Using Program Slicing to Simplify Testing

Mark Harman and Sebastian Danicic
Project *Project*,
School of Computing,
University of North London,
Eden Grove, London, N7 8DB.
tel: +44 (0)171 607 2789
fax: +44 (0)171 753 7009
`e-mail: m.harman@uk.ac.unl`

Keywords: Slicing, Transformation, Robustness, Implicit Computation

Abstract

Program slicing is a technique for automatically identifying the statements of a program which affect a selected subset of its variables.

A large program can be divided into a number of smaller programs (its slices), each constructed for different variable subsets. The slices are typically simpler than the original program, thereby simplifying the process of testing a property of the program which only concerns the corresponding subset of its variables.

However, some aspects of a program's computation are not captured by a set of variables, rendering slicing inapplicable. To overcome this difficulty a program shall be rewritten in a self–checking form by the addition of assignment statements to denote these 'implicit' computations. Initially this makes the program longer. However, slicing *can* now be applied to the introspective program, forming a slice concerned solely with the implicit computation. The simplification power of slicing is then improved using program transformation.

To illustrate this approach, the implicit computation which dictates whether or not a program is robust shall be taken as an example. Whether or not a program is robust is not generally decidable, making the approach described here particularly appealing because the slices constructed are *approximate* answers to the undecidable question "Is the program p robust?".

1 Introduction

Slicing is a way of simplifying a program by focusing upon some subset of its variables. Having chosen a set of variables, the *slice set*, and a point in the program, the process of slicing consists of deleting any statement that cannot affect variables in the slice set at the chosen point in the program. The program so created, called a slice, captures the sub–component of the original program concerned with the chosen slice set and program point. For a given subject program, many slices may be constructed based upon the choice of a slice set and a program point. Each slice set defines a (usually distinct) slice.

The motivation for slicing derives from the fact that the slice of a program p is (usually) *simpler* than p, whilst *maintaining* the effect of p upon the slice set.

Program slicing was introduced in 1979 by Mark Weiser in his seminal PhD thesis (Weiser, 1979). Weiser's original algorithm consisted of finding a solution to a set of iterative data and control flow equations. More recent approaches treat slice construction as a graph reachability problem using the inverse of the Program Dependence Graph (Horwitz and Reps, 1992). This approach was first suggested by Ottenstein and Ottenstein[1] (Ottenstein and Ottenstein, 1984). The present authors have developed the first parallel slicing algorithm, closely based upon Weiser's original data and control flow equations (Harman et al., 1995b).

This paper is concerned with slicing in the static paradigm. The other two paradigms for slicing are the dynamic (Korel and Laski, 1988; Agrawal and Horgan, 1990; Mariam Kamkar and Fritzson, 1992; Gopal, 1991), in which a slice is constructed with respect to an initial state and the quasi–static (Venkatesh, 1991; Tip, 1995a), in which a slice is constructed with respect to a set of possible initial states.

Much of the literature on program slicing is concerned with improving the original algorithms for slicing introduced by Weiser (Weiser, 1979; Weiser, 1981; Weiser, 1984) and Ottenstein and Ott[enstein] (Ottenstein and Ottenstein, 1984) to cope non–trivial linguistic features such as pointers (Agrawal et al., 1991; Livadas and Rosenstein, 1994; Ernst, 1994; Tip, 1995a), procedures (Horwitz et al., 1988), arbitrary jump statements (Choi and Ferrante, 1994; Agrawal, 1994; Ball and Horwitz, 1992), concurrent programming (Cheng, 1993) and, by highlighting the parts of a program which have no effect upon the slicing criterion, side effects within expressions (Tip, 1995a; Ernst, 1994).

Slicing has many applications including measuring cohesion (Bieman and Ott, 1994; Ott and Thuss, 1993; Lakhotia, 1993; Harman et al., 1995a), algorithmic debugging (Shahmehri, 1991; Kamkar, 1993), re–engineering (Lui and Ellis, 1993; Simpson et al., 1993), component re–use (Beck and Eichmann, 1993), automatic parallelisation (Weiser, 1983), maintenance and debugging (Gallagher and Lyle, 1991; Lyle and Weiser, 1987), program integration (Horwitz et al., 1989), and assisted verification (Qi et al., 1988). Frank Tip (Tip, 1995b) provides an excellent and thorough survey of the paradigms and techniques for program slicing.

In general, the applications of slicing derive from the way in which it can be used as part of a 'divide and conquer' approach to program comprehension – a large program is better understood as a union of *smaller* programs (its slices), each of which capture some sub–component of the overall computation.

To illustrate, suppose a particularly rigorous testing approach is to be applied to the safety critical component of a system's behaviour, for example the raising of an alarm in the control system of a chemical plant. Slicing will remove statements which do not affect the alarm, thereby simplifying the process of testing and analysis of assertions concerning the alarm–raising mechanism.

The rest of this paper is organised as follows:-

In section 2 and 3 the concept of program slicing is described, showing how it can be used to

[1] Linda Ottenstein has changed her name to Linda Ott, and this is the name which appears on her more recent publications.

reduce testing effort. In sections 4, 5 and 6 a program transformation algorithm, $\mathcal{T}$, is defined. $\mathcal{T}$ adds assignments to a pseudo–variable, `robust`, which capture explicitly the, previously implicit, answer to the question of whether or not the program is robust. The existence of the pseudo–variable `robust` in the transformed, self–checking, program allows a 'robustness slice' to be constructed (according to an algorithm involving slicing and transformation introduced in section 7). Section 8 contains a case study, describing, in detail, the steps taken to produce a robustness slice. Section 9 shows that the question of whether or not a program is robust is undecidable, and discusses the implications of this result for robustness slicing. Section 10 discusses the relationship between the robustness slice and work on assertion based approaches to program testing and verification.

