Identifying Procedural Structure in Cobol Programs

John Field G. Ramalingam

IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY, 10598, USA

{jfield,rama}@watson.ibm.com

Abstract

The principal control-flow abstraction mechanism in the Cobol language is the PERFORM statement. Normally, PERFORM statements are used in a straightforward manner to define parameterless procedures (where global variables are used to pass data into and out of procedure bodies). However, unlike most procedural constructs, distinct PERFORMed procedures can share code in arbitrarily complicated ways. In addition, PERFORMs can also be used in such a way as to cause transfers of control that do not correspond to normal call/return semantics.

In this paper, we show how a Cobol program can be efficiently transformed into a semantically-equivalent *procedurally well-structured* representation, in which conventional procedures (i.e., with the usual call and return semantics and without code sharing) and procedure call statements replace PERFORMed code and PERFORM statements. This transformation process properly accounts for the non-procedural control flow that can result from ill-behaved PERFORM statements.

The program representation derived from our analysis can be used directly in program understanding applications, program restructuring tools, and inter-language translators. In addition, it can be used as the starting point for a variety of context-sensitive program analyses, e.g., program slicing.

1 Problem Setting

At this writing, the Cobol programming language is 40 years old. By computing standards, this is a singularly advanced age. Nonetheless, Cobol continues to be the linguistic backbone of much of commercial computing. Although Cobol has evolved considerably since its 1959 inception, its fundamental control and data constructs remain largely unchanged.

The PERFORM statement

The principal program structuring mechanism in the Cobol language is the PERFORM statement[1]. Normally, PERFORM statements are used in a straightforward manner to define parameterless procedures (where global variables are used to pass data into and out of procedure bodies).

Consider, for example, the program depicted in outline form in Fig. 1. This program, which is representative of many Cobol applications, reads a file containing employee payroll records, and for each record in that file, writes a corresponding paycheck record to another file. The executable part of the program is the PROCEDURE DIVISION, which in this case contains two PERFORM statements. In its most basic form, a PERFORM statements specifies a *range* of labeled Cobol code blocks ("PARAGRAPHs") to be executed. Thus, for example, PERFORM OVERTIME-CALC THRU OVERTIME-CALC-EXIT specifies that all of the PARAGRAPHs in the range beginning with OVERTIME-CALC and ending with OVERTIME-CALC-EXIT are to be executed when the PERFORM is invoked. After the statements in the PERFORM's range are executed, control passes to the successor of the invoking PERFORM statement in a manner similar to a procedure call return.

Cobol contains a number of variant PERFORM constructs which elaborate on the basic PERFORM statement in minor ways. For example, PERFORM WRITE-PAYCHECK UNTIL... in Fig. 1 specifies a PERFORM range consisting of a single PARAGRAPH, which is executed repeatedly until the condition W-EOF = "Y" becomes true. However, all of variant forms of PERFORM can be easily expressed in terms of the basic form

PERFORM $label_s$ THRU $label_e$

where $label_s$ is the label of the entry PARAGRAPH in the range, and $label_e$ is the label of the exit PARAGRAPH.

Understanding the semantics of PERFORMs

At first glance, it may appear that a PERFORM is simply a syntactically-awkward form of procedure call. Unlike most procedural constructs, however, distinct PERFORMed procedures can share code in arbitrarily complicated ways. Consider, for example, Figures 2–5. In Fig. 2, one range of PERFORMed code is nested within a second range. Fig. 3 depicts nested ranges *and* nested PERFORMs. In Fig. 4, the statements of the two ranges overlap inexactly. Finally, Fig. 5 depicts a program where the exit label of the PERFORM lexically *succeeds* the entry label. All of these examples are perfectly valid in Cobol.

Do all syntactically well-formed PERFORMs have a valid semantic interpretation? The answer is "no", but determining precisely which PERFORMs are valid and which are not is difficult to ascertain from the rather informal semantics found in various Cobol language references. Indeed, the "correct" semantics for PERFORMs has evolved over time; this evolution appears to be due

[1] A "main" Cobol PROGRAM may take parameters, and thus function in a manner similar to an ordinary parameterized procedure. Although standard analysis techniques are readily applicable to these constructs, the vast majority of extant Cobol programs rely on PERFORMs as their principal control-flow abstraction mechanism.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PAYROLL.
...
DATA DIVISION.
FILE SECTION.

FD PAYROLL-FILE.
01 PAYROLL.
   05 NAME       PIC X(20).
   05 PAYRATE    PIC 99V999.
   ...

FD PAYCHECK-FILE.
01 PAYCHECK.
   05 NAME       PIC X(20).
   05 HRS-WORKED PIC 999V99.
   ...

WORKING-STORAGE SECTION.
01 W-EOF         PIC X VALUE "N"
01 W-OVERTIME    PIC 99V9.
...

PROCEDURE DIVISION.
MAIN-LINE-ROUTINE.
    OPEN INPUT PAYROLL-FILE
        OUTPUT PAYCHECK-FILE.
    READ PAYROLL-FILE
        AT END MOVE "Y" TO W-EOF.
    PERFORM WRITE-PAYCHECK
        UNTIL W-EOF = "Y".
    STOP RUN.

WRITE-PAYCHECK.
    ...
    IF HRS-WORKED IS GREATER THAN 40
        PERFORM OVERTIME-CALC THRU
        OVERTIME-CALC-EXIT.
    ...
    WRITE PAYCHECK.
    READ PAYROLL-FILE
        AT END MOVE "Y" TO W-EOF.

OVERTIME-CALC.
    ...
OVERTIME-CALC-EXIT.
```

Figure 1: Typical Cobol Program (in schematic form)

at least in part to user exploitation of particular PERFORM implementation techniques.

Consider the following description [2, pp. 120–121] of correct PERFORM semantics as defined by the 1965 Cobol standard:

> A PERFORM statement can include in its range one or more PERFORM statements. When this occurs, each inner PERFORM statement is said to be "nested" in the next outer PERFORM statement. The following three rules must be observed when one is using nested PERFORM statements:
>
> 1. The range of each PERFORM statement must terminate with a different section or paragraph name.
> 2. An embedded PERFORM statement must have its range either totally inside or totally outside the range of each of its outer PERFORM statement[s]. That is, [a] statement in the range of a PERFORM statement cannot be continued within the range of the next embedded PERFORM statement...
> 3. The ranges of two PERFORM statement can overlap, provided that the second PERFORM statement is not located within the range of the first PERFORM statement.

The current draft revision of the 1985 Cobol standard [5, pp. 484–485, 810] expands the 1965 definition of PERFORM validity thus:

```
A. PERFORM B THRU C.
   ...
   PERFORM C.
   STOP RUN.
B. ...
C. ...
```

Figure 2: Nested PERFORM ranges.

