AdaSlicer: An Ada Program Slicer

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
The ability to slice a program or parts of a program is useful in a variety of domains, including re-engineering, dependency analysis, test case generation, test coverage evaluation and debugging. Although a variety of static analysis tools are available to support these activities for Ada programs, none of these tools provide a program slicing capability. Fortuitously, the Ada Semantic Interface Specification (ASIS) provides access to the Ada program information required to slice the program. This paper describes our use of ASIS to implement an Ada program slicer, called AdaSlicer.

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

General Terms: Languages, Verification.

Keywords: Program slicing, ASIS.

1. INTRODUCTION
When we slice a program, we extract the components of the program that can affect the value of a particular variable. Program slicing is a well defined process (see [16]) that projects the behavior from the original program that is required to produce the value of a variable. The resulting behavior is collected into a new program that is a slice of the original program.

Program slices can be used in a variety of ways. For example, a maintainer considering a change to a particular statement in the program can generate program slices to identify the other components of the program that could be affected by the change. A tester generating or examining test cases, particularly those using dataflow coverage criteria [11], can more easily consider the components of the program affected by the test cases that address a particular variable by examining the appropriate program slices. A programmer trying to debug a program could identify the variables with erroneous values and use program slices to isolate the program statements that affect those variables.

SIGAda'03, December 7–11, 2003, San Diego, California, USA.

Because program slices can be generated using information available at compile time, program slicing is classified as a static analysis technique. Numerous static analysis tools have been developed for Ada programs, but none of the tools that we know of provide the capability to generate program slices for the program under analysis. A tool providing a program slicing capability for Ada programs would therefore be useful for those who maintain or test Ada programs.

The Ada Semantic Interface Specification (ASIS) can be used to examine compile-time information about an Ada program. Since program slicing is a static analysis technique, the information provided by ASIS is sufficient for generating program slices. Although ASIS has been standardized for some time [8], tool implementation using ASIS is still of interest to the Ada community. We implemented the program slicer, called AdaSlicer, described in this paper using the information provided by ASIS.

In this paper, we discuss related work, program slicing in general, how we used ASIS as the foundation of our program slicing tool, our progress to date, and future efforts that will use AdaSlicer.

2. RELATED WORK
Program slicing was originally described by Weiser [16]. Program slicing tools have been implemented for a variety of programming languages including C [15, 17], Java [7], Oberon-2 [10], and others. Program slicing tools for Ada, however, do not appear to be available.

Dependency analysis for Ada programs is provided in the ProDAG system [13], which has also been integrated with the TAOS system [12] to support test case generation and coverage analysis. The tool described in this paper uses ASIS to generate program slices, while the ProDAG system does not use ASIS and does not provide explicit program slicing capabilities. Other researchers have also developed methodologies to support dependency analysis of Ada programs (see [9] for example), though we have not discovered any that provide program slicing capabilities using the ASIS.

Several Ada program analysis tools have been developed using the ASIS. The ASIS Working Group [5] provides links to ASIS-based products. Examples include static analysis tools that collect program metrics and identify coding standards violations [3] and other tools that collect information about program characteristics to support quality assessments [1]. Another tool extracts object information and UML diagrams from Ada code [2].

3. PROGRAM SLICING

In general, program slicing is a projection of behavior from an original program into a new program called a *program slice* [16]. Program slicing is a static analysis process that relies on information about which variables and, consequently, program statements are required to produce a single variable called the *slice variable*. For each statement, the variables that are defined by that statement are collected into a DEF set and the variables referenced in the statement are collected into a REF set. A variable is *defined* in a statement if the variable can be assigned a new value in that statement. For example, variables on the left-hand-side of an assignment statement are defined by the statement. Any variables on the right-hand-side of an assignment statement are *referenced* in that statement and, by definition, appear in the REF set for the statement.

The program slices produced by AdaSlicer are considered to be *conservative* slices because they may include more than the minimal statements required to produce the slice variable. More extensive data flow analysis could be used to remove statements that are overshadowed by later statements that redefine variables in the relevant set. For simplicity, the data flow analysis in AdaSlicer is not this extensive, but the slices are guaranteed to produce the correct value of the slice variable as required by the definition of a program slice.

