Restructuring Functions with Low Cohesion

Arun Lakhotia
The Center for Advanced Computer Studies
The University of Southwestern Louisiana
Lafayette, LA 70506, USA
arun@cacs.usl.edu

Jean-Christophe Deprez
The Center for Advanced Computer Studies
The University of Southwestern Louisiana
Lafayette, LA 70506, USA
jxd7803@cacs.usl.edu

Abstract

We present a technique for restructuring functions with low cohesion into functions with high cohesion. Such restructuring is desirable when re-architecting a legacy system into an object-oriented architecture. The restructured system has functions with higher cohesion and hence lower coupling. This enables finer-grained grouping of functions into objects.

Automatically decomposing a function is difficult when its computations are interleaved. The challenge lies in programmatically identifying and separating the various activities performed by a complex code segment. The technique presented partitions the set of output variables of a function on the basis of their pairwise cohesion. Program slicing is then used to identify the statements that perform computations for each variable group in the partition. New functions corresponding to the slices are created to replace the original function.

Experiences with restructuring real-world code using a tool that implements the technique are presented.

1. Introduction

Legacy software is a software that is hard to change and is still alive (operational), an indication that its users are still in business. The premise of this paper, and that of most work on software restructuring and reengineering, is that (a) to continue to stay in business the users (and, more so, the developers) realize that the structure of the legacy system needs to be overhauled such that it is easier to adapt to changes and (b) it is not pragmatic to redevelop or replace the system. This premise is supported by several opinion leaders of the software industry [8, 27, 42].

In this paper the terms software restructuring and reengineering are used as defined by Chikofsky and Cross [10]. "Software restructuring is the transformation [of software] from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics). Reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form."

A software system may be restructured to make it less costly to maintain by making it easier to understand and change [2]. Restructuring may also be the *enabling* step for reengineering a system [35, 40], and for reverse engineering a system to extract its abstraction [7, 16, 39].

Restructuring in the early days of structured programming implied removing the 'goto' statements [3, 4, 20]. This notion of restructuring is quite mature and has resulted into several automated tools, surveyed previously by Arnold [2]. Even though automatic removal of goto statements does not always produce programs that are desirable [9], such restructuring is a necessary step for creating higher, logic-based abstractions from code [7, 16, 39, 40].

Restructuring by removing goto statements is not the subject of this paper. This paper makes a contribution towards restucturing program fragments by breaking them into small, cohesive pieces. Decomposing functions with low cohesion into several functions with high cohesion is an important step of Stankovic's method of improving the architecture of a software system without compromising its performance [36]. Stankovic had proposed manual transformations for this step. The restructuring technique presented, developed without prior knowledge of Stankovic's method, provides a way to automate that step.

The need for decomposing large (usually non-cohesive) code fragments is not new. It has been experienced by almost every programmer since the early days of computers. The difficulty in decomposing large code fragments lies not so much in creating small functions, but in creating small functions that are meaningful. This is exemplified by the following story:

> In the late 1960s most data processing managers began to recognize the worth of modularity. Unfortunately many existing programs were monolithic, e.g., 20,000 lines of undocumented FORTRAN with one 2500 line subroutine. To bring his environment to the state of the art, a manager asked his staff to modularize such a program that underwent maintenance continuously. This was to be done "in your spare time."
>
> Under the gun, one staff member asked (innocently) the proper length for a module. "Seventy-five lines of code," came the reply. She then obtained a red pen and a ruler, measured the linear distance taken by 75 lines of source

```
1  Procedure Sale_Pay_Profit (days: integer;
                   cost: float; var sale: int_array;
                   var pay:  float; var profit: float;
                   process: boolean);
2   var i: integer;total_sale, total_pay: float;
3   begin
4     i:=0;
5     while i < days do begin
6          i := i + 1;
7          readln(sale[i])
8     end;
9     if process = True then begin
10      total_sale:=0;
11      total_pay:=0;
12      for i := 1 to days do begin
13          total_sale := total_sale + sale[i];
14          total_pay := total_pay + 0.1 * sale[i];
15          if sale[i] > 1000 then
16              total_pay := total_pay + 50;
17      end;
18      pay := total_pay / days + 100;
19      profit := 0.9 * total_sale - cost;
20    end;
21  end;
```

Figure 1 Sample non-cohesive code. This function uses the same input to compute different outputs. Its computation also depend on a flag passed as a parameter. This function is an example of code with *interleaved* computations [32].

> code, and drew a red line on the source listing, then another and another. Each red line indicated a module boundary. [31, page 334]

While this approach appears humorous, it does highlight the central problem in decomposing code fragments:

> How does one [programmatically] decide which set of statements may be extracted as independent units (such as functions, procedures)?

The approach used in the above anecdote is not satisfactory because it does not take into account whether there is any relationship between the computations performed by each 75 consecutive lines of code extracted as a FORTRAN subroutine.

