Program Comprehension Assisted by Slicing and Transformation

Mark Harman, Sebastian Danicic and Yoga Sivagurunathan
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, Program Simplification

Abstract

Program slicing is a technique for program simplification based upon the deletion of statements which cannot affect the values of a chosen set of variables. Because slicing extracts a subcomponent of the program concerned with some specific computation on a set of variables, it can be used to assist program comprehension, allowing a programmer to remodularise a program according to arbitrarily selected slicing criteria.

In this paper it is shown that the simplification power of slicing can be improved if the syntactic restriction to statement deletion is removed, allowing slices to be constructed using *any* simplifying transformation which preserves the effect of the original program upon the set of variables of interest. It is also shown that quasi static slicing, first proposed by Venkatesh (and defined here in a slightly more general form), is the most suitable slicing paradigm for program comprehension.

The various forms of slice are formally defined, an algorithm, based upon transformation, symbolic execution and conventional slicing is introduced for computing syntactically unrestricted, quasi static slices. A worked example is used to show how this approach supports program comprehension by case analysis and simplification.

1 Introduction

One familiar approach to program comprehension is that of 'divide and conquer'. A large program often consists of many smaller sub–programs, and in many cases, in particular with legacy code, these sub–programs are not contained in separate modules. Even in cases where an appropriate module boundaries have been drawn across a large system, it is often helpful to trace particular computations which may cross module boundaries.

Weiser [52] has shown in empirical studies of programmer behaviour, that programmers mentally construct 'slices' when understanding and debugging programs. In this paper a slice–based approach to program comprehension is advocated, however, it is demonstrated, that the traditional definition of a slice requires modification to make the technique suitable for program comprehension.

The original definition given by Weiser, and its subsequent modifications by Korel and Laski [32] (for the dynamic paradigm) and by Venkatesh [50] and Tip [48] (for the quasi static paradigm) share the property that a slice is to be constructed using a single transformation method, that of statement deletion. This restriction to command deletion is removed in the definition of slice presented here, making slicing more useful as a starting point for program comprehension.

The rest of the paper is organised as follows:-

Section 2 briefly presents conventional program slicing and in section 3 a projection semantics is defined that forms the basis for subsequent formal definitions of slicing. In section 4 it is argued that whilst the restriction to statement deletion is essential for debugging, a more general definition of slice (which does not require statement deletion) is required for program comprehension problems.

`1   i=1;`	`1   i=1;`
`2   p=1;`	
`3   s=0 ;`	`3   s=0;`
`4   while(i<MAX)`	`4   while(i<MAX)`
`5     { s=s+i;`	`5     { s=s+1;`
`6       p=p*i;`	
`7       i=i+1;`	`7       i=i+1;`
`8     }`	`8     }`
`9     output(s,p);`	
The Original Program	The Slice Constructed for the Final Value of `s`

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

In section 5 the quasi static paradigm is formally defined (in a form slightly different to that presented by Venkatesh [50] and Tip [48]), and shown to subsume both the static and dynamic paradigm. In section 6 an algorithm is sketched for constructing syntactically unrestricted, quasi static slices, and a worked example is presented, demonstrating how this approach can be used to support program comprehension by case analysis and simplification. Finally sections 7 and 8 describe related and future work.

2 Conventional Program Slicing

Conventionally, a slice is constructed with respect to a slicing criterion, (V, n). Commands which have no effect on variables in the set V at the point n within the original program are removed to form the slice. Consider the simple loop program given in the left–hand box of figure 1, for which a slice has been constructed with respect to the slicing criterion $(\{\mathtt{s}\}, 9)$, to capture the effect of the program upon the variable `s`, at the end of the program. 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. Slicing has many applications including measuring cohesion [8, 40, 33, 24], algorithmic debugging [44, 29], re–engineering [35, 45], component re–use [6], automatic parallelisation [53], maintenance and debugging [18, 36], program integration [26], testing [23, 29] and assisted verification [41].

Much of the literature on slicing is concerned with the technical problems associated with slicing algorithms [27, 15, 25, 1, 5, 2, 10]. Tip [49] presents a detailed survey of the paradigms and techniques for program slicing.

A slice is also an aid to program comprehension, because it is a simplified version of the original program, concerned solely with a projection of the semantics of the original. For example, understanding the effect of the original program in figure 1 upon the variable `s`, is clearly easier using the slice constructed for `s`, because statements irrelevant to the computation of `s` have been removed.

In this paper slices will always be constructed for a slicing criterion where the point of interest is the last line of the program to be executed. This form of slicing is called 'end slicing' by Lakhotia [33]. However, the approach advocated here is applicable to program fragments as well as to entire programs – it is envisaged that slices will be frequently constructed for sections of code which are particularly hard to understand.

3 A Formal Definition of End Slicing

The definition of an end slice presented here is based upon the assumption that the meaning of a program can be adequately described by a state to state mapping [47, 43].

