Detecting Interleaving

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

The various goals and requirements of a system are realized in software as fragments of code that are typically "interleaved" in that they may be woven together in the same contiguous textual area of code. The fragments of code are often delocalized and overlap rather than being composed in a simple linear sequence. Interleaving severely complicates software comprehension and maintenance. To address this problem, we are developing analysis tools, based on the Software Refinery. This paper describes our experiences in detecting interleaving in a corpus of mathematical software written in Fortran from the Jet Propulsion Laboratory. In particular, it discusses how feasible it is to detect interleaving of various types and the ability of existing tools to assist these types of detection.

1 Introduction and Motivation

To understand a program, one often has to unravel multiple, interwoven strands of computation, each responsible for accomplishing a distinct purpose. 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 [13, 19]. 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.

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.

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 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 a variety of tasks, including extracting reusable components, localizing the effects of maintenance changes, and migrating to object-oriented languages.

What is needed is to isolate the separate strands of computation, understand them individually and then see how they interrelate. We are building analysis tools to do this. Traditional slicing techniques [25] are applicable to this problem but are not powerful enough to disentangle all forms of interleaving. They rely only on data and control flow; whereas our tools also use deeper knowledge of what plans are interleaved in the code.

1.1 Case Study Context

Software must be understood at multiple levels of abstraction simultaneously. Plans can occur and be interleaved at any abstraction level from source code text to application domain structures. The process of

understanding a piece of software involves two parallel knowledge acquisition activities [4, 16, 23]:

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 help to 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. In particular, we have a corpus of software and an incomplete model of the software's application domain. We are targeting our detection of interleaving toward elaborating the existing domain model for this software. We are also looking for ways in which the current knowledge in the domain model can guide detection and ultimately comprehension.

SPICELIB

Our corpus of software is a library, called `SPICELIB`, of approximately 600 mathematical programs, written in Fortran 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, such as coordinate frame conversions, intersections of rays, ellipses, planes, and ellipsoids, and light-time calculations.

Amphion Domain Model

We have obtained a partial model of the application domain of this software from NASA Ames researchers, who have developed a component-based software synthesis system, called Amphion [14]. Amphion composes routines from `SPICELIB` 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 library 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 code.

In collaboration with NASA Ames researchers, we are detecting ways in which the domain model Amphion uses is incomplete and developing program comprehension techniques to extend it.

Some of 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 (perhaps extracted from legacy software), and to improve the software synthesized. From the program comprehension 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.

There are at least two ways in which the existing domain model is incomplete that are particularly interesting from the point of view of interleaving. One incompleteness is that the model does not fully cover the functionality of the routines. Some routines compute more than one result (e.g., the nearest point on an ellipsoid to a line and the shortest distance between that point and the ellipsoid). However, the domain model does not always capture all the values that are computed. 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 model. A second example of incompleteness is that the current domain model does not capture preconditions on the use of the library routines (e.g., that 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). The code responsible for checking these preconditions is usually tightly interleaved with the code for the primary computation as it is sprinkled throughout the routine and often uses intermediate results computed for the primary computation.

Software Refinery

We are developing mechanisms for detecting various classes of interleaving with the aim of growing a partial model of the application domain. We are building these mechanisms on a collection of commercial tools, called the Software Refinery [11]. 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 [22], 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.

1.2 Contributions and Outline of Paper

We have developed an initial characterization of interleaving [20], based primarily on an empirical study of `SPICELIB`. We briefly summarize our characterization in Section 2. We then present a detailed example of one of the mechanisms we have built to detect a particular type of interleaving – the interleaving of exception-handling code with the program's primary computation. This simple but pervasive type of interleaving is particularly useful to detect and extract for the purposes of elaborating specifications with preconditions. In Section 4, we describe a set of analyses that we have formulated to detect various types of interleaving, in addition to exception handling. We discuss the results we have obtained so far in implementing these analyses using the Software Refinery and predict what is needed to carry out the rest. In Section 5, we reflect on our experiences in using the Software Refinery to build interleaving detection mechanisms and discuss future directions we would like to explore in detecting and extracting interleaving.

