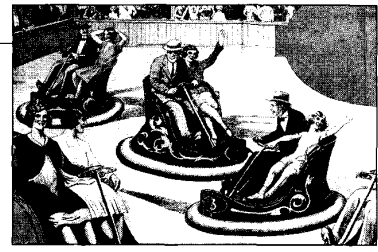


Maintenance Tools

Paul Oman, *University of Idaho*

Whether it's perfective, adaptive, or just plain corrective, the key to effective maintenance is program comprehension. Programmers need to understand what code does — and what it's supposed to do — *before* implementing a change.

Several prominent researchers have suggested that tools and techniques for understanding large programs will be one of the major challenges of the 1990s, so this section concentrates on tools to help the maintenance programmer analyze and understand code.

(Program testing, debugging, version control, and configuration management — all essential components of program maintenance — are left to the testing and management sections that follow this one.)

Not by accident, all the tools in this section are code-visualization tools. Program visualization may be even more beneficial in the maintenance arena than it has proven to be in analysis and design. While all these tools show how a program is structured, they use different means to achieve different ends.

The Objective-C Browser uses a windowing approach that displays hierarchical, functional, and inheritance information about code objects in C or Objective-C. Vifor, on the other hand, takes a different approach for Fortran visualization. Its browser has a graphical interface that lets you select, move, and zoom into icons representing parts of your program. As such, it gives you a graphical editing capability.

Seela takes yet another approach to visualization. Using reverse engineering, Seela converts code into a program-design language and lets you edit the structure chart, cut and paste to and from the code, and generate high-level documentation describing the code structure. Thus, it gives you an electronic path between code and its corresponding design language.

Unlike Seela, which works with many languages, Grasp/Ada is an example of a comprehension tool tailored to a specific language. Grasp/Ada builds graphical control-structure diagrams that highlight the control paths in and among Ada tasks. Act and Battle Map take an entirely different approach to visualizing control paths: They graph out all control paths, letting you quickly and easily see the control flow and corresponding complexity.

Program slicing is another technique for collapsing and depicting large amounts of code in a small viewport. EDSA is a static analyzer that lets you identify which structures you are interested in and then removes extraneous (intervening) code, thereby letting you see the big picture. With a similar intent but different form, Surgeon's Assistant slices up programs, extracts pertinent information, and displays data links and related characteristics so you can track the changes and influence of targeted structures.

Last, the Dependency Analysis Tool Set is not just a dependency analyzer — it is also a tool for building comprehension tools. The intent behind the tool is to provide a basis for determining program dependencies (data, calling, functional, and definitional) so, by creating your own application-specific front end, you can tailor-make your own comprehension aid. ❖

0740-7459/90/0500/0059/$01.00 © 1990 IEEE

Objective-C Browser details class structures

Stepstone's Objective-C Browser static-analysis tool lets you view C source code using the object-oriented extensions made by the Objective-C language. The browser can present information about class definitions that span multiple source files.

Information types. Version 1.0 provides three types of information source code:

- the contents,
- available cross-referencing data, and
- source-code contents with respect to the inheritance hierarchy.

The browser gets all three types of information from source code by using a data-file-generating utility; it presents the information via its graphical user interface.

The Objective-C Browser provides a top-down view of application source code based on the contents of the generated data file. Starting with the application, you can access information in greater detail by selecting from a set of menus. Each selection narrows the scope of the information.

To study the effects of a proposed software change, the Objective-C Browser can cross-reference items like functions, methods, and variables against all the functions, methods, classes, and files contained in a hierarchy data file. This lets you identify areas affected by modifications.

Code analysis. The Objective-C Browser also helps you analyze source code. This analysis capability includes generating class descriptions that reflect inheritance, summaries of the contents and usage of selected items in the source code, and usage diagrams that reflect where message sends and function calls are resolved.

By providing class descriptions that reflect the inheritance hierarchy, the Objective-C Browser lets you answer questions like "What methods does a class respond to?" The browser does this by using a set of heuristics that match the Objective-C runtime message-resolution sequence, including the rules for use of the Pose As method.

