OSU: Integrating CASE and UIMS

Sherry Yang
T.G. Lewis
Chia-Chi Hsieh
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

88-60-13

OSU: Integrating CASE and UIMS

Sherry Yang
T. G. Lewis
Chia-Chi Hsieh

Computer Science Dept.
Oregon State University
Corvallis, OR. 97331
ph (503)-754-3273
CSNet: yang@mist.cs.orst.edu
lewis@mist.cs.orst.edu

ABSTRACT

Oregon Speedcode Universe (OSU) is a software development environment for design, implementation, and maintenance of large software systems. Designed to be highly visual, OSU combines traditional structured analysis techniques found in most CASE tools with advanced graphical user interface management systems (UIMS) found on most contemporary workstations. This paper describes the design and implementation of four unique features of OSU: 1) combination of functional decomposition with object-oriented design, 2) alternate architectural views, e.g. call graph, uses graph, object graph, and graphical display of procedures, 3) program understanding tools for design and maintenance, and 4) merging the user interface specification with design and coding specifications.

KEYWORDS

CASE, prototyping, object-oriented design, structured analysis, hierarchical decomposition, visual programming, graphical user interface, user interface management, software specification, software design.

CASE and UIMS in OSU

OSU (Oregon Speedcode Universe) is an experimental programming system designed for Macintosh programmers who want to very rapidly produce applications. The aim of OSU is to realize a 100 to 1000-fold increase in programmer productivity by combining rapid prototyping, reusable units, program generation, and expert systems technology under an integrated environment we call "speedcode universe".

OSU combines a UIMS (User Interface Management System) with a structured design facility which allows a programmer to quickly prototype the user interface of a given application and then connect that interface to program design tools traditionally found in most CASE (Computer-Aided Software Engineering) systems. UIMS allows a programmer to create and directly manipulate icons, menus, windows, dialogs, alerts, and user-defined procedures. UIMS consists of three parts: 1) a resource editor called RezDez (Resource Designer) for WYSIWYG creation and editing of icons, menus, windows, dialogs, and alerts; 2) a WYSIWYG "sequencer" for "training" the application to behave the way it should when the application is used by an end user; and 3) a program generator that writes a Pascal source code program for implementing the behavior specified by "sequencing". For a complete discussion of UIMS and rapid prototyping in OSU, see [Lewis 88 ,Bose 88, Handloser 88]. The purpose of this paper is to describe the integration of UIMS with CASE tools, and to show how user interface specifications can be combined with functional specifications to arrive at a total application specification.

How OSU Works

CASE can be combined with UIMS by integrating the user interface specifications with the functional specifications as follows. All user interface objects are designed and stored along with the "sequence" information needed to make the user interface "simulate" the final application. These objects and their sequence information are called a *vacuous prototype* in OSU because they specify the user interface, but none of the application's functionality. In the process of sequencing through the user interface, functionality of the system can be added in the form of: 1) new code produced by hand; 2) a reusable component taken from a library of reusable modules; or 3) modules which are automatically produced by one of several software accelerators [Lewis 88]. If produced by hand, OSU provides CASE-like tools for structured design. If selected from an existing library of reusable components or software accelerator, the CASE tools of OSU can be used to display the modular structure of the reusable component. In addition, OSU provides tools for program comprehension and understanding. The contribution of this paper is to show how UIMS can be combined with CASE to render a number of different, and useful, software architectural points of view.

ARCHITECTURAL POINTS OF VIEW

Structure Chart View

Structure charts are a form of functional decomposition. The basic building block of a program is a module, and structured programs are organized as a hierarchy of modules. The structure chart is a tree or hierarchical diagram that defines the overall architecture of a program through its *call graph* [DeMarco 78].

It is important to distinguish between a program module and a program function/procedure. A module is a separately compiled unit of code consisting of function/procedure definitions along with interface specifications. In OSU, the separate compiland called a Macintosh Pascal *unit* is used to form modules. Every unit consists of an interface part and an implementation part, very similar to modules in Modula-2, and packages in Ada™.

Modules are not procedures, and because a structure chart is a call graph showing the interconnections among procedure/functions, the structure chart is only one of several architectural points of view. We can, for example, render a program in terms of other points of view, such as its *uses relation*, which consists of a graph showing what units are used by each module in the program or an object-oriented rendering showing both call graph and uses relations. We will describe these other architectural points of view, later.

In OSU, rectangular boxes represent procedure/functions and arcs connecting the boxes represent invocations or calls to each procedure/function. The unit name and procedure/function name is displayed in each box to allow combination of functional decomposition and object-oriented design methodologies. (Each unit is treated as an object in the object-oriented view of the system -- more on object-oriented design later).

The structure chart in Figure 1 shows a call graph for a file input routine. All the file related routines are grouped in a unit called FILE. Macintosh provides some standard file routines for opening, reading and writing files. All of the Macintosh ROM-based routines are collectively called the TOOLBOX routines. ErrorCheck is a reusable routine from the GrabBag reusable library. DIA_showAlert is a general alert routine in the DIALOG unit, which is called in case of a file open error.

Modules are interrelated by a control structure. The structure chart shows the interrelationships by arranging modules in levels and connecting the levels by arcs. An arc drawn between two modules at successive levels means that at execution time program control is passed from one module to another in the top-down direction. Data and control transfers between modules is usually in the form of parameter passing. Conventional methods of displaying the structure chart draws the parameters next to the arc that connects the two modules, but for the purpose of readablity, OSU's structure chart editor hides the details of the parameters. The programmer must double-click the arc to display the parameters as shown in Figure 2.