2 The Conventional Program Slice

Conceptually, slicing is attractively simple and easy to define – delete from a program all those statements which *cannot* affect the values of variables of interest.

Definition 2.1 presents an informal definition of a (static) program slice.

Definition 2.1 (Slice) A slice of a program p, at a line number n, with respect to a set of variables K, is a program p', constructed by deleting certain commands from p. The commands which may be deleted are those which can have no effect upon the values residing in any of the variables in K, when execution reaches line n. The pair (K, n) is known as the 'slicing criterion'.

In this paper the only slices considered will be those constructed to capture the final values of variables, and thus the slicing criterion shall be a set of variables alone, rather than the more traditional pair containing a variable set and point of interest within the program to be sliced.

Consider the simple loop program given in the left–hand box of figure 1. To investigate and analyse the termination of the loop a slice would be formed with respect to the slicing criterion $(\{\mathtt{i}\}, 6)$, giving the slice in the right–hand box of figure 1. For this small example, the simplification caused by slicing makes little difference. For larger subject programs the simplification can be quite spectacular, especially if the slicing algorithm is not restricted to producing slices which are a syntactic subset of the subject program. This possibility is explored further in section 8.1.

3 Slicing as Part of a Testing Strategy

Slicing can be viewed as a way of choosing an *arbitrary* modularisation of a program, each module being formed with respect to a slicing criterion. Of course, a well designed system will already be modularised. Good design, however, does *not* diminish the applicability of program slicing; slicing allows us to *choose many different* modularisation criteria, each suiting a particular line of enquiry. Returning to the chemical control system example, it may be that the software consists of a separate module for the code controlling each of the system's input sensors. This would give low coupling and high cohesion and so represent 'good design'. The raising of an alarm, however, will depend upon the behaviour of *parts* of *many* sensor modules. In this case, the modularisation of the overall system will therefore not *simplify* testing or analysis with respect to the alarm raising mechanism. Slicing will effectively allow us to 'remodularise' the program to create (for the purpose of analysis) an 'alarm control module'.

4 Implicit Computation

Suppose slicing is to be used to analyse the way the loop program of figure 1 indexes the array `a`, in order to ensure that the program is robust (i.e. that it does not attempt to index an array element which is 'out of range'). Unfortunately, since robustness is *not* denoted by a set of variables it will not be possible to form an appropriate slicing criterion; the robustness or otherwise of a program is *implicit*.

This problem is typical of a large class of related problems, where the aspect of the program under consideration is not 'stored' in a set of program variables. To make the approach advocated here more definite, the problem of robustness testing and analysis will be treated in some detail, but other implicit computations can be treated in a similar manner. Other implicit aspects of a program's behaviour include:

- its demand upon internal and external resources,
- its generation of errors,
- its communication and Input/Output behaviour.

The implicit nature of the robustness question can be overcome by transforming a program into a *self–checking* form. That is, assignments to a pseudo–variable are added so that a program computes aspects of its own implicit behaviour. Pseudo–variables are previously unused variables that are normal in all other respects. Initially, the introduction of pseudo–variables simply makes the subject program longer. However, slicing technology *is now* applicable to the self–checking version of the program. The slice constructed captures the program's effect upon its pseudo–variables thereby isolating the implicit computation in an explicit slice.

5 A Subset of the Language C

The algorithm for creating a 'robustness slice' operates on a subset of the `C` programming language, `SmallC`. The BNF syntax of `SmallC` is given in figure 2. A suitable definition is assumed to exist for the non–terminal symbols, `<char>`, characters (in single quotes), `<string>`, strings (in double quotes), `<id>`, the identifiers and `<num>`, the numeric constants of the language.

Side–effects present a problem for many conventional slicing algorithms (Weiser, 1984; Agrawal, 1994; Jiang et al., 1991), which do not attempt to deal with expressions which may cause side–effects. The side–effects are removed using a source–to–source transformation, which rewrites a `SmallC` program in a slightly different language, `TinyC` , in which side–effects are prohibited. `TinyC` is created from `SmallC` by replacing the clause `<exp>;` in the production rule for `<stat>` with the clause `<assign>;`, and removing the clauses `<assign>` and `<post>` from the production rule for `<exp>`.

6 Introducing a Pseudo–Variable

To form a self–checking version of a program which explicitly computes its own robustness, only one pseudo–variable, a 'flag' variable, called `robust`, is required.

The transformation $\mathcal{T}$ takes a side–effect free statement, c, and produces an introspective statement c'. After the execution of c', the variable `robust` will contain either:

- 1 (the program c is robust) or
- 0 (the program c is not robust or c' will terminate abnormally with an index out of range error).

The transformation function $\mathcal{T}$ is described in figure 3. Quine's Quasi Quotes ($[\![\ldots]\!]$) are used to enclose syntactic constructions (Stoy, 1985). Such constructions contain variables which stand for arbitrary members of syntactic domains. The syntactic variables and the domains they represent are listed at the top of figure 3.

The transformation $\mathcal{T}$ is defined using the auxiliary transformation functions $\mathcal{R}$, $\mathcal{E}$ and $\mathcal{C}$. The function $\mathcal{R}$, takes an expression, e, and an array identifier, i, and returns code to update the pseudo–variable `robust`. The expression e is used to index the array i, so the program will be robust if (and only if) two criteria are met:

1. the program is currently robust,
2. the expression e is within the range of valid indices for the array i.

The first criterion is required because once an error has occurred the program must be considered to be in error, and the value of the flag `robust` must remain at zero, indicating that the program is not robust. The function $MAX(i)$ returns the largest valid index for the array i. This information can be obtained from the symbol table created when array declarations are encountered. In order to simplify the exposition, `SmallC` does not contain declarations. The minimum value for a valid index is always zero.

