Amorphous Procedure Extraction

Mark Harman[1] **David Binkley**[2] **Ranjit Singh**[1] **Robert M. Hierons**[1]

[1]Brunel University, Uxbridge, Middlesex, UB8 3PH, UK.

[2]Loyola College, Baltimore MD, 21210-2699, USA

Keywords: Program Slicing, Program Transformation, Variable Dependence Analysis

Abstract

The procedure extraction problem is concerned with the meaning preserving formation of a procedure from a (not necessarily contiguous) selected set of statements. Previous approaches to the problem have used dependence analysis to identify the non-selected statements which must be 'promoted' (also selected) in order to preserve semantics. All previous approaches to the problem have been syntax preserving.

This paper shows that by allowing transformation of the program's syntax it is possible to extract both procedures and functions in an amorphous manner. That is, although the amorphous extraction process is meaning preserving it is not necessarily syntax preserving.

The amorphous approach is advantageous in a variety of situations. These include when it is desirable to avoid promotion, when a value-returning function is to be extracted from a scattered set of assignments to a variable, and when side effects are present in the program from which the procedure is to be extracted.

1 Introduction

Procedure extraction, as introduced by Komondoor and Horwitz [18], is the problem of extracting a collection of marked statements into a free–standing procedure while maintaining the semantics of the program. Initial marking of statements might come from an attempt at refactoring, clone detection, or the result of an attempt at reuse. The specific details of how the set of statements comes to be initially marked is not the focus of this paper and is assumed to have been performed. The focus of this paper is how transformation of the source code can facilitate the more precise extraction of a set of statements.

One of the key technical issues with the procedure extraction concerns statement promotion. Promotion occurs when a statement not marked for extraction must, nonetheless be extracted, in order to preserve the program's semantics. Such an initially unmarked statement is said to be 'promoted' as it effectively becomes marked during the extraction process.

An important goal of procedure extraction is to minimize the need for promotion, because it increases the size of the extracted procedure; thus, reducing precision. The closer the extraction process can come to extracting only those statements initially marked, the better.

Prior approaches to procedure extraction have been syntax–preserving. That is, the extracted procedure is formed entirely from statements in the original program (both those originally marked and those promoted). Furthermore, apart from the call to the new procedure, the original program is formed entirely by deletion of extracted statements and the occasional replication of predicates from the original program.

This paper describes an *amorphous* approach to procedure and function extraction. The amorphous approach to procedure extraction allows additional flexibility in the extraction process. This helps reduce the need for promotion and predicate replication. It is achieved by relaxing the syntactic constraints on the extraction process; thus, the amorphous version can use transformation as well as deletion and copying. However, amorphous procedure extraction retains the requirement that the extracted program and the original must be semantically equivalent.

The resulting algorithms presented herein seem a more natural approach to adopt for procedure extraction, since the need to retain a strict syntactic link between the extracted and original version of the program is not important in many applications. Also, since the program is (to some extent) being transformed by the extraction process itself, it seems that the user of procedure extraction has already accepted some form of program transformation. Finally, the amorphous approach is better suited to handling programs with side effects.

The principal contributions of the paper are as follows:

1. Amorphous extraction is introduced.
2. Amorphous extraction is shown to reduce the need for statement promotion; thus, leading to more precise extraction.
3. Amorphous extraction is shown to be able to extract value-returning functions.
4. Amorphous extraction is shown to be more effective when applied to programs with side effects.

The rest of this paper is organized as follows. Section 2 reviews the procedure extraction problem and formalizes the definition of both syntactic and amorphous procedure extraction. Section 3 shows how the amorphous approach reduced the need for promotion. Section 4 shows how the amorphous approach is well-suited to handling the related problem of value–returning function extraction. Section 5 shows how the amorphous approach is better suited than the syntax–preserving approach in the presence of side–effects. Finally Section 6 concludes with directions for future research.