```
A. PERFORM B THRU D.
   STOP RUN.
B. PERFORM D.
C. ...
D. ...
```

Figure 3: Nested PERFORM ranges; nested PERFORMs.

> 7) The range of a PERFORM statement consists logically of all those statements that are executed as a result of executing the PERFORM statement through execution of the implicit transfer of control to the end of the PERFORM statement. The range includes all statements that are executed as a result of the transfer of control by... PERFORM statements in the range of the PERFORM statement... The statements in the range of a PERFORM statement need not appear consecutively in the source element...
>
> 9) The results of executing the following sequence of PERFORM statements are undefined and no exception condition is set to exist when the sequence is executed:
>
> - a PERFORM statement is executed and has not yet terminated, then
> - within the range of that PERFORM statement another PERFORM statement is executed, then
> - the execution of the second PERFORM statement passes through the exit of the first PERFORM statement...
>
> 90) PERFORM statement. A common exit for multiple active PERFORM statements is allowed.

Note that the definition of PERFORM validity in [5] is more permissive than the earlier definition in [2]. However, even the former leaves some issues open to question, e.g., whether recursive PERFORMs are allowed.

To get a precise answer to the question of PERFORM semantics, it is helpful to examine their implementation. Consider again the

```
A. PERFORM B THRU D.
   ...
   PERFORM C THRU E.
   STOP RUN.
B. ...
C. ...
D. ...
E. ...
```

Figure 4: Overlapping PERFORM ranges.

```
A. PERFORM C THRU B.
   STOP RUN.
B. ...
C. GOTO B.
```

Figure 5: PERFORM with “out of order” labels

```
    D_continuation := address(D1);
A:  A_saveCont     := D_continuation;
    D_continuation := address(A1);
    jump address(B);
A1: D_continuation := A_saveCont;

    ...

B:  B_saveCont     := D_continuation;
    D_continuation := address(B1);
    jump address(D);
B1: D_continuation := B_saveCont;

    ...

C:  ...

D:  ...
    jump D_continuation;
D1: ...
```

Figure 6: Implementation of example in Fig. 3.

example in Fig. 3. The implementation of this example in the IBM compiler described in [4] (which we believe to be similar to implementations in other modern compilers) is depicted in pseudocode form in Fig.6. The PERFORM implementation scheme this example illustrates is straightforward:

Every distinct PERFORM exit PARAGRAPH e has a “continuation” address stored in a special storage area associated with e, $\mathit{continuation}(e)$. All continuations are initialized at program startup time to the address of the syntactic successor of the exit PARAGRAPH (its “normal” or “fall-through” successor). When a PERFORM statement u for a range with exit e is invoked, the current value of $\mathit{continuation}(e)$ is first saved in a storage area associated with u, $\mathit{saveCont}(u)$. $\mathit{continuation}(e)$ is then updated to contain the address of a postlude block of code which restores the previous value of $\mathit{continuation}(e)$ from $\mathit{saveCont}(u)$; the postlude block's control successor is the syntactic successor of the PERFORM statement u. Finally, control is transferred to the entry of u's range.

The PERFORM implementation scheme above has the following characteristics:

1. PERFORM return addresses are associated with PERFORM range exit PARAGRAPHs (i.e., with $\mathit{continuation}(e)$).

2. Each PERFORM of a range with exit PARAGRAPH e can store exactly one pending continuation for e.

3. After a PERFORM of a range with exit e, the stored continuation for e is not reset unless and until control reaches e.

As a consequence of (2), recursive PERFORMs are clearly not possible in general. More importantly, because of (3), the correct behavior of a sequence of PERFORMs is dependent on control reaching each of the corresponding PERFORM range exits in the reverse order of PERFORM invocation; i.e., the last range entered should be the first range exited (we will refer to this property as *LIFO* behavior). Aside from the issue of recursion, the implementation technique illustrated by Fig. 6 appears to agree with semantics given in [5].

