Optimizing the SPARK Program Slicer

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

Leemon C. Baird III
Department of Computer Science
U.S. Air Force Academy, CO
719-333-8321
leemon.baird@usafa.af.mil

ABSTRACT

Recent trends in software re-engineering have included tools to extract program slices from existing Ada procedures. One such tool has already been developed that extracts program slices from SPARK procedures along with a proof that the functionality of the original procedure is equivalent to the functionality of the collection of resulting slices. This paper extends this work by showing how assumptions in the proof can cause inefficiencies in SPARKSlicer and by presenting alternatives that optimize out the inefficiencies. The original proof is modified to show that the SPARK program slicer still produces functionally equivalent program slices from SPARK procedures with these optimizations.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments – *Formal Methods.*

General Terms

Languages, Formal Methods.

Keywords

Program slicing, ASIS.

1. INTRODUCTION

In recent years, the technique known as program slicing has been applied to several areas of software engineering including program understanding, software maintenance, re-engineering, testing, debugging, and reuse [13, 9, 6]. *Program slicing* is a well defined process that extracts the statements from an existing program that are required to produce a value from the program [12]. As program slicing is used more and more in software engineering it becomes pertinent to answer the question of whether or not the collection of program slices maintain the functionality of the original program. Specifically, given the same inputs, will the collection of program slices produce the same outputs as the original code? It has already been shown [10] that for one such program slicer based on the SPARK programming language, the collection of program slices are functionally equivalent to the original code. However, some assumptions were made in order to properly handle input/output parameters in the proof of functional equivalence. These assumptions, if implemented, would result in very inefficient code. This paper extends this work by providing an alternate way of handling input/output parameters and showing that the proof of functional equivalence can still be made.

Previous work in the area of program slicing includes slicers for various languages such as C [11, 13], Java [3], Oberon-2 [4], and others. A program slicing tool called AdaSlicer has also been developed for the Ada programming language [7]. The slicing tool presented in this paper builds on the AdaSlicer tool producing slices for programs written in the SPARK programming language [2], which finds its roots in Ada. Our program slicer for SPARK called SPARKSlicer is written using the Ada Semantic Interface Specification (ASIS) [1].

2. PROGRAM SLICING

This paper assumes that the reader already has an understanding of *program slicing*. For more detailed descriptions see Weiser [12] or Sward and Chamillard [7]. In general, program slicing is a projection of behavior from an original program into a new program called a *program slice* [12]. 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 definition set or *DEF* set and the variables referenced in the statement are collected into a reference or *REF* set. A variable V is *defined* in a statement S if V is assigned a new value in S. For example, variables on the left-hand-side of an assignment statement are defined by the statement. A variable V is referenced in a statement S if any part of S includes V. For example, 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.

To produce a program slice for a procedure P, the statements S_1 to S_n included in P are analyzed. The analysis begins with S_n and proceeds up to S_1 since variables in statements that appear after a statement S_i can be affected by S_i. The slice variable V_s is placed in a relevant set R that includes any variables relevant to the program slice. If the *DEF* set for statement S_i includes any variables in R, then S_i is added to the program slice. All variables in the *REF* set for S_i are added to R since the definition of V_s is derived from the variables contained in S_i. This process continues until all statements S_1 to S_n are analyzed. The resulting collection of statements S_j to S_k is a subset of S_1 to S_n that preserves the sequential ordering of the statements from P. Weiser [12] defines the program slice S_j to S_k as a *projection* $Pr(V_s)$ of the behavior from P required to produce V_s. The definition of program slicing given here is taken from the more complete, formal algorithm provided by Weiser [12].

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

```
package Gather_Summary_Info_Pkg is
   type Integer_Array is array(1..100) of Integer;
   procedure Gather_Summary_Info (
         Num_Students : in     Integer;
         Min_Choice   : in     Integer_Array;
         Max_Choice   : in     Integer_Array;
         Lowest_Min   :    out Integer;
         Highest_Max  :    out Integer;
         Increment    : in out Integer         );
   --# derives Lowest_Min from Min_Choice,
Num_Students &
               Highest_Max from Max_Choice,
Num_Students;
   --# pre (Num_Students >= 0)
   --# post for all E in range 1..Num_Students =>
                Lowest_Min <= Min_Choice(E) &
            for all E in range 1..Num_Students =>
                Highest_Max >= Max_Choice(E);
end Gather_Summary_Info_Pkg;
```