2 Characterizing Interleaving

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 performance of multiple, *independent* roles in the program's overall purpose. (Our characterization of interleaving and several detailed examples are presented in more depth in [20].)

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. This makes it difficult to gather together all the pieces to ensure consistent maintenance [13]. Distracting details also get in the way and must be filtered out from the midst of the delocalized plan.

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.

There may also 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. 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. Often this relationship centers on some common resource, such as intermediate data results, control flags, and lexical module structures. This causes the implementations of the interleaved plans to overlap in that a single structural element contributes to multiple goals. 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 one of the actual uses of the resource (variable, value, control flag, data structure slot, etc.).

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. Although interleaving is necessary for efficiency, it obscures the independence of the components involved. Ironically, this hinders activities like parallelization and objectivization that improve the efficiency and reusability of the code.

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.

3 Extraction of Preconditions

Using the Software Refinery, we have been able to automate 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. In particular, they are usually part of a larger plan that detects exceptional (usually erroneous) conditions in the state of a running program, and takes alternative action when these conditions arise (such as returning with an error code, signalling, or invoking error handlers).

Precondition checks make explicit the assumptions a subprogram places on its inputs. Ideally a tool to

```
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
C     vector. If it is 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: Input vector is zero vector.' )
         CALL SIGERR ( 'SPICE(ZEROVECTOR)' )
         CALL CHKOUT ( 'SURFPT' )
         RETURN
      END IF
      ...
```

Figure 1: 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.

aid in program understanding removes the interleaved precondition checks from the code and uses the information they represent to help the user *elaborate* a high-level specification of the subprogram. Such a tool addresses program understanding by extracting an interleaved plan, and by assisting in the composition of a specification by suggesting subprogram preconditions. We have created a tool that detects these checks and extracts the preconditions into a documentation form suitable for expression as a partial specification.

We found numerous examples of precondition checks in our empirical analysis of the SPICELIB. One such check occurs in the subprogram SURFPT shown in Figure 1. 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).

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 handle exceptions. 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 of this is that the Fortran specific portion is not likely to need changing when we apply the tool to a new application; whereas the application specific portion will certainly need to change. With this in mind, we chose a tool architecture that gives a great deal of freedom in the expression of the procedure for detecting exception plans.

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 plan for handling an exception. In some other application, the form of this plan will be much different. It is, therefore, necessary to design the plan detection 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). Briefly, a rule is a construct in the Refine language that encapsulates a transformation of the state of the system (that is, a side effect). Rules are specifically designed to be applied to the nodes of an AST during tree walks, so they always have a single input parameter that is some subclass of AST nodes. Rules are more declarative than functions in that they specify state changes by listing the *conditions* before and after the change without specifying exactly how the change must occur. Syntactically, this is expressed through the form: *preconditions* $\rightarrow$ *postconditions*. 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 represent application specific exception handler plan clues using two rules. The first searches for a RETURN statement in an AST:

```
rule IS-RETURN ( s : rf::program-unit-statement )
   ~*FOUND-RETURN* &
   rf::return-statement(s)
   -->
   *FOUND-RETURN*
```

This rule states that if, during an AST walk, a RETURN statement has not yet been discovered (represented by the negating the value of the boolean global variable

FOUND-RETURN) and the current AST node (bound by the parameter s) in the tree walk is of class rf::return-statement, then establish that a RETURN has been found (similarly represented by the global variable *FOUND-RETURN*). The careful reader may note that the first conjunct in the precondition of this rule is not logically necessary (~*FOUND-RETURN*). It is included here because rule application in Refine continues until no more rules can be matched on any AST node. So rules must somehow invalidate their precondition when they are applied or else the rule application will go into an infinite loop.