```
A. PERFORM D THRU E.
B. PERFORM C THRU F.
   STOP RUN.
C. ...
D. IF ERROR GOTO B.
E. ...
F. ...
```

Figure 7: Badly-behaved PERFORM: The first PERFORM may leave a “mine” behind, which the subsequent PERFORM will trip.

PERFORMing badly

Consider now the example in Fig. 7. In this case, there is a check in PARAGRAPH D for a condition (e.g., an error condition), which, if true, will allow PERFORM C THRU F to be executed before the exit PARAGRAPH E of the *previous* PERFORM's range is reached. As a result, E's exit continuation will contain the address of B, rather than its “fall-through” successor, F. This “active” continuation can be thought of as a armed “mine”. As a result, when the second PERFORM is invoked, if the error condition does not recur, then execution will trip the armed mine at E, causing control once again to be passed to B, rather than continuing on to F as the programmer presumably intended.

While behavior such as that illustrated above is most likely to result from programmer error, there is anecdotal evidence from IBM's Cobol implementors that programmers occasionally abuse the semantics of PERFORMs by *intentionally* tripping mines. Note, by the way, that arming a mine without subsequently tripping it is perfectly legal. Indeed, a legitimate programming idiom for handling run-time error conditions is to transfer control via GOTO from the program point where the error is detected to error handling code, the latter of which ends in a program exit (“STOP RUN”). If the GOTO occurs inside an active PERFORM range, the result will be an armed, but otherwise benign, mine.

Creating a well-structured procedural representation

From the preceding examples, it should be clear that the unusual semantics and code sharing capabilities of PERFORMs renders them problematic, both for humans attempting to understand extant code, and for implementors of program analysis algorithms that must be correct and efficient when applied to Cobol.

The goal of this paper is to describe an algorithm for efficiently transforming Cobol programs containing PERFORM statements into a semantically equivalent representation in which all PERFORMs are replaced by ordinary procedures. We will refer to the result of this transformation as a *procedurally well-structured* representation. The procedural representation produced by our algorithm has two principal applications:

- For poorly-structured programs, the transformed representation, appropriately rendered, may be easier for a programmer to understand than the original code, thus aiding program maintenance activities. In Cobol, feasible control flow can often be difficult to ascertain even in the absence of GOTOs. Consider, for example, the paragraph labeled OVERTIME-CALC in Fig. 1. It is difficult to discern from inspection whether control can flow from the labeled statement's syntactic predecessor (READ PAYROLL-FILE...) into the labeled PARAGRAPH. Our representation makes clear that such control-flow is infeasible.

- By making the procedural structure of the program explicit, our transformation enables a variety of *context-sensitive* (i.e., interprocedurally-precise) program analyses [6, ch. 19], [10, 11, 8], to be applied to the program. Without our representation, it would be difficult to apply such analyses, since the procedural abstractions and "calling" contexts that are the basis for context-sensitive analyses are generally not syntactically apparent. Although context-*in*sensitive analyses could be applied in a very straightforward way to Cobol programs, the results will usually be much less accurate than context-sensitive analyses.

Our Approach

Given a Cobol program, numerous procedurally well-structured representations are possible, via a variety of analyses. Our approach is as follows:

- We extend the standard definition of *(interprocedurally) valid* program path to Cobol programs, and observe that this definition formalizes the notion of "well-behaved" PERFORMs given in [5]. Those PERFORM ranges whose operationally feasible paths are valid are deemed *LIFO*.
- We define a simple conservative static criterion to identify LIFO PERFORM ranges. Such ranges are deemed *structurally-LIFO*, or *SLIFO*. In our experiments, the vast majority of Cobol programs we encountered contained only SLIFO ranges.
- We describe an efficient algorithm to determine those PERFORM ranges that are SLIFO.
- Given Cobol program P, we produce a procedurally well-structured representation P' which is related to P as follows:
  - The set of all valid static program paths in P' includes all of the operationally feasible (dynamic) paths in P.
  - Moreover, for programs containing only SLIFO ranges, the set of valid static paths in P corresponds exactly to the valid static paths in P'.
- We do *not* assume *a priori* that all PERFORM ranges are LIFO; our representation correctly accounts for intentional or unintentional misuse of PERFORMs that may occur when the implementation scheme illustrated in Fig. 6 is used.
- Although in the worst case our analysis runs in time quadratic in the size of the original program, for a very large class of programs (including, e.g., SLIFO programs with non-overlapping PERFORM ranges), our analysis runs in linear time, yielding a "natural" procedural representation that respects the structure of the original code.
- Similarly, although the size of our procedural representation is also quadratic in the worst case, it is linear for SLIFO, non-overlapping ranges.

Running Example

Consider the example in Fig. 8, which is designed to illustrate most of the issues our algorithm addresses. Our algorithm will yield the procedurally well-structured representation depicted in Fig. 9. Note that each PERFORM range in the original program has a corresponding procedure in the well-structured representation. In addition, several code fragments that are common to multiple ranges have been "abstracted" into new procedures. Some of the procedures are mutually recursive, in order to account for the possible loop created by the GOTO in the original program. Finally, the code that can be executed when possible mines at exits BX and EX are encountered are represented respectively by special procedures mx-abnml-cont() and pb-abnml-cont().

```
ME. ...
PA. PERFORM AE THRU AX.
PE. PERFORM EE THRU EX.
PB. PERFORM BE THRU BX.
MX. STOP RUN.