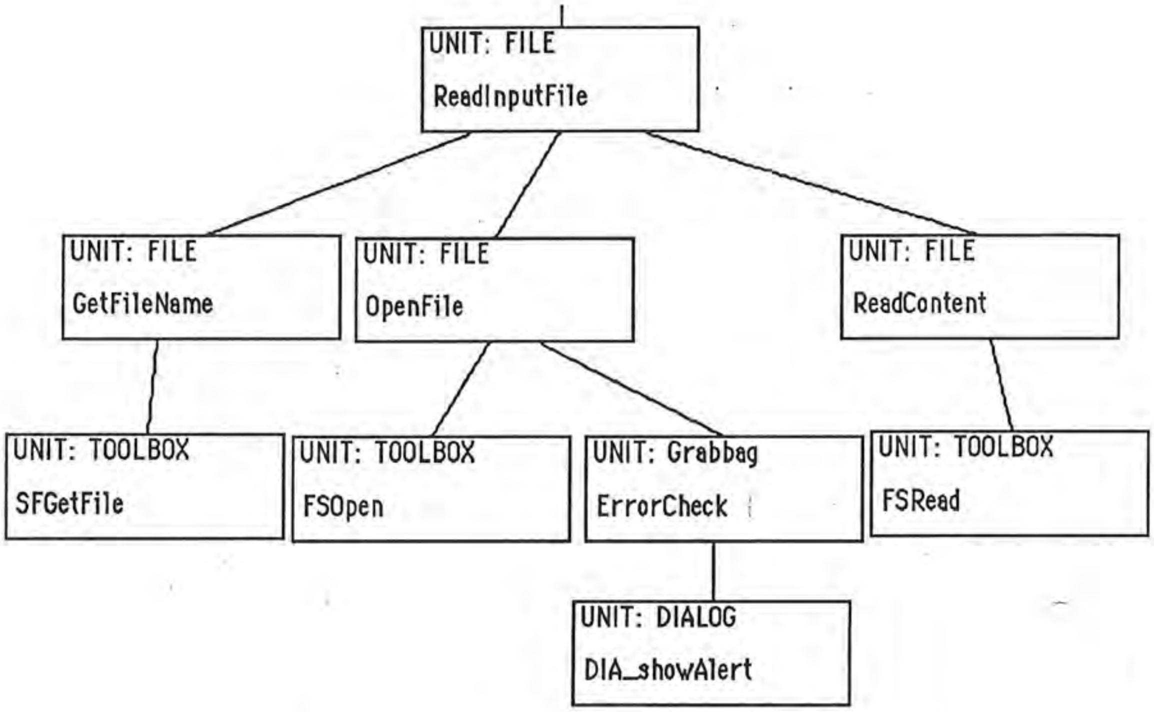
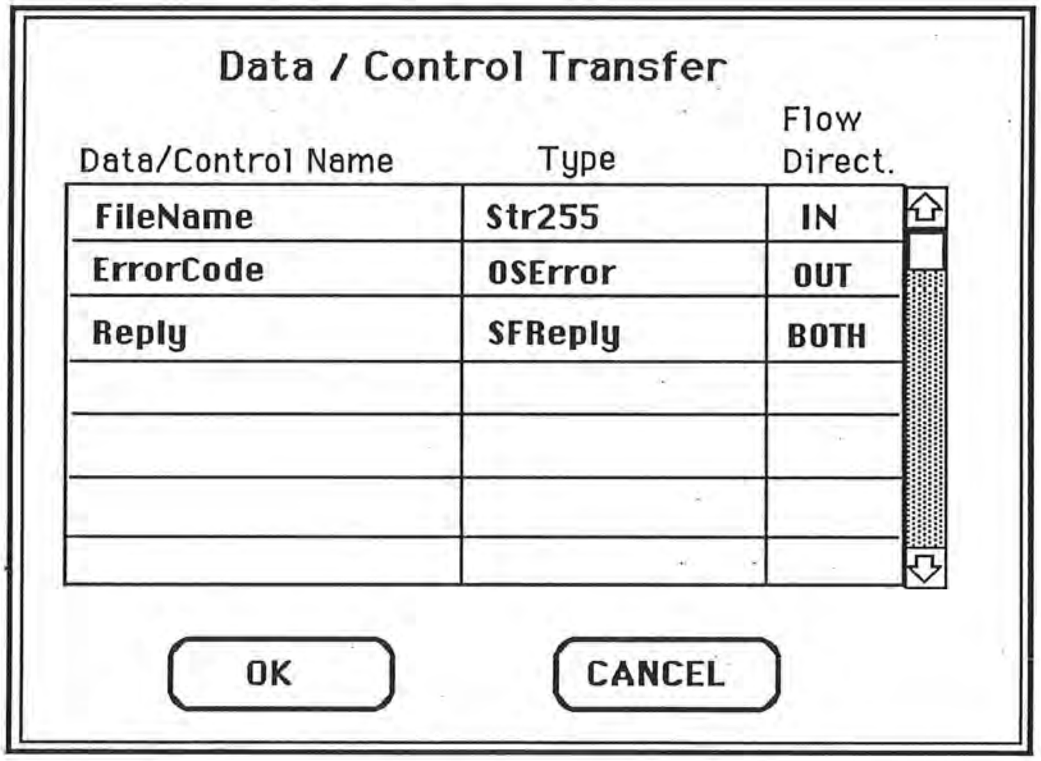


Figure 1. A sample structure chart created by OSU.



Data/Control Name	Type	Flow Direct.
FileName	Str255	IN
ErrorCode	OSError	OUT
Reply	SFReply	BOTH

Figure 2. A dialog for displaying data/control tranfers between modules.

Figure 2 shows the dialog obtained by double-clicking an arc in the structure chart. This dialog displays parameter information in the form of data and control flow in and out of the function/procedure. In addition, the interface may be altered -- there is a scrollable list that allows the programmer to enter and edit all data names. For each data item, the name, data type and flow direction is required. **IN** flow means the information is coming into the module. **OUT** flow means the information is returned from the module. **BOTH** means the information is both coming in from and passed back to the calling module. In generating PASCAL source code, **IN** data are the value parameters, **OUT** data are the variable parameters that do not need to be initialized upon calling the procedure, and **BOTH** data are the variable parameters that need to be initialized before calling the particular procedure.

Object-Oriented View

Object-oriented development is an approach to software design in which the decomposition of a system is based on the concept of an object [Booch 86]. An object is an abstract data type entity with the ability to inherit properties from classes of other objects. An object has state and function -- state in the form of data, and function in the form of function/procedures. In Macintosh Pascal, an object is defined as a unit. Units cannot inherit functions from other units, so our approach is not pure. Instead, units are used to encapsulate state in the form of constants, types, and variables, and function in the form of functions and procedures. An object hierarchy is established as a uses graph -- one more architectural point of view of interest to us.

Syntactically, Pascal units are connected by the "uses" clause which is a mechanism for import/export of constants, types, variables, procedures, and functions that are visible from outside of a unit. Thus modules are connected via their interface parts and access procedure invocations.

Rather than factoring the system into modules that denote operations, we structure the system around its objects, or units. Each object is represented as a rectangular box with its name at the top, and all operations defined on the object are listed within the rectangle. Interconnections between objects are shown as arcs and represent function invocation just as in the structure chart view. Objects are arranged hierarchically based on their uses relations.

Figure 3 shows the object-oriented representation of the ReadInputFile procedure of Figure 1. There are 4 units, FILE, TOOLBOX, GRABBAG, and DIALOG. Units are arranged in a hierarchical fashion. TOOLBOX and GRABBAG units are directly called from the FILE unit, so they are arranged 1 identation from the FILE unit. The DIALOG unit is called from the GRABBAG unit, so it is arranged 1 identation from the GRABBAG unit. The visible procedure/function names of each unit are displayed in the small rectangle inside each unit.

This representation differs from other proposals for displaying object-oriented designs [Booch 86]. In particular, our representation does not distinguish between an object and its class. In object-oriented terminology, our model does not show instantiated classes, but instead, shows only the concrete objects actually used. This is partly a result of weaknesses in Pascal, and partly our desire to simplify the representation.

Why is the object-oriented view important? According to [Booch 86], the object-oriented approach to design mitigates weaknesses in functional decomposition such as lack of data abstraction and information hiding. Object-oriented design is generally thought to be better for problem domains with natural concurrency, and is more responsive to changes in the problem space.

