DS-Viewer—An interactive graphical data structure presentation facility

by D. P. Pazel

DS-Viewer is a tool that is the result of a research project in data structure presentation within a program state. This tool addresses two distinct issues in this area: (1) to effectively present data structures themselves for a given program state and (2) to present groups of data structures and their interrelationships as described by their pointer definitions. Graphical presentations were developed to address these issues. For the data structure presentation, the user is provided a display window for any single data structure instance formatted with its fields and field values. Flexibility in display is provided by allowing the user a choice from the various value formats for each field. For groups of data structure instances, a graphical drawing space is provided in which pictures of these data structure instances and their interrelationships are drawn as blocks and arrows. The computer assists the user in drawing such a picture by describing its components, allowing the user to choose which to draw and to construct as much of the picture as desired.

Program maintenance occupies a large portion of a software product's cost and affects its life cycle. Estimates vary greatly, but it is reasonable to say that 50 to 90 percent of the total cost can be attributed to continued enhancements, fixing defects, or adapting to new software or hardware. Consequently, there is a need for effective, efficient, and user-friendly debugging tools to help decrease this overhead.

Major advances in debugging tools have been seen in the last decade. The days of debugging directly with the primitive machine state are nearing an end as source level tools are bringing the debugging process closer to the source program code. It is not unusual for a debugging tool to present to the user execution-level information in the context of the source program code itself (e.g., current source line, program variable values, etc.). Many debugging tools are now highly interactive, making excellent use of full-screen technology by initiating debugging actions through cursor-sensitive operations.

The personal computer, in conjunction with high-resolution graphical displays, presents an opportunity to the next generation of debugging tool developers. By using workstations, it is possible to enhance the debugging process with highly visual graphical displays and a high degree of interactivity. This paper presents the results of research focused primarily on data structure presentation. This research exploits the graphical display capabilities of current IBM PC or PS/2 workstations while at the same time presenting data to the user in a familiar paradigm.

After a brief survey, a general discussion of the problem domain is presented. Following that is a description of the DS-Viewer tool, its function and presentation schemes. A detailed examination of the

various data displays and their features follows. The graphical presentation of groups of structures and their pointer relationships is discussed in detail. This mode of presentation is the result of a highly interactive session whereby the computer describes the picture's components and the user selects which components to show.

Recent work

There have been a number of significant advances in graphical debugging systems. Each approaches the problem differently and has its own particular limitations. A debugger known as the Blit debugger[1] is a C program debugger for programs on a particular multiprocessing bitmap terminal. Among the most useful features of the Blit debugger are the effective use of overlapping windows, menu bars, and a mouse. This debugger provides some function for the display of data structures, but not for the display of multiple data structures and their pointers, such as graphical displays of linked lists.

Another debugging tool known as Incense[2] targets the Mesa programming language. Among its most notable techniques is the use of formatted displays of data. For example, data structures may be illustrated as rectangles with nested rectangles to indicate substructures. Another interesting feature of this debugging tool is its capacity to display groups of related structures and their pointers. Both in displaying individual and related structures, Incense capitalizes on the graphical ability to compress information on the screen in order to see the character of the overall picture. The conceptual payoff of this approach is not clear in very large, complex structures, nor when there is a very large number of structures to display. Even with a minimum shrink size, eventually structures simply become clusters of blurred images.

VIPS[3] is a debugger for ADA that works on the PERQ workstation. VIPS attempts to address all aspects of a debugging system using a multiwindowed environment. The techniques used in data structure display are quite similar to those used in Incense but are slightly less powerful when it comes to displaying groups of data structures. It is not clear what happens when the complexity of data structures tests the size of a data display window. Nor is it clear what happens to the display resolution when the display is severely shrunk.

Related efforts in this area include the Pecan debugger,[4] Balsa,[5] and program visualization.[6]

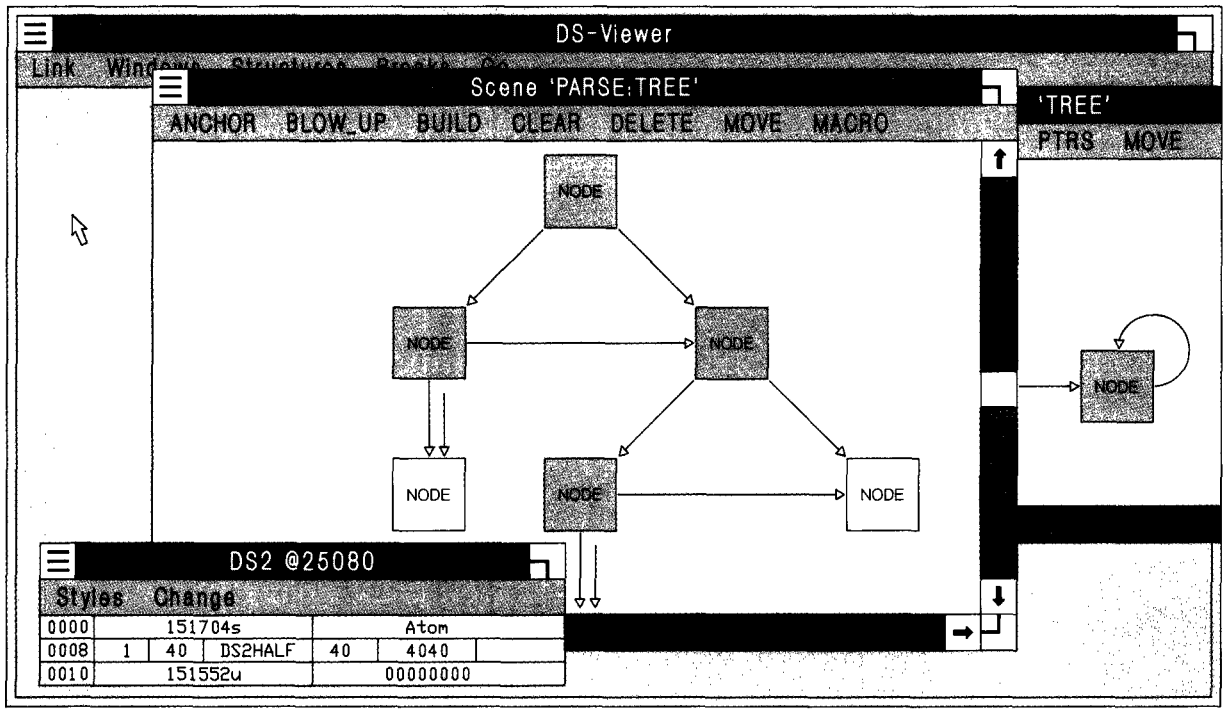
The problem

The need for data structure presentation tools is most keenly felt in debugging applications that rely on the data structure features of languages such as Pascal, C, or PL/I or the unrestricted data structure features of an assembler language. The data structure semantics of such languages are usually quite open-ended, allowing the user to implement very complex data structure operations consisting of many kinds of data structures and data structure interrelationships. Typical of such data structure interrelationships and operations are linked lists and trees, where the nodes of each are data structures and their interrelationships are implemented through pointers within the data structure definitions. However, it is not unusual for more complex structures to be designed for an application, and the programs to manage such structures are similarly complex and fault-prone.

In debugging these types of applications, several issues become apparent. Problems usually appear within the data structures themselves. Values may be incorrectly set or, even worse, the pointer to a structure may be misused as a pointer to an incorrect data structure type, resulting in data structure field abuse. Pointers are often misplaced or missing altogether. In this case, the current program state contains violations of the intended data structure interrelationships defined for the application. Attempting to find these violations usually involves trying to construct a picture of these interrelationships from the program state by locating many data structure pointers and following them. Additionally, the problem is compounded by the number of data structures that may be present at any given program state. From the viewpoint of a debugger, the first problem indicates a need for a facility to present formatted data structures to easily validate the values of fields. The second problem presents a need for a mechanism that allows the user to visualize large groups of data structures and how they interrelate or point to each other, to see whether the program state is faithful to the intended data structure plan. Generally speaking, current debuggers provide the formatted data structures but do not provide means for visualizing large groups of data structures.