Finally, the Objective-C Browser offers a set of two diagrams for debugging behavior:

- the Self/Super usage diagram, which shows where messages sent to Self and Super are resolved, and
- the method/function usage diagram, which shows where all messages sent or functions called are resolved.

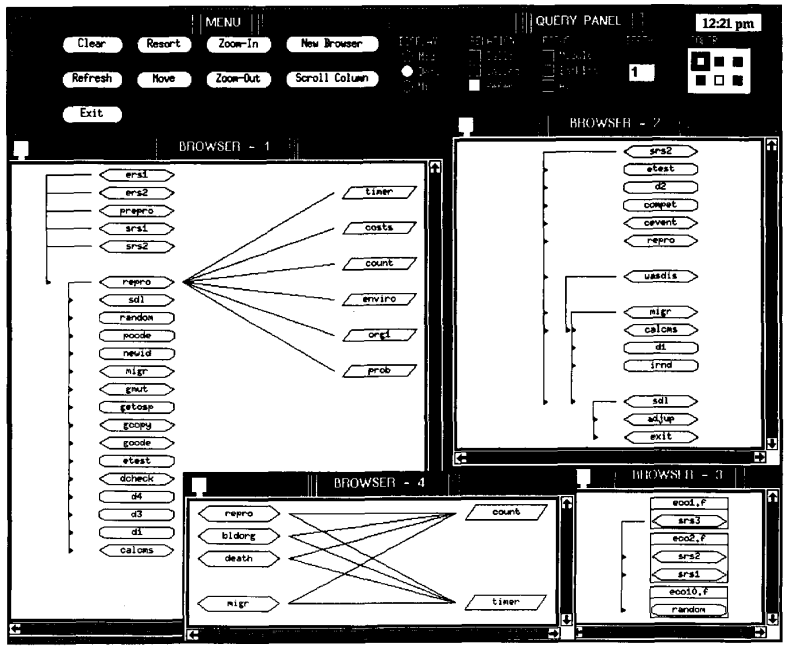
Both diagrams recursively check methods or functions called by the starting method or function, being careful to resolve circular references.

— Andrew Novobilski, Stepstone

Objective-C Browser runs on Sun 3, 4, and 386i, HP 9000, DEC VAX, and IBM RT PC workstations under Unix. It costs $995.
RS 164

Vifor transforms code skeletons to graphs

Software Tools and Technologies' Vifor lets you display and edit Fortran programs in two forms: as traditional code and as visual components with relationships.



Vifor contains transformation tools for both directions — from code to visual form and from visual form to code skeletons.

Vifor's graphical interface is implemented with browsers you control through a dialogue box with multiple operations.

To overcome the problem of display complexity and to improve readability, Vifor has several view levels and graph-rearrangement operations. Views are subsets of the database that contain structural information about the program.

Vifor uses two-column graphs, which is an original layout developed for Vifor. In these graphs, the left column consists of processes (the main program, subroutines, and functions) and the right column consists of common variables. The arrows on the left represent the call relationships and the lines in the middle represent reference relationships.

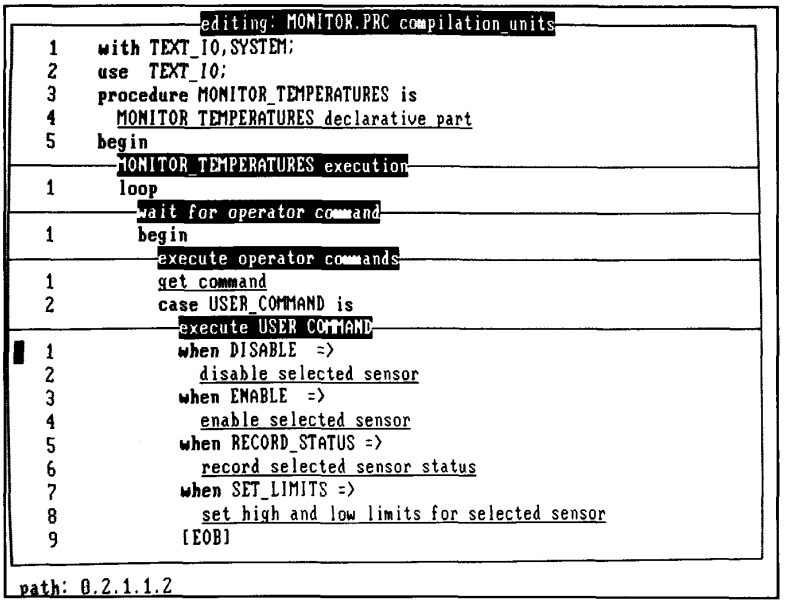
— Vaclav Rajlich, Software Tools and Technologies