To produce a program slice, analysis begins at the last statement in the program and proceeds up to the first statement. The slice variable is placed in a *relevant set* that includes all variables relevant to the program slice. Any statement that defines a variable in the relevant set is included in the program slice. Any variables referenced in statements included in the slice are added to the relevant set since they can affect the definition of the slice variable. This process continues until all statements are analyzed. A more complete, formal algorithm is provided by Weiser [16].

3.1 Processing Statements

When processing *assignment* statements, a variable is defined if it appears on the left-hand-side of the statement. Programs with assignment statements can easily be sliced using AdaSlicer. For example, consider the program shown in Figure 1.

```
procedure Assignscalar is
   A : Integer := 0;
   B : Integer := 1;
   C : Integer := 2;
   D : Integer := 3;
   E : Integer := 4;
begin
   C := A + B;          -- statement #1
   D := B * 2;          -- statement #2
   E := D * 3;          -- statement #3
end Assignscalar;
```

Figure 1. Original program

This program includes several assignments statements and variables that depend on the value of other variables. As an example, consider the processing required for a slice on variable C. Initially, the relevant set includes only the variable C. Processing begins with statement #3 and continues up through statement #1. Since variable C is defined by statement #1, this statement is needed in the slice. The variables A and B are added to the relevant set since now any statements that define A and B will affect the value of C. Since statement #1 is the last statement to be processed in the procedure, processing is complete. The program slice produced for the variable C is shown in Figure 2.

```
procedure Assignscalar_C is
   A : Integer := 0;
   B : Integer := 1;
   C : Integer := 2;
begin
   C := A + B;
end Assignscalar_C;
```

Figure 2. Slice produced for variable C

Notice in the slice, the variables A, B, and C have been properly declared. As a final step of building a program slice, we include the original declarations of any variables in the relevant set. The program slice produced for variable E is shown in Figure 3.

```
procedure Assignscalar_E is
   B : Integer := 1;
   D : Integer := 3;
   E : Integer := 4;
begin
   D := B * 2;
   E := D * 3;
end Assignscalar_E;
```

Figure 3. Slice produced for variable E

Here, the value of E is dependent on the value of variable D. The value of variable D is dependent on the value of variable B. The two assignment statements are included in the slice and the variables B, D, and E are also included in the slice. Figure 3 shows how the dependencies between assignment statements are captured by AdaSlicer.

When processing *if-then* statements, each path of the statement is recursively analyzed to determine if any statements in the path affect the value of variables in the relevant set. If any such statements are found in any of the paths, the if-then statement is included in the slice. The conditional expression of the if-then statement is included intact and the variables from the conditional expression are added to the relevant set. For example, Figure 4 shows an Ada program with a simple if-then-else statement.

```
procedure Ifthenelsescalar is
   C : Integer;
   D : Integer;
   E : Integer;
begin
   C := 0;
   D := 0;
   E := 0;
   if C = 2  then
      D := D + 1;
   else
      E := E + 1;
   end if;
end Ifthenelsescalar;
```

Figure 4. If then else statement

Figure 5 shows the slice produced for the variable D. Note that in Figure 5, the else clause is not included in the slice.

```
procedure Ifthenelsescalar_D is
   C : Integer;
   D : Integer;
begin
   C := 0;
   D := 0;
   if C = 2 then
      D := D + 1;
   end if;
end Ifthenelsescalar_D;
```

Figure 5. Slice produced for variable D

This is a design decision built into AdaSlicer that removes the else clause from the if-then statement if there are no statements in the else path to be included in the slice. Figure 6 shows the slice produced for the variable E.

```
procedure Ifthenelsescalar_E is
   C : Integer;
   E : Integer;
begin
   C := 0;
   E := 0;
   if C = 2 then
      null;
   else
      E := E + 1;
   end if;
end Ifthenelsescalar_E;
```

Figure 6. Slice produced for the variable E

In this slice, none of the statements from the if-path of the original if-then statement need to be included in the slice. Our design decision was to include a null statement in the then-clause of the slice in this situation. It is not trivial to remove the then-clause in this case because it would require the Boolean expression of the original if-then statement to be negated. With our solution, the original Boolean expression is used and the null statement is included in the then-clause.