The problem of restructuring functions into "separate [functions] which can be compiled and tested separately and which can be connected to other [functions] through a parameter interface" was first studied by Sneed and Jandrasics [35]. This problem has more recently been studied by Kim et al. [21] and Kang and Bieman [19]

We present a restructuring technique that, we argue, attempts to restructure a program using rules a good human programmer would use. These rules, referred to as the *rules of cohesion*, were arrived at by machine encoding [22, 29] the "associative principles" of cohesion discovered two decades ago by Steves et al. [38, 43][*]. These associative principles summarize decisions made by human designers in choosing between alternative design decompositions, and form a fundamental component of the Structured Design method of software development. The use of these principles in decomposing functions thus promises to create small, cohesive functions that perform a *natural* unit of activity.

The rest of the paper is organized as follows. Section 2 precisely formulates the restructuring problem studied in this paper. Section 3 summarizes Lakhotia and Nandigam measure for cohesion. Section 4 presents our restructuring technique. Section 5 gives an overview of its implementation in WolfPack. Section 6 presents the results of using this technique on benchmark programs and real-world code. Section 7 presents a comparison with other related works. Section 8 presents our conclusions.

2. Problem formulation

We now precisely describe the problem to be solved in this paper, and also differentiate the intended goals from our previous results.

The problem addressed in this paper may be stated as follows:

> Given a function that performs several activities, how can one "automatically" decompose that function into several functions, each performing only a single activity or a single set of related activities?

The word "activity" may take different meanings at different level of abstractions of a program. In this paper we consider the modification of an output variable as an activity. An output variable is any variable, reference parameter or global variable, modified by a function. Files, or more generally I/O streams, are also considered output variables by associating implicit global variables to them. Multiple modifications to the same output variable is considered a single activity.

Consider for example the program in Figure 1. The function reads the amount of `sale` per day, for a given number of `days` and computes (a) the `total_sale` for the period, (b) the commission to be paid, `pay`, as 10% of the sale, with an added bonus of $50 if the sale for a particular day is over $1,000, and (c) the `profit` for the whole period, given the `cost` at the beginning, as a percentage of sale. While the function is small, that it performs several activities makes it non-cohesive.

Figure 2 contains a program equivalent to that of Figure 1. Though it is larger in size, as measured in number of

[*] The reader is encouraged to review the references [28, 38, 43] to assess our use of the word "discovered." The notion of cohesion and the associative principles were actually identified by Stevens and Constantine by interviewing expert software designers.

```
Procedure Sale_Pay_Profit (days: integer;
                  cost: float; var sale: int_array;
                  var pay:  float; var profit:
float;
                    process: boolean);
begin
    Read_Input(days, sale);
    if process = True then begin
        pay := Compute_Avg_Pay(days, sale);
       profit := Compute_Profit(cost, sale);
    end;
end;

Procedure Read_Input(days:integer;
    var sale: int_array);
var i: integer;
begin
    i:=0;
    while i < days do begin
        i := i + 1;
        readln(sale[i]);
    end;
end;
```

```
Function  Compute_Pay(days: integer;
        sale: int_array): float;
var  total_pay: float;
       j: integer;
begin
    total_pay := 0;
    for j := 1 to days do
    begin
        total_pay := total_pay + 0.1 * sale[j];
        if sale[j] > 1000 then
            total_pay := total_pay + 50;
    end;
    return (total_pay);
end;

Function Compute_Sale(days: integer;
       sale: int_array): float;
var  total_sale: float;
      j: integer;
begin
    total_sale := 0;
    for j := 1 to days do
    begin
        total_sale := total_sale + sale[j];
    end;
    return (total_sale);
end;
```

```
Function Compute_Avg_Pay
    (days: integer; sale: int_array): float;
var total_pay: integer;
     pay: float;
begin
   total_pay := Compute_Pay(days, sale);
    pay := total_pay / days + 100;
    return (pay);
end;

Function Compute_Profit
    (cost: float; sale: int_array): float;
var  total_sale, profit: float;
begin
    total_sale := Compute_Sale(days, sale);
    profit := 0.9 * total_sale - cost;
    return (profit);
end;
```

Figure 2 Results expected from "automatically" restructuring program in Figure 1

lines, this program is cohesive. Instead of one monolithic function that performs several activities, it has several small functions, each performing a single activity.

This paper addresses the problem of "automatically" restructuring program of Figure 1 to the program of Figure 2.

In an earlier paper we introduced formal definitions and algorithms for a transformation called *tuck* that aids in the type of restructuring being discussed [24]. Besides the program being restructured, the tuck transformation takes two inputs: (a) a set of seed statements S and (b) a single-entry, single-entry (SESE) region called the *restructuring context*. The tuck transformation (1) identifies the code affecting the computation at seed statements S within the restructuring context, (2) moves this code into a new function, and (3) replaces the moved code with a call to the new function. A tuck transformation, if needed, may duplicate code to ensure that the behavior after tucking remains unchanged. The tuck transformation is performed as a sequence of three (smaller) transformations: *wedge*, *split*, and *fold*.

Transformation: *Wedge*. A wedge is a program slice [41] bounded within a given restructuring context. The wedge is computed for a set of seed statements S, similar to a slicing criterion.

Transformation: *Split*. The split transformation splits a restructuring context into two SESE regions, one containing all the computations relevant to a set of statements S and the other containing all the remaining computations. The transformation introduces new variables or renames variables and composes the two new regions such that the overall computation remains unchanged. When it is not feasible to split a region in such a way, the transformation leaves the region unchanged.

Transformation: *Fold*. The fold transformation creates a function for a given set of statements and replaces the statements by a call to this function.

