An Overview of Program Slicing

Mark Harman and Robert M. Hierons

Department of Information Systems and Computing,
Brunel University,
Uxbridge, Middlesex, UB8 3PH.

Phone: +44 (0)1895 816 237
FAX: +44 (0)1895 251 686
e−mail: mark.harman@brunel.ac.uk
web: http://www.brunel.ac.uk/~csstmmh2/

1. Introduction

Program slicing is a technique for simplifying programs by focusing on selected aspects of semantics. The process of slicing deletes those parts of the program which can be determined to have no effect upon the semantics of interest. Slicing has applications in testing and debugging, re−engineering, program comprehension and software measurement. For example, in debugging, there is little point in the (human) debugger analysing sections of the source code which cannot have caused the bug. Slicing avoids this by removing these parts of the program, focusing attention on those parts of the program which may contain a fault.

This article reviews three semantic paradigms for slicing: static, dynamic and conditioned and two syntactic paradigms: syntax−preserving and amorphous. Slicing has been applied to many software development problems including testing, reuse, maintenance and evolution. This paper describes the main forms of program slice and some of the applications to which slicing has been put.

There are two dimensions to slicing: a syntactic dimension and a semantic dimension. The semantic dimension describes that which is to be preserved. The program's static behaviour is preserved by static slicing criteria and its dynamic bevahiour is preserved by dynamic criteria. Sections 2 and 3 describe the differences between these semantic paradigms in more detail. Section 4, describes a form of slicing, conditioned slicing, which bridges the gap between static and dynamic.

In the syntactic dimension there is comparatively less choice. There are two possibilities. Firstly, the slice can preserve the program's original syntax wherever possible, merely removing parts of the program which can be found to have no effect on the semantics of interest. This is known as a syntax−observing slice. Secondly, the slice can be free to perform any syntactic transformation which preserves the semantic constraints desired. This is known as amorphous slicing. Most work on slicing has considered the syntax−preserving form, and sections 2,3 and 4 all describe the semantics of slicing with respect to the syntax−preserving criterion. Section 5 relaxes this restriction to syntax−preservation, considering the amorphous variation of program slicing.

Slicing has many application areas. Essentially, any area of software engineering and development in which it is helpful to extract subprograms based upon arbitrary semantic criteria are potential applications of slicing. Section 6 describes four application areas: Software Measurement (specifically cohesion measurement), debugging, program comprehension and software maintenance.

Slicing is an evolving field. Section 7 considers future directions for the ongoing slicing research programme. Naturally, this section is a personal selection of research chosen by the authors. It does not aim to answer the question "what work is currently taking planing in program slicing research". Rather, it addresses the authors' selection of current research trends.

This is an introductory article, and presents only an overview of slicing paradigms and applications. More detailed papers are available in the wider literature. Section 8 contains some pointers to the literature to enable the reader to follow up the topics which this article is able only to cover in overview or which it is not possible to cover in the space available. Section 9 concludes.

2. Static Slicing

There are many forms of slice, so it will be helpful to start off with a simple form of slice; the static slice. The other forms of slice can be thought of as augmentations of this static form.

A slice [WEI84,HRB90] is constructed by deleting those parts of the program that are irrelevant to the values stored in the chosen set of variables at the chosen point. The point of interest is usually identified by annotating the program with line numbers which identify each primitive statement and each branch node.

The point of interest will be indicated by adding a comment to the program. In general slices are constructed for a set of variables, but in this article only slices constructed for a single variable will be considered. Thus, given a variable *v* and a point of interest *n*, a slice will be constructed for *v* at *n*. This is not restrictive, because the slice with respect to a set of variables *V* can be formed from the union of the slices on each variable in V.

Having picked a slicing criterion, we can construct one of two forms of slice: a backward slice or a forward slice. A backward slice contains the statements of the program which can have some *effect on* the slicing criterion, whereas a forward slice contains those statements of the program which are *affected by* the slicing criterion. Hereinafter, only backward slicing will be considered. However, the reader should be aware that all the forms of slices which will be discussed have 'forward' counterparts.

To see how slicing works, consider the simple example fragment of C code in Figure 1. Suppose we only care about the effect this fragment of code has on the variable `z` at the end of the program. We can construct a backward slice on `z` at the end of the program to focus attention on this aspect of the fragment. The slice of Figure 1 on `z` at the end of the program is shown in Figure 2. It is easy to see that the line `r = x;` can not affect the final value of `z` and this is why it is not included in the slice. Slightly less obvious is the fact that the assignment `z = y-2;` also has no effect upon the final value of `z`, and so it too is not included in the slice. This is because, although this statement updates the value of `z` in the middle of the program, the value assigned is overridden by the last line, thereby losing the effect of the first assignment.

