Object-Oriented and Concurrent Program Design Issues in Ada 95

Stephen H. Kaisler
Director, Systems Architecture
Office of the Sergeant at Arms

United States Senate
Washington, DC 20510
202-224-2582
Steve_Kaisler@saa.senate.gov

Michael B. Feldman
Professor of Engineering and Applied Science
Dept. of Electrical Engineering
and Computer Science
George Washington University
Washington, DC 20052
202-994-6083
mfeldman@seas.gwu.edu

1. ABSTRACT

Kaisler [4] identified several design issues that arose from developing a process for transforming object-oriented programs written in Ada 95 or other object-oriented programming languages to process-oriented programs written in Ada 95. These issues limit - in the authors' opinion - the flexibility of Ada 95 in writing concurrent programs. These problems arise from specific decisions made by the Ada 95 designers. This paper addresses three of these issues: circular references are invalid in packages, lack of a subclassing mechanism for protected types, and an inability to internally invoke access statements from within a task body. Our goal is to raise the Ada 95 community's awareness concerning these design issues and provide some suggestions for correcting these problems for a future revision of the Ada programming language.

2. INTRODUCTION

Kaisler [4] described research leading to a methodology for converting object-oriented programs to process-oriented programs and, thereby, making the concurrency implicit in object-oriented programs explicit. This research was motivated by the fact that writing concurrent programs is difficult to do - no general methodology exists - and by our recognition of the similarity between object-oriented and process-oriented programs.

We believed that object-oriented programs, which exhibit a potential for concurrency, could be converted to process-oriented programs, thereby making the concurrency explicit. Object-oriented programs exhibit the property of encapsulation, i.e., that attributes and methods of a class are defined in a single syntactic structure. Similarly, process-oriented programs also exhibit encapsulation within the process or task structure. Message passing is a basic mechanism for communication within object-oriented and process-oriented programs. Analysis of these features suggested that object-oriented programs could be converted to process-oriented programs with no loss of functionality. Rather than writing concurrent programs anew, our work focused on reusing the source code and logic of an object-oriented program to develop the process-oriented program. We developed a methodology and transformation algorithms for converting both the structure and the procedural code of the object-oriented program to the corresponding components of a process-oriented program.

However, as we applied the methodology and the algorithms to several programs written in Ada 95 and other object-oriented programming languages (OOPLs), we discovered several limitations of Ada 95 that arise from specific decisions made during the language design process. This paper describes three of these limitations and proposes some possible modifications to the Ada programming language for consideration in a future revision.

3. CIRCULAR REFERENCES IN ADA 95 PACKAGES

Perhaps the most constraining feature of Ada 95 is its inability to support circular package references among two or more packages. In C++ [9], references may be made both forward and backward to other class definitions, which can result in circular references between two files containing C++ class definitions. The C++ compiler notes the occurrence of such circular references and generates the appropriate code without entering an infinite loop. Circular references between objects seem to be a natural outcome of O-O analysis and design methodologies. Circular references are allowed between Eiffel types [5, 6], Sather classes [7], and Common Lisp objects [8] among other modern object-oriented programming languages.

An example of a circular reference from the Sather 1.0 chess program, a demonstration program from the Sather distribution, is depicted in figures 1(a) and 1(b).

```
type $PIECE is
alive:BOOL;
alive(set:BOOL);
worth:INT;
iswhite:BOOL;
position:POS;
valid_move(to:POS,board:BOARD):BOOL;
update_position(position:POS);
update_position(position:STR);
move!(b:BOARD,to_piece:BOOL):POS;
fig:CHAR;
ispawn : BOOL;
isrook : BOOL;
isking : BOOL;
end; -- of type $PIECE

-----------------------------------------------------------------
class PIECE < $PIECE is

  -- General constants that are used throughout
  -- the descendants of $PIECE

  const white         : BOOL := true;
  const black         : BOOL := ~white;
  const ordinary      : BOOL := false;
  -- alters behavior of move!

  const for_check_test : BOOL := true;

   -- Attributes that are specific to a PIECE

  attr  alive    : BOOL;
  attr  iswhite : BOOL;
  attr  position : POS;

  -- Constants that are specific to a PIECE

  const worth    : INT := 0;
  const fig      : CHAR := ' ';

  const ispawn  : BOOL := false;
  const isking  : BOOL := false;
  const isrook  : BOOL := false;

   ••• <other code omitted here!>

valid_move(to:POS,board:BOARD):BOOL is
    ret : BOOL := false;
    loop
          valid_to::=move!(board,ordinary);
          if to=valid_to
                then ret:=true;
                break!;
          end;
    end;
    return ret;
  end;
```

