Understanding Interleaved Code

Spencer Rugaber, Kurt Stirewalt, and Linda M. Wills
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
{spencer, kurt, linda}@cc.gatech.edu

Abstract

Complex programs often contain multiple, interwoven strands of computation, each responsible for accomplishing a distinct goal. The individual strands responsible for each goal are typically delocalized and overlap rather than being composed in a simple linear sequence. We refer to these code fragments as being *interleaved*. Interleaving may be intentional–for example, in optimizing a program, a programmer might use some intermediate result for several purposes–or it may creep into a program unintentionally, due to patches, quick fixes, or other hasty maintenance practices. To understand this phenomenon, we have looked at a variety of instances of interleaving in actual programs and have distilled characteristic features. This paper presents our characterization of interleaving and the implications it has for tools that detect interleaving and extract the individual strands of computation. Our exploration of interleaving has been done in the context of a case study of a corpus of production mathematical software, written in Fortran from the Jet Propulsion Laboratory. This paper also describes our experiences in developing tools to detect interleaving in this software, driven by the application of program comprehension to improving the description of this software library's components. This in turn aids in the automated component-based synthesis of software using the library.

With every leaf a miracle.
— Walt Whitman.

1 Introduction

Imagine being handed a software system you have never seen before. Perhaps you need to track down a bug, rewrite the software in another language or extend it in some way. We know that software maintenance tasks such as these consume the majority of software costs [3], and we know that reading and understanding the code requires more effort than actually making the changes [9]. But we don't know what makes understanding the code itself so difficult.

Letovsky has observed that programmers engaged in software understanding activities typically ask "how" questions and "why" questions [17]. The former require an in-depth knowledge of the programming language and the ways in which programmers express their software designs. This includes knowledge of common algorithms and data structures and even concerns style issues such as indentation and use of comments. Nevertheless, the answers to "how" questions can be derived from the program text. "Why" questions are more troubling. Answering them requires not only comprehending the program text but relating it to the program's *purpose*—solving some sort of problem. And the problem being solved may not be explicitly stated in the program text; nor is the rationale the programmer had for choosing the particular solution usually visible.

This paper is concerned with a specific difficulty that arises when trying to answer "why" questions about computer programs. In particular, it is concerned with the phenomenon of "interleaving" in which one section of a program accomplishes several purposes, and disentangling the code responsible for each purposes is difficult. Unraveling interleaved code involves discovering the purpose of each strand of computation, as well as understanding why the programmer decided to interleave the strands. To demonstrate this problem, we examine an example program in a step-by-step fashion, trying to answer the questions "why is this program the way it is?" and "what makes it difficult to understand?"

1.1 NPEDLN

The Fortran program, called `NPEDLN`, is part of the `SPICELIB` library obtained from the Jet Propulsion Laboratory and intended to help space scientists analyze data returned from space missions. The acronym `NPEDLN` stands for Nearest Point on Ellipsoid to Line. The ellipsoid is specified by the length of its three semi-axes (`A`, `B`, and `C`), which are oriented with the x, y, and z coordinate axes. The line is specified by a point (`LINEPT`) and a direction vector (`LINEDR`). The nearest point is contained in a variable called `PNEAR`. The full program consists of 565 lines; an abridged version can be found in the Appendix with a brief description of subroutines it calls and variables it uses. The executable statements, with comments and declarations removed, are shown in Figure 1.

The lines of code in `NPEDLN` that actually compute the nearest point are somewhat hard to locate. One reason for this has to with error checking. It turns out that `SPICELIB` includes an elaborate mechanism for reporting and recovering from errors, and roughly half of the code in `NPEDLN` is used for this purpose. We have indicated those lines by shading in Figure 2. The important point to note is that although it is natural to program in a way that intersperses error checks with computational code, it is not necessary to do so. In principal, an entirely separate routine could be constructed to make the checks and `NPEDLN` called only when all the checks are passed. Although this approach would require redundant computation and potentially more total lines of code, the resultant computations in `NPEDLN` would be shorter and easier to follow.

In some sense, the error handling code and the rest of the routine realize *independent* plans. We use the term *plan* to denote a description or representation of a computational structure that the designers have proposed as a way of achieving some purpose or goal in a program.[1] Plans can occur

[1]This definition is distilled from definitions in [18, 28, 34]. Note that a plan is not necessarily stereotypical or used repeatedly; it may be novel or idiosyncratic. Following [28, 34], we reserve the term *cliché* for a plan that represents a standard, stereotypical form, which can be detected by recognition techniques, such as [12, 17, 16, 25, 29, 40].

```
SUBROUTINE NPEDLN ( A, B, C, LINEPT, LINEDR,
                    PNEAR, DIST )
C
   IF ( RETURN () ) THEN
      RETURN
   ELSE
      CALL CHKIN ( 'NPEDLN' )
   END IF
C
   CALL UNORM ( LINEDR, UDIR, MAG )
   IF ( MAG .EQ. 0 ) THEN
      CALL SETMSG( 'Line direction vector
                    is the zero vector. ' )
      CALL SIGERR( 'SPICE(ZEROVECTOR)' )
      CALL CHKOUT( 'NPEDLN' )
      RETURN
   ELSE IF (         ( A .LE. 0.D0 )
     .        .OR.   ( B .LE. 0.D0 )
     .        .OR.   ( C .LE. 0.D0 ) )
        THEN
      CALL SETMSG ( 'Semi-axes: A = #,
                      B = #,  C = #.'  )
      CALL ERRDP  ( '#', A )
      CALL ERRDP  ( '#', B )
      CALL ERRDP  ( '#', C )
      CALL SIGERR ('SPICE(INVALIDAXISLENGTH)')
      CALL CHKOUT ( 'NPEDLN' )
      RETURN
   END IF
   SCALE  =  MAX ( DABS(A), DABS(B), DABS(C) )
   SCLA   =  A / SCALE
   SCLB   =  B / SCALE
   SCLC   =  C / SCALE
   IF (     ( SCLA**2  .LE.  0.D0 )
     . .OR. ( SCLB**2 .LE. 0.D0 )
     . .OR. ( SCLC**2 .LE. 0.D0 ) )  THEN
      CALL SETMSG ( 'Semi-axis too small:
                    A = #, B = #, C = #. ' )
      CALL ERRDP  ( '#', A )
      CALL ERRDP  ( '#', B )
      CALL ERRDP  ( '#', C )
      CALL SIGERR ('SPICE(DEGENERATECASE)')
      CALL CHKOUT ( 'NPEDLN' )
      RETURN
   END IF
C  Scale LINEPT.
   SCLPT(1)  =  LINEPT(1) / SCALE
   SCLPT(2)  =  LINEPT(2) / SCALE
   SCLPT(3)  =  LINEPT(3) / SCALE
   CALL VMINUS (UDIR, OPPDIR )
```

```
   CALL SURFPT (SCLPT, UDIR, SCLA, SCLB,
                SCLC, PT(1,1), FOUND(1))
   CALL SURFPT (SCLPT, OPPDIR, SCLA,
                SCLB, SCLC, PT(1,2), FOUND(2))

   DO 50001
     .   I = 1, 2
      IF ( FOUND(I) ) THEN
         DIST  =  0.0D0
         CALL VEQU   ( PT(1,I),  PNEAR )
         CALL VSCL   ( SCALE,PNEAR, PNEAR )
         CALL CHKOUT ( 'NPEDLN' )
         RETURN
      END IF
50001 CONTINUE
C
   NORMAL(1)  =  UDIR(1) / SCLA**2
   NORMAL(2)  =  UDIR(2) / SCLB**2
   NORMAL(3)  =  UDIR(3) / SCLC**2
   CALL NVC2PL ( NORMAL, 0.D0, CANDPL )
   CALL INEDPL ( SCLA, SCLB, SCLC, CANDPL,
                 CAND, XFOUND )
   IF ( .NOT. XFOUND ) THEN
      CALL SETMSG ( 'Candidate ellipse could not
                     be found.'  )
      CALL SIGERR ( 'SPICE(DEGENERATECASE)' )
      CALL CHKOUT ( 'NPEDLN' )
      RETURN
   END IF
   CALL NVC2PL ( UDIR,  0.D0,   PRJPL )
   CALL PJELPL ( CAND,  PRJPL,  PRJEL )
C
   CALL VPRJP  ( SCLPT,   PRJPL,  PRJPT  )
   CALL NPELPT ( PRJPT,   PRJEL,  PRJNPT )
   DIST = VDIST ( PRJNPT, PRJPT )
C
   CALL VPRJPI ( PRJNPT, PRJPL, CANDPL, PNEAR,
                 IFOUND )
   IF ( .NOT. IFOUND ) THEN
      CALL SETMSG ( 'Inverse projection could not
                     be found.'  )
      CALL SIGERR ( 'SPICE(DEGENERATECASE)' )
      CALL CHKOUT ( 'NPEDLN' )
      RETURN
   END IF
   CALL VSCL ( SCALE,  PNEAR,  PNEAR )
   DIST    =   SCALE * DIST
   CALL CHKOUT ( 'NPEDLN' )
   RETURN
   END
```