AE. ...
PC. PERFORM CE THRU CX.
BE. ...
CE. ...
CX. ...
AX. ...
D1. ...
D2. ...
EE. ...
     IF ... GOTO PB.
EX. ...
BX. ...
```

Figure 8: Complex PERFORM example, containing various PERFORM nestings, and both well- and ill-behaved ranges.

2 Problem Formalization

Control-flow representation

In the sequel, we will require two related control-flow representations of a Cobol program, an *implicit* control-flow graph, or *ICFG*, and an *explicit* control-flow graph, or *ECFG*. The ICFG and ECFG share the same set of vertices, but represent *inter*procedural control-flow, or transfer of control caused by PERFORMs differently, using different sets of edges. In both representations, the vertices will be divided into four classes:

- *entry* vertices, representing the beginning of PERFORM ranges
- *exit* vertices, representing the end of PERFORM ranges
- *perform* vertices, representing PERFORM statements
- *computational* vertices, representing all other statement types

It will be convenient to treat the main program as if it were a PERFORM range with distinguished *entry* vertex s_m and distinguished *exit* vertex e_m. A PERFORM range is represented by a pair $\langle s,\ e\rangle$ consisting of an *entry* vertex s and an *exit* vertex e. For any *perform* vertex u, we denote the corresponding PERFORMed range by *range*(u). The set of all ranges in the program, *PerfRanges*, is defined to be

$$\{\ \langle s_m,\ e_m\rangle\ \} \cup \{\ \textit{range}(u)\ \mid\ u \text{ is a } \textit{perform} \text{ vertex}\ \}$$

Note that not all *entry-exit* pairs constitute a range.

For both the ICFG and ECFG, the outgoing edges of *computational* and *entry* vertices denote conventional intraprocedural control flow. In both representations, an *exit* vertex will also

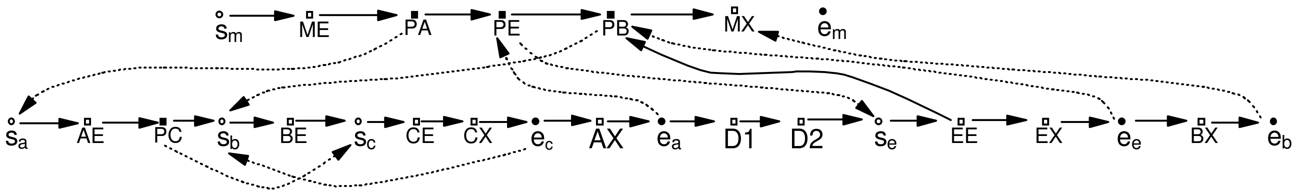

Figure 10: A combined representation of the ICFG and ECFG for the program in Fig. 8. The ICFG includes everything except the dotted edges, while the ECFG includes everything except the dashed edges.

have an outgoing *fallThrough* edge that denotes the transfer of control that occurs from an *exit* vertex to its syntactic successor if the vertex is reached when the exit is inactive. In the ICFG, every *perform* vertex u has a unique successor, the syntactic successor of the `PERFORM` statement. By contrast, in the ECFG, u has an outgoing *call* edge to the *entry* vertex of $range(u)$. The *exit* vertex of $range(u)$ has a corresponding *return* edge to the syntactic successor of u. Fig. 10 illustrates the control-flow graph representation of the program in Fig. 8.

The ICFG and ECFG are similar in many respects to other representations commonly used for procedural programs [10]. However, there are two significant differences:

- An ICFG for a conventional procedural language would typically consist of a collection of disjoint graphs, one per procedure. However, our ICFG is a monolithic graph. This graph reflects the possibility of code sharing among `PERFORM` ranges, both via explicit `GOTO` statements and via *fallThrough* edges.

- In a conventional procedural language, every operationally *feasible* path through the ECFG will contain *matched* pairs of *call* and *return* edges (we formalize this concept below). While every execution of a Cobol program also follows a path in the ECFG, some operationally feasible paths may have *mismatched call* and *return* edges (see, e.g., the program in Fig. 7).

Valid and feasible program paths

In context-sensitive analyses, e.g., [10, 8], the concept of a *valid* path is typically used to characterize paths that do not contain mismatched call and return edges. We can easily extend this concept to Cobol programs, as follows: For every *perform* vertex u, attach the label $(_u$ to the call edge from u to the *entry* vertex of $range(u)$, and the label $)_u$ to the corresponding return edge from the *exit* vertex of $range(u)$ to the syntactic successor of u.

A path in ECFG is said to be *same-level valid* if the sequence of labels on the call and return edges of the path belongs to the language of balanced parentheses generated from the nonterminal *matched* by the following context-free grammar:

$$\begin{array}{rcl} matched & \rightarrow & (_u \; matched \;)_u \; matched \\ & & \text{for every } \textit{perform} \text{ vertex } u \\ & | & \epsilon \end{array}$$

A path in ECFG is said to be *(interprocedurally) valid* if the sequence of labels on the call and return edges of the path belongs to the language generated from the nonterminal *valid* by the following context-free grammar:

$$\begin{array}{rcll} valid & \rightarrow & valid \; (_u \; matched & \text{for every } \textit{perform} \text{ vertex } u \\ & | & matched & \end{array}$$

Given the definitions above, we can now formalize LIFO `PERFORM` ranges, i.e., those that are "well-behaved," as follows:

Definition 1 *A* `PERFORM` *range r is said to be* LIFO *if for every feasible path $\alpha\beta$ where the first edge of β is an invocation edge to r, either a nonempty prefix of β is a same-level valid path or β itself is a valid path.*

We will say that a path in the ECFG is operationally *feasible* if the sequence of vertices on the path is consistent with the sequence of actual computations in some execution history. Due to the existence of mines, it is possible for a path in ECFG to be feasible but not valid. This is problematic if we wish to apply existing context-sensitive analyses (or those based on context-free reachability) to Cobol, since such analyses assume that all feasible paths are valid.

The goal of this work is to produce a procedurally well-structured representation suitable for such analyses, in the sense that every feasible path in the original program *is* a valid path in the resulting program. Note that we can relate paths in the input and output programs by allowing the *computational* vertices in the two graphs to have labels; a path in the output graph represents a path in the input graph if the sequence of labels on the labeled vertices of the two paths are the same.

3 Structurally-LIFO Ranges

As a prerequisite to constructing a procedurally well-structured program representation, we first perform an analysis to conservatively identify `PERFORM` ranges that are LIFO. In this section, we present a simple inductive criterion for a range to be LIFO. This criterion is conservative, in the sense that it will not identify all LIFO ranges. However, it appears to capture the vast majority of cases that occur in practice.

We first define the notion of a range's *body*:

Definition 2 *Given a range $\langle s,\ e \rangle$, its* body *consists of the set of all vertices in the ICFG that can be reached from vertex s without using the fallThrough edge out of vertex e.*

We say that a range $\langle s,\ e \rangle$ (directly) *invokes* a range $\langle s',\ e' \rangle$ if $\langle s',\ e' \rangle = range(u)$ for some *perform* vertex u in the body of $\langle s,\ e \rangle$. We say that a range $\langle s,\ e \rangle$ *spans* an *exit* vertex e' if e' is an *exit* vertex in the body of $\langle s,\ e \rangle$ and $e' \neq e$.

Fig. 11 and Fig. 12 illustrate the above definitions using the example of Fig. 8.

Definition 3 *A range r is said to be* structurally-LIFO *(or* SLIFO*) iff (i) Every range that r invokes is SLIFO, and (ii) For every exit vertex e that r spans, every range in PerfRanges that terminates at e is SLIFO.*

We say that an *exit* vertex e is SLIFO if every range that terminates at e is SLIFO. Thus, the second condition above requires that every *exit* vertex e that r spans be SLIFO.

```
proc me-mx() {
  me: ...
  pa: ae-ax();
  pe: ee-ex();
      pb-mx();
      return; }

proc ae-ax() {
  ae: ...
  pc: ce-cx();
      be();
      ce-cx();
      ax();
      return; }

proc be-bx() {
      be();
      ce-cx();
      ax();
  d1: ...
  d2: ...
  ee: ...
      if ... pb-mx();
      ex();
      if active(ex)
        pb-abnml-cont();
  bx: if active(bx)
        mx-abnml-cont();
      return;  }