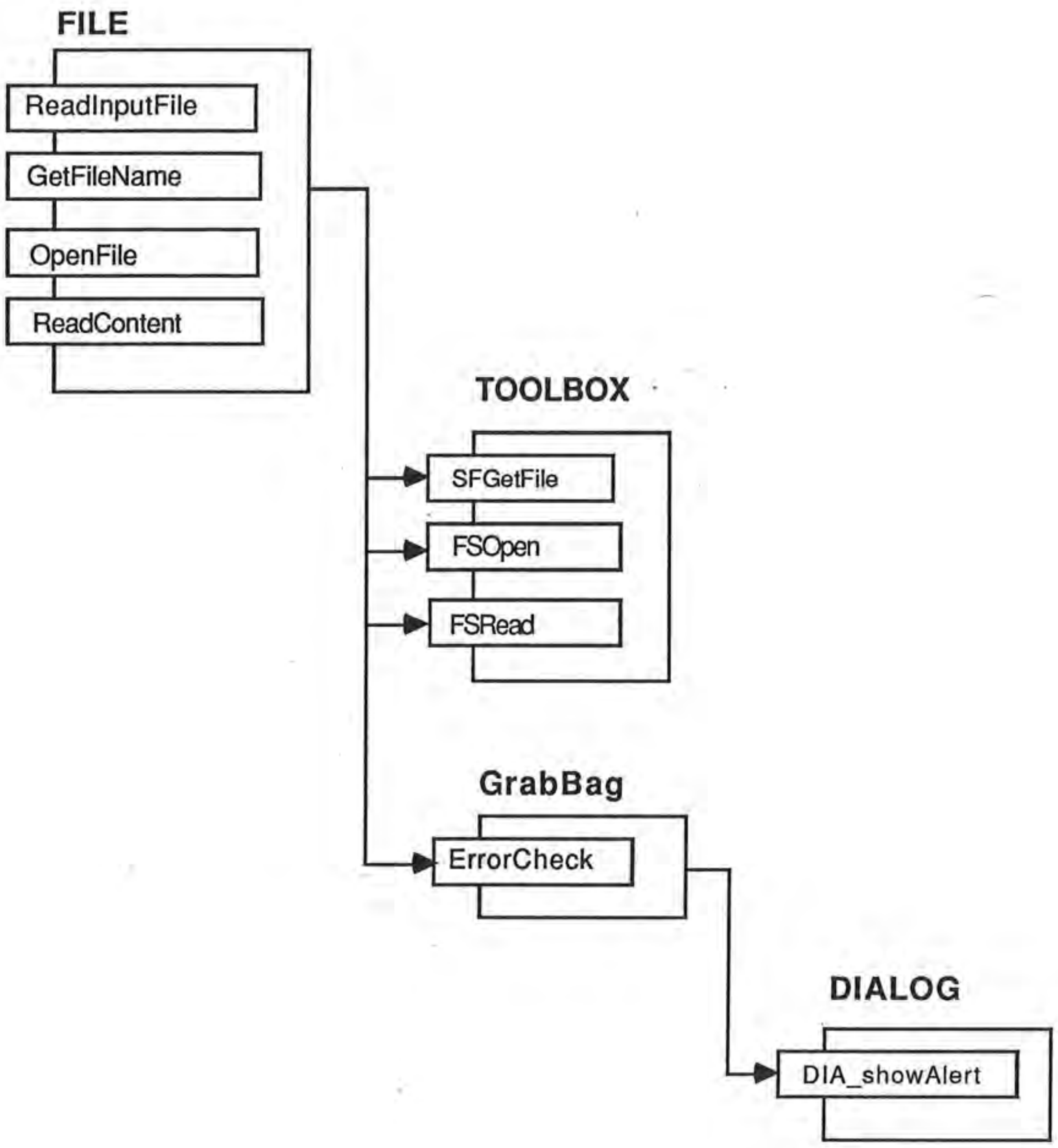


Figure 3. An object-oriented view of Figure 1.

Uses Relation Graph

The problem with object-oriented design is coming up with an object-oriented rendition of the entire system. Unlike other forms of representation, object-oriented designs do not incorporate hierarchical strucutre. To reduce the clutter of an un-leveled object-oriented view, a simplified uses graph can be generated as shown in Figure 4. The uses graph supresses the connections in the system stemming from calls and shows only the import/export properties of the objects.

The graph in Figure 4 shows only the *uses relation* among units. Again the units are arranged in an hierarchical fashion based on their uses order.