Figure 1: NPEDLN minus comments and declarations.

```
SUBROUTINE NPEDLN ( A, B, C, LINEPT, LINEDR,
                    PNEAR, DIST )
C
   IF ( RETURN () ) THEN
      RETURN
   ELSE
      CALL CHKIN ( 'NPEDLN' )
   END IF
C
   CALL UNORM ( LINEDR, UDIR, MAG )
   IF ( MAG .EQ. 0 ) THEN
      CALL SETMSG( 'Line direction vector
                    is the zero vector. ' )
      CALL SIGERR( 'SPICE(ZEROVECTOR)' )
      CALL CHKOUT( 'NPEDLN' )
      RETURN
   ELSE IF (         ( A .LE. 0.D0 )
     .        .OR.   ( B .LE. 0.D0 )
     .        .OR.   ( C .LE. 0.D0 ) )
        THEN
      CALL SETMSG ( 'Semi-axes: A = #,
                      B = #,  C = #.'  )
      CALL ERRDP  ( '#', A )
      CALL ERRDP  ( '#', B )
      CALL ERRDP  ( '#', C )
      CALL SIGERR ('SPICE(INVALIDAXISLENGTH)')
      CALL CHKOUT ( 'NPEDLN' )
      RETURN
   END IF
   SCALE  =  MAX ( DABS(A), DABS(B), DABS(C) )
   SCLA   =  A / SCALE
   SCLB   =  B / SCALE
   SCLC   =  C / SCALE
   IF (     ( SCLA**2  .LE.  0.D0 )
     . .OR. ( SCLB**2 .LE. 0.D0 )
     . .OR. ( SCLC**2 .LE. 0.D0 ) )  THEN
      CALL SETMSG ( 'Semi-axis too small:
                    A = #, B = #, C = #. ' )
      CALL ERRDP  ( '#', A )
      CALL ERRDP  ( '#', B )
      CALL ERRDP  ( '#', C )
      CALL SIGERR ('SPICE(DEGENERATECASE)')
      CALL CHKOUT ( 'NPEDLN' )
      RETURN
   END IF
C  Scale LINEPT.
   SCLPT(1)  =  LINEPT(1) / SCALE
   SCLPT(2)  =  LINEPT(2) / SCALE
   SCLPT(3)  =  LINEPT(3) / SCALE
   CALL VMINUS (UDIR, OPPDIR )
```

```
   CALL SURFPT (SCLPT, UDIR, SCLA, SCLB,
                SCLC, PT(1,1), FOUND(1))
   CALL SURFPT (SCLPT, OPPDIR, SCLA,
                SCLB, SCLC, PT(1,2), FOUND(2))

   DO 50001
     .   I = 1, 2
      IF ( FOUND(I) ) THEN
         DIST  =  0.0D0
         CALL VEQU   ( PT(1,I),  PNEAR )
         CALL VSCL   ( SCALE,PNEAR, PNEAR )
         CALL CHKOUT ( 'NPEDLN' )
         RETURN
      END IF
50001 CONTINUE
C
   NORMAL(1)  =  UDIR(1) / SCLA**2
   NORMAL(2)  =  UDIR(2) / SCLB**2
   NORMAL(3)  =  UDIR(3) / SCLC**2
   CALL NVC2PL ( NORMAL, 0.D0, CANDPL )
   CALL INEDPL ( SCLA, SCLB, SCLC, CANDPL,
                 CAND, XFOUND )
   IF ( .NOT. XFOUND ) THEN
      CALL SETMSG ( 'Candidate ellipse could not
                     be found.'  )
      CALL SIGERR ( 'SPICE(DEGENERATECASE)' )
      CALL CHKOUT ( 'NPEDLN' )
      RETURN
   END IF
   CALL NVC2PL ( UDIR,  0.D0,   PRJPL )
   CALL PJELPL ( CAND,  PRJPL,  PRJEL )
C
   CALL VPRJP  ( SCLPT,   PRJPL,  PRJPT  )
   CALL NPELPT ( PRJPT,   PRJEL,  PRJNPT )
   DIST = VDIST ( PRJNPT, PRJPT )
C
   CALL VPRJPI ( PRJNPT, PRJPL, CANDPL, PNEAR,
                 IFOUND )
   IF ( .NOT. IFOUND ) THEN
      CALL SETMSG ( 'Inverse projection could not
                     be found.'  )
      CALL SIGERR ( 'SPICE(DEGENERATECASE)' )
      CALL CHKOUT ( 'NPEDLN' )
      RETURN
   END IF
   CALL VSCL ( SCALE,  PNEAR,  PNEAR )
   DIST    =   SCALE * DIST
   CALL CHKOUT ( 'NPEDLN' )
   RETURN
   END
```

Figure 2: Code with error handling highlighted.

```
SUBROUTINE NPEDLN ( A, B, C, LINEPT, LINEDR,
                    PNEAR, DIST )
C
   CALL UNORM ( LINEDR, UDIR, MAG )
   SCALE  =  MAX ( DABS(A), DABS(B), DABS(C) )
   SCLA   =  A / SCALE
   SCLB   =  B / SCALE
   SCLC   =  C / SCALE
C
   SCLPT(1)  =  LINEPT(1) / SCALE
   SCLPT(2)  =  LINEPT(2) / SCALE
   SCLPT(3)  =  LINEPT(3) / SCALE
C
   CALL VMINUS (UDIR, OPPDIR )
   CALL SURFPT (SCLPT, UDIR, SCLA, SCLB,
                SCLC, PT(1,1), FOUND(1))
   CALL SURFPT (SCLPT, OPPDIR, SCLA,
                SCLB, SCLC, PT(1,2), FOUND(2))
   DO 50001
     .   I = 1, 2
      IF ( FOUND(I) ) THEN
         DIST  =  0.0D0
         CALL VEQU   ( PT(1,I),  PNEAR )
         CALL VSCL   ( SCALE,PNEAR, PNEAR )
         RETURN
      END IF
50001 CONTINUE
```

```
C
   NORMAL(1)  =  UDIR(1) / SCLA**2
   NORMAL(2)  =  UDIR(2) / SCLB**2
   NORMAL(3)  =  UDIR(3) / SCLC**2
   CALL NVC2PL ( NORMAL, 0.D0, CANDPL )
   CALL INEDPL ( SCLA, SCLB, SCLC, CANDPL,
                 CAND, XFOUND )
   CALL NVC2PL ( UDIR,  0.D0,   PRJPL )
   CALL PJELPL ( CAND,  PRJPL,  PRJEL )
C
   CALL VPRJP  ( SCLPT,   PRJPL,  PRJPT  )
   CALL NPELPT ( PRJPT,   PRJEL,  PRJNPT )
   DIST = VDIST ( PRJNPT, PRJPT )
C
   CALL VPRJPI ( PRJNPT, PRJPL, CANDPL, PNEAR,
                 IFOUND )
   CALL VSCL ( SCALE,  PNEAR,  PNEAR )
   DIST    =   SCALE * DIST
   RETURN
   END
```

Figure 3: The residual code without the error handling plan.

at any level of abstraction from architectural overviews to code. By extracting the error checking plan from NPEDLN, we get the much smaller and, presumably, more understandable program shown in Figure 3.

The structure of an understanding process begins to emerge: detect a plan, such as error checking, in the code and extract it, leaving a smaller and more coherent residue for further analysis; document the extracted plan independently; and note the way in which it interacts with the rest of the code.

We can apply this approach further to NPEDLN's residual code in Figure 3. NPEDLN has a primary goal of computing the nearest point on an ellipsoid to a specified line. It also has an orthogonal goal of ensuring that the computations involved have stable numerical behavior; that is, that the computations are accurate in the presence of a wide range of numerical inputs. A standard trick in numerical programming for achieving stability is to scale the data involved in a computation and then unscale the results. The code responsible for doing this in NPEDLN is scattered throughout the program's text. It is highlighted in the excerpt shown in Figure 4.

The *delocalized* nature of this "scale-unscale" plan makes it difficult to gather together all the pieces involved for consistent maintenance. It also gets in the way of understanding the rest of the code, since it provides distractions that must be filtered out. Letovsky and Soloway's cognitive study [18] shows the deleterious effects of delocalization on comprehension and maintenance.

When we extract the scale-unscale code from NPEDLN, we are left with a much smaller code segment shown in Figure 5 that more directly expresses the program's purpose: computing the nearest point.

```
SUBROUTINE NPEDLN ( A, B, C, LINEPT, LINEDR,
                    PNEAR, DIST )
C
   CALL UNORM ( LINEDR, UDIR, MAG )
   SCALE  =  MAX ( DABS(A), DABS(B), DABS(C) )
   SCLA   =  A / SCALE
   SCLB   =  B / SCALE
   SCLC   =  C / SCALE
C
   SCLPT(1)  =  LINEPT(1) / SCALE
   SCLPT(2)  =  LINEPT(2) / SCALE
   SCLPT(3)  =  LINEPT(3) / SCALE
C
   CALL VMINUS (UDIR, OPPDIR )
   CALL SURFPT (SCLPT, UDIR, SCLA, SCLB,
                SCLC, PT(1,1), FOUND(1))
   CALL SURFPT (SCLPT, OPPDIR, SCLA,
                SCLB, SCLC, PT(1,2), FOUND(2))
   DO 50001
     .   I = 1, 2
      IF ( FOUND(I) ) THEN
         DIST  =  0.0D0
         CALL VEQU   ( PT(1,I),  PNEAR )
         CALL VSCL   ( SCALE,PNEAR, PNEAR )
         RETURN
      END IF
50001 CONTINUE
```

```
C
   NORMAL(1)  =  UDIR(1) / SCLA**2
   NORMAL(2)  =  UDIR(2) / SCLB**2
   NORMAL(3)  =  UDIR(3) / SCLC**2
   CALL NVC2PL ( NORMAL, 0.D0, CANDPL )
   CALL INEDPL ( SCLA, SCLB, SCLC, CANDPL,
                 CAND, XFOUND )
   CALL NVC2PL ( UDIR,  0.D0,   PRJPL )
   CALL PJELPL ( CAND,  PRJPL,  PRJEL )
C
   CALL VPRJP  ( SCLPT,   PRJPL,  PRJPT  )
   CALL NPELPT ( PRJPT,   PRJEL,  PRJNPT )
   DIST = VDIST ( PRJNPT, PRJPT )
C
   CALL VPRJPI ( PRJNPT, PRJPL, CANDPL, PNEAR,
                 IFOUND )
   CALL VSCL ( SCALE,  PNEAR,  PNEAR )
   DIST    =   SCALE * DIST
   RETURN
   END
```

Figure 4: Code with scale-unscale plan highlighted.

```
SUBROUTINE NPEDLN ( A, B, C, LINEPT, LINEDR,
                    PNEAR, DIST )
C
   CALL UNORM ( LINEDR, UDIR, MAG )
C
   CALL VMINUS (UDIR, OPPDIR )
   CALL SURFPT (SCLPT, UDIR, SCLA, SCLB,
                SCLC, PT(1,1), FOUND(1))
   CALL SURFPT (SCLPT, OPPDIR, SCLA,
                SCLB, SCLC, PT(1,2), FOUND(2))
   DO 50001
     .   I = 1, 2
      IF ( FOUND(I) ) THEN
         DIST  =  0.0D0
         CALL VEQU   ( PT(1,I),  PNEAR )
         RETURN
      END IF
50001 CONTINUE
```

```
C
   NORMAL(1)  =  UDIR(1) / SCLA**2
   NORMAL(2)  =  UDIR(2) / SCLB**2
   NORMAL(3)  =  UDIR(3) / SCLC**2
   CALL NVC2PL ( NORMAL, 0.D0, CANDPL )
   CALL INEDPL ( SCLA, SCLB, SCLC, CANDPL,
                 CAND, XFOUND )
   CALL NVC2PL ( UDIR,  0.D0,   PRJPL )
   CALL PJELPL ( CAND,  PRJPL,  PRJEL )
C
   CALL VPRJP  ( SCLPT,   PRJPL,  PRJPT  )
   CALL NPELPT ( PRJPT,   PRJEL,  PRJNPT )
   DIST = VDIST ( PRJNPT, PRJPT )
C
   CALL VPRJPI ( PRJNPT, PRJPL, CANDPL, PNEAR,
                 IFOUND )
   RETURN
   END
```