The function $\mathcal{E}$, takes an expression, e, and transforms it into a (possibly empty) statement sequence, c. The sequence c is created using $\mathcal{R}$ to construct appropriate tests for all array indices in c.

The function $\mathcal{C}$, takes a command, c, and transforms it into a command sequence c'. This sequence preserves the effect of the original program, but it also captures the robustness of the command c by means of appropriate assignments to the pseudo–variable `robust`. These are introduced using the function $\mathcal{E}$ applied to the expressions of c.

The function $\mathcal{T}$, takes a `TinyC` program and transforms it into a program which includes robustness checking code. $\mathcal{T}$ simply uses the function $\mathcal{C}$ to 'do the work', adding an initial assignment, `robust=1;`, reflecting the fact that, initially, the program *is* robust.

7 Creating a Robustness Slice

$\mathcal{T}$ is used as part of an algorithm to construct a slice concerned solely with the robustness of the subject program. The algorithm for creating such a robustness slice is presented in figure 4. The first step is to transform the subject program into a side–effect free form. $\mathcal{T}$ is then used to create a self–checking version of the side–effect free program.

7.1 Transformation

Often slicing the introspective program does not yield a significant reduction in size when compared with the original program. However, simple meaning preserving transformations can be applied to simplify the slice (often, quite considerably).

In figure 5, a suitable set of typical transformations is defined. These transformation rules are only partially correct with respect to termination. The transformation rules are written as axioms and rules of inference. A rule of the form: $\frac{A}{B \Rightarrow C}$ can be interpreted as "If A holds then the fragment B can be transformed to the fragment C"

The function, **REF** takes a side–effect free expression and returns the set of variables it references (called REF'ed variables in (Aho et al., 1986) and (Weiser, 1984)). The function, **DEF** takes a statement and returns the set of variables it defines (called DEF'ed variables in (Aho et al., 1986) and (Weiser, 1984)). The function $\mathbf{SUB}(E_1, I, E_2)$ returns the expression that results from substituting all occurrences of the variable I in the expression E_1, with the expression E_2. All expressions (including the result) being side–effect free.

The axioms of commutativity and idempotence are only applicable to side–effect free expressions. Strictly speaking, the axiom of commutativity should be written as a rule, since it may only be inferred that two arguments to an `&&` can be swapped if the resulting expression will not behave differently due to short circuit evaluation. To keep the presentation as clear as possible, such semantic complications have been ignored.

8 A Case Study

In this section the algorithm given in figure 4 is illustrated using the example program below, which counts the number of occurrences of alphabetic characters in an array:

Example: Original Program

```
printf("Enter a string") ;
scanf("%s",source);

i=0 ; while(i<=128) { result[i]=0; i++; }
i=0 ; while((x=source[i++])!='\0') result[x]++;

b=s=bc=0 ; sc=MAX+1 ; i='a' ;
while(i<128) { if (result[i]<sc) { s=i; sc=result[i]; }
               if (result[i]>bc) { b=i; bc=result[i]; }
               i++; }

printf("RESULTS\n");
printf("Character    No. of Occurrences");
i='a' ; while(i<='z')
          { printf("%c            %d\n",i,result[i]);
            i++; }
printf("Most  frequent : %c occurs with frequency %d\n",b,bc);
printf("Least frequent : %c occurs with frequency %d\n",s,sc);
```

The first step is to transform the program into a side–effect free form. In the case of the example program this transformation is relatively uncomplicated and it results in the equivalent side–effect free program given in step 1 below:

Example Step 1 (The Side–Effect Free Version)

```
printf("Enter a string") ;
scanf("%s",source);

i=0 ; while(i<=128) { result[i] = 0; i=i+1; }

i=0 ; while(source[i]!='\0') { x=source[i];
                               i=i+1;
                               result[x]=result[x]+1; }

b=0; s=0; bc=0 ; sc=MAX+1; i='a' ;
while(i<128) { if(result[i]<sc) { s=i; sc=result[i]; }
               if(result[i]>bc) { b=i; bc=result[i]; }
               i=i+1; }

printf("RESULTS\n");
printf("Character    No. of Occurrences");
i='a' ; while(i<='z')
          { printf("%c            %d\n",i,result[i]);
            i=i+1; }
printf("Most  frequent : %c occurs with frequency %d\n",b,bc);
printf("Least frequent : %c occurs with frequency %d\n",s,sc);
```

The pseudo–variable `robust` shall now be introduced to model the robustness characteristics of the program using the transformation function, $\mathcal{T}$. This creates the robustness self–checking program step 2 below (where $MAX(result) = 127$ and $MAX(source) = MAX$):

Example Step 2 (The Robustness Self–Testing Program)

```
robust = 1; /* starts out robust */
printf("Enter a string") ;
scanf("%s",source);

i=0 ; while(i<=128) { robust = robust && i>=0 && i<=127 ;
                      result[i] = 0;
                      i=i+1; }

i=0 ; robust = robust && i>=0 && i<=MAX ;
      while(source[i]!='\0') { robust = robust && i>=0 && i<=MAX;
                               x=source[i];
                               i=i+1;
                               robust = robust && x>=0 && x<=127;
                               robust = robust && x>=0 && x<=127;
```

```
                              result[x]=result[x]+1; }

b=0; s=0; bc=0 ; sc=MAX+1; i='a' ;
while(i<128)
  { robust = robust && i>=0 && i<=127;
    if(result[i]<sc) { s=i;
                       robust = robust && i>=0 && i<=127;
                       sc=result[i]; }
    robust = robust && i>=0 && i<=127;
    if(result[i]>bc) { b=i;
                       robust = robust && i>=0 && i<=127;
                       bc=result[i]; }
    i=i+1;}

printf("RESULTS\n");
printf("Character    No. of Occurrences");
i='a';
while(i<='z')
  { robust = robust && i>=0 && i<=127 ;
    printf("%c              %d\n",i,result[i]);
    i=i+1; }
printf("Most  frequent : %c occurs with frequency %d\n",b,bc);
printf("Least frequent : %c occurs with frequency %d\n",s,sc);
printf("robust = %d",robust);
```

The robustness self–checking program is *longer* than the original since it contains robustness information *explicitly* as an assignment to the variable `robust`, rather than as array indices for which the language semantics *implicitly* dictates the robustness characteristics.