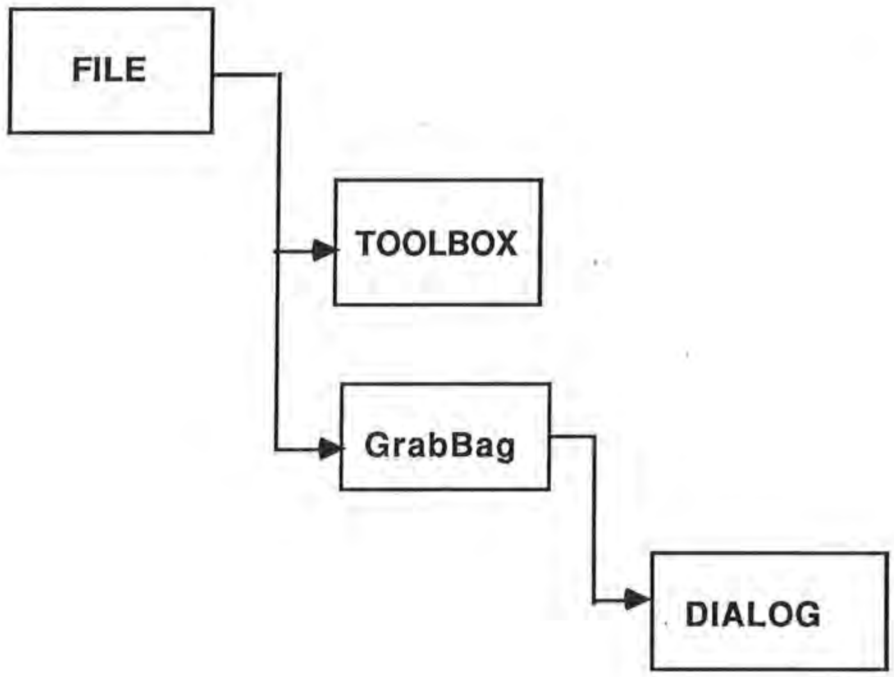
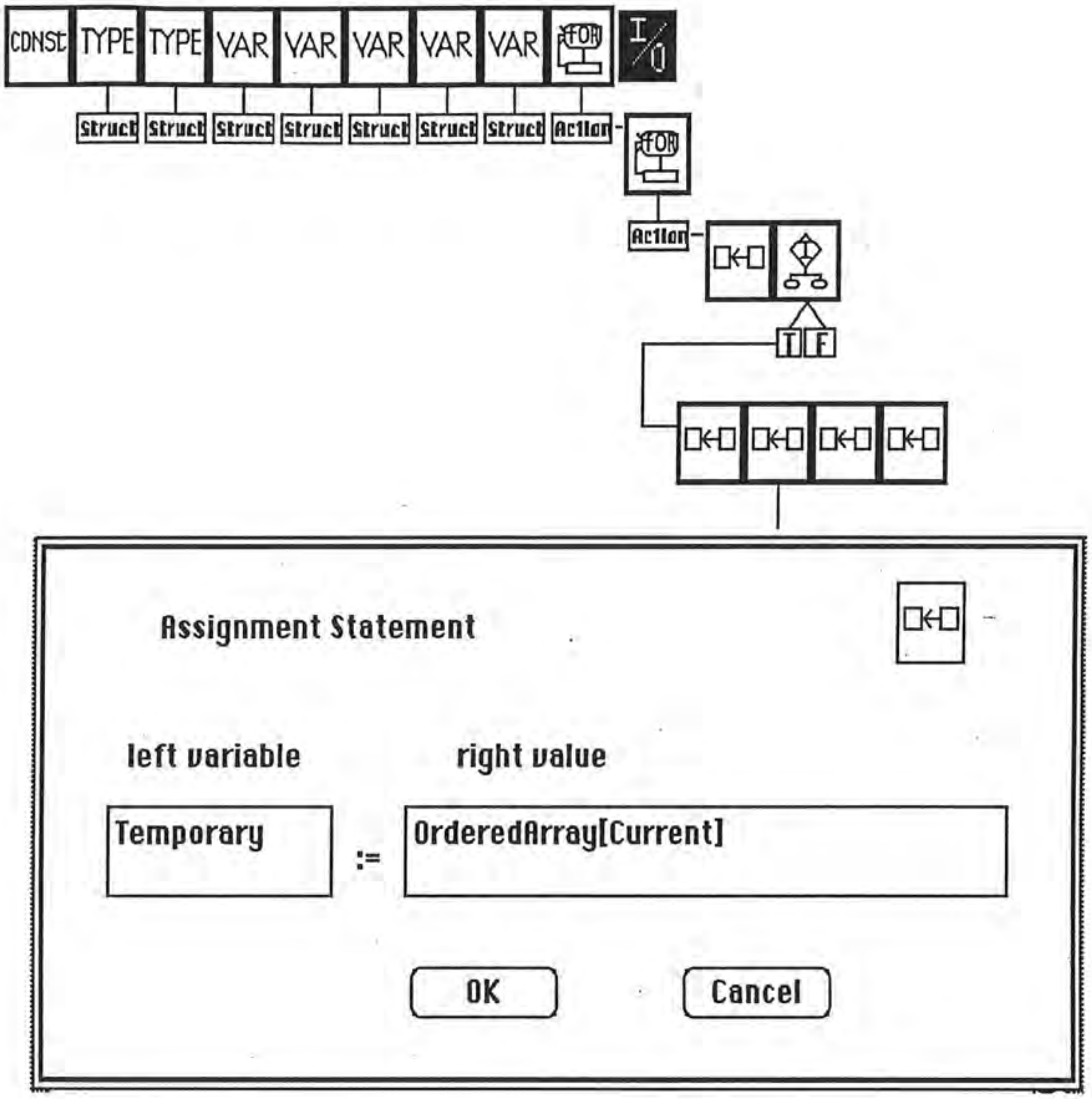


Figure 4. Unit Uses Graph for Figure 3.



Assignment Statement

left variable

right value

Temporary := OrderedArray[Current]

Legend:

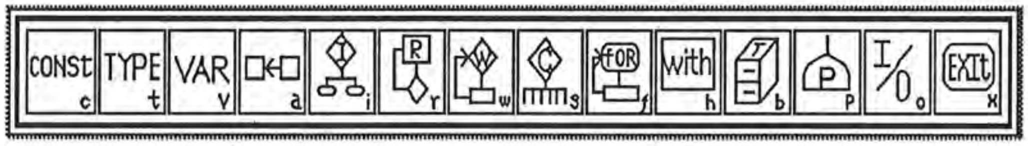


Figure 5. Hierarchical Detailed Design of BubbleSort Procedure

Detailed Design Editor

A major drawback of the structure chart method as a design representation is that control structures such as repetition, sequence, and alternation are not easily represented. Instead they are regarded as details which are shown by a different technique, such as pseudocode, flowcharts, etc. Such details can be rendered by a *detailed design tool.*

Detailed designs are visually constructed in OSU using a Plum Diagram Editor, which is a graphical tool for editing Pascal source code [Hsieh 88]. A new procedure or function can be created by either reading an existing procedure of function from a reusable unit and modifying it, or by creating an entirely new unit. In either case, the programmer designs the procedure from visual building blocks.

Figure 5 illustrates the Plum Diagram point of view for BubbleSort. The legend at the bottom of the figure lists all visual building blocks -- from data types to control structures. These blocks are dragged next to each other to form a sequence. Each hierarchical structure has associated with it, a "next-level" icon which hides the details below it in the hierarchy. The rendering of Figure 5 is a "flattened" or "leveled" view of this hierarchy where the lower levels are shown immediately to the right of each higher level construction. (We have shown only one of the many decompositions, here).

For example, the FOR-loop at the top of the hierarchy is decomposed into a nested FOR-loop at level two. The level two FOR-loop is itself decomposed into a sequence consisiting of an assignment statement and an IF-branch. The IF-branch is decomposed into TRUE and FALSE clauses, and the TRUE clause is shown here expanded to a sequence of four assignments. Finally, one of the assignments is shown expanded to a "primitive" consisting of the left-hand and right-hand sides.

Pascal source code is automatically generated from a Plum Diagram design. The code generated by the BubbleSort design shown in Figure 5 is listed in Figure 6. It can be inserted into a module and compiled for use by OSU. Each box in the top-most level corresponds to a tree in the hierarchy. To see what is in the tree, open each box, etc., until the lowest level is reached. Actual source code is revealed at the lowest level as shown in Figure 5. Notice the correspondence between the detailed design diagram and the actual source code in Figure 6.

The Plum Diagram Editor serves both as an understanding tool and a maintenance tool through two techniques: Program slicing and complexity metrics.

