Restructuring programs by tucking statements into functions

Arun Lakhotia and Jean-Christophe Deprez

The Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504
(318) 482-6766, -5791 (Fax)
arun@cacs.usl.edu

Abstract

Changing the internal structure of a program without changing its behavior is called restructuring. This paper presents a transformation called *tuck* for restructuring programs by decomposing large functions into small functions. Tuck consists of three steps: Wedge, Split, and Fold. A wedge—a subset of statements in a slice—contains computations that are related and that may create a meaningful function. The statements in a wedge are split from the rest of the code and folded into a new function. A call to the new function is placed in the now restructured function. That tuck does not alter the behavior of the original function follows from the semantic preserving properties of a slice.

Keywords: Program Restructuring; Program Slicing

1 Introduction

Software restructuring is the transformation of software from one representation to another at the same relative abstraction level, without changing the external behavior of the subject system [CC90]. A software system may be restructured to make it easier to understand and change, and therefore less costly to maintain [Arn89]. Restructuring may also be the *enabling* step for reengineering a system [SJ87, Wat88], and for reverse engineering a system to extract its abstractions [BL91, HPLH90, War93].

Restructuring in the early days of structured programming implied removing the **goto** statements [AM71, Bak77, Kas74]. This notion of restructuring is quite mature and has led to several automated tools [Arn89]. Even though automatic removal of **goto** statements does not always produce programs that are desirable [Cal88], such restructuring is a necessary step for creating higher, logic-based abstractions from code [BL91, HPLH90, War93, Wat88].

This paper investigates the problem of restructuring programs by breaking its large code fragments and tucking them into new functions. The technical challenge in creating new functions lies in *capturing computations that are meaningfully related*. If that was not necessary, one could simply create functions by breaking off contiguous pieces of code of some pre-set size, such as done by Sneed and Jandrasics [SJ87]. Such a straightforward approach may not yield good functions because of the interleaving of unrelated computations in real-world code [RSW96].

We present a restructuring transformation *tuck* to decompose large, non-cohesive code fragments into small, cohesive functions [Dep97]. To *tuck*, according to the American Heritage Dictionary, is to “gather and fold.” This is precisely what our transformation does. Tuck is a composition of three primitive

```
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 function. This function uses the same input to compute different outputs. Its computation also depends on a flag passed as a parameter. This function is a representative of code with *interleaved* computations [RSW96]. In Section 6, using the tuck transformation, this program is restructured into a collection of functions with an object-based architecture.

```
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;
12       for i := 1 to days do begin
13           total_sale := total_sale + sale[i];
17        end;
          ComputeTotalPay(days, sale, total_pay);
18        pay := total_pay / days + 100;
19        profit := 0.9 * total_sale - cost;
20     end;
21  end;
```

```
  Procedure ComputeTotalPay (days: integer;
                   sale: int_array;
                   var total_pay:  float;);
  var i: integer;
   begin
11       total_pay:=0;
12       for i := 1 to days do begin
14           total_pay := total_pay + 0.1 * sale[i];
15           if sale[i] > 1000 then
16               total_pay := total_pay + 50;
17        end;
     end;
```

Figure 2 Result of tucking computation in Figure 1. This program is created by tucking all statements assigning a value to the variable `total_pay`.

transformations: Wedge, Split, and Fold. To tuck a code fragment, a programmer first gatheres related code by driving a wedge in the function, then splits the code isolated by the wedge, and then folds [BD77] the split code into a function. Such restructuring may be performed in order to improve the architecture of a software system [Sta82].

Figures 1 and 2 enumerate by example the type of restructuring performed by tucking. The program in Figure 1 is not cohesive [SMC74] in that it performs several activities at the same time. It inputs the `sale` data for a given number of `days`. Depending on the value of the flag `process`, it also computes `pay`, the commission to be paid as a percentage of `sale`, and the resulting `profit`. Figure 2 contains a program resulting from tucking all the statements modifying the variable `total_pay` into a function `ComputeTotalPay`. Besides the assignments to `total_pay`, the new function also contains the `for` and the `if` statements so that the computation of `total_pay` is preserved. A copy of the `for` statement has been retained in the restructured function to ensure that `total_sale` is correctly computed.

The tuck transformation improves upon the `extract-function` transformation contained in Griswold and Notkin's catalogue of restructuring transformations [Gri91, GN93]. Griswold and Notkin's `extract-function` creates a new function from *contiguous* code fragments. Their transformation is limited to *structured* programs. In contrast, the tuck transformation even creates functions from *non-contiguous* code, as enumerated by the example in Figures 1 and 2. Our transformation is defined for *unstructured* programs as well.

This paper presents a summary of the tuck transformations and its application. The rest of this paper is organized as follows. Section 2 defines the problem and highlights the technical challenge in decomposing functions to create smaller functions. Section 3 presents some background definitions used later. Section 4 presents our tranformation for folding a contiguous piece of code into a function. Section 5 contains our transformation for tucking non-contiguous code. Section 6 gives an example of using this transformation for restructuring a program. Section 7 presents a comparison of our work with other related work. Section 8 contains our concluding remarks and plans for future research.

2 Problem definition

We define the problem of tucking statements as follows:

Definition: *Tuck.* To tuck a set of statements S of a function f_{old} is to create two functions f_{new} and f_S such that (a) f_{old} is equivalent to f_{new}, (b) f_{new} calls f_S, and (c) f_S contains the statements in S (and may be other statements).

For this discussion we only consider procedural programs that do not contain global variables and I/O statements. A program is made up of functions consisting of statements, both structured and unstructured. Data is passed back and forth between functions through value and reference parameters. We consider two functions to be equivalent if they produce the same outputs for the same inputs, where the input and output of a function are defined in terms of its parameters and return values.

When restructuring programs function f_{new} will replace function f_{old}. Hence the two functions, besides being equivalent in input/output mapping, will also have the same name. We use the subscript *new* and *old* to differentiate these two functions. It may not always be possible to tuck a given set of statements and ensure that f_{new} is equivalent to f_{old}. In such case, we assume that tuck terminates with error, i.e., without making any changes to the program.

The definition of tuck does not state whether statements in S should be included in f_{new} or not. Generally, these statements would not be included in f_{new}. However, requiring that statements in S not be included in f_{new} may be too strong a constraint. Sometimes it may be necessary to retain some statements of S in f_{new} in order to ensure that the three conditions are satisfied. In such case, the relevant statements are duplicated in both f_{new} and f_S.