Fig. 1. Package specification for SPARK code example

```
package body Gather_Summary_Info_Pkg is

   procedure Gather_Summary_Info (
         Num_Students : in     Integer;
         Min_Choice   : in     Integer_Array;
         Max_Choice   : in     Integer_Array;
         Lowest_Min   :    out Integer;
         Highest_Max  :    out Integer;
         Increment    : in out Integer         ) is
   begin
      Increment := Increment + 1;
      Lowest_Min:= 1000;
      Highest_Max:= 0;
      for Student in 1 .. Num_Students loop
         if ( Min_Choice(Student) < Lowest_Min)
then
            Lowest_Min:= Min_Choice(Student);
         end if;
         if ( Max_Choice(Student) > Highest_Max)
then
            Highest_Max:= Max_Choice(Student);
         end if;
      end loop;
   end Gather_Summary_Info;

end Gather_Summary_Info_Pkg;
```

Fig. 2. Package body for SPARK code example

3. PROGRAM SLICING

A prototype tool called SPARKSlicer [10] has been developed that produces program slices from SPARK code. Given a package specification and body that includes at least one procedure P, SPARKSlicer extracts a program slice $Pr(V_s)$ from P. Program slices are produced only for output parameters of P, so the user indicates which procedure and parameter to slice on. The result is the program slice $Pr(V_s)$ built as a new procedure that includes those statements needed to produce V_s. The name of the new procedure is built by appending the name of the parameter to the name of the procedure. Only those parameters needed for the new procedure are included in the parameter list for the program slice. Similarly, only those variables needed for the new procedure are included as local variables of the new procedure.

Figure 1 shows the specification file for an example SPARK program. Figure 2 shows the package body for this example SPARK code. This example includes a package that contains one procedure Gather_Summary_Info. This procedure contains three input parameters, two output parameters, and one input/output parameter. Since slicing is done on the values produced from a procedure, we can slice on the Lowest_Min parameter, the Highest_Max parameter, and the Increment parameter.

```
procedure Gather_Summary_Info_Lowest_Min (
      Num_Students : in     Integer;
      Min_Choice   : in     Integer_Array;
      Lowest_Min   :    out Integer         ) is
begin
   Lowest_Min:= 1000;
   for Student in 1 .. Num_Students loop
      if ( Min_Choice(Student) < Lowest_Min) then
         Lowest_Min:= Min_Choice(Student);
      end if;
   end loop;
end Gather_Summary_Info_Lowest_Min;
```

Fig. 3. Slice produced for Lowest_Min

Figure 3 shows the slice built for the Lowest_Min parameter. This new procedure contains only those statements from Gather_Summary_Info that are needed to produce the parameter Lowest_Min. Only the Num_Students and Min_Choice input parameters are needed to produce the output parameter Lowest_Min.

```
procedure Gather_Summary_Info_Highest_Max (
      Num_Students : in     Integer;
      Max_Choice   : in     Integer_Array;
      Highest_Max  :    out Integer         ) is
begin
   Highest_Max:= 0;
   for Student in 1 .. Num_Students loop
      if ( Max_Choice(Student) > Highest_Max) then
         Highest_Max:= Max_Choice(Student);
      end if;
   end loop;
end Gather_Summary_Info_Highest_Max;
```

Fig. 4. Slice produced for Highest_Max

Figure 4 shows the slice built for the Highest_Max parameter. This new procedure contains only those statements from Gather_Summary_Info that are needed to produce the parameter Highest_Max. Only the Num_Students and Max_Choice input parameters are needed to produce the output parameter Highest_Max.

```
procedure Gather_Summary_Info_Increment (
      Increment_In  : in     Integer;
      Increment_Out :    out Integer         ) is
   Increment_Local : Integer := Increment_In;
begin
   Increment_Local := Increment_Local + 1;
   Increment_Out := Increment_Local;
end Gather_Summary_Info_Increment;
```

Fig. 5. Slice produced for Increment

Figure 5 shows the slice built for the Increment parameter. This new procedure contains only those statements from

Gather_Summary_Info that are needed to produce the parameter Increment. Notice that a different approach has been taken by SPARKSlicer for this parameter since it is an input/output parameter. As explained in [10], the input/output parameter is replaced with one input parameter, Increment_In, and one output parameter, Increment_Out. This is done to enforce the copy-in, copy-out semantics for the slice, which is needed for the proof of functional equivalence [10]. Each occurrence of Increment in the statements of the procedure is replaced with a local variable, Increment_Local, which is initialized to the value of Increment_In. After all processing has been completed in the procedure, the value of Increment_Local is copied into the output parameter, Increment_Out, thus enforcing copy-in, copy-out semantics.