Vifor runs on DEC VAXstation 2000 and MicroVAX IIGPSs under Ultrix and on Sun workstations under Sun News. It costs $1,995. **RS 165**

Seela aids maintenance with code-block focus

Tuval Software Industries' Seela interactive reverse-engineering tool supports the maintenance and documentation of structured programs. It features a top-down program display that increases the readability of structured programs and includes a structure editor, browser, pretty printer, and source-code document generator. Seela works with Ada, Cobol, C, Pascal, PL/M, and Fortran code.

Seela was designed to bridge the gap between the project's design description and the source code. Instead of requiring a separate program-design-language description, it analyzes the source code and projects it on the screen so it appears as a readable program-design-language code.



Design unit. Seela focuses your attention on a design unit not typically used: the program block, which is a sequence of language statements set off by keyword pairs like Procedure/End, If/Endif, and Do/Enddo. The program block is an intermediate design unit between the simple statement and the compound statement.

A program block will generally have a meaning for the application; it might be a set of declarations or a sequence of statements that executes a certain function. Accordingly, Seela encourages you to assign it a name much as you would a variable or a subprogram. This block name describes the block's contents or function in the application.

The Seela display breaks up your program into its program blocks, each block appearing in a window at a single level of detail. Seela explicitly shows the highest-level program lines; it shows lower level nested blocks as one-line, user-formulated block names. The figure above shows a sample screen.

To see a lower level nested block, you select its block name and get the program-design-language display of the nested block. You can do this to nested blocks at each subsequent block, descending through a path of interest in the program's block structure.

Productivity tools. Seela provides four productivity tools: a structure editor, a program browser, a program formatter, and a program documenter.

The Seela structure editor lets you work directly with the program blocks as design units. Operations include locating program blocks via the block directory, naming natural blocks and user blocks, creating and deleting user blocks, cutting and pasting directly on the Seela program display, interfacing to VMS's program editor, and navigating through the module-calling tree.

The locater commands help you find program sections quickly. The Seela display shows you the desired section as well as its location in the program's block structure. You can also locate a program block by just selecting its block name from the block directory.

The user-block facility helps you organize, outline, and simplify a program by creating new blocks out of sequences of program statements. You can create a user block from almost any section of code: comments, declarations, statements, or other user blocks.

You can cut and paste directly on the Seela display. Because program blocks are represented by single lines, you can manipulate them as you would a single line of code in a program editor. You can also cut and paste blocks from file to file.

You can invoke any VMS program editor from the Seela structure editor. Seela sets up the program editor and transfers selected parts of code for editing.

Seela produces a comprehensive and readable project document from source-code files. The document includes top-down and sequential listings, block directories, and an index of module definitions. The document presents your source code as a design document in the same way as the top-down program display. With a schema command set, you create the document and can tailor it for a range of documentation requirements.

—Joel Harband,
Tuval Software Industries

Seela runs on DEC VAX/VMS mainframes and workstations. It costs $2,000 on VAXstations, $4,000 on MicroVAX IIs and VAX 7*xx* mainframes, $5,500 on MicroVAX 3*xxx* workstations, and $7,000 on VAX 6*xxx* and 8*xxx* mainframes for a single-user, single-language license. **RS 166**

Battle Map, Act show code structure, complexity

McCabe & Associates' Battle Map tool lets developers and managers reverse-engineer the design and quality attributes from existing code. Battle Map displays the structure of any system or subsystem graphically, using special symbols to indicate the complexity of each piece of code in the design. You may also use Battle Map to drive integration and unit testing by producing test paths and conditions.