Figure 1a. PIECE Class Declaration

```
class BOARD is

  private attr whitepieces   : ARRAY{$PIECE};
  private attr blackpieces   : ARRAY{$PIECE};

  -- The board stores information about which color
  -- is to play next.

   ••• <other code omitted here!>

 private move_valid_so_far(move:MOVE):BOOL
  pre ~move.isquit
  is
    ret : BOOL := false;

    -- A valid move must start at a position where
    -- one of my pieces is....

    if has_my_piece(move.from)
        then
           p::=piece(move.from);

    -- ... and it must be a valid move with respect
    -- to the mobility of the piece at the current
    -- state of the board.

      if p.valid_move(move.to, self)
```

```
      then
         ret := true;
         --
         -- since the move seems to be valid,
         -- the moving piece is stored
         -- in the move object. That eases future
         -- access to the moving piece
         -- and allows for un-doing of moves.

        move.piece := p;

      end;
    end;
    return ret;
  end;
```

Figure 1b. BOARD Class Declaration

The corresponding references in the two classes depicted in the figure above are highlighted in bold. The individual chess pieces (pawn, knight, rook and so forth) each inherit attributes and methods from the class PIECE. Because PIECE references the BOARD class through typing of arguments to its methods, each of the objects for the chess pieces must also do so. The BOARD class references the PIECE class directly in the array declarations and objects for each chess piece class through dynamic dispatching from the statement:

```
if p.valid_move(move.to, self) then
•••
```

where P is an instance of a chess piece class which has inherited valid_move from the PIECE class. P takes as its value a handle of one of the instances of the chess pieces. So, at run-time, the appropriate implementation of valid_move is determined by the class to which P belongs.

Figures 2(a) and 2(b) depict the headers for C++ files for Panorama and Scrollpane that demonstrate a circular reference used in a user interface application. As before, the circular references are highlighted in bold.

```
class CPanorama : public CPane
{                               // Class Declaration

public:

                                // Data Members

   // Bounds defining Pane coordinates
   LongRect bounds;
   // Pixels per horizontal unit
   short          hScale;
   // Pixels per vertical unit
   short          vScale;
   // Location of frame in Panorama
   LongPt         position;
   // Save for later restoration
   LongPt         savePosition;

   // Here is the circular reference!!

   // Scroll pane for this Panorama
   CScrollPane  *itsScrollPane;

                        // Member Functions
                        // Construction/Destruction

   CPanorama();
   CPanorama(CView *anEnclosure,
                CBureaucrat *aSupervisor,
                short aWidth = 0, short aHeight = 0,
                short aHEncl = 0, short aVEncl = 0,
                SizingOption aHSizing = sizELASTIC,
                SizingOption aVSizing = sizELASTIC);

   <Other declarations of methods - not relevant here>

   };
```

Figure 2(a). C++ Header for Panorama Class

```
class CScrollPane : public CPane
{
                                // class DECLARATION
public:

                                // Data Members

   // Here is the circular reference !!

   CPanorama   *itsPanorama;       // View which scrolls
   CScrollBar  *itsHorizSBar;      // Horizontal scroll bar
   CScrollBar  *itsVertSBar;       // Vertical scroll bar
   CSizeBox    *itsSizeBox;        // Grow box
   long        hExtent;
   long        vExtent;
   short       hUnit;
   short       vUnit;
   short       hSpan;
   short       vSpan;
   short       hStep;
   short       vStep;
   short       hOverlap;
   short       vOverlap;

                // Member Functions
                // Construction/Destruction

   CScrollPane();
   CScrollPane(CView *anEnclosure,
                CBureaucrat *aSupervisor,
                short aWidth, short aHeight,
                short aHEncl, short aVEncl,
                SizingOption aHSizing,
```

```
                SizingOption aVSizing,
                Boolean hasHoriz,
                Boolean hasVert,
                Boolean hasSizeBox);

<Other declarations of methods - not relevant here >

};
```

Figure 2(b). C++ Header for Scrollpane Class

Ada 95 does not allow circular references between packages because all packages are elaborated prior to the beginning of execution. Circular references would occur when one package WITHs another package and the second package attempts to WITH the first package. In Ada 95, circular references are detected at compile time with an error message; compilation is terminated. Using the methodology and transformation algorithms in [4], we converted the above code to Ada 95. Upon compilation, which terminated, the circular reference was detected.