Consider the program fragment in Figure 3. This program is intended to compute the sum and the produce of numbers from 1 to some input *n*. The sum is stored in the variable `s` and the product in the variable `p`. The computation in `s` is correct, but that on `p` is incorrect. Since the computation of `s` is correct there is no point looking at computation which only affects this variable, when trying to track the fault which causes the erroneous computation on `p`. Slicing on `p` is a natural starting point for debugging, because it removes computations which can be shown (statically) to leave `p` unaffected.

The static slice on the final value of the variable `p` is shown in Figure 4. As the slice is simpler than the original program in Figure 3, yet contains all the statements which could possibly affect the final (and incorrect) value of the variable `p`, examining the slice will allow us to find the bug faster than examining the original. In this case, the bug lies in the initialisation of the variable `p`. The variable should be initialised to 1 and not to 0.

Although static slicing can assist the debugging effort by simplifying the program under consideration, the slices constructed by static slicing tend to be rather large. This is particularly true for well−constructed programs, which are typically highly cohesive. This high level of cohesion results in programs where the computation of the value of each variable is highly dependent upon the values of many other variables.

3. Dynamic Slicing

When we debug a program, we will usually have executed it, and presumably found that it produced an unexpected value. For example, the program in Figure 3 might have been executed with the input value 0 for the variable `n`. (This would be a highly likely choice, as one tends to test loops with 'special case' values, such as 0.) Now, when we execute the program in Figure 3 with the value 0 we will find that the variable `p` contains the wrong value. Instead of consulting the static slice to locate the cause of this bug, it would make more sense to construct a slice which *exploited* the information available about the input which caused the program to go wrong. Such a slice is called a dynamic slice [KR98], because it is constructed with respect to the traditional static slicing criterion *together* with dynamic information (the input sequence supplied to the program, during some specific execution).

We construct a dynamic slice with respect to three pieces of information. Two of these − the variable whose value appears to be wrong and the point of interest within the program − are just the same as we find in static slicing. The third is the sequence of input values for which the program was executed. Collectively, this information is called the 'dynamic slicing criterion'. We shall say that we construct a dynamic slice *for* a variable *v*, *at* a point *n*, *on* an input *i*. To describe the input sequence, *i*, we shall enclose the sequence of values in angled brackets. Thus <1,4,6> represents a sequence of

three input values the first of which is 1, the second of which is 4 and the last of which is 6. In the case of the example in Figure 3, we shall construct a dynamic slice for the variable `p` at the end of the program on the input sequence <0>. The dynamic slice constructed with respect to this criterion is shown in Figure 5. It is far simpler than the corresponding static slice (Figure 4) and it clearly highlights the bug in the original program.

Obviously, we were rather lucky when we constructed the dynamic slice for the example program in Figure 3; we achieved the maximum possible level of simplification. The example was chosen to highlight the advantage of dynamic slicing. In practice we shall probably be less lucky. Of course, if we have a number of test cases, all of which cause some failure, then there's nothing to stop us having the slicing tool choose which case we should deal with by constructing a dynamic slice for each and selecting the smallest.

3.1 Is Dynamic Slicing Always Better Than Static Slicing?

The first part of this article has shown that dynamic slices are very attractive as an aid to debugging, and in particular, that they are superior to static slices for this application. The reader would be forgiven for concluding that static slicing is rendered obsolete by dynamic slicing. However, we *do* still require static slicing for some applications where the slice *has* to be sound for *every* possible execution.

For example, suppose we are interested in reusing the part of a program which implements a particularly efficient and well−tested approach to some problem. Often, in such situations (particularly with legacy code), the code we want to reuse will be intermingled with all sorts of other unrelated code which we do not want. In this situation static slicing is ideal as a technique for extracting the part of the program we require, while leaving behind the part of the program we are not interested in.

This observation highlights the trade−off between the static and dynamic paradigms. Static slices will typically be larger, but will cater for every possible execution of the original program. Dynamic slices will typically be much smaller, but they will only cater for a single input.

4. Conditioned Slicing

The static and dynamic paradigms represent two extremes − either we say *nothing* about the input to the program (static slicing) or we say *everything* (dynamic slicing). Conditioned slicing allows us to bridge the gap.

Fortunately, we can provide information to the slicing tool about the input *without* being so specific as to give the precise values. Consider the program fragment in Figure 6. We can use a boolean expression, for example, `x==y+4`, to relate the possible values of the two inputs `x` and `y`. When the program is executed in a state which satisfies this boolean condition, we know that the assignment `z = 2;` will not be executed. Any slice constructed with respect to this condition may therefore omit that statement. This approach to slicing is called the 'conditioned approach' [CCD98,DF+00], because the slice is conditioned by knowledge about the condition in which the program is to be executed.