2 The Amorphous and Syntax–Preserving Procedure Extraction Problems

This section introduces the two forms of procedure extraction studied and addresses the question of why program slicing cannot simply be used to perform extraction. To begin with, Komondoor and Horwitz [18] define the (syntax–preserving) procedure extraction problem as a three step process:

Step 1. Identify the '*marked*' statements (*i.e.*, those statements whose extraction is desired).

Step 2. Form a single-entry/single-exit block of contiguous statements containing the marked statements.

Step 3. Extract the contiguous block into a procedure.

The first step is performed by an identification tool or a programmer and is application dependent. For example, the statements might be identified by a clone detection tool. Alternatively, the marked statements might have been identified by a source code analysis tool or a software maintainer as a single thread of computation to be better encapsulated in a single procedure.

The second step involves code motion and promotion. Code motion attempts to move unmarked statements either before the first marked statement or after the final marked statement. When code motion is not possible (perhaps because of data-flow constraints), statements are promoted to the set of included statements. This process continues until the marked statements form a contiguous code fragment. Komondoor and Horwitz define this step in terms of a control-flow graph (CFG) [18, 19]. Their algorithm then identifies a single-entry/single-exit subgraph to be extracted. Statements (CFG nodes) required to make the subgraph single-entry/single-exit and that can not be moved to a position before or after the marked statements are promoted (and thus extracted).

The final step is the extraction itself. This includes determining the location of the call to the new procedure, the formal parameters of this procedure, and the actual parameters used at the call. These can be determined by a dependence analysis similar to that used in program slicing [2, 3, 7, 26]. That is, a variable in the procedure body will be passed as a call-by-value formal parameter if it is referenced but not defined. It will be passed as a call-by-reference formal parameter if it is defined in the procedure body.

Following Komondoor and Horwitz [18, 19], the focus of this paper is upon Step 2. In particular, new techniques are introduced that allow certain promotions to be avoided resulting in a smaller (*i.e.*, more precise) set of statement to be extracted. The paper does not consider the initial marking of statements (Step 1) nor the creation of a procedure from the statements identified by Step 2 (Step 3).

2.1 Syntax–Preserving Procedure Extraction

All previous approaches to procedure (and function) extraction are syntax preserving in the sense that the statements of the body of the procedure exist in the original program[1]. Thus, the syntax of the original program is *preserved* in the procedure body.

In order to form a contiguous block of statements for extraction, it is desirable to 'move aside' statements which are not marked, thereby producing a sequence of only marked statements. Unfortunately, data and control dependencies may prevent unmarked statements from being moved aside. Where a statement cannot be moved aside, it must be promoted. That is, it must become marked for extraction.

For example, consider the fragment of code in Figure 1. Assume that Step 1 has identified an initial col-

[1] In order to ensure that the return is executed identically in the original and extracted versions of the program, a minor 'amorphousness' is allowed by Komondoor and Horwitz. This is the introduction of a 'return flag'. Perhaps this suggests that the nature of the procedure extraction problem forces the consideration of some form of amorphousness.

```
1   ++  evens = 0
2       sum = 0
3   ++  i = 0
4   ++  if (A[i] != 0)
5       {
6          A[i] = abs(A[i])
7   ++     if (A[i]%2 == 0)
8   ++        evens = evens + 1
9          sum = sum + A[i]
10      }
```

Figure 1. An example requiring assignment promotion and predicate replication.

lection of statements. Following Komondoor and Horwitz, these are marked "++" in the figures of this paper. The goal of Step 2 is to extract these statements while excluding the intervening unmarked statements (in this case, Statements 2 and 6). For Statement 2, the absence of data dependence between Statements 1 and 2 allows Statement 2 to be pushed backwards past Statement 1. Thus, Statement 2 need not be promoted. However, using standard[2] dataflow arguments [30, 15], Statement 6 "`A[i] = abs(A[i])`" cannot be pushed backwards or forwards because it depends upon Statement 4, which precedes it, and Statement 7, which follows it, depends on it. Therefore, in order to preserve both syntax and semantics, Statement 6 must be promoted.

