# An Interprocedural Amorphous Slicer for WSL

**Mark Harman**
**Lin Hu**
Brunel University
Uxbridge, Middlesex
UB8 3PH, UK.

**Malcolm Munro**
**Xingyuan Zhang**
University of Durham
South Road, Durham
DH1 3LE, UK.

**Sebastian Danicic**
**Mohammed Daoudi**
**Lahcen Ouarbya**
Goldsmiths College
New Cross, London
SE14 6NW, UK.

Keywords: Amorphous Slicing, Conditioned Slicing, Transformation, WSL, FermaT

**Abstract**

*This paper presents a simple interprocedural algorithm for amorphous slicing and illustrates the way in which interprocedural amorphous slicing improves upon interprocedural syntax–preserving slicing. The paper also presents results from an empirical study of an implementation of this algorithm for Ward's Wide Spectrum Language, WSL. The implementation uses the FermaT transformation workbench. It combines FermaT transformations with the results produced by a syntax-preserving slicer for WSL. Finally, it is shown that the combination of amorphous slicing and conditioned slicing can be particularly attractive, by combining results from the amorphous slicer with results from a prototype conditioned slicer for WSL.*

## 1 Introduction

Program slicing is an automated source code extraction technique which produces a version of a program that preserves a projection of the original program's semantics [2, 5, 9, 12, 17]. Traditionally, this projection is defined in terms of a subset of variables of interest and is constructed using the sole transformation of statement deletion [17]. The slice is therefore a subprogram which preserves a subcomputation.

Amorphous slicing [1, 8, 10] is a variation of traditional slicing, in which the semantic properties are retained (the slice maintains a subcomputation), while the syntactic restriction to statement deletion is relaxed. By sacrificing the connection with syntax, smaller slices can be produced. Amorphous slicing is therefore useful in applications of slicing where syntax is less important than size reduction.

The idea of improving syntax-preserving slicing using transformations on an underlying intermediate representation was first suggested by Tip [11] and Ernst [7]. The idea of using these transformations to produce amorphous slices was first suggested by Harman and Danicic [8]. Relaxing the syntactic restriction of traditional slicing typically produces smaller slices. However, the extra simplification potential of amorphous slicing can only be achieved if algorithms can be found which produce good quality (i.e. small) slices in reasonable time.

The construction of amorphous slices is harder than the construction of syntax-preserving slices, because *any* transformation can potentially be applied in amorhous slicing. Previously published algorithms for amorphous slicing [1, 10] have concerned only intraprocedural languages. In this paper we present a simple interprocedural amorphous slicing algorithm and initial results on the performance of the dependence reduction transformation at the heart of our amorphous slicing algorithm. Our current approach simply 'pushes' the intraprocedural analysis into procedure and function bodies[1]. However, even given the limitations of the current approach, we are able to construct some interesting slices which illustrate the way in which interprocedural amorphous slicing improves upon interprocedural syntax-preserving slicing.

For example, consider the example in the leftmost section of Figure 1. The program is written in WSL. Though the syntax may not be familiar, the meaning of this simple example should be relatively straightforward. The main program consists of the final four lines of code. It makes two calls to the procedure called `@BoundedFind` (the presence of the `@` symbol is required by WSL, but it is unimportant). The first three parameters to this function are `Bound`, `List` and `Key` and they are all passed by value. The re-

[1] In future work we plan to extend this to provide additional domain-specific interprocedural dependence reduction transformations to further improve our amorphous slicer.

| Original Program | Amorphous Slice for Final Value of Result |
|---|---|
| `MW_PROC @BoundedFind(Bound, List, Key VAR R,Flag) ==`<br>`VAR < i:=0 >:`<br>`Flag := 0;`<br>`WHILE i<=Bound AND i<=LENGTH(List) DO`<br>`IF List[i] = Key THEN`<br>`R := i; Flag := 1; FI;`<br>`i := i + 1;`<br>`OD;`<br>`ENDVAR .;`<br><br>`@BoundedFind(LENGTH(L), L, Peter VAR PResult, PFlag);`<br>`@BoundedFind(LENGTH(L), L, Jane VAR JResult, JFlag);`<br>`Result:=0;`<br>`IF PResult < JResult THEN Result := 1 FI` | `MW_PROC @BoundedFind(List, Key VAR R) ==`<br>`VAR < i:=0 >:`<br>`WHILE i<=LENGTH(List) DO`<br>`IF List[i] = Key THEN R := i FI;`<br>`i := i + 1;`<br>`OD;`<br>`ENDVAR .;`<br><br>`@BoundedFind(L, Peter VAR PResult);`<br>`@BoundedFind(L, Jane VAR JResult);`<br>`Result:=0;`<br>`IF PResult < JResult THEN Result := 1 FI` |