Battle Map can recognize factors like cyclomatic and essential complexity and the presence or absence of a specification for each module. Cyclomatic complexity is becoming more widely used as a measure of reliability and maintainability, and essential complexity is proving to be equally powerful as an indicator of code structure. These factors quickly identify high-cost areas for maintenance, as well as revealing unreliable or unmaintainable design structures.

In its graphical display, Battle Map indicates specifications with a horizontal bar above the module. Because you define what modules have specifications, you can tailor this indicator to represent any other module characteristic. For example, you could apply the results of cost accounting to the structure chart, using the specification bar to represent those modules that exceeded a certain cost threshold.

Battle Map uses color to describe the degree of complexity of each module. Green indicates that the module has low complexity and has relatively good structure. Yellow represents modules with high cyclomatic complexity but with low essential complexity — the module is not unmanageable, but you should partition it to assure future maintainability. Red modules have both high cyclomatic and high essential complexity metrics. By using this stoplight effect, Battle Map produces a color snapshot of the system design, revealing where problems exist and where trouble spots are likely to appear.

Battle Map lets you work with all or part of the design. For the software engineer who is rebuilding a part of a system, Battle Map lets him view only that branch of the design as a complete structure chart. For the manager concerned with the complexity of driver modules as a measure of quality, Battle Map allows partial and complete stubbing of design subsystems. By combining these two facilities, you can isolate parts of the program structure in a graphical summary.

The Act complexity-analysis tool, which is available separately as well as part of Battle Map, computes the cyclomatic-complexity measure for each module of source code. It automates the baseline testing methodology and McCabe Structured Testing methodology. The tool produces flow graphs of each module's logic and generates the end-to-end test paths for each module. Act expands compound conditions, their test conditions, and test paths. Act pinpoints where the software is too complex to be reliable and quantifies the number of tests needed.

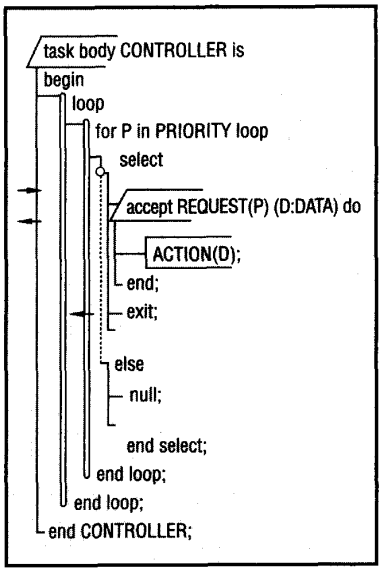
Act analyzes C, Fortran, Cobol, Ada, Basic, PL/I, Pascal, and 8086 and 6502 assembly languages, as well as program-design languages. Act interfaces with Battle Map for design analysis and software reengineering.

— Thomas McCabe, Jr., McCabe & Associates

Battle Map and Act run on PCs under MS-DOS, on HP, Sun, and DEC VAX workstations under Unix, and on DEC VAX mainframes under VMS with Ultrix. Prices begin at $6,500 for the PC version, $21,500 for the workstation version, and $29,000 for a 16-user VAX mainframe version. **RS 168**

Grasp/Ada uses control structure

Auburn University's Grasp/Ada project seeks to develop tools to extract and generate graphical representations from Ada program-design-language or source code. The project has three phases: algorithmic (program-design-language/code), architectural, and system-level diagrams. Phase 1 is completed, resulting in a program-design language-/code-level graphical notation and prototype supporting tool.



The tool uses control-structure diagrams as the basis for a graphical representation that maps directly to Ada control constructs. You can overlay such diagrams directly onto pretty-printed Ada code, as the figure at left shows. The improved readability provided by control-structure diagrams reduces the time required to understand the code.

