Practical subprogram verification: an approach which uses slicing, metrics and axiomatic verification

M. Harman & S. Danicic

School of Computing, University of North London, Eden Grove, London, N7 8DB, UK

Abstract

DeMillo, Lipton and Perlis [4] and Fetzer [7] have argued that one significant reason for the inapplicability of formal verification techniques such as the Axiomatic Method of Hoare [13] and Dijkstra[5], is the sheer size of real software systems.

In this paper we argue that slicing offers a way to overcome this difficulty opening up the way to a 'slice-and-verify' approach to guaranteeing software quality.

Slicing, introduced by Weiser in [21, 22] and extended in [17, 1, 8, 10], is a technique for simplifying programs by focusing upon subcomponents of their computation.

Since a slice is guaranteed to preserve the original program’s effect with respect to a given set of variables, verification of the slice automatically ‘carries over’ to the overall program from which the slice was constructed. Our approach to the problem of size is to slice the program with respect to some ‘important’ set of variables in the hope that the resulting simplification will make verification a realistic possibility.

Often it will not be easy to tell which set of variables are ‘important’. We discuss the way in which metrics calculated for individual variables may help to identify these ‘important’ components. The metric we use is based on standard data flow relations (for example, those used by Bergeretti and Carré in [3]).

We also show how our ‘effect-minimal’ slices [10] lend themselves to this slice-and-verify approach.
1 Introduction

There are several techniques for program verification which can be used to prove that a program meets its specification [13, 5]. However, anyone who has attempted to apply these techniques will have run into two apparently insoluble problems.

The first problem which arises is one of scale [4]. The simple and elegant proofs found in expository literature such as [14, 2, 9] are no longer practicable when the size of a program rises above even very modest proportions.

The second problem is that most large programs exploit a variety of language idiosyncrasies, for which a formal treatment is either hard to find in the literature or for which formal treatment is cumbersome and consequently error-prone.

In this paper we address the problem of scale, arguing that program slicing, and in particular ‘effect-minimal’ slicing [10], may be used to prove properties of a system in terms of its subcomponents.

Even if formal verification is not the object of the exercise, the mere fact that slicing causes simplification is a great aid to understanding programs and reasoning about them informally.

The work reported here forms a part of Project Project [12], a project set up to investigate and implement analysis, transformation and slicing using standard techniques and effect-minimal slicing.

2 Conventional Program Slicing

A conventional program slice is constructed with respect to a set of variables. Commands in the program which do not affect the value of variables in the set (at some line number within the program) are deleted. This creates a subprogram (the slice) whose effect on the slice set is identical to that of the original. Conventional slicing algorithms attempt to produce a ‘command-minimal’ slice, as the goal of slicing is to simplify the program.

Sadly, the number of commands which may be deleted is often limited because the lines which must be preserved depend upon the effect of lines which we would otherwise wish to delete. Example 2.1 below, illustrates this. The program and its conventional slice are taken from [8].
Example 2.1

```c
main()
{
int a,b,c,d,e,f;
c=4;
b=c;
a=b+c;
d=a+c;
f=d+b;
e=d+8;
b=30+f;
a=b+c;
}
```

A conventional slice of this program with respect to \{b\} in the penultimate line is the simplified program below [8].

```c
main()
{
int a,b,c,d,e,f;
c=4;
b=c;
a=b+c;
d=a+c;
f=d+b;
b=30+f;
}
```

3 Effect–Minimal Slicing

In this case, only two commands were deleted. Only two could possibly be deleted since all the other commands contribute to the value of \(b\) in the penultimate line. Each of these contributing lines also, however, assigns a value to a variable that we are not interested in. These ‘unwanted’ assignments make the conventional slice over complicated for a program solely concerned with the variable \(b\).
An effect-minimal slice is one which not only preserves the effect of the original program upon the slice set, but also has the property that no variables outside the slice set are affected [10]. In producing effect–minimal slices, we are free to construct any program which satisfies this definition. In particular there is no restriction upon the syntactic process by which an effect–minimal slice is to be produced.

This gives rise to simpler slices, since we can exploit symbolic execution to ‘evaluate out’ assignments to unwanted variables. For example, the effect-minimal slice of example 2.1 above with respect to \{b\} is simply:

```c
main()
{
    int b;
    b=46;
}
```

Effect–minimal slices such as this can be constructed by projecting the functional model of an imperative program [11].

Our approach to understanding programs is based upon the idea that one large program can be most easily understood, verified and manipulated as a selection of effect-minimal slices.