In some cases, the problem of debugging applications using complex data structures is exacerbated by implementation languages such as assembler or PL/I which allow the use of untyped pointers. In these cases, a pointer may indicate arbitrary data structure types since the semantics of pointer definition may

Figure 1 A DS-Viewer session screen



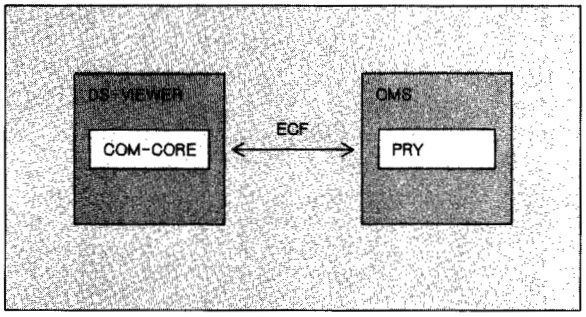
not include the type of data structure. An understanding of the intended data structure scenario must either be extracted from documentation or inferred from the program. Even with the aid of a debugging tool that solves the problems described above, the user would find it helpful to know details about the data structure. The information would help detect aberrations in data structure or verify the intended usage of pointers.

An overview of DS-Viewer

The DS-Viewer tool is the result of a research project in program data structure presentation that grew out of an earlier effort in multiwindowed machine-level debugging.[7] The tool addresses two distinct issues in data structure display: the effective presentation of data structures themselves for a given program state, and the presentation of groups of data structures and their interrelationships as described by their pointer definitions. The tool is written in C and runs under Microsoft Windows® on an IBM PC/AT or PS/2. Being a Microsoft Windows application, DS-Viewer is multiwindowed by nature. Microsoft Windows provides a standard interface to several important graphical display aids such as menu bars, pull-down menus, and scroll bars, and provides a standard interface to a mouse. These features facilitated the development of the tool. The graphical support within Microsoft Windows makes it possible to implement a presentation of data using graphical pictures. Figure 1 shows DS-Viewer in operation.

DS-Viewer displays data structure information through user-supplied semantic information about the data structures. In part, DS-Viewer obtains this information from a local file containing all or some of the application's data structure definitions. The file is provided by the user. From this, DS-Viewer provides the capability to recall a data structure definition as a display window on the screen which is formatted with field definition information. Similar displays are provided for *data structure instance* (a specific data structure based on a generalized data structure definition) which are comprised of actual program memory data within a program state interpreted against some given data structure definition. DS-Viewer uses the definitions to format a display window with details of that data structure instance's field values. In either case, a window is dedicated to

Figure 2 DS-Viewer's architecture configuration



the display, and the window possesses features which are menu- and mouse-driven, to change the presentation of the information at the user's request.

The remaining input semantic information relates to the definition of pointers. Pointer information consists of the identity of the data structure definition that owns the pointer, the field in the owner that represents the pointer, and an identifier which represents the type of data indicated by the pointer. All of this information comprises what is called a *pointer relationship*. Since all information about a pointer relationship is part of the language and is therefore static, the input file could provide information about pointer relationships. However, DS-Viewer attempts to address implementation languages which contain both typed and untyped pointers, such as PL/I or assembler. In particular, DS-Viewer as presented in this paper is tailored towards S/370 Assembler Language.[8] In DS-Viewer, pointer relationships are defined by means of a graphical editor that is part of the tool. With this editor, data structure definitions are visualized as blocks and pointer relationships as arrows. The user creates and manipulates these objects to define pointer relationships. A collection of data structure definitions and pointer relationships is called a *format*, and the user may edit any number of formats during a session. From a user-provided format and a given program state, DS-Viewer is able to construct with user interaction a picture consisting of blocks to represent data structure instances and arrows to represent actual pointers as defined by the pointer relationships. This picture is called a *scene* and the user may construct any number of scenes during a session.

A unique feature of DS-Viewer is that it is a PC/AT workstation tool that displays data structure information from active mainframe programs. More specifically, DS-Viewer interacts with executing CMS application programs on VM/370 systems much as a debugging tool would, except that DS-Viewer initiates that interaction from a PC/AT workstation. A schematic of the workstation/host interaction is illustrated in Figure 2. Under CMS, an application program executes. Coresident (i.e., nucleus-resident) with this application is a software probe which contains debugging and communications functions by which the executing application can be monitored. The software probe used is an internal IBM research debugging tool called PRY. This tool debugs on the 370 code level and provides a noninteractive mode whereby alternate means of debugger command input may be provided, e.g., through a communications protocol, as described. On the workstation, DS-Viewer exists as a task under Microsoft Windows. DS-Viewer contains a communication component (Com-Core) that utilizes ECF[9] in its protocol to the software probe on CMS. When the user initiates an activity such as the display of a data structure, DS-Viewer translates that activity into a debugging command (e.g., acquiring memory contents). That command is issued by Com-Core to the software probe via the communications path. The activity is completed upon receipt of the information by Com-Core and DS-Viewer. Although the function is limited, it is expandable since any debugging command could be issued by Com-Core to the probe in CMS. DS-Viewer provides a primitive program control mechanism by which breakpoints may be set or unset, and for resuming program execution. Two menu items on the main window menu bar provide that function. The *Breaks* option provides a standard dialog window for adding and deleting program break points. The *Go* option issues a program resume command to the executing program. The *Go* option becomes highlighted and remains so as long as the program has not hit a break point. Thus, when *Go* becomes unhighlighted, the programmer knows that the program is in a state to be probed by DS-Viewer. However, the main focus of this work is data presentation.

The sample screen in Figure 1 shows several of the typical windows found in a DS-Viewer session. The small window titled DS2 @25080 is an example of a data structure instance. It indicates that DS2 is the data structure definition and 25080 is the program data memory address of the instance. The figure illustrates how the tool displays a particular data structure instance as a window formatted with field values. The larger window titled PARSE:TREE is an example of a scene of data structure instances with

Figure 3 (A) List view; (B) block view of a data structure definition

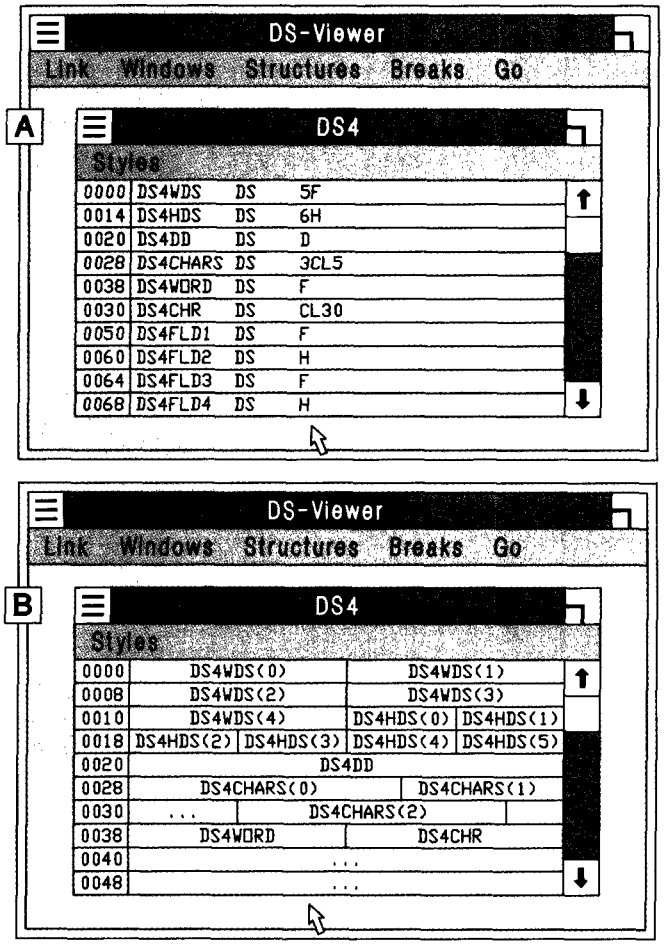
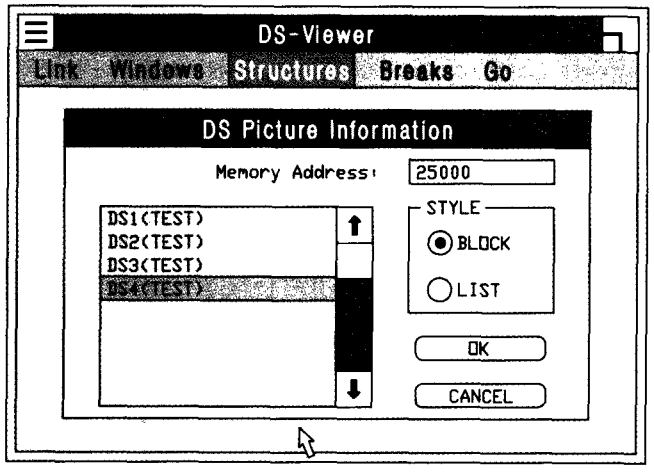