Figure 1: A Program Fragment and one of its Interprocedural Amorphous Slices

maining two parameters, R and Flag are passed by value-result, and constitute the results of the procedure. The VAR . . . ENDVAR construct introduces a local variable, i, which is used as a WHILE loop counter. The rest of the program uses the Algol-like subset of WSL and is, hopefully, relatively intuitive.

The procedure @BoundedFind searches the list passed to it, until either an upper bound, specified by the formal parameter Bound, is reached or the end of the list is reached. If the element Key is located by this search, then r is set to the index of the element and Flag is set to 1 (which denotes true). Otherwise, R is unaffected and Flag is set to 0 (which denotes false).

The main program attempts to use two calls to @BoundedFind to determine whether the element Peter occurs strictly before the element Jane in the list L. It sets the flag variable Result accordingly. Slicing on Result illustrates one difference between syntax-preserving slicing and amorphous slicing, showing how amorphous slicing allows for more simplification. Observe that, for the purpose of determining the value of Result, only the returned parameter R is required. The Flag parameter can be discarded. Syntax preserving slicing can reveal this, because the slice of the body of the procedure does not contain assignments to the formal parameter Flag. However, the parameter Bound also is not required in the slice, because the WHILE loop test becomes (by substitution of formal for actual parameters and simple code motion):

```
i<=LENGTH(L) AND i<=LENGTH(L)
```

in both calls and this can be simplified to

```
i<=LENGTH(L)
```

Such a substitution and simplification, makes the formal parameter Bound redundant and so it is dropped from the definition and call in the amorphous slice. Syntax-preserving slicing is prevented from exploiting this simplification opportunity, because the syntactic restriction prevents it from substituting an actual parameter for the corresponding formal parameter. The amorphous slice is therefore simpler than the corresponding syntax-preserving slice (although it is not a syntactic subset of it).

The rest of this paper is organised as follows. Section 2 presents the interprocedural dependence reduction transformations. These simply push intraprocedural dependence reduction transformations [10] into the bodies of procedures. Section 3 describes our implementation, LinIAS, for interprocedural amorphous slicing of WSL, while section 4 presents some initial empirical results concerning the execution speed of LinIAS in terms of best and worst case examples. Section 5 illustrates the way in which amorphous and conditioned slicing complement one-another providing greater simplification than each is capable of separately. Section 6 concludes and section 7 presents some directions for future improvements in the simplification power of LinIAS.

Figure 2 describes the top level algorithm for amorphous slicing, which repeats the application of syntax-preserving slicing and dependency reduction transformation to simplify program until it reaches a fixed point.

The set of 'push' code motion transformations play an important role in reducing dependencies. These DRT transformation rules are presented in Figure 3, where the term $\mathrm{SUB}(e_2, i, e_1)$ returns the expression that results from substituting all occurrences of the

1. Syntax-preserving slicing. (preprocessing)
2. Dependency Reduction Transformation (DRT):
   (a) Reducing dependence in main procedure;
   (b) Reducing dependence in sub-procedures (if any)
   (c) Reducing dependence in functions (if any)
3. Syntax-preserving slicing.
4. If syntax-preserving slicing at step 3 did not delete any statements, then stop, else goto step 2.

Figure 2: Top Level Amorphous Slicing Algorithm

| | | |
|---|---|---|
| `y:=x+a;`<br>`x:=b+1;`<br>`z:=x+y` | `y:=x+a;`<br>`z:=b+1+y;`<br>`x:=b+1` | `z:=b+1+x+a;`<br>`y:=x+a;`<br>`x:=b+1` |
| $p$ | $p'$ | $p''$ |

Figure 4: Simple Dependence Reduction

variable $i$ in the expression $e_2$, with the expression $e_1$. REF is a function which returns the referenced variables of an expression and DEF is a function which returns the defined variables of an expression.

## 2 Algorithm for Amorphous Slicing

The algorithm consists of two main components:

1. A syntax-preserving slicer, developed by the VASTT group[2].
2. An interprocedural Dependency Reduction Transformer (DRT).