Since the robustness of the program is now explicitly captured in the computation upon a single variable, the explicit version of the program may now be sliced with respect to this variable, focusing solely upon the robustness of the original program. In step 3, the self–checking program has been sliced with respect to the slicing criterion $(\{\texttt{robust}\}, L)$, where L is the last line of the program produced as step 2.

Example Step 3 (The Conventional Slice of the Transformed Program)

```
robust = 1;
scanf("%s",source);

i=0 ; while(i<=128) { robust = robust && i>=0 && i<=127 ;
                      result[i] = 0;
                      i=i+1; }

i=0 ; robust = robust && i>=0 && i<=MAX ;
      while(source[i]!='\0'){ robust = robust && i>=0 && i<=MAX;
                              x=source[i];
                              i=i+1;
                              robust = robust && x>=0 && x<=127 ;
```

```
                                  robust = robust && x>=0 && x<=127 ;
                                  result[x]=result[x]+1; }

  bc=0; sc=MAX+1;  i='a' ;
  while(i<128)
    { robust = robust && i>=0 && i<=127;
      if(result[i]<sc)
         { robust = robust && i>=0 && i<=127;
           sc=result[i]; }
      robust = robust && i>=0 && i<=127;
      if(result[i]>bc)
         { robust = robust && i>=0 && i<=127;
           bc=result[i]; }
      i=i+1; }

  i='a'; while(i<='z') { robust = robust && i>=0 && i<=127 ; i=i+1; }
```

8.1 Meaning–Preserving Transformations

Slicing creates rather a large slice, partly because $\mathcal{T}$ has a propensity to create repetitious code and partly because slicing is restricted to command deletion.

An example of the redundant code created by $\mathcal{T}$ occurs with statements like `a[i]=a[i]+1;` which generate *two* range checks for the variable `i`, where only one is required.

As an example of the restrictiveness of command deletion in program slicing, consider the program fragment below:

```
x = 1;
y = x+1;
z = 2*y+x;
```

If the program is sliced with respect to the final value of the variable `z`, then no lines can be deleted, because every statement of the program fragment contributes to the final value of `z`. However, the *simplest* program fragment which produces the same final value for `z` is simply:

```
z = 5;
```

It would be possible to elaborate considerably upon the transformation, $\mathcal{T}$, but the more complex $\mathcal{T}$ becomes, the more likely it is that an erroneous transformation step will be introduced. Rather than tinker with the definition of $\mathcal{T}$, *simple* meaning–preserving transformations shall be applied to the program produced by $\mathcal{T}$, each of which may be easily verified, but which in combination can yield significant slice simplification.

Some of these transformations are listed in figure 5. To see how they can be applied to a slice, consider the following fragment (taken from the body of the second loop in step 3):

```
robust = robust && i>=0 && i<=MAX ;
x = source[i] ;
i = i+1 ;
robust = robust && x>=0 && x<=127 ;
robust = robust && x>=0 && x<=127 ;
result[x]=result[x]+1;
```

Applying 'Push Assignment' (rule 2) twice to the assignment statement `i = i+1;` effectively moves this assignment forward two statement positions. As the two statements passed by the `i = i+1;` do not reference `i`, no substitution occurs.

Thus the program below is obtained:

```
robust = robust && i>=0 && i<=MAX ;
x = source[i] ;
robust = robust && x>=0 && x<=127 ;
robust = robust && x>=0 && x<=127 ;
i = i+1;
result[x]=result[x]+1;
```

Applying 'Push Assignment' twice to the statement `x = source[i];`, entails the substitution of the expression `source[i]` for `x` in each statement passed. Thus the program below is obtained:

```
robust = robust && i>=0 && i<=MAX ;
robust = robust && source[i]>=0 && source[i]<=127 ;
robust = robust && source[i]>=0 && source[i]<=127 ;
x=source[i] ;
i = i+1;
result[x]=result[x]+1;
```

Applying 'Unfold Assignment' (rule 3) to the first two statements, and then again to the first two statements of the resulting code fragment, has the effect of collecting together the effect of the first three assignments to `robust`, producing:

```
robust = robust && i>=0 && i<=MAX && source[i]>=0 && source[i]<=127
                                  && source[i]>=0 && source[i]<=127 ;
x=source[i] ;
i = i+1;
result[x]=result[x]+1;
```

Having coalesced the assignments to the variable `robust`, the idempotency axiom can be applied to simplify the boolean expression, giving:

```
robust = robust && i>=0 && i<=MAX && source[i]>=0 && source[i]<=127 ;
x=source[i] ;
i = i+1;
result[x]=result[x]+1;
```

Clearly, a method for deciding when to *stop* applying axioms such as the commutativity axiom is required to prevent non–termination. One way of addressing this issue is to use normal forms for programming language expressions, an approach borrowed from the study of computer algebra (Davenport et al., 1989).