In addition to code motion and promotion, extraction needs to consider predicates. The interesting case is a predicate with subordinate marked and an unmarked statements. Here there are two possibilities: mark all the subordinate statements or replicate the predicate. As there may be many unmarked statements, it is often preferable to replicate the predicate.

For example, in Figure 1, Statement 9 "`sum = sum + A[i]`" is not marked, but is controlled by Statement 4 "`A[i] != 0`" which is marked. In order to avoid the promotion (and therefore extraction) of Statement 9 it is necessary to replicate Statement 4.

Komondoor and Horwitz do not provide a formal definition of procedure extraction. Instead, they define procedure extraction by giving an algorithm. Definition 1, given below, captures the essence of the Komondoor and Horwitz algorithm, with the exception of their work on return flags, which is a special case that involves a limited form of amorphousness. This definition is given to facilitate comparison between the syntax–preserving and amorphous procedure extraction.

[2]It is interesting to note that a more sophisticated dataflow argument would reveal that statement 6 can, indeed, be moved forward because taking the absolute value of a variable does not affect its evenness. However, such dataflow arguments are typically computationally expensive to implement.

Definition 1 (Syntax Preserving Procedure Extraction)

Given a set of (not-necessarily contiguous) marked statements M from a program P, let S_α be the statement preceding (lexigraphically) the first statement in M, let S_ω be the statement following the last statement in M, and let M^+ be a contiguous set of statements such that $M \subseteq M^+$. A *Syntax–Preserving Procedure Extraction* of M is a procedure Q which contains M^+ and a program P' obtained from P by replacing M^+ with a single call to Q, such that the region bounded by S_α and S_ω in P and P' are semantically equivalent (*i.e.*, when executed starting at S_α on the same state if either reaches S_ω then both reach S_ω and furthermore they do so in states having the same values for all live variables.) $\square$

Observe that, since P and P' are syntactically equivalent outside of the region bounded by S_α and S_ω, the above implies that they are semantically equivalent (*i.e.*, for all initial states on which either terminates they both terminate in identical final states). Also observe that promotion never adds a statement beyond the last statement in M nor from before the first statement of M.

2.2 Amorphous Procedure Extraction

Amorphous procedure extraction is a relaxation of the syntax–preserving formulation, which allows transformation of the original program in order to 'massage out' the semantics associated with the marked statements. Compare this with syntax–preserving procedure extraction, in which the marked statements are extracted 'in tact' and may bring with them additional statements via promotion.

Definition 2 (Amorphous Procedure Extraction)
Given a set of (not-necessarily contiguous) marked statements M from a program P, let S_α be the statement preceding (lexigraphically) the first statement in M, let S_ω be the statement following the last statement in M, and let the sequence "*prefix*; M^+; *postfix*" be obtained from the statements between S_α and S_ω through transformation. An *Amorphous Procedure Extraction* of M is a procedure Q and a program P' obtained from P by replacing the statements between S_α and S_ω with the sequence "*prefix*; call Q; *postfix*", such that the region bounded by S_α and S_ω in P and P' are semantically equivalent

(*i.e.*, when executed from S_α on the same state if either reached S_ω then both reach S_ω in states having the same values for all live variables.) □

2.3 Why Extraction is not Just a Special Case of Program Slicing

At first sight it might appear that procedure extraction is merely a special case of program slicing [30, 15]. A program slice captures a semantic projection of a program. It is constructed by deleting from a program those statements and predicates that do not affect the value of a chosen set of variables at a chosen point within the program. Thus, one might ask, why not simply slice the original program on the variables assigned in the set of marked statements? Alternatively, why not perform a simultaneous slice on all the marked statements; something akin to a decomposition slice [10]?