Figure 4 Dialog for constructing a data structure instance



their pointer relationships. The NODE blocks represent specific data structure instances of NODEs, and the arrows represent pointers. Later it will be seen that from these NODE blocks the user can easily produce the formatted data structure instance windows. Partially obscured by the scene window is an edit window for the format used to generate the scene shown.

DS-Viewer also provides a number of other windows that contain information pertinent to the machine state, such as the PSW, general-purpose registers, floating-point registers, and arbitrary storage areas. These windows are displayed through the *Windows* option on the main window's menu bar. These windows are self-descriptive and will not be discussed in this paper.

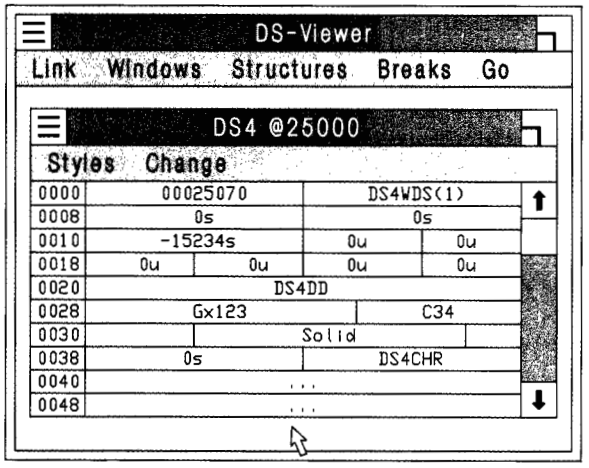
Data structure definition windows

A data structure definition consists of an identifying name and the source code that defines it. DS-Viewer receives these definitions as input through a *definition file* which is a local user file containing the definitions extracted from an application's source code. Many definition files may be specified, provided that each definition has a unique name. However, it is possible to use the same name to specify definitions across different files, allowing for different versions of a data structure.

User control in entering, displaying, and deleting definitions is focused through a dialog window[10] obtained from a menu item on the main window. This dialog provides a list of definitions and their originating files, an input area for specifying new definition files, and a variety of options to execute against definitions. For example, when a definition file name is entered at the input field and the ADD option is selected, the name of each data structure definition in that file appears in a scrollable list box and is recognized by DS-Viewer. To aid in further identification, each structure definition name has as a suffix in parentheses the file name from which it originated. During a session the user may identify a definition's origin or distinguish among several definitions with the same name. An individual definition may be deleted by selecting it in the list box and choosing the DELETE option. Similarly, a definition may be displayed with an ILLUSTRATE option through which a window appears showing the contents of the definition.

The two styles of definition display supported by DS-Viewer are shown in Figures 3A and 3B. The same

Figure 5 Block view of a data structure instance



definition is illustrated in these two styles. The example in Figure 3A shows the data structure definition of DS4 in *list* style. This is the assembler language definition named DS4, and each line shows the definition of each field. For example, the field DS4WDS is declared as five full integer words (32-bit words). The running sum in hexadecimal along the left column is an indication of the location of each defined field relative to the structure's base. In this case DS4WDS occupies 20 bytes starting from the beginning of the structure.

The example in Figure 3B shows the data structure definition DS4 in *block* style. This form of definition representation is well known to systems programmers, and this paradigm may be found in numerous program language manuals.[11] Each row in this presentation represents 8 bytes of contiguous memory. The tiling of the picture provides a partitioning of the data structure into the fields. For example, the five indexed instances of DS4WDS correspond to the five adjacent 4-byte words that define that field, DS4DD is a double-word (8-byte) field, etc. Also, continuations of fields across rows is indicated by "..." appearing in the succeeding areas. This representation provides a topography of the fields. With this knowledge of field adjacency, the programmer is more able to observe data patterns or distinguish data structure instances in foreign memory areas.

During a session, the user may request any number of definition windows to be displayed. However, two definition windows cannot present the same definition at the same time—a definition display is unique across the user session. Also, the user may flip-flop between the two presentations, BLOCK and LIST, through menu options on the definition window itself. Since the visual image of a definition may easily exceed the size of the screen, scroll bars are provided to view other parts of a definition, and of course the window may be resized to the user's liking.

Data structure instance windows

The display of a data structure instance is closely allied to that of the definition. A data structure instance is, after all, program memory data interpreted against a data structure definition. In fact the same display styles are used, but with some modifications to allow the user to obtain different field value representations.

The focus of operations on data structure instances is a dialog invoked through an option on the main window menu bar. This dialog window invoked is illustrated in Figure 4. The user is given a complete selection of definitions in a list box and a prompt area to enter the location of the data structure within program memory. The definition is selected by depressing a mouse button on an entry in the list box. When OK is selected, the data structure instance window is displayed.

Figure 5 shows an example of a data structure instance in block style. In this case a data structure instance is based on the data structure definition DS4. Similarly, like the block style of definition display, this display shows an area partitioned into field areas. However, the values of the fields are displayed instead of field names. The representation of the values may be changed by using the right mouse button on any field area. With that, a decimal typed field (declared *F* or *H*) will change to signed decimal, unsigned decimal, character, and hexadecimal cyclically through successive depressions of that button. The symbols *s* and *u* provide the necessary clues to the user as to the current representation. For character fields (declared *C*) the change is strictly between character interpretation and hexadecimal. But in addition, a field area can flip-flop between its name written in black and its value written in red by using the left mouse button on any field area. In the example, DS4WDS(1), DS4DD, and DS4CHR are field names acquired in this manner.

Again, similar to data structure definitions, data structure instances may be displayed in list style. In such a display, the window shows each field name

followed by the value of the field itself. The user may change the field representation by depressing a mouse button on the field as mentioned above. Scroll bars are provided to scan through all the fields or across any field's value that extends beyond the limits of the window.

The strong similarity between the data structure definition and instance displays provides a consistency for the user to interact with them. For example, both have a *Styles* menu option which allows the user to switch between list and block styles within the same window. However, data structure instances additionally allow a *Change* option whereby the user may quickly change the instance memory location or the interpreting data structure definition. The selection of this option provides a dialog window, offering a complete list of additional definitions from which one must be selected (the current one is highlighted by default), and a prompt to enter a new instance location in program memory (the current location is present by default). This feature allows the user to explore the identity of unknown program memory data by interpreting it against different data structure definitions. It also allows the user to adjust the instance location against a given data structure definition, possibly revealing pointer values that may be "off" by a few memory bytes.

The user is presented a drawing space into which a picture or schema representing the format will be drawn.