Program Understanding Metrics

Program understandability and program maintainablility are parallel concepts: the more difficult a program is to understand, the more difficult it is to maintain [Berns 84]. Attempts have been made to quantify program difficulty by manipulating simple counts of selected program attributes, e.g. lines of code, operations, operands, etc. The Plum Diagram Editor performs a number of these counts, but we make no claims as to the validity of these counts.

```
PROCEDURE BubbleSort;
  CONST
    ArrayLimit = 26;
  TYPE
    CharData = RECORD
          TheCharacter : char;
          Count : integer;
       END;
    RecordArray = ARRAY[1..ArrayLimit] OF CharData;
  VAR
    Last : 2..ArrayLimit;
    Current : 1..ArrayLimit;
    Comparisons, Switches : integer;
    OrderedArray : RecordArray;
    Temporary : CharData;
BEGIN
  FOR Last := ArrayLimit DOWNTO 2 DO
    BEGIN
       FOR Current := 1 TO Last - 1 DO
          BEGIN
          Comparisons := Comparisons + 1;
          IF OrderedArray[Current].Count < OrderedArray[Current + 1].Count THEN
             BEGIN
                   Switches := Switches + 1;
                   Temporary := OrderedArray[Current];
                   OrderedArray[Current] := OrderedArray[Current + 1];
                   OrderedArray[Current + 1] := Temporary;
             END
          ELSE
             BEGIN
             END;{else}
       END;{for}
    END;{for}
  writeln(Comparisons : 2, ' comparisons, ', Switches : 2, ' switches.');
END;{procedure}
```

Figure 6. Pascal Program Generated From Detailed Design Diagram

The Plum Diagram Editor can compute the Barry-Meekings [Harrison 86], Halstead [Kearney 86], Weight of Difficulty [Berns 84], and McCabe [Redish 86] complexity metrics as well as compute a variety of counts such as the number of symbolic constants, number of comments, and percentage of indentation tabs. Figure 7 shows a sample report with various complexity measurements:

– Weight of Difficulty [Berns 84] assumes that program difficulty can be measured by summing the difficulties of its constituent elements, and these elements can be quantified by the use of selected weights and factors. In this approach, weights are assigned to various elements of the program: constants, operators, symbolic names, statement types, etc. A program is analyzed line by line and element by element to determine its difficulty by assigning weights. Berns reports results for FORTRAN programs and states that "...good scores are now considered to be less than 1,200". In our application of this technique, we permit the programmer to assign weights to each feature found in Pascal. The significance of the resulting score is not known.

– Halstead measurements [Kearney 86] are derived from the following formula:

Volume (V) $= (N_1 + N_2) Log_2 (n_1 + n_2)$

Program Difficulty (D) $= n_1 * N_2 / 2n_2$

Effort (E) $= D * V$ *where* n_1 *- the number of unique operators*

n_2 *- the number of unique operands*

N_1 *- the total number of operators*

N_2 *- the total number of operands*

Halstead's metrics have been studied for many years and their validity disputed by many researchers. We make no claims as to the significance of the volume, difficulty, and effort metrics.

– Variable, comments lines, and McCabe's total IF counts. See [Redish 86]. These counts, like the others, are subjective, and we make no claims as to their validity. It is interesting, however, to examine the various metrics and the code that produced them.

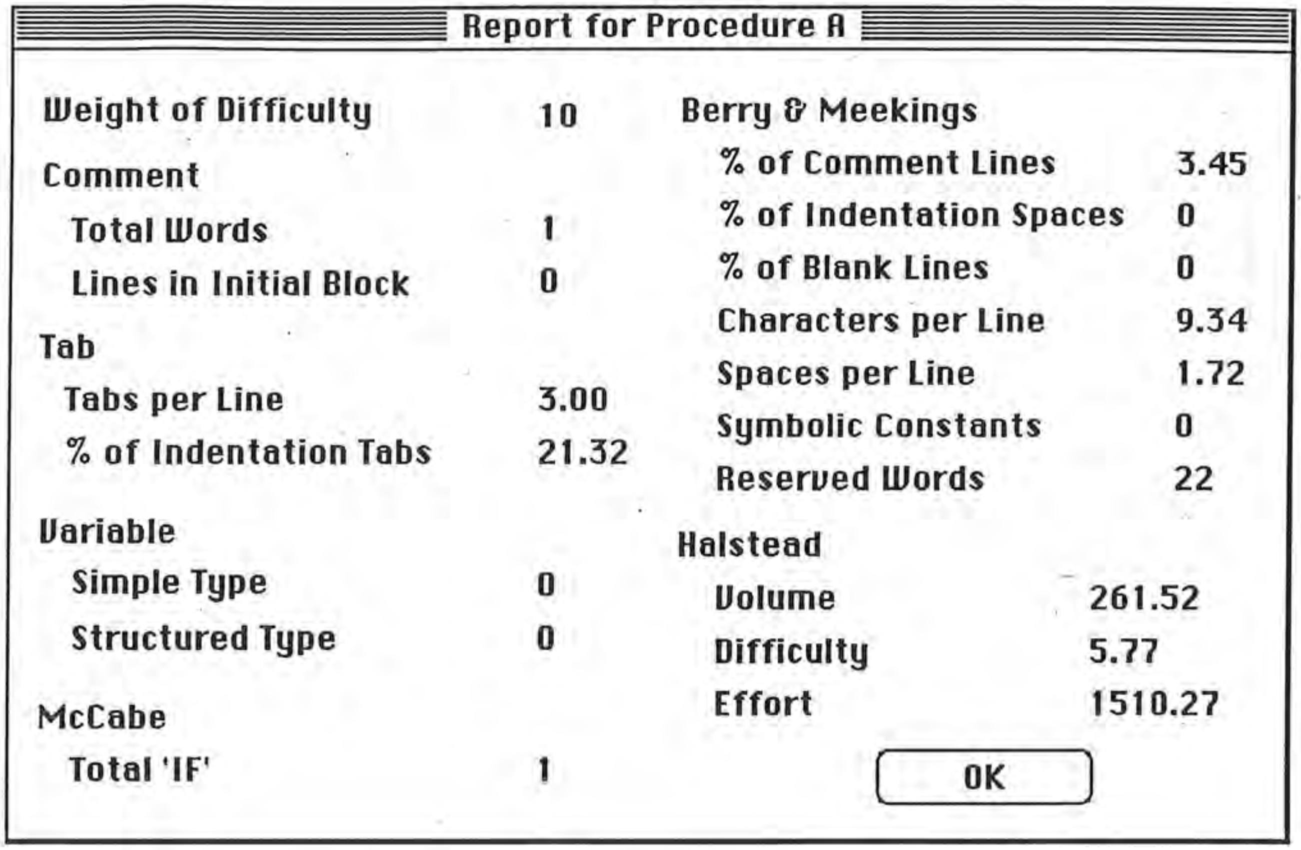
– Percentage of comment lines, percentage of indentation spaces to all characters, percentage of blank lines, average number of non-blank characters per line, average number of spaces per line, percentage of symbolic constants used, and number of reserved words used are obviously of interest to a programmer and software manager because they summarize the documentation quality of a program. Harrison and Cook [Harrison 85] have studied the relationship between the Berry - Meekings style metrics and programming style. While controversial, it appears that some conclusions can be made about the "quality" of a program and these counts. Again, this is an area which needs more research before conclusive results can be stated.

Program Slicing

One of the most interesting features of the Plum Diagram Editor is its ability to emphasize parts of a procedure while de-emphasizing other parts. This is done through a modified form of *program slicing*, which is a method of abstracting programs.