Conditioned slicing addresses just the kind of problems maintainers face when presented with the task of understanding large legacy systems. Often, in this situation, we find ourselves asking questions like

> 'suppose we know that `x` is greater than `y` and that `z` is equal to 4, then which statements would affect the value of the variable `v` at line 38 in the program'.

Using conditioned slicing, we can obtain an answer to this question automatically. The slice would be constructed for `v`, at 38, on `x>y && z==4`. By building up a collage of conditioned slices which isolate different aspects of the program's behaviour, we can quickly obtain a picture of how the program behaves under various conditions. Conditioned slicing is really a tool−assisted form of the familiar approach to program comprehension of divide and conquer.

5. Amorphous Slicing

All approaches to slicing discussed so far have been 'syntax preserving'. That is, they are constructed by the sole transformation of statement deletion. The statements which remain in the slice are therefore a syntactic subset of the original program from which the slice was constructed. By contrast, amorphous slices [HD97,BH+00] are constructed using *any* program transformation which simplifies the program and which preserves the effect of the program with respect to the slicing criterion.

This syntactic freedom allows amorphous slicing to perform greater simplification with the result that amorphous slices are never larger than their syntax−preserving counterparts. Often they are considerably smaller.

Amorphous slicing allows a slicing tool to 'wring out' the semantics of interest, aiding program comprehension, analysis and reuse. For example, consider the program fragment in Figure 7. This program fragment is intended to compute the biggest element of the array `a` in the variable `biggest`. In fact, it does this correctly. However, the syntax preserving slice does not make this very clear. This is because all that can be achieved by statement deletion is the removal of the final statement. By contrast, amorphous slicing on the final value for the variable `biggest` transforms the program to

```
for(i=1, biggest=a[0]; i<20; ++i)
   if (a[i] > biggest) biggest = a[i];
```

From this amorphous slice, it is far clearer that the computation of the original fragment on **`biggest`** is, indeed, correct. Notice how amorphous slicing retains the semantic guarantee that the program and slice behave identically with respect to the slicing criterion. Only the syntactic properties of slicing differ between traditional (syntax preserving) and amorphous slicing.

Consider once more, the code fragment in Figure 7. The variable `average` is supposed to contain the average value of the elements of the array `a`. Suppose a slice is constructed for the variable `average`. The syntax preserving slice for this variable removes the conditional statement which forms the middle two lines of the fragment. This helps comprehension somewhat. However, the amorphous slice, improves upon this simplification. It yields the slice :

```
average=a[19]/20;
```

This amorphous slice clearly indicates that there is a problem with the original program's computation on `average`. That is, we would expect this computation to involve looping of some form. The fact that amorphous slicing has been able to remove the loop is a clear sign that the original program does not perform correctly for the variable `average`.

6. Applications of Program Slicing

This section describes four of the applications to which program slicing has been put. The section starts with a discussion of debugging, which was the original motivation for program slicing. Since then slicing has been applied to many other problem areas. To give a flavour for the breadth of applications three additional topics are described: cohesion measurement, comprehension and maintenance. The first of these is covered in detail, while the latter two are presented in overview.

6.1 Debugging

The original motivation for program slicing was to aid the location of faults during debugging activities. The idea was that the slice would contain the fault, but would not contain lines of code which could not have caused the failure observed. This is achieved by setting the slicing criterion to be the variable for which an incorrect value is observed. Weiser [Wei79,Wei82, LW86] investigated the ways in which programmers mentally slice a program when attempting to understand and debug it, and this formed an original motivation for the consideration of techniques for automated slicing.

Clearly this approach will only work where the fault is an 'error of commission' rather than an error of omission. Slicing cannot be used to identify bugs such as 'missing initialisation of variable'. If the original program does not contain a line of code then the slice will not contain it either. Although slicing cannot identify errors of omission, it has been argued that slicing can be used to aid the detection of such errors [HD97].

The application of debugging also motivated the introduction of dynamic slicing. The observation here was that slices were typically constructed once a program had produced a failure. In such a situation, the input to the program that caused the fault is available and can be used to narrow the search for the fault, by including only those statement which could have caused this fault to have occurred on the particular execution of interest.

Dynamic slicing was also an important step forward, because it had been observed that static slices for many interesting slicing criteria tended to be rather large. The observation that static slices tended to be large also stimulated interest in slicing as a means of measuring program cohesion. That is, in a cohesive program it should be relatively hard to slice the program up into separate components. Bieman and Ott investigated the hypothesis in detail, leading to several source level software metrics for cohesion measurement, which are described in the next section.

