The Interleaving Problem in Program Understanding

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

PROGRAM
I BLE
UNDERSTANDING
T V

Abstract

One of the factors that can make a program difficult to understand is that code responsible for accomplishing more than one purpose may be woven together in a single section. We call this interleaving, and it may arise either intentionally – for example, in optimizing a program, a programmer may use some intermediate result for several purposes – or unintentionally, due to patches, quick fixes, or other hasty maintenance practices. To understand this phenomenon, we have looked at a variety of interleaving instances in actual programs and have distilled characteristic features. If the characterization proves to be robust then it will enable the design of tools for detection of interleavings and the extraction of the individual strands of computation.[1]

1 Introduction and Motivation

Sometimes the relations between the various sections make up a maze of interwoven threads that only detailed analysis can unravel.
– David and Mendel in *The Bach Reader.*

A program is an expression of its designers' efforts to accomplish some purpose. One aspect of understanding a program is to recreate that purpose. A particularly vexing difficulty in understanding a program is that a contiguous textual area of code can often contain fragments intended to accomplish multiple, seemingly unrelated purposes. The code fragments responsible for each of the purposes typically are delocalized and overlap rather than being composed in a simple linear sequence. We refer to these code fragments as being *interleaved.*

A trivial example is a single loop responsible for computing both the maximum element of a vector and its position. A less trivial example is a program intended to write a report that summarizes data extracted from sorted input records. The program has two purposes: computing the summary data and managing the construction of the report (headers, page breaks, page counts, etc.) In an object-oriented program, these two purposes might well be realized by two separate objects. In traditional code, however, the implementations of these functions are often interleaved, and understanding the code is significantly complicated.

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.[2] Plans can occur at any level of abstraction from architectural overviews to code. Interleaving expresses the merging of two or more distinct plans within some contiguous textual area of a program.

Interleaving may arise for efficiency reasons. For example, it may be more efficient to compute two related values in one place than to do so separately. Or interleaving may be the 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 typ-

[1] This paper discusses a number of examples of interleaving which we are making available on the World Wide Web at URL http://www.cc.gatech.edu/reverse/interleaving.html. We hope these will provide a set of challenges to other researchers interested in this problem, and we welcome additions to the set.

[2] This definition is distilled from definitions in [15, 21, 25]. Note that a plan is not necessarily stereotypical or used repeatedly; it may be novel or idiosyncratic. Following [21, 25], we reserve the term *cliché* for a plan that represents a standard, stereotypical form.

0-8186-7111-4/95 $4.00 © 1995 IEEE

ically 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 why 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, each individual code fragment responsible for a separate purpose is more spread out than it would be if it were encapsulated. Another reason why interleaving presents a problem is that it may be the result of poorly thought out maintenance activities, where the original, highly coherent structure of the system has degraded as "patches" and "quick fixes" are introduced. Finally, there may be occasions where interleaving is intentionally introduced, such as for purposes of optimization. But expressing intricate optimizations in a clean and well-documented fashion is not typically done. For all of these reasons, our ability to comprehend code containing interleaved fragments is compromised. Hopefully, we can have more success by isolating the separate concerns, understanding them individually, and only then seeing how they relate.

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. This paper describes our exploration, which has been based primarily on an examination of existing code from three sources:

- A Cobol database report writing system from the US Army (IMCSRS), which summarizes information provided as monthly updates to a master file of equipment maintenance information.
- A library of mathematical software (SPICELIB), written in Fortran by programmers at the Jet Propulsion Laboratory (JPL) for analyzing data sent back from space missions. The software performs calculations in the domain of solar system geometry.
- A program for finding the roots of functions (ZEROIN), presented and analyzed in [1] and [22].

The goal of the paper is to characterize the interleaving problem, relating it to existing concepts in the literature, such as delocalized plans [15], coupling [29], and redistribution of intermediate results [8, 9]. The paper presents and categorizes examples of types of interleaving to begin to map out the space of interleaving situations. It also describes how the decision to introduce interleaving into a program interacts with other design decisions. It then discusses implications for detection and extraction techniques and the types of representations needed to facilitate them.