Using the transformation rules in this way we obtain the simplified version of the conventional slice below:

```
robust = 1;
scanf("%s",source);

i=0 ; while(i<=128) { robust = robust && i>=0 && i<=127 ;
                      result[i] = 0;
                      i=i+1; }

i=0 ; robust = robust && i>=0 && i<=MAX ;
      while(source[i]!='\0'){ robust = robust && i>=0 && i<=MAX
                                              && source[i]>=0 && source[i]<=127;
                              x=source[i];
                              i=i+1;
                              result[x]=result[x]+1; }

bc=0; sc=MAX+1;  i='a' ;
while(i<128) { robust = robust && i>=0 && i<=127; i=i+1; }
i='a'; while(i<='z') { robust = robust && i>=0 && i<=127 ; i=i+1; }
```

Finally, if the transformed program is sliced with respect to the final value of `robust` once more, the (now unnecessary) assignments to the variables `x`, `bc`, `sc` and `result` will be removed, to produce step 4 below:

Example Step 4 (The Non–Conventional Slice)

```
robust = 1;
scanf("%s",source);

i=0 ; while(i<=128) { robust = robust && i>=0 && i<=127 ; i=i+1; }
i=0 ; robust = robust && i>=0 && i<=MAX;
while(source[i]!='\0')
  {robust = robust && i>=0 && i<=MAX && source[i]>=0 && source[i]<=127; i=i+1;}

i='a' ; while(i<128) { robust = robust && i>=0 && i<=127; i=i+1; }
i='a' ; while(i<='z') { robust = robust && i>=0 && i<=127 ; i=i+1; }
```

Step 4 is *not* a slice in the conventional sense of the word (Weiser, 1984). This is because the transformations, although *semantics* preserving, may not be *syntax* preserving; they do not simply *delete* statements from the original program, rather they *replace* original statements with equivalent statements. For a more detailed discussion of this issue see (Harman and Danicic, 1994a; Harman and Danicic, 1994b).

9 Discussion

9.1 Computability: A Motivation for the Robustness Slice Approach

Robustness is not a property of a program which can generally be computed, because the proposition "The program p is robust", is undecidable. To see that robustness is undecidable, consider the following code fragment. The fragment is a schema, where `f` and `g` are arbitrary functions and `a` is an array, the maximum permissible index of which is `MAX`.

```
scanf("%d",&x); if (f(x) != g(x)) a[MAX+1]=10;
```

This fragment will definitely be robust if the two functions `f` and `g` happen to be equivalent ($\forall x.f(x) = g(x)$), and will definitely not be robust if `f` and `g` are not equivalent. Therefore, if there exists an *automatic* procedure for deciding whether a program is definitely robust, there shall also exist an algorithm to decide whether two arbitrary functions are equivalent. As the question of whether two arbitrary functions are equivalent is known to be undecidable, the robustness question must, likewise, be considered to be undecidable. A similar argument demonstrates the undecidability of the other implicit computations listed in section 4.

If an attempt is made to implement a system for automatically deciding robustness, the user of such a system will, therefore, have to be content with a yes/no/maybe response – the undecidability of the question of robustness will often mean that such an implementation will answer 'maybe'.

The 'robustness slice' approach avoids this issue by returning a *program* rather than attempting to produce a *definitive answer* to the robustness question. Ideally, the algorithm for constructing a robustness slice will produce the program fragment:

```
robust = 1;
```

signifying that the subject program is robust, or it will produce:

```
robust = 0;
```

signifying that the subject program is not robust.

In 'maybe' cases however, the algorithm will *approximate* the complete yes/no information by producing a simpler program which stores either a zero or a one in the variable `robust`, but which does so using more than one program statement. Such a 'maybe' program constitutes an approximation to the two simplest (one line) programs.

Using the robustness slice approach it will thus be possible to produce complete information in the cases where robustness characteristics *can* be computed, but with the significant bonus, that where such information may not be computable, the algorithm shall be able to assist manual analysis by program simplification.

9.2 The Size of the Slices Produced

For the example program used in the case study, a robustness slice construction algorithm *should* be able to produce the *ideal* robustness slice, that is, the simplest possible slice (see Example: Final Step below), revealing *immediately* that the original program is *not* robust. Providing transformation rules which would produce such a slice is less difficult than it seems. In order to move from step 4 to the final step, all that is required is a transformation rule which exploits the fact that the boolean expression of a while loop is true at the outset of the execution of its body, and that the result of the boolean operation `&&`, is false whenever either operand is false. Of course, due to the fact that robustness is undecidable it will not always be possible to produce an ideal (one statement) slice, such as this.

Example: Final Step

```
robust = 0 ;
```

The table given in figure 6 gives the sizes of each program produced in the course of the case study, where size is the number of nodes in the program's Control Flow Graph. It is of the nature of the approach that the size of a program will increase as the result of side–effect removal and the introduction of the pseudo–variable, `robust`.

The hope is that in many cases the simplification produced by slicing and transformation will yield a robustness slice which is significantly shorter than the subject program. This will be an objective measure of simplification that can be used to assess the suitability of the approach on a case–by–case basis. Of course, the robustness slice will always be semantically very simple, as it is merely concerned with storing one of two values in a single variable. This alone, may make the robustness slice easier to analyse than the subject program. The principal task for further work rests upon finding the best transformation tactics with which to simplify the slices produced. It is upon the power of the slicing and simplification transformations that the applicability of the robustness slice approach critically depends.

9.3 More Complex Language Semantics

The language `SmallC` is extremely simple, and this simplicity hides some issues which arise when a larger subset of `C` is chosen. For example, when arrays may occur as the parameter to a function, the function definition need not specify the array bounds. Each call to the function can potentially pass an array of different length. In order to correctly assign to the pseudo–variable `robust`, an extra parameter to such function definitions will be required, giving the length of the array parameter. However, this extra parameter may be introduced merely by defining another clause in the definition of $\mathcal{T}$. A similar issue arises with dynamically allocated storage and with the exploitation of the array/pointer duality in `C`.

10 Related Work

Rosenblum (Rosenblum, 1995) has implemented a pre–processor system, called *APP*, which replaces the traditional `C` pre–processor. In addition to the normal `C` pre–processing tasks, *APP* recognises comments of the form `/*@ ... @*/` as containing assertions about the state of the program. An assertion takes the form of pre– and post– conditions at the functional level of abstraction. The pre–conditions capture the set of initial states in which the programmer expects a function's body to be executed. The post–conditions capture the set of final states in which the function will terminate. Two different post–conditions are permitted –

$$\texttt{promise}(E);$$

which delimits the set of admissible final states for the function body, and

$$\texttt{return}\ I\ \texttt{where}\ E$$

which constrains the possible values returned by the function.