Figure 5: The residual code without the scale-unscale plan.

```
SUBROUTINE NPEDLN ( A, B, C, LINEPT, LINEDR,
                    PNEAR, DIST )
C
   CALL UNORM ( LINEDR, UDIR, MAG )
C
   CALL VMINUS (UDIR, OPPDIR )
   CALL SURFPT (SCLPT, UDIR, SCLA, SCLB,
                SCLC, PT(1,1), FOUND(1))
   CALL SURFPT (SCLPT, OPPDIR, SCLA,
                SCLB, SCLC, PT(1,2), FOUND(2))
   DO 50001
     .   I = 1, 2
      IF ( FOUND(I) ) THEN
         DIST  =  0.0D0
         CALL VEQU   ( PT(1,I),  PNEAR )
         RETURN
      END IF
50001 CONTINUE
```

```
C
   NORMAL(1)  =  UDIR(1) / SCLA**2
   NORMAL(2)  =  UDIR(2) / SCLB**2
   NORMAL(3)  =  UDIR(3) / SCLC**2
   CALL NVC2PL ( NORMAL, 0.D0, CANDPL )
   CALL INEDPL ( SCLA, SCLB, SCLC, CANDPL,
                 CAND, XFOUND )
   CALL NVC2PL ( UDIR,  0.D0,   PRJPL )
   CALL PJELPL ( CAND,  PRJPL,  PRJEL )
C
   CALL VPRJP  ( SCLPT,   PRJPL,  PRJPT  )
   CALL NPELPT ( PRJPT,   PRJEL,  PRJNPT )
   DIST = VDIST ( PRJNPT, PRJPT )
C
   CALL VPRJPI ( PRJNPT, PRJPL, CANDPL, PNEAR,
                 IFOUND )
   RETURN
   END
```

Figure 6: Code with distance plan highlighted.

There is one further complication, however. It turns out that NPEDLN not only computes the nearest point from a line to an ellipsoid, it also computes the shortest distance between the line and the ellipsoid. This additional output (DIST) is convenient to construct, based on intermediate results obtained while computing the primary output (PNEAR). This is illustrated in Figure 6.[2]

Note that an alternative way to structure SPICELIB would be to have *independent* routines for computing the nearest point and the distance. The two routines would each be more coherent, but the common intermediate computations would have to be repeated, both in the code and at runtime.

The "pure" nearest point computation is shown in Figure 7. It is now much easier to see the primary computational purpose of this code.

The production version of NPEDLN contains several interleaved plans. Intermediate Fortran computations are shared by the nearest point and distance plans. A delocalized scaling plan is used to improve numerical stability, and an independent error handling plan is used to deal with unacceptable input. Knowledge of the existence of the several plans, how they are related and why they were interleaved is required for a deep understanding of NPEDLN.

1.2 Contributions

In this paper, we present a characterization of interleaving incorporating three aspects that make interleaved code difficult to understand: independence, delocalization, and resource sharing. We have distilled this characterization from an empirical examination of existing software—primarily

[2] The computation of DIST using VDIST is actually the last computation performed by the subroutine NPELPT, which NPEDLN calls; we have pulled this computation out of NPELPT for clarity of presentation.

```
SUBROUTINE NPEDLN ( A, B, C, LINEPT, LINEDR,
                    PNEAR, DIST )
C
   CALL UNORM ( LINEDR, UDIR, MAG )
C
   CALL VMINUS (UDIR, OPPDIR )
   CALL SURFPT (SCLPT, UDIR, SCLA, SCLB,
                SCLC, PT(1,1), FOUND(1))
   CALL SURFPT (SCLPT, OPPDIR, SCLA,
                SCLB, SCLC, PT(1,2), FOUND(2))
   DO 50001
     .   I = 1, 2
      IF ( FOUND(I) ) THEN
         CALL VEQU   ( PT(1,I),  PNEAR )
         RETURN
      END IF
50001 CONTINUE
```

```
C
   NORMAL(1)  =  UDIR(1) / SCLA**2
   NORMAL(2)  =  UDIR(2) / SCLB**2
   NORMAL(3)  =  UDIR(3) / SCLC**2
   CALL NVC2PL ( NORMAL, 0.D0, CANDPL )
   CALL INEDPL ( SCLA, SCLB, SCLC, CANDPL,
                 CAND, XFOUND )
   CALL NVC2PL ( UDIR,  0.D0,   PRJPL )
   CALL PJELPL ( CAND,  PRJPL,  PRJEL )
C
   CALL VPRJP  ( SCLPT,   PRJPL,  PRJPT  )
   CALL NPELPT ( PRJPT,   PRJEL,  PRJNPT )
C
   CALL VPRJPI ( PRJNPT, PRJPL, CANDPL, PNEAR,
                 IFOUND )
   RETURN
   END
```

Figure 7: The residual code without the distance plan.

SPICELIB. Secondary sources of existing software which we also examined are a Cobol database report writing system from the US Army and a program for finding the roots of functions (ZEROIN), presented and analyzed in [1] and [30]. We relate our characterization of interleaving to existing concepts in the literature, such as delocalized plans [18], coupling [41], and redistribution of intermediate results [10, 11].

We then describe the context in which we are exploring and applying these ideas. Our driving program comprehension problem is to elaborate and validate existing partial specifications of the JPL library routines to facilitate the automation of specification-driven generation of programs using these routines. We are developing analysis tools, based on the Software Refinery, to detect interleaving. We describe the analyses that we have formulated to detect classes of interleaving that are particularly useful to elaborating specifications. This paper concludes with a reflection on the feasibility of building tools to assist interleaving detection and extraction, open issues in the requirements on software and plan representations that detection imposes, the role of domain model guidance and cliché recognition in addressing the interleaving problem, and issues that arise when scaling up these analyses to the architectural level.

2 Interleaving

Programmers solve problems by breaking them into pieces. Pieces are programming language implementations of plans, and it is common for multiple plans to occur in a single code segment. We use the term "interleaving" to denote this merging.

> *Interleaving* expresses the merging of two or more distinct plans within some contiguous textual area of a program. Interleaving can be characterized by the *delocalization* of the code for the individual plans involved, the *sharing* of some resource, and the implementation of multiple, *independent* plans in the program's overall purpose.

We are characterizing the types of interleaving that typically occur in programs in order to develop techniques for detecting and extracting interleaved, but logically cohesive plans.

Interleaving may arise for several reasons. It may be intentionally introduced for efficiency. For example, it may be more efficient to compute two related values in one place than to do so separately. Intentional interleaving may also be performed to deal with non-functional requirements, such as numerical stability, that impose global constraints which are satisfied by diffuse computational structures. Interleaving may also creep into a program unintentionally, as a result of inadequate software maintenance, such as adding a feature locally to an existing routine rather than undertaking a thorough redesign. Or interleaving may arise as a natural by-product of expressing separate but related plans in a linear, textual medium. For example, accessors and constructors for manipulating data structures are typically interleaved throughout programs written in traditional programming languages due to their procedural, rather than object-oriented structure. Regardless of why interleaving is introduced, it severely complicates understanding a program. This makes it difficult to perform tasks such as extracting reusable components, localizing the effects of maintenance changes, and migrating to object-oriented languages.

There are several reasons interleaving is a source of difficulties. The first has to do with delocalization. Because two or more design purposes are implemented in a single segment of code, the individual code fragments responsible for each purpose are more spread out than they would be if they were encapsulated. Another reason interleaving presents a problem is that when it is the result of poorly thought out maintenance activities, the original, highly coherent structure of the system is typically degraded as "patches" and "quick fixes" are introduced. Finally, while interleaving may be introduced for purposes of optimization, expressing intricate optimizations in a clean and well-documented fashion is not typically done. The rationale behind the decision to intentionally introduce interleaving is often not explicitly recorded in the program. For all of these reasons, our ability to comprehend code containing interleaved fragments is compromised.

We now examine each of the characteristics of interleaving—delocalization, sharing, independence—in more detail.

2.1 Delocalization

Delocalization is one of the key characteristics of interleaving: one or more parts of a plan are spatially separated from other parts by code from other plans with which they are interleaved.

The "scale-unscale" pattern found in `NPEDLN` is a simple example of a more general delocalized plan that we refer to as a *reformulation wrapper*, which is frequently interleaved with computations in `SPICELIB`. Reformulation wrappers transform one problem into another that is simpler to solve

```
SUBROUTINE NPEDLN(A, B, C, LINEPT, LINEDR,
  .               PNEAR, DIST)
  ...
  CALL UNORM ( LINEDR, UDIR, MAG )
  ... [error checks]
  SCALE  =  MAX ( DABS(A), DABS(B), DABS(C) )
  SCLA   =  A / SCALE
  SCLB   =  B / SCALE
  SCLC   =  C / SCALE
  ... [error checks]
  SCLPT(1)  =  LINEPT(1) / SCALE
  SCLPT(2)  =  LINEPT(2) / SCALE
  SCLPT(3)  =  LINEPT(3) / SCALE
  CALL VMINUS ( UDIR, OPPDIR )
  CALL SURFPT ( SCLPT, UDIR, SCLA, SCLB,
  .             SCLC,PT(1,1), FOUND(1))
  CALL SURFPT ( SCLPT, OPPDIR,SCLA, SCLB,
  .             SCLC, PT(1,2), FOUND(2))
  ... [checking for intersection of the
       line with the ellipsoid]
      IF ( FOUND(I) ) THEN
         DIST  =  0.0D0
         CALL VSCL   (SCALE, PNEAR, PNEAR)
         ...
         RETURN
      END IF
  ... [handling the non-intercept case]
  CALL VSCL ( SCALE,  PNEAR,  PNEAR )
  DIST    =   SCALE * DIST
  ...
  RETURN
  END
```

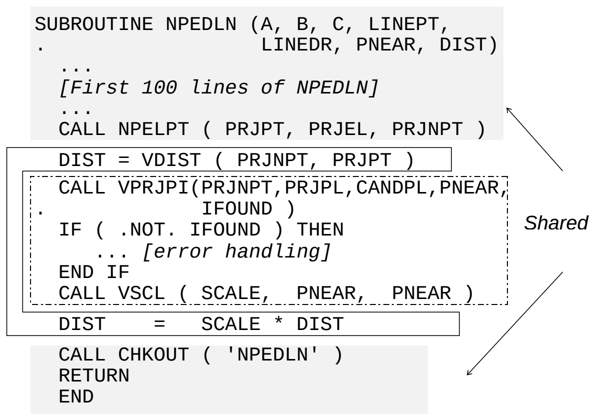
Figure 8: Portions of the NPEDLN Fortran program. Shaded regions highlight the lines of code responsible for scaling and unscaling.

and then transfer the solution back to the original situation. Other examples of reformulation wrappers in SPICELIB are reducing a three-dimensional geometry problem to a two-dimensional one and mapping an ellipsoid to the unit sphere to make it easier to solve intersection problems.