In the following definitions, $\mathcal{C}$ is the semantic function which describes the meaning of a program. That is, $\mathcal{C}[\![p]\!]$ is the state to state mapping denoted by the program p and $\mathcal{C}[\![p]\!]\sigma$ is thus the final state produced when the program p is executed in the initial state σ. In this exposition a state will be considered to be a mapping from variable names to the values they denote in the state, so $\mathcal{C}[\![p]\!]\sigma i$ is the final value of the variable i when the program p is executed in the state σ.

First, the projection of a state to state mapping, μ, onto a set of variables, K, is defined as follows:

Definition 3.1 (Projection)
Let μ be a state to state mapping and K be a set of variables, then for all states, q and for all identifiers, i,
$(\mu \rhd K)\sigma i = \begin{cases} \mu\sigma i & \text{if } i \in K \\ \sigma i & \text{otherwise.} \end{cases}$

Consider example 3.1 below[1]:

Example 3.1

```
x=p;
y=x+1;
```

Example 3.1 denotes the state to state mapping $\lambda\sigma.\sigma \oplus \{\mathtt{x} \mapsto \sigma(\mathtt{p}), \mathtt{y} \mapsto \sigma(\mathtt{p}) + 1\}$ where $\oplus$ is the function overriding operator [46].

Using definition 3.1 and $\mathcal{C}$, it is possible to define the equivalence which exists between an end slice (with respect to the set of variables K), and the program from which it is constructed (this equivalence shall be termed 'K equivalence').

Definition 3.2 (K equivalence)
A program p' is K equivalent to p if and only if $\mathcal{C}[\![p]\!] \rhd K = \mathcal{C}[\![p']\!] \rhd K$.

Two programs are thus K equivalent if, ultimately, they have an identical effect upon all variables in K. A conventional end slice with respect to K is simply a K equivalent program constructed by statement deletion:

Definition 3.3 (End Slice)
Given two programs p and p' and a set of variables, K, p' is an end slice of p with respect to K iff p is K equivalent to p' and p' is obtained from p by deleting zero or more statements from p.

For example, the fragment `x=p;` is $\{\mathtt{x}\}$ equivalent to example 3.1 and as it is obtained by deleting one statement from example 3.1 it is an end slice (with respect to $\{\mathtt{x}\}$) of example 3.1. By contrast, the only end slice (with respect to $\{\mathtt{y}\}$) of example 3.1 is example 3.1 itself, since all other $\{\mathtt{y}\}$ equivalent programs *cannot* be obtained by the process of statement deletion. This is a shame (from the point of view of program comprehension) as the fragment `y=p+1;` *is* $\{\mathtt{y}\}$ equivalent to example 3.1, and, as it is the simplest such program fragment, it clearly assists analysis of the effect of example 3.1 upon the variable `y`. This observation motivates the consideration of 'syntactically unrestricted program slicing', which is introduced in the next section.

4 Syntactically Unrestricted Program Slicing

Notice that definition 3.3 of an end slice *could* have been written

> "An end slice with respect to K is constructed by the application of any K equivalence preserving statement deletion"

When written in this form, it becomes natural to ask why end slices may *only* be produced by one form of K equivalence preserving transformation (that of statement deletion). It is easy to demonstrate that the restriction of conventional slicing to statement deletion inhibits simplification.

Consider example 4.1 below:

Example 4.1

```
x=p;
y=x*x;
z=q+y-2;
```

[1] All programs in this paper shall be written in the `C` programming language.

In the analysis of this fragment's effect upon the variable `z`, a sensible starting point is clearly the end slice constructed for the variable `z`. Sadly, it is not possible to delete *any* statements in the construction of this end slice, because the final use of the variable `z` has data dependence upon every statement in the program.

However, example 4.2 (below) *is* {`z`} equivalent to example 4.1. Although not a slice in the conventional syntactic sense (not being obtained purely by statement deletion), it is, however, a slice in the important *semantic* sense that both it and the original program (example 4.1) are equivalent with respect to their effect upon the variable `z`. For program comprehension this equivalence is the important aspect of slicing, as it allows analysis of properties of the original program to be conducted using the slice. The slice should be as simple as possible to maximise the advantages of analysing it in preference to the original, and so any equivalence preserving transformation which simplifies the program should be admissible.

Example 4.2

```
z=q+p*p-2;
```

Abandoning the statement deletion restriction gives rise to a new weaker[2] definition of a slice, which allows fragments such as example 4.2 to be considered to be slices of example 4.1. The term 'syntactically unrestricted slice' shall be used to describe such slices:

Definition 4.1 (Syntactically Unrestricted End Slice)
Given two programs p and p' and a set of variables, K, p' is a syntactically unrestricted end slice of p with respect to K iff p is K equivalent to p' and if p' is simpler than p.

Notice that the definition of a syntactically unrestricted slice (definition 4.1) is obtained from the definition of a conventional slice (definition 3.3) by simply removing the restriction to statement deletion and substituting for it, the phrase 'is simpler'. In this paper 'is simpler' can be read more rigorously as 'contains fewer nodes in its control flow graph', however, a more general theory of program simplification can be obtained if this 'simplicity measure' is treated as a parameter to the new definition of a slice [22], a possibility which is not pursued further in this paper.