The function f_S may also contain statements other than those in S. For instance, when the statements in S are dispersed throughout the code, interspersed with other statements on which they have control and data dependence. In such case, f_S may contain other statements that affect the computations at the statements in S. These statements may further be contained in f_{new} as well.

Clearly, f_{new} and f_S may also contain statements that are not in f_{old}. One such statement is the call to f_S from f_{new}, needed by condition (b) in the definition of tuck. The definition does not preclude the possibility of including in f_S or f_{new} statements that are not contained in f_{old}. Since statements from f_{old} may be copied in *both* f_{new} and f_S, some new statements may be necessary to ensure that f_{new} is equivalent to f_{old} in spite of the duplication of statements. One use of these additional statements may be to save and restore values of certain variables before or after the call to f_S.

While the definition of tuck does not require that all the statements of f_{old} appear at least once in f_{new} or f_S, if we assume that f_{old} contains only the statements that contribute to its computation, i.e. does not contain redundant statements, then it is pragmatic to require this condition. Thus, we expect that after tucking, every (useful) statement of f_{old} appears at least once in f_{new} or f_S.

Definition: *Original and copy statements.* Let statement s', a statement of either f_{new} or f_{sub}, be a copy of a statement s in f_{old}. Statement s is called the *original* of s' and s' is called a *copy* of s.

Structural constraint: While not required from the definition, our tucking algorithm ensures that every statement of f_{new} or f_{old} has zero or one original statement and every statement of f_{old} has one or two copy statements.

To formalize the behavior preserved by tuck we use Horwitz et al.'s operational semantics of programs, called HPR semantics, to characterize the behavior of a statement [HPR89]. Given an execution of a function—f_{old} or f_{new}—with some initial state σ, the behavior of a statement is defined as the sequence of values generated at that statement. For an assignment statement it is the sequence of values assigned to the variable on the left-hand side. For a predicate statement—such as, if-then-else, while-do—it is the sequence of boolean values to which its predicate evaluates. For a function call, it is the sequence of tuples consisting of the return value and the values of its parameters at completion of the function.

Let s be a statement of f_{old} and s' be one of its copy in f_{new} or f_S. Let $\mathcal{I}(f_{old})(s)(\sigma)$ denote the sequence of values generated by s for an execution of f_{old} with the initial state σ. Let $\mathcal{I}(f_{new})(s')(\sigma)$ denote the sequence of values generated by s' for an execution of f_{new} with the initial state σ. (Note that the initial state is for f_{new} even if s' is in f_S.)

Definition: *Statement equivalence.* A statement s of f_{old} is equivalent to its copy s' in f_{new} or f_S iff for every initial state σ for which f_{old} and f_{new} terminate, s and s' generate the same sequence of values, i.e. $\mathcal{I}(f_{old})(s)(\sigma) = \mathcal{I}(f_{new})(s')(\sigma)$.

It follows that the functions f_{old} and f_{new} are equivalent if all the statements of f_{old} are equivalent to their copies in f_{new} or f_S.

3 Preliminaries

Our discussions and algorithms are restricted to programs in a procedural language without global variables. The language contains assignment statement, branch statements, **goto** statement, and function call statement. For simplicity of presentation we consider a function call to be a statement, i.e., it does not appear in any expression. A function has a fixed number of parameters, each either passed by value or by reference.

We consider a function to be represented as a control flow graph (CFG) and a program as a collection of CFGs. The input and output of our transformations are CFGs, its components, and some relations over these components.

Definition: *Directed graph.* A directed graph $G = (N, E)$ consists of a set of nodes N and a set of edges $E \subseteq N \times N$.

Definition: *Control flow graph.* A CFG is $G =(N, E)$ is a directed graph with a unique *start* node $\varepsilon_G \in N$ such that there is a path from ε_G to every node in N, and a unique *end* node $\chi_G \in N$ such that there is a path to χ_G from every node in N. The edges of a CFG are annotated **T**, **F**, or **Always**, as described below.

The nodes of a CFG G, except nodes ε_N and χ_N, represent statements of the function. Conversely, the statements of a function, except **goto** statements, are represented as nodes in its CFG. The **goto** statements are represented as CFG edges. Henceforth, the term *statement* refers to a *node representing a statement* in a CFG. A branch statement has two outgoing edges, one annotated with **T** and the other with **F**. An assignment statement and a function call statement has only one outgoing edge, which for the sake of uniformity, is labelled **Always**. A CFG edge with tag **T** or **F** is called a *conditional branch*. As is the convention when control dependences are computed, we treat the start node as a branch node with an **F** edge to the end node χ_G and a **T** edge to the first statement of the function. A start node has no incoming edge and an end node as no outgoing edge.

Due to the label on its edges, a CFG is not truly a directed graph. When the labels on an edge do not play any significant role in an operation we omit the label and treat the CFG as a directed graph. This leads to concise expression of relations without losing the generality because the label on an edge is obvious from the context. When the label is important, we treat an edge as 2–tuple, as described below.

Notation: The pair (n_1, c), where n_1 is a statement and $c \in$ {**T**, **F**, **Always**}, represents a CFG edge tagged c starting from the node n_1. The tuple represents a unique CFG edge because there is only one CFG edge with a particular tag starting from a node.

Notation: We use the function Target((n_1, c)) to give the end statement of the edge (n_1, c).

The value and reference parameters at a call statement and a function are modelled as follows.

Definition: Let G be a flowgraph of a function. **Ref**(G) and **Value**(G) give the set of *reference* and *value* parameters, respectively, of G; **Local**(G) gives the set of local variables of G; and **Vars**(G) = **Ref**(G) $\cup$ **Value**(G) $\cup$ **Local**(G) gives all the variables of G.

Definition: Let c be a function call statement in the CFG G. **Calls**(G, c) gives the CFG of the function called by statement c. **Ref**(G, c) and **Value**(G, c) give the set of reference and value parameters for the call site c.

For simplicity and without loss of generality, we assume that only variables are passed as actual parameters in a function call and that at any call statement a variable may occur at most once as an

actual parameter*. We also assume that there is a 1–1 mapping between the reference (similarly, value) parameters of a call site and the reference (value) parameters of the function it calls. As a result the ordering of the parameters is not relevant.