Delocalization may occur for a variety of reasons. One is that there may be an inherently non-local relationship between the components of the plan, as is the case with reformulation wrappers, which makes the spatial separation necessary. Another reason is that the intermediate results of part of a plan may be shared with another plan, causing the plans to overlap and their steps to be shuffled together; the steps of one plan separate those of the other. For example, in Figure 8, part of the scale plan (computing the scaling factor) is separated from the rest of the plan (dividing by the scaling factor) in all scalings, except the scaling of A. This allows the scaling factor to be computed once and the result reused.

Realizing that a reformulation wrapper or some other delocalized plan is interleaved with a particular computation can help prevent severe comprehension failures during maintenance [18]. It can also help detect when the delocalized plan is incomplete, as it was in an earlier version of our example subroutine whose modification history includes the following correction:

```
C- SPICELIB Version 1.2.0, 25-NOV-1992 (NJB)
C Bug fix: in the intercept case, PNEAR is now
C properly re-scaled prior to output.  Formerly,
C it was returned without having been re-scaled.
```



```
SUBROUTINE NPEDLN (A, B, C, LINEPT,
.                  LINEDR, PNEAR, DIST)
  ...
  [First 100 lines of NPEDLN]
  ...
  CALL NPELPT ( PRJPT, PRJEL, PRJNPT )
  DIST = VDIST ( PRJNPT, PRJPT )
  CALL VPRJPI(PRJNPT,PRJPL,CANDPL,PNEAR,
.              IFOUND )
  IF ( .NOT. IFOUND ) THEN
     ... [error handling]
  END IF
  CALL VSCL ( SCALE,  PNEAR,  PNEAR )
  DIST    =   SCALE * DIST
  CALL CHKOUT ( 'NPEDLN' )
  RETURN
  END
```

Figure 9: Portions of NPEDLN, highlighting two overlapping computations.

2.2 Resource Sharing

The sharing of some resource is characteristic of intentional interleaving. When interleaving is introduced into a program, there is normally some implicit relationship between the interleaved plans, motivating the designer to choose to interleave them. An example of this within NPEDLN is shown in Figure 9. The shaded portions of the code shown are *shared* between the two computations for PNEAR and DIST. In this case, the common resources shared by the interleaved plans are intermediate data results. The implementations for computing the nearest point and the shortest distance overlap in that a single structural element contributes to multiple goals.

The sharing of the results of some subcomputation in the implementation of two distinct higher level operations is termed *redistribution of intermediate results* by Hall [10, 11]. More specifically, redistribution is a class of function sharing optimizations which are implemented simply by tapping into the dataflow from some value producer and feeding it to an additional target consumer, introducing fanout into the dataflow. Redistribution covers a wide range of common types of function sharing optimizations, including common subexpression elimination and generalized loop fusion. Hall developed an automated technique for redistributing results for use in optimizing code generated from general-purpose reusable software components. We are interested in "undoing" these types of optimizations, and we can use the redistribution concept to capture forms of interleaving in which the resources shared are *data* values.

The commonality between interleaved plans might be in the form of other shared resources besides data values, such as control structures, lexical module structures, and names.

Control coupling. Control conditions may be redistributed just as data values are. The use of control flags allows control conditions to be determined once but used to affect execution at more than one location in the program. In NPEDLN, for example, SURFPT is called to compute the intersection of the line with the ellipsoid. This routine returns a control flag, FOUND, indicating whether or not the intersection exists. This flag is then used outside of SURFPT to control whether the intercept or

```
      CALL SURFPT ( SCLPT, UDIR,   SCLA, SCLB,
     .              SCLC, PT(1,1), FOUND(1) )
      CALL SURFPT ( SCLPT, OPPDIR, SCLA, SCLB,
     .              SCLC, PT(1,2), FOUND(2) )
      DO 50001
     .   I = 1, 2
         IF ( FOUND(I) ) THEN
           ... [handling the intercept case]
           RETURN
         END IF
50001 CONTINUE
C     Getting here means the line doesn't
C     intersect the ellipsoid.
      ... [handling the non-intercept case]
      RETURN
      END
```

Figure 10: Fragment of subprogram showing control coupling

non-intercept case is to be handled, as is shown in Figure 10.

The use of control flags is a special form of *control coupling*: "any connection between two modules that communicates elements of control [41]," typically in the form of function codes, flags, or switches [21]. This sharing of control information between two modules increases the complexity of the code, complicating comprehension and maintenance.

Content coupling. Another form of resource sharing occurs when the lexical structure of a module is shared among several related functional components. For example, the entire contents of a module may be lexically included in another. This sometimes occurs when a programmer wants to take advantage of a powerful intraprocedural optimizer limited to improving the code in a single routine. Another example occurs when a programmer uses `ENTRY` statements to partially overlap the contents of several routines so that they may share access to some state variables. This is sometimes done in a language, such as Fortran, that does not contain an encapsulation mechanism like packages or objects.

These two practices are examples of a phenomenon called *content coupling* in which [41] — "some or all of the contents of one module are included in the contents of another"—and which often manifests itself in the form of a multiple-entry module. Content coupling makes it difficult to independently modify or maintain the individual functions.

Name Sharing. A simple form of sharing is the use of the same variable name for two different purposes. This can lead to incorrect assumptions about the relationship between subcomputations within a program.

In general, the difficulty that resource sharing introduces is that it causes ambiguity in interpreting the purpose of program pieces. This can lead to incorrect assumptions about what effect changes will have, since the maintainer might be focusing on only one of the actual uses of the resource (variable, value, control flag, data structure slot, etc.).

2.3 Independence

While interleaving is introduced to take advantage of commonalities, it is also true that the interleaved plans each have a distinct purpose. One implication of this is that the decision to interleave the plans can, in principle, always be undone. This may require copying of common code to eliminate resource sharing, resulting in an equivalent, but possibly less efficient, program.

In the `NPEDLN` example, a separate routine could be provided responsible for computing `DIST`. This code would be nearly identical to the original, requiring added maintenance and a likely run-time cost as well. Although interleaving may be necessary for efficiency, it obscures the independence of the components involved. Ironically, this hinders activities for making the code more efficient and reusable in the long run, such as parallelization and objectivization of the code.

3 Case Study

In order to better understand interleaving, we have undertaken a case study of production library software. The library, called `SPICELIB`, consists of approximately 600 mathematical programs, written in Fortran by programmers at the Jet Propulsion Laboratory for analyzing data sent back from space missions. The software performs calculations in the domain of solar system geometry, such as coordinate frame conversions, intersections of rays, ellipses, planes, and ellipsoids, and light-time calculations. `NPEDLN` comes from this library.

We were introduced to `SPICELIB` by researchers at NASA Ames, who have developed a component-based software synthesis system, called Amphion [20, 19, 37]. Amphion automatically constructs programs that compose routines drawn from `SPICELIB`. It does this by making use of a *domain theory* that includes formal specifications of the library routines, connecting them to abstract concepts in the solar system geometry domain. The knowledge of the domain is encoded in a structured representation, expressed as axioms in first-order logic with equality. A space scientist using Amphion can schematically specify the geometry of a problem through a graphical user interface, and Amphion automatically generates Fortran programs to call `SPICELIB` routines to solve the described problem. Amphion is able to do this by proving a theorem about the solvability of the problem in the domain and, as a side effect, generating the appropriate calls. This is shown in the bottom half of Figure 11.

Amphion has been installed at JPL and used by space scientists to successfully generate over one hundred programs to solve solar system kinematics problems. The programs consist of dozens of subroutine calls and are typically synthesized in under three minutes of CPU time using a Sun Sparc 2 [20, 19]. Amphion's success depends on how accurate, consistent, and complete the domain theory is that Amphion uses. An essential program understanding task is to validate the domain theory by checking it against the `SPICELIB` routines and extending it when incompletenesses are found.

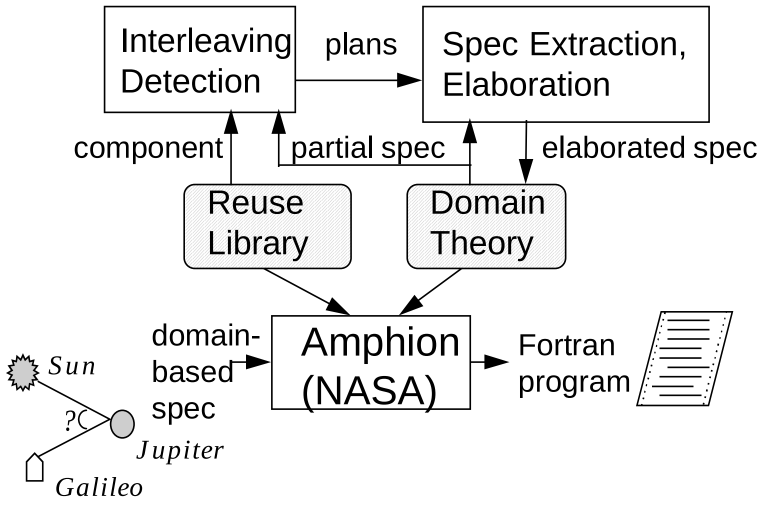


Figure 11: Applying interleaving detection to component-based reuse.

To do this, we need to be able to pull apart interleaved strands. For example, one incompleteness in Amphion's domain theory is that it does not fully cover the functionality of the routines. Some routines compute more than one result. For example, `NPEDLN` computes the nearest point on an ellipsoid to a line as well as the shortest distance between that point and the ellipsoid. However, the domain theory does not always describe all the values that are computed. In the case of `NPEDLN`, only the nearest point computation is modelled, not the shortest distance. In these routines, it is often the case that the code responsible for the secondary functionalities is interleaved with the code for the primary function covered by Amphion's domain theory. Uncovering the secondary functionality requires unraveling and understanding two interleaved computations.

Another way in which Amphion's current domain theory is incomplete is that it does not express preconditions on the use of the library routines; for example, when a line given as input to a routine is not the zero vector or that an ellipsoid's semi-axes must be large enough to be scalable. It is difficult to detect the code responsible for checking these preconditions because it is usually tightly interleaved with the code for the primary computation in order to take advantage of intermediate results computed for the primary computation.