The prototype supporting tool essentially is a pretty printer/editor for control-structure diagrams. Before printing your diagrams, you may preview the control-structure diagram on screen. The editor also lets you suppress the control-structure diagram to display conventional pretty-printed code.

— James Cross II, Auburn University

The Grasp/Ada diagram tool runs on Sun 4 workstations under SunOS 4.0.3 or later and X Windows 11.7 or later; it outputs to HP Laserjet Series II and Postscript printers. The prototype will be available this spring for a $50 distribution fee. **RS 167**

Expert Dataflow and Static Analysis tool

Array Systems Computing's Expert Dataflow and Static Analysis tool lets you view Ada programs with unimportant details skipped over and lets you follow the control flow and dataflow through the program. EDSA helps you analyze and understand the logic of Ada source code while simplifying the reading of the code.

Unlike dynamic-analysis tools like source-level debuggers and embedded diagnostic code that require you to constantly edit and recompile programs and to provide input that forces the execution of all control-flow paths, EDSA helps you traverse all possible paths in the original, uncompiled source code.

Components. EDSA components include pretty printers, flow stepping, and cross-reference generators:

- In a pretty-printed display, you can remove extraneous information in various ways. You can skip over statements nested below a specified level. You can limit the view to where specified variables occur or to specified structural elements like procedures/packages/tasks, compound statements, call structure, and tasking structure. You can also combine views to make a new view. The figure at right shows the usage view of a program.
- You can step through the program's logical control flow (as opposed to its textual order), with EDSA automatically tracking branches and backtracking so all possible paths are examined. You can step through code within a procedure or across procedures. To find out where a particular variable received its value or where a new value assigned to a variable may be used, EDSA lets you track dataflow. Again, EDSA tracks all possible dataflow paths and backtracks so none is missed. You determine whether values are traced into subroutine or entry calls.
- While reading a program, if you need to quickly refer to the declaration of a variable or subprogram, you can move immediately to the declaration and return to where you were working, regardless of how far apart the two are in the program. You can also use this feature to examine each location where a variable is used or defined or where a subprogram is called.

Applying EDSA. You can apply EDSA to two major development and maintenance activities: finding logical errors and determining the consequences of proposed changes.

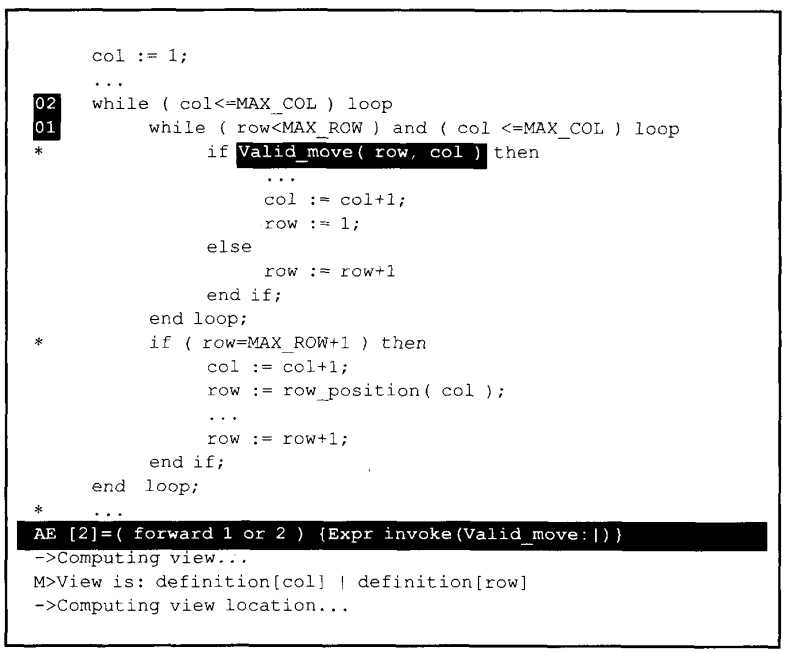
As a debugging tool, the backward dataflow-tracing capability helps you locate where a variable may have acquired an erroneous value. To remove extraneous context, you may reduce the program view to include only occurrences of the variable of interest.