proc ee-ex() {
  ee: ...
      if ... pb-mx();
      ex();
      return; }

proc pb-mx() {
  pb: be-bx();
  mx: exit;
      return; }

proc ce-cx() {
  ce: ...
  cx: ...
      return; }

proc be() {
  be: ...
      return; }

proc ax() {
  ax: ...
      return; }

proc ex() {
  ex: ...
      return(); }

proc pb-abnml-cont() {
  pb: be-bx();
      mx-abnml-cont();
      return;  }

proc mx-abnml-cont() {
  mx: exit;
      return  }
```

Figure 9: Procedurally well-structured representation of the examples in Fig. 8 produced by our algorithm.

Refinements

An interesting variation on the above definition of SLIFO ranges is to add the requirement that a SLIFO range have no program terminating statement (such as `STOP RUN`) unless it is the main program. This stricter definition is of interest because a Cobol program may invoke another Cobol program multiple times, using the `CALL` statement. Some forms of the `CALL` statement cause the called program to begin execution without initializing its exit continuations to the appropriate initial state. In this case, mines left behind during a previous execution of the program will continue to be alive in the current execution. The stricter definition accounts for such a possibility, while Definition 3 is more useful in the common case where the exit continuation state is not preserved across multiple executions of the program. The algorithm we present can be trivially adapted for the stricter definition.

Range	Body
ME:MX	{ s_m, ME, PA, PE, PB }
AE:AX	{ s_a, AE, PC, s_b, BE, s_c, CE, CX, e_c, AX, e_a }
EE:EX	{ s_e, EE, EX, e_e, PB, MX }
BE:BX	{ s_b, BE, s_c, CE, CX, e_c, AX, e_a, D1, D2, s_e, EE, EX, e_e, BX, e_b, PB, MX }
CE:CX	{ s_c, CE, CX, e_c }

Figure 11: This table depicts the bodies of the various `PERFORM` ranges in the example of Fig. 8.

Range	Ranges Invoked	Exits Spanned
ME:MX	{ AE:AX, EE:EX, BE:BX }	{ }
AE:AX	{ CE:CX }	{ e_c }
EE:EX	{ BE:BX }	{ }
BE:BX	{ BE:BX }	{ e_c, e_a, e_e }
CE:CX	{ }	{ }

Figure 12: This table depicts the set of invoked ranges and exits spanned for every `PERFORM` range in the example of Fig. 8.

4 Simple Algorithms

In this section, we outline a simple worklist-based algorithm for identifying the set of SLIFO ranges and SLIFO *exit* vertices in a given program. In Section 5, we will refine this algorithm to in such a way that it has linear complexity for programs containing only SLIFO ranges.

Control-Flow Analysis

Our simple algorithm for identifying SLIFO ranges will make use of the following data structures: For every range r in the program, we maintain *nlri*(r), the set of all ranges that r invokes that have not yet been marked SLIFO, as well as *nles*(r), the set of all *exit* vertices that r spans that have not yet been marked SLIFO. For every *exit* vertex e, we maintain *nlrta*(e), the set of all ranges that terminate at e that have not yet been marked SLIFO.

Initially, none of the *exit* vertices or ranges are marked SLIFO. The first step in the algorithm is to construct the body of every range in the program, which can be done using a simple graph traversal. This lets us identify information about how ranges can invoke one another, and which range spans which *exit* vertex. The data structures above can then be initialized in a straightforward way. The initial information computed in the case of the example in Fig. 8 is shown in Figures 11 and 12.

The next step in the algorithm is to inductively identify the set of all SLIFO ranges and *exit* vertices. In particular, a range r is identified and marked as being SLIFO if and when both *nlri*(r) and *nles*(r) become empty. The ranges initially identified as being SLIFO correspond to the "base case" of the inductive definition: these are ranges that do not invoke any other range and do not span any *exit* vertex. In our example program, the range `CE` through `CX` is identified as being SLIFO in this fashion.

When a range r is identified as being SLIFO, the data structures are updated appropriately by

1. removing r from *nlri*(r') for every r' that invokes r, and

2. removing r from *nlrta*(e), where e is the *exit* vertex of range r

Similarly, an *exit* vertex e is identified and marked SLIFO if and when *nlrta*(e) becomes empty. When an *exit* vertex e is identified as being SLIFO, the data structures are updated by removing e from *nles*(r) for every range r that spans e.

As the data structures are updated, other ranges and *exit* vertices may be identified as being SLIFO and are processed appropriately. Thus, in our example, once the range `CE` through `CX` and the *exit* vertex of this range are identified as being SLIFO, the range `AE` through `AX` is identified as being SLIFO. Consequently, the *exit* vertex of this range is also identified as being SLIFO. The worklist-based algorithm terminates when every range and vertex identified as being SLIFO has been processed as above. In our example, for instance, no further SLIFO ranges or SLIFO *exit* vertices can be identified and the algorithm terminates.

The algorithm above runs in time linear in the sum of the sizes of the body of all ranges in the program. In the case where the bodies of different ranges are disjoint, the algorithm is linear in the size of the input graph and, hence, asymptotically optimal. However, as we have observed before, the bodies of different `PERFORM` ranges in a program may overlap. As a result, in the worst case, the algorithm can take time quadratic in the size of the input program. We will describe an improved algorithm in Section 5 that handles such cases better.

Generating a Procedural Representation

We now turn our attention to the problem of generating a program's procedural representation. For a SLIFO range, generating the representation is quite straightforward: Given a range r, we define *lifoProc*(r) to be (a copy of) the subgraph induced by the set of vertices *body*(r) in the implicit CFG, but without any outgoing edge at the (copy of the) *exit* vertex of r.

Generating a procedural representation for non-SLIFO ranges is more complicated. Let r be a non-SLIFO range. If a non-SLIFO *exit* vertex e other than the exit of r is encountered during the execution of r, we regard such an exit as a mine, and assume that execution could proceed to the successor of some *perform* vertex that invoked a range terminating at e, rather than following the *fallThrough* edge of e. Our representation must account for this possibility.

One conservative way of accounting for non-SLIFO behavior is to create a single procedure to represent all non-SLIFO ranges, modeling any control-flow between two different non-SLIFO ranges as simple intraprocedural control-flow. This is straightforward to do, since any `PERFORM` invocation of a non-SLIFO range must occur within a non-SLIFO range. However, it is possible to produce a more precise representation of non-SLIFO ranges without excessive overhead.