The tuck transformations, and its sub-transformations, provide the theoretical foundation for splitting functions into smaller functions [24]. The application of the tuck transformation requires identifying its two parameters: the seed statements and the SESE region forming the restructuring context. These parameters may either be identified by a programmer or may be identified automatically, or a combination of the two. Automatic identification of these parameters, the subject of this current paper, can lead to *batch* tools for restructuring programs. On the other hand, having a programmer identify the parameters leads to *interactive* tools.

Our proposal for the DIME environment is oriented towards an interactive approach for software restructuring [23]. Using the DIME environment, currently under development, a programmer may interactively, using a mouse, identify the seed statements and the restructuring context, and the system would perform the tucking operation. The interactive environment gives better control to the programmer. However, it may not always be the best alternative when a major restructuring of a program is desired.

In this paper we focus on the problems related to developing a batch tool for restructuring. Though we do not

Table 1 Stevens *et al.*'s rules for cohesion. Associative principle between two processing elements and the corresponding cohesion in increasing order of levels [22].

Cohesion	*Associative principles*
Coincidental (Low)	None of the following associations hold.
Logical	At each invocation, one of them is executed.
Temporal	Both are executed within the same limited period of time during the execution of the system.
Procedural	Both are elements of some iteration or decision operation.
Communicational	Both reference the same input data set and/or produce the same output data set.
Sequential	The output of one serves as input for the other.
Functional (High)	Both contribute to a single specific function.

believe that it is possible to restructure a program without human intervention, the intent of this study is to look at the other extreme of the problem. This will give us insight into how much of the work can be deferred to the machine, thereby laying the foundation of a effective strategy for developing restructuring environments.

3. Computing cohesion

The restructuring technique presented in the next section requires computing cohesion between various output variables of a program. We use Lakhotia and Nandigam's rule-based measure for cohesion [22, 29]. Their measure is summarized in this section.

The degree of interrelation between activities performed by a code fragment is called *cohesion*, a term coined by researchers in IBM T. J. Watson Research Laboratories in the early 1970s [38]. These researchers also identified a set of "associative principles" used by systems analysts when evaluating alternative designs. These associative principles, summarized in Table 1, are ordered in seven levels. Designs demonstrating associations placed at higher levels are considered to be better (hence more preferred) than those placed at lower levels.

For instance, a procedure containing a computed `goto` (or a `case` statement) only one of whose branch is executed during any invocation of the procedure is considered to be poorly designed (with logical cohesion). The rationale being that the computation performed in different branches were not related to each other and could as well have been placed in different procedures. On the other hand, a procedure performing a sequence of computations, where the results of one are fed into the other is considered to have a better design, since there is a strong relationship between the computations. However, even such a procedure is not as good as one that performs just a single computation.

Table 2 Rules for computing pairwise cohesion between output variables.

i	*Cohesion* C_i	*Associative principles or Rules* $rule_i : Var \times Var \rightarrow Boolean$
1.	Coincidental	$\neg(\bigvee_{i \in \{2 \ldots 5\}} \quad rule_i(x, y))$
2.	Logical	$\exists znk \forall l. \quad z \rightarrow_{c(n,k)} x \wedge z \rightarrow_{c(n,\neg k)} y \wedge \neg(z \rightarrow_{c(n,l)} x \quad \wedge z \rightarrow_{c(n,l)} y)$
3.	Procedural	$\exists znk. \quad z \rightarrow_{c(n,k)} x \wedge z \rightarrow_{c(n,k)} y$
4.	Communicational	$\exists z. \forall nkl. \quad \neg(z \rightarrow_{c(n,k)} x \quad \wedge z \rightarrow_{c(n,\neg k)} y) \wedge \neg(z \rightarrow_{c(n,k)} x \quad \wedge z \rightarrow_{c(n,k)} y) \wedge ((z \rightarrow x \wedge z \rightarrow y) \quad \vee (x \rightarrow z \wedge y \rightarrow z))$
5.	Sequential	$x \rightarrow y \vee y \rightarrow x$

Lakhotia and Nandigam have translated Stevens *et al.'s* rules into an objective measure [22, 29]. They achieved this by analyzing the sources of ambiguity in the IBM's associative principles and choosing an interpretation that removed this ambiguity. They then translated these rules into formal logic. Finally, Nandigam fine-tuned the rules by experimenting with a large set of real-world programs [29]. This formalization, summarized in Table 2, consists of expressions denoting the data and control flow relationships in a program.

In Lakhotia and Nandigam's approach cohesion of a module is determined by first computing the pairwise cohesion between every pair of output variables of the program. Computation of pairwise cohesion requires computing control and data dependence relationships between variables. Notice that such dependence relationships are usually associated to statements, rather expressions. The dependence between variables are computed by simply *abstracting* the corresponding dependences between assignment statements of the variables.