The example in Figure 4 illustrates the application of the push rules to straight line code. From program $p$, repeating the push rule transformation, we obtain $p'$, finally $p''$. In program $p$, variable `z` is explicitly dependent on `x` and `y` (`a` and `b` are symbolic constants). After transforming, variable `z` is only dependent on the initial value of `x`. With respect to variable `z`, the syntax preserving slice of program fragment $p$ is the whole program fragment. However, from the transformed program $p''$, the following amorphous slice of $p$ is obtained by traditional syntax preserving slicing:

```
z=b+1+x+a;
```

Notice how the push transformations, combined with the application of traditional, syntax preserving slicing, produces greater simplification than traditional slicing alone. The overall effect is that expressions become more complex, but statements become less interdependent, so syntax-preserving slices tend to be smaller (or at least have fewer nodes) and tend to assign to fewer variables.

## 3 The Implementation for WSL

We implemented the amorphous slicing algorithm in a tool called LinIAS. The LinIAS tool produces amorphous slices for WSL (the Wide Spectrum Language introduced by Ward [13, 15] and used in the FermaT[3] transformation workbench [14]).

Recently, Ward [16] published an algorithm for syntax-preserving slicing of WSL, using FermaT. This slicing tool is now available as a built-in transformation in the current implementation in FermaT. However, it was unavailable at the time we developed our amorphous slicer, and so we implemented our own, in-house, interprocedural syntax-preserving slicer for WSL. The algorithm used by this syntax-preserving slicer is interesting (particularly because it produces executable slices for programs with side effects), but a description is beyond the scope of the present paper, which simply treats syntax-preserving slicing as a 'black box' transformation step.

The LinIAS tool can currently handle a core of WSL statement constructs (loops, conditionals, assignments and sequencing), together with all the primitive and structured data types and procedure and function definitions and calls. We plan to implement converters (to and from) more popular languages to this core WSL language, so that we can produce amorphous slices for other languages using WSL as an intermediate representation.

## 4 Performance Analysis

The execution time of LinIAS depends predominantly upon the analysis of applicability of push rules and the consequent manipulation of the abstract syntax tree when code motion occurs. This section explores the performance of dependence reduction with respect to best and worst cases. Performance degrades if more code can be deleted, when

[2] Verification and Analysis using Slicing, Testing and Transformation. `http://www.brunel.ac.uk/~csstmmh2/vast/`

[3] Fermat is available under GPL from `www.dur.ac.uk/~dcs6mpw/martin/`.

*Push Rules:*

**Rule 1 (Assignment to Assignment: Absorb)**

$$\frac{e_3 = \mathbf{SUB}(e_2, i, e_1)}{[\![i := e_1;\ i := e_2]\!] \Rightarrow [\![i := e_3]\!]}$$

**Rule 2 (Assignment to Assignment: MoveToRight)**

$$\frac{i_1 \neq i_2, i_2 \notin \mathrm{REF}(e_1), e_3 = \mathrm{SUB}(e_2, i_1, e_1)}{[\![i_1 := e_1;\ i_2 := e_2]\!] \Rightarrow [\![i_2 := e_3;\ i_1 := e_1]\!]}$$

**Rule 3 (Assignment to If-Then)**

$$\frac{e_2' := \mathbf{SUB}(e_2, i, e_1), [\![i{:=}e_1;\ c]\!] \Rightarrow [\![c'\ \ i{:=}e_1;]\!]}{[\![i{:=}e_1;\ \texttt{IF}\ \ (e_2)\ \ \mathrm{THEN}\ c\ \mathrm{FI}]\!] \Rightarrow [\![\texttt{IF}\ \ (e_2')\ \ \mathrm{THEN}\ c'\ \mathrm{FI};\ i{:=}e_1]\!]}$$

**Rule 4 (Assignment to If-Then-Else)**

$$\frac{e_2' = \mathbf{SUB}(e_2, i, e_1),\quad [\![i{:=}e_1;\ \ c_1]\!] \Rightarrow [\![c_1'\ \ i{:=}e_1;]\!],\quad [\![i{:=}e_1;\ \ c_2]\!] \Rightarrow [\![c_2'\ \ i{=}e_1;]\!]}{[\![i{:=}e_1;\ \ \texttt{IF}\ \ (e_2)\ \ \texttt{THEN}\ c_1\ \ \texttt{ELSE}\ c_2\ \texttt{FI}]\!] \Rightarrow [\![\texttt{IF}\ \ (e_2')\ \ \texttt{THEN}\ c_1'\ \ \texttt{ELSE}\ c_2'\ \texttt{FI};\ \ i{:=}e_1]\!]}$$