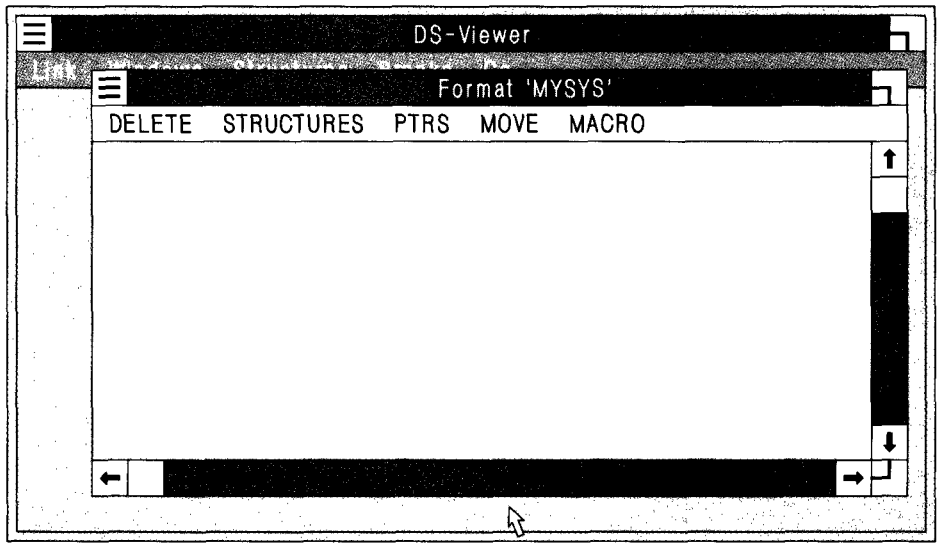
Formats and scenes

DS-Viewer's semantic information about data structures up to this point has been confined to information about each structure definition individually. The presentation of a scene of data structure instances and their pointer relationships requires far more information. For DS-Viewer to accomplish such a presentation, interstructure information must be provided in the form of pointer relationships. A pointer relationship consists of a *field* contained in a *source* data structure definition, and a *target* data structure definition representing the type of data structure to which the field points. An application's pointer relationships are usually provided by the definition of implementation language's data structure syntax. For implementation languages with untyped pointers this is not the case; the target data structure definition cannot be deduced from the pointer definition. Thus to specify the pointer relationships, DS-Viewer provides a graphical editor called the *format editor*. The format editor allows the user to specify the pointer relationships through a paradigm familiar to most programmers. As an example, data structure types are represented by blocks and pointer relationships are represented by arrows. This phase of semantic information input will be described in the next section.

The scene-building process is an interactive process between the user and DS-Viewer. This is achieved through a graphical scene editor based on the same paradigm used in the format editor. At this time blocks represent data structure instances and arrows represent actual pointers. Through a user-specified format, the scene editor enumerates and describes to the user all the pointers for any specific data structure instance. From that list, the user may select specific pointers and the data structures to which they point and add them to the scene, a process called *resolving* a data structure instance. Thus, a typical user scenario for the scene editor is as follows. The user places on the scene a few known data structure instances. These initially placed data structures are referred to as *anchors*. This term's intent is to convey a sense that the scene is built from a foundation of a few known data structure instances. The user then proceeds to ask DS-Viewer to calculate and list the pointers for some anchor. The various pointers and instances to which they point are presented to the user who selects those that are to be added to the scene. Then the user selects another anchor or data structure instance for resolution and so forth, until the user determines that the scene is sufficiently built. The full process of scene building will be described later.

Several interesting points should be made. Although format building may be necessitated by the weakness of languages which contain typeless pointers, it could be argued that this process is necessary in any case. Format editing provides a focus for scene construction by specifying what items to present. The number

Figure 6 Format editing screen



of pointer relationships in a complex application in any language is apt to be large. Format editing provides a way to consider just those that are of interest to the programmer. In the case of languages with typed pointers, a format editor would be used to restrict attention to specific pointer relationships used to implement specific data structure constructs, as linked list and trees. Generally speaking, languages with typeless pointers and languages with typed pointers would use a format editor in complementary ways; the typeless pointer for providing detail and the typed pointers for restricting detail. Format editing provides a way to improvise formats to reveal the true function of the pointer's usage in the case of applications where a pointer's usage is not clearly defined.

The interactive method of building a scene previously described bears an advantage over fully computer-generated scenes from a format. Even with a highly restricted format, the full detail encompassed in an associated scene could be very complex. User interaction provides a way to explore around a scene in unstructured but user-directed ways. For example, a full tree or linked list does not have to be fully built onto the scene. Instead, branches of a tree or partial traversals of lists may suffice. Also, with the user "growing" the scene through a graphical interface, the scene can be constructed so as to augment the known semantics of the application. That is, the user can place the data structure instances and pointers so as to reveal higher-level relationships among them that are not apparent, such as related link lists, trees, and ragged tables.

Format editing

A format editor defines formats, collections of data structure, and pointer relationship definitions, to the DS-Viewer. An unspecified number of formats may be defined during a session. Each is given a unique name by the user and in this way is distinguishable. The format editor is invoked through an option on the main window and focuses the editing session through another dialog window. This dialog window provides basic services for operating on formats such as reviewing, adding, deleting, or changing formats. A list box provides a list of current formats and prompting information is provided to enter the names of formats. The various options mentioned are provided as visual push buttons. To add a new format, a name is entered at the prompt and the ADD option is selected. For example, to edit a format, a format name is selected in the list box and the EDIT option is selected. This action invokes the format edit window.

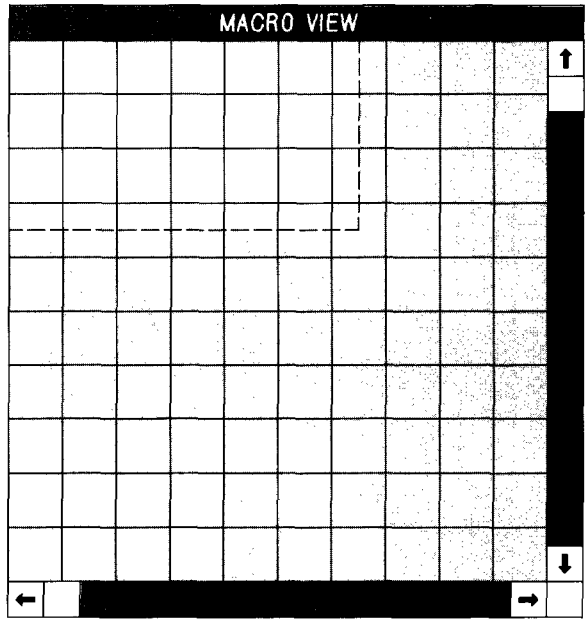
An example of a format edit window is shown in Figure 6. The user is presented a drawing space into which a picture or schema representing the format will be drawn. To accommodate large formats, the drawing space is much larger than the window size.

During the editing process, the user may move through different areas of the drawing space with the scroll bars shown at the right and bottom. However, a "bird's eye" view of the entire drawing space may be acquired through the MACRO option on the format edit window menu bar. Selection of this option creates a child window of the format edit window depicting the full scale of the drawing area. This is shown in the MACRO VIEW window in Figure 7. This window shows the full drawing area divided into a 10-by-10 grid and a broader rectangular outline reflecting the area encompassed by the current editing window, called the *focus*. The scroll bars on the right and bottom of the MACRO VIEW window allow the user to move through different areas of the drawing space, that is, the focus may be changed. Being a child window, MACRO VIEW is totally constrained to the format window, and in fact could be shoved off to the side of the format edit window even to the point of being clipped by its parent window.[12] The MACRO option on the format edit window's menu bar is highlighted as long as the MACRO VIEW is present. The child can be discarded by selecting MACRO again, in which case the MACRO option is no longer highlighted.

The user specifies to the editor which structure definitions are part of the format through a dialog window invoked through a menu option on the format edit window. The user is provided a list of structure definitions available for constructing the format, and definitions that are already part of the format are not presented. The user selects multiple definitions from the list box with the mouse.[13] A CANCEL option cancels any selections made during the dialog session and terminates the dialog.

After choices have been made and the user exits the dialog session, the user begins to graphically draw the format definition based on the previous selections. For each selection, the user is prompted to find a location on the screen for the definition's graphical representation. An option is also provided for skipping a selection (an ad hoc way to "undo" a selection made earlier). At the same time, grid lines as referenced on the MACRO VIEW window appear on the screen as a guide for location reference. The user indicates a location for the graphical representation by selecting a vacant grid square with the mouse. Thereupon a graphical representation of the structure, a small box inscribed with the name of the structure, appears centered on that grid square. The user proceeds with the successive prompts until all selections have been acted upon. After that the grid

Figure 7 Macro view for format editing



vanishes from the screen, and the user may revisit the structure selection dialog later. Figure 8 shows an example of a format edit window with several structure definitions represented. Notice the MACRO VIEW window also indicates the presence of structure definitions, providing more topographical information about the entire drawing space.