The assertions recognised by *APP* are introduced *by the programmer*, and are checked at *run–time* by code added at source level by the *APP* pre–processor. The 'robustness slice' approach, by contrast, does not require the programmer to add the assignment statements which capture the program's robustness, rather these are added automatically by the transformation function $\mathcal{T}$. The programmer is then expected to analyse (at compile time) the 'robustness slice' produced by slicing and transformation.

David Flater and his colleagues (Flater et al., 1993) describe an extension of the `C` programming language, in which assertions concerning, *inter alia*, robustness, are *automatically* added to the `C` program to create a program which performs checks upon its behaviour at run–time. The process of introducing checking code for the integrity of array indexing is almost identical to the process of adding assignments to the pseudo–variable `robust`.

The principal difference between `Robust C` and the robustness slice approach is to be found in the way the self–checking program is used. A `Robust C` program is *executed*, with invalid array indexes causing the execution of the program to terminate with an appropriate error message. Notice that this is not the case with 'normal' `C` programs, where, as Flater and his colleagues point out, the existence of this form of defect may (often) cause no fault to occur at run–time.

The self–checking program produced by $\mathcal{T}$ is *not* produced with execution in mind. Indeed, executing the self–checking program produced by $\mathcal{T}$ would have *no* observable effect other than an increase in execution time. Instead the result of $\mathcal{T}$ is an intermediate step in the production of a *simplified* program, which abstracts that part of the program concerned with robustness in the form of a robustness slice. The robustness slice, is not necessarily executed (although it may be), but is analysed by the programmer, who has the benefit of considering the robustness characteristics of the original program in isolation.

In this paper the only form of robustness considered is the integrity of array indices. In (Flater et al., 1993) Falter and his colleagues consider a wider range of robustness issues, including variable reference before definition, loop bounds checking and null pointer de–reference checking. Bearing in mind the close similarity between the automatic introduction of assertions to check these execution characteristics, and the introduction of assignments to pseudo–variables described in this paper, it would be interesting to consider the possibility of constructing other forms of robustness slice, each of which could be used to explicitly capture one of these forms of robustness behaviour in a simplified program.

11 Conclusion

The robustness slice approach combines slicing, meaning–preserving transformation and pseudo–variable introduction, three simple ideas which allow us to form very simple 'robustness slices', which capture the robustness of the subject program from which they are constructed.

Slicing simplifies the task of testing a program's behaviour with respect to a particular subset of its variables. Slicing cannot, as it stands, however, be applied to the analysis of 'implicit' aspects of a program's behaviour (those not denoted by a set of program variables). In such cases it is possible to introduce pseudo–variables which capture these implicit aspects of program behaviour, forming a self–checking program reminiscent of the kind formed by the automatic introduction of assertions to test robustness at various program points.

Applying a combination of conventional slicing algorithms and meaning preserving transformations to the self–checking program, a new program is produced, the 'robustness slice', which captures a chosen implicit computation explicitly, and in isolation.

This approach cannot always produce the simplest program (a single assignment statement) since the robustness question is undecidable. Therefore, the slicing and simplification transformations must be refined as far as possible in order to achieve the best possible approximation to the simplest program.

Acknowledgements

We would like to thank Information Porcessing Limitted for Supplying us with their Cantata Tool for Software Testing and Measurement and Dan Simpson and his colleagues at Brighton University for their support and collaboration with this project. We would also like to express our gratitude to Yoga Sivagurunathan and the anonymous referees for their helpful comments and revisions which greatly improved the presentation of this paper.

References

Agrawal, H. (1994). On slicing programs with jump statements. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 302–312, Orlando, Florida. Available as SIGPlan Notices, 29(6), June 1994.

Agrawal, H., DeMillo, R. A., and Spafford, E. H. (1991). Dynamic slicing in the presence of unconstrained pointers. In *ACM 4th. Symposium on Testing, Analysis, and Verification (TAV4)*, pages 60–73. Appears as Purdue University Technical Report SERC-TR-93-P.

Agrawal, H. and Horgan, J. R. (1990). Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, New York.

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, techniques and tools.* Addison Wesley.

Ball, T. and Horwitz, S. (1992). Slicing programs with arbitrary control–flow. In Fritzson, P., editor, 1^{st} *Conference on Automated Algorithmic Debugging*, pages 206–222. Springer. Also available as Uniersity of Wisconsin–Madison, technical report (in extended form), TR-1128, december, 1992.

Beck, J. and Eichmann, D. (1993). Program and interface slicing for reverse engineering. In *IEEE/ACM* 15^{th} *Conference on Software Engineering (ICSE'93)*, pages 509–518.

Bieman, J. M. and Ott, L. M. (1994). Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657.

Cheng, J. (1993). Slicing concurrent programs – a graph–theoretical approach. In Fritzson, P., editor, 1^{st} *Autmoatic Algorithmic Debugging Conference (AADEGUB'93)*, pages 223–240. Appears as Springer Lecture Notes in Computer Science vol 749.

Choi, J. and Ferrante, J. (1994). Static slicing in the presence of goto statements. *IEEE Transactions on Software Engineering*, 16(4):1097–1113.

Davenport, J. H., Siret, Y., and Tournier, E. (1989). *Computer algebra: Systems and algorithms for algebraic computation.* Academic Press.

Ernst, M. D. (1994). Practical fine–grained static slicing of optimised code. Technical Report MSR-TR-94-14, Microsoft research, Redmond, WA.

