Properties of Slicing Definitions

Martin Ward
Software Technology Research Lab
De Montfort University
The Gateway,
Leicester LE1 9BH, UK
Email: martin@gkc.org.uk
http://www.cse.dmu.ac.uk/∼mward/

***Abstract*—Weiser's original papers on slicing defined the concept in an informal way. Since then there have been several attempts to formalise slicing using various formal methods and semantics of programs. In this paper we start by defining some properties that a definition of slicing might reasonably be expected to satisfy and then compare different definitions of slicing to see which properties are satisfied.**

Properties are classified into "floor" requirements: all slices satisfying the property must be considered as valid, and "ceiling" requirements: slices which do not satisfy the property must not be considered valid. Any slicing relation which lies above a "floor" requirement, or below a "ceiling" requirement, satisfies the property in question.

The main result of the paper is the proof that, given a certain property of the programming language (informally: it is possible to write an infinite loop in the language), two of the most basic and fundamental properties of slicing are sufficient to completely characterise the semantic part of the slicing relation. These properties are: behaviour preservation and truncation.

I. Introduction

Many papers on slicing start with an informal or more formal definition of slicing, and then go on to present a new slicing algorithm, or the results of analysis of an existing slicing algorithm. Occasionally, various properties of the slicing relation may be mentioned: for example, the fact that the slice of a terminating program should be a terminating program.

In this paper we will start with a list of properties that we might reasonably expect a slicing relation to hold, and then examine a selection of published definitions of slicing to see which properties are satisfied and, more interestingly, which properties are not satisfied.

Some properties will use the following definition:

Definition 1.1: A primitive statement (that is: any statement which is not composed of other statements) is *x-preserving*, for a given variable x, if the value of x after executing the statement is the same as the value of x before executing the statement. A compound statement is *x-preserving* if every primitive statement composing it is x-preserving.

For example, the assignment $x := x$ is x-preserving, as is an assignment to any other variable. The statement **while true do skip od** is also x-preserving. The compound statement $x := x + 1;\ x := x - 1$ is *not* x-preserving since neither of the two primitive assignments is x-preserving.

If X is any set of variables, then a statement **S** is X-preserving if **S** is x-preserving for every variable x in X.

II. Properties

In this section, we will assume for simplicity that we are slicing on the end of the program. In the following, if **S** and **P** are programs and X is a set of variables, then we say "**S** is a slice of **P** for X" as shorthand for "**S** is a valid slice of **P** when we are slicing at the end of **P** on the set X of variables of interest".

Most properties are either syntactic or semantic: a syntactic property is a property of the syntax of the program while a semantic property is determined by the semantics of the program. The property of being x-preserving is intended to be a syntactic property which can be determined by a simple inspection of the primitive statements. One could argue the assignment $x := f(x)$ may or may not be x-preserving, depending on the semantics of the function f: instead we define a statement to be x-preserving only when it can be determined from the syntax of each primitive statement that the value of x after executing the statement is always the same as the value before executing the statement, for any possible semantics.

Generally, the properties fall into two categories: "floor" requirements and "ceiling" requirements. A "floor" requirement lists things which *must* be included in the slicing relation, in order for the relation to satisfy the requirement. For example, Property 3 states that the slicing relation must allow **P** itself as a valid slice of **P** for any variable set. A "ceiling" requirement defines things which must *not* be included in the slicing relation. For example, Property 6 says that a non-terminating program *cannot* be a valid slice of a terminating program.

If we fix on a particular original program and a slicing criterion, then a floor requirement defines a set of programs which must all be considered valid slices, for the slicing relation in question to satisfy the requirement (there may be other slices outside the given set). A ceiling requirement defines a set of potential slices such that all the valid slices must be in the given set: anything outside the given set must not be a valid slice, for the slicing relation in question to satisfy the requirement (the slicing relation does not have to include all the slices in the set).

(A floor requirement is like those signs outside fairground rides: "You must be this tall to go on this ride". A ceiling requirement is like those signs outside children's play areas: "You must not be taller than this to play on this equipment").

More formally: if we define a slicing relation as a set of triples $\langle \mathbf{S}, \mathbf{P}, X \rangle$ where **S** and **P** are programs and X is a set of variables, then:

$$R =_{\text{DF}} \{ \langle \mathbf{S}, \mathbf{P}, X \rangle \mid \mathbf{S} \text{ is a slice of } \mathbf{P} \text{ for } X \}$$

If we define a floor requirement F as the set of all triples which meet the requirement, then for R to satisfy F we must have $F \subseteq R$. For simplicity, we will also define a ceiling requirement C as the set of all triples which are valid according to the requirement. So the invalid triples, none of which must be in R, are precisely those outside C. With this definition, a slicing relation R satisfies C when $R \subseteq C$.

The first four properties are trivial, and non-controversial floor properties: most people would agree that any slicing relation ought to include these slices as a minimum:

1) **Weaken Criterion:** If **S** is a slice of **P** for X and X' is a subset of X, then **S** is also a slice of **P** for X'. (Syntactic).
2) **Strengthen Criterion:** If **S** is a slice of **P** for X and variable y does not appear in **P** or **S**, then **S** is also a slice of **P** for $X \cup \{y\}$. (Syntactic).
3) **Identity Slice:** For any program **P** and set X, **P** is a slice of **P** for X. (Syntactic).
4) **Total Slice:** If **P** is X-preserving then **skip** is a valid slice of **P** for X. (Syntactic).

