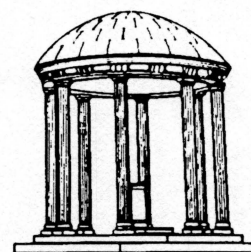# A Tool for Viewing IDL Data Structures

Ralph Cook

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175
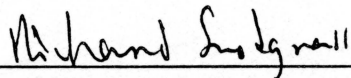
A Tool for Viewing IDL Data Structures

by

Ralph Cook

A Thesis submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.
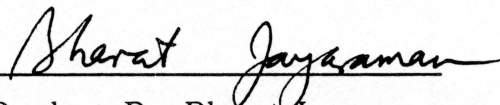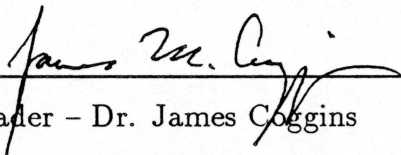
Chapel Hill

1988

Approved by:

Advisor – Dr. Richard Snodgrass

Reader – Dr. Bharat Jayaraman

Reader – Dr. James Coggins

RALPH ELLSWORTH COOK. A Tool for Viewing IDL Data Structures (Under the direction of RICHARD SNODGRASS.)

Abstract

The Interface Description Language (IDL) allows specification of complex data structures and provides run-time procedures to help use them. A useful debugging tool for interactive display of selected run-time IDL structures can be built using an existing debugger, workstation windowing software, and information generated by the IDL compiler. This tool's design can allow graphics output as an enhancement, though it is not clear a graphics version will be significantly more useful. This thesis describes the design and implementation of such a tool: IDLView's features include transparent determination of node type, display of IDL nodes, classes, sets, and sequences, and expansion of node attributes. IDLVIEW uses an instance of the intermediate representation generated by the IDL compiler to interpret the run-time instances of IDL data structures.

Acknowledgements

I would like to thank Sundar Varadarajan, Ed McKenzie, and Vikram Biyani for technical help, Pamela Manning for figure drawing, the students of the COMP 240 compiler construction course for reviewing the program, and especially Rick Snodgrass for tireless and excellent advice, editing, and overall support.

Contents

List of Figures

1   Introduction

The use of large data structures of heterogeneous data types is routine for many significant programming efforts today. Debugging environments, however, do not usually provide complete tools for displaying such data structures; the availability of graphics workstations suggest ways such displays could be implemented.

With the Interface Description Language (IDL), a programmer defines data structures made up of scalar and aggregate types, including sets and sequences [14]. The IDL compiler converts the data structure definitions into source code in a target programming language, including run-time procedures for manipulating instances of the structures. The programmer then writes programs in the target language to implement specific algorithms on the data structures described, using the IDL-generated data structures and procedures.

Displaying IDL data structure instances during debugging presents certain problems. Most available IDL tools operate only on a representation external to a program, and are therefore awkward for debugging [13]. Any standard Unix debugger requires the programmer to understand IDL implementation details he does not otherwise have to know, and to type many commands to examine an IDL instance of even moderate size.

We believe the most useful debugging tools display data structure instances at the same level at which the programmer created them. If the debugger displays data at a lower level (e.g. strings as sequences of hexadecimal values), the programmer must interpret the display as well as the behavior of his program. If the display is at a much higher level (e.g. display of an integer and an array as a 'stack'), the programmer has to interpret the display as a representation of his own structure, and may have difficulty determining whether a problem is with the model or the code.

This thesis describes IDLVIEW, a program which displays IDL data instances at the programmer's level. First we discuss research in display of large data structures. We then explain some basic IDL concepts and mechanics, and describe current problems encountered examining IDL data structures. The final sections describe the design and implementation of IDLVIEW itself, concluding with a summary of

current features and ideas for future work.

2 Previous Work

This section discusses current research work on display of large data structures, especially tools suitable for a debugging environment.

2.1 DBXTOOL and DEBUG

The current state of commercial debugging programs on mid-size computer systems is represented by Sun Microsystem's DBXTOOL [15] and Digital Equipment Corporation's VAX DEBUG [1]. Both provide features commonly expected of a "high-level" debugging environment: display of source code which scrolls with execution, debugger output to a file, conditional breakpoints, watchpoints, updated display of selected variables during execution, execution of commands when a breakpoint is encountered, and so forth. Since DBXTOOL runs on a graphics workstation, it can and does provide a mouse-and-cursor command interface; VAX DEBUG runs on a traditional 24-line 80-column terminal, and includes a complete facility for defining keyboard keys as commands and parts of commands.

Both debuggers provide display of entire arrays and structures, but differ when displaying a data structure which may have different sub-structures in different instances (In Pascal, such a structure is called a *variant record*, in C, a *union*). VAX DEBUG displays all possible variables of such a structure, DBXTOOL requires the user to specify one variant to be displayed. The VAX DEBUG approach helps the programmer figure out which variant exists, while the DBXTOOL approach can display only those fields in the variant specified.