Flater, D. W., Yesha, Y., and Park, E. K. (1993). Entensions to the C programming language for enhanced fault detection. *Software – Practice and Experience*, 23(6):617–628.

Gallagher, K. B. and Lyle, J. R. (1991). Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761.

Gopal, R. (1991). Dynamic program slicing based on dependence graphs. In *IEEE Conference on Software Maintenance*, pages 191–200.

Harman, M. and Danicic, S. (1994a). A framework for defining equivalence preserving program simplification and its application to program slicing. Technical report, University of North London, Project Project. Available by anonymous ftp to `ftp.unl.ac.uk/pub/text/M.Harman/papers/tecrepts`.

Harman, M. and Danicic, S. (1994b). A new approach to program slicing. In 7^{th} *International Quality Week*, pages Paper 4–T–4, pp 1–14, San Francisco.

Harman, M., Danicic, S., Sivagurunathan, B., Jones, B., and Sivagurunathan, Y. (1995a). Cohesion metrics. In 8^{th} *International Quality Week*, pages Paper 3–T–2, pp 1–14, San Francisco.

Harman, M., Danicic, S., and Sivagurunathan, Y. (1995b). A parallel algorithm for static program slicing. Technical report, University of North London, Project Project. Available by anonymous ftp to `ftp.unl.ac.uk/pub/text/M.Harman/papers/tecrepts`.

Horwitz, S., Prins, J., and Reps, T. (1989). Integrating non–interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387.

Horwitz, S. and Reps, T. (1992). The use of program dependence graphs in software engineering. In *14th. International Conference on Software Engineering*, pages 392–411, Melbourne, Australia.

Horwitz, S., Reps, T., and Binkley, D. (1988). Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–46, Atlanta, Georgia. Proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

Jiang, J., Zhou, X., and Robson, D. J. (1991). Program slicing for C – The problems in implementation. In *IEEE Conference on Software Maintenance*, pages 182–190, Italy.

Kamkar, M. (1993). *Interprocedural dynamic slicing with applications to debugging and testing.* PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden. Available as Linköping Studies in Science and Technology, Dissertations, Number 297.

Korel, B. and Laski, J. (1988). Dynamic program slicing. *Information Processing Letters*, 29(3):155–163.

Lakhotia, A. (1993). Rule–based approach to computing module cohesion. In *Proceedings of the 15th. Conference on Software Engineering (ICSE-15)*, pages 34–44.

Livadas, P. E. and Rosenstein, A. (1994). Slicing in the presence of pointer variables. Technical Report SERC-TR-74-F, Computer Science and Information Services Department, University of Florida, Gainesville, FL.

Lui, L. and Ellis, R. (1993). An approach to eliminating COMMON blocks and deriving ADTs from Fortran programs. Technical report, University of Westminster, UK.

Lyle, J. R. and Weiser, M. (1987). Automatic program bug location by program slicing. In 2^{nd} *International Conference on Computers and Applications*, pages 877–882, Peking.

Mariam Kamkar, N. S. and Fritzson, P. (1992). Interprocedural dynamic slicing. In *Proceedings of the 4th. Conference on Programming Language Implementation and Logic Programming*, pages 380–384.

Ott, L. M. and Thuss, J. J. (1993). Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium*, pages 78–81.

Ottenstein, K. and Ottenstein, L. (1984). The program dependence graph in software development environments. *SIGPLAN Notices*, 19(5):177–184.

Qi, L., Fubo, Z., and Jiahua, Q. (1988). Program slicing. *Journal of Computer Science and Technology*, 3(1):29–39.

Rosenblum, D. S. (1995). A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31.

Shahmehri, N. (1991). *Generalized algorithmic debugging.* PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden. Available as Linköping Studies in Science and Technology, Dissertations, Number 260.

Simpson, D., Valentine, S. H., Mitchel, R., Lui, L., and Ellis, R. (1993). Recoup – Maintaining Fortran. *ACM SIGPlan Fortran forum*, 12(3):26–32.

Stoy, J. E. (1985). *Denotational semantics: The Scott–Strachey approach to programming language theory.* MIT Press. Third edition.

Tip, F. (1995a). *Generation of Program Analysis Tools.* PhD thesis, Centrum voor Wiskunde en Informatica, Amsterdam.

Tip, F. (1995b). A survey of program slicing techniques. *Journal of Programming Languages.* To appear.

Venkatesh, G. A. (1991). The semantic approach to program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–28, Toronto, Canada. Proceedings in *SIGPLAN Notices*, 26(6), pp.107–119, 1991.

Weiser, M. (1979). *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI.

Weiser, M. (1981). Program slicing. In *5th International Conference on Software Engineering*, pages 439–449, San Diego, CA.

Weiser, M. (1983). Reconstructing sequential behaviour from parallel behaviour projections. *Information Processing Letters*, 17(10):129–135.

Weiser, M. (1984). Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357.

1 `i=1;`	1 `i=1;`
2 `s=0 ;`	
3 `while(i<MAX)`	3 `while(i<MAX)`
4 `{ s=s+a[i];`	4 `{`
5 `i=i+1;`	5 `i=i+1;`
6 `}`	6 `}`
The Original Program	The Slice on $(\{i\}, 6)$

Figure 1: A Simple Loop Fragment and one of its Slices

```
<prog>       ::= <stat>
<stat>       ::= <out> | <in> | <exp>; | <while> | <braced> | <if>
<out>        ::= printf(<string>,<exp-list>) ; | printf(<string>) ;
<in>         ::= scanf(<string>,<var-list>) ;
<while>      ::= while(<exp>) <stat>
<braced>     ::= { <stat-list> }
<if>         ::= if(<exp>) <stat>
<exp>        ::= <post> | <assign> | <id> | <num> | <char> |
                 <exp> <bin-op> <exp> | <array-lu>
<bin-op>     ::= < | > | <= | >= |  ==   |  !=   |  && | + | - | *
<exp-list>   ::= <exp> | <exp>, <exp-list>
<var-list>   ::= <id> | <id> , <var-list>
<stat-list>  ::=      | <stat> <statlist>
<assign>     ::= <id> = <exp> | <id>[ <exp> ] = <exp>
<array-lu>   ::= <id>[ <exp> ]
<post>       ::= <id>  ++
```