**Rule 5 (Others: Statement to Statement)**

$$\frac{\texttt{DEF}(\texttt{st}_1) \cap \texttt{REF}(\texttt{st}_2) = \phi,\ \texttt{REF}(\texttt{st}_1) \cap \texttt{DEF}(\texttt{st}_2) = \phi, \texttt{DEF}(\texttt{st}_1) \cap \texttt{DEF}(\texttt{st}_2) = \phi}{[\![st_1;\ st_2]\!] \Rightarrow [\![st_2;\ st_1]\!]}$$

**Rule 6 (If Statement)**

$$\frac{\texttt{c}_1' = \texttt{DRT}(\texttt{c}_1),\ \ \texttt{c}_2' = \texttt{DRT}(\texttt{c}_2)}{[\![\texttt{IF}\ \ (\texttt{e})\ \ \texttt{THEN}\ \ \texttt{c}_1\ \ \texttt{ELSE}\ \ \texttt{c}_2\ \ \texttt{FI}]\!] \Rightarrow [\![\texttt{IF}\ \ (\texttt{e})\ \ \texttt{THEN}\ \ \texttt{c}_1'\ \ \texttt{ELSE}\ \ \texttt{c}_2'\ \ \texttt{FI}]\!]}$$

**Rule 7 (While Loop)**

$$\frac{\texttt{c}' = \texttt{DRT}(\texttt{c})}{[\![\texttt{WHILE}\ \ (\texttt{e})\ \ \texttt{DO}\ \ \texttt{c}\ \ \texttt{OD}]\!] \Rightarrow [\![\texttt{WHILE}\ \ (\texttt{e})\ \ \texttt{DO}\ \ \texttt{c}'\ \ \texttt{OD}]\!]}$$

**Rule 8 (Procedure)**

$$\frac{\texttt{ReducedBody} = \texttt{DRT}(\texttt{Body})}{[\![\texttt{MW\_PROC}\ \ \texttt{@P(X VAR Y)}\ \ ==\ \ \texttt{Body.}]\!] \Rightarrow [\![\texttt{MW\_PROC}\ \ \texttt{@P(X VAR Y)}\ \ ==\ \ \texttt{ReducedBody.}]\!]}$$

**Rule 9 (Function)**

$$\frac{\texttt{ReducedBody} = \texttt{DRT}(\texttt{Body})}{\begin{array}{c}[\![\texttt{MW\_FUNCT}\ \ \texttt{@foo(X)}\ \ ==:\ \ \texttt{Body}\ \ \texttt{(ReturnedExpression).}]\!] \\ \Rightarrow \\ [\![\texttt{MW\_FUNCT}\ \ \texttt{@foo(X)}\ \ ==:\ \ \texttt{ReducedBody}\ \ \texttt{(ReturnedExpression).}]\!]\end{array}}$$

*Simplify Transformation Rules:*

**Rule 10 (Expression)** *Using WSL symbolic computation library attempt to simplify expression*

**Rule 11 IF** (e) **SKIP ELSE c FI** $\Rightarrow$ **IF Not**(e) **THEN c FI**

**Rule 12 IF FALSE THEN $\mathbf{c_1}$ ELSE $\mathbf{c_2}$ FI** $\Rightarrow$ $\mathbf{c_2}$

**Rule 13 IF FALSE THEN c FI** $\Rightarrow$ **SKIP**

Figure 3: A Subset of the DRT Rules Applied by Step 2 of the Top Level Algorithm

| | | |
|---|---|---|
| 1 | `y := x*x;` | `x1 := a;` |
| 2 | `x := y*y;` | `x2 := b;` |
| 3 | `y := x*x;` | `x3 := x2 - x1;` |
| 4 | `x := y*y;` | `x4 := x3 - x2;` |
| 5 | `y := x*x;` | `x5 := x4 - x3;` |
| ... | ...... | ...... |
| 20 | `x := y*y;` | `x20 := x19 - x18;` |
| ... | ...... | ...... |
| n | `WHILE f(x,y)>0 DO x := x+1 OD` | $x_n := x_{n-1} - x_{n-2}$ |
| LOC | The best case (to be sliced on x) | The worst case (to be sliced on $x_n$) |