The other rule we use to analyze SPICELIB is:

```
rule IS-SIGERR ( s : rf::program-unit-statement )
   ~*FOUND-SIGERR* &
   rf::call-statement(s) &
   rf::identifier-name(rf::called-object(s)) =
                                 'RFU::SIGERR
   -->
   *FOUND-SIGERR*
```

This rule is similar to the rule to detect RETURN, but in this case we are checking not only that the current AST node (s) belongs to a certain class (rf::call-statement), but also that the name of the called object is SIGERR. The code that applies these rules to sequences of statements is shown in Figure 2. Note here the application of the function preorder-transform which applies a sequence of rules to an AST until no more rules apply. We specify the actual sequence of rules to apply by defining the variable *CHECK-EXCEPTION-RULES* as follows:

```
var *CHECK-EXCEPTION-RULES* : seq(symbol) =
  [ 'IS-RETURN,
    'IS-SIGERR ]
```

That is, *CHECK-EXCEPTION-RULES* is a variable whose type is a sequence of symbols, assigned to the literal sequence of symbols denoting the rules we have defined.

Detecting Guards Discovering IF statements that depend only upon input parameters (guards) 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. We address this problem in our analysis by using an approximate dataflow algorithm which propagates a set of immutable variables through the sequence of statements in the subroutine. (We do this in the absence of a robust and complete dataflow analysis tool.) At each

```
"Given a sequence of Fortran statements
 (assumed to be the body of an IF-THEN-ELSE
 block), report whether or not the statements
 appear to be raising an exception."
function Looks-Like-Exception(
        stmt-seq : seq(program-unit-statement) ) :
boolean =
   let (*FOUND-RETURN* : boolean = false,
        *FOUND-SIGERR* : boolean = false)
      (enumerate a-stmt over stmt-seq do
         preorder-transform(
               a-stmt,*CHECK-EXCEPTION-RULES*));
      *FOUND-RETURN* & *FOUND-SIGERR*
```

Figure 2: Refine code that checks for exception plans by applying the pattern our initial analysis allowed us to deduce.

statement, if a variable X in the set might be modified by the execution of the statement, then X is removed from the propagating set. We define a Refine function Propagate-Through-Statement that does this propagation on a per statement basis. The function is used in our function Get-Preconditions shown in Figure 3.

Intuitively, Get-Preconditions computes the parameters of its input subr using Refine/Fortran primitives and then puts these parameters into a working set worklist that is propagated through statements in the subprogram. The local variable candidates is used to store the candidate preconditions found during a propagation. When an IF-THEN-ELSE statement is found, candidates is updated to be the concatenation of its previous value and a new sequence of preconditions. The function Match returns these sequences of preconditions, returning the empty sequence if the given IF statement did not match our pattern.

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. We include an example here of the preconditions for the subroutine SURFPT. When applied to SURFPT our tool generated the LaTeX source which when included without change into this document looks like:

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

Taken literally, this 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

```
function Get-Preconditions( subr : program-unit ) :
condition-tuple-type =
  let ( parameters : set(symbol) =
         { Get-Parameter-Name(p) |
           (p) p in formals(unit-heading(subr)) })
    let (worklist   : set(symbol) = params,
         candidates : condition-tuple-seq = [])
       (enumerate s over unit-body(subr) do
          (if block-if-statement(s)
           then candidates <-
                  concat(candidates,
                         Matches(s, worklist)));
          worklist <- Propagate-Through-Statement(
                                s,worklist));
        candidates
```

Figure 3: Given the AST representation of a Fortran subprogram, compute a candidate set of preconditions for the subprogram.

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 more abstract precondition.

4 Experiences

We formulated a number of other interleaving analyses in addition to precondition detection. Each answers some empirical question related to interleaving in code libraries. Since some depend upon application domain information, we took advantage of the information encoded in the Amphion domain axioms. We were, however, careful to design the tools so as not to restrict their applicability to this particular domain. We have used our experience with the Software Refinery to implement some of these analyses and to estimate the difficulty of implementing the others.