Observe that the `PERFORM` invocation of a non-SLIFO range r does not directly cause non-SLIFO execution behavior. For any *exit* vertex e, let *nltargets*(e) denote the set of successors of *perform* vertices that invoke some non-SLIFO range terminating at e. Non-SLIFO execution behavior arises *only* when (i) a non-SLIFO *exit* vertex e other than the *exit* vertex of r is encountered during the execution of r and control-flow proceeds to some vertex in *nltargets*(e) or (ii) the exit vertex of a recursive *perform* is reached.

For any vertex s that is in some set *nltargets*(e) we create a procedure cc_s to capture the possible subsequent execution behavior of the program (*i.e.*, the "continuation") when control reaches s as described in condition (i) above. This procedure contains (copies of) any vertex that is reachable from s using any sequence of intraprocedural edges, *fallThrough* edges *or return edges*. (Note that here we use the ICFG augmented by the return edges.)

Given such auxiliary procedures, we create a procedural representation for non-SLIFO ranges as follows: *nonLifoProc*(r) is defined to be the graph *lifoProc*(r) augmented as follows for every non-SLIFO *exit* vertex e spanned by r: for every s in *nltargets*(e), add a vertex c_s representing a procedure call to cc_s as well as an edge from e to c_s to the graph. If the range r is recursive, the *exit* vertex e of r itself needs to be modified similarly, to represent a call to one of the procedures in $\{\ cc_s \mid s \in \mathit{nltargets}(e)\ \}$.

Fig. 9 illustrates some aspects of the algorithm above, even though it is intended primarily to illustrate the procedural representation created by the refined algorithm of Section 5. For instance, the range `BE-BX` spans the non-SLIFO exit vertex of range `EE-EX`, and the code in procedure `be-bx` shows how we create a call to the continuation procedure `pb-abnml-cont` to represent this non-SLIFO *exit* vertex. Similarly, the range `BE-BX` is itself potentially recursive, and, hence, its own *exit* vertex is represented by a call to the continuation procedure `mx-abnml-cont`.

5 Achieving Code Reuse Procedurally

The algorithm of Section 4 handles *overlapping* ranges (such as those in Fig. 3 correctly, but potentially inefficiently, by duplicating the shared code as necessary. In the worst case, the size of the generated procedural representation, as well as the time required to generate it, can be quadratic in the size of the input program. Aside from efficiency concerns, such an approach also has drawbacks in the context of a program understanding tool. We now describe how our earlier algorithm can be refined to deal with shared code better

Identifying Reusable Units

The first step in the modified algorithm is to identify "reusable" code units. In the example of Fig. 8, the code contained in the range `CE` through `CX` is an example of a "reusable unit". That is, this subrange has the following property: for every `PERFORM` range in the program, the subrange is either completely contained in (the body of) that range or is completely disjoint from (the body of) that range. Further, this subrange is a maximal subrange that has this property. The general problem, however, is more complex than simply identifying the overlap between a collection of line segments in a single dimension: we need to identify the units of sharing between a *collection* of graphs, since each `PERFORM` range corresponds to a graph (e.g., due to `GOTO` statements).

A *sparse evaluation representation* can be used to identify appropriate units of sharing. In particular, we use the algorithm outlined in [7] to identify the sparse evaluation representation of the ICFG with respect to the set of *entry* and *exit* vertices in the graph. We briefly describe this algorithm here for the sake of completeness; the reader is referred to [7] for a more complete explanation of the algorithm:

1. Let $C_1, \cdots, C_k$ denote the maximal strongly connected components (SCCs) of the subgraph of the ICFG induced by the set of *computational* and *perform* vertices in topological sort order.

2. Collapse each SCC C_i into a single, new, vertex w_i.

3. Visit vertices w_1 to w_k in that order, merging the visited vertex with its predecessor *if it has a unique predecessor*.

We will refer to the vertices of the resulting graph as supervertices, and the resulting graph as the sparse graph. The supervertices identify the units of sharing in the ICFG. In particular, each supervertex corresponds to a subgraph of the ICFG, which has been collapsed into the single supervertex. Further, the subgraph corresponding to any supervertex is guaranteed to contain at most one (range) *entry* or (range) *exit* vertex. We refer to the supervertex as a *entry* supervertex if the corresponding subgraph has a single (range) *entry* vertex, as a *exit* supervertex if the corresponding subgraph has a single (range) *exit* vertex, and as a *join* supervertex otherwise (*i.e.*, if the corresponding subgraph has no *entry* or *exit* vertex). We will abuse notation and use the same name for an *entry* (*exit*) supervertex as well as the *entry* (*exit*) vertex it contains. It is worth noting that an *exit* supervertex u denotes a region in the ICFG that may be executed *after the fallThrough edge at vertex* u *is taken*. Fig. 13(a) illustrates the sparse graph generated by applying the above algorithm to the ICFG shown in Fig. 10.

Control-Flow Analysis

We now show how our earlier algorithm for identifying the set of SLIFO ranges and SLIFO *exit* vertices can be adapted to work with the sparse graph using a graph transformational approach.

For every supervertex u we maintain *nlri*(u), the set of all ranges invoked from the subgraph corresponding to u that have not yet been marked SLIFO. For every *exit* supervertex e, we also maintain *nlrta*(e), the set of all ranges that terminate at e that have not yet been marked SLIFO.

Once these data structures have been initialized, we apply a sequence of transformations to the sparse graph as well as these data structures. Each transformation can enable further transformations. Our algorithm is a worklist-driven one that terminates when no more transformations are applicable. The transformations we use are as follows:

i. Let $\langle s,\ e\rangle$ be a `PERFORM` range. If and when the set *nlri*(s) becomes empty, and s has no successors other than e, the range $\langle s,\ e\rangle$ is identified and marked as being SLIFO. At this point, $\langle s,\ e\rangle$ is removed from any set *nlri*(u) in which it occurs, as well as from the set *nlrta*(e).