6.2 Cohesion Measurement

This section presents a brief overview of the work of Ott and Bieman [OB98] on the application of syntax−preserving static program slicing to the problem of measuring programs for cohesiveness.

A cohesive program is one in which the modularisation of the program is performed 'correctly'. More precisely, a cohesive function or procedure should perform related tasks. A cohesive function might calculate the result and remainder arising from the division of one number by another. A less cohesive function would be one which returns the largest of two numbers together with the product of the two numbers. In the first case, the two actions are related, whereas in the second they are not.

The motivation for assessing the cohesiveness of a program or a part of it rests upon observations and claims that highly cohesive programs are easier to maintain, modify and reuse. Cohesion was a product of the effort to define principles for programming, turning the activity from a craft into an engineering discipline. A modern development on the 'cohesion theme' is the idea of encapsulation in object−oriented programming. A well encapsulated object contains all the necessary data and function members associated with the object.

A slice captures a thread through a program, which is concerned with the computation of some variable. The idea of functional cohesion is that a function should perform related tasks. If we took several slices from a function, each for a different variable, and we found that these slices had a lot of code in common, then we would be justified in thinking that the variables were related in some way. We might go further, and decide that the function's 'tasks' are captured by the computation it performs on these variables, and therefore we would conclude that the function's tasks were strongly related and that the function was thus highly cohesive.

We shall base the measurement of functional cohesion on just this kind of reasoning.

6.2.1 Processing elements and output variables

The first step we need to take is to decide what the processing elements of a function are, so that we can decide whether or not these processing elements are related. We shall take the view that a processing element is a piece of code which computes a value. This value must be visible outside the function for it to constitute a 'result'. Such values are those

- printed by the function
- stored in a global variable
- stored in a reference parameter passed to the function

Each of these three forms of calculation can be thought of as the calculation of a value for a variable. We shall therefore focus our attention on the values printed out by a function. To simplify matters further, we shall assume that the values are printed out at the end of the function, and that each output consists of printing the value of one of the function's local variables. Relaxing these simplifying assumptions does make the calculation of cohesion slightly more involved, but does not render the approach we shall describe inapplicable.

As a processing element is to be the code within a function which contributes to the calculation of an output, we can isolate this processing element by slicing the program for the variable whose value is output. Consider, for example, the function depicted in Figure 8. This function outputs the value of three variables and therefore has three processing elements —one for each variable. Each of these processing elements can be isolated by slicing. Figures 9,10 and 11 show the processing elements associated with the computation of the variables `Count`, `Pass` and `Fail` respectively.

6.2.2 The Cohesive section

There is very little overlap between these three processing elements. This is because the calculation of the number of passes, fails and the count are largely independent of one another. The cohesive section of a function is made up of the statements which are in all the function's processing elements. In the case of the function `Marks` in Figure 8, the cohesive section is the code in the slices for all

three variables `Pass`, `Fail` and `Count`. This can be visualised by placing markers by each line of the program, indicating which of the three slices the line belongs to. This has been done in Figure 12. The marker `P` indicates the lines in the slice for `Pass`, `F` for `Fail` and `C` for `Count`. Only two of the ten lines of code in the function find their way into all three slices. This gives us a crude measure of the function's cohesion as 2/10 or 0.2.

Consider the function `MinMax` in Figure 13. This function has two processing elements, as it outputs the value of two variables. Two slices are constructed, one for each of these two variables, with the results being depicted using labelling. The label `L` indicates lines which are in the slice for `Largest`. The label `S` indicates lines which are in the slice for `Smallest`. The function `MinMax` is measured as being more cohesive than the function in Figure 8, as comparatively more lines find their way into all processing elements of the function; six of the ten lines of the function are in both processing elements, giving a measure of 6/10 or 0.6.

The measure of cohesion we are using is the number of statements in the function's cohesive section relative to the number of statements in the function as a whole.

6.3 Comprehension

It is widely believed that the maintenance and evolution phases of the software development lifecycle require about 70% of the total costs and so tools which speed up this part of the process are important keys to improved productivity. The maintenance phase often starts with program comprehension. This is particularly true where legacy systems are concerned, where documentation may be sparse and the original developers may no longer be available.

Slicing can help with the comprehension phase of maintenance. De Lucia, Fasolino and Munro [DFM96] use conditioned slicing in order to facilitate program comprehension. Field, Ramalingham and Tip [FRT96] independently introduced a similar technique called constrained slicing. Both share the property that a slice is constructed with respect to a condition in addition to the traditional static slicing criterion. The condition can be used to identify the cases of interest in slicing. Harman and Danicic [HD97] showed how this notion of a conditioned slice could be combined with the notion of an amorphous slice to further assist program comprehension. Essentially a program is understood in terms of a set of cases, each of which is captured by a condition. The set of conditions is used to construct a set of cases.