As explained in [10], this process of copying the new input parameter, e.g. Increment_In, into the new local variable, e.g. Increment_Local, will be inefficient. The new local variable is also copied into the new output parameter, e.g. Increment_Out, which incurs another inefficient copy operation. As pointed out in [10], this will preserve correctness even when using arrays and records, but is inefficient. Two copy operations must be done each time the program slice is called, and for large record structures or arrays, this can be a quite CPU-intensive operation that may take a considerable amount of time. The next section of this paper will propose an alternative approach for handling input/output parameters.

```
procedure Gather_Summary_Info_Glue (
      Num_Students : in     Integer;
      Max_Choice   : in     Integer_Array;
      Highest_Max  :    out Integer;
      Min_Choice   : in     Integer_Array;
      Lowest_Min   :    out Integer;
      Increment    : in out Integer        ) is
   Increment_Local : Integer := Increment;
begin
   Gather_Summary_Info_Highest_Max (
      Num_Students, Max_Choice, Highest_Max);
   Gather_Summary_Info_Lowest_Min (
      Num_Students, Min_Choice, Lowest_Min);
   Gather_Summary_Info_Increment (
      Increment_Local, Increment);
end Gather_Summary_Info_Glue;
```

Fig. 6. Example of glue code

Now that the original procedure has been sliced into three new procedures, it is useful to have some standard way of calling the three procedures to perform the same operation as the original, unsliced procedure. Figure 6 shows a short procedure consisting of *glue code* which calls the three slices and which will behave identically to the original procedure (except perhaps for the time and memory required). In this glue code procedure, the three slices are called one after the other with the proper actual parameters to match the formal parameters of the slices. The parameters needed for these calls to the slices are included as formal parameters to the glue code procedure.

Note that due to the original SPARKSlicer algorithm the parameters for the Gather_Summary_Info_Increment slice are handled in a unique way. The original parameters to the glue code procedure are identical to the original procedure. The value of the input/output parameter, Increment, is stored in the local variable, Increment_Local. This variable is passed to any of the slices that have been built with Increment_In as an input formal parameter. Notice this local variable is passed to Gather_Summary_Info_Increment as the input actual parameter. For any slices that have been built with Increment_Out as an output formal parameter, the parameter Increment is passed as the actual parameter. Gather_Summary_Info_Increment includes Increment as an output actual parameter.

All this special processing for the input/output parameters, such as Increment, is needed to ensure that the other slices do not interfere with the original value of Increment and that there is only one value of Increment produced from the procedure Gather_Summary_Info_Glue. This special handling of input/output parameters is needed for the proof of functional equivalence as explained below. The following section explains an alternative approach for handling the input/output parameters so as not be be so inefficient.

There are many features of the SPARK language that make proving functional equivalence possible. We will not go into great depth about the exact nature of these features, but instead refer the reader to Sward and Baird [10] for a discussion of how these features are useful for proving functional equivalence. The reader is further referred to Barnes [2] for a more thorough discussion of these features and the rationale for excluding them from the SPARK language.

4. AN ALTERNATIVE APPROACH

As pointed out in the previous section, the original handling of input/output parameters may result in very inefficient program slices. An alternative approach to handling these parameters is presented in this section. The approach relies on more robust static data analysis of the program slices that have been built from the original procedure.

