Program Slicing Using Weakest Preconditions

Joseph J. Comuzzi and Johnson M. Hart

Peritus Software Services, Inc.
304 Concord Road
Billerica, MA 01821-3485, U.S.A.

Abstract. Program slices have long been used as an aid to program understanding, especially in maintenance activities. Most slicing methods involve data and control flow analysis to determine what statements might affect a set of variables. Here, we develop a more precise slicing concept, called *p-slices*, defined using Dijkstra's weakest precondition (wp), to determine which statements will affect a specified predicate. Weakest preconditions are already known to be an effective technique for program understanding and analysis, and this paper unifies wp analysis and slicing and simplifies existing slicing algorithms. Slicing rules for assignment, conditional, and repetition statements are developed. The authors are currently using these techniques in their work with software maintenance teams and are incorporating p-slice computation into a program analysis tool.

1 Introduction

The concept of *slicing* a program in order to aid understanding, maintenance, and resuse goes back to Weiser [13, 14]. The basic idea is to create a reduced program consisting of only those statements that affect a given variable or set of variables. In this way, the software engineer can concentrate on the relevant parts of the code. Livadas and Croll [11] review the case for slicing and the related techniques of *dicing* and *ripple analysis*, showing the benefits for code understanding, code simplification, reuse, maintenance, and other tasks. [11] then proceeds to present an improved slicing algorithm based on program and system dependence graphs.

The present authors have become convinced of the value of *logical code analysis* (LCA) in all aspects of software maintenance [8, 9] as well as in earlier phases of the software life-cycle. LCA applies the theory of *weakest precondition program semantics* (wp) as developed by Dijkstra [4] and extended in such works as [3, 5, 7]. wp theory was originally developed for *program synthesis* and *proofs of correctness*, but, as mentioned above, we have found that the wp theory, or what we call LCA, is extremely effective for program understanding and maintenance. There are a number of reasons for this, one of which is that LCA efficiently represents control and data flow and precisely captures the relevant information that the slicing algorithms capture with dependence graphs.

The wp-based definition of a slice (called a *p-slice* or *predicate slice*) gives a desirable alternative to the classical slice concept, and, in addition, p-slices can be computed quickly, assuming reasonable constraints.

The paper starts by giving a background in slicing and weakest precondition semantics. Following that, we define the p-slice and show why it is a valuable alternative to Weiser's classical slice definition. Next, we develop some p-slice theorems for alternation and repetition statements and proceed to use the theorems to compute p-slices in some examples. We conclude with some comments on directions for future work.

2 Comments on the Relationship to Formal Methods

This work can be placed in the context of a much larger subject; namely formal methods. Frequently, the goal of formal methods research is the formal specification, development, and verification of programming systems. These systems may be very large and are often mission-critical or even real-time. Formal methods emphasize a program's logical properties as opposed to its operational behavior. The goal of creating error free software requiring minimal testing is, of course, ambitious. Furthermore, the debates on the value of formal methods are frequently polarized between advocates and opponents.

Our goals are less ambitious, as are our claims. We do claim, however, based on [8, 9] and our own experience in software maintenance, that LCA is indispensable in analyzing and understanding the code produced by real programmers. Many examples show the value, even the necessity, of this sort of analysis even for very small (a few lines) code segments. While one critique of formal methods targets scaling problems, it is difficult to comprehend how one could claim to understand and trust a large system that contains unreliable short code segments that no one comprehends. Furthermore, while many formal systems use different semantic models, we feel that Dijkstra's wp is a precise and sharp analytical tool. [5] makes the wp argument effectively.

In turn, classical program slicing as developed in [13, 14, 6, 11] should be considered a "formal method" as it, too, is based on code analysis rather than its actual execution. Therefore, it is not surprising that we are able to treat slicing in terms of wp semantics. Furthermore, the early slicing papers [13, 14] refer to analyzing programs backwards from the point of interest, just as wp analysis depends on backward analysis. Finally, slicing partially addresses the scaling issue by reducing the amount of code under consideration so that other forms of logical analysis are more practical.

3 Program Slices

The objective of slicing program S at some location, p, is to create a smaller program, S', that has exactly the same effect as S (at point p) on a predicate relating

a set of variables, or *slicing criterion*. *S'* is formed from *S* by removing statements so that every statement in *S'* is also in *S*, and statement order remains unchanged. Declaration statements can also be removed, but *S'* should be syntactically correct and executable.

Livadas and Croll [11] give the following "classical" definition, which they say is less general than Weiser's definition [13, 14] but is sufficient for practical purposes.