For example, consider the program fragment from the Kernighan and Ritchie `C` Programming book [KR88] in Figure 14. This program converts a string in `s` into an integer in `r`. It would be natural to ask what the effect of this code is when the string in question starts with a `'+'` character. This question can be investigated using amorphous conditioned slicing. The slicing criterion has the condition `s[0]=='+'` and the variable of interest is `r`. The slice is shown in Figure 15. Notice that this is the cliché code which converts a sequence of digits into an integer.

The slice for the condition `s[0]=='-'` where the variable of interest is `r` is shown in Figure 16. This looks almost identical to the slice in Figure 15, except that the line `r=n;` has become `r=-n;`, clearly indicating that the code performs as expected for strings starting with a minus sign.

Finally, the programmer may wonder what the effect of the code is when the string is empty. The slice for this case is shown in Figure 17. Notice how this slice immediately informs the programmer that the behviour of the original program is correct. In this case, the conditioned slice is effectively a dynamic slice, because the slicing criterion has the condition `s==""`, which is only satisfied by a single initial state.

In this manner, conditioned slicing combined with amorphous slicing, provides the programmer with a convenient tool for program comprehension and analysis. The conditions can be used to capture cases of interest allowing the program to be broken up into fragment each of which are relevant to a particular form of computation. The amorphous slicing with respect to the condition, removes parts of the program which are irrelevant, focusing attention on the conditions of interest − a software focus.

6.4 Maintenance and Re−engineering

Software maintenance is often followed by re−engineering effort, whereby a system is manipulated to improve it. Canfora at al. [CCD94] introduced conditioned slicing in order to allow conditions to be used to extract code based on conditions. The idea is to isolate desired functionality so that it can

be ‘salvaged’ and reused. This application of conditioned slicing to reuse is further developed by Cimitile, De Lucia and Munro [CDM96] as part of the RE−squared reverse engineering project.

Gallagher and Lyle [GL91] show how a decomposition slice can be built for each variable of a program, The decomposition slices are used to divide up the software with a view to limiting the impact of software changes during maintenance activities. The decomposition slice has a complement. The complement contains those parts of the software which can be altered without affecting computation of the variable for which the decomposition slice is constructed. This allows the programmer to perform software surgery [Gal90], altering the software, without causing unintentional changes due to the ripple effects of the changes.

Lakhotia and Deprez [LD98] introduced a program tucking transformation, which uses slicing and other code isolation techniques to extract code components with the aim of improving the cohesion and coupling of the program. A software ‘wedge‘ is driven into the code to bound the area of interest. Statements which concern a particular computation are sliced out of a procedure and folded into a new procedure. The sequence of transformations wedge−slice−fold construct a ‘tuck‘ transformation.

7. Directions for Future Work

This paper has described several approaches to program slicing. Work on amorphous and conditioned slicing is comparatively less mature than work on static and dynamic syntax preserving slicing. Current implementations of static slicing such as Unravel [Unr] and the Wisconsin Program Slicing System [WPS] can handle a very large subset of the C programming language and produce slices in reasonable time. By comparison, amorphous and conditioned slicers have only been defined for toy languages. More work is required to find good algorithms for these newer forms of slicing. There is also a possibility that they can be combined to produce, for example, amorphous conditioned slices.

The authors are also interested in exploring the connections between slicing, transformation and testing. Initial work in this area has shown that amorphous static slicing forms a good support to detection of equivalent mutants in mutation testing [HHD99], that conditioned slicing can complement partition−base testing [HH00] and that slicing and related dependence analyses can be use to support mutation testing [HHD00].

8. Follow up Reading on Program Slicing

This section provides some pointers to the literature.

Tip [Tip95] and Binkley and Gallagher [BG96] provide detailed surveys of program slicing. A more recent overview survey has also been produced by De Lucia [DeL01]. The special Issue of Information and Software Technology on Program Slicing [IST98] also contains an overview of the field and a set of papers which cover applications, theory and techniques.

The original slicing algorithms used an iterative approach, where the slice is computed as a solution to a set of data flow equations [Wei84,DHS95]. More recent work has tended to use the System Dependence Graph [HRB90].

Semantic issues and the theory of slicing are addressed in Weiser’s thesis [Wei79] and papers by Cartwright and Felleisen [CF89], Venkatesh [Ven91] and Harman, Danicic and Simpson [HDS94].

Two variations on the slicing theme, which are closely related to slicing are Chopping [JR94] and Dicing [LW86,CC93]. In Chopping dependence chains from a source (of the dependence) to a sink (of the dependence) are identified. In dicing, the slice (for incorrect bevahiour) is further reduced by considering traces of the program for which behaviour is correct.