Definition: *Postdominator.* A CFG node w *postdominates* another CFG node v, $v \neq w$, in the CFG G iff every path from node v to node t contains node w. Node w is the *immediate postdominator* of node v iff every other postdominator of v also postdominates w. Let ipdom(v) give the immediate postdominator of a node v.

Definition: *Control dependence.* [FOW87] A node n_j is *control dependent* on an edge (n_i, c) iff

1. n_j postdominates Target((n_i, c)) and
2. if $n_j \neq n_i$ then n_j does not postdominate n_i.

Notation: (n_1, c) $\in$ CD(G, n_2) iff in the CFG G the node n_2 is control dependent on the conditional branch (n_1, c) and $n_1 \neq n_2$.

Definition: *Data dependence.* A statement n_j is *data dependent* on a statement n_i in function G iff (a) there exists a variable v that is used by n_j and is defined by n_i and (b) in the CFG of G there exists a path from n_i to n_j in which the variable v is not defined by any intermediate node.

Notation: A statement $n_1 \in$ DD(G, n_2) iff in CFG G statement n_2 is data dependent on the statement n_1.

For this discussion we assume that a function call uses all its value parameters and defines and uses all its reference parameters. This assumption is conservative and ensures that our analysis produces safe result. More precise information from interprocedural analysis may be used to improve the quality of the results.

Definition: *Dependence.* A statement n_j is *dependent* in statement n_i iff it is either data dependent on n_i or control dependent on some edge (n_i, c), for some c.

Notation: A statement $n_1 \in$ D(G, n_2) iff in CFG G statement n_2 is dependent on statement n_1.

Definition: *Slice.* A statement n_i is in the *slice* of statement n_j iff n_j is transitively dependent on n_i or $n_i = n_j$.

$n_i \in$Slice(G, n_j) iff $n_i = n_j$ or $n_i \in$ D(G, n_j) or $\exists n_k \in$ D(G, n_j) and $n_i \in$ Slice(G, n_k).

Definition: *Slice for a statement set.* Let S be a set of statements, Slice(G, S) = $\bigcup_{s \in S}$ Slice(G, s).

4 Folding contiguous code

In this section we present a transformation to *fold* contiguous code segments into new functions. The fold transformation, also sometimes called *lambda lifting*, was first developed by Burstall and Darlington in the context of functional programming [BD77] and subsequently studied for logic programs [TS84]. Griswold and Notkin developed this transformation, calling it `extract-function`, for a structured, imperative language [Gri91, GN93]. We now extend the transformation to an unstructured, imperative language.

In the functional domain the fold transformation may be applied to *any* expression. In the logic domain this transformation may be applied to *any* primitive predicate or any conjunction of primitive predicates. Similarly, in structured, imperative programs, any sequence of statements may be folded into

* Since we treat a function call as an atomic, undecomposable unit, the issue of aliasing—as a result of using the same variable multiple times in a function call—is not important.

Fold(G, G') = <$G1, G2$>
where

$G = (N, E)$ is a flowgraph
$G' = (N', E')$ is a foldable subgraph of G'
$N" = N' - \{\chi_{G'}\}$
$e2$ is a new start node and $r2$ is a new end node
$N2 = N" \cup \{e2, r2\}$
$E2 = (E \cap N" \times N") \cup \{(e2, \varepsilon_{G'})\} \cup \{(n', r2) \mid (n', \chi_{G'}) \in E'\}$
$G2 = (N2, E2)$ is a flowgraph with $\varepsilon_{G2} = e2$ and $\chi_{G2} = r2$
$f1$ is a new call statement in $G1$ such that **Calls**$(G1, f1) = G2$.
$N1 = (N - N") \cup \{f1\}$
$E1 = (E \cap N1 \times N1) \cup \{(f1, \chi_{G'})\} \cup \{(n, f1) | (n, \varepsilon_{G'}) \in E\}$
$G1 = (N1, E1)$ is a flowgraph with $\varepsilon_{G1} = \varepsilon_G$ and $\chi_{G1} = \chi_G$
Ref$(G2)$ = **Ref**$(G1, f1)$ = Outvar($G1, N"$)
Value$(G2)$ = **Value**$(G1, f1)$ = Value($G1, N"$)
Local$(G2)$ = Local$(G1, N")$

Figure 3 A transformation to fold a foldable subgraph

a function. However, in the context of unstructured, imperative programs one cannot extrapolate that the fold transformation may be applied to *any* statement or any sequence of statements. In order to be converted to a function, it is important that the sequence of statements form a *single-entry, single-exit* subgraph (SESE subgraph)—since the new function created will have only a single entry and a single exit point. Not every sequence of statements in an unstructured, imperative program may satisfy this constraint. Hence, unlike its functional and logic counterpart, not every sequence of statements of an unstructued imperative program can be folded.

Definition: *Subgraph.* A directed graph $G' = (N', E')$ is a subgraph of the flowgraph $G = (N, E)$ iff $N' \subseteq N$ and $E' = E \cap (N' \times N')$.

Definition: *SESE subgraph.* A subgraph $G' = (N', E')$ is a single-entry, single-exit (SESE) subgraph of a directed graph (N, E) iff $\exists \varepsilon_{G'}, \chi_{G'} \in N'. \forall q \in (N - N'), q' \in N'. (q, q') \in E \Rightarrow q' = \varepsilon_{G'}$ and $(q', q) \in E \Rightarrow q' = \chi_{G'}$

To fold a SESE subgraph is to create a new CFG representing a new function and to replace the subgraph in the original CFG by a node representing a call to the new function. When folding a SESE subgraph all nodes, except the end node, in the subgraph are moved into the new function. This implies that a SESE subgraph containing an edge from its end node to some intermediate node cannot be folded. Hence we define a foldable subclass of SESE subgraphs.

Definition: *Foldable (SESE) subgraph.* A SESE subgraph $G' = (N', E')$ is a foldable subgraph of $G = (N, E)$ iff there is no edge from $\chi_{G'}$ to any node in N', except $\varepsilon_{G'}$ or $\chi_{G'}$, i.e.,$\forall n' \in N'. (\chi_{G'}, n') \in E' \Rightarrow n' = \chi_{G'}, n' = \varepsilon_{G'}$.