To define pointer relationships, the user enters a pointer definition editor "mode" through a menu option. In this mode, the user may only perform activities related to defining pointer relationships. As before, the window grid appears as an aid to the user in navigating through the drawing area to find other structures in the format. In this mode, the user selects a source structure type by using the left mouse button, and a target structure type by using the right mouse button. Visually, the source and the target turn are displayed in different colors on both the the edit and MACRO VIEW windows upon selection. To complete the operation, a dialog window appears along with a display of the source definition. Figure 9 shows an example of this with definitions DS4 as the source and DS3 as the target. The user selects the field name, here DS4WDS(0), by selecting the field in the source definition display with the mouse. The dialog focuses information about the pointer definition. For example, the name of the field selected will automatically appear in the dialog window. Addi-

Figure 8 Definitions represented during format editing

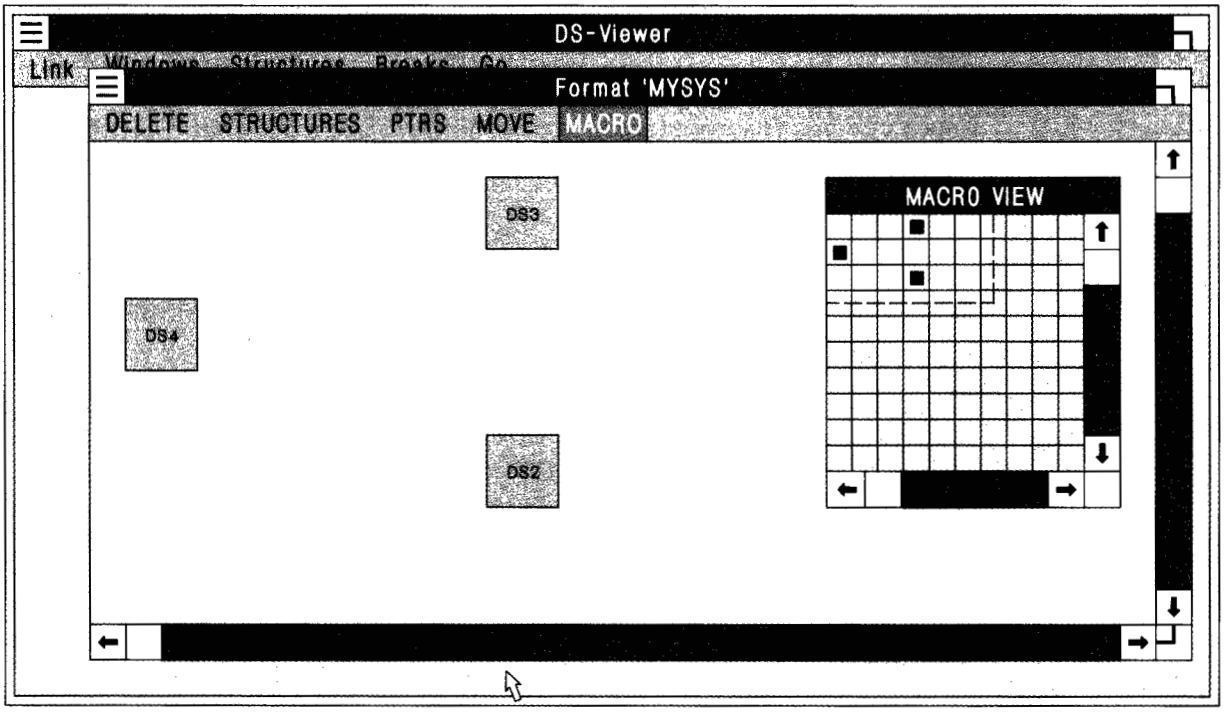


Figure 9 Pointer definition during format editing

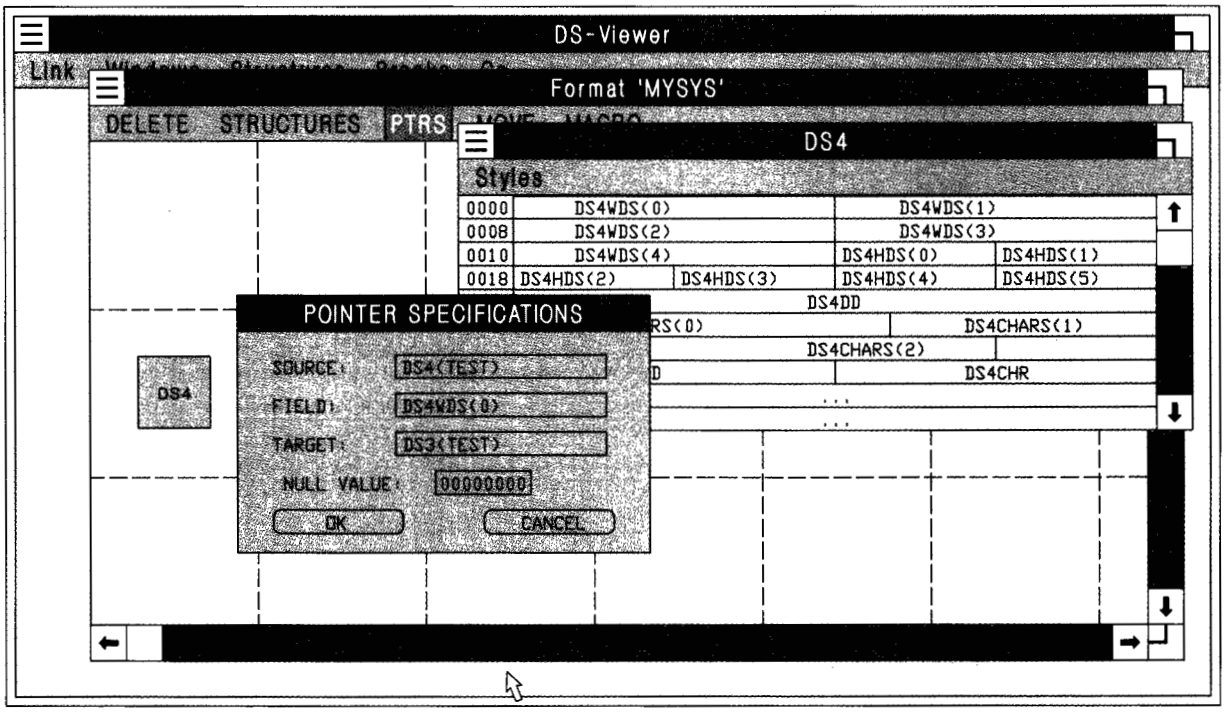
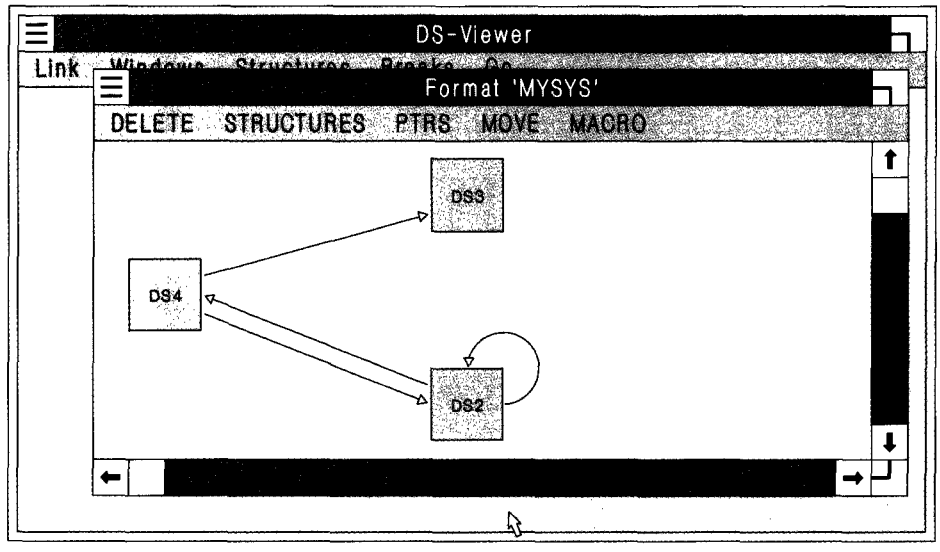


Figure 10 A fully defined format in a format editing session



tionally, the user may specify a null value for the pointer field, that is, a value indicating that the pointer points to no structure instance. When the user selects OK, the dialog vanishes and the pointer is represented as an arrow on the edit window. Optionally, the user may CANCEL this pointer specification.