There are two reasons why slicing is not appropriate. The following also applies to other slicing related operations such as limited slicing [31], chopping [16, 20], and the 'wedge' [21]. First, slicing captures unwanted statements on which marked statements depend. As these statements are not tagged for extraction, slice–based approaches are inexact. For example, any statement in the slice that occurs before the first marked statement need not be extracted as it is already 'out of the way.' Second, there can be statements between marked statements that are in the slice on marked statements, but should not be extracted. For example, consider

```
++  x = 1;
    y = 1;
++  a = y + x;
```

For this code fragment, the slice on the marked statements includes the whole fragment. In particular, the assignment to y is in the slice because of the data dependence of the assignment to a upon it. However, procedure extraction moves the y assignment back before the first statement and then extracts just the assignments to x and a.

3 Increased Precision: Avoiding Promotion

The next three sections present the three improvements amorphous procedure extraction brings to the extraction problem. First, using an amorphous approach it becomes possible to avoid promoting statements that would otherwise require promotion. This leads to smaller extracted procedures when compared with the syntax-preserving counterparts. The relationship between amorphous and syntax–preserving procedure extraction is similar to the relationship between amorphous and syntax–preserving slicing [12]: amorphous slices are typically smaller than their corresponding syntax–preserving counterparts. To illustrate the issue, consider the following simple program fragment

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

The problem faced by syntax–preserving extraction is that the assignment to x is unwanted in the extracted procedure, but it cannot be moved 'out of the way.' It cannot move before the assignment to z, because it references z. It cannot move after the assignment to y because it assigns to x. With the syntax–preserving approach, the only solution is to *promote* the assignment to x so that it, too, becomes marked. However, this makes the extracted procedure less precise, as it contains statements not originally marked for extraction.

Using amorphous procedure extraction, it becomes possible to transform the program such that the assignment to x can move after the assignment of y. This is achieved by a substitution of the value assigned to x in the assignment to y:

```
++    z = z * 2;
++    y = (z+1) + 2;
      x = z + 1;
```

The resulting assignment to y can then be simplified, yielding the following code:

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

This transformation and simplification is typical of amorphous slicing [12, 14]. In the context of procedure extraction, the sacrifice of the syntax–preserving nature of traditional procedure extraction allows the extraction of fewer statements with the amorphous approach.

4 Function Extraction for Defined Variable

The second of the three improvements amorphous procedure extraction brings to the extraction problem deals with extracting functions. The function extraction problem is a natural counterpart to the procedure extraction problem. While the techniques introduced by Komondoor and Horwitz extends to function extraction, this section shows that an amorphous approach is highly suited to the function extraction problem, leading to smaller extracted functions.

To begin with consider the fragment shown in the left column of Figure 2, which calculates total and average rainfall for a set of readings stored in an array

```
anomalous = 0
count = 0
Total = 0
while (count < MAX &&
   A[i] != Terminator)
{
  Total = Total + A[i]
  count++
}
if (Total < 0)
  anomalous = 1
if (Total == 0
    || count == 0)
{
  anomalous = 1
  average = 0
}
else
{
  average = Total / count
  if (average > Highest)
  {
    anomalous = 1
    Highest = average
  }
}
if (!anomalous)
   Report(Total, Average)
else
   ReportAnomalous()
```

```
++ anomalous = 0
   count = 0
   Total = 0
   while (count < MAX &&
     A[i] != Terminator)
   {
     Total = Total + A[i]
     count++
   }
   if (Total < 0)
++    anomalous = 1
   if (Total == 0
    || count == 0)
   {
++   anomalous = 1
     average = 0
   }
   else
   {
     average = Total / count
     if (average > Highest)
     {
++     anomalous = 1
       Highest = average
     }
   }
   if (!anomalous)
     Report(Total, Average)
   else
     ReportAnomalous(;)
```

```
   count = 0
   Total = 0
   while (count < MAX &&
     A[i] != Terminator)
   {
     Total = Total + A[i]
     count++
   }
++ anomalous = 0
   if (Total < 0)
++   anomalous = 1
   if (Total == 0
     || count == 0)
   {
++   anomalous = 1
     average = 0
   }
   else
   {
     average = Total / count
     if (average > Highest)
     {
++     anomalous = 1
       Highest = average
     }
   }
   if (!anomalous)
     Report(Total, Average)
   else
     ReportAnomalous()
```

Figure 2. Anomalous function extraction example.