Sometimes, however, a bug does not manifest itself as a variable with a bad value. If you can determine that a particular source statement was reached before any problem occurred, EDSA can help you trace all control-flow paths forward from that statement to locate a statement that may be problematic. To aid this, EDSA lets you mark all statements that you have examined and confirmed correct. You can propagate these marks to logically successive code using dataflow, so you need only verify a statement locally and let EDSA deem it to be correct (or globally verified) only when the statements producing its inputs have themselves been globally verified. Similarly, subsequent revocation of a statement's validation causes its dataflow successors to lose their global validation.

As a maintenance tool, EDSA's forward dataflow tracing lets you find all locations where changes to how a variable is updated may affect the program. This lets you predict side effects of proposed changes and make sure that these changes do not introduce any unwanted side effect.

— Leonard Vanek and Linda Davis, Array Systems Computing

EDSA runs on PCs under MS-DOS, on Sun, Apollo, and DEC workstations under Unix, and on VAX mainframes under Unix. It costs $2,450 for PCs, $3,250 for Sun, Apollo, and VAXstation workstations, $5,500 for MicroVAXs, and $11,000 to $22,000 for VAX mainframes, depending on VAX model. Quantity discounts are available. It uses Termcap and Terminfo under Unix and does not support VT220, VT100, or IBM PC-compatible keyboards. **RS 169**



Surgeon's Assistant limits side effects

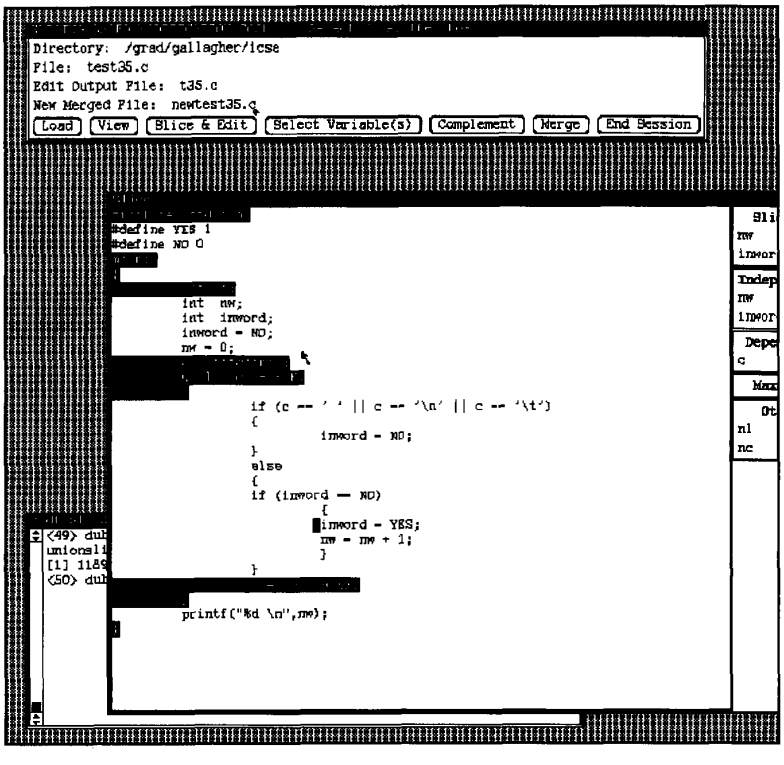
Loyola College's Surgeon's Assistant delivers semantic information and editing guidance to help you formulate a maintenance solution with no undetected links to unmodified code, thereby eliminating the need for regression testing.

Surgeon's Assistant was constructed to validate decomposition slicing as a maintenance technique and to assess its effects on the maintenance process. The prototype tool, which works with C code, is under continuing development at Loyola College in Maryland and the University of Maryland at Baltimore.

The tool is invoked as a background process from a command window, as the figure below shows. The tool initially presents one window, the upper left window, composed of the file interface and the tool commands. There are four other windows: a viewing window, a decomposition-variable selection window, an editing window, and a complement-viewing window. These four windows are initially invisible.