The next four properties are ceiling properties which define statements which should not be considered slices (or, equivalently, define properties which all slices must have):

5) **Behaviour Preserving:** If **S** is a slice of **P** for X, then for any initial state on which **P** terminates, **S** must also terminate and, the final value of any variable x in X must be the same for both **S** and **P**. For nondeterministic programs, the set of possible final values must be the same for both **S** and **P**. (Semantic).
6) **Termination Preserving:** If **S** is a slice of **P** then **S** should terminate on each initial state for which **P** terminates. In other words, you cannot have a non-terminating slice of a terminating program. (Semantic).
7) **Non-termination Preserving:** If **P** does not terminate for some initial state, then the slice **S** must also not terminate for that initial state. (This property is "preservation of non-termination", which is different from "non-preservation of the property of termination"!) (Semantic).
8) **Syntactic Slice:** If **S** is a slice of **P**, then it should be possible to construct **S** from **P** by deleting statements only (or by replacing statements by **skip** statements). Replacing statements by **skip** statements makes it clear *which* statements have actually been deleted. For example, if **P** is the program $x := 1;\ x := x+1;\ x := 1$ then a valid slice is the program $x := 1$, but it is not clear which of the original $x := 1$ assignments has been deleted and which has been retained. If we replace statements by **skip** statements then we can distinguish the two slices: $x := 1$; **skip**; **skip** and **skip**; **skip**; $x := 1$. Note that this is a purely syntactic relation which is orthogonal to all the purely semantic relations in this list. See [23] for the formal definition of the *reduction* relation. (Syntactic).

The next seven properties are floor properties which define statements which should be allowed as slices:

9) **Replacement Property:** If any component of a statement is replaced by a semantically-equivalent component, then the resulting program should be semantically equivalent. In terms of slicing: replacing a component of the slice by a semantically equivalent component should produce a valid (semantic) slice. Similarly, the original slice should be a valid slice of any program generated from the original program by replacing any component by a semantically equivalent component. Note that this conflicts with Property 8, so for syntactic slicing the replacement property should only apply to a replacement which is also a reduction. (Syntactic and semantic).
10) **Ditchability:** The slicing relation should allow deletion of any code which does not (directly or indirectly) affect the value of any variable of interest. This definition is purposefully vague about what precisely counts as "affecting" the variables of interest, for reasons which will become clear later. (Syntactic and Semantic).
11) **Truncation:** A special case of ditchability is deleting code at the very end of the program which does not modify any variable of interest. In a sense, this is the simplest possible example of deleting irrelevant code. A precise formal definition of the property is the following: if **P** is of the form $\mathbf{S}_1;\ \mathbf{S}_2$ and $\mathbf{S}_2$ is X-preserving, then $\mathbf{S}_1$ should be a valid slice of **P** on X. (This is a purely syntactic subset of Property 10).
12) **Unreachable Code:** The slicing relation should allow deletion of unreachable code: that is, code which is syntactically unreachable (e.g. a procedure definition for which there are no calls), or code which cannot be executed on any possible execution of the program. An example of the latter case is the program:

if false then $x := 1$ **fi**

To satisfy this property, a slicing relation should allow deletion of this statement, even when x is a variable of interest. More generally:

$x := 1$; **if** $x = 1$ **then** $\mathbf{S}_1$ **else** $\mathbf{S}_2$ **fi**

A slicing relation should allow deletion of $\mathbf{S}_2$, since it will never be executed. (Syntactically unreachable code is a syntactic property, the more general case is both syntactic and semantic—and, like most semantic properties, non-computable).

13) **Dependency Slice:** The System Dependence Graph (SDG) [16] is an extension of the Program Dependence Graph (PDG) [7,14,17,18]. This graph records control and data dependencies. A slice can be constructed from the SDG by forming the transitive closure of control and data dependencies [14] . To satisfy this property, the slicing relation should allow all slices so constructed as valid slices. (Syntactic).

14) **Pruned Dependency Slice:** The SDG is constructed from the Control Flow Graph (CFG) of the program. Naïve algorithms for constructing CFGs may include edges which are never taken during execution. A "pruned CFG" is a CFG from which unreachable edges and nodes have been deleted. A "pruned SDG" is an SDG constructed from a pruned CFG. This property requires that any slice constructed by applying the SDG algorithm to a pruned SDG should be a valid slice. This is therefore a special case of Property 12 (unreachable code). As an example, consider the statement:

    if false then $x := y + 1$ **fi**

    where x is the variable of interest. A direct translation to a CFG followed by computation of the SDG slice would include the statement $x := y + 1$. Any preceding code to compute the value of y would also be included. Note that Property 12 (unreachable code) requires that this statement can be deleted, but does not require deletion of the code which computes y. (Either syntactic, or syntactic and semantic, depending on how the unreachable code is determined).

15) **Existing Implementations:** What actual slicers do is useful: in other words, if a well-known slicer deletes some code and it is generally agreed that the code can be deleted (i.e. it is not a bug), then the definition of slicing should allow the deletion. Note that we should be careful not to mandate "flaws" in current implementations: i.e. just because an implementation does not delete a particular statement does not mean that deleting the statement produces an invalid slice. (Syntactic).

The final property is a ceiling property which is rather more controversial than the others:

16) **Small Ripple Principle:** Small changes in the program should produce only small changes in its slices. At first sight, this may seem a reasonable request, but consider the second example from Property 12. A slicer with Property 12 is allowed to delete $\mathbf{S}_2$, leaving $\mathbf{S}_1$ as the slice. Now we make the smallest possible change to the program (change a single bit) to give:

    $x := 0$; **if** $x = 1$ **then** $\mathbf{S}_1$ **else** $\mathbf{S}_2$ **fi**

    Now the slicer can delete $\mathbf{S}_1$ leaving $\mathbf{S}_2$ as the result. Now, $\mathbf{S}_1$ and $\mathbf{S}_2$ may have arbitrarily large differences, so any slicer which satisfies Property 12, is such that a small change in the program can produce arbitrarily large changes in its slices. Because of this problem, the "small ripple principle" will not be considered any further. (Syntactic).