5 The Quasi Static Paradigm

Program slicing was originally introduced as a static analysis technique [51]. Subsequently, a dynamic paradigm was introduced [32], partly to allow for dynamic analysis and partly because results from static slicing produced little simplification on real programs due to the typically high level of statement inter–dependencies exhibited by such programs[3].

A static slice is constructed at compile time, using no information about the input to a program. A dynamic slice is constructed at run time with respect to a particular input. A static slice is often considerably larger than a dynamic slice, but a dynamic slice is specific to one program execution. However, for complete program comprehension, understanding one execution of the program will not be enough, yet static slicing may sometimes produce little simplification, even in the syntactically unrestricted form.

Quasi static slicing was first introduced by Venkatesh [50]. His suggestion was that a slice should be constructed with respect to an initial prefix of the input sequence to a program (in this respect Venkatesh's approach is reminiscent of partial evaluation (see section 7)). Tip [48] also uses a quasi static formulation of program slicing, in which a slice is constructed with respect to a set of inequalities which capture partial information about the initial state in which a program is to be executed.

The quasi static paradigm provides a convenient 'compromise' between the poles of static and dynamic slicing and is most appropriate to the application of program comprehension as it allows for *analysis by cases* in addition to simplification (see section 6.2 for an example). As will be seen, the quasi static definition subsumes both the static and dynamic definitions of slice, so nothing is lost by adopting it as the central paradigm for program comprehension assisted by slicing.

Quasi static slicing can be defined in terms of the quasi static equivalence relation which must be preserved by the slicing process. In a syntactically restricted form, this quasi static equivalence gives rise to the definition of

[2] in the sense that a conventional slice is always a syntactically unrestricted slice, but the converse is not true.
[3] This observation was the motivation for the exploration of program slices as a basis for a cohesion measure [39, 40, 33, 8, 24].

quasi static slicing (definition 5.2), and in the unrestricted form it produces the related definition of syntactically unrestricted quasi static slicing (definition 5.3).

Static equivalence (definition 3.2 of K equivalence) is defined with respect to a set of variables, K. For quasi static equivalence the equivalence relation is parameterised, not just by a set of variables, but also by a set of states, Σ. Σ represents the set of states for which two equivalent programs will behave identically with respect to the set of variables K.

Definition 5.1 ((Σ, K) equivalence)
Let Σ be a set of states and K be a set of variables. A program p' is (Σ, K) equivalent to p iff $\forall \sigma \in \Sigma . (\mathcal{C}[\![p]\!] \rhd K)\sigma = (\mathcal{C}[\![p']\!] \rhd K)\sigma$.

Notice that a program p is (Σ, K) equivalent to a program p' if and only if it has an identical effect to p' upon the variables in K, when executed in initial states drawn from Σ. However, p may have any effect upon variables not in K and, in addition, may have any effect upon variables *in* K when executed in a state outside Σ.

Using this notion of quasi static equivalence, it is possible to define restricted and unrestricted quasi static slices as follows:

Definition 5.2 (Syntactically Restricted Quasi Static End Slice)
Given two programs p and p', a set of variables, K, and a set of states Σ, p' is a quasi static end slice of p with respect to K iff p is (Σ, K) equivalent to p' and p' can be obtained by the deletion of zero or more statements from p.

Definition 5.3 (Syntactically Unrestricted Quasi Static End Slice)
Given two programs p and p', a set of variables, K, and a set of states Σ, p' is a syntactically unrestricted quasi static end slice of p with respect to K iff p is (Σ, K) equivalent to p' and p' is simpler than p.

5.1 The Relationship Between Quasi Static, Static and Dynamic Slicing

All forms of slice introduced in this paper and in the literature are constructed with respect to a set of initial states in which a program and its slice are to be executed.

For a dynamic slice[4] all initial states have a completely defined input sequence. As dynamic slicing is traditionally applied to entire programs and not program fragments, a dynamic slice will be defined with respect to a single initial state[5].

A quasi static slice, according to Venkatesh [50], is constructed with respect to an initial prefix, π, of the input sequence, and thus is constructed with respect to the set of states in which the input is prefixed by π.

A quasi static slice, according to Tip [48], is constructed with respect to a set of inequalities. These inequalities (if they have a solution) have solutions which define sets of initial states.

A static slice is defined in terms of a set of variables alone, and so is effectively constructed with respect to $\mathcal{S}$, the set of all states.

Since all approaches to slicing can be defined in terms of a set of initial states, all can be captured by the quasi static slicing paradigm introduced here (in fact 'quasi dynamic' is just as good a term as 'quasi static'). Using quasi static slicing is thus merely a generalisation which allows the 'user' to focus upon arbitrary sets of states capturing particular initial execution conditions. This is particularly important for program comprehension, where a programmer will often want to consider the behaviour of a program in various specialised initial conditions – a form of case analysis.