A large computer program is more easily constructed, understood and maintained when broken into smaller pieces called slices [Weiser 81]. The Plum Diagram Editor displays a sliced program in two ways; 1) by hierarchical arrangement of nested constructs; and 2) by selective constructs. Hierarchical slicing is shown in Figure 5. Selective slicing is shown in Figure 8.

The hierarchically sliced BubbleSort program of Figure 5 is displayed in Figure 8 after selectively slicing into parts containing only **FOR** and **IF/Then/Else** constructs. All other parts of the program are shaded to indicate that they are to be ignored. Other methods of slicing produce different results. For example, slicing on a certain variable quickly gives a picture of dataflow through the procedure or function. OSU can slice on any variable and control construct found in any Pascal source program.



Report for Procedure A

Weight of Difficulty	10	Berry & Meekings	
Comment		% of Comment Lines	3.45
Total Words	1	% of Indentation Spaces	0
Lines in Initial Block	0	% of Blank Lines	0
Tab		Characters per Line	9.34
Tabs per Line	3.00	Spaces per Line	1.72
% of Indentation Tabs	21.32	Symbolic Constants	0
		Reserved Words	22
Variable		Halstead	
Simple Type	0	Volume	261.52
Structured Type	0	Difficulty	5.77
McCabe		Effort	1510.27
Total 'IF'	1		

OK

Figure 7. A Sample Report of Complexity Metric

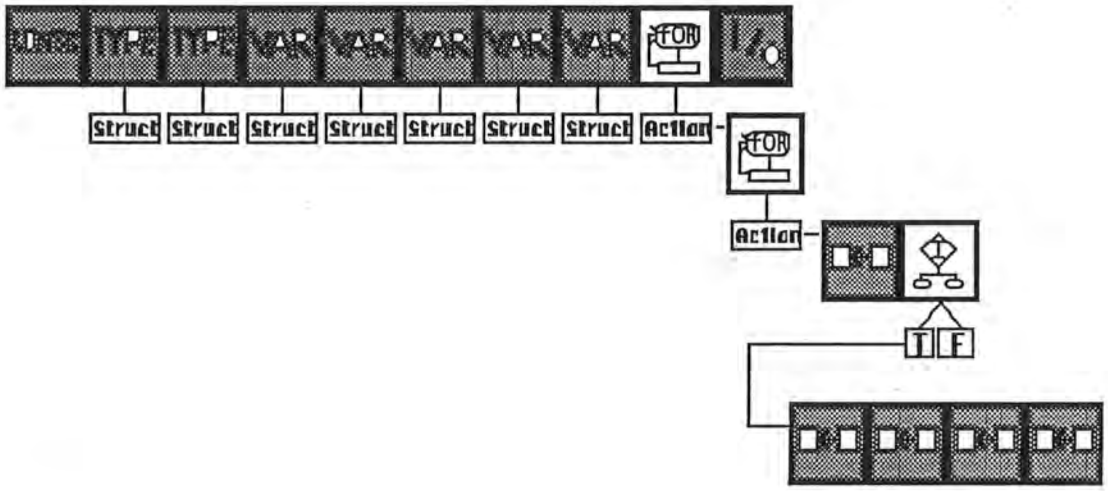


Figure 8. Sliced Plum Chart of BubbleSort Showing FOR and IF/THEN/ELSE

IMPLEMENTATION OF ARCHITECTURAL POINTS OF VIEW

Structure Chart Editor

Functional decomposition is the most familiar form of structured programming to most programmers and so it is the focal point of CASE activity in OSU. A Structure Chart Editor is integrated into the user interface management system to simplify switching from one to the other.

The structure chart is stored as a tree structure in memory. Each node of the tree contains the following information:

-- Unit name.

-- Routine name.

-- Routine Type (New piece of code, reusable piece or toolbox routine).

-- Parameter information in a linked list.

-- Graphical information (Position of the box on the screen, Current selected flag,etc.).

-- Filename of the plum diagram file. The plum diagram gives detail structure of the node.

-- Sibling pointer points to the node that is on the same level of call hierarchy.

-- Child pointer points to the node that is on the next level of call hierarchy.

The Editor is designed to be as visual and intuitive as possible. Most "editing" is done by direct manipulation of graphical icons in a drawing window as shown in Figure 9. Some graphical features provided by the Editor are:

-- Drag/Re-position a node or a sub-tree.

-- Create a new box by dragging from the bottom of any box -- an arc and new box will be drawn automatically.

-- Reduce to fit feature allows the programmer to view the whole chart in reduced form.

-- Scrollable window allows the programmer to scroll both horizontally and vertically.

-- Import/Export tree structure as ASCII text format.

Once in the Structure Chart Editor, the programmer is asked to identify the unit name and routine name of the root node. This is the root node of a particular sub-tree of the system. The programmer is also asked to choose which type of routine it is -- a new routine that the programmer produces by hand, a reusable component taken from some library or other part of the application under construction, or a **Toolbox** routine. (Toolbox routines are stored in the Macintosh ROM).

The Structure Chart Editor is invoked from the graphical sequencer of OSU, at a point where a new procedure needs to be inserted in the application. As shown in Figure 10, the programmer selects **Do a Procedure**, then control of the system is transfered from graphical sequencer to the structure chart editor. A structure chart window comes up along with the structure chart menu. The programmer is asked to define the unit name, routine name, and the type of the root node of the structure chart.