III. Some Published Definitions of Slicing

In this section we will consider a selection of formal definitions of slicing from the literature against the properties listed in the previous section.

A. *Weiser's Definition*

Weiser [25,26,27] defined a slice $\mathbf{S}$ as a *reduced, executable program* obtained from a program $\mathbf{P}$ by removing statements, such that $\mathbf{S}$ replicates part of the behaviour of $\mathbf{P}$. He mentions that slices of a terminating program must also be terminating, so this definition clearly satisfies Properties 5 and 6 (behaviour and termination preservation). Weiser does not directly address Property 7 (non-termination preservation), but his algorithm for slicing will delete code which has no dependencies on the variables of interest, regardless of whether or not it terminates. So Weiser's form of slicing does not satisfy Property 7 (non-termination preservation).

```
1  read(X)
2  if (X)
         then
         ...
         perform any function not involving X here
         ...
3        X := 1
4  else  X := 2 endif
5  write(X)
```

Figure 1. Weiser's Example Program

Regarding Property 12 (unreachable code), Weiser's algorithm does not necessarily delete unreachable code but his argument proving the non-existence of an algorithm to find minimal slices assumes that unreachable code *can* be deleted to give a valid slice. He presents the program fragment in Figure 1 and writes: *Imagine slicing on the value of X at line 5. An algorithm to find a statement-minimal slice would include line 3 if and only if the function before line 3 did* *halt. Thus such an algorithm could determine if an arbitrary program could halt, which is impossible*. Properties 13 and 14, which concern slices computed from dependencies, are therefore also satisfied by Weiser's form of slicing.

B. Reps and Yang's Definition

Reps and Yang [19] *define* a slice in terms of data and control dependencies. In effect they make Property 13 a ceiling requirement as well as a floor requirement. For Weiser's example in Section III-A, therefore, line 3 *must* be in the slice, regardless of whether or not the function before line 3 terminates. The main results of their paper are to prove that the PDG slice satisfies Properties 5 (behaviour preservation) and 6 (termination preservation). They note that a slice may terminate when the original program diverges, so their definition does not satisfy Property 7 (non-termination preservation).

C. Binkley and Gallagher's Definition

Binkley and Gallagher's survey of program slicing [5] defines a slice as follows:

Definition 3.1: For statement s and variable v, the slice **S** of program **P** with respect to the slicing criterion $\langle s;\ v\rangle$ is any executable program with the following properties:

1) **S** can be obtained by deleting zero or more statements from **P**.
2) If **P** halts on input I, then the value of v at statement s each time s is executed in **P** is the same in **P** and **S**. If **P** fails to terminate normally s may execute more times in **S** that in **P**, but **P** and **S** compute the same values each time s is executed by **P**.

This definition satisfies Property 5 (behaviour preservation). It also satisfies Property 6 (termination preservation): at least for cases where the slicing point is at the end of the program. Again, Property 7 is not satisfied: if **P** fails to terminate normally, then the slice may execute the statement in the slicing criterion more times than **P**. In the case of end-slicing, therefore, the slice may terminate (executing the statement at the end of the program exactly once) when the original program did not terminate.

However, this definition does not satisfy Property 3 (identity slice) when the programming language includes nondeterminism. Consider the program **P**:

if true $\rightarrow x := 1$
$\square$ **true** $\rightarrow x := 2$ **fi**;
$y := x$

where we are slicing on the value of x at the assignment to y. According to Binkley and Gallagher's definition, there are *no* valid slices of **P**! Even **P** itself is not a valid slice: since the value of x at $y := x$ may be different each time $y := x$ is executed. This might appear to be unimportant in practice (since most executable programs are also deterministic), but in the context of program analysis and reverse engineering it is quite common to "abstract away" some implementation details and end up with a nondeterministic abstraction of the original program. If one wishes to carry out further abstraction on this program via slicing, then it is essential that the definition of slicing, and the algorithms implementing the definition, are able to cope with nondeterminism.

D. Slicing Defined In Terms Of Refinement

Weiser's informal definition of a program slice (see Section III-A) contains two elements:

1) A syntactic element: the slice is obtained from the original program by deleting statements and
2) A semantic element: the slice preserves part of the behaviour of the original program.

In [20] the author defined slicing as a combination of a syntactic relation (reduction) and a semantic relation (refinement on a restriction of the semantics of the program to the variables of interest). A slice is a *reduction* of the original program if it can be obtained from the original program by replacing statements by **skip** statements. This is equivalent to statement deletion, since a **skip** statement has no effect, but allows the programmer to map each statement in the slice back to the corresponding statement in the original program in an unambiguous way.

Using refinement as the semantic relation will allow irrelevant code to be deleted, regardless of whether or not the deleted code terminates. So Properties 10 and 11 (ditchability and truncation) are satisfied. Properties 13 and 14 are also satisfied. As with many other definitions, Property 7 (non-termination preservation) is not satisfied.

A major flaw with this definition of slicing is that it is not behaviour preserving for nondeterministic programs (Property 5). For example the nondeterministic program:

$x := 1;$
if true $\rightarrow x := 1$
$\square$ **true** $\rightarrow x := 2$ **fi**

always terminates and assigns x the value 1 or 2 nondeterministically. A refinement of this program is:

$x := 1$

which always assigns the value 1. A programmer cannot analyse a slice of the program and then deduce facts about the value assigned to a variable in the original program (assuming that the original program terminates). In the example above, the slice always sets x to the value 1, but the original program may set x to the value 2. So this definition of slicing does not satisfy Property 5 (behaviour preserving).