4 Program Verification Using Slices

Slicing simplifies a program, and so it simplifies the work required in order to verify properties of the program. For this reason conventional slicing has been suggested as a first step in verification of programs [19]. We claim that effect–minimal slices will further improve the situation, and in the next section we illustrate the use of effect–minimal slicing in verifying properties of software.

5 Selecting Practical Slice Sets

The first step in any slice–and–verify approach is to decide precisely which sets of variables lead to the construction of the most suitable slices. There appear to be two criteria which make for a ‘good’ slice:

1. The slice should be simple enough to significantly reduce the analytic effect involved in understanding the original program’s effect on the slice set.
2. The slice set should contain ‘important’ variables. That is, those which play a significant role in the quality of the software.

An example of a slice which satisfies the first criterion but not the second, is one which tells us the final value of a ‘for loop’ variable.

An example of a slice which satisfies the second criterion and not the first, is the entire program (which certainly tells us the values of all important variables, but, unfortunately, without causing any simplification).

5.1 Integrity Flags

‘Flag’ variables, are often important variables. Slicing a program with a singleton set containing a flag variable will help a programmer to deduce the event signalled by the flag, since the slice will be a projection of the program concerned solely with this event. In constructing such a slice we are mimicking, at compile time, the run-time activities of a programmer using a run-time debugger. The advantage being that, at run-time, anything we discover will concern one execution only, whereas, at compile-time, any properties of the variables investigated, will hold for every run-time situation.

5.1.1 Example: Conventional Slicing on a Flag Variable

The conventional slice below has been constructed for the program in appendix A. We have taken the program-fragment consisting of lines 39 through to 46 inclusive and constructed a conventional slice for the slice set \{flag2\} at line 46.

```plaintext
while (ch<>' ') and not(eoln) do
begin
  if (ch=' ') or eoln then flag2 := true ;
  read(ch)
end
```

flag2 is, in fact, invariant in this slice. In order to verify this formally, we shall have to prove the following theorem:

5.1.2 Simple Axiomatic-Style Proof

The Axioms and Rules used in the proof are listed in appendix B.
Theorem 5.1 (Flag2 Invariant)

Let $W$ denote the slice and
Let $S$ denote the body of the loop
Prove $\{\text{flag2} = v\} W \{\text{flag2} = v\}$ for some $v$
By the Irrelevant Read Rule $\{\text{flag2} = v\} \text{read(ch)} \{\text{flag2} = v\}$
By the Conditional Axiom

$$\{\neg (\text{ch} = \)' \vee \text{eoln}) \wedge \text{flag2} = v\}$$

if (ch = ') or eoln then flag2 := true

$$\{\text{flag2} = v\}$$

By the Composition Rule

$$\{\neg (\text{ch} = \)' \vee \text{eoln}) \wedge \text{flag2} = v\} S \{\text{flag2} = v\}$$

The theorem then follows from the While Loop Rule and DeMorgan's Law.

5.1.3 Effect–Minimal Slicing on the Same Flag Variable

However, if instead of conventional slicing, we use effect-minimal slicing on $\{\text{flag2}\}$ we obtain the empty slice (a program which affects no variables), indicating immediately that flag2 is invariant with no need for any further human investigation.

Perhaps this is more a reflection of the laborious nature of completely formal axiomatic proof than an indication of the power of effect-minimal slicing. Nonetheless, any theorem capable of automated proof can be exploited in the automated construction of an effect–minimal slice. This is not possible for conventional slice construction because of the restriction to command–deletion.

5.2 Investigating Implicit Aspects of Software Quality

Often there is no variable in the program which captures the ‘important’ aspect of quality we wish to verify. In order to form a slice capturing an ‘implicit’ aspect of quality such as this, we shall have to introduce a variable. This approach is described in more detail in [9]. As an example of this approach, we show how the robustness of the program in appendix A could be guaranteed using a slice–and–verify approach.
5.2.1 Example: Verification of Robustness

We add the assignment

\[ \text{robust} := \text{true} \]

between lines 28 and 29 of the program in appendix A, reflecting the fact that, prior to the execution of any commands, the program is robust by definition. We add the assignment

\[ \text{if } j > 20 \text{ then robust} := \text{false} \]

between lines 41 and 42, reflecting the fact that if \( j > 20 \) then the assignment \( \text{realsent}.\text{word}[j] := \text{ch} \) at line number 41 will cause a run-time error: ‘array index out of range’ (or something similar). We construct the conventional slice at the last line with the slice set \{ robust \} for the modified program containing the new variable robust:

\[
j := 1; \ \text{robust} := \text{true}; \ \\
\text{while } (\text{ch} \not= \ ' ') \text{ and not(eoln)} \text{ do} \ \\
\quad \text{begin} \ \\
\quad \quad j := j+1; \ \\
\quad \quad \text{if } j > 20 \text{ then robust} := \text{false}; \ \\
\quad \quad \text{read(ch)} \ \\
\quad \text{end} \ \\
\]

Now, when we attempt to prove that \( \text{robust} \) is always assigned the value ‘true’ by this slice we fail (because the original program is not robust).