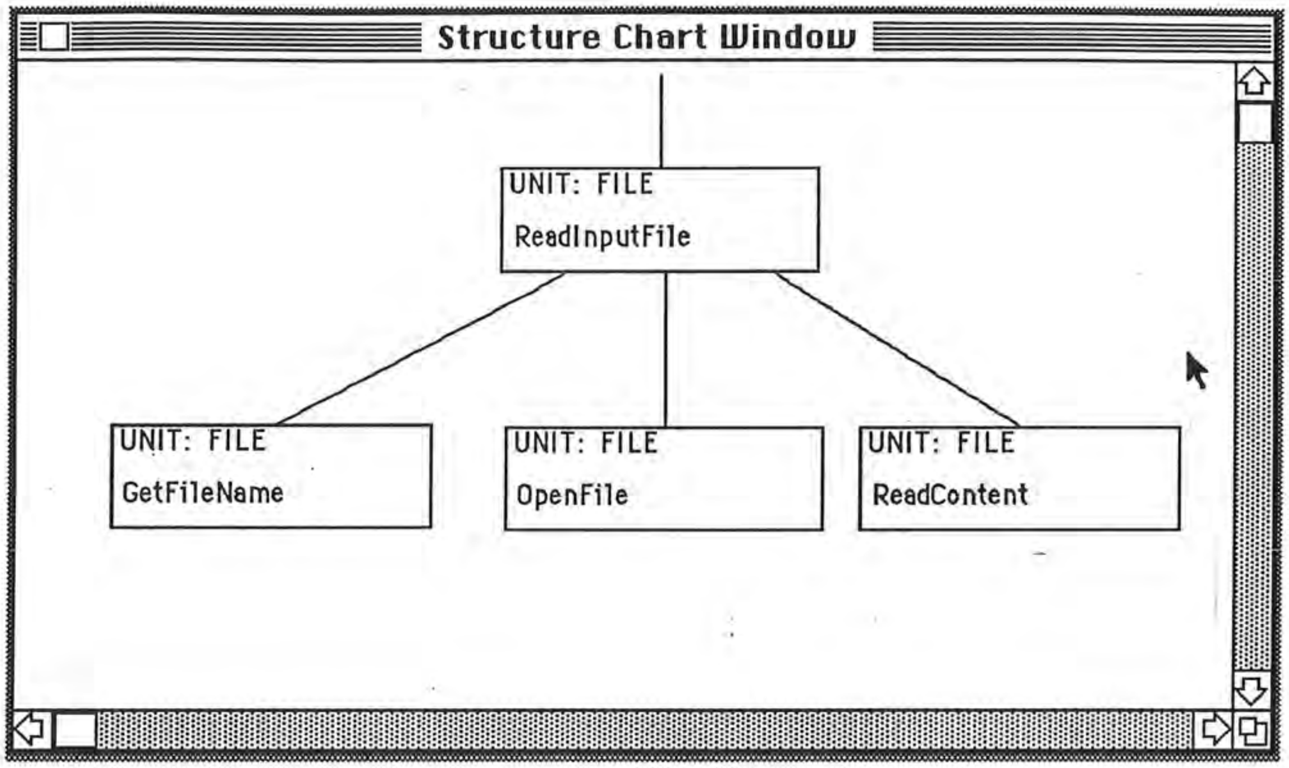
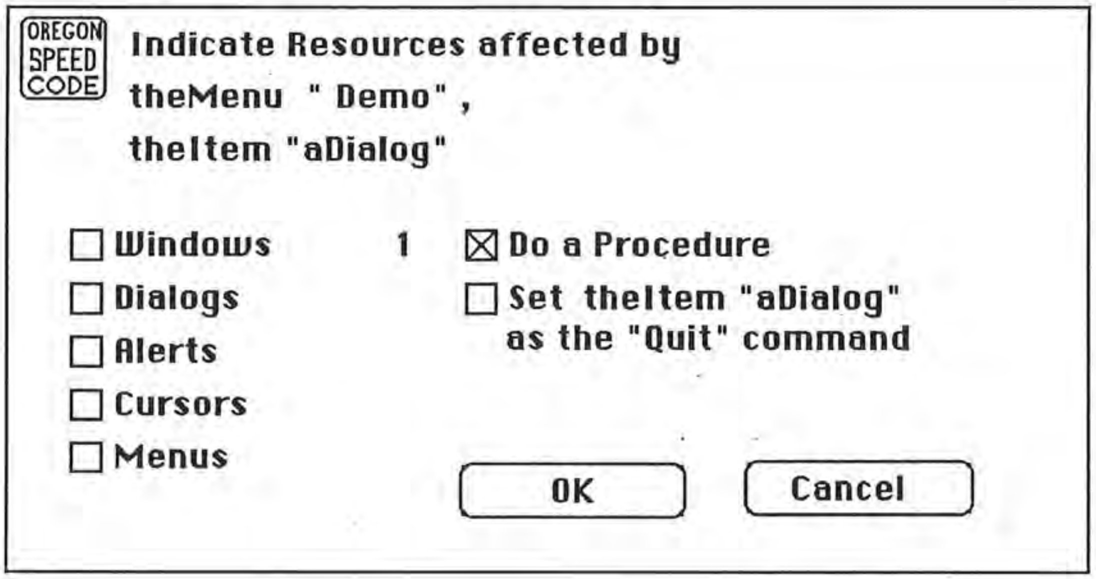


Figure 9. Structure Chart Editor Window.



Indicate Resources affected by
theMenu " Demo" ,
theItem "aDialog"

- [ ] Windows
- [ ] Dialogs
- [ ] Alerts
- [ ] Cursors
- [ ] Menus

1

- [✓] Do a Procedure
- [ ] Set theItem "aDialog" as the "Quit" command

Figure 10. Selecting of DO Procedure From the Graphical Sequencer

The newly inserted procedure/function is called an *active object* in OSU, and is added to the sequence in the form of a **DO** routine. The sequence of user interface interactions and active objects is recorded and written in a *sequence language script*. When the Pascal source code is generated from the recorded sequence language script, a place-holder is inserted for each of the programmer-defined active objects.

At any time when the programmer wishes to view the entire modular structure of the whole system, the Structure Chart Editor reads the current sequence language file to pick up all the DO routines and combines them into a complete structure chart. The various architectural points of view can be re-constructed and displayed from these interactions and scripts.

Structure charts are tree structures, so the only way to add an additional box is to add an arc to it as well. Thus, recursion and multiple instances are represented by duplicates in the tree. Also, arcs don't have to contain data or control information, so it is perfectly acceptable to call a routine without parameters. The Editor allows for other operations such as deletion. For example, deletion is done by first selecting a box to be removed, then selecting the DELETE opinion from the EDIT menu.

Structure charts may be saved and re-loaded into the editor. There are two different kinds of saves. A complete save will save the entire chart along with its graphical information, such as coordinates on the screen, etc. A partial save, which is useful for exporting just the modular structure, will save only the unit name, routine name and parameters in an hierarchical ASCII text format, see Figure 11.

```
FILE: ReadInputFile
PARAMETERS:  <FileName: str255:OUT> <ErrorCode: OSError:OUT> <Reply: SFReply:OUT>
CALLS:
    FILE:GetFileName
    FILE: OpenFile
    FILE: ReadContent

FILE: GetFileName
PARAMETERS: <FileName: str255:OUT>
CALLS:
    TOOLBOX: SFGetFile

TOOLBOX: SFGetFile
PARAMETERS: <FileName: str255: OUT>
CALLS:
    <NONE>
```

Figure 11. ASCII Text Description of a Call Structure

Object-oriented Design Charts

A structure chart can be viewed as an object-oriented design (OOD) chart by choosing the **Display OOD Chart** choice under the strucutre chart feature menu. OOD charts can be generated either for the current structure chart or for the whole system using the information in the current sequence language script file.

To construct the OOD or uses graph, we traverse the structure chart tree in memory, and combine the call graph and uses graph information into a single OOD tree structure. The OOD tree consists of a tree equivalent of the uses graph, plus a linked list of procedure/functions belonging to each unit. When drawing the uses graph on the screen, this tree is scanned in breadth-first order. When constructing the OOD chart the OOD tree data structure is scanned in breadth-first order -- the list of procedure/functions in each unit are displayed along with each unit.

A lay-out algorithm is applied to the internal tree strucuture of unit **uses** relationships to obtain the view shown in Figure 3. This lay-out algorithm properly draws a labeled box on the screen for each unit and draws all the necessary arcs connecting the units. In addition, each procedure/function in the cluster is shown as an internal function defined on the object (unit = object). The lay-out algorithm indents each subordinate unit to show hierarchy -- the uses portion of the OOD chart, and draws connecting arcs to show messages -- the call graph portion of the OOD chart.