ii. Let e be an *exit* supervertex. If and when the set *nlrta*(e) becomes empty, e is marked as being a SLIFO *exit* vertex.

iii. Let e be an *exit* supervertex. If and when both *nlrta*(e) and *nlri*(e) become empty, the vertex e is "eliminated" by replacing vertex e as well as all edges incident on e by direct edges from every predecessor of e to every successor of e. (If e has any self loop, the self loop is simply deleted.)

iv. Let j denote a *join* supervertex. If and when *nlri*(j) become empty, vertex j is eliminated (in a similar fashion).

v. Let s denote an *entry* supervertex. If and when *nlri*(s) become empty, vertex s can be bypassed by replacing all incoming edges of s by direct edges from predecessors of s to successors of s. We do not, however, immediately eliminate vertex s. We eliminate vertex s (and all its outgoing edges) once all ranges starting at s have been marked SLIFO and s has been bypassed.

Our algorithm simply applies the transformations above to the graph repeatedly until no more transformations are applicable. At this point, we are guaranteed to have found all SLIFO ranges and vertices. While the transformations can be applied in any order (from the point of view of correctness), our algorithm applies the *entry*-bypass transformation only if no other transformation is applicable, as this improves the efficiency of the algorithm. Fig. 13 illustrates how the algorithm above transforms the sparse graph of the example in Fig. 8, identifying SLIFO ranges during the process.

Generating a Procedural Representation

We now turn our attention to the problem of generating a procedural representation of the program.

SLIFO Procedures We first focus on generating a procedural representation for SLIFO ranges. Let $\langle s,\ e\rangle$ be a SLIFO range. As before, the essential idea is to identify the "body" of this range by performing a graph traversal starting at vertex s and stopping at vertex e, just as in our original algorithm. However, however, we elaborate on this basic theme in two respects:

First, we perform the traversal over the sparse graph, rather than the original graph. (This creates a "skeletal representation" for the procedure in terms of supervertices. The complete representation can be generated by replacing every supervertex by the corresponding subgraph, but more about this later.)

Second, the graph traversal is modified to identify other SLIFO ranges that are completely contained within the range $\langle s,\ e\rangle$. In particular, if a SLIFO range $\langle s',\ e'\rangle$ is completely contained within the range $\langle s,\ e\rangle$, then we will insert a call to a procedure representing $\langle s',\ e'\rangle$ within the generated representation for $\langle s,\ e\rangle$, instead of creating a complete copy of the range $\langle s',\ e'\rangle$. We will refer to this process as *abstracting* the range $\langle s',\ e'\rangle$.

Determining whether one range can be abstracted from another is straightforward after the bodies of the two ranges have been constructed, requiring only a containment test. However, it is possible to test for containment efficiently on the fly, as the procedure bodies are being constructed: Assume that the graph traversal (during the construction of the body of $\langle s,\ e\rangle$) reaches the *entry* vertex s' of some SLIFO range $\langle s',\ e'\rangle$. The body of range $\langle s',\ e'\rangle$ is guaranteed to be contained within the body of range $\langle s,\ e\rangle$ iff $\langle s',\ e'\rangle$ was marked as SLIFO before $\langle s,\ e\rangle$ was marked as SLIFO or $e = e'$. Hence, if we remember the order in which ranges are marked SLIFO during the control-flow analysis phase, this test can be performed efficiently.

Note that many different ranges starting at vertex s' may, in fact, be abstractable from range $\langle s,\ e\rangle$. In this case, we would like to choose the candidate range that is maximal among all candidate abstractable ranges. Such a range is readily identified by simply selecting the abstractable range that was marked SLIFO *last* (among all candidates). This guarantees the desired maximality property.

Once an abstractable range $\langle s',\ e'\rangle$ has been selected, instead of creating a copy of the supervertex s' in the body of $\langle s,\ e\rangle$, we create a call to $\langle s',\ e'\rangle$, and continue the graph traversal from the *exit vertex* e'.

Non-SLIFO Procedures The procedural representation for non-SLIFO ranges is constructed from the sparse graph, just as in our original algorithm, modified to identify occurrences of abstractable SLIFO ranges and to replace them by procedure calls.

Continuation Procedures As before, we also need to create a representation of the "continuation" procedure cc_t for every non-SLIFO target t. Here too, we simply adapt our original algorithm to replace occurrences of abstractable SLIFO ranges by procedure calls. Unlike in the construction of the SLIFO and non-SLIFO procedures, however, we can not use the sparse graph for generating the continuation procedures, as the "entry point" of these procedures, which are the successors of certain *perform* vertices, may correspond to an "interior" vertex of the subgraph corresponding to a supervertex. In other words, the bodies of continuation procedures may include *parts* of subgraphs corresponding to supervertices. Hence, the construction of these procedures utilizes the original graph.

It is also worth mentioning here that shared code between different "continuation" procedures can be factored out easily. In particular, assume that the same vertex u occurs in two different continuation procedures cc_{t_1} and cc_{t_2}. The possible paths program execution can follow beyond vertex u is the same in both procedures (in our conservative model). In other words, the continuation is the same for both occurrences of u. Hence, we can create a single

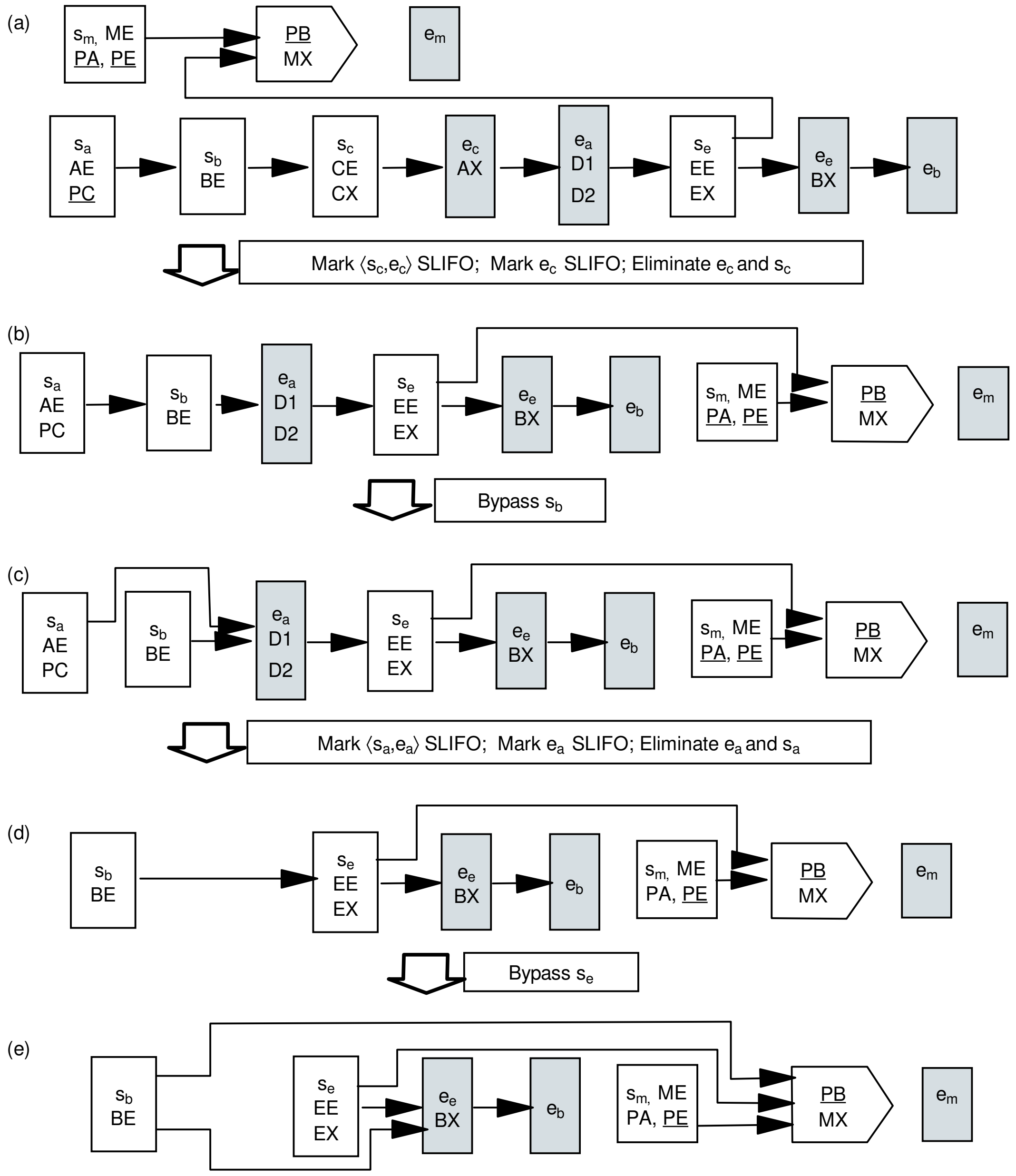

Figure 13: An illustration of how our algorithm identifies SLIFO ranges for the example of Fig. 8. Unshaded rectangles denote *entry* supervertices, while shaded rectangles denote *exit* supervertices. Perform invocations to ranges not yet marked SLIFO are shown underlined.

"continuation procedure" cc_u for vertex u and call it from both cc_{t_1} and cc_{t_2} instead of duplicating vertices in both procedures.

Let T denote the set of all non-SLIFO targets. The set of all vertices for which we will need to create a continuation procedure to avoid duplication as above is simply the iterated join set [1] of T. We simply create a continuation procedure cc_t for every t in T'. The body of the continuation procedure cc_t is constructed using a simple graph traversal from t. When the traversal reaches any vertex t' in T', instead of continuing the traversal beyond t', we insert a call to $cc_{t'}$ instead.