> Let *S* be a program, let *p* be a point in *S*, and let *v* be a variable of *S* that is either defined (Author Note: "defined" means "assigned a value" rather than "declared") or used at *p*. A static slice or simply a slice of *S* relative to the slicing criterion $\langle p, v \rangle$ is defined as the set of all statements and predicates of *S* that *might* affect the value of variable *v* at point *p*.

Weiser's definition [14] allows for a set of variables, *V*, and *slicing criteria* $\langle p, V \rangle$. The definition is ambiguous in referring to statements that *might* affect the variable, and our p-slices will differ from classical slices by tightening up the definition. In this way, we will be able to create smaller slices than those produced by the classical method.

Gallagher and Lyle [6] have an interesting example based on the source code for the UNIX `wc` utility, which counts characters, words, and lines in text files. Using different slicing criteria, they produce separate word counter, character counter, and line counter programs.

4 Weakest Precondition Semantics

The weakest precondition predicate transformer can be regarded as a function of two arguments. The first argument is a program, *S*, the second is a logical predicate, *P*, and the arguments are separated by periods. The value of the predicate transformer is, of course, another predicate. We say:

$$wp.S.P = Q$$

to mean that it is necessary and sufficient that program *S* start in state *Q* (that is, the predicate *Q* is *true*) to reach state *P*. (Note: We will confine our attention to deterministic programs.)

A few examples and wp calculation rules will help and will also hint at the p-slice definition. In all our examples we will use the following conventions, often with subscripts:

S, *T*, ... denote programs (statements), possibly compound statements.

P, *Q*, ... denote predicates, often in simple propositional calculus and occasionally in first order predicate calculus with quantifiers. These predicates are not part of a program; rather they are used to express the state of program variables.

B will denote a predicate, or logical expression, that is part of a program and is evaluated as part of the execution of alternation and repetition statements.

Sample programs are written in ANSI C [10], much as they might appear to a software engineer.

Dijkstra's language of *guarded commands* (DGC) [3, 4, 5, 7] is used as an *intermediate language* as the wp is defined in this language. This conforms to the operation of our analysis tool which converts the program source (C, PL/I, COBOL, FORTRAN, ...) into DGC as an intermediate language. We will explain DGC as required.

Initially, we will compute the wp on DGC programs, but, later in the paper, we will go directly from C source language to the wp predicate.

We will use the terms "statement" and "program" interchangeably in most cases. Therefore, a statement can be a compound of many other statements. Where there is a difference, we imply that a program has all the required declarations and other features to make it completely executable.

Except where explicitly stated, we will ignore overflow and other issues associated with computer arithmetic, but one example indicates how to treat these issues.

Example 1: S_1 is the single statement program:

In C: `x = x + 1;` or `x++;`

In DGC: $x := x + 1$
Note: := is the DGC assign operator

Here are the weakest preconditions for two predicates:

$$wp.S_1.(x = 2) = (x = 1)$$

$$wp.S_1.(x > 0) = (x > -1)$$

In words, program S will terminate in the state $(x = 2)$ if and only if it starts in the state $(x = 1)$, and it will terminate with x greater than 0 if and only if it starts with x greater than -1. While these results are obvious for these simple examples, it is necessary to have a more general rule.

wp Assignment Statement Semantics: The semantics of the assignment statement $x := e$, for any expression e, is defined to be:

$$wp.(x := e).P = P(x \leftarrow e)$$

where the notation $P(x \leftarrow e)$ means that all occurrences of the variable x in predicate P are textually replaced by (e). The parentheses are required to resolve operator precedence issues. It could be, of course, that the variable does not occur in the predicate so the predicate is unchanged by the wp predicate transformer. The next example is an instance of this.

Note: Assignment statement semantics assume that Hoare's axiom (see below) holds, which causes problems in certain situations involving pointers, variable aliasing, and the like. Bijlsma [2] and Manna and Waldinger [12] discuss this

problem. For now, assume that we can use assignment statement semantics as defined above.

Hoare's axiom is stated in [12] as a state transition, but it is essentially equivalent to the wp assignment semantics.

Example 2: S_2 is the composition of two statements (all variables are signed integers):

In C:
```
x = x + 1;
y = y * y;
```

In DGC:
$x := x + 1;$
$y := y * y;$

First, here is the rule for the composition of statements:

wp Composition Semantics: For any two statements S_a and S_b, the semantics of the composition of the two statements, $S_a; S_b$ is:

$$wp.S_a; S_b.P = wp.S_a.(wp.S_b.P)$$

This is the "backward computation." The wp of the last statement is the predicate used in the wp computation of the first statement.