In collaboration with NASA Ames researchers, we are detecting ways in which Amphion's domain theory is incomplete, and we are building program comprehension techniques to extend it. As the top half of Figure 11 shows, we are developing mechanisms for detecting particular classes of interleaving, with the aim of extending a partial model of the software's application domain. In the process, we are also performing analyses to gather empirical information about how much of `SPICELIB` is covered by the domain theory.

We are building interleaving detection mechanisms and empirical analysis tools using a collection of commercial tools, called the Software Refinery [14]. This is a comprehensive tool suite including language-specific analyzers and browsers for Fortran, C, Ada, and Cobol, language extension mechanisms for building new analyzers, and a user interface construction tool for displaying the results of analysis. It maintains an object-oriented repository for holding the results of analyses, such as abstract syntax trees and symbol tables. It provides a powerful wide-spectrum language, called Refine [35], which supports pattern matching and querying the repository. Using the Software Refinery allows us to leverage commercially available tools as well as evaluate the strengths and limitations of its approach to program analysis, which we discuss in Section 4.1.

Section 3.1 briefly describes our motivation for building tools to elaborate a domain model, from the perspective of both synthesis and analysis. Section 3.2 describes mechanisms for detecting and reporting precondition checks in the library routines. Section 3.3 describes heuristic mechanisms still under development for finding candidate places where interleaving is likely to occur.

3.1 Domain Model Elaboration in Synthesis and Analysis

Our motivations for validating and extending a partial domain model of existing software come both from the synthesis and from the analysis perspectives. The primary motivations for doing this from the *synthesis* perspective are to make component retrieval more accurate, to assist in updating and growing the domain model as new software components are added, and to improve the software synthesized.

From the software *analysis* perspective, the refinement and elaboration of domain knowledge, based on what is discovered in the code, is a primary activity, driving the generation of hypotheses and informing future analyses. The process of understanding software involves two parallel knowledge acquisition activities [5, 23, 36]:

1. using domain knowledge to understand the code—knowledge about the application sets up expectations about how abstract concepts are typically manifested in concrete code implementations;

2. using knowledge of the code to understand the domain—what is discovered in the code is used to build up a description of various aspects of the application and to help answer questions about why certain code structures exist and what is their purpose with respect to the application.

We are studying interleaving in the context of performing these activities, given `SPICELIB` and an incomplete model of its application domain. We are targeting our detection of interleaving toward elaborating the existing domain model. We are also looking for ways in which the current knowledge in the domain model can guide detection and ultimately comprehension.

```
C$Procedure      SURFPT ( Surface point on an ellipsoid )
      SUBROUTINE SURFPT ( POSITN, U, A, B, C, POINT, FOUND )
      DOUBLE PRECISION U      ( 3 )
      ...declarations...
C     Check the input vector to see if its the zero vector. If it is
C     signal an error and return.
C
      IF  ( ( U(1) .EQ. 0.0D0 ) .AND.
     .      ( U(2) .EQ. 0.0D0 ) .AND.
     .      ( U(3) .EQ. 0.0D0 )        )  THEN
         CALL SETMSG ( 'SURFPT: The input vector is the zero vector.' )
         CALL SIGERR ( 'SPICE(ZEROVECTOR)' )
         CALL CHKOUT ( 'SURFPT' )
         RETURN
      END IF
      ...
```

Figure 12: A fragment of the subroutine `SURFPT` in `SPICELIB`. This fragment shows a precondition check which invokes an exception if all of the elements of the `U` array are 0.

3.2 Extracting Preconditions

Using the Software Refinery, we automated a number of program analyses, one of which is the detection of subroutine parameter precondition checks. Because precondition checks are often interspersed throughout a subprogram, they tend to delocalize the plans that perform the primary computational work. They are usually part of a larger plan that detects exceptional, usually erroneous conditions in the state of a running program, and then takes alternative action when these conditions arise, such as returning with an error code, signaling, or invoking error handlers.

We found many examples of precondition checks in our empirical analysis of the `SPICELIB`. One such check occurs in the subprogram `SURFPT` and is shown in Figure 12. `SURFPT` finds the intersection (`POINT`) of a ray (represented by a point `POSITN` and a direction vector `U`) with an ellipsoid (represented as three semi-axes lengths `A`, `B`, and `C`), if such an intersection exists (indicated by `FOUND`). One of the preconditions checked by `SURFPT` is that the direction vector `U` is not the zero-vector.

Precondition checks make explicit the assumptions a subprogram places on its inputs. The process of understanding a subprogram can be facilitated by removing its precondition checks and using the information they encode to elaborate a high-level specification of the subprogram. We have created a tool that detects precondition checks and extracts the preconditions into a documentation form suitable for expression as a partial specification. The specifications can then be compared against the Amphion domain model.

Precondition checks are particularly difficult to understand when they are sprinkled throughout the code of a subroutine as opposed to being localized at the beginning. We discovered that, though interleaved, these checks could be reliably identified by searching for `IF` statements whose conditions are a function of the input parameters and whose bodies invoke exception handlers. The analysis that decides whether or not `IF` statements test only input parameters is specific to the Fortran language; whereas the analysis that decides if a code fragment is an exception plan is application domain specific. The implication is that the Fortran specific portion is not likely to need changing when we apply the tool to a new Fortran application; whereas the application specific portion will certainly need to change. With this in mind, we chose a tool architecture that allows flexibility in keeping these different types of pattern knowledge separate and independently adaptible.

Detecting Exception Handlers In general, we need application specific knowledge about usage patterns in order to discover exception handlers. `SPICELIB`, for example, provides a routine `SIGERR` that sets an error condition. Typically, `SIGERR` is followed almost immediately by a `RETURN` statement. Hence, a call to `SIGERR` followed closely by a `RETURN` indicates a cliché for handling an exception. In some other application, the form of this cliché will be much different. It is, therefore, necessary to design the recognition component of our architecture around this need to specialize the tool with knowledge about the application of a system being analyzed.

The Software Refinery provides excellent support for this design principle through the use of the **rule** construct and a tree-walker that applies these rules to an abstract syntax tree (AST). Rules declaratively specify state changes by listing the *conditions* before and after the change without specifying exactly how the change occurs. This is useful for linking application specific pattern knowledge into a system because it allows the independent, declarative expression of the different facets of the pattern.

We recognize application-specific exception handlers using two rules which search the AST for a call to `SIGERR`, followed by a `RETURN` statement. These rules and the Refine code that applies them are presented in detail in [31].

Detecting Guards Discovering "guards," which are `IF` statements that depend only upon input parameters, involves keeping track of whether or not these parameters have been modified before the check. If they have been modified before the check, then the check probably is not a precondition check on inputs. In Fortran, a variable X can be modified by:

1. appearing on the left hand side of an assignment statement,
2. being passed into a subprogram which then modifies the formal parameter bound to X by the call,
3. being implicitly passed into another subprogram in a `COMMON` block and modified in this other subprogram, or
4. explicitly aliased by an `EQUIVALENCE` statement to another variable which is then modified.

Currently our analysis does not detect modification through `COMMON` or `EQUIVALENCE` because none of the code in `SPICELIB` uses these features with formal parameters. We track modifications to input parameters by using an approximate dataflow algorithm that propagates a set of unmodified

RECGEO $\neg(F \geq 1) \wedge \neg(RE \leq 0.0D0)$

REMSUB $\neg((LEFT > RIGHT) \vee (RIGHT < 1) \vee (LEFT < 1) \vee (RIGHT > LEN(IN)) \vee (LEFT > LEN(IN)))$

SURFPT $\neg((U(1) = 0.0D0) \wedge (U(2) = 0.0D0) \wedge (U(3) = 0.0D0))$

XPOSBL $\neg((MOD(NCOL,BSIZE) \neq 0) \vee (MOD(NROW,BSIZE) \neq 0)) \wedge \neg(NCOL < 1) \wedge \neg(NROW < 1) \wedge \neg(BSIZE < 1)$

Figure 13: Preconditions extracted for some of the subroutines in `SPICELIB`.

variables through the sequence of statements in the subroutine. At each statement, if a variable X in the set could be modified by the execution of the statement, then X is removed from the set. After the propagation, we can easily check whether or not an `IF` statement is a guard.

Results The result of this analysis is a table of preconditions associated with each subroutine. Since we are targetting partial specification elaboration, we chose to make the tool output the preconditions in LaTeX form so as to generate nicely formatted reports. Figure 13 gives examples of preconditions extracted for a few `SPICELIB` subroutines. Our tool generated the LaTeX source included in Figure 13 without change.

Taken literally, the precondition for `SURFPT`, for example, states that one of the first three elements of the `U` array parameter must be non-zero. In the domain model, `U` is seen as a vector, so the more abstract precondition can be stated as "U is not the zero vector." Extracting the precondition into the literal representation is the first step to being able to express the precondition in the more abstract form.

The other preconditions listed in Figure 13, stated in their abstract form, are the following. The subroutine `RECGEO` converts the rectangular coordinates of a point `RECTAN` to geodetic coordinates, with respect to a given reference spheroid whose equatorial radius is `RE`, using a flattening coefficient `F`. Its precondition is that the radius is greater than 0 and the flattening coefficient is less than 1. The subroutine `REMSUB` removes the substring `(LEFT:RIGHT)` from a character string `IN`. It requires that the positions of the first character `LEFT` and the last character `RIGHT` to be removed are in the range 1 to the length of the string and that the position of the first character is less than the position of the last. Finally, the subroutine `XPOSBL` transposes the square blocks within a matrix `BMAT`. Its preconditions are that the block size `BSIZE` must evenly divide both the number of rows `NROW` in `BMAT` and the number of columns `NCOL` and that the block size, number of rows, and number of columns are all at least 1.

3.3 Finding Interleaving Candidates

There are other analyses that we are implementing which are heuristic techniques for finding likely interleaving candidates.

3.3.1 Routines with Multiple Outputs

One heuristic for finding instances of interleaving is to determine which subroutines compute more than one output. When this occurs, the subroutine is returning either the results of multiple distinct computations or a result whose type can not be directly expressed in the Fortran type system (e.g., as a data aggregate). In the former case, the subroutine is realized as the interleaving of multiple distinct plans, as is the case with `NPEDLN`'s computation of both the nearest point and the shortest distance.

In the latter case, the subroutine may be implementing only a single plan, but a maintainer's conceptual categorization of the subroutine is still obscured by the appearance of some number of seemingly distinct outputs. A good example of this case occurs in the `SPICELIB` subroutine `SURFPT`, which conceptually returns the intersection of a vector with the surface of an ellipsoid. However, it is possible to give `SURFPT` a vector and an ellipsoid that do not intersect. In such a situation the output parameter `POINT` will be undefined, but the Fortran type system cannot express the type: `DOUBLE PRECISION` $\vee$ *Undefined*. The programmer was forced to simulate a variable of this type using two variables, `POINT` and `FOUND`, adopting the convention that when `FOUND` is **false**, the return value is *Undefined*, and when `FOUND` is **true**, the return value is `POINT`.