When processing if-then statements with elsif paths, special care is taken to include a minimal set of clauses. For example, Figure 7 shows an Ada program that includes an if-then statement with three elsif paths.

Since an if-then statement with elsif paths can be converted to a functionally equivalent if-then statement with nested if-then-else statements, our approach was to process elsif paths in the same way we processed if-then statements, as described above. By starting with the statement that is most deeply nested, we treat the else-path if the entire if-then-else statement as the else clause of the last elsif-path. We work up through the nesting, processing each nested if-then-else statement as described above. In Figure 7, for example, processing begins with the else clause, which is part of the nested if-then statement #4. If there are statements in the else clause that are required in the slice, then the entire if-then statement #4 is required in the slice. Since this if-then is nested inside the else clause of if-then #3, the else clause for if-then #3 is also needed in the slice. If-then #3 is nested in if-then #2, so if-then #2 is needed, and, of course, if-then #2 is nested in if-then #1, so if-then #1 is needed. This, in effect, cascades up the nested if-then statements resulting in all the if-then statements being included. This would be the case if we slice on variable "J", for example, and the resulting slice is shown in Figure 8.

```
procedure Elsiftest is
        F : Integer := 0;
        G : Integer := 0;
        H : Integer := 0;
        I : Integer := 0;
        J : Integer := 0;
        T : Integer := 0;
        W : Integer := 0;
        Z : Integer := 0;
begin
   W := W - 5;
   if F = 3 then       -- if #1
        G := G + 1;
   elsif H = 4 then    -- if #2, nested
        I := I + 2;
   elsif T = 7 then    -- if #3, nested
        T := T + 1;
   elsif W = 1 then    -- if #4, nested
        Z := -3;
   else                -- else clause
        J := J + 3;
   end if;
   W := W + 1;
end Elsiftest;
```

Figure 7. Elsif program

```
procedure Elsiftest_J is
   F : Integer := 0;
   H : Integer := 0;
   J : Integer := 0;
   T : Integer := 0;
   W : Integer := 0;
begin
   W := W - 5;
   if F = 3 then
      null;
   elsif W = 1 then
      null;
   elsif T = 7 then
      null;
   elsif H = 4 then
      null;
   else
      J := J + 3;
   end if;
end Elsiftest_J;
```

Figure 8. The slice produced for variable J

Note that we could have built a single if statement for the above example by manipulating the conditions appropriately. In our example, the condition for the new if statement would be

```
((F /= 3) and (W /= 1) and (T /= 7) and (H /= 4))
```

Instead, we decided that it would be more effective to retain the same structure as that contained in the original code. We believe this will make it easier for programmers, maintainers, and testers to see the relationships between the program slices and the original code.

If there are no statements from the original else clause (as shown in Figure 7) that are needed in the slice, then the else clause is not needed in the slice. It can be removed from if-then #4. If there are statements in the then clause of if-then #4 that are required in the slice, then if-then #4 will be included in the slice minus the else clause. This cascades up the nested if-then statements including all the nested statements above #4. The interesting case is when there are no statements in the else clause of #4 or in the then clause of #4. In this case, if-then #4 does not need to be included in the slice. This means the else clause for #3 does not need to be included in the slice. If there are no statements in the then clause of #3, then #3 does not need to be included in the slice, and so on. As an example,

consider slicing on variable "I". Figure 9 shows the slice produced from slicing on variable "I".

```
procedure Elsiftest_I is
   F : Integer := 0;
   H : Integer := 0;
   I : Integer := 0;
begin
   if F = 3 then
      null;
   elsif H = 4 then
      I := I + 2;
   end if;
end Elsiftest_I;
```

Figure 9. The slice produced for variable I

As shown in Figure 9, only if-then statements #1 and #2 are included in the resulting slice since if-then statements #3 and #4 do not affect the value of the variable I.