Continuing with Example 2 for several predicates:

$$wp.(x := x + 1; y := y * y).(x > 1) = (x > 0)$$

$$wp.(x := x + 1; y := y * y).(y > 4) = (y > 2 \vee y < -2)$$

$$wp.(x := x + 1; y := y * y).(x > 1 \wedge y > 4) = (x > 0 \wedge (y > 2 \vee y < -2))$$

Example 3: S_3 is the program:

In C:
```
if (x > 0) x = x - 1;
```

In DGC:
if $x > 0 \rightarrow x := x - 1;$
$[] x \leq 0 \rightarrow$ *skip*;
fi

This example introduces the (*guarded*) *alternation statement*. In general, a list of *guards* (arbitrary predicates) and statement pairs are separated by [] ("fat bar") and surrounded by *if...fi*. A statement will be executed only if its guard is true; if more than one guard is true, a single statement is selected nondeterministically. If no guards are true, the program aborts and no postcondition is possible since the program never reaches the end of the statement. We will define guarded command semantics in a later section; for the time being, it is fairly straightforward to verify the following. (For several of these examples, the fact that x is an integer is important.)

$wp.S_3.(x > 2) = (x > 3)$

$wp.S_3.(x > 0) = (x > 1)$

$wp.S_3.(x \geq 0) = (x \geq 0)$

$wp.S_3.(x < 0) = (x < 0)$

The only remaining DGC construct is the repetition (loop), which is described later. There are also two important special DGC statements: *skip* (used above) and *abort*. They are defined by their *wp* semantics.

$wp.abort.P = false$

for any P because the program never reaches the end.

$wp.skip.P = P$

for any P because the *skip* does not change anything.

Here are several other simple but important *wp* facts that we will use from time to time.

$wp.S.false = false$ for any program S.

This is the "law of the excluded miracle."

$wp.S.true = true$ for any program S, if S contains no *abort* and all repetitions are assured of termination

Program S and program T are semantically equivalent if and only if, for all predicates, P,

$wp.S.P = wp.T.P$

Equivalent programs need not be textually equal, of course; they simply are equivalent as predicate transformers.

For any program S, the following three programs are semantically equivalent:

S

skip; S and

S; *skip*

skip, then, by itself, is the null program. We will eliminate extra *skip* statements without comment.

In summary, the language of DGC statements can be defined recursively, as follows:

- Assignment statements, *skip*, and *abort* are DGC statements, sometimes called *primitive statements*.
- *Compound DGC statements* can be formed from other DGC statements by composition, alternation statements, and repetition statements.

- Nothing else is a DGC statement.

This definition corresponds to the way in which statements in C and other languages are defined recursively with BNF or some other grammar; see, for instance, Aho, Sethi, and Ullman [1]. Whenever we use the term *statement*, we mean it in this sense, regardless of whether we are discussing C or DGC programs.

We will complete the semantic definitions as required, and, what is more, we will define the p-slice for each type of DGC statement.

5 Weakest Precondition Slices

The examples above give a good hint as to how to define slices in terms of the weakest preconditions. Our definition slices code that does not affect a predicate, rather than on the basis of potential effect on a variable, as in the classical definition. First, it is necessary to define a partial order on statements so that we have a concept of removing statements from a compound statement or program.

Definition (Statement Portion): Let S be a statement and let S' be the same as S except that a single statement (not necessarily primitive) in S is replaced by *skip*. Then S' is a *reduction of* S. Let $\subseteq$ be the transitive, reflexive closure of the reduction relation. If T is a statement with $T \subseteq S$, we say that T is a *portion* of S. $T \subset S$ is used to indicate the additional fact that the two statements are not the same.

A slice can now be defined as a portion that has the exact same behavior (semantics) with respect to a given predicate.

Definition (p-slice): Let P be a predicate and S and T be programs. If $T \subset S$ and $wp.T.P = wp.S.P$, then T is a *p-slice of* S with respect to P.

It will turn out that there can more than one p-slice, so we write:

$$T \in PSlice(S, P)$$

A p-slice of S is simply a statement with some statements replaced by *skip* without changing the behavior with respect to P. Again, redundant *skip*s are omitted without comment.

It follows immediately from the definition that:

$skip \in PSlice(S, P)$	if and only if $wp.S.P = P$
$skip \in PSlice(S, true)$	for any statement S, if S contains no *abort* and all repetitions are assured of termination
$skip \in PSlice(S, false)$	for any statement, S

Example 4: Let S_4 (in DGC) be $x := 2 * x$ where $*$ denotes multiplication and x is an integer. Then:

$$skip \in PSlice(S_4, x > 0)$$

However, S_4 has no slices at all with respect to the predicate $(x > 2)$.

This example immediately shows that the p-slice is not the same as the classical slice, which would not slice the statement if x were part of the slicing criteria. Example 4 shows that a statement can sometimes be eliminated even if it assigns a value to a variable in the slicing predicate.