Norman H. Cohen [3] suggested one workaround to achieving circular references among packages where tagged records are involved. The record fields to be cross-referenced are removed from the tagged records and moved to new tagged records from which the original ones inherit. This approach is depicted in figures 3(a) through 3(d) from packages used in a user interface application, which was derived from the C++ code depicted above. So, ui_scrollpane_root is the parent record of ui_scrollpane and ui_panorama_root is the parent record of ui_panorama. Note that ui_scrollpane_record cross-references an access variable for ui_panorama. Similarly, ui_panorama_record cross-references an access variable to ui_scrollpane_record.

By inspecting the code in the figures above, we see that a panorama refers to a scrollpane and vice versa. Ada 95 does not allow cross-referencing between these two packages and their tagged records to take place. But, by extracting the information required in the other package from each package and moving it to a root record, the panorama record can reference the scrollpane_root and vice versa. This is depicted graphically in figure 4.

```
package ui_scrollpane_root is

type UI_ScrollPane_Root_Record is
    new UI_Pane_Record with
        record
            myHorizScrollBar: Any_ScrollBar;
            myVertScrollBar: Any_ScrollBar;
            •••
        end record;

type UI_Scrollpane_Root is
    access all UI_Scrollpane_Root_Record'Class;

subtype Any_Scrollpane_Root is UI_Scrollpane_Root;

end ui_scrollpane_root;
```

Figure 3(a). Scrollpane Root Class

```
package ui_scrollpane is

 type UI_Scrollpane_Record is
    new UI_Scrollpane_Root_Record with
        record
            myPanorama: Any_Panorama;
        end record;

 type UI_ScrollPane is
        access all UI_ScrollPane_Record'Class;

 subtype Any_Scrollpane is UI_Scrollpane;

 procedure AdjustScrollMaximum(
    aScrollPane: access UI_ScrollPane_Record'Class);

end ui_scrollpane;
```

Figure 3(b). Scrollpane Class

```
package ui_panorama_root is

type UI_Panorama_Root_Record is
    new UI_Pane_Record with
        record
            bounds: Rectangle;
            •••
        end record;

type UI_Panorama_Root is
    access all UI_Panorama_Root_Record'Class;

subtype Any_Panorama_Root is UI_Panorama_Root;

end ui_panorama_root;
```

Figure 3(c). Panorama Root Class

```
package ui_panorama is

type UI_Panorama_Record is
    new UI_Panorama_Root_Record with
        record
            myScrollpane: Any_Scrollpane_Root;
        end record;
```

```
type UI_Panorama is
      access all UI_Panorama_Record'Class;

subtype Any_Panorama is UI_Panorama;

end ui_panorama;
```

Figure 3(d). Panorama Class

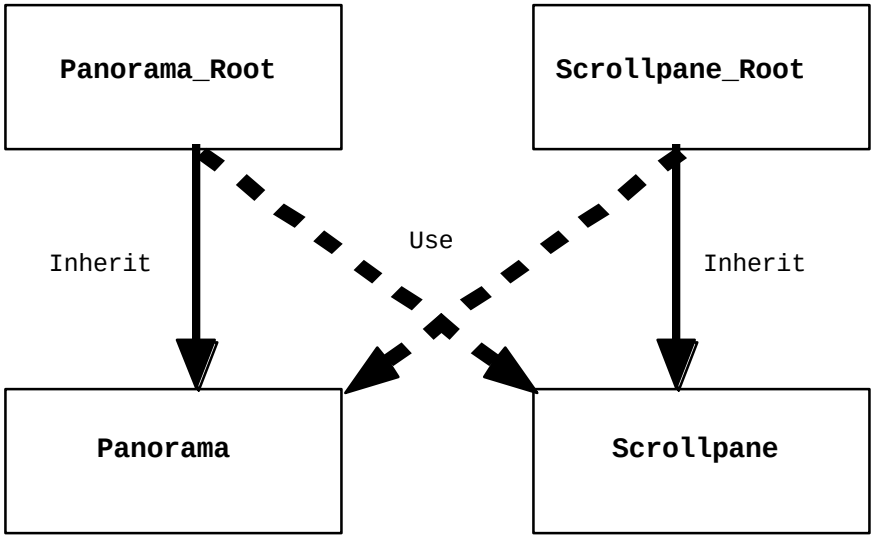

Figure 4. Graphical Depiction of Legal Cross-Referencing

An alternative approach is to duplicate methods from one class as internal procedures in the calling class. This eliminates the circular reference problem because the calling package now calls the routine internally. However, this approach is not always feasible because of interdependencies between the procedures and the data structures. In converting the Sather Chess program to Ada 95 using the methodology described in [4], this approach was used to resolve one case of circular referencing.