When processing CASE statements, a different approach is needed than the one used for if-then statements with elsif-paths. A CASE statement is functionally equivalent to an if-then statement with elsif-paths; however, the Boolean expression used in the CASE statement's when-clauses is based on a single variable. Because of this, we were not able to remove any of the branches of the CASE statements, but instead included null statements in branches that did not have any statements to be included in the slice. If none of the branches of the CASE statement included any statements required for the slice, then the CASE statement was not included in the slice. Figure 10 shows an example of an Ada CASE statement.

```
procedure Casescalar is
   K : Integer := 0;
   L : Integer := 0;
   N : Integer := 0;
   O : Integer := 0;
   Q : Integer := 0;
begin
   case K is
      when 5 =>
         L := L + 1;
      when 6 =>
         N := N + 1;
      when 7 =>
         O := O + 1;
      when 8 =>
         Q := Q + 1;
      when others =>
         Q := Q + 2;
   end case;
end Casescalar;
```

Figure 10. A CASE statement

In this statement, the CASE expression is based on the variable K and each of the when-clauses is for the different values of K. To illustrate why all the when-clauses must be included, consider the slice on variable Q in Figure 11.

In this slice, the variable Q is defined in the "when 8" clause and also in the "when others" clause. If the "when 7" clause had been removed, and the value of K was 7 when this slice was executed, Q would actually be changed and incremented by 2. This is not correct because in the original CASE statement, the value of Q was not changed when the value of K equals 7. Therefore, all CASE when-clauses are included in the slice and null statements are included in when-clause that do not included any statements required for the slice.

```
procedure Casescalar_Q is
   K : Integer := 0;
   Q : Integer := 0;
begin
   case K is
      when 5 =>
         null;
      when 6 =>
         null;
      when 7 =>
         null;
      when 8 =>
         Q := Q + 1;
      when others =>
         Q := Q + 2;
   end case;
end Casescalar_Q;
```

Figure 11. The slice produced for variable Q

When processing *loop* statements, special care is taken to extract the correct statements required to produce the value of the slice variable. Consider, for example, the while loop shown in Figure 12.

```
procedure Looptest is
   C : Integer := 0;
   D : Integer := 0;
   E : Integer := 0;
begin
   C := 0;
   while C <= 10 loop
      D := D + E;     -- loop statement #1
      E := E + C;     -- loop statement #2
      C := C + 1;     -- loop statement #3
   end loop;
end Looptest;
```

Figure 12. An Ada while loop

In this loop when we slice on the variable C, processing begins with statement #3 and proceeds up through statement #1. In this case, the only statement that affects the value of C is loop statement #3. It is the only statement that is needed inside the loop when we slice on the variable C. Figure 13 shows the slice generated for variable C.

When we slice on the variable D, however, the processing is not as straightforward. Again, processing will start with statement #3 and work up through statement #1. When we process statements #3 and #2, the relevant set will include only variable D. These statements do not affect the value of the variable D, so at this point they do not appear to be needed in the slice. When we process statement #1 which defines D, we include this statement in the slice and add the variable E to the relevant set as required when processing an assignment statement. However, now that E

```
procedure Looptest_C is
   C : Integer := 0;
begin
   C := 0;
   while C <= 10 loop
      C := C + 1;
   end loop;
end Looptest_C;
```

Figure 13. The slice produced for variable C

is in the relevant set and the loop might possibly repeat the statements inside the loop, we need to include statement #2 in the slice because it affects the value of E. So far, our statement processing has been from bottom up, but for loops we need a different approach. Whenever we process statements inside a loop, we repeatedly process the statements until we are sure that we have included all the statements that affect variables in the relevant set. As we identify statements that are needed in the slice, we add variables to the relevant set. We can use this fact to detect when processing of the statements in a loop is complete.