This flaw was recognised and fixed in a later formalisation of slicing (see Section III-G).

E. Slicing Defined In Terms Of Equivalence

Noting that Weiser's definition is a combination of a syntactic relation and a semantic relation, Harman et al

[12] presented a very general definition of a slice as a combination of:

1) A computable partial order and;
2) An equivalence relation on the semantics of the program.

Since the partial order is a *computable* relation, it must be a purely syntactic relation (due to the Halting Problem, the semantics of a program cannot be determined by a computable function, so the partial order cannot be a semantic relation). This implicit fact is made explicit in later work [4,11] where the partial order is defined as a syntactic relation. The semantic equivalence relation is not otherwise specified, so this formulation defines a very large number of "slicing" relations: including degenerate relations in which any program is a slice of any other program, and relations in which no program has any slices apart from itself. It is presumably intended that the slices should be behaviour preserving.

Defining slices in terms of semantic equivalence (using a standard semantics) will not allow deletion of irrelevant code when the irrelevant code may or may not terminate. For example, suppose we are slicing on the final value of x in the following program:

…lots of code which determines the value of y …;
$x := 1$;
while $y = 0$ **do skip od**;

The **while** loop does not affect the value of x, so we would like to delete it (along with all the code which sets y). But if the preceding code sets y to the value zero, then the loop will not terminate. So the loop, and all the code which determines the value of y, has to be included. So this definition does not satisfy Properties 10 or 11.

Kamkar [15] defined slicing according to strict semantic equivalence. This form of slicing therefore preserves Properties 6 and 7: preserving termination and non-termination of the original program, but at the expense of larger slices in general. This approach does allow the programmer to deduce that the original program terminates, if it can be proved that the slice terminates. But it would make more sense to divide the problem into two smaller, orthogonal problems: (1) Determine the conditions under which the original program terminates, and (2) Compute a slice without needing to preserve non-termination (since removing this restriction is likely to give a smaller slice).

In [4], Binkley et al refer to the lazy (or demand driven) semantics of [6] and in [3] they claim that "slicing is more closely related to forms of lazy semantics than to standard semantics". With a lazy semantics, roughly speaking, code is only executed if the value of an assigned variable is "demanded". For example, if we are slicing on x, then a loop such as:

while $y > 0$ **do** $y := y + 1$ **od**

(where y can hold any integer value) is semantically equivalent to **skip**, regardless of whether or not it terminates. If the value of y is demanded, then the loop is equivalent to **abort** if $y > 0$ initially.

Lazy semantics has a number of problems:

1) The semantics does not satisfy the replacement property. The loop **while true do** $x := x$ **od** is an **abort** if x is demanded. But the assignment $x := x$ is equivalent to **skip**. Replacing the assignment by an equivalent **skip** statement in the loop gives: **while y do skip od**, which in lazy semantics is always equivalent to **skip**. So, replacing a component of a program by an equivalent component does *not* necessarily produce an equivalent program, and the slicing relation does not satisfy Property 9 (replacement).
2) The semantics defines some infinite loops as terminating. But in reality, these loops do *not* terminate when the program is actually executed on a machine. So the slicing relation is neither termination preserving nor non-termination preserving (for the usual meaning of "termination"). For example, consider the program:

   $y := 0$;
   $y := 1$;
   while $y = 0$ **do skip od**

   This program always terminates. A lazy-semantic equivalent slice (when x is demanded) is:

   $y := 0$;
   while $y = 0$ **do skip od**

   which never terminates (this example contradicts the claim in [13] that slicing using lazy semantic equivalence "shall never introduce nontermination"). On the other hand, the program:

   $y := 1$;
   $y := 0$;
   while $y = 0$ **do skip od**

   does not terminate, but the lazy-semantic equivalent slice:

   $y := 1$;
   while $y = 0$ **do skip od**

   does terminate.
3) Since some infinite loops are equivalent to **skip** and others are not, the semantics sometimes allows code after an infinite loop to be deleted and in other cases it does not. Consider Weiser's example program in Figure 1 and suppose that the code before line 3 is a non-terminating loop such as **while true do skip od**. Weiser says that line 3 can be deleted in that case. The lazy semantics allows deletion of the infinite loop, but does *not* allow deletion of the assignment on line 3, even though we know that the assignment can never be reached. So this definition of slicing

does not satisfy Property 12 (deletion of unreachable code) or Property 14 (pruned SDG). Ironically, if the infinite loop contained the assignment $x := x$, then the assignment on line 3 can be deleted, but now the loop cannot be deleted!

A later reformulation of the lazy semantics [9] does satisfy the replacement property, and so fixes problem 1, but the other problems remain.

The intent of lazy semantics is to "capture" the semantics of program dependency [6,9]. But the semantics of program dependency is not completely captured. For example, with the program:

if false then $x := 1$ **fi**

the PDG algorithm says that the final value of x depends on the assignment in the **if** statement, but the lazy semantics says that this assignment can be deleted. The pruned-SDG algorithm will prune the assignment and therefore give a result consistent with the lazy semantics. But the lazy semantics does not capture the semantics of the pruned-SDG slice either! Consider the example:

while true do skip od;
$x := 1$

The pruned-SDG property says that the assignment to x is not needed: since it can never be reached by any execution of the program. To be precise: in the CFG, the edge from the loop test to the assignment is labelled **false**, and so can be deleted by the pruning process, along with the assignment. But the lazy semantics says that the assignment *is* needed!