A foldable subgraph G' of a graph G may be uniquely represented by (a) the tuple (N', E'), or (b) its pair of entry and exit nodes, i.e., $(\varepsilon_{G'}, \chi_{G'})$, or (c) by the set of nodes in the subgraph, i.e., N'. In the following discussion we use the three representations interchangeably.

Our foldable SESE subgraph is different from the SESE region of Johnson, Pearson, and Pingali (JPP) [JPP94]. The single-entry, single-exit condition of a JPP-SESE region is defined in terms of a pair of edges, not a pair of nodes. In addition a JPP-SESE region also has a stronger constraint that every cycle containing the start edge also contains the end edge, and vice-versa. The JPP definition is motivated by the need to compute control dependence regions. In contrast our definition is designed for folding computation into a function.

In order to introduce a function call we also need to identify the parameters that are passed between the new call site and the new procedure. The following definitions are used to identify these parameters.

Definition: *Region.* A *region* is a set of statements of a function.

Definition: *Variables of a region* (Vars). Let Vars(G, X) give the set of all variables that are defined or used at any statement in the region X of CFG G.

Definition: *Input variable of a region* (IN?). A variable v is an input variable for a region X iff there is at least one definition of v outside X that reaches a use of the variable v inside X.

IN?(G, X, v) = $\exists d \notin X.d$ defines $v \wedge \exists u \in X.d \in$ DD(G, u)

Definition: *Output variable of a region* (OUT?). A variable v is an output variable for a set of statements X iff there is at least one definition of v inside X that reaches a use of the variable v outside X.

OUT?(G, X, v) = $\exists u \in X.u$ defines $v \wedge \exists d \notin X.u \in$ DD(G, d)

Definition: *All output variables of a region* (Outvar). The set of all variables that are output variables of a set of statements.

Outvar(G, X) = $\{v \mid$OUT?$(G, X, v)\}$

Definition: *Value variables of a region* (Value). A variable v is a value variable for a set of statements X iff it is its input variable but not its output variable.

Value(G, X) = $\{v \mid$IN?$(G, X, v) \wedge \neg$ OUT?$(G, X, v)\}$

Definition: *Local variables of a region* (Local). A variable v is a local variable for a set of statements X iff it is neither its input variable nor its output variable.

Local(G, X) = $\{v \mid \neg$IN?$(G, X, v) \wedge \neg$ OUT?$(G, X, v)\}$

Lemma: Every variable in Vars(G, X) is contained in one and only one of the sets Outvar(G, X), Value(G, X), Local(G, X).

Proof: From definitions.

Definition: *Fold transformation.* Let $G, G1, G2$ be flowgraphs, G' be a foldable subgraph of G, Fold(G, G') = <$G1, G2$> as described in Figure 3.

The fold transformation does not alter the HPR semantics of the original flowgraph.

Lemma: Let Fold(G, G') = <$G1, G2$>. $\forall \psi, n . \mathcal{I}(G)(n)\psi = \mathcal{I}(G1)(n)\psi$.

Proof: By construction, the fold transformation places a statement from G in either $G1$ or $G2$, i.e., $N \subseteq N1 \cup N2$ and $N1 \cap N2 = \phi$. Since G' is a foldable subgraph of G, the fold transformation neither removes any control flow path nor introduces any new control flow path. Hence, it does not alter the HPR semantics of the program.

5 Tucking non-contiguous code

We now present our transformation for tucking a set of statements. This transformation takes three inputs: G_{old}, a CFG representing function; S, a set of statements of G_{old}; and G_S, a foldable subgraph of G_{old} containing S. If the statements S can be tucked without changing the external behavior of the function G_{old}, the transformation returns two CFGs, $G1$ and $G2$, where $G1$ replaces G_{old} and $G2$ is a new function containing the statements S.

The tuck transformation is composed of three transformations: Wedge, Split, and Fold. The fold transformation was introduced in the previous section. The other two transformations are developed in this section.

Definition: *Wedge*. Let S be a set of statements and $G_S = (N_S, E_S)$ be a foldable subgraph of G_{old} such that G_S contains all the statements in S.

Wedge(G_{old}, G_S, S) = Slice(G_{old}, S) $\cap$ N_S.

Wedge takes the same three inputs as Tuck and returns those statements in Slice(G_{old}, S) which are also in the foldable subgraph G_S.

To define the split transformation we use the following graph operations.

Definition: *Graph union.* $G_1 = (N_1, E_1)$, $G_2 = (N_2, E_2)$, $G_1 \cup G_2 = (N_1 \cup N_2, E_1 \cup E_2)$.

Definition: *Graph difference.* $G_1 = (N_1, E_1)$, $G_2 = (N_2, E_2)$, $G_1 - G_2 = (N_1 - N_2, E_1 - E_2)$.

Definition: *Node deletion.* Delete the node $n \in N$ from the graph $G = (N, E)$. $G/n = (N', E')$ where $N' = N - \{n\}$ and $E' = (E \cap (N' \times N')) \cup \{(m, \text{ipdom}(n)) \mid (m, n) \in E\}$. The edge $(m, \text{ipdom}(n))$ has the same tag as the edge (m, n).

Definition: *Deleting a set of nodes.* Delete the set of nodes X from the graph G. $G/X = \bigcup_{x \in X} G/x$.

The node deletion operation is the same as the CFG node elimintation operation defined by Ball and Horwitz [BH93].

Definition: *Node replacement.* In the graph $G = (N, E)$ replace the node $n \in N$ by a new node $z \notin N$. $[z/n]G = (N', E')$ where $N' = (N - \{n\}) \cup \{z\}$ and $E' = \{(x', y') \mid (x, y) \in (E \cap (N' \times N')).(x = n \wedge x' = z \vee x \neq n \wedge x' = x)$ and $(y = n \wedge y' = z \vee y \neq n \wedge y' = y)\}$

Definition: *Subgraph replacement.* In the graph G replace the SESE subgraph G_x by the SESE subgraph G_z.

$[G_z/G_x]G = (([\varepsilon_{G_z}/\varepsilon_{G_x}, \chi_{G_z}/\chi_{G_x}]G) - G_x) \cup G_z$

We use two operations for sequentially composing graphs. In the first operation two graphs are composed by introducing a new edge from the end of one graph to the start of the other graph. In the second operation the composition is performed by replacing the end node of the first graph by the start node of the second graph.

