sub-program but are not declared there, i.e. they are declared outside the scope of the sub-program. *Global variables* are those non-local variables that are also visible to the entire program.

In Ada, we can distinguish variables further by pointing out the use of *package variables*. With the advent of the software engineering practice of Abstract Data Types (ADTs), certain packages are built with variables that are global to all procedures in that package. Procedures within the ADT are then used to access the package variables and provide a clean, secure interface to the variables. This is considered a beneficial practice and the global visibility of package variables is essential to the success of ADTs. For these reasons, we do not consider package variables harmful if they are used properly.

For example, an ADT that implements a stack is very useful. The stack variable can be encapsulated in the ADT as a package variable. The programmer can define an interface to the package variable via procedures such as *pop* and *push*. The ADT provides an interface to the stack variable via the procedures, which access the package variable as a non-local variable. This is desirable as long as the only procedures that access the stack variable are those procedures in the outermost scope of the package. A procedure nested within another procedure should not access the stack variable as a non-local variable.

For simplicity, we present an example that is smaller than a stack package to demonstrate our re-engineering tool. Consider the program shown in Figure 1. Variable X is a package variable defined in the specification of package One_Global. It is beneficial for the procedure Outera to have access to the variable X even though X is not declared in the local scope of Outera or passed as a parameter to Outera. This is standard practice when building an ADT. However, consider the use of variable X in the procedure Innerb. In this procedure, X is still visible because it is a global variable, but Innerb is not in the outermost scope of the package, it is nested within the scope of Outera. This highlights the improper use of the package variable X as a global variable in the procedure Innerb. Using package variable X in the procedure Outera is admissible, but using the package variable X in the procedure Innerb is not.

```
package One_Global is
   X : Integer := 10;
   procedure Outera(A : in out Integer);
end One_Global;

package body One_Global is
   procedure Outera ( A : in out Integer ) is

      procedure Innerb ( B : in out Integer ) is
      begin
         B := X;     -- X is a global var
         X := X + 1;
      end Innerb;

   begin
      Innerb(A);
   end Outera;
end One_Global;
```

Figure 1. Example of global variable X

Given this discussion, we can distinguish between the proper versus improper use of package variables as global variables. Using package variables in procedures that are in the outermost scope of packages can be a beneficial and desirable programming practice for ADTs. Using package variables within nested procedures places them in the category of "global variables" and is a harmful programming practice because of possible side effects. The work presented in this paper focuses on the identification and removal of global variables, but recognizes the value and benefit of using package variables properly.

Although is it possible to write an Ada procedure that is not contained within a package, all the examples in this paper involve packages. In procedures not in packages, the same definitions for the scope of a variable apply, but obviously package variables are not a consideration. Our tool works on these procedures as well.

One assumption we have made in our work is that all programs we pass to our tool already compile properly, i.e. there are no syntax errors in the programs that we process. Since Ada uses *static scoping*, the scope of a variable can be determined at compile time. Using syntactically correct programs and static analysis has greatly contributed to the identification of global variables and the re-engineering of Ada procedures.

3.1 Identifying Global Variables

The first step in our process is to identify global variables in Ada procedures. As shown in Figure 1, the variable X in the procedure Innerb is a non-local variable. To determine if X is a global variable, we consider the scope of its declaration. If a variable being used in a procedure is declared in the local scope it is not global and is ignored. If the variable is passed into the procedure as a parameter, it is not global and is ignored. If the variable is not declared in the local scope and is not a parameter to the procedure, then the declaration of the variable is examined. The only permissible case in this scenario is that the variable is a package variable being used in the outermost scope of the package. In any other case, the variable is considered a global variable.

For example, consider the block diagram in Figure 2 that depicts the scoping of package One_Global (shown in Figure 1).

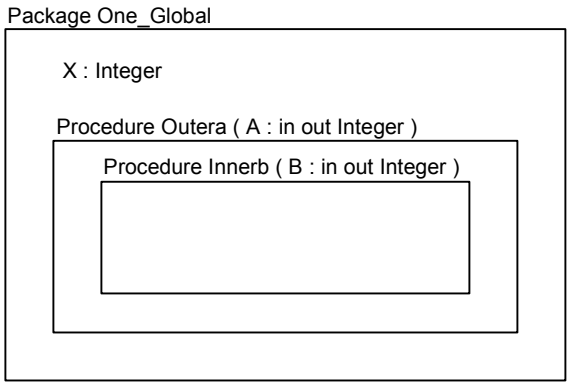

Figure 2 – Block diagram of scope nesting

As Figure 2 shows, the package One_Global defines a scope at the outermost level. The variable X and the procedure Outera are both in this outermost scope, which we designate as scope level zero. The procedure Innerb, however, is nested within the scope of procedure Outera and is at scope level one. This scoping and

nesting information is crucial to the identification of global variables.