Our algorithm for processing statements inside a loop is as follows. We process statements in the loop from the bottom up. If a statement is to be included in the slice we add variables to the relevant set as needed. When we are done with the statements inside the loop, we compare the relevant set at the end of the processing of the loop to what it was at the beginning of the processing of the loop. If variables have been added to the relevant set, i.e. the relevant set is different at the end than it was at the start, then we process the statements in the loop again from bottom to top. We continue to process the statements until the relevant set remains unchanged from start to finish of processing.

For example, in Figure 12 when we slice on the variable D, processing begins at statement #3 and continues up to statement #1. The relevant set at the start of processing includes only the variable D, but at the end of processing it includes both D and E. We then process the statements again from statement #3 to statement #1. In this pass, we see that statement #2 affects the value of E, so we include it in the slice along with statement #1.

```
procedure Looptest_D is
   C : Integer := 0;
   D : Integer := 0;
   E : Integer := 0;
begin
   C := 0;
   while C <= 10 loop
      D := D + E;
      E := E + C;
      C := C + 1;
   end loop;
end Looptest_D;
```

Figure 14. Slice produced for variable D.

We add C to the relevant set as required for an assignment statement. The relevant set at the start of processing included D and E, but at the end of processing it includes C, D, and E. We processin the statements again from bottom to top. In this pass, we see that statement #3 affects the value of C, so we include it in the slice along with statements #1 and #2. There are no additional variables to add to the relevant set in this pass, so now the relevant set is the same at the end of processing the loop as it was at the start of processing the loop. We now stop processing the statements in the loop and return statements #1, #2, and #3 as the statements needed inside the loop statement that will be included in the slice for variable D. Figure 14 shows the slice for variable D.

3.2 Inter-procedural Slicing

So far we have dealt with procedures that do not call other procedures. Weiser [16] defines two forms of program slicing: intra-procedural and inter-procedural. As the name suggests, intra-procedural slicing produces program slices by considering statements within the procedure. Inter-procedural slicing considers calls to other sub-programs and may project those sub-programs into their own program slices. Inter-procedural slicing is used in AdaSlicer when processing procedure call statements. If the procedure call statement defines a variable in the relevant set (through an out or in out parameter), the called procedure must be sliced to collect the statements that affect that variable. For example, consider the two procedures shown in Figure 15.

```
package body Proc_Scalar is
   procedure Casescalar (
         A : in out Integer;
         M : in     Integer;
         D : in out Integer  ) is
   begin
      A := 0;
      D := 0;   -- slice var
      case M is
         when 1 =>
            D := D + 1;
         when 2 =>
            A := A + 1;
         when others =>
            null;
      end case;
   end Casescalar;

   procedure Loopscalar (
         B : in out Integer ) is
      D : Integer := 0;
   begin
      B := 0;   -- slice var
      Casescalar(
         A => B,
         M => 18,
         D => D);
   end Loopscalar;
end Proc_Scalar;
```

Figure 15. Procedure call statements

As shown in the figure, procedure Loopscalar calls procedure Casescalar passing in the variable B and the value 18 as actual parameters. If we are slicing the Loopscalar procedure on the variable B, we need to consider the call to Casescalar. By looking at the declaration of the procedure Casescalar, we see that the formal parameter A is an in out parameter. In the call to Casescalar, variable B is passed as the actual parameter corresponding to the formal parameter A. Since A is an in out parameter, the variable B is defined by the procedure call to Casescalar. When slicing on B in Loopscalar, we must use inter-procedural slicing to extract the statements from Casescalar that affect the value of the variable B. The algorithm we use for inter-procedural slicing is to first find which variables are defined by a procedure call statement. If the procedure call statement does not define any of the variables in the relevant set, it is not included in the program slice. If it does define a variable in the relevant set, such as the variable B, a new program slice must be built that includes statements from the called procedure that affect the value of the defined variable and a new procedure call statement must be built to call the new program slice.