The pairwise cohesion is computed as the highest cohesion level assigned to a pair of variables as per the rules in Table 2. In this table x and y denote output variables. The notation $x \rightarrow_{c(n,k)} y$ means that the variable y is defined in the kth branch of the branch statement n whose pred-

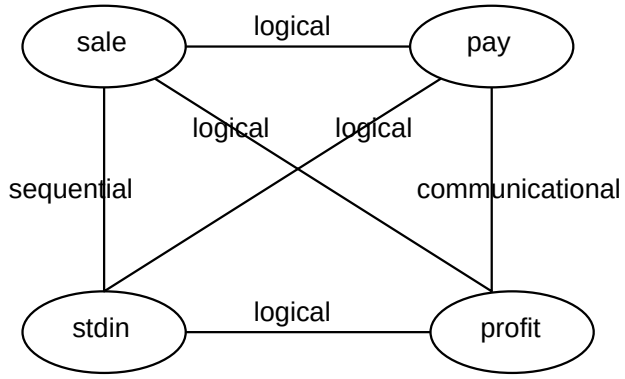

Figure 3 Pairwise cohesion graph for output variables of function in Figure 1

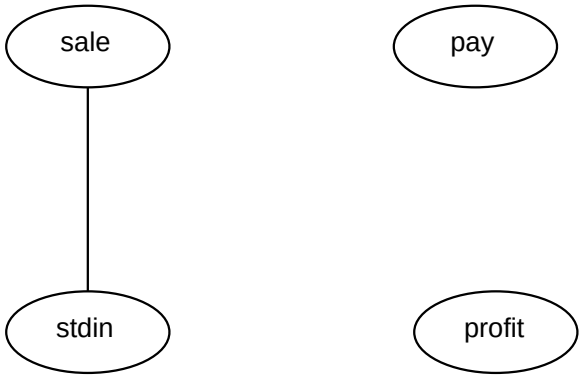

Figure 4 Cohesion graph after removing edges representing cohesion level below (and including) communicational cohesion

icate contains the variable x. The expression $x \rightarrow_{c(n,k)} \neg y$ is analogous except that the variable y is defined in a branch other than *k*. The notation $x \rightarrow_d y$ means that here is a def-use chain [1] from a statement defining x to a statement defining y. [22]

Lakhotia and Nandigam's approach of assessing cohesion has an important advantage over methods that assign numeric values [6, 30]. The rule-based approach does not just assign levels of cohesion to a module, it can also give the rationale behind that assignment. As a result we find Lakhotia and Nandigam's formalization is especially suitable for program restructuring.

4. Restructuring technique

The automated restructuring we desire may be performed by repeatedly tucking computations of a function into a new function. Each application of the tuck transformation requires two parameters, the seed statements and the restructuring context. Our restructuring problem may be therefore be reduced to that of automatically finding parameters to the successive tuck transformations.

A restructuring context for a tuck transformation is a SESE region containing all the seed statements. For any set of seed statements, there are as many alternatives for restructuring context as the SESE regions containing the seed statements. An automatic restructuring technique must therefore automatically choose one of these many restructuring contexts. In this study we simply choose the SESE region defined by the function entry as the restructuring context. The function entry is a SESE region for every possible seed statements and is also the largest SESE region. Though this simplistic choice does not always yield optimal results, it helps us narrow the focus further.

Having fixed the restructuring context our problem then is reduced to finding seed statements for tuck transformations. A method to do precisely that is the key technical contribution of this paper. We use the following steps to find the seed statements identifying computations to be tucked:

1. Identify all the output variables of the function.
2. Compute pairwise cohesion between these output variables using Lakhotia and Nandigam's rule-based measure for computing cohesion (Table 2). The pairwise cohesion can be represented by a completely connected graph.
3. Remove from the pairwise cohesion graph all edges representing cohesion below a given threshold level.
4. Partition the set of output variables based on the new graph. Each connected component of the new graph defines an equivalence class.
5. Each equivalence class of variables defines a single tuck transformation. All the assignment statements of the variables in an equivalence class form the parameter to tuck.

These steps are elaborated by applying them to the function (really, procedure) in Figure 1.

Our example function has four output variables `sale`, `pay`, `profit` and `stdin`. The first three are explicitly declared as reference parameters. The last one is the variable representing the input stream implicitly associated with the `readln` statement.

The graph in Figure 3 gives the pairwise cohesion between these variables. Variables `sale` and `stdin` have sequential cohesion because `sale` has data dependence on `stdin`. There is logical cohesion between `sale` and `pay` because `pay` is not computed if the value of `process` is `False`. For the same reason there is logical cohesion between `sale` and `profit`, between `stdin` and `pay`, and between `stdin` and `profit`. There is communicational cohesion between `pay` and `profit` because they both depend on a common variable `sale`.

```
Procedure Sale_Pay_Profit (days: integer;
                  cost: float; var sale: int_array;
                  var pay:  float; var profit: float;
                    process: boolean);
begin
  F_sale(days, sale);
  F_pay(pay, days, sale, process);
  F_profit(profit, cost,  days, sale, process);
 end;
Procedure F_sale(days:integer;
    var sale: int_array);
var i: integer;
begin
    i:=0;
    while i < days do begin
        i := i + 1;
        readln(sale[i]);
    end;
end;
```

```
Function  F_pay(var pay: float;
      days: integer;
        sale: int_array;
         process: boolean) ;
var  total_pay: float;
       j: integer;
begin
 If process = True then begin
    total_pay := 0;
    for j := 1 to days do
    begin
        total_pay := total_pay + 0.1 * sale[j];
        if sale[j] > 1000 then
            total_pay := total_pay + 50;
    end;
     pay := total_pay/days + 100
   end;
end;
```

```
Function F_profit(var profit: float;
      cost: float;
      days: integer;
       sale: int_array;
       process: boolean);
var  total_sale, profit: float;
      j: integer;
begin
 If process = True then begin
    total_sale := 0;
    for j := 1 to days do
    begin
        total_sale := total_sale + sale[j];
    end;
    total_sale := Compute_Sale(days, sale);
    profit := 0.9 * total_sale - cost;
   end;
 end;
```