As part of the re-engineering process, it is necessary to record the definition and reference sets for each global variable. A variable is *defined* in a statement if that statement assigns a new value to that. The definition set or DEF set is built for each statement collecting those variables defined by the statement. Our tool builds the DEF set for an entire procedure as part of the global variable processing. Similarly, a variable is referenced in a statement if it appears in any part of the statement. The reference set or REF set is built collecting variables referenced in a statement. Our tool also builds the REF set of an entire procedure.

In our tool, a *symbol table* is used to maintain the information about the declaration of variables, their enclosing scope, and the level of nesting of the declaration. To determine if a variable is global, the declaration information for any non-local variables is retrieved from the symbol table. If the declaration is at the package level, which is nesting level zero, it is a package variable. These variables may be accessed properly by procedures at the outermost level of nesting, such as procedure Outera. However, if the nesting level of the procedure using the variable is greater than zero, then the variable is identified as a global variable.

All global variables identified in a procedure are collected into a set of variables. In order to re-engineer the harmful use of global variables, our tool provides two options. The first option is to annotate the global variable using a SPARK annotation. Specifically, the *global* annotation can be added to the procedure that is using the global variable. The second option is to include the variable in the parameter list of the procedure that is using the variable. This implies that any calls to the procedure must also now include this new parameter in the call. The following sections discuss these two options in more detail.

3.2 Annotating Global Variables

After a global variable has been identified in a procedure, one option is to annotate the global variable using the proper SPARK annotation. The general form of the SPARK *global* annotation is shown below [4].

```
--# global <mode> <variable name>;
```

The "--#" symbols indicate that this is a SPARK annotation. In this annotation the *variable name* and *mode* of the global variable are listed after the keyword "global".

To determine the mode of the global variable we examine how the variable is used in the procedure. Using the terminology introduced in our discussion of DEF and REF sets, a global variable that is only referenced and not defined, i.e. is only in the REF set, is an IN mode variable. A global variable that is defined and not referenced, i.e. only in the DEF set, is an OUT mode variable. A global variable that is referenced and defined, i.e. in both the REF and DEF sets, is an IN_OUT mode variable. We recognize that this variable could be an OUT mode variable that is referenced in the body of the procedure, as allowed in Ada 95; however, identifying it as an IN_OUT mode variable is the most conservative approach. The completed *global* annotation is added the procedure that is using the global variable.

For example, as shown in Figure 1, the variable X is a global variable being used in the procedure Innerb. For the procedure Innerb, the REF set includes B and X since both variables are referenced in the statements of the procedure. The DEF set for Innerb also includes both B and X since both variables are defined by statements in the procedure. Given this information the following *global* annotation is added to the procedure Innerb to meet this particular SPARK requirement.

```
--# global in out X
```

Since the global variable X appears in both the DEF and REF sets for the Innerb procedure, the mode of the variable is determined to be IN_OUT mode.

Figure 3 shows the updated code with the proper SPARK annotations included.

```
package One_Global is
   X : Integer := 10;
   procedure Outera(A : in out Integer);
end One_Global;

package body One_Global is
   procedure Outera ( A : in out Integer ) is

      procedure Innerb ( B : in out Integer ) is
      --# global in out X
      begin
         B := X;     -- X is a global var
         X := X + 1;
      end Innerb;

   begin
      Innerb(A);
   end Outera;
end One_Global;
```

Figure 3 – Global SPARK annotations

As we have shown, our tool is useful for identifying and annotating global variables in SPARK code.

3.3 Re-engineering Global Variables

The second option after a global variable has been identified in a procedure is to re-engineer the procedure and add the variable to its list of parameters. Any calls to this procedure must be updated as well. After this re-engineering, the variable will be in the local scope of the procedure rather than a global variable used inappropriately in that procedure.

For example, consider the code shown in Figure 1. The variable X has been identified as a global variable in the procedure Innerb. To re-engineer this code, X must be added as a formal parameter in the declaration of the Innerb procedure. It must also be added as an actual parameter in any calls to the procedure. The only call to Innerb is found in Outera.