A. The code contains a "flag" variable, `anomalous`, which is initially `false` (the value 0). This flag is set to `true` (the value 1) when an anomalous situation is detected. If there is no anomaly, then a standard report is generated by procedure `Report`. If there is an anomaly, then an anomalous report is generated by procedure `ReportAnomalous`. The details of these procedures are not relevant to this discussion.

While this example is a little contrived, it is intended to illustrate the way in which code may become fragmented in its processing of 'special cases.' That is, when the code was original written, the anomalous flag may have only denote a single kind of anomaly. Perhaps the anomaly that the count was zero and therefore than the average could not be properly calculated. However, as the code evolved, more and more anomalies were added resulting in multiple assignments to `anomalous`. At some point, the logic becomes rather obscure and the maintainer of the system may decide to attempt to extract the computation of `anomalous` into a boolean function.

In order to extract the code relevant to the variable `anomalous`, all the points at which `anomalous` is assigned are first marked as shown in the centre of Figure 2. The next step applies a statement push transformations [14] in an attempt to collect the assignments of `anomalous` into a single loop–free code segment. If this can be achieved, then it will be possible to extract a single conditional expression which determines the value of `anomalous`. The push transformation in essence moves assignment past other statements updating them as necessary. In this example, the initialization of `anomalous` can be pushed forward past the while loop. The result is shown in the right hand column of Figure 2.

Once a segment of loop–free code that contains the assignments to the variable of interest has been obtained, a modified amorphous slice is used to extract a single assignment statement to this variable. This can be achieved with a modified amorphous slice taken with respect to the final value of this variable in the segment of code containing the assignments. The modification excludes from the slice statements before the first marked statement and statements after the last marked statement. In this case, the amorphous slice is taken with response to the following code.

```
++  anomalous= 0
    if (Total < 0)
++    anomalous= 1
    if (Total == 0||count == 0)
    {
++    anomalous= 1
      average= 0
    }
    else
    {
      average= Total/count
      if (average > Highest)
      {
++      anomalous= 1
        Highest= average
      }
    }
```

Prior to simplification, the resulting amorphous slice is

```
anomalous =
(Total < 0) ?
   ((Total == 0 || count == 0) ?
   1:
   (((Total / count) > Highest) ?
      1:
      1)):
   ((Total == 0 || count == 0) ?
      1:
      (((Total / count) > Highest)
         1:
         0))
```

This is simplified[3] to

[3]The simplification algorithm [14] is defined for WSL [27]. The implementation uses the FermaT `Simplify` transformation [28] tactic to do this series of simplifications.

```
anomalous =
(Total < 0) ?
   1:
   (Total == 0 || count == 0) ?
      1:
      (Total / count) > Highest
```

which can be further simplified to (although the current implementation of the amorphous slicer does not discover this simplification)

```
anomalous =
  Total <= 0 || count == 0
  || (Total/count) > Highest
```

This assignment captures the meaning of the flag variable `anomalous`. It can be extracted into a function, called `anomalous` and inserted back into a transformed version of the original program as follows:

```
count = 0
Total = 0
while (count < MAX &&
       A[i] != Terminator)
{
  Total = Total + A[i]
  count++
}
if (Total < 0)
  ;
if (Total == 0 || count == 0)
   average = 0
else
{
  average = Total / count
  if (average > Highest)
    Highest = average
}
if (Anomalous(Total, Count, Highest))
  Report(Total, Average)
else
  ReportAnomalous()
```

In this version, all assignments to `anomalous` have been replaced by the use of a call to the following function

```
int Anomalous(int Total, int Count,
              int Highest)
{
  return Total <= 0 || count == 0 ||
       (Total/count) > Highest
}
```

Of course, removing the assignments to `anomalous` in the original program fragment, may occasion some additional simplification, for example, deletion of the (now empty) statement:

```
if (Total < 0)
  ;
```

This example has illustrated how amorphous function extraction can extract a function which captures the computation of a selected variable. The original program is re-factored to contain a single simple function which isolates all the computation relevant to the variable. This allows the remaining part of the program to be simplified. In the example, anomalous data is handled in one single place rather than scattered throughout the program.