When multiple pointers exist between two specific structure types, the pointers are represented by arcs stretching out on each side of the straight line connecting the two. For pointers where the source and target are identical, the pointers are represented by concentric arcs along the upper right of the structure type's image.[14] A full format is illustrated in Figure 10, and one fairly direct interpretation of this information follows. This format shows that each instance of DS4 has a pointer to a DS3 and a pointer to a DS2. The DS2s are self-referential indicating a queue of them, and the pointer from DS2 to DS4 indicates each member of the queue points back to the owner.

The format edit window provides a unique method of querying the pointer relationships and structure types. By pointing to an arc or line representing the pointer and using the left mouse button, additional information describing the pointer appears on the screen. It disappears when the button is released. This feature is illustrated in Figure 11. As the structure type's name may be truncated, a similar feature exists when the mouse is on the structure type's image and the button is pressed.

The editor also provides a move mode initiated by the MOVE option on the menu bar. In this mode the left mouse button is used to select some block and the right mouse button to indicate a new position (vacant grid square). The structure image is repositioned and all the pointer lines and arcs are adjusted to accommodate. This method was preferred over "dragging" the block to allow the user to scroll to other areas of the drawing space in order to place the block.

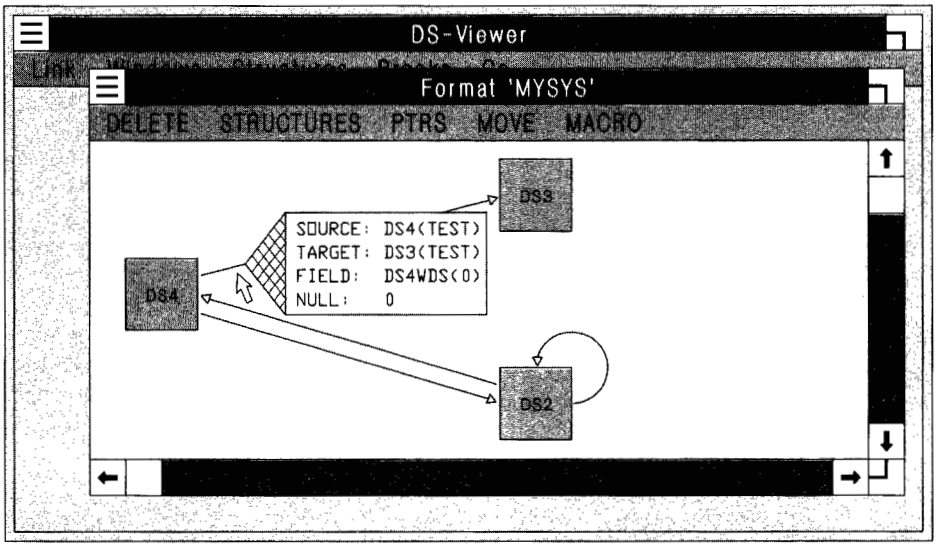
Similarly, delete mode allows selection of a pointer or structure type with the left mouse button, whereupon a dialog window appears with information about the pointer or structure type. The user may elect either to confirm this deletion or cancel it. If confirmed and a pointer is selected, the pointer line disappears and the pointer represented by it is no longer part of the format. If a structure type, its image and all pointers to and from it vanish and are deleted from the format definition.

Finally, the user many keep any number of edit sessions open during a DS-Viewer session. An edit session is ended by the usual Microsoft Windows close option on the menu of the edit window's system.

Scene editing

The process of building a scene is similar to that of building a format. Both involve drawing a picture

Figure 11 Querying a pointer definition



on a drawing space and the pictures of each are quite similar. The chief differences are that in a scene, each block image represents a unique data structure instance and each arrow an actual pointer.

The scene editing process is initiated by selecting the *Scene* option on the main window menu. The user works with a dialog window containing a list box with the names of all available scenes for this session. Since a scene is associated with a format, two prompts for a unique name for the scene and the name of an associated format are provided, as well as options to add, delete, and present (show) the scene. As mentioned before, the format will help drive the building of a scene and so is an essential part of a scene's specification. If only the scene name is given in adding a new scene, DS-Viewer will conveniently provide a second dialog window listing all the available formats from which the user may choose.

When *Present* is selected, the user is given a drawing space in which to construct the scene as shown in Figure 12. The drawing space is quite similar to the drawing space for formats except that it is much larger as depicted in MACRO VIEW window. The reason for this is that the number of data structure instances could be large, and the user could require a large drawing space. A grid square's size on the scene window is the same size as that on the format window; however, the scene window encompasses a 61-by-61 drawing area. The scroll bars on the scene window operate in the same way as the format window but with a small difference. Since the drawing space is large, if the scroll bars on the scene editor were scaled to the entire drawing space, a slight change in the scroll elevator position would radically change the focus of the window. Thus, movement of the scene editor's scroll bars is scaled to a fraction of the total drawing space. Changing the focus to remote positions of the drawing space is accomplished by the scroll bars of the MACRO VIEW window, whereafter the scene editor's scroll bars could be used for finer adjustment.

The first step in constructing a scene is to specify one or more anchors. This is initiated by selecting the ANCHOR option on the menu bar. The user is given the full list of data structure types in a list box and a prompt for a program memory location. The full list of data structure types is given as opposed to only those of the format to allow inclusion of structure instances on the scene that would add more meaning to the scene. After selecting a type and entering a location, the user is prompted to place the anchor, and a grid is drawn to aid the user in navigating through the drawing space. After placing the anchor, the grid vanishes and the instance is represented as a square image with the definition name inscribed within as shown in Figure 13.

Figure 12 Scene editing screen

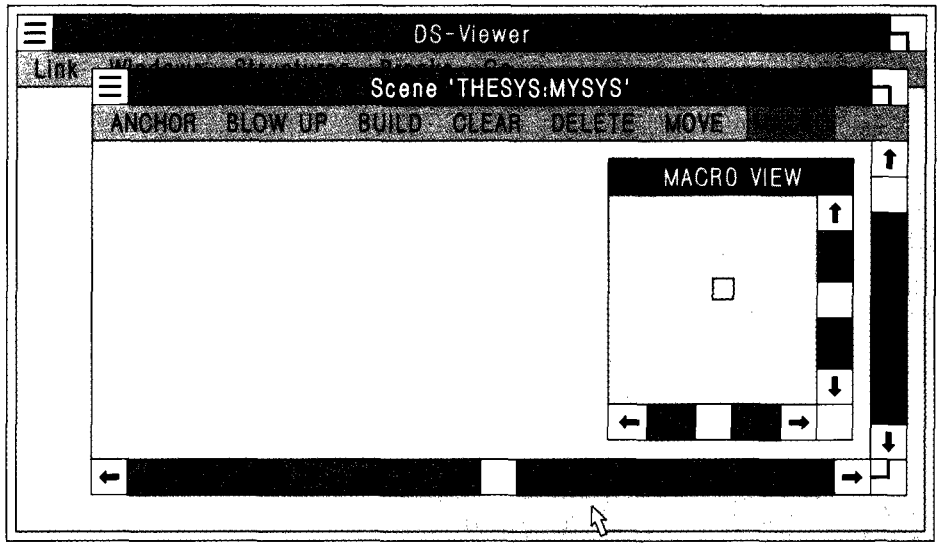
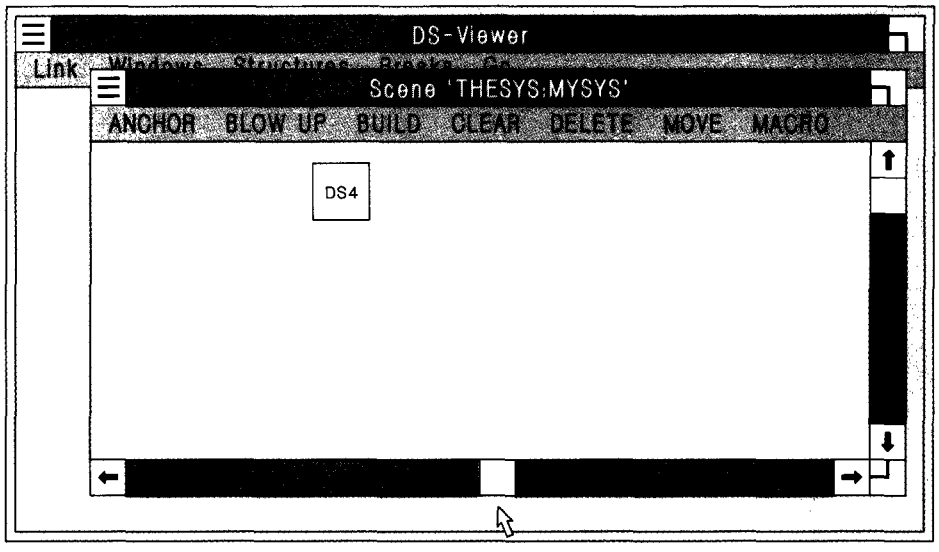