[4] Some technical details relating to Korel and Laski's definition of a dynamic slicing criterion[32] have been omitted here. A more detailed treatment is presented in [22].

[5] Some authors [3, 20, 30] regard a dynamic slice as merely that set of statements which affect the slicing criterion, and not necessarily an *executable* program which preserves the effect of the original upon the slicing criterion. This form of slicing, known as 'closure slicing' [50], cannot be captured by definitions similar to those presented here, as the definitions are based upon the assumption that a slice is an executable program.

step 1	Preprocess
step 2	Symbolically execute
step 3	Slice
step 4	Postprocess

Figure 2: The outline of an algorithm for constructing syntactically unrestricted quasi static slices

6 Towards Automated Construction of Syntactically Unrestricted Slices

In order to construct a syntactically unrestricted slice *any* program transformation may be used provided it yields a simpler program preserves the projected meaning of the original. This allows considerable flexibility. For example, conventional slicing, transformation [21], computer algebra [13], partial evaluation and mixed computation [9, 37, 38, 17] and symbolic execution [11, 12] are all candidates together with a variety of well known compiler optimisation techniques [4], such as loop unfolding, code motion, and constant propagation.

The authors are currently working on the development of an algorithm for constructing syntactically unrestricted quasi static slices, based upon the sketch of the algorithm structure given in the next section.

6.1 The Algorithm

The outline of the algorithm is presented in figure 2.

The pre–processing phase removes side effects, producing an equivalent side–effect free form. This phase also modifies the program to establish the initial state as one of the states from the set of possible initial states contained in the quasi static slicing criterion. For example, sets of states captured by a boolean expression which involve only equality operators may be established using assignment statements.

The symbolic execution phase executes and transforms the program with respect to a symbolic state, a mapping from identifiers to the expressions that they denote. Right hand sides of assignments are replaced by expressions whose referenced variables have been substituted from the expressions recorded in the symbolic state (called 'copy propagation' [12]). On encountering a boolean expression, the symbolic executor attempts to reduce the boolean to a constant, thereby simplifying the structure of the program by removing unnecessary branches. Notice that such 'unnecessary branches' will occur more often in this process than in the symbolic execution of programs in general, due to the fact that a quasi static slice need only be constructed with respect to a set of possible initial states. The smaller this set of initial states, the more 'degenerate branches' the program will , in general, contain.

Expressions are also simplified, where possible, using a set of algebraic identities.

By substituting referenced variables with the expressions they denote in the symbolic state, the symbolic executor prepares the way for the slicing phase of the algorithm, which will be able to delete more statements than it would otherwise, because the level of statement interdependence will have been reduced.

Finally, the program is post–processed to present its syntax in a more readable form. During this phase of the algorithm, the conventional meaning of the program will remain unchanged, thus any transformation used will be completely general and will not make use of information contained in the quasi static slicing criterion.

It is possible that steps two and three will need to be iterated until no further statements are removed during the slicing phase. Consider the program fragment below:

```
1    x = y*2;
2    y = z+4;
3    z = x-1;
```

Suppose a slice is to be constructed with respect to the slicing criterion $(\{\sigma \mid \texttt{true}\}, \{\texttt{z}\})$. The symbolic execution phase will leave the program untransformed. The subsequent slicing phase will remove line 2. If the algorithm stops there the result will be the fragment:

```
1    x = y*2;
3    z = x-1;
```

which would have been obtained using conventional slicing. However, if the process is repeated then the algorithm will produce:

```
3   z = y*2-1;
```

6.2 A Worked Example

The approach adopted by the algorithm sketched in the previous section can be demonstrated with the `atoi` program below (taken from page 61 of [31], but modified so that the two function calls to `isspace` and `isdigit` have been inlined, and the whole function, `atoi`, turned into a program fragment):

```
for(i=0;s[i]==' ';i++);
sign=(s[i]=='-')?-1:1;
if(s[i]=='+' || s[i]=='-') i++;
for(n=0;s[i]>='0' && s[i] <= '9';i++)
  n = 10*n + (s[i]-'0');
r=sign*n;
```

This program fragment converts the ASCII string `s` into an integer, storing the result in `r`. It is typical of the 'compact' and often unclear style of `C` programming and is thus an ideal example with which to examine program comprehension problems.

Suppose the algorithm is to be applied to produce a syntactically unrestricted quasi static slice with respect to quasi static slicing criterion $(\{\sigma \mid \sigma(\mathtt{s[0]}) = \text{'}-\text{'}\}, \mathtt{r})$. Such a slice will capture the effect of the original fragment upon the variable `r`, when executed in a state which records a `'-'` character as the first element of the array `s`.