Note: This result ignored the realities of computer arithmetic; i.e., we made the implicit assumption that x was relatively small. Removing this assumption on a machine with 16-bit signed (2's complement) arithmetic gives:

$$wp.S_4.(x > 0) = (x > 0 \wedge x < 16384) \\ \vee (x > -32767 \wedge x < -16384)$$

Therefore, the slice really could not be performed in Example 4 without the underlying assumption that x is small. Programmers, of course, commonly make such assumptions, and programs often break when the hidden assumptions no longer hold.

It is tempting to expect that there would be a unique *minimal slice* (in the sense of the portion relation) for any statement and predicate. Weiser [14] argues that there is no minimal slice on decidability grounds. A simple example will also suffice to show that there are local minimum slices, but no unique minimum slice.

Example 5: Consider the DGC statement:

$$S_5: x := x - 1; x := x + 1; x := -1 + x$$

Two statements, $x := x - 1$ and $x := -1 + x$ are both minimal slices with respect to any predicate involving x.

Example 6: Recall Example 2 where we had:

$$S_2: x := x + 1; y := y * y$$

Using the predicates in Example 2, we have:

$$x := x + 1 \in PSlice(S_2, x > 0)$$

$$y := y * y \in PSlice(S_2, y > 4)$$

However, S_2 has no slices with respect to the predicate $(x > 0 \wedge y > 4)$.

Example 6 is typical of the situation that slicing was probably intended to address originally where program statements can be clearly distinguished by the variables that they affect. Slicing removes those statements that do not affect variables in the slicing predicate.

Classical Slices and P-slices: Example 4 showed how predicate slices can differ from the classical slices which are defined in terms of statements that *might* affect a variable. There is no straightforward way to obtain classical slices from p-slices, even by using free variables (which show all statements that affect a variable in any

way at all). A free variable is one that does not occur in the program and is usually denoted by a Greek letter, as in the next example.

Example 7: Consider (in C)

S_7: `y = 2 * y; if (y > 0) x = 2 * x;` Then:

$wp.\ S_7.(x = \eta) = (y > 0 \wedge 2 * x = \eta) \vee (y \leq 0 \wedge x = \eta)$

The *wp* is exactly the same without the first assignment statement (`y = 2 * y`), so this statement can be sliced (using p-slicing), even though it *might* affect variable `x`. Classical slicing would not remove this assignment.

The next step is to complete the semantic definitions of *alternation* and *iteration* statements and then show our principal *slicing theorems*. Following that, we will give an algorithm to obtain a minimal p-slice for any predicate.

6 Alternation Semantics And Slicing

Following Dijkstra [4], Cohen [3] and Gries [7], the "if" or *alternation* statement takes the form:

if $B_1 \rightarrow S_1$
[] $B_2 \rightarrow S_2$
[] ...
[] $B_n \rightarrow S_n$
fi

Each B_i is a *guard*, that is, an arbitrary predicate, and each S_i is a *guarded statement*. Operationally speaking, a statement is executed for some *true* guard, and the statement aborts if the guards are all *false*. The alternation is nondeterministic as several guards may hold, but only one statement is executed.

Alternation semantics are then defined as:

$wp.(if..fi).P \equiv$
$\exists i: 1 \leq i \leq n: B_i \wedge (\forall i: 1 \leq i \leq n: B_i \Rightarrow wp.\ S_i.P)$

That is, to obtain the postcondition, the necessary and sufficient precondition requires that at least one guard must hold (or else the alternation aborts) and, for every guard that is *true*, the wp of the guarded statement must hold.

In the practical, but special, case of deterministic programs derived from syntactically correct compound *if...then...else* statements (in C or nearly any other programming language), the guards are mutually exclusive and exactly one will hold. Be certain, however, to include *skip* statements for missing *else* clauses, as we did in Examples 3 and 7. Dijkstra and Scholten [5, p. 144] prove what we call the "ITE" (*if...then...else*) rule; namely, if the alternation is derived from compound conditional statements in a deterministic language, then:

$$wp.(if..fi).P \equiv$$
$$(\exists i: 1 \le i \le n: (B_i \wedge wp.\ S_i.P))$$

That is, to obtain the postcondition, it is necessary and sufficient that there is at least one *true* guard (in fact, there is only one) and the wp of the guarded statement holds (for the postcondition). We implicitly used the ITE rule in Examples 3 and 7.

Note: PL/I is the only common language that implements a deterministic conditional statement that resembles the DGC alternation. The C *switch* statement is much different as the *cases* are not general predicates and, unless there is a *break* after every *case* group, the semantics are different.