A number of remedies are possible for this problem. We suggest that at a minimum, an elaboration pragma could be provided by compiler writers to ignore checking circularities among packages. Programmers would code the pragma in their source code, but would need to be fully aware of the implications of doing so. At run time, it is possible that an infinite loop could result from an erroneous circular reference. Programmers would be fully responsible for checking for and avoiding of the occurrence of infinite loops; the run-time environment might be unable to provide any help.

A stronger recommendation would be to allow referential circularity in data structure declarations, but not among procedures contained in two different packages. At compile time, the compiler can recognize elaboration circularities and develop the appropriate information to pass to the binder and the linker for storage allocation. The run-time environment could detect infinite loops among data structure references and generate a constraint error.

The strongest recommendation would be to enhance both the compiler and the run-time envionment to recognize and manage circular references. At compile time, the compiler would recognize circular references and pass the information to the binder and the linker. It could also generate warnings to ensure the programmer is aware of the consequences. At package elaboration prior to beginning execution, the run-time environment would be informed through information passed from the linker which packages have circular references. The run-time environment would detect infinite loops in either data structure references or procedure calls and generate the appropriate errors.

4. PROTECTED TYPE SUBCLASSING

Ada 95 provides some object-oriented features, but does not provide subclassing of all abstract data types. In particular, subclassing is not extended to protected types [1, section 9.4]. An implementation of subclassing could be based on a combination of protected types and tagged

types. Why do this? Some object classes need to have subclasses that further specialize their descriptions. For example, one type hierarchy might include (view, window, editing window). It would be useful to specify a protected type VIEW, which is a tagged type, and then specify a protected type WINDOW, also a tagged type, which inherits both the attributes and the operations of VIEW. The Ada 95 declaration might look like figure 5.

```
---
--- This construct is not allowed in Ada 95!
--- The “tagged” syntax is not allowed
---
protected type View is tagged
   record
         ---
         end record;
         <operations on View>;
end View;

protected body View is
   ---
   --- declarations for the body of the protected type
   ---
end View;

---
--- This construct is not allowed by Ada 95!
---
protected type Window is new View with
         record
         ---
         end record;

         <declaration of operations on Window>;

end Window;

protected body Window is
         ---
         --- specifications for the body of the protected type
         ---
end Window;
```

Figure 5. Sample Tagged Protected Type

```
---
--- THIS IS NOT LEGAL ADA 95!
---
generic
   -- import S which is the general object
   type S is tagged private;

   protected type View is

      <procedure and function declarations>

   private

      --- protected types not allowed to have record
      --- type declarations

      type T is new S with
            record
            --- <declarations for the record>
            end record;

      <operations on T>

   end View;

   protected body VIEW is

      type T is tagged private;
      --- <procedure and function definitions>

   end View;

   --
   -- make a Window be a View plus some
   -- additional functionality
   --
   protected type Window is

      --- <procedure and function declarations>

   private

      ---
      --- protected types not allowed to have record
      --- type declarations
      ---

      type X is new T
         with record
            --- <declarations for the record>
      end record;

      <operations on X>
   end Window;
```

Figure 6. Emulation of a Tagged Protected Type

```
--
-- THIS IS NOT LEGAL ADA 95!!
--
protected type View is

      < data declarations >
      < entry declarations >

private
```

```
    <other entry declarations >

end View;

protected type Window is new View

    < additional entry declarations >

private

    < additional entry declarations >

end Window;
```

Figure 7. Suggested Modification to Protected Type Declarations

The AARM does not allow the definition of a protected type which is also a tagged type. But, we might emulate this construct as shown in figure 6, which extends VIEW and specializes it in WINDOW. The visibility rules for this subclass feature could correspond to those applying to Child Units. However, according to the AARM, the protected type definition takes only a <defining_identifier> in its specification. The data structures for the protected type are defined in a private clause. And, packages cannot be used in the private section of the protected type.

At a minimum, we suggest that protected types should be able to at least support extensions through the addition of new entries that operate upon the data structures contained within the protected type. This might be declared as shown in figure 7.

Using this format, the protected type can be enhanced with additional entries for subprograms, but cannot be extended through the addition of new variables.