An interesting application of amorphous function extraction is when the extracted function may not be retained. This is in contrast to previous proposed uses of syntax–preserving procedure extraction (*e.g.*, clone detection). Thus, in the above example, the extraction is performed entirely for the sake of comprehension. Indeed, the maintainer may decide to jettison the extracted function, once he or she has used it to assist in understanding the logic captured by the flag variable.

Another example in which the extracted function is not retained is flag removal as used in testability transformation [13, 1]. In this application, it is desirable to remove flags from a *version* of the program purely to generate test data. Once the test data is generated it is executed on the original program and not on the transformed program; the transformed program is no longer needed and can be 'thrown away.'

5 Procedure Extraction in the Presence of Side Effects

The final improvement amorphous procedure extraction brings to the extraction problem deals with side effects. A side effect is any state change caused by the evaluation of an expression. In contrast, side–effect free expressions, when evaluated simply returns a value, causing no change in state.

Side effects significantly increase the complexity of procedure extraction. It may become necessary to perform (otherwise) unnecessary promotions, simply to extract whole statements. Amorphous extraction can avoid their promotion. For example, consider the program fragment shown in Figure 3. In this code, it is inconvenient to use the traditional "++" notation to denote marked statements as the program contains expressions with side effects and it may be desirable to extract an expression with a side–effect, rather than the statement that contains it. In order to show the text to be extracted, such text will appear `shadowed`.

```
x = a - b++ - c
y = x*2
x++
```

Figure 3. An example with side effects.

```
x = a - b - c
b++
y = x*2
x++
```

Figure 4. The example from Figure 3 with side effects removed.

The program fragment shown in Figure 3 can be transformed to remove side–effects [8]. The result, shown in Figure 4, is a longer program, but one in which the side effects in expressions are made explicit in the form of added statements.

For this example, the removal process is straightforward. In the presence of arrays, loops, or conditional statements, the process of side–effect removal is more complex and can significantly alter the program syntax. Therefore, any approach based upon prior side–effect removal will be inherently amorphous.

With the side–effect free program, it is "easy" to extract the procedure, by moving the assignment to `y` backwards past the assignment to `b`. Without prior side effect removal, this transformation is not possible as the `y` assignment cannot be pushed backwards past the `x` assignment since the `y` assignment references `x`. It also cannot be pushed forwards past the increment of `x` for the same reason. Thus, the assignment to `y` would have to be promoted by the syntax–preserving extraction process.

Using the amorphous approach it becomes possible to "free" this statement. In this case, side effect removal, separates out the side–effecting part of the program (which is targeted for extraction) from the side–effect free part (which is not targeted for extraction). This separation, allows the unwanted `y` assignment to be pushed up past the side effects, though not past the (side effect free version of) the original `x` assignment.

The presence of side–effects prohibitively complicates the application of many software engineering techniques, such as symbolic execution [6], partial evaluation [4, 9] and transformation [29, 22, 24], which typically consider only side effect-free systems. Such approaches can usually be defined to handle side effects, but this often requires effort disproportionate to the perceived gain.

The presence of side-effects is also widely believed to inhibit program comprehension, and is deprecated in many guides, standards, and sets of style rules for programming [17, 25, 23, 11, 5]. However, despite this advice many programs, in particular C language programs, typically contain significant side effect using expressions.