Figure 2: `SmallC`, A Subset of C

$$
\begin{array}{rcl}
i & : & \text{identifiers} \\
n & : & \text{numeric constants} \\
b & : & \text{binary operators} \\
p & : & \text{programs} \\
c & : & \text{commands} \\
h & : & \text{characters} \\
s & : & \text{strings} \\
\lambda & : & \text{the empty statement} \\
\\
\mathcal{T}[\![p]\!] & = & [\![\{\texttt{robust = 1;}\ \mathcal{C}[\![p]\!]\}]\!] \\
\\
\mathcal{C}[\![\texttt{printf}(s, e_1, \ldots, e_n);]\!] & = & \mathcal{E}[\![e_1]\!] \ldots \mathcal{E}[\![e_n]\!][\![\texttt{printf}(s, e_1, \ldots, e_n);]\!] \\
\mathcal{C}[\![\texttt{scanf}(s, i_1, \ldots, i_n);]\!] & = & [\![\texttt{scanf}(s, i_1, \ldots, i_n);]\!] \\
\mathcal{C}[\![\texttt{while}(e)c]\!] & = & \mathcal{E}[\![e]\!][\![\texttt{while}(e)\{\mathcal{C}[\![c]\!]\ \mathcal{E}[\![e]\!]\}]\!] \\
\mathcal{C}[\![\{c_1 \ldots c_n\}]\!] & = & [\![\{\mathcal{C}[\![c_1]\!] \ldots \mathcal{C}[\![c_n]\!]\}]\!] \\
\mathcal{C}[\![\texttt{if}(e)c]\!] & = & \mathcal{E}[\![e]\!][\![\texttt{if}(e)\ \mathcal{C}[\![c]\!]]\!] \\
\mathcal{C}[\![i{=}e;]\!] & = & \mathcal{E}[\![e]\!][\![i{=}e;]\!] \\
\mathcal{C}[\![i[e_1] = e_2;]\!] & = & \mathcal{R}([\![e_1]\!], i)\ \mathcal{E}[\![e_2]\!][\![i[e_1] = e_2;]\!] \\
\\
\mathcal{E}[\![i]\!] & = & \lambda \\
\mathcal{E}[\![n]\!] & = & \lambda \\
\mathcal{E}[\![h]\!] & = & \lambda \\
\mathcal{E}[\![e_1\ b\ e_2]\!] & = & \mathcal{E}[\![e_1]\!]\ \mathcal{E}[\![e_2]\!] \\
\mathcal{E}[\![i[e]]\!] & = & \mathcal{R}([\![e]\!], i) \\
\\
\mathcal{R}(e, i) & = & [\![\texttt{robust} = \texttt{robust}\ \ \&\&\ \ e{>}{=}0\ \ \&\&\ \ e{<}{=}MAX(i);]\!]
\end{array}
$$

Figure 3: The Introspection Transformation, $\mathcal{T}$

Step 1	Remove side effects
Step 2	Apply $\mathcal{T}$, forming a self–checking program
Step 3	Slice the program with respect to `robust` at its last line
Step 4	Use Rules 2, 3, 4 and 1 to Coalesce assignments to `robust`
Step 5	Use Axioms 1 and 2 to simplify expressions involving `robust`
Step 6	Repeat steps 3–5 until slicing fails to delete any statements

Figure 4: An Algorithm for Creating a Robustness Slice

Axiom 1 (Commutativity of $\&\&$**)**

$$[\![e_1 \;\&\&\; e_2]\!] \;\;\Rightarrow\;\; [\![e_2 \;\&\&\; e_1]\!]$$

Axiom 2 (Idempotence of $\&\&$**)**

$$[\![e \;\&\&\; e]\!] \;\;\Rightarrow\;\; [\![e]\!]$$

Rule 1 (Conditional Repetition)

$$\frac{\texttt{robust} \notin \mathbf{DEF}(s)}{[\![\texttt{robust} = e_1; \texttt{if}(e_2)\ \{\texttt{robust} = e_1; s;\}]\!] \Rightarrow [\![\texttt{robust} = e_1;]\!]}$$

Rule 2 (Push Assignment)

$$\frac{i_1 \neq i_2 \wedge i_2 \notin \mathbf{REF}(e_1) \wedge e_3 = \mathbf{SUB}(e_2, i_1, e_1)}{[\![i_1 = e_1; i_2 = e_2;]\!] \Rightarrow [\![i_2 = e_3; i_1 = e_1;]\!]}$$

Rule 3 (Unfold Assignment)

$$\frac{i \in \mathbf{REF}(e_2) \wedge e_3 = \mathbf{SUB}(e_2, i, e_1)}{[\![i = e_1; i = e_2;]\!] \Rightarrow [\![i = e_3;]\!]}$$

Rule 4 (Remove Assignment)

$$\frac{i \notin \mathbf{REF}(e_2)}{[\![i = e_1; i = e_2;]\!] \Rightarrow [\![i = e_2;]\!]}$$

Figure 5: Transformation Rules to Simplify Slices

Program	Flowgraph Nodes	Description
Example: The Original Porgram	28	The Original Program
Example Step 1	32	The Side–Effect Free Version
Example Step 2	44	The Robustness Self–Testing Program
Example Step 3	33	The Conventional Slice
Example Step 4	19	The Non–Conventional Slice
Example: Final Step	1	The Best Slice

Figure 6: The Size of the Case Study Example at Each Stage in Transformation