Figure 4 shows the re-engineered code. In the figure, the global variable X has been added as a parameter to the Innerb procedure. The call to this procedure has also been updated to include variable X as an actual parameter of the procedure call. This example shows the changes performed to re-engineer the code and remove the global variable.

Once a global variable has been identified (see Section 3.1), information is needed to proceed with the re-engineering. To ensure the semantics of the legacy code is preserved, the identifier and data type are gleaned from the original declaration of the global variable. These are used to build the new formal parameter for the procedure using the global variable. As is done for the SPARK global annotation, the mode of the parameter is determined automatically from the DEF and REF sets for this procedure. A global variable

that is only in the REF set is built as an IN mode parameter. A global variable that is only in the DEF set is built as an OUT mode parameter. A global variable that is in both the REF and DEF sets is built as an IN_OUT mode parameter.

```
package One_Global_New is
   X : Integer := 10;
   procedure Outera ( A : in out Integer );
end One_Global_New;

package body One_Global_New is
   procedure Outera ( A : in out Integer ) is

      procedure Innerb (
            B : in out Integer;
            X : in out Integer  ) is
      begin
         B := X;
         X := X + 1;
      end Innerb;

   begin
      Innerb(B => A, X => X);
   end Outera;
end One_Global_New;
```

Figure 4 – Global added as parameter

Again, we recognize that parameters identified as IN_OUT mode may actually be OUT mode parameters which are reference in the body of the procedure, but as with SPARK global annotations, we take the most conservative approach. The new actual parameter needed for the re-engineered procedure call is also built using information from the original declaration, which ensures functional equivalence.

```
package Two_Globals is
   X : Integer := 10;
   Y : Integer := 0;
   procedure Outera ( A : in out Integer );
end Two_Globals;

package body Two_Globals is
   procedure Outera ( A : in out Integer ) is

      procedure Innerb (  B : in out Integer ) is

         procedure Innerc (
               C : in out Integer ) is
         begin
            C := Y;
            Y := Y + 1;
         end Innerc;

      begin
         B := B + 1;
         Innerc(C => B);
      end Innerb;

   begin
      Innerb(B => A);
   end Outera;

end Two_Globals;
```

Figure 5 – Double nested procedure Innerc

Another case to consider is an access to a global variable that requires changing two procedures in order to remove inappropriate uses of global variables, as shown in Figure 5.

In the figure, there is a package variable Y that is referenced in the procedure Innerc. Since the procedure Innerc is not in the outermost scope of the package, the variable Y is identified as a global variable. To re-engineer this program, the variable Y must be added to the formal parameter list for the procedure Innerc. It must also be added as an actual parameter of the call to Innerc found in the Innerb procedure. However, adding this variable Y to the procedure call in Innerb now introduces Y as a global variable in the procedure Innerb. The variable Y is a non-local package variable, but Innerb is not in the outermost scope of the package. Figure 6 shows this intermediate, incomplete result.

```
package Two_Globals_new is
   X : Integer := 10;
   Y : Integer := 0;
   procedure Outera ( A : in out Integer );
end Two_Globals_new;

package body Two_Globals_new is

   procedure Outera (
         A : in out Integer ) is

      procedure Innerb (
            B : in out Integer) is  ← need Y here

         procedure Innerc (
            C : in out Integer;
            Y : in out Integer ) is
         begin
            C := Y;
            Y := Y + 1;
         end Innerc;

      begin
         B := B + 1;
         Innerc(C => B, Y => Y); ← Y is global!
      end Innerb;

   begin
      Innerb(B => A);  ← need Y here
   end Outera;

end Two_Globals_new;
```

Figure 6 – Intermediate result, global Y in Innerb

To re-engineer this procedure, the variable Y must be added to the formal parameter list of the procedure Innerb. It must also be added to the actual parameter list of the call to Innerb found in Innera. Adding the variable Y to the procedure call in Innera does not introduce Y as a global variable in Innera. The variable Y is a non-local package variable, but the procedure Outera is in the outermost scope level of the package. This use of the package variable Y is allowed. The re-engineering of the Ada code is complete.