2 Characterizing Interleaving

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.

To illustrate these primary characteristics, we focus on an example subroutine from SPICELIB. Since the program contains 113 source lines of code, the full text of the program is given in the appendix. Shorter, representative excerpts will be presented and discussed in the main body of the paper.

The program, called **NPEDLN**, computes the nearest point (**PNEAR**) on an ellipsoid to a specified line. It also computes the shortest distance (**DIST**) from the ellipsoid to the 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 algorithm used to compute the nearest point checks whether the line intersects the ellipsoid (the "intercept case") and, if so, computes the intersection point. Otherwise (the "non-intercept case") it performs a more complex computation to find the line segment connecting the ellipsoid and the given line that is orthogonal to both at its endpoints; the endpoints of the computed line segment form the closest pair of points.

2.1 Delocalization

Keep related words together. The position of the words in a sentence is the principal means of showing their relationship. Confusion and ambiguity result when words are badly placed.
– Strunk and White in *The Elements of Style*.

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. 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 1.

```
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 1: **Portions of the `NPEDLN` Fortran program. Shaded regions highlight the lines of code responsible for scaling and unscaling.**

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 distracting details that must be filtered out. *Delocalization* is one of the key characteristics of interleaving: one or more components of a plan are spatially separated from other components by the components of other plans with which they are interleaved. Letovsky and Soloway's cognitive study [15] shows the deleterious effects of delocalization on comprehension and maintenance.

The "scale-unscale" pattern is a simple example of a more general delocalized plan, ("transform-untransform") which is frequently interleaved with computations in `SPICELIB`. The general form, which we refer to as *reformulation wrappers*, is used to transform one problem into another that is simpler to solve, and then to transfer the solution back to the original situation. Some 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 three-dimensional intersection problems.

Delocalization may occur for a variety of different 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 1, 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. (The role of sharing in interleaving is discussed more extensively in Section 2.2.)

Realizing that a reformulation wrapper or some other delocalized plan is interleaved with a particular computation can help prevent severe comprehension failures during maintenance [15]. 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:

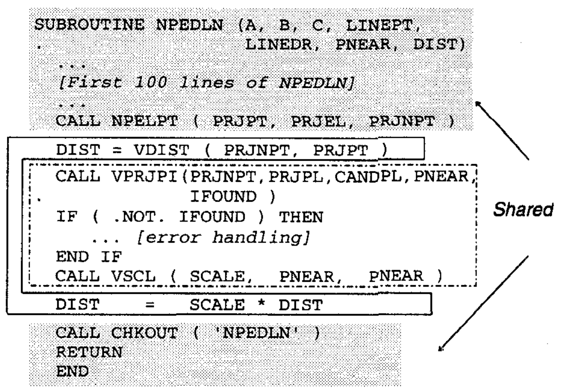
```
C- SPICELIB Version 1.2.0, 25-NOV-1992 (NJB)
C Bug fix: in the intercept case, PNEAR is now
C properly re-scaled prior to output.  Formerly,
C it was returned without having been re-scaled.
```

2.2 Resource Sharing

The complexity of multifunctioning elements can
sometimes turn data graphics into visual puzzles,
crypto-graphical mysteries for the viewer to decode.
– Edward Tufte in *The Visual Display of Quantitative Information.*

In addition to computing the nearest point on the ellipsoid to the line, `NPEDLN` 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 2. The shaded portions of the code shown are shared between the two computations for `PNEAR` and `DIST`. (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.)

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. In this case, interleaved plans share some common resource – intermediate data results. Their implementations



```
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 2: **Portions of NPEDLN, highlighting two overlapping computations.**

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 [8, 9]. 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, essentially introducing fanout in 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 entities, 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 non-intercept case is to be handled as is shown below.

```
        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
```