5. INTERNAL ACCESS TO ACCEPT STATEMENTS

Many OOPLs seem to allow one method in an object class to invoke another method within the same class because methods are usually implemented as procedure calls. Indeed, some OOPLs allow a method in an class to recursively call itself. Our methodology calls for a direct translation of class methods and their bodies to Ada 95 task types and their Enter/Accept bodies. However, this self-invocation of methods cannot be translated to Ada 95 accept statements.

Ada 95 does not allow an accept statement within a task to invoke another accept statement in the same task instance. Figure 8 depicts this problem.

```
---
--- partial Ada 95 declaration for a task
---
task body <x> is

        <local variable declarations>

begin
   loop
        select
             accept <1> do
                  <accept body statements>
             end <1>;
        or
             <other accept statements>
        or
             accept<k> do
                  <accept body statements>
                  -- NOTE: THIS IS AN ILLEGAL ADA 95
                  -- INVOCATION!
                  <1>(args);
                  <accept body statements>
             end <k>;
        end select;
   end loop;
end <x>;
```

Figure 8. Accept Invocation in a Task

```
---
--- Partial fragment of an Ada task
---
accept move(
   p: in Piece_Handle;
   mode: in Boolean;
   new_positions: out Position_Array;
   nummoves: out Integer) do
  --
  move_internal(p, mode,
             iswhite, position,
             new_positions, nummoves);
end move;
```

Figure 9(a). Accept Statement

```
--
-- Program fragment showing a procedure
--
procedure move_internal(
    p: in Piece_Handle;
    mode: in Boolean;
    isw: in Boolean;
    king_pos: in Pos_String;
```

```
    new_positions: out Position_Array;
    nummoves: out Integer) is
     ••• <remainder of procedure body>
end move_internal;
```

Figure 9(b). Move_internal Code

```
--
-- Program fragment showing a procedure
--
procedure pos_in_check_internal(
     p: in Pos_String;
     isw: in Boolean;
     king_pos: in Pos_String;
     ret: out Boolean) is
              •••
loop
   -- other statements
   case pieces(i).Figure is
        •••
        --
        when 'K' | 'k' =>
          aKing := King_Conversion.To_Pointer(
                       pieces(i).Addr);
          move_internal(
                            pieces(i),
                            for_check_test,
                            isw, king_pos,
                            positions, nummoves);
         ••• other statements
   end pos_in_check_internal;
```

Figure 9(c). Pos_in_check_internal Code

A simple solution is easily implemented. Each class method invokes an internal procedure as its sole method body. This internal procedure performs the computation and returns a result, possibly, which is then returned as the result of the method. When the method is translated into an accept statement in Ada 95, other accept statements invoke the internal procedure without blocking.

Our methodology requires that a self-invocation that is valid in an OOPL (other than Ada 95) be translated into two parts: the externally invocable interface (e.g., the accept statement) and an internal procedure which is declared in the private part of the task type. The translated body of the class method is embedded in the internal procedure. In the accept statement, an invocation to the internal procedure is embedded as shown in figure 9(a). Values returned from the internal procedure are returned by the accept statement if it returns anything at all. Figures 9(a) through 9(c) depict a sample taken from KING.ADB of the Schess program [4].

In the original Sather source code, pos_in_check_internal calls upon the move method of the King piece to determine if an opponent's king is putting my king in check. The pos_in_check_internal code was duplicated in KING.ADB to eliminate a circular reference. But, this introduced the problem of calling an entry point from within the task (e.g., the call to move). Therefore, the body of move had to be extracted from the accept statement and embedded in an internal procedure in order to allow this code to be used from within the task. Note that the internal procedure should eb declare in the private part of the package that includes the move_internal code.

6. CONCLUSIONS AND FURTHER WORK

We believe that some of the problems we encountered in translating object-oriented programs written in Ada 95 to process-oriented programs written in Ada 95 offer some insights into further enhancements and revisions to the Ada 95 program language. These insights may apply to other concurrent programming languages as well. This paper has attempted to highlight three of these difficulties and suggest work-arounds or possible revisions to the Ada 95 programming language.

The problem that had the greatest impact was the inability to perform circular elaboration of packages as described in section 2. Other object-oriented programming languages that we surveyed - including C++, Sather, Eiffel, and Java - permit forward references to classes contained in other compilation units. One compilation unit is allowed to make backward references to another compilation unit which makes a reference to the first compilation unit. The solution to this problem required the introduction of artificial classes - which we believe would be difficult to automate - and the substantial movement of blocks of code from the original compilation units' source code.