Definition: *Sequential composition by edge introduction.* Let G_1 and G_2 be foldable subgraphs. $G_1; G_2 = G_1 \cup G_2 \cup (\phi, \{(\chi_{G_1}, \varepsilon_{G_2})\}$.

Definition: *Sequential composition by end node substitution.* Let G_1 and G_2 be foldable subgraphs. $G_1 \oplus G_2 = ([\varepsilon_{G_2}/\chi_{G_1}]G_1) \cup G_2$.

Definition: *Split transformation.* As defined in Figure 4.

The Split transformation splits a foldable subgraph into two foldable subgraphs without altering the behavior of the original program. This transformation takes a CFG G_{old}, a set of seed statements S, and

Split(G_{old}, G_S, S) = G_{new}
where

$G_{old} = (N_{old}, E_{old})$
$G_S = (N_S, E_S)$ is a foldable subgraph of G_{old}
- - Separate computation related to S and the renaming computation
$A = N_S - \chi_{G_S}$
X = Wedge(G_{old}, G_S, S) = Slice(G_{old}, S) $\cap$ N_S
$\hat{X} = A - X$
Y = Wedge($G_{old}, G_S, \hat{X}$) = Slice($G_{old}, \hat{X}$) $\cap$ N_S
$\hat{Y} = A - Y$
- - Determine if splitting will cause conflict
if Outvar(G_{old}, X) $\cap$ Outvar(G_{old}, Y) $\neq \phi$ **then**

conflict

else

- - Find the variable that should be renamed
V = Local(G_{old}, Y) $\cup$ Value(G_{old}, Y)
- - Compute two new subgraphs
$G_X = G_S/\hat{X}$
$G_Y = G_S/\hat{Y}$
- - Compose the two new subgraphs
$G_{XY} = I[\![V]\!]; G_X \oplus [\![V]\!]G_Y$
- - Replace the composed subgraph in the original graph
$G_{new} = [G_{XY}/G_S]G_{old}$

fi

Figure 4 Transformation for splitting a foldable subgraph

a foldable subgraph G_S containing statements S. It creates a new CFG that is similar to G_{old}, except that the subgraph G_S is split into two subgraphs G_X and G_Y that are sequentially composed into $G_X \oplus G_Y$.

While the statements in G_X and G_Y are derived from the statements in G_S, they are not necessarily identical to those statements. The split transformation may rename some variables in the new subgraphs in order to preserve the behavior of the original program. In doing so the transformation may also introduce some additional assignments to initialize the newly introduced variables. The following operations are used for this purpose.

Definition: *Introduce new variables.* Let V be a set of variables. $[\![V]\!]$ represents a set of ordered pairs (v, v') where $v \in V$ and v' is a new variable (not used in the CFG being transformed). Each variable v is paired with a different new variable v'.

Definition: *Introduce new assignments.* Let V be a set of variables. $I[\![V]\!]$ represents a graph consisting of a sequence of assignment statements of the form $v' := v$, $\forall (v, v') \in [\![V]\!]$. The order of the statements is not significant since a variable is only renamed once.

Definition: *Rename variables in subgraph.* Let V be a set of variables and G' a foldable subgraph. $[\![V]\!]G'$ is the flowgraph resulting from consistently replacing all occurrences of a variable v by v' in the statements of G', where $(v, v') \in [\![V]\!]$.

The split transformation is given in Figure 4. It consists of the following steps:

1. *Separate computation related to S from the remaining computation*. This requires separating the statements of G_S in two possibly overlapping sets X and Y, where X contains the computation related to S and Y contains the remaining computation.
2. *Determine if splitting will cause conflict*. A conflict occurs when both X and Y modify a variable that is used outside of X and Y, respectively. That is, Outvar(X) $\cap$ Outvar(Y) $\neq \phi$.
3. *Find the variables that should be renamed.* If there is no conflict then the set of variables V of Y that should be renamed is determined. These are variables that are modified or used in Y, but that do not belong to Outvar(Y). This is a conservative computation. It renames a larger set of variables than may be necessary.
4. *Compute two new subgraphs*. Create two subgraphs G_X and G_Y from G_S. The subgraph G_X (similarly, G_Y) is created by deleting from G_S all the statements that are not in X (similarly, Y). Both G_X and G_Y contain χ_{G_S}.
5. *Compose the two new subgraphs*. Create a new subgraph G_{XY} by composing (1) $I[\![V]\!]$, a sequence of new assignment statements that initialize the renamed variables, (2) G_X, and (3) $[\![V]\!]G_Y$, the graph G_Y with variables of V renamed. $I[\![V]\!]$ and G_X are composed by introducing a new edge, whereas G_X and $[\![V]\!]G_Y$ are composed using node deletion. Since χ_{G_S} is contained in both G_X and G_Y node deletion ensures that G_{XY} has only one χ_{G_S}.
6. *Replace the composed subgraph in the original graph.* Create a graph G_{new} from G_{old} by replacing G_{XY} for G_S.

In the following discussions the symbols are used with respect to Figure 4.

Lemma: $[\varepsilon_{G_Y}/\chi_{G_X}]G_X$ and $[\![V]\!]G_Y$ are foldable subgraphs of G_{new}.

Proof: Follows from construction.

The set A consists of all the statements of G_S, except χ_{G_S}. The sets X and Y are not disjoint but contain all the statements of A. Hence every statement of A may be mapped to at least one statement of G_{new}, either in G_X or in $[\![V]\!]G_Y$, and at most two statements, one in G_X and the second in $[\![V]\!]G_Y$.

Theorem: The Split transformation is behavior preserving. $\forall(n, n') \in \mathcal{C}.\forall\psi.\mathcal{I}(G_{old})(n)\psi = \mathcal{I}(G_{new})(n')\psi$, where $\mathcal{C}$ is a set of ordered pair (n, n') such that n belongs to G_{old}, n' belongs to G_{new} and n' is a copy of n.

Proof: Not included.

Definition: *Tuck*. As defined in Figure 5.

That tuck does preserves the HPR semantics of the original program follows from the similar property of Wedge, Split, and Fold.

6 Implementation and use

We now present a scenario of using the above transformations to restructure functions and discuss our experience with implementing these transformations.