The use of control flags is a special form of *control coupling*: "any connection between two modules that communicates elements of control [29]," typically in the form of function codes, flags, or switches [16]. 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. The entire contents of a module may be lexically included in another or there may be partial overlap of modules. This is called *content coupling* [29] – "some or all of the contents of one module are included in the contents of another" – and 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.).

```
SUBROUTINE NPEDLN (A, B, C, LINEPT,
                   LINEDR, PNEAR, DIST)
 ...
 CALL UNORM ( LINEDR, UDIR, MAG )
 IF ( MAG .EQ. 0 ) THEN
    CALL SETMSG('Dir. is zero vector.')
    ... [error signaling]
    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=#.')
    ... [error signaling]
    RETURN
 END IF
 ... [scaling]
 IF (    ( SCLA**2  .LE.  0.D0 )
.    .OR.( SCLB**2  .LE.  0.D0 )
.    .OR.( SCLC**2  .LE.  0.D0 ))   THEN
    CALL SETMSG('Too small:A=#,B=#,C=#.')
    ... [error signaling]
    RETURN
 END IF
 ... [scaling]
 ... [handling the intercept case]
 ... [handling the non-intercept case]
 IF ( .NOT. XFOUND ) THEN
   CALL SETMSG ('Cand ellipse not found.')
   ... [error signaling]
   RETURN
 END IF
 ...
```

Figure 3: **Portions of NPEDLN, highlighting code responsible for checking preconditions.**

2.3 Independence

... You know they're going to get together ..., but it's fun to watch how they keep missing... It's emotionally satisfying because you get that cathartic moment at the end... It's intellectually satisfying because the plot's always twisting in on itself. – Susan Seidelman, director, on romantic comedy in *American Cinema.*

While interleaving is introduced to take advantage of commonalities, the flip side of the coin is 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 usually requires copying of common code to eliminate resource sharing, resulting in an equivalent, but possibly less efficient, program.

The NPEDLN program, for instance, computes the nearest point on an ellipsoid to a specified line. It also checks a set of preconditions which must be true for the computation to be carried out correctly: the line must not be a zero vector, the ellipsoid's semi-axes must have positive length and must be large enough to be scalable, and the ellipsoid must not be too flat or needle-shaped. Figure 3 shows an excerpt of NPEDLN, highlighting the precondition checking code.

In principle, all these checks can be performed beforehand, pulling them out of the midst of the primary computation and having them preface it. This would require duplicating almost the entire program, since the precondition checking makes heavy use of intermediate results computed throughout the program.

Although interleaving is 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 Implications for Representation, Detection, and Extraction

We have identified the characteristics of interleaving that make it troublesome in understanding programs. We can now consider the implications of what we have learned from our empirical study. In particular, we are interested in understanding how the decision to interleave interacts with other design decisions; what types of program representations will facilitate the detection of interleaving decisions and the extraction of interleaved components; and how can interleaved components be automatically detected and extracted from code?

3.1 Interactions with Other Decisions

Intentional interleaving is one of several ways in which design decisions are elaborated in code [22]. More familiar examples include the decomposition of a complex concept into its constituents (aggregation), the processing of a general problem in terms of various individual cases (specialization), the elaboration of a generic concept as a more concrete artifact (instantiation), and the modeling of one structure in terms of another (representation). The converse of interleaving is encapsulation, where a designer intentionally delineates several distinct design elements behind some sort of barrier, such as provided by the PACKAGE mechanism in Ada.

The history of the design of a program can be viewed as a network of artifacts [2] where a connection between two artifacts indicates that one of them is the refinement of the other that results from one of the kinds of design decisions listed above. In this view, interleaving is indicated by a network node with inputs from *two or more* other nodes. It may well be the case, however, that by tracing further back in the history some common ancestor of the two inputs can be found. For example, NPEDLN uses a control flag to indicate whether to handle the intercept or non-intercept case. In looking at the code for the subprogram, we see only instances of interleaved code fragments. In reviewing the design history, however, we may find that what is more fundamental is that a decision has been made to specialize the code into these two cases, that the flag distinguishes the two cases, and the interleaving is really just a manifestation of the higher level specialization decision.