Binkley, Harman and Danicic [4] define a *state trajectory* as a *finite* sequence of pairs of the form: (n, s) where n is a line number and s is a state. They define a projection function on state trajectories which restricts the trajectory to the line number of interest and the variable(s) of interest. They define a form of "equivalence" which states that for every initial state, the state trajectories for the two programs must agree on the variable(s) of interest at the line number of interest. The problem with these definitions is that a non-terminating program does not give rise to *any* state trajectory (since a state trajectory is defined to be finite). So a non-terminating program agrees (vacuously) with *any* other program on any initial state and slicing criterion! So *any* non-terminating program is a valid slice of *any* other program. (This does not mean that any program is semantically equivalent to any other program: their semantic relation is not transitive, so it is not actually an equivalence relation).

A problem with *every* attempt to define slicing in terms of semantic equivalence (not just lazy semantics) is that *no* slicing definition based on equivalence can satisfy both Property 5 (behaviour) and Property 11 (truncation): this will be proved later in the paper (Theorem 4.5). A corollary of this result is that no slicing relation based on semantic equivalence satisfies Property 10 (ditchability).

F. Slicing Defined In Terms Of Transfinite Semantics

Giacobazzi and Mastroeni [10] claim that program slicing requires techniques which can observe "computations that may occur after a given number of infinite loops", claiming that "This generalization is necessary to deal with ... program slicing", presumably because such computations are included in the traditional dataflow-bases slices. They present a transfinite semantics which defines the behaviour of a program as a sequence of states whose length may be a transfinite ordinal. But this seems to confuse a necessarily imperfect *algorithm* for slicing with the *definition* of slicing. There is no need to introduce a new non-standard semantics for slicing in order to work around a deficiency in a popular algorithm for computing slices. (Their semantics also does not satisfy the replacement property, Property 9.)

On the other hand, if their aim is really to capture the behaviour of a dataflow-based slicing algorithm in the semantics of the program, then their semantics fails to do so, because although code after a **false** branch out of an infinite loop is included in the semantics, code after a **false** branch in a conditional statement is still *not* included. In the transfinite semantics **if false then** $x := 1$ **else** $x := 2$ **fi** is equivalent to $x := 2$, but the dataflow-based slicing algorithm will include the statement $x := 1$ if we are slicing on x. So the transfinite semantics does not capture the behaviour of dataflow-based slicing algorithms.

Nestra [Harmal Nestra 2006] claims that dataflow slices are "not correct w.r.t. standard semantics" because infinite loops may be sliced away. But this assumes that the semantic relation between a program and its slice has to be an equivalence relation. His "fractional semantics" has the same problems as other transfinite semantics.

G. Slicing Defined In Terms Of Semi-Refinement

To fix the problems caused by defining slicing in terms of refinement, we invented a new semantic relation: *semi-refinement* [21,23,24]. Program $\mathbf{S}_2$ is a semi-refinement of $\mathbf{S}_1$, written $\mathbf{S}_1 \preccurlyeq \mathbf{S}_2$, provided:

1) If $\mathbf{S}_1$ terminates on some initial state, then $\mathbf{S}_2$ also terminates with the same set of possible final states;
2) If $\mathbf{S}_1$ does not terminate, then $\mathbf{S}_2$ can do anything at all.

In terms of a denotational semantics which maps each initial state to the set of possible final states, if f_1 is the semantics for $\mathbf{S}_1$ and f_2 is the semantics for $\mathbf{S}_2$, then $\mathbf{S}_1 \preccurlyeq \mathbf{S}_2$ iff:

$$\forall s. (\perp \in f_1(s) \ \vee \ f_2(s) = f_1(s))$$

where $\perp$ is the special state denoting nontermination.

Semi-refinement can equivalently be defined in terms of weakest preconditions (see [23]).

For any program $\mathbf{S}$ and condition $\mathbf{R}$ on the final state, the *weakest precondition* WP$(\mathbf{S}, \mathbf{R})$ is the weakest condition on the initial state such that if $\mathbf{S}$ is started in a state satisfying this condition, then it is guaranteed to terminate in a state satisfying $\mathbf{R}$. Using weakest preconditions:

$$\mathbf{S}_1 \preccurlyeq \mathbf{S}_2 \quad \text{iff} \quad \mathbf{S}_1 \text{ is equivalent to } \{\text{WP}(\mathbf{S}_1, \textbf{true})\};\ \mathbf{S}_2$$

If $\mathbf{S}_1$ does not terminate, then $\text{WP}(\mathbf{S}_1, \textbf{true})$ is false and the assertion $\{\text{WP}(\mathbf{S}_1, \textbf{true})\}$ is **abort**. In this case, both sides of the equivalence are **abort**, and the equivalence is satisfied, whatever the semantics of $\mathbf{S}_2$. If $\mathbf{S}_1$ does terminate, then $\text{WP}(\mathbf{S}_1, \textbf{true})$ is **true** and the assertion is a **skip** statement. In this case, we must have $\mathbf{S}_1$ semantically equivalent to $\mathbf{S}_2$.

Slicing based on semi-refinement satisfies properties 1, 2, 3, 4, 5, 6, 9, 10, 11, 12, 13 and 14. Syntactic slicing, defined as program reduction plus semi-refinement, also satisfied Property 8. As we will see later, Property 7 (non-termination preservation) is incompatible with Property 11 (truncation) and therefore it is also incompatible with Property 10 (ditchability).