Protected type subclassing seems to be a problem merely of omission which, we believe, should be corrected by modifying the syntax and semantics of Ada in a future version of the language. At a minimum, we suggest that protected type subclassing with the ability to declare additional entry subprograms be allowed. This would seem to require only minor modifications to the AARM and current compilers.

The problem of internal access to accept statements arose as a result of applying our methodology to object-oriented Ada 95 programs. A student learning Ada 95 would not normally consider such usage because the (AA)RM does not allow it. However, it is a powerful mechanism that is used frequently in other OOPLs such as Sather and

Common Lisp's CLOS. Little modification to the syntax of Ada 95 is required to correct this problem; but, some modification to the semantics of how a task body is entered and an accept statement is executed will be required.

Ada 95 is a major improvement over Ada 83. While remaining largely upward compatible, it introduces some new language features that make it easier to write object-oriented and concurrent programs. However, as discussed in this paper, the inability to make circular references among packages is a major limitation which can be corrected through better compiler techniques. Solutions to the other two problems, which do require changes to the syntax and the semantics of Ada 95, are harder to make, but would will make it easier to write concurrent programs. We believe that making these modifications to Ada will make it easier to develop some of the concurrent programs that resulted from applying our methodology..

Identification of the three limitations reported here arose from an innovative approach to developing concurrent programs. As we continue our research into our methodology and transformation algorithms for converting object-oriented to process-oriented programs, it is likely that we will find some other limitations of Ada 95 that should be investigated further. We will report on these limitations as they are encountered and we develop potential solutions or work-arounds.

7. ACKNOWLDGEMENTS

Steve Kaisler wishes to acknowledge the assistance of the IIT Research Institute in providing tuition assistance while completing his degree. He also wishes to acknowledge the numerous comments of other members of his dissertation committee: John Sibert, Arnold Meltzer, Massoud Moussavi, and Alan Goldschen. And, he wishes to acknowledge the guidance and numerous contributions of his dissertation advisor, Mike Feldman, without whom this research would not have reached a successful conclusion.

8. REFERENCES

[1] Ada 95 Mapping/Revision Team. 1994. Programming Language Ada: Language and Standard Libraries, Annotated Draft, Version 5.95. November

[2] Ada 9X Mapping/Revision Team. 1994. Rationale for the Programming Language Ada; Intermetrics, Cambridge, MA

[3] Cohen, N.H. 1995. Private Communication via electronic mail

[4] Kaisler, S.H. 1997. Making Concurrency Explicit: Converting Object-Oriented to Process-Oriented Programs. D.Sc. Dissertation, Department of Electrical Engineering and Computer Science, George Washington University, Washington, DC

[5] Meyer, B. 1988. *Object-Oriented Software Construction*. Englewood Cliffs: Prentice-Hall

[6] Meyer, B. 1993. "Systematic Concurrent Object-Oriented Programming". Communications of the ACM, 36(9): 56-80

[7] Omohundro, S.M. 1993. "The Sather Programming Language" Dr. Dobb's Journal, 18(11):42-48

[8] Steele, G.L., Jr. 1990. Common Lisp: The Language, Second Edition. Digital Press, Maynard, MA.

[9] Stroustrop, B. 1986. *The C++ Programming Language*. Addison-Wesley, Reading, MA

9. BIOGRAPHY

Stephen H. Kaisler is currently the Director, System Architecture in the Office of the Sergeant at Arms of the U.S. Senate. He was previously a Science Advisor with the IIT Research Institute in Lanham, MD where he developed information system architecture frameworks and processes for the U.S. Treasury. He completed his doctoral research under the direction of Dr. Michael Feldman at the George Washington University. Dr. Kaisler is also a Adjunct Professor of Engineering in the Department of Electrical Engineering and Computer Science where he teaches undergraduate and graduate computer science courses. He has previously been Chief Scientist for Analytics and a Program Manager at the Defense Advanced Research Projects Agency and the Central intelligence Agency. Dr. Kaisler also holds a B.S. in Physics and an M.S. in Computer Science from the University of Maryland, College Park.

Michael B. Feldman is a Professor in the Department of Electrical Engineering and Computer Science at The George Washington University. He holds the M.S. and Ph.D. in Computer and Information Sciences from the University of Pennsylvania and the B.S. in Electrical Engineering from Princeton University. Prof. Feldman has published widely on concurrent programmng and on undergraduate education, and is the author of several well-received textbooks.