Tuck(G_{old}, G_S, S) = <G_1, G_2>
where

G_{new} = Split(G_{old}, G_S, S)
G_X is as defined in Figure 4
< G_1, G_2 >= Fold(G_{new}, G_X)

Figure 5 Transformation for tucking code in a foldable subgraph

The tuck transformations requires three parameters: the function to be restructured, a set of seed statements, and a foldable subgraph containing the seed statements. A tool implementing tuck must address how these parameters would be identified. This in turn would depend on whether the tool is a batch or interactive. We first discuss our vision of an interactive environment we are developing [Lak98] and then discuss our experience with developing a batch solution [Lak97].

Whether a batch or an interactive tool, the selection of a foldable subgraph containing the seed statements is a problem that requires some automated support. Since there may be several foldable subgraphs containing a given set of seed statements, the problem may be split into two steps. First, identify all the foldable subgraphs containing the seed statements. Second, select one foldable subgraph and use it as a parameter for tuck.

Johnson et al.'s definition of SESE region [JPP94] is stronger than our definition of a foldable (SESE) subgraph, in that every JPP-SESE region also defines a foldable subgraph, whereas there may be foldable subgraphs that do not define a JPP-SESE region. The JPP-SESE regions of a program can be ordered as a tree, called the *program structure tree*, each of whose node defines a JPP-SESE region. Since this tree can be computed in linear-time, in our subsequent discussion, we use a JPP-SESE region as a foldable region.

Definition: *Single definite control* (SDC). The SDC of a set of statements S is a JPP-SESE region containing the statements S.

The nearest common ancestor of the statements S in the program structure tree is an SDC of S. This is called the *nearest* SDC. Similarly, all the ancestors of this nearest SDC of S are also SDCs of S.

Lemma: The entry node of ε_G is an SDC of every subset of statements contained in CFG G.

Proof: By definition.

Figures 6 through 8 illustrate how we envision the tuck transformation will be used to interactively restructure programs. A restructuring step in this interactive model consists of the following activities:

1. The user selects a set of seed statements.
2. (a) The system highlights the SDCs of the seed statements. (b) User picks an SDC which the system uses to create a wedge.
3. The system verifies whether the wedged code can be tucked into a function. If so, it tucks it.

Figures 6 (a) and (b) shows the details of performing the above activities once.

1.1 The user selects the *readln* statement as the seed.
1.2a The system highlights two SDCs, the **while** statement and the procedure entry.
1.2b The user selects the procedure entry and the system computes a wedge within this region.

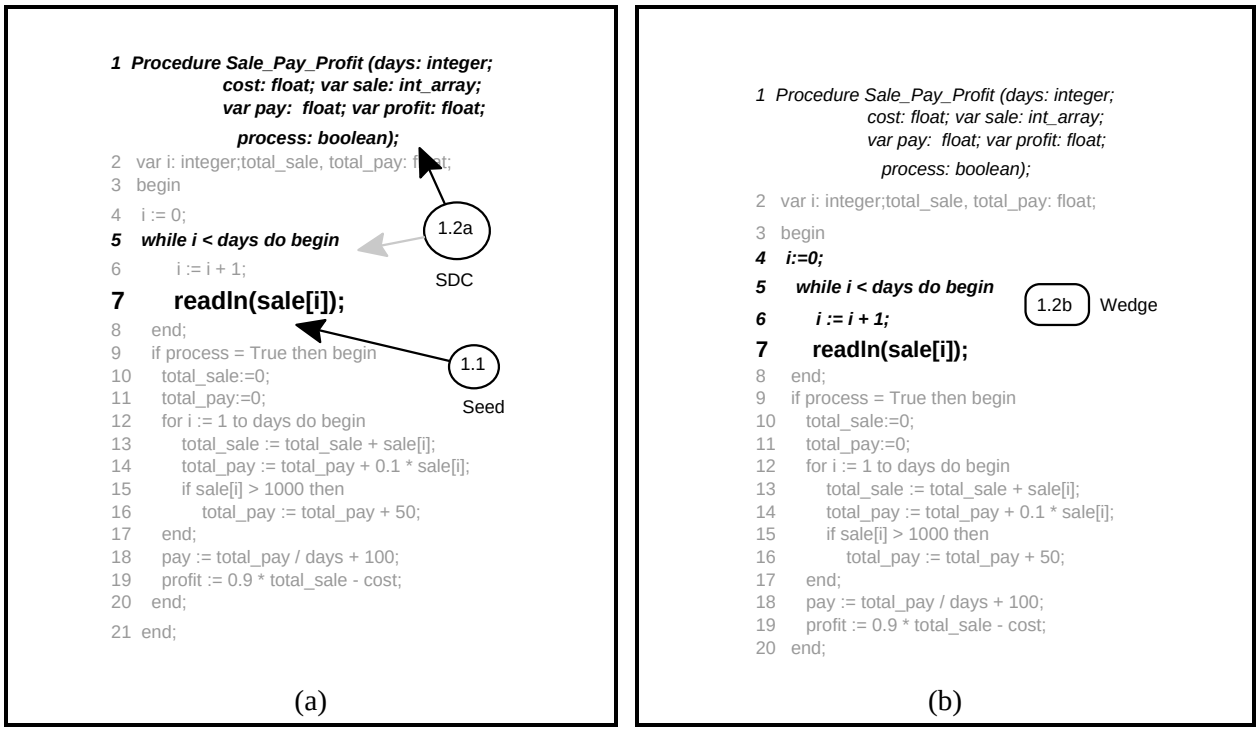

Figure 6 Selection of code to be extracted into a function. (1.1) The user selects the seed. (1.2a) The system highlights the two SDCs of the seed. Each SDC defines a SESE region in which to bound the slice. The user selects one SDC. (1.2b) The system identifies the statements influencing the seed within the region defined by the SDC. In this step, the user selects the *readln* statement as the seed with the intent to separate the user interface from the computation.

1.3 The wedged code is tucked into a new procedure.

The result of the wedge transformation is shown in Figure 6(b). The subsequent restructuring is performed using similar steps. This example has been taken from Deprez [Dep97]. Details about the intermediate steps and the formal definition of the transformations may be found in his thesis.