Indefinite Loops and Dataflow Analysis Precision. Most of the analyses for detecting interleaving rely on some form of dataflow analysis. Dataflow analysis propagates data usage information along all possible *sequences* of statements in a subprogram. The accuracy or precision of a dataflow analysis depends upon the ability to cover all such sequences. Unfortunately, programs with loops can sometimes represent *indefinite* sequences of statements. When posed with such programs we must settle for only approximate dataflow information. On the other hand, subprograms whose loops are bounded by constants may be *unrolled* into code each of whose statement sequences may be determined statically. Such subprograms can be analyzed exactly with dataflow techniques. Loops of this nature occur frequently in `SPICELIB`. For example, because three-dimensional vectors are stored as arrays of length three, it is typical to see many `DO` loops with only three iterations.

Since so many of our interleaving analyses depend upon the precision of dataflow analyses, we would like some measure of a tool's maximum analysis potential over the library. We chose to measure this as the percentage of routines with only definite (statically determinable bounds) loops and developed a statistics gathering tool in Refine. The tool works by defining rules to detect indefinite loops and then applying these rules through an AST walk similar to that of the precondition detection tool.

The analysis showed that roughly 68 percent of the subprograms had no indefinite loops. This number indicates that we will be able to completely analyze two thirds of the routines. Moreover, in the other cases, approximate dataflow information may be good enough to capture the spirit of the analysis.

Routines with Multiple Outputs. Some subroutines in `SPICELIB` 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. This interleaving not only complicates the task of understanding the code, but also clouds a maintainer's conceptual categorization of the subroutine.

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`:

```
SUBROUTINE SURFPT (POSITN,U,A,B,C,POINT,FOUND)
```

which conceptually returns the intersection of a vector with the surface of an ellipsoid. However, it is possible to give 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.

Fortunately, the Software Refinery provides a package to create structure chart (call graph) objects. The nodes of these structure charts are annotated with direction information about parameters. We were thus able to build our entire analysis using only the structure chart object without having to do *any* Fortran AST analysis. This greatly simplified the implementation.

The resulting analysis showed that 25 percent of the subprograms had multiple output parameters. We were thus able to focus our work on these routines first, as they are likely to involve interleaving.

Domain Model Coverage. NASA has developed a formal model of `SPICELIB`'s application domain. This model is incomplete with respect to the library. One analysis that would focus efforts to complete the domain model stems from a notion of *coverage*. We say a subprogram is an element of the cover induced by a domain model if it is either directly linked to some entity in the domain model or is invoked by another subprogram that is in the cover. A domain model covers a library if every subprogram in the library is in the domain model's cover.

We constructed a tool that determines the set of subroutines in the cover of a domain model. The rich choice of data structures in Refine significantly eases this task. The major constituent of the tool converts a structure chart into a relation that maps subprograms to the subprograms that they call and then computing the transitive closure of this relation. Since Refine provides a transitive closure operation (`tclosure`) in the language, this was a simple task. The result of this analysis was that only 35 percent of the library was covered by the domain model.

Dead End Dataflows. Many of the `SPICELIB` subprograms that exhibit interleaving also exhibit a behavior detectable in the domain model, called a "dead end dataflow." It occurs when the domain model references a subprogram in the library, and this subprogram returns a value that is not mapped to anything in the domain model. For example, a `SPICELIB` subprogram that computes the nearest point on an ellipsoid to a line also computes the shortest distance between the line and the ellipsoid, but only the near point output is mapped to a domain entity (the nearest point); the distance output is a dead end dataflow. 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% have some output parameters that are dead end dataflows.

Control Coupling. Sometimes a programmer will implement a subprogram that uses one of its input parameters as a flag to choose among a set of possible computations to perform. This is a form of control coupling [27]: "any connection between two modules that communicates elements of control," and it is a class of interleaving involving delocalized control. Subprogram calls with constant parameter values are potentially involved in control coupling. Our strategy for detecting control coupling instances first computes a set of candidate subprograms that are invoked with a constant parameter in the library or the 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 not yet implemented this analysis, but the two components of our strategy should be easy to implement in Refine. In addition to applying this information to the elaboration of specifications, we are also interested in seeing if the strategy is significantly more precise than the following variant. It seems likely that if every call of a subprogram passes a constant parameter, then that subprogram is involved in control coupling interleaving. We expect to test this variant along with the original analysis to see if there is a correspondence of results.