Figure 7 shows the re-engineered code for the package Two_Globals. In this code, the variable Y has been added as a formal parameter in the declarations of the Innerc and Innerb procedures. The variable has also been added as an actual parameter in the calls to these procedures found in procedure Innerb and Innera, respectively. Adding Y as a formal parameter to Innerc introduced it into the local scope of Innerc. Adding Y as an actual parameter in the procedure call to Innerc introduced Y as a global variable in Innerb. This required adding Y as a formal parameter to Innerb and as an actual parameter in the call to Innerb propagating Y out to the scope of procedure Innera. Innera is at the outermost scope of the package, so Y is not considered a global variable in Innera and the propagation of Y is complete.

4. USING ASIS

As explained by the ASIS Working Group:

> "ASIS is an interface between an Ada environment as defined by ISO/IEC 8652:1995 (the Ada Reference Manual) and any tool requiring information from this environment." [2]

The static analysis information provided by the ASIS tool was invaluable to this project. ASIS includes procedures and functions that can be used to access information found in the Abstract Syntax Tree (AST) generated at compile time for Ada programs. The statement-level information gleaned from the AST was instrumental in determining the scope of variables during global variable identification. ASIS provides AST traversal routines that we used in identifying global variables in assignment statements and procedure calls.

```
package Two_Globals_new is
   X : Integer := 10;
   Y : Integer := 0;
   procedure Outera ( A : in out Integer );
end Two_Globals_new;

package body Two_Globals_new is

   procedure Outera (
         A : in out Integer ) is

      procedure Innerb (
            B : in out Integer;
            Y : in out Integer ) is

         procedure Innerc (
            C : in out Integer;
            Y : in out Integer ) is
         begin
            C := Y;
            Y := Y + 1;
         end Innerc;

      begin
         B := B + 1;
         Innerc(C => B, Y => Y);
      end Innerb;

   begin
      Innerb(B => A, Y => Y);
   end Outera;

end Two_Globals_new;
```

Figure 7 – Double nested procedures re-engineered

5. FUTURE DIRECTIONS

We have completed a prototype tool that can either annotate legacy Ada code with the SPARK *global* annotation or include global variables in the parameter lists of procedures. To extend this work, other SPARK annotations such as the *own* annotation could be automatically inserted into the package specification.

Another extension could be to identify global variables in functions and function calls. Our tool currently includes only procedures and procedure calls in the re-engineering process. This could be extended to add annotations to functions or add the global variable to the function parameter list and the function call.

With the power of the ASIS tool, we have been able to successfully analyze and re-engineer legacy Ada code. With our prototype tool, the user has the option to automatically annotate Ada code or remove global variable and introduce them into the local scope of Ada procedures.

6. REFERENCES

[1] Ada Language Reference Manual. Version 6.0 21 December 1994. Copyright (C) 1992,1993,1994,1995 Intermetrics, Inc.

[2] *ASIS Basic Concepts*. Retrieved June 3, 2003, from www.acm.org/sigada/wg/asiswg/basics.html, 1998.

[3] *ASIS Implementation, Products, and Services*. Retrieved June 3, 2003, from www.acm.org/sigada/wg/asiswg/products.html#Products , 1998.

[4] Barnes, J. *High Integrity Software, The SPARK Approach to Safety and Security.* c2003, Praxis Critical Systems Limited. Addison-Wesley, London UK.

[5] Ghezzi, C. and Jazayeri, M. *Programming Language Concepts*, c1982, John Wiley and Sons, New York, NY.

[6] Henry, S.; Kafura, D.: *Software Structure Metrics Based on Information Flow*. IEEE Transactions on Software Engineering, 7(1981)5, pp. 510-518

[7] ISO/IEC 15291:1999. Information technology -- Programming languages -- Ada Semantic Interface Specification, 1999.

[8] Liu, S.S. and Wilde, N. Identifying objects in a conventional procedural language: An example of data design recovery. In *Conference in Software Maintenance*, pp. 266–71. IEEE Computer Society Press, 1990.

[9] Panorama's Solution. Retrived 30 Apr from http://www.softwareautomation.com/www/reengin.htm

[10] Sahraoui H. A., Lounis, H., Melo, W., and Mili, H. A Concept Formation Based Approach to Object Identification in Procedural Code. In *Automated Software Engineering Journal,* Volume 6 No 4, Kluwer Academic Publishers, 1999, pp. 387-410.

[11] Sebesta, R. *Programming Language Concepts*. c2003, Addison Wesley, Reading MA.

[12] TROOP database re-engineering. Retrieved on 30 Apr 2004 from http://www.ercim.org/publication/Ercim_News/enw23/-castelli.html