The concern in this paper is not the impact upon comprehension, but the increased need for promotion caused by side–effects when using syntax–preserving extraction. In some cases, the requirement to preserve syntax may lead to large scale promotion of unwanted code, making the extraction rather ineffective. Amorphous function extraction is much better suited to extract code from C programs. For example, consider the following C fragment.

```
for (count = tot = average = 0;
     c = A[count++] != 0;
     tot += c)
{
  // empty
}
average = tot/count
```

The fragment is intended to compute the total and average of the values in array A, up to, but not including the first zero–valued element of the array. Suppose the programmer wishes to examine the effect of the program solely upon the variable `count`. The programmer may extract this variable into a single procedure (or a function which returns its value). With the syntax-preserving form of extraction, the whole loop must be extracted in tact. The amorphous version begins by first transforming the loop to be side effect free:

```
count = 0
tot = 0
average = 0
for (;A[count] != 0;)
{
  c = A[count]
  count++
  tot += c
}
count++
average = tot/count
```

Notice that after side–effect removal, the `for` loop has effectively become a `while` loop. However, this is less important than the way in which side–effect removal has clearly highlighted a bug in the original version of the program. The value of `count` is incremented one more time than necessary by the loop and so the computation of `average` is incorrect.

Nonetheless, the programmer may not notice this until the variable `count` is extracted from the program into a single procedure. The extracted body of which will be

```
count = 0;
for (;A[count] != 0;)
  count++;
count++;
```

Notice how the amorphous procedure extracted clearly highlights the bug in the computation of `count` and the role which side–effect removal has played in this process. The fact the side effect removal is inherently amorphous, suggests that function (and procedure) extraction in the presence of side effects will profit from adopting an 'amorphous' approach.

6 Conclusion and Future Work

This paper has introduced a variation of Komondoor and Horwitz's procedure extraction. Whereas the original formulation of the problem was largely syntax–preserving, the new version is amorphous. That is, while it preserves the semantic connection between the extracted and unextracted program, there is not necessarily a syntactic link.

The paper shows that the amorphous approach conveys three advantage. It reduces the need for statement promotion (reducing the size of the extracted procedure); thus, increasing precision. It is also better suited to handling the related problem of function extraction, where it again produces smaller extracted functions than the syntax-preserving version. Third, the amorphous approach produces smaller extracted procedures in the presence of side effects. Future work on amorphous procedure extraction includes empirical studies that will help asses the degree to which the these advantages lead to practical improvements for real programs.

References

[1] BARESEL, A., BINKLEY, D. W., HARMAN, M., AND KOREL, B. Evolutionary testing in the presence of loop–assigned flags: A testability transformation approach. In *International Symposium on Software Testing and Analysis (ISSTA 2004)* (Omni Parker House Hotel, Boston, Massachusetts, July 2004). To appear.

[2] BINKLEY, D. W., AND GALLAGHER, K. B. Program slicing. In *Advances in Computing, Volume 43*, M. Zelkowitz, Ed. Academic Press, 1996, pp. 1–50.

[3] BINKLEY, D. W., AND HARMAN, M. A survey of empirical results on program slicing. *Advances in Computers 62* (2004), 105–178.

[4] BJØRNER, D., ERSHOV, A. P., AND JONES, N. D. *Partial evaluation and mixed computation*. North–Holland, 1987.

[5] CANNON, L., ELLIOTT, R., KIRCHHOFF, L., MILLER, J., MILNER, J., MITZE, R., SCHAN, E., WHITTINGTON, N., SPENCER, H., KEPPEL, D., AND BRADER, M. Recommended C style and coding standards, 2000. http://www.cs.umd.edu/users/cml/cstyle/indhill-cstyle.html.

[6] COEN-PORISINI, A., AND DE PAOLI, F. SYMBAD: A symbolic executor of sequential Ada programs. In *IFAC SAFECOMP'90* (London, 1990), pp. 105–111.

[7] DE LUCIA, A. Program slicing: Methods and applications. In 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation* (Florence, Italy, 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 142–149.

[8] DOLADO, J. J., HARMAN, M., OTERO, M. C., AND HU, L. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE Transactions on Software Engineering 29*, 7 (2003), 665–670.

[9] FUTAMURA, Y., AND NOGI, K. Generalized partial computation. In *IFIP TC2 Workshop on Partial Evaluation and Mixed Computation* (1987), D. Bjørner, A. P. Ershov, and N. D. Jones, Eds., North–Holland, pp. 133–151.