In order to slice the called procedure, we must find the correct variable to slice on in the called procedure. For example, as shown in Figure 15, the variable B is defined by the call to Casescalar. However, we do not slice Casescalar on the variable B. We need to find the formal parameter in Casescalar that corresponds to the defined variable in the procedure call statement. As shown in Figure 15, the variable A is the formal parameter in Casescalar that corresponds to the actual parameter B in the procedure call

statement. Once the formal parameter is determined, we slice on that formal parameter in the called procedure. Figure 16 shows the slice produced by slicing on the variable A in Casescalar.

```
procedure Casescalar_A (
      A : in out Integer;
      M : in     Integer) is
begin
   A := 0;
   case M is
      when 1 =>
         null;
      when 2 =>
         A := A + 1;
      when others =>
         null;
   end case;
end Casescalar_A;
```

Figure 16. The slice produced for variable A

As shown in Figure 16, only the statements that affect the value of A have been included in the slice. Since the variable D is not included in the slice, it is not included as a formal parameter for the slice Casescalar_A. Now that the called procedure has been sliced on the variable A, we can build the new procedure call statement to be included in the procedure Loopscalar. Figure 17 shows the completed slice for the procedure Loopscalar with the new procedure call statement produced by inter-procedural slicing.

```
procedure Loopscalar_B (
      B : in out Integer ) is
begin
   B := 0;
   Casescalar_A(
      A => B,
      M => 18);
end Loopscalar_B;
```

Figure 17. The slice produced for variable B

Notice in Figure 17, that the call to Casescalar has been replaced by a call to the new slice procedure Casescalar_A. The parameters in the call to Casescalar_A are matched with the new definition of the sliced procedure, which means B is passed to the formal parameter A and 18 is passed to the formal parameter M. The parameter D is not included in the new procedure call statement.

The inter-procedural slicing algorithm that we have implemented is a recursive algorithm that slices each called procedure if the procedure call statement defines a variable in the relevant set. This algorithm is guaranteed to terminate only if there is no recursion or mutual recursion in the procedures being sliced. Each called procedure is sliced using the same algorithms for the statements already discussed above. If there is not a procedure call statement in the called procedure, the recursion reaches the base case, returns the sliced procedure, and builds the new procedure call statement for the sliced procedure.

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." [4]

ASIS is distributed in public versions and also by private vendors. There are particular versions of ASIS that correspond to versions of the GNAT Ada compiler. As the authors quickly discovered, the version number for ASIS much exactly match the version number for the GNAT compiler. This is due to the tight coupling of ASIS to the Abstract Syntax Tree (AST) produced using the -gnatt compiler option. The AdaSlicer tool was built using version 3.15a1 of ASIS and the version 3.15a1 of the GNAT compiler. The ASIS distribution comes with documentation, tutorials, source files and templates for building applications. Once unzipped, the ASIS files must be compiled using the correct version of the compiler and installed as an Ada library on the development machine.

We have observed that, for novice ASIS programmers, the relationships between the ASIS source code files is somewhat difficult to initially understand. The examples provided in the ASIS distribution are good initial starting points for developing an application, and the most useful part of the distribution was the template provided for traversing the AST produced by GNAT. This template demonstrated how to peel back the layers of ASIS all the way down to the statements and their components stored in the AST.

In ASIS, the outermost layer consists of a *context*. Any ASIS application must associate the application with a context in order to access the ASTs. Each context contains any number of *compilation units*. Compilation units model the procedures, functions, and packages from the program being analyzed by the application. Each compilation unit includes a model of the declaration of the unit and these declarations include *elements* of the program. An element is the basic unit of ASIS and includes such things as statements, expressions, and exception handlers. It also includes more compiler oriented items such as statement paths, clauses, definitions and declarations. Once the ASIS application has reached the level of *element*, the application can access the statements from the program being analyzed and accomplish static analysis of the program.

In the AdaSlicer tool, we tunnel down to the element level in which we reach the procedure body declaration of the original procedure. From that point we apply the program slicing algorithm to the statements in the procedure body to produce the program slice. Because we require DEF and REF information for each statement, we use the queries provided by ASIS to build the DEF and REF sets for each statement. These queries are very powerful. For example, ASIS provides a link to the declaration of the called subprogram given a procedure call statement. ASIS also provides associations between the actual parameters in procedure calls and the corresponding formal parameters in the procedure declaration. To determine DEF and REF sets for procedure calls, we also need access to the modes of the parameters; we use ASIS queries to identify those modes.