Figure 5 Final program created by "automatically" restructuring the program in Figure 1

Figure 4 gives the graph resulting from removing all edges below (and including) communicational cohesion. The equivalence classes defined by the connected components of the resulting graph are: {`stdin`, `sale`}, {`pay`}, and {`profit`}. These classes define seed statements to three tuck transformation, as follows:

1. Statement `readlin(sale[i])`, since it assigns variables `stdin` and `sale`.
2. Statement `pay := total_pay/days +100`, since it assigns to variable `pay`.
3. Statement `profit := 0.9 * total_sale – cost`, since it assigns to variable `profit`.

The result of applying the three tuck transformations, see Figure 5, contains three new functions, one reads the input and computes `sale`, another computes `pay`, and the last computes `profit`. These three functions are individually more cohesive than the original function.

The function decomposition in Figure 5, the final decomposition, is not the same as that in Figure 2, the expected decomposition, the one we set out to achieve. The differences between the two give insight into the limitation of automated restructuring. There are two major differences between the two decompositions. First, the expected decomposition has more functions than the final decomposition. Second, the `if` statement controlling whether `pay` and `profit` are computed is in the highest level function in the expected decomposition, whereas in the final decomposition this statement is moved to lower level functions and is also duplicated.

It appears at first glance that the extra functions in the expected decomposition can be constructed by further restructuring the new functions created in the final decomposition. That, however, is not true. Our restructuring algorithm only separates computations related to output variables. Each of the two new functions have a single output variable, so they cannot be split any further. Thus, to split a function with only one output variable we should either adapt our current strategy or else invent a new one.

In the final decomposition, the `if` statement has been moved to the new function because we define the function entry as the restructuring context. To keep the `if` statement at the topmost level, the restructuring context should be the body of the `if` statement. This could be achieved if the restructuring context was limited to the smallest SESE region containing the seed statements. For structured programs, this smallest region would be the same as the nearest common ancestor of all the seed statements in the abstract syntax tree. While that alternative may yield the desired result for the example function chosen, it is not guaranteed to behave as desired for all functions.

Instead of using communicational cohesion as the threshold level if we use procedural cohesion as the threshold we get the following equivalence classes: {`stdin`, `sale`}, {`pay`, `profit`}. These define seed statements for two tuck transformations. The first transformation is the same as earlier. The seed statements of the second transformation will consist of two statements, the assignments to `pay` and `profit`. The result contains a function that computes both `pay` and `profit`. This is still more cohesive than the original function, but it is less cohesive than the previous result.

If a search for the most cohesive function is the purpose, then one would set the threshold level to functional cohesion. We then get four equivalence classes, one for each output variable. In this case however the tuck transformation is not feasible. Reason, `sale` cannot be computed without performing `readln` so it must be in the function defining `sale`. In addition `stdin` may be computed in a separate function by just performing `readln`. This will lead to duplication of `readln`, something that cannot be permitted since `stdin` is an implicit global variable.

5. Implementation

We have developed a system called `WolfPack`* that implements the proposed restructuring approach. WolfPack is developed using CBMS, formerly called Software Refinery, from Reasoning, Inc.†. It restructures C programs.

The WolfPack system consists of the following significant modules:

Module 1. Convert source code to internal format (control-flow graph, abstract syntax tree).
Module 2. Compute data and control dependence relations on statements.
Module 3. Abstract data and control dependence relations for variables.
Module 4. Compute pairwise cohesion graph.
Module 5. Identify seed statements for tucking.
Module 6. Tuck computation related to a set of seed statements into separate functions.
Module 7. Generate source code.

The tuck transformation, Module 6, operates on abstract syntax tree (AST) of the original function and creates ASTs for the new functions. Module 6 prints these ASTs into format.

The system, as implemented, is quite limited in that it does not perform alias analysis. Hence the results it produces are unsafe. It does, however, provide a reasonable platform to experiment with restructuring programs that do not have aliases.

6. Empirical observations

We have used WolfPack to restructure over 100 small and large functions of programs from the public domain and those of our sponsors. The largest program we experimented with is of 35,000 lines. This section summarizes our observations.

The set of functions we analyzed may be classified into three categories:

1. Classic
2. Real
3. Special.

The classic category contains functions handcrafted to represent classic textbook examples used for demonstrating a particular type of cohesion, and to represent classic examples used by other researchers to demonstrate the application of slicing for decomposition. These functions represent the best case scenario for the type of restructuring discussed. One expects the technique to perform well at least on these functions.

The real category contains functions extracted from actual programs, taken from third-party sources, such as student programs, textbooks, public domain, and our sponsors. These programs are real in that they were not handcrafted by us.