[10] GALLAGHER, K. B., AND LYLE, J. R. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering 17*, 8 (Aug. 1991), 751–761.

[11] HAAHR, P. A programming style for java, Oct. 1999. http://www.webcom.com/˜haahr/essays/java-style/.

[12] HARMAN, M., BINKLEY, D. W., AND DANICIC, S. Amorphous program slicing. *Journal of Systems and Software 68*, 1 (Oct. 2003), 45–64.

[13] HARMAN, M., HU, L., HIERONS, R. M., WEGENER, J., STHAMER, H., BARESEL, A., AND ROPER, M. Testability transformation. *IEEE Transactions on Software Engineering 30*, 1 (Jan. 2004), 3–16.

[14] HARMAN, M., HU, L., MUNRO, M., ZHANG, X., BINKLEY, D. W., DANICIC, S., DAOUDI, M., AND OUARBYA, L. Syntax-directed amorphous slicing. *Journal of Automated Software Engineering 11*, 1 (Jan. 2004), 27–61.

[15] HORWITZ, S., REPS, T., AND BINKLEY, D. W. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12*, 1 (1990), 26–61.

[16] JACKSON, D., AND ROLLINS, E. J. A new model of program dependences for reverse engineering. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering* (Dec. 1994), pp. 2–10.

[17] KERNIGHAN, B. W., AND PIKE, R. *The practice of programming*. Addison-Wesley Longman, Reading, Massachusetts, 1999.

[18] KOMONDOOR, R., AND HORWITZ, S. Semantics-preserving procedure extraction. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00)* (N.Y., Jan. 19–21 2000), ACM Press, pp. 155–169.

[19] KOMONDOOR, R., AND HORWITZ, S. Effective automatic procedure extraction. In *11th IEEE International Workshop on Program Comprehension* (Portland, Oregon, USA, May 2003), IEEE Computer Society Press, Los Alamitos, California, USA, p. To appear.

[20] KRINKE, J. Evaluating context-sensitive slicing and chopping. In *IEEE International Conference on Software Maintenance (ICSM 2002)* (Montreal, Canada, Oct. 2002), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 22–31.

[21] LAKHOTIA, A., AND DEPREZ, J.-C. Restructuring programs by tucking statements into functions. In *Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier, 1998, pp. 677–689.

[22] MEHLICH, M., AND BAXTER, I. Mechanical tool support for high integrity software development. In *High Integrity Systems '97* (1997), IEEE Computer Society Press, Los Alamitos, California, USA.

[23] MEYER, B. *Object-oriented Software Construction*, second ed. Prentice Hall, New York, NY, 1997.

[24] PARTSCH, H. A. *The Specification and Transformation of Programs: A Formal Approach to Software Development*. Springer, 1990.

[25] THOMAS, P. *Learning to program in C*, 2 ed. Plum Hall, Inc., 1989.

[26] TIP, F. A survey of program slicing techniques. *Journal of Programming Languages 3*, 3 (Sept. 1995), 121–189.

[27] WARD, M. *Proving Program Refinements and Transformations*. DPhil Thesis, Oxford University, 1989.

[28] WARD, M. Assembler to C migration using the FermaT transformation system. In *IEEE International Conference on Software Maintenance (ICSM'99)* (Oxford, UK, Aug. 1999), IEEE Computer Society Press, Los Alamitos, California, USA.

[29] WARD, M., CALLISS, F. W., AND MUNRO, M. The maintainer's assistant. In *Proceedings of the International Conference on Software Maintenance 1989* (1989), IEEE Computer Society Press, Los Alamitos, California, USA, p. 307.

[30] WEISER, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984), 352–357.

[31] YANG, W., HORWITZ, S., AND REPS, T. A program integration algorithm that accommodates semantics-preserving transformations. *ACM Transactions on Software Engineering and Methodology 1*, 3 (July 1992), 310–354.