First the fragment is transformed into a side–effect free[6] form, and an initial assignment, `s[0]='-';` is added to capture the information concerning the initial state. To this version of the program, line numbers have been added, so that individual statements can be easily referred to in what follows. The process of side effect removal, line numbering and adding the initial assignment produces the fragment below:

```
1    s[0]='-';
2    i=0;
3    while(s[i]==' ') i=i+1;
4    sign=(s[i]=='-')?-1:1;
5    if(s[i]=='+' || s[i]=='-') i=i+1;
6    n=0;
7    while(s[i]>='0' && s[i]<='9') {n=10*n+(s[i]-'0'); i=i+1;}
8    r=sign*n;
```

The symbolic state on reaching line three is

$$\{(\mathtt{s[0]}, \text{'}-\text{'}), (\mathtt{i}, 0)\}$$

Substituting known values from the state for the boolean expression at line three gives the expression `s[0]==' '`, which is in both disjunctive normal form (DNF) and conjunctive normal form (CNF). Searching the state for `s[0]` reveals that this boolean expression has the constant value, `false`, and so line three is removed.

Upon reaching line four, the expression on the right hand side of the assignment statement is replaced with known values from the symbolic state to give `s[0]=='-'?-1:1`. Searching the state for `s[0]` reveals that this expression has the constant value `true` so the expression is rewritten `true?-1:1`. Applying algebraic identities to this expression causes it to be simplified to `-1`. The assignment at line four is thus replaced by the assignment `sign = -1;`.

[6] The algorithm used to perform these transformations has not been formally published, but is based upon a few simple transformation rules, which keep track of side effects and 'massage' them out of expressions. As a by–product the side–effect removal algorithm also converts `for` loops into equivalent `while` loops. The algorithm is described in a B.Sc. project report [14], and has been implemented as part of the authors' simple slice tool, available by anonymous ftp (see references). Side effect removal such as this can also be achieved using a similar approach advocated by Lightfoot [34].

Upon reaching line five, the boolean expression is substituted with known values from the symbolic state to give `s[0]=='+' || s[0]=='-'`, which is in DNF. Searching the state for `s[0]` reveals that the right hand term of this boolean expression is `true`, and so the expression is replaced with the expression `s[0]=='+' || true`. Applying the algebraic identities to this expression simplifies it to `true`. Thus the conditional at line five can be replaced by its consequent, `i=i+1`, and symbolic execution continues at the modified line five, with the symbolic state

$$\{(\texttt{s[0]}, '-'), (\texttt{i}, 0), (\texttt{sign}, -1)\}$$

The right hand side of the assignment at (the new version of) line five is replaced by known values from the symbolic state to give the expression `0+1`, which is evaluated to `1`, and the assignment at line five is thus replaced by `i=1;`.

Symbolic execution now reaches line six with the symbolic state

$$\{(\texttt{s[0]}, '-'), (\texttt{i}, 1), (\texttt{sign}, -1)\}$$

As there is no substitution to be performed, the only effect of symbolic execution of line six is to add a new maplet to the symbolic state, recording the value for `n`.

Symbolic execution now reaches line seven with the symbolic state

$$\{(\texttt{s[0]}, '-'), (\texttt{i}, 1), (\texttt{sign}, -1), (\texttt{n}, 0)\}$$

Substituting known values into the boolean expression at line seven has no effect and so the body of the loop is symbolically executed in the state

$$\{(\texttt{s[0]}, '-'), (\texttt{sign}, -1)\}$$

Notice that the maplets $\{\texttt{i} \mapsto 1\}$ and $\{\texttt{n} \mapsto 0\}$ are removed from the symbolic state in the execution of the body of the loop as the variables `i` and `n` are members of the defined variable set for the loop body, and thus their symbolic values are not 'known' during symbolic execution of the loop body.

Symbolic execution of the loop body has no effect and symbolic execution continues at line eight with the symbolic state

$$\{(\texttt{s[0]}, '-'), (\texttt{sign}, -1)\}$$

Substituting known values from the state into the right hand side of the assignment at line eight gives the expression `-1*n`, which is simplified using the algebraic identities to `-n`. The assignment at line eight is thus replaced by the assignment `r=-n;`.

Symbolic execution is now complete and has produced the transformed program below:

```
1    s[0]='-';
2    i=0;
3    /* removed */
4    sign=-1;
5    i=1;
6    n=0;
7    while(s[i]>='0' && s[i]<='9') {n=10*n+(s[i]-'0'); i=i+1;}
8    r=-n;
```

Conventional end slicing for this fragment with respect to $\{\texttt{r}\}$ causes simplification because the process of symbolic execution has reduced the interdependencies between statements. The end sliced program fragment is:

```
1    s[0]='-';
5    i=1;
6    n=0;
7    while(s[i]>='0' && s[i]<='9') {n=10*n+(s[i]-'0'); i=i+1;}
8    r=-n;
```

Having constructed the slice, the initial assignment (line one), which was added to establish the initial state (as dictated by the quasi static slicing criterion) is removed.

There is one extra 'post–processing' transformation might be considered, depending upon the programmer's taste for syntactic style. A `while` loop, and its preceding assignments may be transformed to a `for` loop according to the following transformation rules:

Rule 1 The `while` loop $while(E)S$ can be transformed into the `for` loop $for(;E;)S$.