There are two possibilities when determining p-slices of an alternation statement (in the standard form at the start of this section) with respect to a slicing predicate, P. The first possibility is that one of the conditionally executed statements, S_i, can be sliced, even though the entire alternation cannot be sliced. This gives:

Alternation Slicing Theorem: For an alternation, if, for some j, $B_j \Rightarrow (wp.\ S_j.P \equiv P)$, then S_j may be sliced (replaced with a *skip*). This is immediate, since:

$$\begin{aligned}
& wp.(if..fi).P \\
\equiv\ & \langle\ WP\ of\ alternation\ \rangle \\
& \exists i::\ B_i \wedge (\forall i::\ B_i \Rightarrow wp.\ S_i.P) \\
\equiv\ & \langle\ p.c.\ \rangle \\
& \exists i::\ B_i \wedge (\forall i:\ i \neq j:\ B_i \Rightarrow wp.S_i.\ P) \wedge (B_j \Rightarrow wp.S_j.P) \\
\equiv\ & \langle\ p.c.,\ B_j \Rightarrow (wp.S_j.P \equiv P)\ \rangle \\
& \exists i::\ B_i \wedge (\forall i:\ i \neq j:\ B_i \Rightarrow wp.S_i.\ P) \wedge (B_j \Rightarrow P)
\end{aligned}$$

This later expression is readily seen to be the wp of the alternation with S_j replaced with *skip*.

Note: We have used the calculational proof style developed and advocated by Dijkstra and Scholten [5]. The bracketed comments are an aid to understanding the calculation step, and when we write "p.c." we have used common predicate calculus identities.

Alternation Slicing Corollary: If all the S_j can be sliced, then the entire alternation can be sliced if $P \Rightarrow \exists i::\ B_i$. That is, if P implies the disjunction of the guards (equivalently, at least one guard is *true*, which is always the case when starting from compound conditionals in languages such as C).

This is immediate, since:

$$\begin{aligned}
& wp.(if..fi).P \\
\equiv\ & \langle\ WP\ of\ alternation\ \rangle \\
& \exists i::\ B_i \wedge (\forall i::\ B_i \Rightarrow wp.\ S_i.P) \\
\equiv\ & \langle\ \forall i::\ B_i \Rightarrow (wp.\ S_i.P \equiv P)\ \rangle \\
& \exists i::\ B_i \wedge (\forall i::\ B_i \Rightarrow P)
\end{aligned}$$

$$\equiv \langle\ p.c.\ \rangle$$
$$\quad \exists i::\ B_i \wedge P$$
$$\equiv \langle\ P \Rightarrow \exists i::\ B_i\ \rangle$$
$$\quad P$$

Thus, we may cut an alternation entirely if:

$$P \Rightarrow \exists i::\ B_i \wedge \{\ (\forall i::\ B_i \Rightarrow (wp.\ S_i.P \equiv P)\ \}$$

This latter result is not that important in practice since it is usually simple enough to directly check whether:

$$wp.(if..fi).P \equiv P$$

7 Repetition Semantics and Slicing

The DGC repetition statement (or "do loop," or, in C, the "while loop") is of the form:

$$do\ B \rightarrow S\ od$$

B guards the "loop body," S, which is repeated so long as B evaluates to *true*. If B is initially *false*, then S is never executed. To be useful, of course, S must have "progress properties" that assure that B will eventually be *false* (although many systems, such as an operating system, loop forever without any progress).

Repetition semantics must be defined in terms of a "loop invariant" which is a predicate that holds before and after S; that is, S does not transform the invariant predicate. A necessary and sufficient condition for predicate P_I to be an invariant of S is that:

$$B \wedge P_I \Rightarrow wp.S.P_I$$

This states that, once past the guard, we know that both B and the invariant hold, and, if the invariant is to hold after the loop body, we must have the wp.

When the statement terminates, the guard will be *false*, but the invariant must still hold. Furthermore, the invariant must hold initially. These observations lead to the semantics (wp) of the repetition statement in the special case of loops that make progress and are therefore guaranteed to terminate:

$$wp.(\ do\ B \rightarrow S\ od\ ).(\neg B \wedge P_I) \equiv P_I$$

Invariants are not unique. If a loop body does not contain any *abort* statements or non-terminating loops, then *true* is a trivial invariant. Furthermore, the conjunction of two invariants is also an invariant. In order to understand a loop completely, it is necessary to determine the "strongest" invariant. Needless to say, this is not always an easy, or even possible, task (computational complexity and decidability issues can come into play). In many practical situations, such as isolating defects, partial invariants are sufficient [8, 9].

In the following theorem about repetition statements, S is the repetition body. We also use Gries' $H_k(P)$ [7], which is defined recursively by:

$$H_0(P) \equiv \neg B \wedge P$$
$$H_k(P) \equiv H_0(P) \vee B \wedge wp.S.H_{k-1}(P)$$

$H_k(P)$ holds if and only if the repetition terminates with P *true* in k or fewer iterations.