We have developed a batch tool that uses the tuck transformation to restructure code [Lak97]. We identify functions that need to be restructured using a measure of cohesion proposed by Lakhotia and Nandigam [Lak93, Nan95]. The cohesion of a module is computed as a function of the cohesion between pairs of variables. The pairwise cohesion between variables is also used to create the seed for tucking. This is achieved by creating a partition of variables, each partition containing variables transitively related to another variable by some minimum threshold cohesion. The assignment statements for the variables in each partition belong to the seed. Our current implementation uses the whole function as the foldable subgraph parameter of tuck.

The batch implementation we have thus takes a threshold cohesion level as a parameter and restructures all the functions in a system that have a cohesion below that threshold. Furthermore, the new functions created are guaranteed to be at least as cohesive as the given threshold.

7 Related works

One of the strongest evidence of the need for restructuring of the type proposed here comes from observations made by Rugaber et al. [RSW95]. They have investigated the problem of detecting

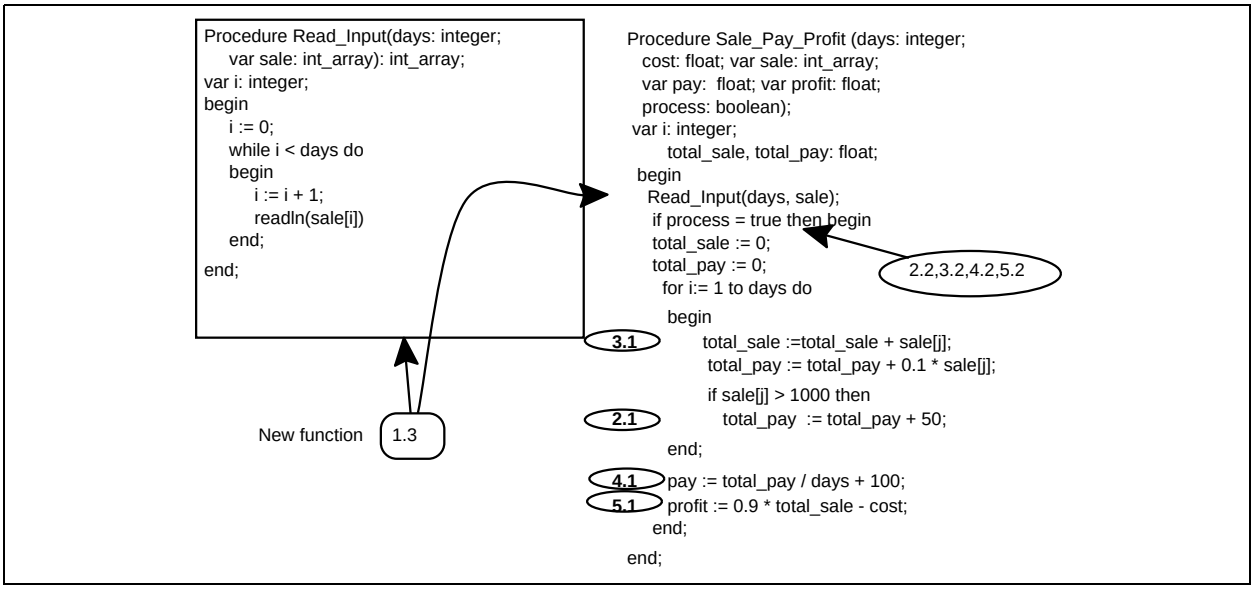

Figure 7 Completion of function extraction, and input for subsequent steps (1.3) The code selected in Figure 6 is extracted and converted into a function. The selected code is replaced by a call to this function. Since the selected code did not interleave with any other code the decision about where to place the call was straightforward. User selected seeds and the SDC for steps 2, 3, 4, and 5 are shown. The next figure contains the result of these steps.

```
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;                    (1)
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;                 (2)
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;                 (3)
begin
    total_sale := 0;
    for j := 1 to days do
    begin
        total_sale := total_sale + sale[j];
    end;
    return (total_sale);
```

```
Function Compute_Avg_Pay
    (days: integer; sale: int_array): float;
var total_pay: integer;
     pay: float;                   (4)
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;    (5)
begin
    total_sale := Compute_Sale(days, sale);
    profit := 0.9 * total_sale - cost;
    return (profit);
end;
```

Figure 8 Final result of restructuring program in Figure 1. The annotations **1** to **5** indicate the restructuring steps, with respect to previous figures, in which the function was created. To create functions in steps **2** and **3** required separating interleaved computations. This was achieved by duplicating some code segment.

"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." 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 not all of which were used by the calling routines. The transformation we propose here may be used to restructure such libraries, say by specializing functions with multiple output to compute only a certain set of outputs.

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 [SJ87]. Griswold and Notkin's identified the extraction of code into a separate function as one of several transformations useful in program restructuring [Gri91, GN93]. They developed an automated restructuring tool that was subsequently used by Bowdidge for extracting functions belonging to an abstract data type [Bow95]. Sneed, Griswold and Notkin, and Bowdidge efforts differs from ours in that they limited their focus to moving contiguous pieces of code into separate functions.

The restructuring techniques of Kim et al. [KCK94] and Kang and Bieman [KB96] 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. Their focus has been in the application of cohesion measures for identifying related computations. They have not developed the formal foundation for using program slicing for this purpose. Hence, they have overlooked several issues, as for instance, restructuring unstructured programs, creating slices that span only a part of a function, and resolving conflict between the slice to be extracted and the remaining statements.

The formal transformation to tuck computation into a function provided in this paper complements (and completes) Kim et al. [KCK94] and Kang and Bieman's [KB96] effort. The measures of cohesion they use, or the one proposed by Lakhotia and Nandigam [Lak93, Nan95], or Bieman and Ott [BO94] may be used by an automated tool to identify non-cohesive modules and to create the initial seed to restructure these modules. The computation related to the seed may then be tucked using our transformation.

From the very beginning program slicing has been proposed as a method for decomposing programs to aid various software maintenance activities. Weiser [Wei79] envisaged its use for system generation, which in today's terminology may be called the extraction of a specialized system from an existing system or the extraction of reusable components. The extraction of such specialized components using program slicing has recently been studied by Lanubile and Visaggio [LV97] and by Reps and Turnridge [RT96].

Gallagher and Lyle [GL91, Gal92, Gal96] use program slicing to decompose programs but not for the purpose of restructuring. They compute slices on the output variables of a function at its last statement and organize the resulting slices into a lattice based on a partial order relation defined between slices. They have developed an editor that, using this lattice, identifies the program fragments that should not be affected when computation for a variable is changed and then hides (or write protects) such fragments. This helps in identifying and controlling ripple effect during code modifications.