Figure 5: Examples of Best and Worst Case

| | | |
|---|---|---|
| 1 | `x3 := x2 - x1;` | `y := x*x;` |
| 2 | `x4 := x3 - x2;` | `x := y*y;` |
| 3 | `y := x*x;` | `WHILE f(-x1,x,y)>0 DO x := x+1 OD;` |
| 4 | `x := y*y;` | `y := x*x;` |
| 5 | `WHILE f(x4,x,y)>0 DO x := x+1 OD;` | `x := y*y;` |
| 6 | `x5 := x4 - x3;` | `WHILE f(x1-x2,x,y)>0 DO x := x+1 OD` |
| 7 | `x6 := x5 - x4;` | |
| 8 | `y := x*x;` | |
| 9 | `x := y*y;` | |
| 10 | `WHILE f(x6,x,y)>0 DO x := x+1 OD` | |
| LOC | Program | Amorphous slice with respect to $x$ |

Figure 6: Synthetic Program Fragment which can be Reduced 40% by Amorphous Slicing

the discovery that this code can be deleted requires repeated applications of the iteration embodied in the top level algorithm. Though this is a pathological case, which is unlikely to occur in practice, it allows us to provide an upper bound on the time taken to compute an amorphous slice in terms of the lines of code in the program to be sliced.

An example fragment which denotes the paradigm of the worst case is presented in the rightmost column of Figure 5. In this program every assignment is dependent on all previous ones, and each assignment can be pushed down. However, in order to produce the final amorphous slice, the algorithm has to push each line down through the entire program, transforming every statement as it passes and slicing at the end of each iteration. For this worst case, (human) analysis yields that the value of $x_n$ is one of following:

$$x_n := \begin{cases} (-1)^i a & \text{if } n \text{ is } 3i+1 \\ (-1)^i b & \text{if } n \text{ is } 3i+2 \\ (-1)^i (b-a) & \text{if } n \text{ is } 3i+3 \end{cases}$$

where $i \in \{0, 1, 2, ...\}$.

LinIAS achieves the above result by applying the push rules $n(n-1)/2$ times (for $n$ lines of code). We therefore expect worst case performance to be quadratic.

| | | |
|---|---|---|
| 20 | 0.01 | 0.08 |
| 30 | 0.02 | 0.15 |
| 50 | 0.03 | 0.38 |
| 100 | 0.11 | 1.83 |
| 200 | 0.27 | 8.76 |
| 300 | 0.43 | 26.70 |
| 400 | 0.63 | 65.69 |
| 500 | 0.91 | 148.71 |
| Size (LOC) | Best Case | Worst Case |

Figure 7: Execution times (in seconds) for a Pentium III, 700MHz

In the best case, no lines of code are deleted. This case is 'best' from the point of view of performance, but is clearly less attractive in terms of the results it produces as slicing is only useful when statements are deleted. Fortunately, the best case (in terms of performance) is also a somewhat pathological example, and it is presented merely to provide a lower bound on the computation time required to produce and amorphous slice.

The archetypical code fragment which typifies the best case is presented in the leftmost column of Figure 5. In this program, no dependence can be reduced, the amorphous slice is the original program itself. Although best case performance produces the identify transformation, it involves checking each statement of the program and so we expect best case performance to be linear, rather than constant time.

The size of the slice viewed as a 'percentage reduction rate' ranges from 0% in the best case (for performance) to (almost) 100% in the worst case (for performance). In the worst case, the amorphous slice becomes a single line of code and so the simplification produced is dramatic. Combining the patterns of the best case and worst case in Figure 5 we can generate synthetic program fragments which denote other cases in which a precisely defined level of size reduction will be achieved. For example, given 10 as the number of lines of code and a 'slice reduction rate' of 40%, we can construct the example program fragment in Figure 6. This example fragment is a synthetic code fragment in which the algorithm is guaranteed to consider 10 lines of code and produce a 40% reduction in size.

These 'synthetic' program examples are, of course, of theoretical interest only. In practical code examples, the slice time will be different. This may appear to make the consideration of such synthetic cases rather pointless. However, there is a difficulty in constructing an empirical study in terms of 'examples of real applications to real software'. What examples typify 'real software'? Furthermore, what slicing criteria should we choose to slice upon? These problems of subject selection would have forced us to make cautious statements in terms of real programs, such as:

> "given the set of programs we considered and the method used to determine a set of *reasonable* slicing criteria, our results show that execution time tends to be ...'

Our examples are syntheticly constructed. This provides us with a more definite statement about the behaviour with respect to *any* real program. For example, using the worst case synthetic program, we are able to say:

> "Performance of the DRT on real programs will never be worse than the results presented."