Unravel [Unr] is a freely available slicing tool which runs under a UNIX/Linux environment, and which slices are large subset of the C programming language. The Wisconsin Program Slicing System [WPS] is a commercially available tool for slicing C programs and is marketed by GrammaTech Inc.

The idea of program slicing has also been applied to Object−Oriented programs [Tip96,LH96,Ste98], logic programs [SD96,KNN99,ZCU97], functional programs [JH99] and specifications [WA98].

9. Summary

Slicing is a simplification process defined with respect to a slicing criterion. The idea behind all approaches to program slicing is to produce the simplest program possible that maintains the meaning of the original program with regard to this slicing criterion.

This paper has introduced three forms of slicing criteria: static, dynamic and conditioned. The conditioned criterion is the most general of these, subsuming both static and dynamic criteria as special cases. The conditioned criterion consists of a set of variables, a program point of interest and a condition.

Traditional syntax preserving slices are constructed by statement deletion. Statements of the program are deleted if they are irrelevant when attention is focused on the variables of interest at the point of interest and in states which satisfy this condition. The slice thus has the property that it can be executed and behaves identically to the original with respect to the slicing criterion.

In static slicing, the condition is simply 'true', so the condition part of the criterion effectively plays no role in the simplification process. In dynamic slicing the condition is a conjunction of equalities, which defines values for each input to the program. Therefore, using the dynamic criterion, the condition can be thought of as a complete specification of the input sequence to the program. Static slices tend to be thicker but more general than their dynamic counterparts. Conditioned slicing provides a convenient bridge between the two poles of static and dynamic slicing.

For any slicing criterion, the slice may be constructed in a forward or backward direction. A forward slice is constructed by deleting statements which cannot be *affected by* the slicing criterion, whereas a backward slice is constructed by removing those parts of the program which have no *effect upon* the slicing criterion.

Traditional slicing is syntax preserving because statements are deleted, but not altered. The statements which remain in the slice are therefore a syntactic subset of the original program from which the slice was constructed. By contrast, amorphous slices are constructed using any program transformation which simplifies the program and which preserves the effect of the program with respect to the slicing criterion. Amorphous slices are not syntax preserving. However, the removal of syntactic restrictions means that they are never larger, and often smaller, than their corresponding syntax preserving counterparts.

References

BH+00 D. W. Binkley, M. Harman, L. R. Raszewski and C. Smith.
An empirical study of amorphous slicing as a program comprehension support tool.
8th IEEE International Workshop on Program Comprehension (IWPC 2000),
Limerick, Ireland, June, 2000. Pages 161−170.

BG96 D. W. Binkley and K. B. Gallagher,
Program Slicing.
In Advances in Computers, Volume 43, 1996.
Marvin Zelkowitz, Editor, Academic Press San Diego, CA.

CCD94 G. Canfora, A. Cimitile, A. De Lucia and G. A. Di Lucca.
Software Salvaging Based on Conditions.
IEEE International Conference on Software Maintenance (ICSM 1994),
Victoria, Canada, September 1994, pages 424−433.

CCD98 G. Canfora, A. Cimitile and A. De Lucia.
Conditioned program slicing.
Information and Software Technology special issue on Program Slicing,
40(11−12), pages 595−607, December 1998.

CF89 R. Cartwright and M. Felleisen
The Semantics of Program Dependence.
SIGPLAN Notices, 24(7), pp. 13−27, July 1989.

CC93 T. Y. Chen and Y. Y. Cheung
Dynamic Program Dicing.
IEEE Conference on Software Maintenance (ICSM 1993), pp. 378−385, 1993.

CDM96 A. Cimitile, A. De Lucia and M. Munro.
A Specification Driven Slicing Process for Identifying Reusable Functions.
Software Maintenance: Research and Practice,
8, pages 145−178, 1996.

DHS95 S. Danicic, M. Harman and Y. Sivagurunathan
A parallel algorithm for static program slicing.
Information Processing Letters, 56(6), pp. 307−313, December 1995.

DF+00 S. Danicic, C. J. Fox, M. Harman and R. M. Hierons.
ConSIT: A Conditioned Program Slicer.
IEEE International Conference on Software Maintenance (ICSM 2000),
San Jose, California, USA, October, 2000. pages 216−226.

Del01 A. DeLucia

Program Slicing: Methods and Applications.
Invited paper,
IEEE workshop on Source Code Analysis and Manipulation (SCAM 2001),
Florence, Italy, 10th November 2001, to appear.