Jain introduces a novel notion of a *projection* of a logic program [Jai95]. Jain's projections remove pieces of code to obtain the core control structure of a logic program. His approach works well for incremental program construction methods. Jain's decomposition of logic programs using projections is not for restructuring like we discuss here; instead, he uses projections to orthogonally decompose interleaving computations so that the interleaving computations can be independently developed and understood.

Horwitz et al. [HPR89] have used program slicing to decompose programs but for the purpose of separating modifications performed in two different versions of the same program. Once the modifications are identified they create a composite program that integrates the changes from both the versions.

8 Conclusions and future research

The need for reengineering and restructuring software is motivated by Lehman's second law of software evolution: "As a large program is continuously changed, its complexity which reflects deteriorating structure, increases unless work is done to maintain or reduce it" [LB85, p. 253]. To reduce the deterioration of a program's structure a programmer typically has to undo some previous design decisions and modify the code such that it conforms to a new design that is more suitable for the changed requirements. That is essentially the aim of software reengineering and restructuring.

We have presented a transformation, Tuck, that may be used to restructure a program by breaking its large functions into small functions, without changing it behavior. Tuck creates a new function containing a given set of statements, called the seed statements, and replaces these statements by a call to this new function. If the seed statements are not contiguous, the transformation identifies the statements needed to perform all the computation in a single-entry, single-exit (SESE) subgraph, also provided as a parameter.

The tuck transformation consists of three steps: Wedge, Split, and Fold. Given a set of *seed* statements one first creates a *wedge* that contains all the statements that influence the seed statements. A wedge is a program slice bounded within a SESE subgraph of the control flow graph. The depth of the wedge is controlled by means of selecting a subgraph that contains all the statements in the slicing criterion, the set of seed statements. The function is then *split* such that all the statements in the wedge are placed contiguously in its flowgraph. This contiguous piece of code, that also forms a SESE subgraph, is then *folded* into a function.

The automated restructuring transformation presented in this paper may be used to reduce the deterioration of a program's structure. The transformation may be incorporated in a completely automated tool for restructuring legacy code or it may be part of an interactive program development environment that provides restructuring operations as primitives. An automated tool would automatically identify code that ought to be restructured and restructure it as well [KB96, KCK94, Lak97, SJ87]. In an interactive environment a programmer may interactively select the transformation during his routine program modification activities, thus preventing the code from deteriorating in the first place [Lak98].

Acknowledgments: 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.

9 References

[AM71] 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.

[Arn89] Robert S. Arnold. Software restructuring. *Proc. IEEE*, 77(4):607–617, April 1989.

[Bak77] B. Baker. An algorithm for structuring flowgraphs. *Journal of the ACM*, 24(1):98–120, January 1977.

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BH93] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In P. Fritzson, editor, *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 206–222. Springer-Verlag, 1993.

[BL91] Peter T. Breuer and Kevin Lano. Creating specifications from code; reverse-engineering techniques. *Journal of Software Maintenance: Research and Practice*, 3:145–162, 1991.

[BO94] James M. Bieman and Linda M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):476–493, June 1994.

[Bow95] Robert W. Bowdidge. *Supporting the Restructuring of Data Abstractions through Manipulation of a Program Visualization*. PhD thesis, University of California, San Diego, November 1995.

[Cal88] Frank W. Calliss. Problems with automatic restructurers. *SIGPLAN Notices*, 23:13–21, March 1988.

[CC90] Elliot J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[Dep97] Jean-Christophe Deprez. A context-sensitive formal transformation for restructuring programs. Master's thesis, The Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, Louisiana, December 1997.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[Gal92] K. 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.

[Gal96] Keith Gallagher. Visual impact analysis. In *International Conference on Software Maintenance*, 1996.

[GL91] Keith B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[GN93] William G. Griswold and David Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering*, 2(3):228–269, July 1993.

[Gri91] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, July 1991.

[HPLH90] 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.

[HPR89] 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.

[Jai95] Ashish Jain. Projections of logic programs using symbol mappings. In *Proceedings of the Twelfth International Conference on Logic Programming*, pages 483–496. The MIT Press, 1995.

[JPP94] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 171–185. ACM Press, 1994.

[Kas74] Takumi Kasai. Translatability of flowcharts into while programs. *Journal of Computer and System Sciences*, 9:177–195, 1974.

[KB96] 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.

[KCK94] 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.

[Lak93] 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.

[Lak97] Arun Lakhotia. Restructuring complex program fragments into small cohesive units. `http://www.cacs.usl.edu/ arun/Wolf` , May 1997.

[Lak98] Arun Lakhotia. DIME: A direct manipulation envirobment for evolutionary development of software. In *Proceedings of the International Workshop on Program Comprehension (IWPC'98)*, page to appear, Los Alamitos, CA, June 1998. IEEE Computer Society Press.

[LB85] M. M. Lehman and L. A. Belady. *Program Evolution*. Academic Press, 1985.

[LV97] 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.

[Nan95] Jagadeesh Nandigam. *A measure for module cohesion*. PhD thesis, University of Southwestern Louisiana, The Center for Advanced Computer Studies, Lafayette, Louisiana, 1995.

[RSW95] 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.

[RSW96] Spencer Rugaber, Kurt Stirewalt, and Linda Wills. Understanding interleaved code. *Automated Software Engineering*, 3(1-2):47–76, June 1996.

[RT96] Thomas Reps and T. Turnidge. Program specialization via program slicing. In O. Danvy, R. Glueck, and P. Thiemann, editors, *Lecture Notes in Computer Science*, volume 1110, pages 409–429. Springer-Verlag, New York, NY, 1996.

[SJ87] 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.

[SMC74] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[Sta82] 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.

[TS84] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In *Proceedings of Second International Conference on Logic Programming, (Sweden)*, pages 127–138, 1984.

[War93] Martin Ward. Abstracting a specification from code. *Journal of Software Maintenance: Research and Practice*, 5:101–122, 1993.

[Wat88] Richard C. Waters. Program translation via abstraction and reimplementation. *IEEE Transactions on Software Engineering*, 14(8):1207–1228, August 1988.

[Wei79] M. Weiser. *Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, Ann Arbor, Michigan, 1979.