Clearly subprograms with multiple outputs complicate program understanding. We built a tool that determines the multiple output subprograms in a library by analyzing the direction of dataflow in parameters of functions and subroutines. A parameter's direction is either: **in** if the parameter is only read in the subprogram, **out** if the parameter is only written in the subprogram, or **in-out** if the parameter is both read and written in the subprogram. Multiple output subprograms will have more than one parameter with direction out or in-out.

Our tool bases its analysis on the structure chart (call graph) objects that the Software Refinery creates. The nodes of these structure charts are annotated with direction information about parameters. The resulting analysis showed that 25 percent of the subprograms in `SPICELIB` had multiple output parameters. We were thus able to focus our work on these routines first, as they are likely to involve interleaving.

In addition, we performed an empirical analysis to determine, for those routines covered by the Amphion domain model (35 percent of the library), which ones have multiple output parameters, some of which are not covered by the domain model. We refer to outputs that are not mapped to anything in the domain model as *dead end dataflows*, since the programs that Amphion creates can never make use of these return values; they have not been associated with any meaning in the application domain. For example, `NPEDLN`'s distance output (`DIST`) is a dead end dataflow as far as the domain model is concerned. Dead end dataflows imply interleaving in the subprogram and/or an incompleteness in the domain model. Our analysis revealed that of the subroutines covered by the domain model, 30 percent have some output parameters that are dead end dataflows. These are good focal points for detecting interleaved plans that might be relevant to elaborating the domain theory.

3.3.2 Control Coupling

Another heuristic for detecting potential interleaving finds candidate routines that may be involved in *control coupling*. In particular, it asks the question: *Which calls to library routines, supply a constant as a parameter to other routines as opposed to a variable?* The constant parameter may be a flag that is being used to choose among a set of possible computations to perform. A strategy we use for detecting control coupling first computes a set of candidate routines that are invoked with a constant parameter in the library or by code generated from the Amphion domain model. Each member of this set is then analyzed to see if the formal parameter associated with the constant actual parameter is used to conditionally execute disjoint sections of code. We have discovered instances of control coupling in `SPICELIB` and have computed the set of routines that are invoked with a constant parameter. This set accounts for 19 percent of the total routines in `SPICELIB`, and we are currently investigating these to see how many of them actually exhibit control coupling.

3.3.3 Reformulation Wrappers

A third heuristic for locating interleaving is to ask: *Which pairs of routines "co-occur"?* Two routines co-occur if they are always called by the same routines, they are executed under the same conditions, and there is a flow of computed data from one to the other. We would like to detect co-occurrence pairs because they are likely to form *reformulation wrappers*. Detecting co-occurrence pairs in the library can heuristically help generate candidates for reformulation wrappers. The pairs need to be checked further to see whether they are inverses of each other. Of course, in general we would like to consider any code fragments as potential pairs, not just library routines.

Through empirical investigation of `SPICELIB`, we have discovered co-occurrence pairs that form reformulation wrappers and are currently building tools to perform this analysis automatically. The idea is to generate a set of all possible pairs of routines in the system and then eliminate pairs that do not co-occur. Given a pair $\langle S_1, S_2 \rangle$ we say that S_1 and S_2 co-occur if and only if: 1) every subroutine in the library that references S_1 also references S_2, 2) execution of S_1 implies execution of S_2 and vice versa, and 3) there is a flow of computed data from S_1 to S_2. We describe how to implement this test in [31].

4 Open Issues

We are convinced that interleaving seriously complicates understanding computer programs. But recognizing a problem is different from knowing how to fix it. Questions arise as to how powerful tools need to be to detect and extract interleaved components, what form of representation is appropriate to hold the extracted information, how information about the application domain can be used to detect plans, and the extent to which the concept of interleaving and its detection mechanisms scale up to the architectural level.

4.1 Tool Support

We used the Software Refinery from Reasoning Systems in our analyses. This comprehensive toolkit provides a set of language-specific browsers and analyzers, a parser generator, a user interface builder, and an object-oriented repository for holding the results of analyses. We made particular use of two other features of the toolkit. The first is called the Workbench, and it provided pre-existing analyses for traditional graphs and reports such as structure charts, dataflow diagrams, and cross reference lists. The results of the analyses can be accessed from the repository using small, Refine language programs such as those described in [31]. The Refine compiler was the other feature we used, compiling a Refine program into compiled Lisp.

The approach taken by the Refine language and tool suite has many advantages for attacking problems like ours. The language itself combines features of imperative, object-oriented, functional, and rule-based programming, thus providing flexibility and generality. Of particular value to us is its rule-based constructs. Before-and-after condition patterns define the properties of constructs without indicating how to find them. We had merely to add a simple tree walking routine to apply the rules to the abstract syntax tree. In addition to the rule-based features, Refine provides abstract data structures, such as sets, maps, and sequences, which manage their own memory requirements, thereby reducing programmer work. The object-oriented repository further reduces programmer responsibility by providing persistence and memory management.

We also take full advantage of Reasoning Systems' existing Fortran language model and its structure chart analysis. These allowed us a running start on our analysis and provided a robust handling of Fortran constructs that are not typically available from non-commercial research tools.

We can see several ways in which the Refine approach can be extended. In particular, the availability of other analyses, such as control flow graphs for Fortran and general dataflow analysis, would prove useful. Robust dataflow analysis is particularly important to the precision of precondition extraction. The Refine language itself might also be extended further. Currently it does not support passing functions as parameters, and the application of the rule construct is somewhat restricted.

4.2 Representation

The strategy for building program analysis tools is to formulate a *program representation* whose structural properties correspond to interesting program properties. A programming style tool, for example, uses a control flow graph that explicitly represents transfer of execution flow in programs. Irreducible control flow graphs signify the use of unstructured `GO TO` statements. The style tool uses this structural property to report violations of structured programming style. Since we want to build tools for interleaving detection we have to formulate a representation that captures the properties of interleaving. We do this by first listing structural properties that correspond to each of the three characteristics of interleaving and then searching for a representation that has these structural properties.

The key characteristics of interleaving are delocalization, resource sharing, and independence.

In sequential languages like Fortran, delocalization often can not be avoided when two or more plans share data. The components of the plans have to be serialized with respect to the dataflow constraints. This typically means that components of plans cluster around the computation of the data being shared as opposed to clustering around other components of the same plan. This total ordering is necessary due to the lack of support for concurrency in most high level programming languages. It follows then that in order to express a delocalized plan, a representation must impose a partial rather than a total execution ordering on the components of plans.

The partial execution ordering requirement suggests that some form of graphical representation is appropriate. Graph representations naturally express a partial execution ordering via implicit concurrency and explicit transfer of control and data. Since there are a number of such representations to choose from, we narrow the possibilities by noting that:

1. independent plans must be localized as much as possible, with no explicit ordering between them;
2. sharing must be detectable (shared resources should explicitly flow from one plan to another); similarly if two plans p_1, p_2 both share a resource provided by a plan p_3 then p_1 and p_2 should appear in the graph as siblings with a common ancestor p_3);
3. the representation must support multiple views of the program as the interaction of plans at various levels of abstraction, since interleaving may occur at any level of abstraction.

An existing formalism that meets these criteria is Rich's *Plan Calculus* [26, 27, 28]. A plan in the Plan Calculus is encoded as a graphical depiction of the plan's structural parts and the constraints (e.g., data and control flow connections) between them. This diagrammatic notation is complemented with an axiomatized description of the plan that defines its formal semantics. This allows us to develop correctness preserving transformations to extract interleaved plans. The Plan Calculus also provides a mechanism, called *overlays*, for representing correspondences and relationships between pairs of plans (e.g., implementation and optimization relationships). This enables the viewing of plans at multiple levels of abstraction. Overlays also support a very general notion of plan composition which takes into account resource sharing at all levels of abstraction by allowing overlapping points of view.

4.3 Domain Knowledge

Most of the current technology available to help understand programs addresses *how* questions; that is, it is driven by the syntactic structure of programs written in some programming language. But the tasks that require the understanding—perfective, adaptive, and corrective maintenance—are driven by the problem the program is solving; that is, its application domain. These tasks really require answers to *why* questions. For example, if a maintenance task requires extending `NPEDLN` to handle symmetric situations where more than one "nearest point" exist to a line, then the programmer needs to figure out what to do about the distance calculation also computed by `NPEDLN`. Why was `DIST` computed inside of the routine instead of separately? Was it only for efficiency reasons, or might the nearest point and the distance be considered a *pair* of results by its callers? In the former case, a single DIST return value is still appropriate, in the latter, a pair of identical values is indicated. To answer questions like these, programmers need to know which plans pieces of code are implementing. And plans are direct reflections of the application domain, not the program.

Another example from NPEDLN concerns reformulation wrappers. These plans are inherently delocalized. In fact, they only make sense as plans at all when considered in the context of the application domain: stable computations of solar system geometry. Without this understanding, the best hope is to recognize that the code has uniformly multiplied the elements of a vector in two places, without knowing why this was done and how the multiplications are connected.

The underlying issue is that any scheme for code understanding based solely on a top-down or a bottom-up approach is inherently limited. As illustrated by the examples, a bottom-up approach cannot hope to relate delocalized segments or disentangle interleavings without being able to relate to the plans being implemented. And a top-down approach cannot hope to find where a plan is implemented without being able understand how plan implementations are related syntactically and via data flows. The implication is that a coordinated strategy is indicated, where plans generate expectations that guide program analysis and program analysis generates related segments that need explanation.

4.4 Architectural Interleaving

We can characterize the ways interleaving manifests itself in source code along two orthogonal dimensions. These form a possible design space of solutions to the interleaving problem and can help relate existing techniques that might be applicable. One dimension is the *scope* of the interleaving, which can range from intraprocedural to interprocedural to object to architectural. Another dimension is the *structural mechanism* providing the interleaving, which may be naming, control, data, or protocol. Protocols are global constraints, such as maintaining stack discipline or synchronization mechanisms for cooperating processes. For example, the use of control flags is a control-based mechanism for interleaving with interprocedural scope. The common iteration construct involved in loop fusion is another control-based mechanism, but this interleaving has intraprocedural scope. Reformulation wrappers use a protocol mechanism, usually at the intraprocedural level, but they can have interprocedural scope. Multiple-inheritance is an example of a data-centered interleaving mechanism with object scope.