2.2 Blit Debugger

Cargill's Blit debugger, implemented on a "bitmap terminal" controlled by a 68000 microprocessor, runs on less sophisticated equipment than DBXTOOL or DEBUG, and provides slightly less function than they do [4]. The Blit debugger does not display source code during execution; it occupies a certain portion of the limited memory of the Blit terminal during debugging, and virtual memory is not supported.

Despite the limitations, Cargill's debugger does provide some useful ideas. For

instance, it implements an interactive method for viewing selected parts of structured data types: when a structure or structure pointer is selected for display, a menu is made available which contains the structure's fields. The programmer selects one for display; if the selected field is itself a structure, the process is repeated. This menu structure reduces keystrokes when displaying nested structures.

It is interesting that Blit's author discards as "not credible" the idea that "a general purpose debugger will be able to display the arbitrary structures encountered in real programs". While we agree that the entirety of any interesting data structure is too much information for either display or comprehension, we believe we can produce useful displays based on IDL run-time information plus user guidance about what is to be displayed.

2.3 GELO System

The GELO system developed by Steven Reiss provides automatic layout of graphical representation of data structures; the layout is based on user-defined constraints such as minimum and desired size, placement of sub-objects, priority among objects, and sizing constraints [11]. The system reduces these constraints to a system of equations which define the layout. The companion APPLE editor allows user definition of the mapping between a program's data structures and GELO structures, and the PEAR editor allows him to edit a data structure via its graphical representation.

In GELO, Reiss defines basic display objects: simple shapes with enclosed text, *tile* objects with definable interior rectangular regions, *layout* objects containing nodes and arcs, and *arc* objects for connecting display items. A programmer defines how data structures are to be displayed by defining objects for each data structure type.

Reiss states that being restricted to only automatic layout of display objects is unsatisfactory: a programmer often wants a picture to look a particular way enough to spend time producing it. GELO provides the programmer no run-time control over its layout decisions. Reiss also reports that "natural editing" operations are sometimes rendered invalid because his graphics-oriented editor requires consistency with the underlying data structure, rather than allowing the programmer freedom to change the structure as he edits.

2.4 GRAB Directed-graph Browser

The GRAB program allows a user to "browse" through a directed graph structure [12]. The authors list "rules for a designer" for menu-based, graphical applications: provide "as much visual feedback as possible", make interface modes invisible, and provide the correct operations for the user. Two important features mentioned include automatic display layout (instead of user-defined layout) and stability of unchanged portions of the graph while other parts change.

The display heuristics used by GRAB might not be appropriate for a debugging display. For example, GRAB assumes no order dependence for leaves and no level dependence of nodes; this could result in closely-related "sets" of data structures appearing in widely-spaced areas on the display. It is possible the heuristics used could be modified for debugging purposes.

A larger problem is that GRAB calculates placement of every node in the graph before any display, making multiple passes over the graph to refine node placement. This would be too slow for a debugging environment on current workstations, and in fact (its authors felt) too slow for its own browsing application. Other problems reported were unnecessary edge-crossings allowed by their layout algorithm, and the difficulty of viewing large numbers of near-parallel neighboring edges.

2.5 Program Visualization System

The Program Visualization (PV) System at Computer Corporation of America provides "moving pictures" of data structures during program execution [2, 3, 6]. It includes tools for creating pictures based on the algorithms and data structures in a program, both via intrusive (i.e. changing of source code) and non-intrusive methods of display. The authors (and the authors of GRAB) speak of the "challenge (of) the graphical depiction of very large data structures" and of "opening the side of the machine and watching the program run".

It seems to us, however, that viewing a running program is not the same as finding bugs, regardless of the display. Perhaps that is why many examples given of this feature are for algorithm instruction rather than debugging; some tools even require modification of source code to produce the displays. For production systems, this means the code has to be changed *after* debugging. Changed code is undebugged code by definition.

In fact, the pictures produced this way are often at a level higher than the source code: they show stacks, queues, etc. Certainly a programmer unfamiliar with a program can learn from a graphical depiction of its algorithm, but it is not part of the debugging process. Bugs are found at the level at which the programmer created the code.

We feel, therefore, that the proper level for display of information to the programmer is the level of the source code being debugged. This is demonstrated by the far greater ease with which programmers use programs like DBXTOOL, which step one program source line at a time (instead of one machine instruction like older debuggers) and provide examination of variables by name and as declared (instead of the hexadecimal or octal dumps by address found in older debuggers).

2.6 CaseDE Environment

Another idea in this area is to provide commonly used higher-level data structures, with automated displays which are part of the debugging tool and/or the translator run-time. Mateti and Radack describe a "hint language" for their CaseDE environment [8]. In this proposed system, the programmer specifies:

- display of records and pointers as stacks, trees, or lists,
- connection between variables (such as arrays and indices),
- layout methods of certain structures, such as whether elements of an array are displayed horizontally or vertically), and
- which fields of a record to display.