Rule 2 Consider the `for` loop $for(;E;)S$. An assignment statement $I = E'$ in the body of S may be placed in the 'increment' part of the `for` loop iff I is referenced in both E and E' and no statement following $I = E'$ either defines or references I.

Rule 3 An assignment which immediately precedes a `for` loop, may be added to the beginning loop's 'initialisation' section.

Applying rule 1 and 2 to the `while` loop at line seven of the fragment above, transforms the fragment into:

```
5    i=1;
6    n=0;
7    for(;s[i]>='0' && s[i]<='9';i=i+1) {n=10*n+(s[i]-'0');}
8    r=-n;
```

Two applications of rule 3 further transforms the fragment into

```
7    for(i=1,n=0;s[i]>='0' && s[i]<='9';i=i+1) {n=10*n+(s[i]-'0');}
8    r=-n;
```

Part of the post–processing phase could also include a little syntactic 'cleaning up', for example to remove the need for the $\{\ldots\}$ when the contents consist of a single statement.

The final result is thus:

```
for(i=1,n=0;s[i]>='0' && s[i]<='9';i=i+1)    n=10*n+(s[i]-'0');
r=-n;
```

In a similar manner a syntactically unrestricted quasi static slice can be constructed for the final value of `r` when the initial state has a `'+'` character recorded as the first element of the array `s`, producing the simplified program below:

```
for(i=1,n=0;s[i]>='0' && s[i]<='9';i=i+1)    n=10*n+(s[i]-'0');
r=n;
```

Syntactically unrestricted quasi static slicing for the final value of `r` when the string `s` is empty[7], produces:

```
r=0;
```

Notice that for the original program, conventional slicing and conventional quasi static slicing produce no simplification. However, syntactically unrestricted slicing is able to construct three simpler program fragments which capture three of the important cases which would be used to understand the effect of the original. In this way the unrestricted form of slicing can be used to assist in case analysis. The three cases for the final value of `r`, and their syntactically unrestricted quasi static slices are given in figure 3.

Of course, other interesting cases might also be considered, for example, if the quasi static slicing criterion is $(\{\sigma \mid \sigma(\texttt{s[0]}) = '-' \wedge \sigma(\texttt{s[1]}) = ''\}, \{\texttt{r}\})$ then the slice

```
r=0;
```

will be produced, clearly indicating (together with the slice for $(\{\sigma \mid \sigma(\texttt{s[0]}) = '-'\}, \{\texttt{r}\})$) that the leading `'-'` must immediately precede the string of digits in a negative integer, if the fragment is to behave as intended.

[7]that is, when the first character of the array `s` is the null character.

`for(i=1,n=0;`
`s[i]>='0' && s[i]<='9';`
`i=i+1)`
`n=10*n+(s[i]-'0');`
`r=-n;`
	`r=0;`	`for(i=1,n=0;`
		`s[i]>='0' && s[i]<='9';`
		`i=i+1)`
		`n=10*n+(s[i]-'0');`
		`r=n;`

$\{\sigma \mid \sigma(\mathsf{s[0]}) = '-'\}$	$\{\sigma \mid \sigma(\mathsf{s}) = ""\}$	$\{\sigma \mid \sigma(\mathsf{s[0]}) = '+'\}$

Figure 3: The three syntactically unrestricted quasi static slices

7 Related Work

Conventional slicing is an extremely attractive technique for debugging [51, 36, 29, 32, 3, 30], because it allows attention to focus upon the statements of the program which must contain the defect which leads to an observed fault. However, when slicing to assist program comprehension, the principal concern is *simplification*.

Whilst conventional algorithms are ideal for locating bugs in programs, the syntactically unrestricted form of a slice appears to be better suited to program comprehension (and perhaps thus to *detecting* bugs). Of course conventional approaches to slicing are an integral part of the approach to program comprehension advocated here, as they are used as part of the algorithm for constructing syntactically unrestricted quasi static slices.

The concept of a syntactically unrestricted quasi static slice is closely related to that of partial evaluation [9, 17, 38] (also known as mixed computation [16]). Partial evaluation is a method of specialising a program with respect to an initial prefix of the input to a program.

The process of partial evaluation produces a simplified program (called a 'residual program') and a new state, σ, such that when the residual program is executed in the state σ, the resulting final state is identical to that which would be produced by executing the original program in the initial state. This is known as the partial computation principle [16].

The goal of partial evaluation is to produce a more efficient program, the residual program, by 'hard–wiring' into the original program information about the input to the program. Partial evaluators also have the pleasing property that they can be self–applied [19], which allows compilers to be constructed from self–applied partially evaluated interpreters [28].

The approach presented here differs from work on partial evaluation, largely because of its intended application to program comprehension (although techniques associated with partial evaluation may well be found useful in implementing algorithms).

Partial evaluators and compiler optimizers are concerned with speed of execution, not with improved comprehension, and so, for example, operator strength reduction might be employed to transform `5*x` into `x+x+x+x+x`, whereas for program comprehension, the reverse transformation will be preferable.

Also, partial evaluation produces a single residual program which captures all the original program's computation with respect to partial input information, whereas slicing isolates subcomputations, where the goal is to produce as many simple programs as possible from a single larger and more complex program.