### 4.1 Results

The execution times for these best and worst cases are shown in Figure 7. Figure 8 shows the execution times for amorphously slicing programs of different sizes to achieve reduction rates 10%, 15%, 30%, 35%, 45% and the worst case respectively. We call the worst case 100%; it never quite reaches this value, because the last line always remains. However, as the size of the program to be sliced increases, the size reduction approaches 100%.

The results show that, even in the worst case, amorphous slices can be computed in reasonable time for unit level programs. Of course, real performance will be much better than this. For small units and non-pathological cases, we have found that slice construction time is of the order of a few seconds. Currently, these observations remain anecdotal. In future work we plan to explore the issue more thoroughly. Even given the worst case results we have reported here, we can be confident that optimisations in our algorithm and improvements in processor performance will ensure that these times will tend towards the instantaneous computation of amorphous slices. However, instantaneous slice construction is not required in many of the applications of amorphous slicing. Rather, we would be prepared to wait longer (perhaps of the order of a few minutes or longer) for a result of greater value. That is, we are often prepared to trade between precision and speed. Furthermore, LinIAS implements an 'anytime' algorithm; we can terminate the slice construction process at any point and see the result produced 'so far'. This partial result is guaranteed to be a valid amorphous slice, but simply may not be as small as that which could be achieved with a little more patience.

## 5 Mixing amorphous slicing with program conditioning

Conditioned slicing [3, 4, 6], is a variation of traditional slicing in which the slicing criterion is augmented by a condition. Statements and predicates which cannot affect the values of the variables of interest when the condition is satisfied are removed to form the conditioned slice. A conditioned slice can be thought of as a conditioned *program*, which is subsequently *sliced* using a static slicer. The conditioned program is one which must behave identically to the original only when the condition is satisfied. Conditioning a program can involve simplification, since lines which cannot be executed when the condition is met may be removed.

Traditionally, conditioned slices have been constructed by statement deletion and so they produce syntax-preserving slices. However, the conditioning aspect of conditioned slicing is a semantic augmentation; there is no reason *not* to combine the conditioning process with amorphous (rather than syntax