Table I summarises the results for a selection of the more important properties on all of the slicing relations we have considered:

Table I
PROPERTIES OF SLICING

	Behaviour	Termination	Non-Termination	Replacement	Truncation	Dependency	Pruned Dependency
Floor/Ceiling?	C	C	C	F	F	F	F
Weiser	✓	✓	×	×	✓	✓	✓
Reps & Yang	✓	✓	×	×	×	✓	×
Binkley & Gallagher	?[1]	✓	×	✓	✓	✓	✓
Refinement	×	✓	×	✓	✓	✓	✓
Kamkar Equivalence	✓	✓	✓	✓	×	✓	×
Lazy Equivalence	✓	×	×	?[2]	×	✓	×
Semi-Refinement	✓	✓	×	✓	✓	✓	✓

[1]Only for deterministic programs
[2]Depends on the precise semantics

IV. Characterisation of the Slicing Relation

In this section we will prove two theorems which show that properties 5 and 11 are sufficient to completely characterise a slicing relation: this relation turns out to be semi-refinement.

The first theorem shows that no slicing relation defined in terms of equivalence will satisfy Property 11 (truncation), other than the trivial relation in which every program is equivalent to every other program.

We will use very few properties of slicing and of the programming language and its semantics.

Definition 4.1: Given a semantic equivalence relation $\approx$ and a set X of variables in the final state space, the *restriction* of $\approx$ to X, denoted $\approx_X$ is the equivalence relation defined by taking all variables apart from those in X out of all the final states and comparing the result.

Note that if $\mathbf{S}_1 \approx \mathbf{S}_2$ then for any X, we have $\mathbf{S}_1 \approx_X \mathbf{S}_2$. Also, if $\mathbf{S}_1 \approx_X \mathbf{S}_2$ and X contains all the variables in $\mathbf{S}_1$ and $\mathbf{S}_2$, then $\mathbf{S}_1 \approx \mathbf{S}_2$.

The only property we need of the programming language and its semantics is the following:

Definition 4.2: A programming language and its semantics is *potentially divergent* if, for any given set X of variables of interest: there is a program $\textbf{abort}(X)$ which is locally x-preserving for all $x \in X$ and which does not terminate for any $x \in X$. The formal definition is that for any program $\mathbf{S}$:

$$\mathbf{S};\ \textbf{abort}(X)\ \approx_X\ \textbf{abort}(X)$$

This just means that a potentially divergent language is any language in which it is possible to write an infinite loop. For example, in WSL semantics we have **abort**, or $\{\textbf{false}\}$, or **while true do skip od**. These are all suitable candidates for the definition of $\textbf{abort}(X)$ for any set of variables X.

In the lazy semantics of [4,8] we can define $\textbf{abort}(X)$ as

$$\textbf{while true do } x_1 := x_1;\ x_2 := x_2;\ \ldots;\ x_n := x_n \textbf{ od}$$

which contains assignments for all the variables $x_1, \ldots, x_n$ in X, but is still locally x-preserving for all of these variables, since each variable is assigned its current value.

Definition 4.3: A semantic equivalence relation $\approx$ *partially defines* a slicing relation R if whenever $\mathbf{S}'$ is a slice of $\mathbf{S}$, according to R, on variable set X, then $\mathbf{S}$ and $\mathbf{S}'$ are equivalent according to $\approx_X$, ie:

$$\mathbf{S}\ R_X\ \mathbf{S}' \quad \Rightarrow \quad \mathbf{S}\ \approx_X\ \mathbf{S}'$$

Note that programs may be equivalent which are not slices of each other, but that every program is equivalent to all its slices (on the appropriate projection).

Definition 4.4: The *universal equivalence relation* is any semantic equivalence relation $\approx$ such that for all statements $\mathbf{S}_1$ and $\mathbf{S}_2$:

$$\mathbf{S}_1\ \approx\ \mathbf{S}_2$$

With these preliminaries out of the way, we can now present the main result of this section in the form of a theorem:

Theorem 4.5: Any equivalence relation $\approx$ which partially defines a truncating slicing relation R on a potentially divergent language and semantics, is the universal equivalence relation.

Proof: Let R be any truncating slicing relation on a potentially divergent language. Let $\approx$ be any semantic equivalence relation which partially defines R. Let $\mathbf{S}$ be any statement and X be any set of variables.

Consider the program: $\mathbf{S}$; $\textsf{\textbf{abort}}(X)$. By Truncation, $\mathbf{S}$ is a valid slice of $\mathbf{S}$; $\textsf{\textbf{abort}}(X)$, i.e:

$$\mathbf{S}\ \ R_X\ \ \mathbf{S};\ \textsf{\textbf{abort}}(X)$$

Since $\approx$ partially defines R, we have:

$$\mathbf{S}\ \approx_X\ \mathbf{S};\ \textsf{\textbf{abort}}(X)$$

By potential divergence we have:

$$\mathbf{S};\ \textsf{\textbf{abort}}(X)\ \approx_X\ \textsf{\textbf{abort}}(X)$$

Therefore, by the transitivity of $\approx_X$:

$$\mathbf{S}\ \approx_X\ \textsf{\textbf{abort}}(X)$$

But this holds for any statement $\mathbf{S}$. So for any statements $\mathbf{S}_1$ and $\mathbf{S}_2$:

$$\mathbf{S}_1\ \approx_X\ \textsf{\textbf{abort}}(X) \quad \text{and} \quad \mathbf{S}_2\ \approx_X\ \textsf{\textbf{abort}}(X)$$

So by the symmetry and transitivity of $\approx$:

$$\mathbf{S}_1\ \approx_X\ \mathbf{S}_2$$

But this is true for every X: in particular, for the set which contains all variables in $\mathbf{S}_1$ and $\mathbf{S}_2$. So:

$$\mathbf{S}_1\ \approx\ \mathbf{S}_2$$

But this is true for all statements $\mathbf{S}_1$ and $\mathbf{S}_2$, so $\approx$ is therefore the universal equivalence relation. ■

This means that any slicing relation which satisfies Property 11 (truncation), and which is defined in terms of semantic equivalence, must be the universal equivalence relation. In which case, it does not satisfy Property 5 (behaviour preservation).