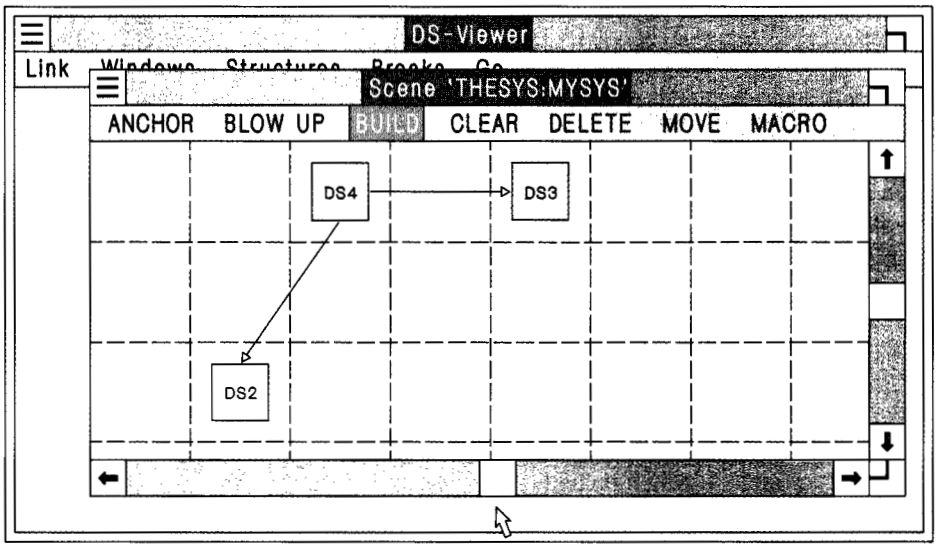
Figure 13 An unresolved data structure instance during scene editing



Any anchor or data structure instance represented on the scene drawing space is either a yellow or green block. Yellow indicates that the instance has at least one pointer (as defined in the format) that is not null and not present. That is, the non-null target instance either does not appear on the scene, or it appears on the scene but the pointer arrow does not. Conversely, green indicates that all of the instance's pointers are "resolved" in the sense implied above. A yellow data structure instance is an invitation to the user to select it for reviewing those pointers not present on the scene and to proceed to select which ones should be drawn. This process is initiated through the BUILD option on the menu bar.

The selection of BUILD actually initiates a mode of operation on the scene. The grid lines appear for navigational purposes, and the left mouse button is

Figure 14 Constructing a scene through resolving pointers



used to select an unresolved data structure instance. If the lone instance of DS4 in Figure 13 is selected in build mode, the user is given a series of prompts, one per pointer to resolve. In this case the DS3 target instance does not exist on the scene, and DS-Viewer indicates that the user must place it on the scene. As in the format editor, the user must indicate a vacant grid square with the mouse for the placement to occur. Optionally, the user may forgo the placement by selecting CANCEL and move on to the next prompt. If the target instance already exists on the scene, a different prompt is given indicating that situation and asking the user only if the arrow for the pointer should be drawn. Figure 14 shows the scene after all the pointers of DS4 have been resolved. The DS4 instance has now turned green indicating full resolution; DS3, being fully resolved, is green; and DS2 is yellow, indicating that it could be selected for resolution. Build mode will remain in effect until the user selects the now highlighted BUILD option on the menu bar.

The fully resolved scene in this example is shown in Figure 15A. The main data structure pattern present is the queue of DS2s having the DS4 as the owner. The last DS2 has a red semi-arc pointing to the right. This arc represents a pointer with a non-null but illegal value. In this case illegal means that the program data memory location indicated by the pointer has no meaning in the user work space. As in the format editor, the user may query the pointers and data structure instances graphically by pointing the mouse cursor to any image or arrow and pressing a mouse button. Figure 15B shows such a query for the illegal pointer in the final DS2.

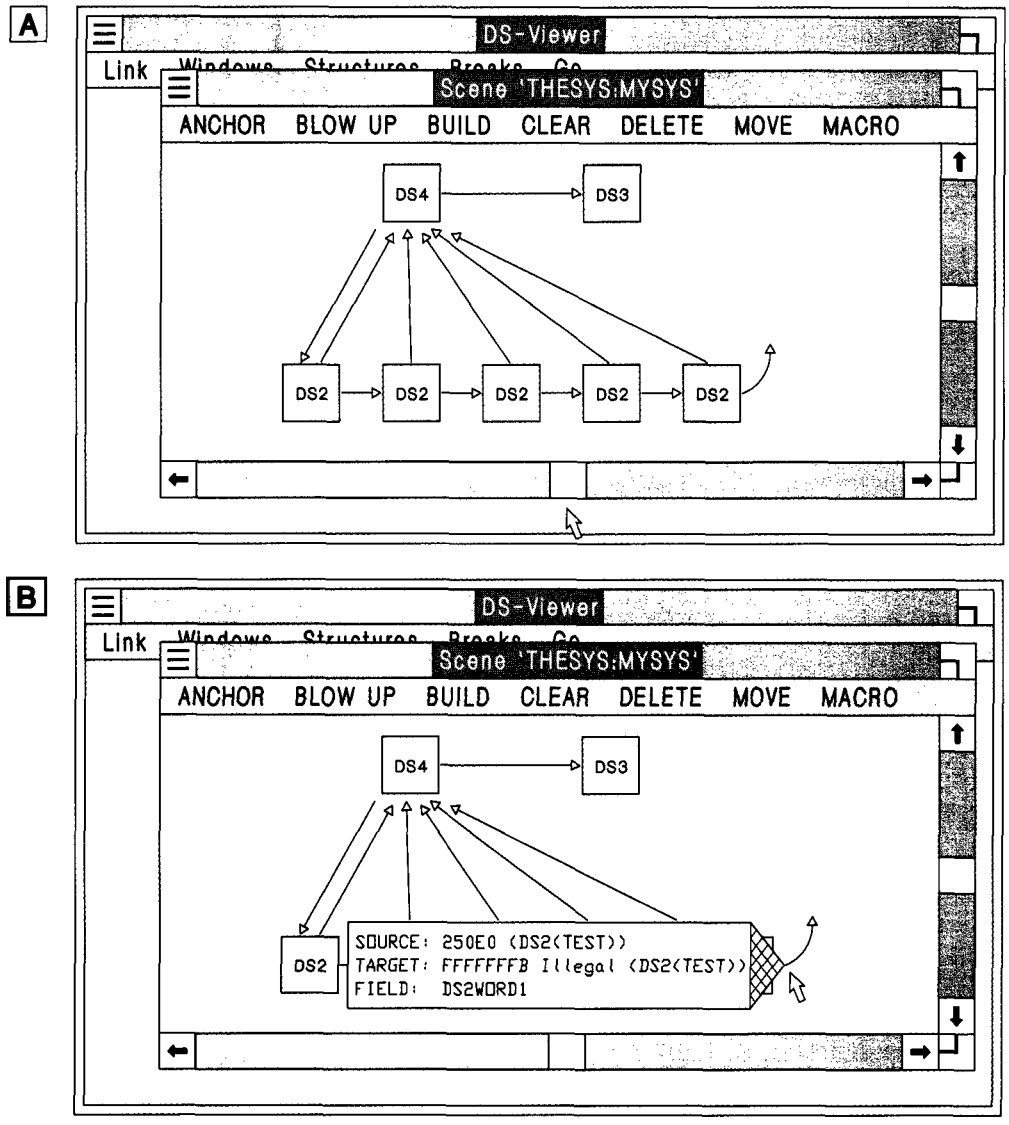
Several other menu items on the scene menu bar are of interest. BLOW_UP initiates a mode of operation whereby the user can acquire a data structure instance window of any structure image on the scene. This is done by pointing to any structure image and pressing a mouse button. With that the data structure instance window appears. CLEAR is an option to totally erase the contents of a scene, allowing a user to reconstruct the scene on a fresh background. DELETE and MOVE are similar to their counterparts on the format editor.