These definitions permit a more general expression of repetition statement semantics, $wp.(\ do\ B \rightarrow S\ od).P$.

Specifically:

$$wp.(do\ B \rightarrow S\ od).P \equiv \exists k\colon 0 \leq k\colon H_k(P)$$

We first prove a Lemma that will be fundamental to subsequent work.

Iteration Termination Lemma:

$$\{\ B \wedge wp.S.P \equiv B \wedge P\ \} \Rightarrow \{\ \forall k\colon 0 \leq k\colon (H_k(P) \equiv (P \wedge H_k(true)))\ \}$$

Note: $\wedge$ is a higher precedence operator than $\equiv$. Also notice that $wp.S.P \equiv P$ is a stronger form of the left side of this implication. Saying that P is an invariant is a weaker statement than the left hand side of the implication.

We prove this by induction. First, we have:

$$\begin{aligned}
& H_0(true) \wedge P \\
\equiv\ & \langle\ Def\ of\ H_0,\ p.c.\ \rangle \\
& \neg B \wedge P \\
\equiv\ & \langle\ Def\ of\ H_0\ \rangle \\
& H_0(P)
\end{aligned}$$

Next we have:

$$\begin{aligned}
& H_k(true) \wedge P \\
\equiv\ & \langle\ Def\ of\ H_k\ \rangle \\
& (H_0(true) \vee B \wedge wp.S.H_{k-1}(true)) \wedge P \\
\equiv\ & \langle\ p.c.,\ B \wedge wp.S.P \equiv B \wedge P\ \rangle \\
& H_0(true) \wedge P \vee B \wedge wp.S.H_{k-1}(true) \wedge wp.S.P \\
\equiv\ & \langle\ H_0(true) \wedge P \equiv H_0(P),\ Conjunctivity\ of\ WP\ \rangle \\
& H_0(P) \vee B \wedge wp.S.(H_{k-1}(true) \wedge P) \\
\equiv\ & \langle\ Induction\ step\ \rangle \\
& H_0(P) \vee B \wedge wp.S.(H_{k-1}(P)) \\
\equiv\ & \langle\ Def\ of\ H_k\ \rangle \\
& H_k(P)
\end{aligned}$$

Armed with this Lemma, we immediately have:

$$\{\ B \wedge wp.S.P \equiv B \wedge P\ \} \Rightarrow$$
$$\{\ wp.(do\ B \rightarrow S\ od).P \equiv P \wedge wp.(do\ B \rightarrow S\ od).(true)\ \}$$

Since:

$$\begin{aligned}
& wp.(do\ B \rightarrow S\ od).P \\
\equiv & \langle\ Def\ of\ wp.(do...od)\ in\ terms\ of\ H_k\ \rangle \\
& \exists k{:}\ 0 \leq k{:}\ H_k(P) \\
\equiv & \langle\ B \wedge wp.S.P \equiv B \wedge P,\ Lemma\ \rangle \\
& \exists k{:}\ 0 \leq k{:}\ (H_k(true) \wedge P) \\
\equiv & \langle\ p.c.\ \rangle \\
& P \wedge \exists k{:}\ 0 \leq k{:}\ H_k(true) \\
\equiv & \langle\ Def\ of\ wp.(do\ B \rightarrow S\ od)\ in\ terms\ of\ H_k\ \rangle \\
& P \wedge wp.(do\ B \rightarrow S\ od).(true)
\end{aligned}$$

From this last result, we immediately have:

$$\{\ B \wedge wp.S.P \equiv B \wedge P\ \} \Rightarrow \{\ wp.(do\ B \rightarrow S\ od).P \Rightarrow P\ \}$$

In words, if P isn't changed by the repetition body, then the wp of the entire repetition implies P. From the above Lemma we can also derive:

Repetition Slice Theorem:

$$\{\ (B \wedge wp.S.P \equiv B \wedge P) \wedge P \Rightarrow (wp.(do\ B \rightarrow S\ od).(true))\ \} \Rightarrow$$
$$\{\ wp.(do\ B \rightarrow S\ od).P \equiv P\ \}$$

Again, in words, if P isn't changed by the repetition body, and P implies the repetition terminates, then the repetition may be eliminated (sliced with respect to P).