Routine Co-occurrence Properties. In [20] we described a general form of interleaving called *reformulation wrappers*. A reformulation wrapper 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.

Often reformulation wrappers are realized as two functions: one that transforms a problem into a different (usually easier or more stable) problem, and one that transforms the result back. We would like to detect instances of this in arbitrary codes, and we believe that the key to this detection is embodied in a notion of subprogram *co-occurrence*. We say that two subprograms S_1 and S_2 in a library co-occur 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 flow of computed data from S_1 to S_2.

This is by far the most ambitious interleaving analysis we have mentioned here. Its solution requires substantial dataflow infrastructure. To get a feel for this requirement, we examine each point in the definition.

Given a pair (S_1, S_2), it is relatively simple to determine if each subprogram in the library references both of them. Refine provides language constructs for specifying first order logic predicates. Using this feature of the language, we could implement this test as:

```
fa(x)(x in *SPICELIB* =>
        (contains(x,s1) => contains(x,s2)) &
        (contains(x,s2) => contains(x,s1)))
```

and define the function `contains` to return whether or not the subprogram in the first parameter contains a call to the subprogram in the second parameter. It should be obvious from our other examples that the `contains` function is simple to construct in Refine.

We can express the execution conditions regarding S_1 and S_2 using a construct from flow analysis called a dominator [1]. We say that the call of S_1 *dominates* the call of S_2 if the call of S_2 implies that S_1 has already been called in that routine. Similarly, the call of S_2 *post-dominates* the call of S_1 if calling S_1 implies there will be a call to S_2 in that routine. Refine does not provide an analysis for dominators, but we have built our own from other Refine packages.

To complete the analysis of reformulation wrappers, we still need to check that data computed in S_1 somehow flows into the inputs of S_2. To compute this, we need a full dataflow analysis component. Refine does not provide such a component, but we are working on one of our own.

5 Conclusions

Interleaving is a pervasive and troubling problem. Disentangling interleaved program strands can improve understandability and provide opportunities for reuse, thus extending the value and lifetime of software assets. Detecting and extracting interleaved strands is a complex problem that we are just beginning to understand. This section describes the strengths and weaknesses of available tools and the successes and failures we have had in understanding instances of interleaving. We also comment on the role of domain information in understanding interleaved code and our plans for continuing to explore this area.

5.1 Tools

We used the Refine tools from Reasoning Systems in our analysis. 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 analysis. We made particular use of two other features of the toolkit. The first was 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 pre-existing analyses can be accessed from the repository using small, Refine language programs such as those described in this paper. 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. By merely defining pre- and post-conditions we are easily able to define the properties of constructs without worrying about how to find them. We had merely to add a simple tree walking routine to apply the rules to the AST. Furthermore, the language 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 reduced 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 also provided a robust handling of Fortran constructs that are not typically available from non-commercial research tools. Finally, we had occasion to take advantage of the existing Refine user community through its mailing list (`refine-users@grace.rt.cs.boeing.com`). Our technical questions were answered nearly instantaneously and always accurately.

We can see several ways in which the Refine approach can be further extended. In particular, the

availability of other analyses, such as control flow graphs for Fortran and general dataflow analysis, would have given us more leverage. The Refine language itself might be extended further, such as by supporting function parameters and more transparent rule application.

In summary, the Refine approach and tool suite provided extensive leverage without which we would not have been able to adequately explore the interleaving question.

5.2 Interleaving

In Section 2, we characterized interleaving in terms of delocalization, resource sharing, and independence. The particular analyses we performed can be viewed in terms of these dimensions. For example, precondition checks are often spread throughout (delocalized in) a routine. The same holds true for reformulation wrappers. In the absence of full dataflow analysis, we approximated the actual analysis we would like to have made, albeit in a safe way. That is, all of the results produced were accurate, but some additional ones may have been missed due to missing information about dataflow relationships. However, we expect these to be few in number.