For example, look closely at the glue code procedure that is shown in Figure 6. The input/output parameter Increment has been split into an input only parameter and an output only parameter. This is done to ensure that the order in which the program slices are called does not have any effect on the value of the parameter Increment. By building the REF and DEF sets for the program slices, we can see that the Increment parameter is in the REF or DEF set for only one of the program slices, i.e. Gather_Summary_Info_Increment. In this case, there is no need to perform the costly copy in and copy out operations as part of the slice. No matter what order the slices are called in, the value of Increment will have the proper value when it is referenced in a program slice. In this case, the program slice does not need to include the input-only and output-only parameters.

```
procedure Gather_Summary_Info_Increment (
      Increment : in out Integer        ) is
begin
   Increment := Increment + 1;
end Gather_Summary_Info_Increment;
```

Fig. 7. Slice produced for Increment

Figure 7 shows this new version of the program slice that can now safely be used with the other program slices. The glue code procedure would also be updated to call this modified slice with Increment as the only actual parameter. With this simple analysis of the REF and DEF sets for each of the slices, we have avoided two costly copy operations in the program slice.

It may also be the case that an input/output parameter of a slice appears in the REF and/or DEF set of more than one of the program slices. In order to keep the calling order of the program slices in the glue code an arbitrary order, we must ensure that the original value of the parameter is restored before each call to a program slice. In [10] the approach was to copy the value into the procedure so as not to destroy the original value.

An alternative approach is to maintain a hash table that records the changes that are made to the parameter. For a record or array, this can be at the element level. In this way, any changes to the parameter can be reversed before the next program slice is called by the glue code procedure. Storing changes to the parameter in the hash table is a much more efficient operation than wholesale copying of data structures.

5. PROVING FUNCTIONAL EQUIVALENCE

In order to prove functional equivalence, we rely on several of the SPARK language limitations. Since we are working with SPARK programs, we assume that there is no tasking or use of threads in the code. We assume there are no exceptions or exception handling. We assume there are no pointers and no dynamic heap allocation. We assume there is no recursive code in the program. We assume no aliasing of procedure parameter, i.e. a single variable cannot be passed as an actual parameter for both an input and output parameter in the same procedure call.

We also assume that there are no global variables in the code. Sward and Chamillard [8] have shown that any global variables in Ada code can be converted to parameters in the calling sub-programs. This can easily be extended to the SPARK language. We also assume no non-determinism in the code. For example, when we say a transformed procedure gives the same result as the untransformed, we'll ignore the possibility of the procedures reading the system clock and returning different times, or calling a true random number generator and return different random numbers. We also assume that all procedures eventually return, for all possible inputs.

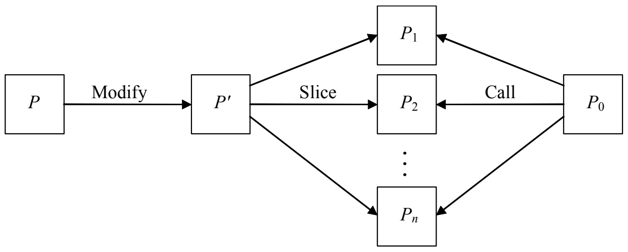

Fig. 8. Original Slicing and Glue Code

For the SPARKSlicer program, we assume that the slicer produces one procedure for each of the "out" or "in out" parameters in the original function, and that the created procedure will also have that parameter as part of its signature. We assume that each slice procedure created will have only a single "out" or "in out" parameter. We also assume that the slice procedure created for that output parameter is a correct projection of the behavior of the original procedure required to produce that parameter.

5.1 Algorithm for creating glue procedure

A SPARK procedure P with n "out" or "in out" parameters is transformed into $n+1$ procedures named P_0 through P_n. This is done by first transforming P into a procedure P', then running the slicer n times on P' to create P_1 through P_n, and finally creating the procedure P_0 which is functionally equivalent to P as shown in Figure 7. In the original algorithm, the procedure P' is identical to P except for the modifications to split input/output parameters into *X_in* and *X_out*.

With the additional static data checking that we propose in this paper, there is no need for the transformation into P'. Instead, the original procedure P, can be sliced into P_1 through P_n, as shown in Figure 8.

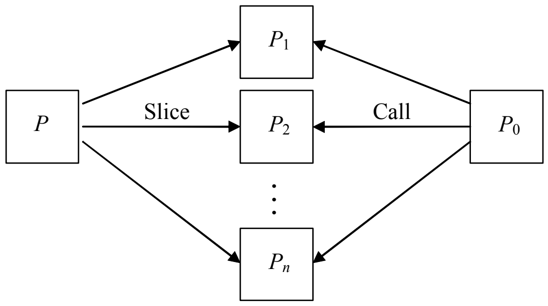

Fig. 9. Alternative Slicing and Glue Code

Additional processing is done in the procedure P_0 to ensure the proper values are passed to the program slices. As explained above, if an input/output parameter is in the REF and DEF sets for only one program slice, no additional processing is needed. In all other cases, a hash table is used to maintain a history of changes to the parameter and these changes are reversed before the next program slice is called.