The special category contains functions handcrafted to represent complex control flows, with goto-statements branching in and out structured programming constructs. These were constructed to verify the correctness of the slicing algorithm and to check the correctness of the algorithm to construct new abstract syntax trees.

WolfPack worked perfectly with the classic and special programs. Since we used these programs to iteratively improve our algorithm, this performance was expected.

In real programs we found WolfPack to have two shortcomings. First, its analysis, being unsafe, did not give correct results. Second, the cohesion rules did not work well when a `struct` like data structure was implemented as a collection of independent variables.

The first shortcoming was not a surprise because WolfPack system was designed as a prototype to demonstrate proof-of-concept. In the role of a prototype it did help us evaluate our strategy for decomposing a function. We found that except for situations captured in the second shortcoming, our method did work extremely well in partitioning output variables. The grouping of variables proposed by our algorithm at various threshold level appeared quite "intelligent."

For instance, the public domain system `sc`, a 10,000 lines C spread sheet program, contains a function called `update`. This function is 142 lines long and has seven output variables: `stcol`, `FullUpdate`, `strow`, `curcol`, `line`, `currow`, and `linelim`. The names of these variables indicate that variables `stcol` and `curcol` are related, `strow` and `currow` are related, and `line` and `linelim` are related. The pairwise cohesion algorithm indeed found sequential cohesion between the first two pairs of variables. It indicated that variables `line` and `linelim` had procedural cohesion, i.e., they were both being modified in the same loop, but did not have any control and data dependence. The algorithm also found that `FullUpdate` had sequential cohesion with `stcol` and `curcol`.

Inspection of the 142 lines of code indeed validated the strength of the relationships as proposed by the system.

* Information on WolfPack is available on-line at `http://www.cacs.usl.edu/~arun/Wolf`,

† Software Refinery and CBMS are trademarks of Reasoning, Inc.

That is, we found that the code segments using `stcol` and `curcol` were quite isolated from the code segments using `strow` and `currow`. It was further surprising because these pairs of variables appeared together in not just one but several *interleaved* code segments. For the purpose of extracting their computation, this grouping was perfect. Similarly, the low cohesion between `line` and `linelim` was also perfect, in spite of similarity in their names. These are, in a relative sense, unrelated variables. Variable `line` was used to compute expression values before placing them into the matrix containing the contents of the spreadsheet. Whereas `linelim` was a flag used to track if certain limit in the matrix was reached. While they are both related the matrix, `linelim` did not impose any limit on `line`. The variable `FullUpdate` was also a flag. It tracked whether the matrix had been changed. It was related to `stcol` and `curcol` because the flag was used to check if the matrix had changed in the context of processing rows. This happens to be a coincidence, for the programmer could as well have checked the flag in the context of processing columns.

In a nutshell, the grouping proposed by the pairwise algorithm matched very well what we, as programmers, would have done.

The system did not always function that well. The one scenario that it invariably failed in indicates a weakness in our method of assigning cohesion, as also of that of others. In a 1000 line program implemented by a graduate student as part of his doctoral work we found that instead of grouping two related variables as `struct`, the author had chosen to keep them as separate parameters. (To visualize, consider an implementation of stack with two variables: `buffer` an array to keep the data and `sp` an index into the array representing the top of stack. The two variables are kept separate, and are not collected into a single unit using `struct`. These variables are passed as parameters to any function operating on stack.) There were functions that modified the two parameters without causing any control or data dependence. As a result, our system indicated that these variables had coincidental cohesion. In the absence of any design knowledge, we would make the same assertion. However, in the context of the additional piece of information treating the two variables as coincidental is definitely incorrect.

The largest program we analyzed, about 30,000 lines, tested the limit of our implementation. Processing programs of this size became unwieldy because of the time and space required to perform data and control flow analysis. That we needed to retain the abstract syntax tree further added to the memory requirement. Which was made worse because CBMS does not provide any (easy) method to delete unneeded intermediate structures, such as the control flow graphs. It was a challenge to just compute all the information of this program. The first time we tried we exhausted 40 MB of swap space after having run the program for a whole day. The second time we exhausted patience. Thus, when experimenting with `sc` we manually located the large code fragments and placed them into separate files, before performing any analysis.

This difficulty with processing large programs is not any indication of the limits of the proposed method or that of performing dataflow analysis. It is primarily an indicator of poor design, something that can be overcome by a better engineered product. For instance, with CBMS we were constrained to keep all the intermediate structures in memory. A better design would be to keep the intermediate structures in files. This would reduce the load on the primary memory, therefore reducing the time spent in managing swap space.

7. Related works

We now compare the work presented with other efforts in (a) restructuring functions [19, 21, 35], (b) using program slicing for decompositions [13, 12, 11, 14, 15], (c) identifying computations that are interleaved [34, 33, 32], (d) and the inverse problem of function composition or integration [17, 25].

Sneed and Jandrasics have presented a technique that uses the control flow of a COBOL program to identify code segments that can be converted into modules [35]. For instance, they create a module for a loop or a section containing more than 200 statements. In the absence of any cue from control statements, they propose breaking off continuous blocks of 800 statements into separate modules. Since a statement is placed in at most one module, their approach does not lead to any duplication of code. On the other hand, the modules this approach creates are also not guaranteed to contain code that performs related computations.