8 Directions for Future Work

The worked example (section 6.2) although small, demonstrates the potential for simplification offered by the quasi static slicing paradigm when freed from the restriction to statement deletion.

An alternative trade–off might entail providing simplification with respect to a set of assumptions (the theorems, which the system cannot prove in acceptable time, but which, through some probabilistic method, it has chosen to assume to hold). In such a situation, the algorithm could provide such assumptions for consideration 'off line' by a human or more sophisticated theorem prover. This approach is adopted (interactively) by Coen–Porisini and his colleagues in their partial executor [12].

More work is also required to establish experimentally, whether this approach assists in program comprehension. Such experiments might take the form of measuring whether a set of programmers find it easier to detect 'seeded bugs' in programs with and without the use of a syntactically unrestricted slicing tool. Such experiments are fraught with methodological difficulties, however, as the time taken to master the tool, the nature of the subjects and method of control are all controversial issues.

An effective syntactically unrestricted slice production algorithm will need to use powerful simplification rules and will need to have the ability to prove simple theorems concerning program predicates. In the rest of this section, various possibilities are briefly discussed. However, a trade–off will surely have to be established between simplification power and slice construction time.

The problem of determining whether or not two expressions are identical is, in general, of course, undecidable. However, considerable progress has been made in this area, as a result of the development of mathematical packages such as Maple and Mathematica [13], and some of the techniques associated with normal forms of expressions used in the associated algorithms may be applicable to the problems of syntactically unrestricted slice construction.

The presence of a set of initial states in the slicing criterion for the quasi static paradigm will mean that certain execution paths become infeasible. For example, the empty program is a quasi static slice of

```
if (p>q)
   x=x+1;
   else if (q==r)
           y=2;
```

with respect to $(\{\sigma \mid \sigma(\mathtt{q}) < \sigma(\mathtt{r})\}, \{\mathtt{y}\})$, because the path which leads to the execution of the statement `y=2;` cannot be executed in any initial state drawn from $\{\sigma \mid \sigma(\mathtt{q}) < \sigma(\mathtt{r})\}$.

Using theorem proving techniques it will be possible for a quasi static slicing algorithm produce the simplest slice, in this case, is the empty program, since it would merely have to prove the theorem

$$q < r \Rightarrow \neg(p > q \wedge q = r)$$

Of course, as slices are to be constructed in real time, care must be taken to select computationally inexpensive approaches to proving such simple theorems.

Alternatively, using a technique borrowed from the theory of program testing [7], it is possible attempt to solve the set of simultaneous equations induced by the predicates controlling the execution of each path. Paths whose corresponding equations have no solution are called infeasible paths [7], that is they can never be executed.

The symbolic state used in the worked example is a mapping from identifiers to expressions. Coen–Porisini et al [12] use a two component state, consisting of a set of bindings of symbolic expressions to identifiers, and a 'path condition', which captures the boolean expression known to hold in the state as a result of the values which must pertain at the predicate nodes of the control flow graph in order for execution to have arrived at the point at which the path condition is constructed.

It would be desirable to allow more elaborate sets of initial states to be defined. Certainly the programmer should be able to use *any* boolean expression in order to establish the set of initial states for which the quasi static slice is to be constructed. This would allow a programmer to generalise the case considered earlier (in the final part of section 6.2), where the set of states was:

$$\{\sigma \mid \sigma(\mathtt{s[0]}) = '-' \ \wedge \sigma(\mathtt{s[1]}) = '\ '\}$$

to the more useful set of states:

$$\{\sigma \mid \sigma(\mathtt{s[0]}) = '-' \ \ \wedge \ (\sigma(\mathtt{s[1]}) < '\mathtt{0}' \ \vee \ \sigma(\mathtt{s[1]}) > '\mathtt{9}')\}$$

Ultimately, it would be extremely desirable to escape the restriction to the expressive power of the language's boolean expression notation altogether, and define such sets of states using predicate calculus. This would allow a programmer to define sets of states such as

$$\{\sigma \mid \exists x.\mathtt{s}[x] = '-' \ \ \wedge \ \forall y.y < x \Rightarrow \mathtt{s}[y] < '\mathtt{0}' \ \vee \ \mathtt{s}[y] > '\mathtt{9}'\}$$

This set of states captures the pleasingly general initial condition that the string contains a '-' character before any digits occur. Using such information in the process of symbolic execution, however, presents many technical difficulties. In [42] Rosenblum defines computable loop structures which can be used to test assertions involving quantification over finite sets. This work is concerned with producing code to automatically check the truth of assertions introduced by the programmer. However, similar constructs might prove applicable for the problem of generating a state which satisfies a predicate involving finite quantification.

9 Conclusion

The quasi static paradigm for program slicing (which, hitherto has received little attention in the program slicing literature), appears to be most suited to program comprehension as it allows a program to be understood as a set of projections, each of which capture the effect of the program when executed in a set of possible initial states, allowing analysis by cases.