FRT96 J. Field and G. Ramalingam and F. Tip.
Parametric Program Slicing
22nd Symposium on Principles of Programming Languages (POPL'95),
January 22−25, 1995, pages 379−392.

Gal90 K. B. Gallagher.
Using Program Slicing in Software Maintenance.
Ph.D. Thesis, University of Maryland Baltimore County, 1990.

GL91 K. B. Gallagher and J. R. Lyle.
Using Program Slicing in Software Maintenance.
IEEE Transactions on Software Engineering,
17(8), pages 751−761, August 1991.

HDS96 M. Harman, D. Simpson and S. Danicic
Slicing Programs in the Presence of Errors.
Formal Aspects of Computing, 8(4), 1996.

HD97 M. Harman and S. Danicic.
Amorphous Program Slicing.
IEEE International Workshop on Program Comprehension (IWPC'97),
Dearborn, Michigan, May 1997, pages 70−79.

HRB90 S. Horwitz, T. Reps, and D. W. Binkley.
Interprocedural slicing using dependence graphs.
ACM Transactions on Programming Languages and Systems,
12(1) pages 35−46, January 1990.

HHD99 R. Hierons, M. Harman and S. Danicic.
Using Program Slicing to Assist in the Detection of Equivalent Mutants
Journal of Software Testing, Verification and Reliability,
9(4), pages 233−262, December 1999.

HH00 R. Hierons and M. Harman.
Program Analysis and Test Hypotheses Complement.
IEEE ICSE International Workshop on Automated Program Analysis,
Testing and Verification, Limerick, Ireland, June 4−5, 2000.

HHD00 M. Harman, R. Hierons and S. Danicic.

The Relationship Between Program Dependence and Mutation Analysis.
Mutation 2000 Workshop,
San Jose, California, USA, October 6th−7th, 2000. pages 15−23.

IST98 Information and Software Technology, Special Issue on Program Slicing.
Journal of Information and Software Technology, 40(11&12), November/December 1998.

JR94 D. Jackson and E. J. Rollins
Chopping: A Generalization of Slicing
Technical Report, Carnegie Mellon University, School of Computer Science,
Number CS−94−169, p. 21, July 1994.

JH99 J. Ahn and T. Han,
Static Slicing of a First−Order Functional Language based on Operational Semantics,
Korea Advanced Institute of Science & Technology (KAIST)
Technical Report CS/TR−99−144, Dec., 1999.

KNN99 G. Kókai and J. Nilson and C. Niss
GIDTS − A Graphical Programming Environment for Prolog.
ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE),
pages 95−104, 1999.

KR88 B. W. Kernighan and D. M. Ritchie.
The C Programming Language.
Prentice Hall, 1988.

KR98 B. Korel and J. Rilling.
Dynamic program slicing methods.
Information and Software Technology special issue on Program Slicing,
40(11−12), pages 647−659, December 1998.

LD98 A. Lakhotia and J. Deprez.
Restructuring Programs by Tucking Statements into Functions.
Information and Software Technology special issue on Program Slicing,
40,(11−12), pages 677−690, November 1998.

LH96 L. D. Larsen and M. J. Harrold.
Slicing object−oriented software.
18th International Conference on Software Engineering (ICSE),
pages 495−505, Berlin, March 1996.

LW86 J. R. Lyle and M. Weiser.
Experiments on Slicing−Based Debugging Tools.
Empirical Studies of Programming,

Elliot Soloway (editor), Ablex Publishing, Norwood, New Jersey, June 1986.

OB98 L. M. Ott and J. M. Bieman.
Program Slices as an Abstraction for Cohesion Measurement.
Information and Software Technology special issue on Program Slicing,
40,(11−12), pages 691−700, November 1998.

SD96 S. Schoenig and M. Ducasse
A Backward Slicing Algorithm for Prolog.
Lecture Notes in Computer Science, Vol. 1145, p. 317−326, 1996.

Ste98 C. Steindl
Intermodular Slicing of Object−Oriented Programs.
International Conference on Compiler Construction,
Lecture Notes in Computer Science, LNCS, Vol. 1383, pages 264−278, 1998.

Tip95 F. Tip.
A survey of program slicing techniques.
Journal of Programming Languages, 3(3):121−189, September 1995.

Tip96 F. Tip, J.−D. Choi, J. Field and G. Ramalingam,
S*licing Class Hierarchies in* `C++.`
SIGPLAN Notices, Vol. 31, 10, pp. 179−197, ACM Press, October 6−10 1996.

Unr The `Unravel` Slicing System.
`http://WWW.NIST.GOV/itl/div897/sqg/unravel/unravel.html`

Ven91 G. A. Venkatesh
The Semantic Approach to Program Slicing.
SIGPLAN Notices, 26(6), pp. 107−119, June 1991.