5.2.2 Correcting the Program

Having discovered this potential ‘bug’ we could fix it by adding an extra condition, \( j \leq 20 \), to the original program’s while loop predicate. The predicate would thus become:

\[ (\text{ch} \not= \ ' ') \text{ and not(eoln)} \text{ and } j \leq 20 \]
5.2.3 Verifying the Corrected Program

Starting with the corrected program, we can add assignments to the variable robust to check for robustness (as before) creating a program for which we construct the conventional slice below. Once again, we are slicing on \{\text{robust}\} at the end of the program.

\[
\begin{align*}
j &:= 1; \text{robust} := \text{true}; \\
\text{while} \ (\text{ch} \neq \ ' ') \ \text{and} \ \text{not(eoln)} \ \text{do} \\
&\quad \begin{align*}
&\quad j := j+1; \\
&\quad \text{if} \ j > 20 \ \text{then} \ \text{robust} := \text{false}; \\
&\quad \text{read(ch)} \\
&\end{align*}
\end{align*}
\]

We could prove this program robust using the Axiomatic Method. This would require a slightly longer (but no less trivial) proof than that we used in section 5.1.2.

5.2.4 Effect–Minimal Slice for the Corrected Program

The effect–minimal slice we would produce for the corrected program, however, requires a dramatically simpler proof, as it contains the single command:

\[
\text{robust} := \text{true}
\]

The verification of this effect–minimal slice follows immediately from the axiom of assignment:

\[
\begin{align*}
\{\text{true}\} \\
\text{robust} := \text{true} \\
\{\text{robust} = \text{true}\}.
\end{align*}
\]

This most trivial of ‘proofs’ guarantees that the corrected program will never crash as a result of the assignment at line number 41. Using a similar approach we could verify that all such assignments are robust, guaranteeing the robustness of the program as a whole (making corrections where necessary and reconstructing the slice). In a similar way we could introduce other variables in order to slice–and–verify other implicit aspects of software quality.
6 Data Flow Metrics

If little is known about the software under investigation, however, it may not be possible for a programmer to decide upon suitable slice sets. We address this by turning to the way in which data flow information could be used as a metric to help a programmer find suitable slice sets.

A variable referenced by many others is a good candidate for inclusion in a slice set, since its effect must be far-reaching. We can use the work on data flow analysis (for example [3, 15]) to find such variables. In so doing, we shall be treating the data flow properties of a program as a metric for evaluating variables used in the program. Those which have a high reference count being considered to be the most 'important'.

This criterion alone is not sufficient to locate those variables which form a 'good' slice set however, since the slice produced may be very large, reflecting the fact that the values of the variables depend upon a large subcomponent of the overall computation.

A variable which is defined by fewer variables will, in general, produce a smaller slice, so we are looking for a metric which balances the importance of a variable against the amount of computation required to define it.

6.1 An Example Data-Flow Metric

Following Bergeretti and Carré [3], we use the relation $\rho_S$ to capture the variable dependency relation for the command $S$. We write $x \rho_S y$ if the computation of the final value of the variable $y$ by the statement $S$ is dependent upon the initial value of the variable $x$ prior to execution of $S$. The computation of this relation will be syntactic rather than semantic, so that if $S$ is $q := p - p$, then $p \rho_S q$, whereas, in fact, since $p - p$ constant, the relation should not hold between $p$ and $q$. The relation, $\rho_S$, computing the syntactic dependency (as in [3]) will, however, be safe in the sense that it will always contain the true semantic dependency relation.

We can use $\rho_S$ to define a metric for deciding upon a variable's suitability for inclusion in an effect–minimal slice set. $M(P, Q, z)$ measures the suitability of slicing on $\{z\}$ at a line number within the program which is followed by the command–sequence $Q$, and which is preceded by the command–sequence $P$.