The restriction to statement deletion in slice construction has been shown to be an unnecessary hindrance when it is applied to program comprehension, although it is clearly necessary for bug location. Removing this restriction allows a wide range of program transformations to be exploited in the simplification process, and when applied in the quasi static paradigm, yields syntactically unrestricted, quasi static slices which allow a programmer to view a program from many different perspectives, each pertinent to a particular line of analysis.

More work is required to implement and evaluate algorithms for constructing syntactically unrestricted quasi static slices. Such algorithms will inevitably involve a trade–off between simplification power and speed of execution.

Acknowledgements

The authors would like to thank Information Processing Limited for supplying us with their Cantata CASE Tool for automated software testing and measurement and Dan Simpson and his colleagues at Brighton University for their collaboration with this project.

References

[1] Agrawal, H. On slicing programs with jump statements. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Orlando, Florida, June 20–24 1994), pp. 302–312. Available as SIGPlan Notices, 29(6), June 1994.

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

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

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

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

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

[7] Beizer, B. *Software Testing Techniques.* Van Nostrand Reinhold, 1990.

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

[9] Bjørner, D., Ershov, A. P., and Jones, N. D. *Partial evaluation and mixed computation.* North–Holland, 1987.

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

[11] Coen-Porisini, A., and De Paoli, F. SYMBAD: A symbolic executor of sequential Ada programs. In *IFAC SAFECOMP'90* (London, 1990), pp. 105–111.

[12] Coen-Porisini, A., De Paoli, F., Ghezzi, C., and Mandrioli, D. Software specialization via symbolic execution. *IEEE Transactions on Software Engineering 17* (Sept. 1991), 884–899.

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

[14] Elliott, D. Program transformation. BSc Thesis, University of North London, May 1994.

[15] Ernst, M. D. Practical fine–grained static slicing of optimised code. Tech. Rep. MSR-TR-94-14, Microsoft research, Redmond, WA, July 1994.

[16] Ershov, A. P. *On the essence of computation.* North–Holland Publishing, 1978, pp. 392–420.

[17] Futamura, Y., and Nogi, K. Generalized partial computation. In *IFIP TC2 Workshop on Partial Evaluation and Mixed Computation* (1987), D. Bjørner, A. P. Ershov, and N. D. Jones, Eds., North–Holland.

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

[19] Gomard, C. K. A self–applicable partial evaluator for the Lambda Calculus: Correctness and pragmatics. *ACM Transactions on Programming Languages and Systems 14* (Apr. 1992), 147–172.

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

[21] Griswold, W. G., and Notkin, D. Automated assistence for program restructuring. Technical Report CS92–221, Department of Computer Science and Engineering, University of California, Sand Diego, Jan. 1993.

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

[23] Harman, M., and Danicic, S. Using program slicing to simplify testing. *Journal of Software Testing, Verification and Reliability* (1995). To appear.

[24] Harman, M., Danicic, S., Sivagurunathan, B., Jones, B., and Sivagurunathan, Y. Cohesion metrics. In 8^{th} *International Quality Week* (San Francisco, May 29th – June 2nd. 1995), pp. Paper 3–T–2, pp 1–14.

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

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

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

[28] Jones, N. D., Sestoft, P., and Søndergaard, H. MIX: A self–applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation 2* (1989), 9–50.

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

[30] Kamkar, M., Shahmehri, N., and Fritzson, P. Interprocedural dynamic slicing. In *Proceedings of the 4th. Conference on Programming Language Implementation and Logic Programming* (1992), pp. 380–384.

[31] Kernighan, B. W., and Ritchie, D. M. *The C programming language.* Prentice Hall, 1988. Second Edition, (ANSI C).

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

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

[34] Lightfoot, D. Exposing side effects in C programs by systematic program transformation. Unpublished report, Oxford Brookes University, United Kingdom.

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

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

[37] Meyer, U. Techniques for partial evaluation of imperative programs. In *Conference on Partial Evaluation and Semantics–Based Program Manipulation (PEPM)* (1991), Association for Computer Machinery. Proceedings in SIGPlan Notices, 26(9), 1991.

[38] Meyer, U. Report on a partial evaluator for a large subset of pascal. Technical Report 9201, Justus–Liebig Universität, Geissen, July 1992.

[39] Ott, L. M., and Thuss, J. J. The relationship between slices and module cohesion. In *Proceedings of the* 11^{th} *ACM conference on Software Engineering* (May 1989), pp. 198–204.

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

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

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

[43] Schmidt, D. A. *Denotational semantics: A Methodology for Language Development.* Allyn and Bacon, 1986.

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

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

[46] Spivey, J. M. *The Z Notation: A Reference Manual.* Prentice Hall, 1989.

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

[48] Tip, F. *Generation of Program Analysis Tools.* PhD thesis, Centrum voor Wiskunde en Informatica, Amsterdam, 1995.

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

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

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

[52] Weiser, M. Programmers use slicing when debugging. *Communications of the ACM 25*, 7 (July 1982), 446–452.

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