Supervertex Procedures We now turn our attention to a final matter related to code sharing. The discussion above was concerned primarily with constructing procedural representations in terms of supervertices. Once such a representation has been generated, a *complete* representation, in terms of the vertices of the original graph, can be generated by replacing every occurrence of a supervertex by the subgraph it denotes. Consider now a supervertex that occurs in multiple procedure bodies. If the subgraph corresponding to the supervertex is a single-entry/single-exit region, then we can avoid duplicating this region in multiple procedure bodies by creating a *procedure* corresponding to this region, and replacing all occurrences of the supervertex by a call to this procedure.

Fig. 9 illustrates the procedural representation created by applying our algorithm to the example of Fig. 8.

6 Experience

The algorithm presented in this paper is a slight refinement of a technique implemented in an IBM program understanding product, *IBM VisualAge for Cobol Professional Redeveloper*. In the tool, the algorithm is used both to generate a visual depiction of the procedurally well-structured representation of the program, and as a prerequisite to generating an intermediate representation for use with interprocedural *program slicing* [9].

Given the size of the Cobol programs that the tool is intended to address (up to about 100,000 lines), we were somewhat concerned that the quadratic worst-case time and output size complexity of our algorithm might pose a problem. In practice, this was not the case; indeed, the program analysis time was dominated not by control-flow rationalization, but by subsequent dataflow analysis required as a prerequisite to building a program representation suitable for slicing.

The test suite used for the program included 13 reasonably large (1000-25,000 line) programs contributed by IBM customers. It is interesting to note that our analysis found only two non-SLIFO `PERFORM` ranges among the 808 `PERFORM` ranges considered by the analysis in those test cases.

7 Future Work

There are a number of interesting topics that we leave for future work:

More precise characterization of PERFORM validity The notion of valid paths defined in Section 2 is a characterization only of *well-behaved*—LIFO—`PERFORM` ranges. Although our procedural representation allows precise static analysis with respect to valid paths in the procedural representation of SLIFO ranges, such analysis is less precise when applied to the procedural representation of non-SLIFO ranges.

It would therefore be useful to provide a precise static characterization of *all* `PERFORM` executions possible under a particular operational semantics, e.g., that illustrated in Fig. 6. Since the behavior of a `PERFORM`'s exit is entirely characterized by the contents of its exit continuation, and since the number of addresses that can be stored in any continuation is finite, one could provide a notion of static `PERFORM` validity that would capture exactly the continuation states that can occur under the assumption that all normal intraprocedural paths are executable. For example, one could build a finite-state machine whose states represent feasible continuation states, then use this state machine to constrain the set of valid paths. One can then define dataflow analyses that are precise with respect to the state machine's characterization of valid paths [3].

Alternate procedural representations In general, it appears that building a well-structured procedural representation of *minimal* size (e.g., one that maximizes sharing while minimizing the number of call sites) is quite difficult, probably NP-hard, even for programs containing only SLIFO ranges. However, if instead of requiring that procedures have a single entry, we allow them to have *multiple* entries, it is straightforward to build a representation whose size is linear in the input by building a multi-entry procedure for each range *exit*. Adapting context-sensitive analyses to multi-entry procedural units is usually not difficult.

A number of refinements are also possible in constructing procedural representations from supervertices. For example, if a supervertex is not single-entry/single-exit, we could abstract maximal single-entry/single-exit regions within the supervertex into procedures.

Faster analysis Our current SLIFO analysis has quadratic worst-case complexity. Given the simple inductive definition for SLIFO ranges, it seems plausible that a linear-time algorithm is possible.

Unified algorithm Our algorithm to construct a procedural representation requires an analysis phase as well as several procedure-building phases. It would be interesting to attempt to design an algorithm that consolidates these phases into a single set of graph transformations whose output is the procedural representation.

References

[1] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. Efficiently computing static single assignment form and control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4 (Oct. 1991), 452–490.

[2] DOCK, V. T. *Structured Cobol: American National Standard*. West Publishing Co., St. Paul, MN, 1979.

[3] HOLLEY, L. H., AND ROSEN, B. K. Qualified data flow problems. *IEEE Trans. Software Eng. SE-7*, 1 (January 1981), 60–78.

[4] IBM CORPORATION. *IBM Cobol for MVS & VM Language Reference*. IBM Corporation, San Jose, CA, 1995. Publication Number SC26-4769-01.

[5] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (IS0). Committee Draft 1.5, Proposed revision of ISO 1989:1985, Programming Language Cobol. Available at URL `http://people.ne.mediaone.net/pennyjs/home.htm`, April 1999.

[6] MUCHNICK, S. S. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, San Francisco, 1997.

[7] RAMALINGAM, G. On sparse evaluation representations. In *Fourth International Static Analysis Symposium* (1997), vol. 1302 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–15.

[8] REPS, T., HORWITZ, S., AND SAGIV, M. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages* (1995), pp. 49–61.

[9] REPS, T., HORWITZ, S., SAGIV, M., AND ROSAY, G. Speeding up slicing. In *Proc. Second ACM SIGSOFT Symp. on the Foundations of Software Engineering* (New Orleans, December 1994), pp. 11–20. Published in ACM SIGSOFT Software Engineering Notes 19, 5, Dec. 1994, pp. 11–20.

[10] SHARIR, M., AND PNUELI, A. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds. Prentice-Hall, Englewood Cliffs, NJ, 1981, ch. 7, pp. 189–233.

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