Wei79 M. Weiser
Experiments on Slicing−Based Debugging Aids.
in Empirical Studies of Programmers, pp. 187−197, Ablex Publishing Corporation, 1986.

Wei82 M. Weiser
Programmers Use Slicing When Debugging.
Communications of the ACM, Vol. 25(7), pp. 446−452, July 1982.

Wei84 M. Weiser
Program slicing.
IEEE Transactions on Software Engineering, 10, pages 352−357, July 1984.

WA98 M. R. Woodward and S. P. Allen

Slicing Algebraic Specifications,
Information and Software Technology, ISSN: 0950−5849, 40(2), 105−118, May 1998.

WPS The Wisconsin Program Slicing System.
`http://www.cs.wisc.edu/wpis/html/`

ZCU97 J. Zhao, J. Cheng, and K. Ushijima.
Slicing concurrent logic programs.
In T. Ida, A. Ohori, and M. Takeichi, editors,
Second Fuji International Workshop on Functional and Logic Programming,
pages 143−162, 1997.

Figures

```
x = 1;
y = 2;
z = y-2;
r = x;
z = x+y;
/* the slice point is the end of the program */
```

Figure 1: A program fragment to be backward sliced

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

Figure 2: The backward slice of Figure 1

```
scanf("%d",&n);
s=0;
p=0;
while (n>0)
{
  s=s+n;
  p=p*n;
  n=n-1;
}
printf("%d%d",p,s);   }
/* the slice point is the end of the program */
```

Figure 3: A program fragment to be sliced

```
scanf("%d",&n);
p=0;
while (n>0)
{
  p=p*n;
  n=n-1;
}
```

Figure 4: A static slice of Figure 3

```
p=0;
```

Figure 5: A dynamic slice of Figure 3

```
scanf("%d",&x);
scanf("%d",&y);
if(x>y)
  z = 1;
else
  z = 2;
printf("%d",z);
```

Figure 6: Conditioned slice example

```
for(i=0,sum=a[0],biggest=sum;i<19;sum=a[++i])
   if (a[i+1] > biggest)
      biggest = a[i+1];
average = sum/20;
```

Figure 7: Program to be Sliced Amorphously

```
void Marks()
{ int Pass, Fail, Count;

Pass = 0 ;
Fail = 0 ;
Count = 0 ;
while (!eof()) {
  input(Marks);
  if (Marks >= 40)
    Pass = Pass + 1;
  if (Marks < 40)
    Fail = Fail + 1;
  Count = Count + 1;}
output(Count) ;
output(Pass) ;
output(Fail) ;
}
```

Figure 8: A program with three processing elements

```
void Processing_element_count()
{ int Count;

Count = 0 ;
while (!eof()) {
  input(Marks);
  Count = Count + 1;}
}
```

Figure 9 : The processing element for Count

```
void Processing_element_Pass()
{ int Pass;

Pass = 0 ;
while (!eof()) {
  input(Marks);
  if (Marks >= 40)
    Pass = Pass + 1;}
}
```

Figure 10: The processing element for Pass

```
void Processing_element_fail()
{ int  Fail;

Fail = 0 ;
while (!eof()) {
  input(Marks);
  if (Marks < 40)
    Fail = Fail + 1;}
}
```

Figure 11: The processing element for Fail

```
void Marks()
{ int Pass, Fail, Count;

Pass = 0 ;                                  P
Fail = 0 ;                                          F
Count = 0 ;                         C
while (!eof()) {          C         P         F
  input(Marks);           C         P         F
  if (Marks >= 40)                  P
    Pass = Pass + 1;                P
  if (Marks < 40)                             F
    Fail = Fail + 1;                          F
  Count = Count + 1;}     C
output(Count) ;
output(Pass) ;
output(Fail) ;
}
```

Figure 12: Depiction of Low Cohesion

```
void MinMax()
{ int Smallest, Largest;
  int num, i;

for (i=0;i<10;i=i+1) {              L         S
input(num);                                   L         S
  NumArray[i] = num;}               L         S
Smallest = NumArray[0];             L         S
Largest = Smallest;                           L
i = 1;                                        L         S
while (i<10) {                                L         S
  if (Smallest > NumArray[i])                 S
    Smallest = NumArray[i];                   S
  if (Largest < NumArray[i])        L
    Largest = NumArray[i];          L
  i = i + 1;}                                 L         S
output(Smallest);
output(Largest);
}
```

Figure 13: Depiction of high cohesion

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

Figure 14: The atoi program

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

Figure 15: The atoi program for strings beginiing with '+'

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

Figure 16: The atoi program for strings beginiing with '−'

```
r=0;
```

Figure 17: The atoi program for empty strings