We seek to balance the number of references to $z$ in $Q$ against the number of variables in $P$ upon whose value $z$ depends.

We can use the metric to select the best point within a program at which to construct a slice for each variable.

$$M(P, Q, z) = N_Q(z) - A_P(z)$$
where \( N_S(z) = \text{The Cardinality of } \{x \mid x \rho_S z\} \)

and \( A_S(z) = \text{The Cardinality of } \{x \mid z \rho_S x\} \)

The metric generalises naturally from a single variable to a set. The generalised \( M(P, Q, Z) \) below measures the suitability of the slice set \( Z \). We can use this metric to decide the best slice set for a particular point in the program.

\[
M(P, Q, Z) = N_Q(Z) - A_P(Z)
\]

where \( N_S(Z) = \text{The Cardinality of } \bigcup_{z_i \in Z} \{x \mid x \rho_S z_i\} \)

and \( A_S(Z) = \text{The Cardinality of } \bigcup_{z_i \in Z} \{x \mid z_i \rho_S x\} \)

Experimentation is required in order to ascertain the correct weighting for each of the two components \( A_S \) and \( N_S \).

Further refinement may also be necessary in the choice of data flow relation. For example, the program dependence graph [15], may be used in place of \( \rho_S \) in the definition of \( N_S \). This is because the relation, \( \rho_S \), defined in [3] relates original values of variables to final values, whereas the size of a slice may depend more critically upon the number of references to intermediate variables, assigned values as the computation proceeds.

7 Software Re–Use

Having located, sliced and verified the effect of some component of a program we shall be in an ideal position to re-use this component in some other program. The approach therefore also finds an application as a way of extracting reliable, reusable software components from old and perhaps poorly documented software.

8 Conclusion

In this paper we have argued for a ‘slice–and–verify’ approach to guaranteeing software quality. In particular we have advocated the use of effect–minimal slicing, guided by variable–based metrics.

The power of slicing derives from its two fundamental properties:
• All slicing algorithms guarantee that the slice is identical to the original program with respect to the slice set.

• The slice is often significantly simpler than the original program.

The first of these ensures that verification of a property of a slice ‘carries over’ to the program from which it is constructed. The second means that the effort required to verify these properties is often significantly reduced (particularly with effect-minimal slicing).

Metrics defined for set of variables used in the program can be calculated to assist in the selection of suitable slice sets.

Variables can be introduced into a program to allow implicit aspects of its quality to be measured and verified using this approach.

A The Example Program

This program is used as an example throughout the paper.

We have added line numbers to the program to allow us to refer to individual lines in the main body of the paper.

```pascal
program Word(Input, Output) ;
type wordtype = array [1..20] of char ;
sentence = ^entry;
    entry = record
        word : wordtype ;
        next : sentence ;
    end ;
var sent : sentence ;
i : integer ;
flag : boolean ;
ch : char ;
realsent : sentence ;
j : integer ;
flag2 : boolean ;
begin (* main program *)
    writeln('type in a sentence up to 20 words long') ;
i := 1; flag := false; new(realsent);
sent := realsent ;
while i <= 20 do
```

begin
if flag then
realsent^.word:=' 
else
begin
realsent^.word := ' 
read(ch); j := 1; flag2 := false ;
while (ch <> ' ') and not(eoln) do
begin
realsent^.word[j] := ch ;
j := j+1 ;
if (ch = ' ') or eoln
then flag2 := true ;
read(ch)
end ;
if eoln then
begin
flag := true;
realsent^.word[j] := ch
end
end ;
i := i+1 ;
new(realsent^.next) ;
realsent := realsent^.next
end;

B Axiomatic Proof Rules Used

These axioms and rules of Pascal are used in the section 5.1.2.

Axiom B.1 (Axiom of Assignment)

\[ \{ P[e/x] \} x := e \{ P \} \]

Rule B.2 (Composition Rule)

\[ \frac{\{ P \} S_1 \{ Q \} \land \{ Q \} S_2 \{ R \} }{ \{ P \} S_1 S_2 \{ R \} } \]

Axiom B.3 (Conditional Axiom)

\[ \{ P \land \neg(Q) \} \text{if } Q \text{ then } S\{ P \} \]
Rule B.4 (Irrelevant Read Rule)

\[
x \notin \text{Men}(P) \\
\{P\} \text{read}(x) \{P\}
\]

Rule B.5 (While Loop Rule)

\[
\{P \land E\} S \{P\} \\
\{P\} \text{while} E \text{do} S \{P \land \neg(E)\}
\]

References