Resource sharing and independence are related concepts. There must be some reason, usually expressible as a shared resource, for two independent fragments to be interleaved. For example, routines with multiple outputs share intermediate computations as a way of improving run-time efficiency. This particular kind of interleaving was also easily detected.

In general, the detection and extraction of interleaved fragments is a hard problem. It is related to *overlapping implementations* [18], *potpourri module detection* [5], *slicing* [25, 15], *cluster analysis* [2, 10, 21], and *objectivization* (the extraction of candidate objects from non-object oriented programs) [3, 6, 7]. To the extent that the analysis can be based on program information only, the approach we have undertaken appears adequate. However, interleaved fragments are instantiations of plans, and many plans are direct expressions of application domain requirements. Hence, extensive use of domain knowledge may be a prerequisite to complete analysis.

5.3 Domains

In our analysis, we had the benefit of an existing partial domain model. The domain model provides expectations and data to be used in analyzing a program. For example, there were several readily distinguishable instances in the domain model of pairs of routines that we used as candidates for detection of reformulation wrappers. On the other hand, we have been able to use the results of program analysis to validate and extend the domain model. For example, there were situations where a routine computed several results, but the domain model only described one of them.

The simultaneous elaboration of application and program understanding is characteristic of Synchronized Refinement, the overarching approach we take to reverse engineering [16]. How best to express and use the domain information is still an open question, however, but some discussion of the issues is provided in [8]. We expect the use of domain information to be a prerequisite for further significant increases in the power of program understanding technology.

5.4 Future Directions

The work described here raises several further questions related to interleaving. One of these concerns the appropriate level of abstraction in the domain model. In our case, the domain model that we used was fairly low level. If the ultimate goal is to map a program back to its requirements, then, program constructs must be described in terms of domain vocabulary. For example, if we can detect certain stereotypical constructs (called *clichés* [19]) and map them back to a domain concept (such as an ellipsoid or an orthogonal projection), then we can use our knowledge of geometry to further understand and describe a program. This requires the use of cliché recognition techniques (e.g., [12, 17, 26]). It also raises the question of describing the domain itself in Refine, and consequently being able to reason about it.

We would also like to look at architectural issues. In particular, the analyses we performed were fairly low level, and, in fact, the `SPICELIB` software itself has a fairly straightforward architecture. But this will not always be true. In fact, architectures can be interleaved just like programs. This problem appears to be quite hard, but we can see attacking it both from the top down, by trying to detect instances of specific architectural styles [9], and from the bottom up, by trying to detect specific kinds of module groupings, such as the afferent, efferent, and transformer modules that are a part of Structured Design [24].

Acknowledgments

Support for this research has been provided by ARPA under order number A870, contract number NAG 2-890. We would like to thank Larry Markosian for helping us to obtain the Software Refinery and the `NAIF` group at `JPL` for enabling our study of `SPICELIB`. We also benefited from insightful discussions with Michael Lowry at Nasa Ames Research Center concerning this research.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[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] 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.

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

[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] J-M. DeBaud, B. Moopen, and S. Rugaber. Domain analysis and reverse engineering. In *IEEE Conf. on Software Maintenance - 1994*, pages 326–335.

[9] A. Garlan and M. Shaw. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Englewood Cliffs, NJ, 1995.

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

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

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

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

[14] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. A formal approach to domain-oriented software design environments. In *Knowledge-based Software Engineering Conference*, 1994.

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

[16] 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.

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

[18] 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.

[19] C. Rich and R. C. Waters. *The Programmer's Apprentice*. Addison-Wesley and ACM Press, 1990.

[20] S. Rugaber, K. Stirewalt, and L. Wills. The interleaving problem in program understanding. In *Proc. of the Second Working Conference on Reverse Engineering*, July 1995. IEEE Computer Society Press.

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

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

[23] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. on Software Engineering*, 10(5):595–609, September 1984.

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

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

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

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