In the case of the loop that is being used to compute both the maximum element in a vector and its index, there may have been an aggregation decision in the design history that justifies why both values are needed and only later an interleaving optimization to save some loop overhead.

In instances where a more fundamental decision has provoked a later interleaving, not only is the rationale connecting the two decisions usually lost, but we end up viewing the code in delocalized form when we really would like to see a factored version.

We postulate that interleaving often co-occurs with certain other design decisions. If further empirical study confirms this, then interleaving removal could be seen as *enabling* the reversal of other decisions or, dually, that certain design decisions enable interleaving by providing opportunities for resource sharing.

3.2 Requirements for Representations

Three key characteristics of interleaving are: delocalization, resource sharing, and independence. Delocalization results from having to serialize the components of two or more separate plans. 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 *undo* delocalization 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 search space 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), and
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* [19, 20], which was developed and used in the Programmer's Apprentice [21] project. 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 which 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.

3.3 Support for Detection and Extraction

Do not be afraid to seize whatever you have written and cut it to ribbons; it can always be restored to its original condition in the morning, if that course seems best. – Strunk & White in *Elements of Style*.

In order to develop tools for detecting and extracting interleaving, it is helpful to consider how interleaving manifests itself in source code. There are three useful, 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* for providing interleaving, which may be naming, control, data, or protocol (i.e., 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 the 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.

The third dimension is the *familiarity* of the plans interleaved: are they clichés (i.e., stereotypical, frequently used plans) or are they unfamiliar plans (i.e., novel, idiosyncratic, or not used repeatedly)?

When what is interleaved is *familiar* (i.e., a cliché), cliché recognition (e.g., [10, 12, 13, 14, 18, 28]) 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 [28], for instance, is its ability to deal with delocalization and redistribution-type function sharing optimizations.

KBEmacs [21, 26] 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 which 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) [28].

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 [3], focuses on *naming* mechanisms, by keying in on the patterns of linguistic idioms used in the program, which suggest the manifestations of domain concepts.

When *unfamiliar* plans are interleaved, other, non-recognition-based methods of delineation are needed. For example, slicing [27, 17] is a widely-used technique for localizing functional components by tracing through data dependencies within the procedural scope. Cluster analysis [3, 11, 23, 24] 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" [5], 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.

[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.

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 good candidates for abstract functions. Techniques have also been developed within the RE^2 project [6, 7], 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).

Interleaving at the scope of objects and architectures and/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 and more knowledge of the domain will be required to detect interleaving as the scope becomes more global in nature. Also, a key open issue is to what extent techniques for detecting and extracting interleaved plans must rely on the familiarity of the plans involved; how far can we go with non-recognition-based techniques?

4 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 under order number A870, 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. We would also like to thank the reviewers for their helpful comments.

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] I. Baxter. Design maintenance systems. *Comm. of the ACM*, 35(4), April 1992.

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

[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] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

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

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

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

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

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

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

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

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

[19] 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.

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

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

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

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

[24] 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.

[25] 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.

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

[27] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, 10:352–357, 1984.

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

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

5 Appendix: NPEDLN

```
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.DO )
.          .OR.   ( B .LE. 0.DO )
.          .OR.   ( C .LE. 0.DO )    )   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 numerical
C behavior.  If squaring any of the scaled lengths
C causes it to underflow to zero, signal an error.
C Otherwise scale the 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.DO )
.      .OR.  ( SCLB**2  .LE.  0.DO )
.      .OR.  ( SCLC**2  .LE.  0.DO )    )   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 ----------------------------------------------
C Descriptions of subroutines called by NPEDLN:
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 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.
```