In order to satisfy Property 5, the semantic slicing relation must be semantic equivalence (on the variables of interest) when the original program terminates. What slices are allowed when the original program does not terminate? The next theorem shows that if the slicing relation satisfies Property 11 (truncation), then it must allow *any* program as a valid slice of a non-terminating program.

Theorem 4.6: Any semantic relation $\preccurlyeq$ which partially defines a truncating slicing relation R on a potentially divergent language and semantics, is such that for all statements $\mathbf{S}$:

$$\textsf{\textbf{abort}}(X) \preccurlyeq_X \mathbf{S}$$

Proof: Let R be any truncating slicing relation on a potentially divergent language. Let $\approx$ be the semantic equivalence relation defined by the semantics (i.e. $\mathbf{S}_1 \approx \mathbf{S}_2$ if the semantics of $\mathbf{S}_1$ and $\mathbf{S}_2$ are identical). Let $\preccurlyeq$ be any semantic relation which partially defines R. Let $\mathbf{S}$ be any statement.

Consider the program $\mathbf{S}$; $\textsf{\textbf{abort}}(X)$. By truncation, $\mathbf{S}$ is a valid slice of $\mathbf{S}$; $\textsf{\textbf{abort}}(X)$, according to R. Since $\preccurlyeq$ partially defines R we have:

$$\mathbf{S};\ \textsf{\textbf{abort}}(X)\ \preccurlyeq_X\ \mathbf{S} \tag{1}$$

By potential divergence, we have:

$$\mathbf{S};\ \textsf{\textbf{abort}}(X)\ \approx\ \textsf{\textbf{abort}}(X) \tag{2}$$

Therefore, by substituting 2 into 1 we have:

$$\textsf{\textbf{abort}}(X)\ \preccurlyeq_X\ \mathbf{S}$$

So, since $\preccurlyeq$ partially defines R, we see that $\mathbf{S}$ is a valid slice of $\textsf{\textbf{abort}}(X)$. ■

Putting these results together, we see that *any* slicing relation which satisfies properties 5 and 11 must:

1) be semantic equivalence where the original program terminates, and;
2) allow any program as a slice when the original program does not terminate.

This is precisely the definition of semi-refinement! So there is only *one* slicing definition which satisfies these two fundamental properties of slicing. Fortunately, this definition also satisfies many other useful properties (see Section III-G). So we can say that these two properties completely characterise the slicing relation.

V. Slicing in FermaT

The FermaT transformation system includes three slicing algorithms:

1) Simple_Slice implements a simple slicing algorithm for structured programs. This algorithm has been formally proved correct by deriving the algorithm from a formal specification for slicing via a process of transformation and refinement.
2) Syntactic_Slice implements an interprocedural syntactic slicer which can handle unstructured code. The algorithm first analyses the program into "basic blocks". It then computes the Static Single Assignment (SSA) form of the program, and the control dependencies of each basic block using Bilardi and Pingali's algorithms [2,17]. FermaT tracks control and data dependencies to determine which statements can be deleted from the blocks. Tracking data dependencies is trivial when the program is in SSA form. FermaT links each basic block to the corresponding statement in the original program, so it can determine which statements from the original program have been deleted.
3) Semantic_Slice which implements a semantic slicer as a combination of transformations and syntactic slicing.

Semantic_Slice uses a number of transformations, including constant propagation, abstraction of non-iterative code to a specification statement, refinement of a specification statement, selective loop unrolling and so on. An example is the following:

$x := 3;$
$v := v + x;$
$\textsf{\textbf{var}}\ \langle x := y \rangle :$

$x := x + 1;$
$z_1 := x;$
$x := z_2 + x;$
$v := v + x;$
$y := 0$ **end**;
$q := v$

Slicing on the final value of q gives this result:

$q := v + y + z_2 + 4$

An example involving selective unrolling is the following program [1,22]:

while $p?(i)$ **do**
  if $q?(c)$
    then $x := f;\ c := g$ **fi**;
  $i := h(i)$ **od**

where we are slicing on the final value of x.

Here, the normal dependency tracking slicing algorithms will assume that the assignment $c := g$ is needed: even though this assignment has no effect on x. So these algorithms will include the whole program. By using selective unrolling, FermaT's semantic slicer is able to generate a much more concise result:

if $p?(i) \wedge q?(c)$ **then** $x := f$ **fi**

VI. Conclusions

This paper is not intended to be a comprehensive classification and analysis of all possible properties and all published slicing definitions. Rather it is a starting point and springboard for further research. The main result of this paper is the fact that two of the most basic and fundamental properties of slicing, together with a basic property of the programming language semantics, are sufficient to completely characterise the semantics of the slicing relation: in the sense that there is only one possible slicing relation which meets all the requirements.

Another way of looking at this is to say that the "ceiling" property of behaviour-preservation and the "floor" property of truncation are identical: there is no room to fit more than one semantic relation (semi-refinement) in between these two properties. (Recall that any relation R satisfies ceiling property C iff $R \subseteq C$, while R satisfies floor property F iff $F \subseteq R$. If $F = C$ then we can only have $F \subseteq R \subseteq C$ when $R = F = C$). By itself, semi-refinement defines Semantic Slicing [23]. If we add the syntactic relation of reduction (Property 8) then we get Syntactic Slicing. In [23] we show that Conditioned Slicing, Dynamic Slicing and many other variants can also be defined in terms of semi-refinement.