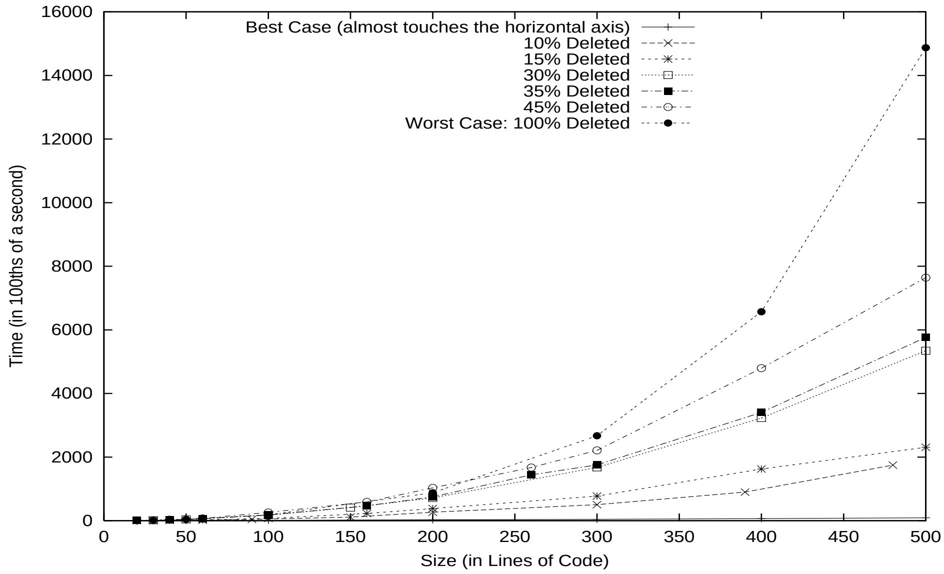


Figure 8: Execution Times for Best and a Set of Worst Cases

```
IF (age>=75) THEN personal := 5980
ELSE IF (age>=65) THEN personal := 5720
     ELSE personal := 4335 FI FI;

IF (age>=65 AND income >16800) THEN
     IF (4335 > personal-((income-16800) / 2)) THEN personal := 4335
     ELSE personal := personal-((income-16800) / 2) FI FI;

IF (blind =1) THEN personal := personal + 1380 FI;

IF (married=1 AND age >=75) THEN pc10 := 6692
ELSE IF (married=1 AND age >= 65) THEN pc10 := 6625
     ELSE IF (married=1 OR widow=1) THEN pc10 := 3470
          ELSE pc10 := 1500 FI FI FI;

IF (married=1 AND age >= 65 AND income > 16800) THEN
    IF (3470 > pc10-(income-16800) / 2) THEN pc10 := 3470
    ELSE pc10 := pc10-((income-16800) / 2) FI FI;

IF (income - personal <= 0) THEN tax := 0
ELSE income := income - personal;
     IF (income <= pc10) THEN tax := income * rate10
     ELSE tax := pc10 * rate10;
          income := income - pc10;
          IF (income <= 28000) THEN tax := tax + income * rate23
          ELSE tax := tax + 28000 * rate23;
               income := income - 28000 ;
               tax := tax + income * rate40 FI FI FI;
```

Figure 9: UK Income Taxation Calculation Program in WSL

| | |
|---|---|
| `personal := 4335;`<br>`pc10 := 3470;`<br>`income := income - personal;`<br>`tax := pc10 * rate10;`<br>`income := income - pc10;`<br>`tax := tax + 28000 * rate23;`<br>`income := income - 28000;`<br>`tax := tax + income * rate40;` | `tax := 3470*rate10 + 28000*rate23 + (income-35805)*rate40;` |
| Syntax-Preserving Conditioned Slice | Amorphous Conditioned Slice |

Figure 10: The Combination of Syntax-Preserving Conditioned Slicing and Amorphous Slicing

preserving) static slicing to yield amorphous conditioned slicing.

We have implemented a prototype conditioned slicer for WSL. The conditioner uses the same approach to symbolic execution used by the ConSIT tool for conditioned slicing of C programs [4], but it uses the '`Simplify`' transformation of WSL to implement a (very light weight, though imprecise) theorem prover. The details of the implementation of this conditioned slicer are beyond the scope of this paper. However, its existence allowed us to experiment with the combination of amorphous slicing and conditioning. While our results are purely anecdotal at this stage, they indicate that this combination can be particularly productive.

For example, consider the UK tax calculation program in Figure 9, which is WSL version of the C program previously used in [4]. The program represents an attempt to capture UK tax regulations concerning the computation of amounts of tax payable, including allowances for a tax payer in the tax year April 1998 to April 1999. Each person has a personal allowance which is an amount of un-taxed income. The personal allowance depends upon the status of the person, reflected by the boolean variables `blind`, `married` and `widowed`, and the integer variable `age`. There are three tax bands, for which tax is charged at the rates of `rate10` (=10%), `rate23` (=23%) and `rate40` (=40%). The width of the 10% tax band is subject to the status of the person, while the 23% and 40% are fixed for all individuals.

Given condition {`age`>=65 `AND age`<75 `AND income`=36000 `AND blind`=0 `AND married`=1}, slicing the program on variable `tax`, using syntax-preserving conditioned slicing, produces the syntax-preserving conditioned slice in leftmost column of Figure 10. However, by replacing the syntax-preserving (static) slicer with the amorphous slicer described in this paper, we obtain the (much smaller) amorphous conditioned slice in rightmost column of Figure 10. In this way we believe that amorphous slicing may further improve the way in which conditioned slicing helps extract 'business rules' based upon a condition of interest.

## 6 Conclusion

This paper has described an interprocedural amorphous slicing system for WSL that mixes dependence reduction transformations and traditional slicing to achieve smaller slices at the interprocedural level.

The algorithm for interprocedural amorphous slicing has been implemented for the WSL language on top of the FermaT transformation workbench. The initial empirical results from executions of this implementation show that it scales to applications which work at unit level even in the worst case (which is somewhat pathological and likely to be rare).

## 7 Future work

Our approach was simply to push the intraprocedural analysis into the bodies of procedures and to iterate the slicing and dependence reduction in these bodies. As we have seen, even this simple-minded approach can produce good results, particularly when combined with conditioning. However, the focus of this paper has been to present the approach and to show that it scales sufficiently well to form a basis for future development. We have not attempted to argue rigorously that the quality of the slices produced justifies the approach. Rather, we hope to have convinced the reader that the approach is a good starting point for future work.