Interleaving at the scope of objects and architectures or involving global protocol mechanisms is not yet well understood. Consequently, few mechanisms for detection and extraction currently exist in these areas. We believe that more knowledge of the domain will be required to detect interleaving as the scope becomes more global in nature.

5 Related Work

Techniques for detecting interleaving and disentangling interleaved plans are likely to build on existing program comprehension and maintenance techniques.

5.1 The Role of Recognition

When what is interleaved is *familiar* (i.e., stereotypical, frequently used plans), cliché recognition (e.g., [12, 15, 16, 17, 25, 29, 40]) is a useful detection mechanism.[3] In fact, most recognition systems deal explicitly with the recognition of clichés that are interleaved in specific ways with unrecognizable code or other clichés. One of the key features of `GRASPR` [40], for instance, is its ability to deal with delocalization and redistribution-type function sharing optimizations.

`KBEmacs` [28, 38] uses a simple, special-purpose recognition strategy to segment loops within programs. This is based on detecting coarse patterns of data and control flow at the procedural level that are indicative of common ways of constructing, augmenting, and interleaving iterative computations. For example, `KBEmacs` looks for minimal sections of a loop body that have data flow feeding back only to themselves. This decomposition enables a powerful form of abstraction, called *temporal abstraction*, which views iterative computations as compositions of operations on sequences of values. The recognition and temporal abstraction of iteration clichés is similarly used in `GRASPR` to enable it to deal with generalized loop fusion forms of interleaving. Loop fusion is viewed as redistribution of sequences of values and treated as any other redistribution optimization [40].

Most existing cliché recognition systems tend to deal with interleaving involving *data* and *control* mechanisms. Domain-based clustering, as explored by `DM-TAO` in the `DESIRE` system [2], focuses on *naming* mechanisms, by keying in on the patterns of linguistic idioms used in the program, which suggest the manifestations of domain concepts.

Mechanisms for dealing with specific types of interleaving have been explicitly built into existing recognition systems. In the future, we envision recognition architectures that detect not only familiar computational patterns, but also recognize familiar types of transformations or design decisions that went into constructing the program. Many existing cliché recognition systems implicitly detect and undo certain types of interleaving design decisions. However, this process is usually done with special-purpose procedural mechanisms that are difficult to extend and that are viewed as having secondary "supporting roles" to the cliché recognition process, rather than as being an orthogonal form of recognition.

5.2 Disentangling Unfamiliar Plans

When what is interleaved is *unfamiliar* (i.e., novel, idiosyncratic, not repeatedly used plans), other, non-recognition-based methods of delineation are needed. For example, slicing [39, 22] is a widely-

[3] Recognition as a program understanding technique deals with clichés, not plans in general. Only clichéd plans can be recognized, since recognition implies noticing something that is familiar.

used technique for localizing functional components by tracing through data dependencies within the procedural scope. Cluster analysis [2, 13, 32, 33] is used to group related sections of code, based on the detection of shared uses of global data, control paths, and names. However, clustering techniques can only provide limited assistance by roughly delineating possible locations of functionally cohesive components. Another technique, called "potpourri module detection" [6], detects modules that provide more than one independent service by looking for multiple proper subgraphs in an entity-to-entity interconnection graph. These graphs show dependencies among global entities within a single module. Presumably, the independent services reflect separate plans in the code.

Research into automating data encapsulation has recently provided mechanisms for hypothesizing possible locations of data plans at the *object* scope. For example, Bowdidge and Griswold [4] use an extended data flow graph representation, called a star diagram, to help human users see all the uses of a particular data structure and to detect frequently occurring computations that are candidates for abstract functions. Techniques have also been developed within the RE^2 project [7, 8], for identifying candidate abstract data types and their associated modules, based on the call graph and dominance relations. Further research is required to develop techniques for extracting objects from pieces of data that have not already been aggregated in programmer-defined data structures. For example, detecting multiple pieces of data that are always used together might suggest candidates for data aggregation (as for example, in `NPEDLN`, where the input parameters `A`, `B`, and `C` are used as a tuple representing an ellipsoid, and the outputs `PNEAR` and `DIST` represent a pair of results related by interleaved, highly overlapping plans).

6 Conclusion

This paper characterizes interleaving, a particularly troublesome feature of programs which makes them difficult to understand. We offer the following definition:

> *Interleaving* expresses the merging of two or more distinct plans within some contiguous textual area of a program. Interleaving can be characterized by the *delocalization* of the code for the individual plans involved, the *sharing* of some resource, and the implementation of multiple, *independent* plans in the program's overall purpose.

Whether this characterization is robust is an issue for future empirical studies. We plan to study the frequency of occurrence of the various types of interleaving we have identified in our example programs. This may lead to better complexity metrics for determining the maintainability and comprehensibility of programs. Ultimately, designing tools for detection and extraction will be the true test of the usefulness of this characterization.

Acknowledgments Support for this research has been provided by ARPA, (contract number NAG 2-890). We are grateful to `JPL`'s `NAIF` group for enabling our study of their `SPICELIB` software. We also benefited from insightful discussions with Michael Lowry at Nasa Ames Research Center concerning this study and interesting future directions.

7 Appendix: NPELDN with Some of Its Documentation

```
C$ Nearest point on ellipsoid to line.
SUBROUTINE NPEDLN(A,B,C,LINEPT,LINEDR,PNEAR,
   .              DIST)
  INTEGER               UBEL
  PARAMETER           ( UBEL = 9 )
  INTEGER               UBPL
  PARAMETER           ( UBPL = 4 )
  DOUBLE PRECISION      A
  DOUBLE PRECISION      B
  DOUBLE PRECISION      C
  DOUBLE PRECISION      LINEPT ( 3 )
  DOUBLE PRECISION      LINEDR ( 3 )
  DOUBLE PRECISION      PNEAR  ( 3 )
  DOUBLE PRECISION      DIST
  LOGICAL               RETURN
  DOUBLE PRECISION      CANDPL ( UBPL )
  DOUBLE PRECISION      CAND   ( UBEL )
  DOUBLE PRECISION      OPPDIR (    3 )
  DOUBLE PRECISION      PRJPL  ( UBPL )
  DOUBLE PRECISION      MAG
  DOUBLE PRECISION      NORMAL (3 )
  DOUBLE PRECISION      PRJEL  ( UBEL )
  DOUBLE PRECISION      PRJPT  (3 )
  DOUBLE PRECISION      PRJNPT (    3 )
  DOUBLE PRECISION      PT     ( 3, 2 )
  DOUBLE PRECISION      SCALE
  DOUBLE PRECISION      SCLA
  DOUBLE PRECISION      SCLB
  DOUBLE PRECISION      SCLC
  DOUBLE PRECISION      SCLPT  (    3 )
  DOUBLE PRECISION      UDIR   (    3 )
  INTEGER               I
  LOGICAL               FOUND  (    2 )
  LOGICAL               IFOUND
  LOGICAL               XFOUND
  IF ( RETURN () ) THEN
     RETURN
  ELSE
     CALL CHKIN ( 'NPEDLN' )
  END IF
  CALL UNORM ( LINEDR, UDIR, MAG )
  IF ( MAG .EQ. 0 ) THEN
     CALL SETMSG('Direction is zero vector.')
     CALL SIGERR('SPICE(ZEROVECTOR)'        )
     CALL CHKOUT('NPEDLN'                   )
     RETURN
   ELSE IF (( A .LE. 0.D0 )
 .          .OR.   ( B .LE. 0.D0 )
 .          .OR.   ( C .LE. 0.D0 )   )   THEN
     CALL SETMSG ('Semi-axes: A=#,B=#,C=#.')
     CALL ERRDP  ('#', A                   )
     CALL ERRDP  ('#', B                   )
     CALL ERRDP  ('#', C                   )
     CALL SIGERR ('SPICE(INVALIDAXISLENGTH)')
     CALL CHKOUT ('NPEDLN'                 )
     RETURN
   END IF
C Scale the semi-axes lengths for better
C numerical behavior.  If squaring any of the
C scaled lengths causes it to underflow to
C zero, signal an error. Otherwise scale the
C point on the input line too.
  SCALE  =  MAX ( DABS(A), DABS(B), DABS(C) )
  SCLA   =  A / SCALE
  SCLB   =  B / SCALE
  SCLC   =  C / SCALE
  IF (( SCLA**2  .LE.  0.D0 )
 .     .OR.  ( SCLB**2  .LE.  0.D0 )
 .     .OR.  ( SCLC**2  .LE.  0.D0 )   ) THEN
   CALL SETMSG ('Axis too small: A=#,B=#,C=#.')
     CALL ERRDP  ('#', A )
     CALL ERRDP  ('#', B )
     CALL ERRDP  ('#', C )
     CALL SIGERR ('SPICE(DEGENERATECASE)')
     CALL CHKOUT ('NPEDLN' )
     RETURN
  END IF
  SCLPT(1)  =  LINEPT(1) / SCALE
  SCLPT(2)  =  LINEPT(2) / SCALE
  SCLPT(3)  =  LINEPT(3) / SCALE
C Hand off the intersection case to SURFPT.
C SURFPT determines whether rays intersect a body,
C so we treat the line as a pair of rays.
  CALL VMINUS(UDIR, OPPDIR)
  CALL SURFPT(SCLPT, UDIR,   SCLA, SCLB,
 .            SCLC, PT(1,1), FOUND(1))
  CALL SURFPT(SCLPT, OPPDIR, SCLA, SCLB,
 .            SCLC, PT(1,2), FOUND(2))
  DO 50001
 .   I = 1, 2
     IF ( FOUND(I) ) THEN
        DIST  =  0.0D0
        CALL VEQU   ( PT(1,I),  PNEAR         )
        CALL VSCL   ( SCALE,    PNEAR,  PNEAR )
        CALL CHKOUT ( 'NPEDLN'                )
        RETURN
     END IF
50001 CONTINUE
C Getting here means the line doesn't intersect
C the ellipsoid. Find the candidate ellipse CAND.
C NORMAL is a normal vector to the plane
C containing the candidate ellipse. Mathematically
C the ellipse must exist; it's the intersection of
C an ellipsoid centered at the origin and a plane
C containing the origin.  Only numerical problems
C can prevent the intersection from being found.
  NORMAL(1)  =  UDIR(1) / SCLA**2
  NORMAL(2)  =  UDIR(2) / SCLB**2
  NORMAL(3)  =  UDIR(3) / SCLC**2
  CALL NVC2PL ( NORMAL, 0.D0, CANDPL )
  CALL INEDPL (SCLA,SCLB,SCLC,CANDPL,CAND,XFOUND)
  IF ( .NOT. XFOUND ) THEN
     CALL SETMSG ( 'Candidate ellipse not found.')
     CALL SIGERR ( 'SPICE(DEGENERATECASE)'       )
     CALL CHKOUT ( 'NPEDLN'                      )
     RETURN
  END IF
C Project the candidate ellipse onto a plane
C orthogonal to the line.  We'll call the plane
C PRJPL and the projected ellipse PRJEL.
  CALL NVC2PL ( UDIR,  0.D0,   PRJPL )
  CALL PJELPL ( CAND,  PRJPL,  PRJEL )
C Find the point on the line lying in the project-
C ion plane, and then find the near point PRJNPT
C on the projected ellipse.  Here PRJPT is the
C point on the line lying in the projection plane.
C The distance between PRJPT and PRJNPT is DIST.
  CALL VPRJP  ( SCLPT,   PRJPL,  PRJPT         )
  CALL NPELPT ( PRJPT,   PRJEL,  PRJNPT )
  DIST = VDIST ( PRJNPT, PRJPT )
C Find the near point PNEAR on the ellipsoid by
C taking the inverse orthogonal projection of
C PRJNPT; this is the point on the candidate
C ellipse that projects to PRJNPT.  The output
C DIST was computed in step 3 and needs only to be
C re-scaled. The inverse projection of PNEAR ought
C to exist, but may not be calculable due to nu-
C merical problems (this can only happen when the
C ellipsoid is extremely flat or needle-shaped).
  CALL VPRJPI(PRJNPT,PRJPL, CANDPL, PNEAR, IFOUND)
  IF ( .NOT. IFOUND ) THEN
     CALL SETMSG ('Inverse projection not found.')
     CALL SIGERR ('SPICE(DEGENERATECASE)'        )
     CALL CHKOUT ('NPEDLN'                       )
     RETURN
  END IF
C Undo the scaling.
  CALL VSCL ( SCALE,  PNEAR,  PNEAR )
  DIST    =   SCALE * DIST
  CALL CHKOUT ( 'NPEDLN' )
  RETURN
  END
```

```
C Descriptions of subroutines called by NPEDLN:
C
C CHKIN  Module Check In (error handling).
C UNORM  Normalize double precision 3-vector.
C SETMSG Set Long Error Message.
C SIGERR Signal Error Condition.
C CHKOUT Module Check Out (error handling).
C ERRDP  Insert DP Number into Error Message Text.
C VMINUS Negate a double precision 3-D vector.
C SURFPT Find intersection of vector w/ ellipsoid.
C VEQU   Make one DP 3-D vector equal to another.
C VSCL   Vector scaling, 3 dimensions.
C NVC2PL Make plane from normal and constant.
C INEDPL Intersection of ellipsoid and plane.
C PJELPL Project ellipse onto plane, orthogonally.
C VPRJP  Project a vector onto plane orthogonally.
C NPELPT Find nearest point on ellipse to point.
C VPRJPI Vector projection onto plane, inverted.
C ____________________________________________
C
C Descriptions of variables used by NPEDLN:
C A       Length of semi-axis in the x direction.
C B       Length of semi-axis in the y direction.
C C       Length of semi-axis in the z direction.
C LINEPT  Point on input line.
C LINEDR  Direction vector of input line.
C PNEAR   Nearest point on ellipsoid to line.
C DIST    Distance of ellipsoid from line.
C UBEL    Upper bound of array containing ellipse.
C UBPL    Upper bound of array containing plane.
C PT      Intersection point of line & ellipsoid.
C CAND    Candidate ellipse.
C CANDPL  Plane containing candidate ellipse.
C NORMAL  Normal to the candidate plane CANDPL.
C UDIR    Unitized line direction vector.
C MAG     Magnitude of line direction vector.
C OPPDIR  Vector in direction opposite to UDIR.
C PRJPL   Projection plane, which the candidate
C         ellipse is projected onto to yield PRJEL.
C PRJEL   Projection of the candidate ellipse
C         CAND onto the projection plane PRJEL.
C PRJPT   Projection of line point.
C PRJNPT  Nearest point on projected ellipse to
C         projection of line point.
C SCALE   Scaling factor.
```