Components and functions. The file interface has four components: Directory, which is the current subdirectory, File, which is the file to be modified, Edit Output File, which is the file into which the modified slice is written for debugging and testing, and New Merged File, which is the name of the file that will contain the modified program.

The tool interface has seven commands. The Load comand builds all of the program's decomposition slices. View opens a window with the current file displayed. Slice & Edit opens a window and invokes the special-purpose editor. Select Variable(s) selects the decomposition variables. Complement opens a window with the complement of the current decomposition displayed. The complement is the part of the program that remains unchanged by the modification. Merge constructs the new program. The merger checks for correct C syntax. End Session terminates the session.



Typical session. In a typical session, you first load the program and select decomposition variables. A window appears, listing all the program's variables. You click on the desired variables, which are then displayed in reverse video. The source lines in normal video are the independent statements; you may modify them as you see fit. The dependent lines, in reverse video, are necessary to the computation of the complement, so the editor prohibits their alteration.

The editor also displays status information in the right subwindows. The top subwindow redisplays the slicing variables. The second subwindow displays independent variables, those whose values can be recomputed without affecting the complement. The second subwindow displays dependent variables, those whose values cannot be recomputed. Variables that will force the current slice to be maximal (decomposable) are next. In the figure, the slice is maximal, so this display is empty. Slices that are not maximal are also visually apparent, since all statements in the editing window are in reverse video. Variables used in the program but not in the slice are displayed last.

After making changes, you save the file and then compile and test the new program. When you accept the changed code, Surgeon's Assistant merges it back into the complement. The merger component needs only verify that any added control flow does not control any dependent statements. This approach means changes are completed without affecting the complement, so regression testing is not needed.

To use slices as a maintenance aid, we required a program slice that was independent of line numbers and captured all the computation on a given variable. Thus, we devised the decomposition slice to determine those statements and variables that can be modified without affecting other decomposition slices.

— Keith Gallagher, Loyola College

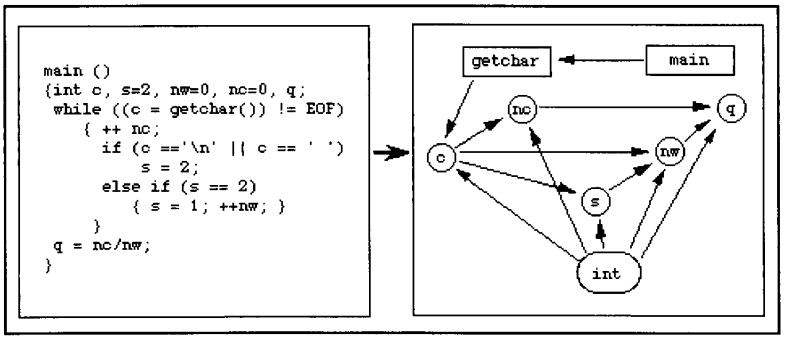
Surgeon's Assistant is a prototype and is not publicly available. It runs on Sun workstations with Sun View under SunOS Version 4.0. **RS 170**

Dependency Analysis Tool Set prototype

The University of West Florida's prototype Dependency Analysis Tool Set helps you understand the complex interrelationships in large C systems.

You can use the tools directly to query system dependencies or as a building block from which to construct other maintenance tools. The tool set contains programs to set up a program database from C code, query the database about types of dependencies, and compare two versions to identify the effects of a change. A simple display program lets you view dependencies, but you can also pipe query output to other tools for more sophisticated analysis.

The tool set assumes that the system has been analyzed and the dependencies stored in a simple dependency graph. Each node of this graph represents a program entity that has an identifier in the original system; each arc represents a dependency between entities (see the figure above).



The query tool lets you select the kinds of dependencies of interest, search for indirect dependencies, and filter out some kinds of false dependencies.

—Norman Wilde, University of West Florida

The Dependency Analysis Tool Set runs on MS-DOS PCs with 2 Mbytes of RAM and on Unix-based workstations. A prototype, the tool set is not publicly available. **RS 171**