Note: An important way to have $P \Rightarrow (wp.(do\ B \rightarrow S\ od).(true))$ is to have $wp.(do\ B \rightarrow S\ od).(true)$ always *true*; that is, if the repetition is guaranteed to terminate. This is the

Bounded Repetition Slice Theorem:

$$\{\ (B \wedge wp.S.P \equiv B \wedge P) \wedge wp.(do\ B \rightarrow S\ od).(true)\ \} \Rightarrow$$
$$\{\ wp.(do\ B \rightarrow S\ od).P \equiv P\ \}$$

This theorem is helpful when we are certain that a repetition terminates, as would be the case in a C repetition of the common form:

```
for (i=0; i < MAX; i++) { /* Loop Body */ }
```

8 Computing a p-Slice

Practical use of p-slices requires an algorithm to find at least one p-slice for a given predicate. Also, in practice, users do not want to slice an entire program, but, rather,

want to examine a sequence of syntactically complete statements. With this requirement in mind, let:

$$S: \quad S_1; S_2; \ldots; S_N$$

be a composition of individual statements, S_i, each of which is a primitive statement (assignment, *skip*, or *abort*), an alternation, or a repetition. Further, suppose that we want to slice S with respect to the slicing predicate P. That is, we want to compute a member of the set *PSlice* (S, P).

Our current implementation (in the code analysis tool) proceeds as follows:

1. Slice (i.e., replace with *skip*) all statements that do not make assignments to variables in P. This assumes, of course, that there is no potential aliasing so that Hoare's axiom holds. (Caution: Also, do not slice statements affecting loop progress.)
2. Compute $P_i \equiv wp.(S_i; S_{i+1}; \ldots; S_N).P$ for $1 \le i \le N$. Set $P_{N+1} \equiv P$.
3. Starting with $j = N+1$, find the smallest $i \le j$ such that $P_i \equiv P_j$ and slice all the statements from S_i to S_j inclusive. (Note: If $i = j$, there is nothing to slice.)
4. Set $j = i-1$ and repeat from step (3) while j is positive.
5. Next, examine all remaining alternation and repetition statements, performing steps (1) to (4) to slice each guarded statement and loop body. If you are examining statement S_i (in the original labeling), then P_{i+1} plays the role of P (the slicing predicate). In simplifying alternations, combine guard predicates where the guarded statements are the same.

We have found this algorithm to be effective for moderately sized code sequences, although its time complexity is, of course, $O(N^2)$. The first step (removing all obviously irrelevant statements) helps in the processing of larger programs.

Note: By modifying Step 3 to test only whether $P_{j-1} = P_j$, we get a weaker, but faster, linear time algorithm.

9 An Extended Example

The following code is a simplification of some code that we actually encountered in tracing a defect. While one might object that good programmers would never write such code, it is an unfortunate fact that they do. Our experience with maintenance teams dealing with day-to-day problems shows that there is no need to contrive examples.

```
typedef struct connect_table {
 int entry;
 int srce;
 int dest;
 } CTABLE;

#define  TSIZE 100
#define  TRUE  1
#define  FALSE 0

main ( )
{
CTABLE list [TSIZE];
   int count, oldc;
   char found;
   int s_target, d_target;
   count = 0;
   oldc = -1;
   found = FALSE;

   while (count < TSIZE && !found) {
       if (list[count].srce == s_target) {
           if (list[count].dest == d_target)
               found = TRUE;
       } else {
           if (list[count].entry > 0) list[count].entry--;
           oldc++;
       }
       count++;
   }
   if (found) {
       list[count-1].entry++;
   }
}
```

Fig. 1. Code Fragment Before Conversion to DGC

In dealing with this code and the defects associated with it, we were interested in a predicate which we had inferred to be necessary for correct operation.

$$P \equiv (\texttt{count == oldc + 1})$$

Putting this code into a form approximating DGC (our code analyzer does this conversion) gives the code in Figure 2 (the declarations are omitted here). Many programmers quickly note the defects once the code is put into this form, and we consider this to be evidence of the clarity of DGC. It is unfortunate that most programming languages allow, indeed encourage, coding such as Figure 1.

```
count = 0;
oldc = -1;
found = FALSE;

while (count < TSIZE && !found) {
    if (list[count].srce == s_target
        && list[count].dest == d_target)
            { found = TRUE; }
    else if (list[count].srce == s_target
        && list[count].dest != d_target)
            { /* skip */ }
    else if (list[count].srce != s_target
        && list[count].entry > 0)
            { list[count].entry--; oldc++; }
    else if (list[count].srce != s_target
        && list[count].entry <= 0)
            { oldc++; }
    count++;
}
if (found)
    { list[count-1].entry++; }
else if  (!found)
    { /* skip */ }
```

Fig. 2. Code Fragment in a DGC-Like Form

It is now possible to determine the p-slice slices, shown in Figure 3. The defect (with respect to maintaining the slicing predicate) is now clear.

```
count = 0;
oldc = -1;
found = FALSE;

while (count < TSIZE && !found) {
    if (list[count].srce == s_target
        && list[count].dest == d_target)
            { found = TRUE; }
    else if (list[count].srce == s_target
        && list[count].dest != d_target)
            { /* skip */ }
    else if (list[count].srce != s_target)
            { oldc++; }
    count++;
}
/* Slicing Predicate: count == oldc + 1  */
```

Fig. 3. A Member of *Slice* (Figure 2, `count == oldc + 1`)

Another interesting slicing predicate is one that asserts that all `entry` fields are positive.