To produce a slice using the AdaSlicer tool, the original program must first be compiled using the -gnatt option of the GNAT complier to produce the AST for the program. The ADT file is then input to the AdaSlicer application along with the slice variable in order to analyze the statements and produce a slice. AdaSlicer outputs the program slice using "put" statements that print the statements needed for the slice and the enclosing procedure body. The new procedure is named using the following convention:

<original program name>_<slice variable>.adb.

The output from AdaSlicer is captured and stored in a text file with a corresponding name.

5. FUTURE DIRECTIONS

Our initial motivation for developing a program slicing capability for Ada programs was to support re-engineering efforts for those programs. Because current re-engineering activities often include conversion of legacy imperative code to object-oriented implementations, we wanted to facilitate the automated extraction of objects from imperative code. One automated technique for doing this kind of object extraction is called the Parameter-Based Object Identification (PBOI) methodology [14], which requires program slicing. Our next step is to apply the PBOI methodology to a legacy Ada system that consists of 13 files, 6,196 lines of code, of which 1,305 end with a semi-colon.

The current incarnation of the tool is invoked from the command line. While this serves our immediate needs, it is clear that a more reasonable user interface is required to improve ease of use. In fact, we plan to incorporate use of this tool in both an undergraduate software engineering course at the U.S. Air Force Academy and a graduate software engineering course at UCCS. To effectively do so, we will need to add a Graphical User Interface to the tool, either by converting it to a Windows application or by using the RAPID [6] GUI design tool.

6. REFERENCES

[1] *Ada Analyzer*. Retrieved June 4, 2003, from www.littletree.com/analyzerdata.html.

[2] *Ada Reverse Engineering for Rational APEX*. Retrieved June 4, 2003, from www.markv.com/markv.com/pr01.htm, 2002.

[3] *AdaStat Product Info*. Retrieved June 4, 2003, from www.adastat.com/productinfo.htm.

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

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

[6] Carlisle, M.C. and Maes, P. RAPID: a free, portable GUI design tool. In *Proceedings of the 1998 Annual ACM SIGAda International Conference on Ada*, Washington D.C., November 1998, pp. 158-164.

[7] Dwyer, M. B., Corbett, J.C., Hatcliff, J., Sokolowski, S., and Zheng, H. *Slicing Multi-Threaded Java Programs: A Case Study.* Tech Report KSU CIS TR 99-7.

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

[9] Moser, L.E. Data dependency graphs for Ada programs. *IEEE Transactions on Software Engineering*, SE-16(5):498-509, May 1990.

[10] *Program Slicing*. Retrieved June 4, 2003 from www.ssw.uni-linz.ac.at/Research/Projects/ProgramSlicing.

[11] Rapps, S. and Weyuker, E.J. Data flow analysis techniques for program test data selection. In *Proceedings of the Sixth International Conference on Software Engineering*, Tokyo, Japan, September 1982, pp. 272-278.

[12] Richardson, D.J. TAOS: testing with analysis and oracle support. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, Seattle, Washington, 1994, pp. 138-153.

[13] Richardson, D.J., O'Malley, T.O., Moore, C.T., and Aha, S.L. Developing and integrating ProDAG in the Arcadia environment. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, Tyson's Corner, Virginia, 1992, pp. 109-119.

[14] Sward, R.E. and Hartrum, T.C. Extracting objects from legacy imperative code. In *Proceedings of the 12th IEEE International Conference on Automated Software Engineering*, Incline Village, Nevada, November 1997, pp. 98-106.

[15] *The Unravel Project*. Retrieved June 4, 2003, from http://hissa.nist.gov/unravel/, 1998.

[16] Weiser, M. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352-357, July 1984.

[17] *The Wisconsin Program-Slicing Tool, Version 1.1*. Retrieved June 4, 2003, from www.cs.wisc.edu/wpis/slicing_tool/, 2000.