References

[1] V.R. Basili and H.D. Mills. Understanding and documenting programs. *IEEE Trans. on Software Engineering*, 8(3):270–283, May 1982.

[2] T. Biggerstaff, B. Mitbander, and D. Webster. Program understanding and the concept assignment problem. *Comm. of the ACM*, 37(5):72–83, May 1994.

[3] Barry Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[4] R. Bowdidge and W. Griswold. Automated support for encapsulating abstract data types. In *Proc. 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering*, pages 97–110, New Orleans, Dec. 1994.

[5] R. Brooks. Towards a theory of the comprehension of computer programs. *Int. Journal of Man-Machine Studies*, 18:543–554, 1983.

[6] F. Calliss and B. Cornelius. Potpourri module detection. In *IEEE Conf. on Software Maintenance – 1990*, pages 46–51, San Diego, CA, November 1990. IEEE Computer Society Press.

[7] G. Canfora, A. Cimitile, and M. Munro. A reverse engineering method for identifying reusable abstract data types. In *Proc. of the First Working Conference on Reverse Engineering*, pages 73–82, Baltimore, Maryland, May 1993. IEEE Computer Society Press.

[8] A. Cimitile, M. Tortorella, and M. Munro. Program comprehension through the identification of abstract data types. In *Proc. 3rd Workshop on Program Comprehension*, pages 12–19, Washington, D.C., November 1994. IEEE Computer Society Press.

[9] R. K. Fjeldstad and W. T. Hamlen. Application program maintenance study: Report to our respondents. In *GUIDE 48*, 4 1979. Also appears in [24].

[10] R. Hall. Program improvement by automatic redistribution of intermediate results. Technical Report 1251, MIT Artificial Intelligence Lab., February 1990. PhD.

[11] R. Hall. Program improvement by automatic redistribution of intermediate results: An overview. In M. Lowry and R. McCartney, editors, *Automating Software Design*. AAAI Press, Menlo Park, CA, 1991.

[12] J. Hartman. Automatic control understanding for natural programs. Technical Report AI 91-161, University of Texas at Austin, 1991. PhD thesis.

[13] D. Hutchens and V. Basili. System structure analysis: Clustering with data bindings. *IEEE Trans. on Software Engineering*, 11(8), August 1985.

[14] Reasoning Systems Incorporated. *Software Refinery Toolkit*. Palo Alto, CA.

[15] W. L. Johnson. *Intention-Based Diagnosis of Novice Programming Errors*. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

[16] W. Kozaczynski and J.Q. Ning. Automated program understanding by concept recognition. *Automated Software Engineering*, 1(1):61–78, March 1994.

[17] S. Letovsky. Plan analysis of programs. Research Report 662, Yale University, December 1988. PhD.

[18] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3), 1986.

[19] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. Amphion: automatic programming for subroutine libraries. In *Proc. 9th Knowledge-Based Software Engineering Conference*, pages 2–11, Monterey, CA, 1994.

[20] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. A formal approach to domain-oriented software design environments. In *Proc. 9th Knowledge-Based Software Engineering Conference*, pages 48–57, Monterey, CA, 1994.

[21] G. Myers. *Reliable Software through Composite Design.* Petrocelli Charter, 1975.

[22] J.Q. Ning, A. Engberts, and W. Kozaczynski. Automated support for legacy code understanding. *Comm. of the ACM*, 37(5):50–57, May 1994.

[23] S. Ornburn and S. Rugaber. Reverse engineering: Resolving conflicts between expected and actual software designs. In *IEEE Conf. on Software Maintenance – 1992*, pages 32–40, Orlando, Florida, November 1992.

[24] G. Parikh and N. Zvegintozov, editors. *Tutorial on Software Maintenance.* IEEE Computer Society, 1983. Order No. EM453.

[25] A. Quilici. A memory-based approach to recognizing programming plans. *Comm. of the ACM*, 37(5):84–93, May 1994.

[26] C. Rich. A formal representation for plans in the Programmer's Apprentice. In *Proc. 7th Int. Joint Conf. Artificial Intelligence*, pages 1044–1052, Vancouver, British Columbia, Canada, August 1981.

[27] C. Rich. Inspection methods in programming. Technical Report 604, MIT Artificial Intelligence Lab., June 1981. PhD thesis.

[28] C. Rich and R. C. Waters. *The Programmer's Apprentice.* Addison-Wesley, Reading, MA and ACM Press, Baltimore, MD, 1990.

[29] C. Rich and L. M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 7(1):82–89, January 1990. Reprinted in P. H. Winston, editor, *Artificial Intelligence at MIT: Expanding Frontiers*, MIT Press, Cambridge, MA, In press.

[30] S. Rugaber, S. Ornburn, and R. LeBlanc. Recognizing design decisions in programs. *IEEE Software*, 7(1):46–54, January 1990.

[31] S. Rugaber, K. Stirewalt, and L. Wills. Detecting interleaving. In *IEEE Conf. on Software Maintenance – 1995*, Nice, France, September 1995. IEEE Computer Society Press. To appear.

[32] R. Schwanke. An intelligent tool for re-engineering software modularity. In *IEEE Conf. on Software Maintenance – 1991*, pages 83–92, 1991.

[33] R. Schwanke, R. Altucher, and M. Platoff. Discovering, visualizing, and controlling software structure. In *Proc. 5th Int. Workshop on Software Specs. and Design*, pages 147–150, Pittsburgh, PA, 1989.

[34] P. Selfridge, R. Waters, and E. Chikofsky. Challenges to the field of reverse engineering – A position paper. In *Proc. of the First Working Conference on Reverse Engineering*, pages 144–150, Baltimore, Maryland, May 1993. IEEE Computer Society Press.

[35] D. Smith, G. Kotik, and S. Westfold. Research on knowledge-based software environments at Kestrel Institute. *IEEE Trans. on Software Engineering*, November 1985.

[36] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. on Software Engineering*, 10(5):595–609, September 1984. Reprinted in C. Rich and R.C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[37] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, I. Underwood, and A. Bundy. Deductive composition of astronomical software from subroutine libraries. In *Proc. 12th International Conference on Automated Deduction*, pages 341–55, Nancy, France, 1994.

[38] R. C. Waters. A method for analyzing loop programs. *IEEE Trans. on Software Engineering*, 5(3):237–247, May 1979.

[39] Mark Weiser. Program slicing. In *5th Int. Conf. on Software Engineering*, pages 439–449, San Diego, CA, 3 1981.

[40] L. Wills. Automated program recognition by graph parsing. Technical Report 1358, MIT Artificial Intelligence Lab., July 1992. PhD Thesis.

[41] E. Yourdon and L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* Prentice-Hall, 1979.