10 A Repetition Example

The following simple loop illustrates one of many useful applications of the *Repetition Slice Theorem*. Consider the following code fragment (in C):

```
int c;
char exit;
    . . . . .
while (!exit) {
    c = c - 1;
    /* . . . code that does not affect c . . . */
    c = c + 1;
}
```

Fig. 4. A Loop with a Positive Counter

Using the slicing predicate:

$$c \geq 0$$

the entire loop can be sliced. Generalizing, suppose that instead of incrementing and decrementing `c` (a counter), we instead allocate and free a resource or enter and leave a critical section. Alternatively, a data structure (such as a search tree) is manipulated at the top of the loop and re-assembled at the bottom (consider an insertion operation).

In all these cases, the slicing predicate states a "safety property." The fact that the loop can be sliced with respect to this property shows that the loop is correct with respect to this safety property, which is useful information. Classical slicing would not yield this information.

Alternatively, consider the following variation, which has a bug with respect to the safety property:

```
while (!exit) {
    c = c - 1;
    if (. . .) {
        exit = true;
    } else {
        c = c + 1;
        /*   other code   */
    }
}
```

Fig. 5. A Defective Loop

This loop would not be sliced using the slicing, or safety, predicate, $c \geq 0$, highlighting the defect.

11 Conclusions and Future Work

Predicate slicing is a useful logical code analysis technique during software maintenance. We also feel that it can be useful during other parts of the software life cycle, including code inspections and quality assurance. We have used this form of slicing successfully to isolate code defects and expect to deploy automated, tool-based slicing in the near future.

There are a number of challenging additional problems that we will need to address in the future in order to realize the full powers of these techniques. These problems appear, in fact, in any use of formal methods to aid program understanding and maintenance. These problems include:

1. Aliasing is a significant problem, and C's liberal use of pointers aggravates the problem beyond the issues examined by Bijlsma [2]. We have found Bijlsma's approach, however, to be effective. The problem also occurs with arrays, especially where indices can go out of bounds.
2. It can be useful to include assertions in the slicing predicates. These assertions typically state "safety properties" to the effect that an array index is within bounds, that a pointer only contains addresses of one type of object, that a counter is always positive, that an array is sorted, and so on. Then, predicate slicing will expose statements that could violate these safety properties. For example, the following assertion could be useful in slicing the code of Figure 1.

   $$0 \leq count \leq N \;\&\&\; -1 \leq oldc \leq N$$

3. Currently, procedures are processed by including the procedure code in-line, as in Gries [7]. We would like to extend our analysis tool to allow the use of procedure specifications (in terms of weakest precondition semantics), possibly extracted from a library.

References

1. Aho, Sethi & Ullman, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.
2. A. Bijlsma, "Calculating with pointers," *Sci. Comput. Programming*, vol. 12, pp. 191-205, 1989.
3. E. Cohen, *Programming in the 1990s.* NY: Springer-Verlag, 1990.
4. E. W. Dijkstra, *A Discipline of Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1976.
5. E.W. Dijkstra and C.S. Scholten, *Predicate Calculus and Program Semantics.* NY: Springer-Verlag, 1989.

6. K. B. Gallagher and J. R. Lyle, "Program slicing in software maintenance," *IEEE Trans. Software Eng.*, vol. 17, no. 8, Aug. 1991.
7. D. Gries, *The Science of Programming*. NY: Springer-Verlag, 1981.
8. J. M. Hart, "Experience with logical code analysis in software reuse and re-engineering," in *AIAA Computing in Aerospace 10* (San Antonio, TX), Mar. 28-30, 1995, pp. 549-558.
9. J. M. Hart, "Experience with logical code analysis in software maintenance," *Software Practice and Experience*, vol. 25, no. 11, pp. 1243-1262, 1995.
10. B. Kernighan and D. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.
11. P. E. Livadas and S. Croll, "A new algorithm for the calculation of transitive dependencies," *Software Maintenance: Research and Practice*, vol. 7, pp. 151-176, May-June 1995.
12. Z. Manna and R. Waldinger, "Problematic features of programming languages: A situational-calculus approach," *Acta Informatica*, vol. 16, pp. 371-426, 1981.
13. M. Weiser, "Programmers use slices when debugging," *Commun. Assoc. Comput. Mach.*, vol. 25, pp. 446-452, July 1982.
14. M. Weiser, "Program slicing," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 352-357, July 1984.