The restructuring techniques of Kim *et al.* [21] and Kang and Bieman [19] is closest to our work. They both use the cohesion (though Kim *et al.* call it coupling) between output variables of a function to identify computations that may be extracted into separate functions and then use program slicing to extract the needed statements. The differences lie in how each technique measures cohesion, how it uses this measure to group related variables, and the class of programs for which the technique is safe (i.e., does not produce incorrect results), and the class for which it produces correct results.

Kim *et al.*'s [21] definition of cohesion has two major weaknesses. First, it appears to split coincidental and logical cohesion into procedural, communicational, and functional cohesion. Second, their cohesion rules are not robust. The cohesion assigned by their rules changes by

slight perturbation of the program, such as introducing assignment statements that simply copy value of variables without changing the program. Kim *et al.* define dependence between output variable as a result of the direct dependence between assignment statements of that variable. They do not account for flow due to a chain of indirect assignments. Thus, output variables that depend on each other may be considered as independent of each other. Furthermore, unlike tuck, their transformation is not formally defined and does not always preserve original behavior. For instance, it introduces calls to the new function in any order without taking into account the data dependences between them.

Kang and Bieman's method for partitioning variables is very close to ours [19]. Their measure is based on Lakhotia's original measure [22], though using a different set of dependence relations between variables. To compute pairwise cohesion we use data and control dependence between *all* the variables whether input, output, or local. In contrast, Kang and Bieman use data and control dependence between only the input and output variables. The information lost due to this abstraction is significant since it leads to unsafe restructuring when two output variables have sequential cohesion, but whose assignments cannot be reordered. They too do not provide a formal transformation for actually extracting a function, thus leading to similar errors as Kim *et al*.

Lanubile and Visaggio's work on extracting reusable functions from program flow graphs solves one part of the problem discussed here [26]. They extract a reusable function without changing the original program. In order to identify a reusable function they require a partial specification of the program's input and output. They too use program slicing, albeit bounded within a region of the flow graph, to extract the reusable function.

Gallagher and Lyle's [13] lattice of decomposition slices is useful in identifying changes that may ripple through to other computations. Their lattice orders the slices of output variables using the proper subset relation, which they refer to as *strong dependence*. That two slices do not have strong dependence does not imply that they have no dependence. The intersection of the slices may still be non-empty, a relation that Gallagher and Lyle call *weak dependence*. Weak dependence between the slices of output variables plays an important role when splitting functions. But this relation is not captured in the decomposition slice. Hence, restructuring based on decomposition slice may yield unsafe results.

Rugaber *et al.* [33] have investigated the problem of detecting "interleaved" computation, where interleaving is defined as "the merging of two or more distinct plans within some contiguous textual area of a program." A plan is a "computational structure to achieve some purpose or goal." A program plan is not necessarily the same as our notion of a program's computation represented by the computation of a variable. Yet Rugaber *et al.* observe that if a subroutine (function) has multiple outputs there is a high likelihood that it has interleaved computation. They report that 25% of subprograms in a library of 600 Fortran programs had multiple outputs. While Rugaber *et al.* have investigated the issue of detecting interleaved computation, they have not investigated how the subprograms may be restructured to reduce or eliminate the interleaving.

Restructuring a function by decomposing it is inversely related to the problem of creating a function by composing smaller functions. This inverse problem has been under the names of program integration [18], program composition [25, 37], and program merging [5].

8. Conclusions

The main contribution of this paper is a method for *automatically* restructuring functions with low cohesion by splitting them into smaller functions with higher cohesion. The method uses Lakhotia and Nandigam measure for computing cohesion [22, 29] to identify computations that can be extracted from a function. It uses program tucking to actually transform the program [24]. This transformation moves into a new function code that affects a set of statements and replaces the moved statement by a call to the new function.

The method proposed may be used to automatically restructure a program. Though good restructuring without human intervention is most likely not feasible, the authors have investigated this problem to gain insights into the limits of machine capability. This work lays the foundation for sophisticated interactive restructuring tools using which a programmer may restructure programs by simply choosing between alternatives provided by the machine.

9. Acknowledgments

Leverage Technologists, Inc., Bethesda, MD helped with implementing code to create abstract syntax trees of the new functions from that of the original function. The work was partially supported by a contract from the Department of Defense and a grant from the Department of Army, US Army Research Office. The contents of the paper do not necessarily reflect the position or the policy of the funding agencies, and no official endorsement should be inferred.

10. References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Robert S. Arnold. Software restructuring. *Proc. IEEE*, 77(4):607–617, April 1989.

[3] E. Aschroft and Z. Manna. The translation of 'goto' programs to 'while' programs. In *Proceedings of the 1971 IFIP Congress*, pages 250–260, Amsterdam, The Netherlands, 1971. North-Holland.

[4] Brenda S. Baker. An algorithm for structuring flowgraphs. *Journal of the ACM*, 24(1):98–120, January 1977.

[5] Valdis Berzins. On merging software extensions. *Acta Informatica*, 23:607–619, 1986.

[6] James M. Bieman and Linda M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):476–493, June 1994.

[7] Peter T. Breuer and Kevin Lano. Creating specifications from code; reverse-engineering techniques. *Journal of Software Maintenance: Research and Practice*, 3:145–162, 1991.