Acknowledgements

The research described in this paper was inspired by an extended email discussion with Dave Binkley, Mark Harman and Sebastian Danicic. Dave Binkley coined the term "ditchability" to describe the property of being able to delete any code which does not (directly or indirectly) affect the value of any variable of interest.

All the opinions are, of course, my own.

References

[1] Anon, "Which Lines do not affect x?," *Ceramic Mug given to attendees of the First Source Code Analysis and Manipulation Workshop, Florence, Italy, 10th November* (2001).

[2] Gianfranco Bilardi & Keshav Pingali, "The Static Single Assignment Form and its Computation," Cornell University Technical Report, July, 1999, ⟨http://www.cs.cornell.edu/Info/Projects/Bernoulli/papers/ssa.ps⟩.

[3] Dave Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss & Bogden Korel, "A Formalisation of the Relationship between Forms of Program Slicing," *Science of Computer Programming* 62 (2006), 228–252.

[4] Dave Binkley, Mark Harman & Sebastian Danicic, "Amorphous Program Slicing," *Journal of Systems and Software* 68 (Oct., 2003), 45–64.

[5] David W. Binkley & Keith B. Gallagher, "A Survey of Program Slicing," *Advances in Computers* 43 (1996), 1–52.

[6] Robin Cartwright & Matthias Felleisen, "The Semantics of Program Dependence," *Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation* 24 (1989), 13–27, Publishes as SIGPLAN Notices.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman & F. K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependance Graph," *Trans. Programming Lang. and Syst.* 13 (July, 1991), 451–490.

[8] Sebastian Danicic, "Dataflow Minimal Slicing," London University, PhD Thesis, 1999.

[9] Sebastian Danicic, Mark Harman, John Howroyd & Lahcen Ouarbya, "A Non-Standard Semantics for Program Slicing and Dependence Analysis," *Logic and Algebraic Programming, Special Issue on Theory and Foundations of Programming Language* 72 (2007), 123–240.

[10] Roberto Giacobazzi & Isabella Mastroeni, "Non-Standard Semantics for Program Slicing," *Higher-Order and Symbolic Computation* 16 (Dec., 2003), 297–339.

[11] M. Harman, M. Munro, D. Binkley, S. Danicic, M. Aoudi & L. Ouarbya, "Syntax-Directed Amorphous Slicing," *Automated Software Engineering (ASE)* 11 (2004), 27–61.

[12] Mark Harman & Sebastian Danicic, "Amorphous Program Slicing," *5th IEEE International Workshop on Program Comprehesion (IWPC'97), Dearborn, Michigan, USA* (May 1997).

[13] Mark Harman, Sebastian Danicic, Yoga Sivagurunathan & Dan Simpson, "The Next 700 Slicing Criteria," *Second UK Workshop on Program Comprehension* (1996).

[14] Susan Horwitz, Thomas Reps & David Binkley, "Interprocedural slicing using dependence graphs," *Trans. Programming Lang. and Syst.* 12 (Jan., 1990), 26–60.

[15] Mariam Kamkar, "Interprocedural Dynamic Slicing with Applications to Debugging and Testing," Linkoping University, PhD Thesis, S-581 83 Linkoping, Sweden, 1993.

[16] Donglin Liang & Mary Jean Harrold, "Slicing Objects Using System Dependence Graph," *International Conference on Software Maintenance (ICSM), Washington DC, USA* (Nov., 1998).

[17] Keshav Pingali & Gianfranco Bilardi, "Optimal Control Dependence Computation and the Roman Chariots Problem," *Trans. Programming Lang. and Syst.* (May, 1997), ⟨http://www.cs.cornell.edu/Info/Projects/Bernoulli/papers/toplas97.ps⟩.

[18] Thomas Reps, "Algebraic properties of program integration," *Science of Computer Programming*, 17 (1991), 139–215.

[19] Thomas Reps & Wuu Yang, "The Semantics of Program Slicing," *Computer Sciences Technical Report* 777 (June, 1988).

[20] M. Ward, "The Formal Transformation Approach to Source Code Analysis and Manipulation," *IEEE International Workshop on Source Code Analysis and Manipulation Florence, Italy, 10th November*, Los Alamitos, California, USA (2001).

[21] M. P. Ward, H. Zedan & T. Hardcastle, "Conditioned Semantic Slicing via Abstraction and Refinement in FermaT," *9th European Conference on Software Maintenance and Reengineering (CSMR) Manchester, UK, March 21–23* (2005).

[22] Martin Ward, "Slicing the SCAM Mug: A Case Study in Semantic Slicing," *Third IEEE International Workshop on Source Code Analysis and Manipulation 26th–27th September*, Los Alamitos, California, USA (2003).

[23] Martin Ward & Hussein Zedan, "Slicing as a Program Transformation," *Trans. Programming Lang. and Syst.* 29 (Apr., 2007), 1–52, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/slicing-t.ps.gz⟩.

[24] Martin Ward, Hussein Zedan, Matthias Ladkau & Stefan Natelberg, "Conditioned Semantic Slicing for Abstraction; Industrial Experiment," *Software Practice and Experience* 38 (Oct., 2008), 1273–1304, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/slicing-paper-final.pdf⟩.

[25] M. Weiser, "Program slicing," *IEEE Trans. Software Eng.* 10 (July, 1984), 352–357.

[26] M. Weiser, "Programmers use slices when debugging," *Comm. ACM* 25 (July, 1984), 352–357.

[27] Mark Weiser, "Program slices: formal, psychological, and practical investigations of an automatic program abstraction method," University of Michigan, PhD Thesis, Ann Arbor, 1979.