Summary and further work

DS-Viewer is a reasonable first step towards a practical display facility for data structures. Features such as the saving and retrieving of formats and scenes, further editing capabilities, and improvements in scene building could be easily added. An effort to extend DS-Viewer into a full-screen or even graphical debugger would be fairly straightforward, providing only some interesting side issues such as initiating scene drawing from source variables, or seeing what program variables refer to specific instances on a scene through mouse selection.

Some fundamental directions for further enhancements concern interactive searching for specific data

Figure 15 (A) Completed scene and (B) queried pointer



structure instances and displaying those results. For example, one might want to search through queues of structure instances to find specific data structure instances that meet certain criteria. One interesting approach to this problem is based on a technique for interactively browsing an entity-relationship database.[15] The DS-Viewer format could be used for graphically selecting structure types and pointer paths as the foundation of the query. Additional field value criteria would be supplied by the user via dialog windows to qualify the query. The resulting data structure instances could be given to the user one by one much as in scene building mode described earlier. However, it would be ideal to see the instance images which satisfy the query criteria in the context of the queue(s) from which they were gleaned, but the presence of the entire queue(s) would be detrimental to the scene's understandability. A subtle change in the presentation may overcome this difficulty. The searched instances could be attached to other members of the queue by dashed lines indicating indirect connection between the two along these pointers. The full impact of this technique requires further exploration.

Another area concerns the validity of a data structure scene after the program has moved to another state. Without information about how the internal data structure state has been affected with the change of program state, it is impossible to change details on a given data structure scene to reflect those changes. A sophisticated knowledge base about the application and its code as found in program understanding projects[16] may provide the kind of information that could make this possible. If a scene could be automatically updated throughout program execution, it would be possible to take successive "snapshots" of the data structure scene as it evolves and play them back as needed through the debug cycle.

Finally, another topic in this area concerns finding strategies for automatic layout of the data structure scene. When dealing with more than a trivial scene, the identification of a reasonable layout policy is not clear. Ideally, an automatic layout policy would allow the user to easily understand a very large part of a scene's structure, as well as allow the user to reasonably peruse and navigate through it. DS-Viewer avoids the layout problem by allowing the user to construct pieces of the picture as needed, but the user does all the layout work which could be very time-consuming. Some compromise might be possible if the scene could be at least partially decomposed into subcomponents built upon some higher-level data structure constructs. Then some degree of automatic picture layout might be achieved without losing clarity of the entire picture. These higher-level constructs might be visualized as some new object or icon on the scene which "opens up" into a window with more detail.

Acknowledgments

I would like to acknowledge Michel Hack and Calvin Swart for their technical aid on this work and Thomas Corbi and Linore Cleveland for their support.

Microsoft Windows is a registered trademark of Microsoft Corporation.

Cited references and notes

1. T. A. Cargill, "The Blit debugger," *SIGPlan Notices* **18,** No. 8, 190–200 (August 1983).
2. B. A. Myers, "Incense: A system for displaying data structures," *Computer Graphics* **17,** No. 3, 115–125 (July 1983).
3. S. Isoda, T. Shimomura, and Y. Ono "VIPS: A visual debugger," *IEEE Software* **4,** No. 3, 8–19 (May 1987).
4. S. P. Reiss, "Pecan: Program development systems that support multiple views," *Proceedings of the Seventh International Conference on Software Engineering*, Los Alamitos, CA; IEEE Computer Society (1984), pp. 324–333.
5. M. H. Brown and R. Sedgewick, "Techniques for algorithm animation," *IEEE Software* **2,** No. 1, 28–39 (January 1985).
6. G. P. Brown, R. T. Carling, C. F. Herot, D. A. Kramlich, and P. Souza, "Program visualization: Graphical support for software development," *Computer* **18,** No. 8, 27–35 (August 1985).
7. D. P. Pazel, *A Graphical Workstation-based Host Debugger*, Research Report RC-12871, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598 (June 23, 1987).
8. *OS/VS-DOS/VSE-VM/370 Assembler Language*, GC33-4010-4, IBM Corporation; available through IBM branch offices.
9. *Introduction to IBM System/370 to IBM Personal Computer Enhanced Connectivity Facilities*, GC23-0957-1, IBM Corporation; available through IBM branch offices.
10. Dialog window represents a feature in Microsoft Windows that allows a variety of ways to enter interactive information to the application. A large variety of input methods are provided such as list boxes, push buttons and radio buttons, and prompts called edit windows.
11. *Virtual Machine/System Product, Data Areas and Control Block Logic Volume 1 (CP)*, LY24-5220-3, IBM Corporation; available through IBM branch offices.
12. Microsoft Windows provides a standard method of moving a window. The user puts the mouse cursor over the window's caption bar, holds down a mouse button and moves an outline of the window to a new area. When the mouse button is released, the window appears in the new location. The same holds for child windows, and the clipping mentioned is a natural effect of the child window being constrained to the parent by Microsoft Windows.
13. Normally in Microsoft Windows, an item selection on a list box discards the prior selection. However, Microsoft Windows provides list boxes whereby multiple selections may be made. In that case, the user selects as usual but while doing so holds the shift key on the keyboard. The prior selections remain in effect (and highlighted) and the new selection is also in effect (and highlighted).
14. Arcs between different structure images are based on a circle computed with three points. Two of the points are the centers of the two structure images. The third point is located normal to the center of the straight line connecting the two centers. For successive pointer arcs, the distance of this point from the straight line increases by a discrete amount on each side of that line.
15. L. M. Burns, J. L. Archibald, and A. Malhotra, "A graphical entity-relationship browser," *Proceedings of the Twenty-First Annual Hawaii International Conference on Systems Science, Vol. II* (January 1988), pp. 694–704.
16. L. Cleveland, "An environment for understanding programs," *Proceedings of the Twenty-First Annual Hawaii International Conference on Systems Science, Vol. II* (January 1988), pp. 500–509.

Donald P. Pazel *IBM Research Division, T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Mr. Pazel is an advisory programmer in the Computer Sciences department at the T. J. Watson Research Center. He joined IBM in 1973 at Morris Plains, New Jersey, where he worked on the Safeguard project for the Federal Systems Division. Since joining IBM Research in 1975, Mr. Pazel has worked in the areas of operating systems, languages, databases, and language debuggers. His current work includes program understanding tools and visual

paradigms in debugging systems. In 1972, Mr. Pazel graduated maxima cum laude from LaSalle College, Philadelphia, with a B.A. in mathematics, and received an M.S. degree in mathematics from the University of Virginia in 1973. He is a member of the Mathematics Association of America and the Association for Computing Machinery.

Reprint Order No. G321-5361.