Procedure P_0 calls each of the procedures P_1 through P_n, in an arbitrary order. When a procedure P_i has a formal parameter that matches a formal parameter of P_0, that parameter from P_0 is passed in as the actual parameter.

5.2 Definitions

The inputs to a procedure are those parameters that are either "in" or "in out". The outputs are those that are "out" or "in out". Two procedures are functionally equivalent if they both yield the same outputs whenever they are passed the same inputs. Weiser [12] defines a *program slice* as a subset of a program that preserves a specified projection of its behavior. We assume here that the slicer generates slices of a procedure that preserve one particular output as a function of its inputs.

$$P_0 \textit{ is functionally equivalent to } P \qquad \text{Theorem } (\mathbf{1})$$

Theorem 1 states that the procedure P_0, which calls procedures P_1 through P_n, is functionally equivalent to the original procedure P.

5.3 Proof

Since we are assuming that the slicer works correctly and because SPARK is designed to make aliasing impossible, then the procedures P_0 and P will return the same outputs when given the same inputs. This can be seen by considering the three types of parameters. The "in" parameters in P_0 are the same as in P and are not changed while it is running because they are "in" parameters in the P_i functions. Each "out" parameter of P_0 is only used by one of the P_i calls, because each P_i was formed by slicing on a different output.

Each "in out" parameter in P_0 is guaranteed to have the original value it had when passed into P because of the following. In the case that the "in out" parameter is in the REF and DEF sets of only one program slice, then the order of the P_i calls will not effect the value of the "in out" parameter since it is only changed by one of the program slices. In the case where the "in out" parameter appears in the REF and DEF sets for multiple program slices, the original value of the "in out" parameter is restored before each of the P_i calls by using the hash table to reverse any operations that had changed the value of the parameter. This is a more efficient way to return the parameter to its original value than copying it in its entirety.

In other words, P_0 is designed to prevent the P_i calls from interfering with each other. The calls can occur in any order, and each one will set one of the outputs from P_0. This ensures that P_0 returns the same values as P. Therefore P_0 and P must always return the same values when given the same inputs, and are therefore functionally equivalent.

6. CONCLUSION

In conclusion, given the features provided in the SPARK language, we have shown that the collection of slices produced from a procedure is functionally equivalent to the original procedure. If the glue code is built as described in this paper, then, as we have shown, functional equivalence is maintained. The prototype implementation of the SPARKSlicer has been built using the Ada Semantic Interface Specification (ASIS) interface [1]. Future research could rewrite this tool using the SPARK language and annotations. This future research would verify that the slicer is a correct implementation of Weiser's algorithm.

As program slicing is used in more and more software engineering applications, it is becoming more desirable to prove that these tools preserve functional equivalence. This paper provides an example of one such proof which shows that the collection of slices produced from our SPARKSlicer tool is functionally equivalent to the original procedure.

7. REFERENCES

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

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

[3] 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.

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

[5] *SPARK 95 – The SPADE Ada 95 Kernel*, copyright Praxis Critical Systems. Edition 4.1, Oct 2003.

[6] Sward, R.E. Extracting Functionally Equivalent Object-Oriented Designs from Legacy Imperative Code. PhD Thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, Sep 1997.

[7] Sward, R.E. and A.T. Chamillard, AdaSlicer: A Program Slicer for Ada. *Proceedings of the ACM International SIGAda 2003 Conference*, Dec 2003, San Diego, CA.

[8] Sward, R.E. and A.T. Chamillard, Re-engineering Global Variables in Ada, *Proceedings of the ACM International SIGAda 2004 Conference*, Nov 2004, Atlanta, GA.

[9] 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.

[10] Sward, R.E. and L.C. Baird III, Proving Functional Equivalence for Program Slicing in SPARK, *Lecture Notes in Computer Science 3555,* pp 105-114, Ada Europe, Jun 2005, York, England.

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

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

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