We are currently working to exploit constant propagation, loop unfolding and a collection of special case domain–specific transformations, aimed at producing increased simplification power. The hope is that by combining these more advanced features with our existing dependence reduction approach, we will achieve good results in a variety of specialised domains. For example, consider the program in Figure 11. This program is typified by use of constants in a procedure call, which is often found in machine generated code and code which uses libraries naïvely.

```
MW_PROC @LookUp(i,j,A,Element VAR Pos, Found) ==
VAR <Count:=i>:
Found := 0;
WHILE Count <=j DO
   IF A[Count] = Element
      THEN Pos := Count;
           Found := 1;
   FI;
   Count := Count +1
OD
ENDVAR.;

@LookUp(5,7,Array,Key,RetPost,RetFound) ;
IF Found=1
   THEN x:= ...
FI
```

Original

```
IF Array[5]=Key OR Array[6]=Key OR Array[7]=Key
   THEN x:= ...
FI
```

Ideal Interprocedural Amorphous slice on x

Figure 11: A Desirable Result for Future Work

The slicing algorithm we currently employ used intraprocedural constant propagation[4]. However, it does not propagate constants into procedure calls. Neither does it exploit constant loop bounds, by finite unfolding. For certain applications, heuristic techniques (like loop unfolding) can produce significant results. Future work will involve enhancements to our system to exploit these kinds of transformations to provide more powerful amorphous slicing in domain specific contexts.


## 8 Acknowledgements

We would like to thank Martin Ward for many helpful and interesting discussions about FermaT and the language WSL, and for providing these tools to the community under GPL. We would also like to thank Dave Binkley for many useful discussions and observations regarding amorphous slicing.


## References


[1] David Wendle Binkley. Computing amorphous program slices using dependence graphs and a data-flow model. In *ACM Symposium on Applied Computing*, pages 519–525, The Menger, San Antonio, Texas, U.S.A., 1999. ACM Press, New York, NY, USA.

[2] David Wendle Binkley and Keith Brian Gallagher. Program slicing. In Marvin Zelkowitz, editor, *Advances of Computing, Volume 43*, pages 1–50. Academic Press, 1996.

[3] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607. Elsevier Science B. V., 1998.

[4] Sebastian Danicic, Chris Fox, Mark Harman, and Rob Mark Hierons. ConSIT: A conditioned program slicer. In *IEEE International Conference on Software Maintenance (ICSM'00)*, pages 216–226, San Jose, California, USA, October 2000. IEEE Computer Society Press, Los Alamitos, California, USA.

[5] Andrea De Lucia. Program slicing: Methods and applications. In $1^{st}$ *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149, Florence, Italy, 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[6] Andrea De Lucia, Anna Rita Fasolino, and Malcolm Munro. Understanding function behaviours through program slicing. In $4^{th}$ *IEEE Workshop on Program Comprehension*, pages 9–18, Berlin, Germany, March 1996. IEEE Computer Society Press, Los Alamitos, California, USA.

[7] Michael D. Ernst. Practical fine–grained static slicing of optimised code. Technical Report MSR-TR-94-14, Microsoft research, Redmond, WA, July 1994.

[8] Mark Harman and Sebastian Danicic. Amorphous program slicing. In $5^{th}$ *IEEE International Workshop on Program Comprenhesion (IWPC'97)*, pages 70–79, Dearborn, Michigan, USA, May 1997. IEEE Computer Society Press, Los Alamitos, California, USA.

[9] Mark Harman and Robert Mark Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[10] Mark Harman, Lin Hu, Xingyuan Zhang, and Malcolm Munro. GUSTT: An amorphous slicing system which combines slicing and transformation. In


[4] This comes 'for free' as a by product of the dependence reduction transformation rules.

$1^{st}$ *Workshop on Analysis, Slicing, and Transformation (AST 2001)*, pages 271–280, Stuttgart, October 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[11] Frank Tip. *Generation of Program Analysis Tools.* PhD thesis, Centrum voor Wiskunde en Informatica, Amsterdam, 1995.

[12] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[13] Martin Ward. *Proving Program Refinements and Transformations.* DPhil Thesis, Oxford University, 1989.

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

[15] Martin Ward. The formal approach to source code analysis and manipulation. In $1^{st}$ *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 185–193, Florence, Italy, 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[16] Martin Ward. Program slicing via FermaT transformations. In $26^{th}$ *IEEE Annual Computer Software and Applications Conference (COMPSAC 2002)*, Oxford, UK, August 2002. IEEE Computer Society Press, Los Alamitos, California, USA. To Appear.

[17] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.