[8] Eric Bush. A CASE for existing systems. Technical report, Language Technology, Salem, MA, 1989.

[9] Frank W. Calliss. Problems with automatic restructurers. *SIGPLAN Notices*, 23:13–21, March 1988.

[10] Elliot J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[11] Keith Gallagher. Visual impact analysis. In *International Conference on Software Maintenance*, 1996.

[12] Keith B. Gallagher. Evaluating the surgeon's assistant: Results of a pilot study. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 236–244, November 1992.

[13] Keith B. Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[14] Rajiv J. Gopal and Stephen R. Schach. Using automatic program decomposition techniques in software maintenance. In *Proceedings of Conference on Software Maintenance*, pages 132–141, 1989.

[15] Rajiv J. Gopal and Stephen R. Schach. Application of automatic decomposition schemes in proof maintenance for evolving programs,. *Journal Software Maintenance: Research and Practice*, 4:183–198, December 1992.

[16] Philip A. Hausler, Mark G. Pleszkoch, Richard C. Linger, and Alan R. Hevner. Using function abstraction to understand program behaviour. *IEEE Software*, 7(1):55–65, January 1990.

[17] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating non-interfering versions of programs. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego*, pages 133–145, 1988.

[18] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[19] Byung-Kyoo Kang and James Bieman. Using design cohesion to visualize, quantify, and restructure software. In *Eighth International Conference on Software Engineering and Knowledge Engineering (SEKE'96)*, pages 222–229, Skokie, IL, June 1996. Knowledge Systems Institute.

[20] Takumi Kasai. Translatability of flowcharts into while programs. *Journal of Computer and System Sciences*, 9:177–195, 1974.

[21] Hyeon Soo Kim, In Sang Chung, and Yong Rae Kwon. Restructuring programs through program slicing. *International Journal of Software Engineering and Knowledge Engineering*, 4(3):349–368, September 1994.

[22] Arun Lakhotia. Rule-based approach to computing module cohesion. In *Proceedings of 15th International Conference on Software Engineering*, pages 35–44, Los Alamitos, CA, May 1993. IEEE Computer Society Press.

[23] Arun Lakhotia. DIME: A direct manipulation environment for evolutionary development of software. In *Proceedings of the International Workshop on Program Comprehension (IWPC'98)*, pages 72–79, Los Alamitos, CA, June 1998. IEEE Computer Society Press.

[24] Arun Lakhotia and Jean-Christophe Deprez. Restructuring programs by tucking statements into functions. *Journal of Information and Software technology*, 40(11-12):677–689, November 1998.

[25] Arun Lakhotia and Leon S. Sterling. Composing recursive logic programs with clausal join. *New Generation Computing*, 6(2):211–225, 1988.

[26] Filippo Lanubile and Giuseppe Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Transactions on Software Engineering*, 23(4):246–258, April 1997.

[27] Carma McClure. *The Three Rs of Software Automation: Reeingineering, Repository, and Reusability*.

Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1992.

[28] Glenford J. Myers. *Composite/Structured Design*. Van Nostrand Reinhold Company, New York, NY, 1978.

[29] Jagadeesh Nandigam. *A measure for module cohesion*. PhD thesis, University of Southwestern Louisiana, The Center for Advanced Computer Studies, Lafayette, Louisiana, 1995.

[30] Linda Ott and James Bieman. Program slices as an abstraction for cohesion measurement. *Journal of Information and Software technology*, 40(11-12):691–699, November 1998.

[31] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw Hill, New York, NY, third edition, 1992.

[32] Spencer Rugaber, Kurt Stirewalt, and Linda Wills. Understanding interleaved code. *Automated Software Engineering*, 3(1-2):47–76, June 1996.

[33] Spencer Rugaber, Kurt Stirewalt, and Linda M. Wills. Detecting interleaving. In *Proceedings of the International Conference on Software Maintenance*, pages 265–274, Los Alamitos, CA, 1995. IEEE Computer Society Press.

[34] Spencer Rugaber, Kurt Stirewalt, and Linda M. Wills. The interleaving problem in program understanding. In *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 166–175, Los Alamitos, CA, July 1995. IEEE Computer Society Press.

[35] Harry M. Sneed and Gabor Jandrasics. Software recycling. In *Proceedings of the Conference on Software Maintenance*, pages 82–90, Los Alamitos, CA, 1987. IEEE Computer Society Press.

[36] John A. Stankovic. Good system structure features: Their complexity and execution time cost. *IEEE Transactions on Software Engineering*, SE-8(4):306–318, July 1982.

[37] Leon Sterling, Ashish Jain, and Marc Kirschenbaum. Composition based on skeletons and techniques. In *ILPS '93 Post Conference Workshop on* Methodologies for Composing Logic Programs, Vancouver, October 1993.

[38] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[39] Martin Ward. Abstracting a specification from code. *Journal of Software Maintenance: Research and Practice*, 5:101–122, 1993.

[40] Richard C. Waters. Program translation via abstraction and reimplementation. *IEEE Transactions on Software Engineering*, 14(8):1207–1228, August 1988.

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

[42] Edward Yourdon. *Decline and Fall of the American Programmer*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992.

[43] Edward Yourdon and Larry L. Constantine. *Structured Design*. Yourdon Press, 1978.