If the programmer chooses to generate an OOD chart for the whole system, the structure chart editor reads the current sequence language script file to pick out all the **DO** routines. Corresponding to each **DO** routine is a strucutre chart, so all of the **DO** strucutre charts are combined into one giant structure chart. Once the system structure chart is obtained, OSU can apply the same procedure to obtain an OOD chart for the whole system.

Uses Relation

The programmer may view uses relations without all the routine names. Uses relation charts can be generated either for the current structure chart or for the whole system. The same algorithm is used to extract the units from the structure charts, but a simplified lay-out algorithm is used to draw only the unit boxes and connecting arcs without all the routine names. As in the case of generating an OOD chart, if the programmer chooses to generate the uses relation for the system, all the **DO** routines from the current sequence language script file must be combined to form one strucutre chart before applying the unit analysis algorithm.

CONCLUSIONS

The major feature that makes this system different from existing graphical editors is integration: 1). functional decomposition with object-oriented design, 2). modular system design with detailed design, 3). CASE tools with UIMS, and 4). the ability to generate Pascal source code automatically from these renditions.

Once the structure chart information is captured, different architectual views can be generated by the appropriate menu options:

-- Display *structure chart* for the current structure chart or for the whole system.

-- Display *OOD* (Object-Oriented Design) *chart* for the current structure chart or for the whole system.

-- Display *uses graph* for the current structure chart or for the whole system.

_ Display detail design of a selected procedure as a *plum chart.*

Most existing structure chart editors are merely drawing tools for creating structure charts. For example, ICONIX Software Engineering Inc.'s SmartChart [Rosenberg 85] allows the programmer to enter a text description of the call structure. Then, a second tool called PowerPDL is used to compile this text description to produce the structure chart automatically. But the SmartChart editor doesn't allow direct manipulation of the resulting structure chart. To make changes, a programmer must modify the text description and recompile it with PowerPDL.

STRADIS/DRAW from MCAUTO [Martin 85] and EXCELLERATOR [Martin 85] from InTech are interactive graphic tools for creating, editing and manipulating structure charts. Though adequate for specifying functional design of the system, all three of these tools lack the ability to: 1) automatically generate code and 2) link the user interface design with the functional design.

According to [Myers 89] functionality of a system is rarely handled by a UIMS, especially in direct manipulation interface systems such as OSU. Some systems that approach the completeness of OSU are RAPID/USE [Wasserman 82] and Peridot[Myer 87a, Myer 87b]. These systems perform various functions similar to OSU but are incomplete.

RAPID (RApid Prototypes of Interactive Dialogues)/USE uses a state transition diagram to specify the interactions between the system and the user. In RAPID's view, an interactive program is a set of transactions between a user and the program. Each transaction consists of a set of messages transmitted between a user and the program. The desired functionality is added to the system by adding the name or number of certain actions to the arcs connecting the nodes in the state transition diagram. The drawback of this approach is that there is no facility for editing/adding new routines -- only those actions that are in the system as reusable routines can be added to the arcs of the transition diagram.

Peridot (Programming by Example for Real-time Interface Design Obviating Typing) allows a programmer to design new interfaces by direct manipulation of rectangles, circles, text and lines. Peridot is able to produce executable code from "showing" sequences much like OSU. But similarity stops there, as Peridot does not go beyond the user interface manipulation step -- functionality of the system is not handled.

Although developed as part of OSU, the Structure Chart Editor can be run as a stand-alone tool that is useful both as a design tool and a maintenance aid. The combination of functional decomposition and object-oriented design allows programmers from both "schools" of design to utilize the tool. Import/exporting of ASCII text description gives flexiblity to the system. Combined with UIMS, detailed design tool, and automatic code generator makes this a complete toolset essential for program development.

REFERENCES

[Berns 84] Berns, G.M., **Assessing Software Maintainability**, Communications of the ACM, vol. 27, no. 1, (Jan. 84), pp. 14-23.

[Booch 86] Booch, G., **Object-Oriented Development**, IEEE Trans. on Software Engineering, Vol. SE-12, No.2 (Feb. 1986) pp. 211 - 221.

[Bose 88] Bose, S. RezDez, **A Graphical Tool for Designing Resources in OSU**. Dept. Computer Science Technical Report 88-60-2, Oregon State University, OR. 97331.

[DeMarco 78] DeMarco, T., **Structure Analysis and System Specification**, Yourdon Press, New York 1978.

[Handloser 88] Handloser III, F.T., **A Graphical Rapid Prototyper for the Macintosh**, Dept. Computer Science Technical Report 88-60-1. Oregon State University, Corvallis, OR. 97331.

[Harrison 86] Harrison W. and C.R. Cook, **A Note on the Berry-Meekings Style Metric,** Communications of the ACM, vo. 29, no.2 (Feb. 1986) pp. 123- 125.

[Hsieh 88] Hsieh, Chia-chi, **A Graphical Editor for Pascal Programming on Macintosh,** Dept. Computer Science Technical Report 88-60-4.

[Lewis 88] Lewis, T.G., Handloser III, F.T., Bose,S and S. Yang, **Prototypes From Standard User Interface Management Systems**, Dept. Computer Science Technical Report 88-60-10, Oregon State University, Corvallis,OR. 97331.

[Kearney 86] Kerney, J.K., Sedlmeyer R.L., Thompson, W.B., Gray M.A., and Michael A. Adler, **Software Complexity Measurement,** Communications of the ACM, vol. 29, no. 11 (Nov. 1986) pp. 1044 - 1050.

[Martin 85] Martin, J. and C. McClure, **Diagramming Techniques for Analysts and Programmers,** Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632. (1985)

[Myer 87a] Myers, B.A., **Creating User Interaction Techniques by Demonstration,** IEEE Computer Graphics and Applications, vol. 7 no. 9, (Sept 87), pp. 51-60.

[Myer 87b] Myers, B.A., **Creating User Interfaces by Demonstration,** PhD Dissertation, Dept. of computer Science, Univ. Toronto, May 1987.

[Myer 89] Myers, B.A., **Tools for Creating User Interfaces: An introduction and Survey,** IEEE Software, vol. 6, no. 1, (Jan 89).

[Redish 86] Redish, K.A. and W.F. Smith, **Program Style Analysis: A Natural By-Product of Program Compilation,** Communications of the ACM, vol. 29, no. 2, (Feb. 86), pp. 126 - 133.

[Rosenberg 85] Rosenberg, D. **PRISM - Productivity Improvement for Software Engineers and Managers,** Proceedings of the 9th International Conference on Software Engineering, (1985) pp. 2 - 6.

[Wasserman 82] Wasserman, A.I. and D.T. Shewmake, **Rapid Prototyping of Interactive Information Systems**, ACM Software Engineering Notes, vol. 7, no. 5, (Dec. 1982), pp. 171-180.

[Weiser 81] Weiser, M.D. **Program Slicing,** Proceedings of the Fifth International Conference on Software Engineering, (1981), pp 430 - 449.