They do not go into more detail, but we can visualize a system with a predefined "stack" display allowing the programmer to name an array and an integer which model a stack and displaying them as a picture of a stack. The addition of data types and their operations is similar to the Modula and Ada "package" constructs, but proposed for a different part of the program development cycle.

2.7 Incense Debugger

Myers' debugger Incense performs automatic layout of data structure displays, though it also provides user-defined displays [9]. It uses pre-defined layouts for placing structures referenced by pointer from other structures, with a simple, attractive

method for drawing edges between them. Myers identifies the user-defined display structures and location of such referents as the "most difficult aspects of INCENSE". His work contains examples of how such a system might be implemented, though INCENSE's limitations seem severe for a production system – only global variables can be accessed by INCENSE, there are memory limitation problems for storing the symbol table, variant records and some other data types are not supported, and there is no automatic garbage collection. INCENSE runs on a machine with no virtual memory system and less than a megabyte of memory – a small computer compared to ones used in most studies discussed here.

Included among Myers' goals for a debugging system were the following:

- reduce the volume of data required from the user,
- adjust output quantity to human capacity,
- output only completely processed results,
- generate pictures automatically,
- require no modification of the source program, and
- execute quickly.

Myers reports success at these goals, though he says the execution speed needs to and will increase in future versions.

2.8 VIPS Visual Debugger

The VIPS visual debugger has almost all the features mentioned so far [7]. As a debugger implemented in and for an Ada programming environment, VIPS displays data flow, source code, execution-stack windows, and user-defined structures. Subrecords of records are displayed in increasingly smaller fonts as remaining room on the display decreases. Each time a subprogram is invoked, a subwindow is created for it and any packages declared in it. User-defined displays are written in the Figure Description Language, a subset of Ada.

VIPS' authors maintain that debugging tools such as BLIT, INCENSE, and the PV and Pecan debuggers "are insufficient for debugging programs because they represent only a part of program execution behavior". To the extent that these and other, lesser tools are in fact used for debugging, the statement seems exaggerated;

we also feel that this judgement of "sufficiency" may be erroneous. We have not used VIPS, but it appears from the examples given that it displays more information than programmers may want.

For example, VIPS adds a sub-window to a part of the display each time a subprogram is invoked; the programmer therefore has a window for each level of the current call depth (plus one for each package implemented by each subprogram called). It is rare that a programmer wants a display of the entire call stack, possibly because few bugs are found there. We maintain that displays that are not likely to be useful should neither compete for the programmer's attention by appearing automatically nor consume processing power being generated. VIPS also recalculates the positions of all nodes in the data window when any node is added to it, implying either a more powerful machine or more patient programmers than we believe IDLVIEW is likely to have.

2.9 Summary

Problems addressed in this research area include choice of information to display, execution speed, and the layout of graphical representation of data structures. Although various user interfaces are presented, there is nothing radically different from existing tools; we conclude that researchers are, in general, satisfied with interfaces currently available. We have also seen that some of the work on displays is not strictly applicable to debugging.

Among the principles mentioned in this section that we either adopted or had considered on our own:

Allow the user to select what is displayed. In order to best use screen space and computing power, the user selects each item to be displayed. IDLVIEW does not expend processing time or screen space on items which are not requested.

Display data instances at the source code level. IDLVIEW displays IDL instances in the same format the programmer uses them. This format is generated automatically and familiar to all IDL programmers; moreover, it is the level at which the programmer finds bugs.

Do not require changes to the user's code. In a debugging environment, such a requirement is unacceptable.

We decided not to attempt a complex graphical display in this version; we had planned to do so in a future version of IDLVIEW, and will now evaluate whether it is necessary. It is no longer clear that a graphics version would provide enough extra information to be worthwhile; we may be able to provide the same additional function without the use of graphics.

3   Some IDL Basics

This section provides enough information to orient a programmer to the Interface Description Language (IDL); it emphasizes those aspects of the language which are involved in run-time display. Though it may be skipped by a reader already conversant with IDL, it may also alert such a reader to issues covered later. The full description of IDL is given in *The Interface Description Language: Definition and Use* [14].

A programmer uses IDL to describe data structures made up of the following types:

scalar Integer, String, Rational, or Boolean.

node An aggregate of *attributes*, where each attribute may be any IDL type listed here. Similar to C *struct* and Pascal *record*.

class An aggregate of *attributes*, where each attribute may be any IDL type listed here; unlike those of a node, the attributes for different instances of a class may vary. Similar to C *union* and Pascal *variant record*.

sequence Zero or more of another IDL type; duplicate members are allowed and order is significant.

set Zero or more of another IDL type; duplicate members are not allowed and order is not significant.

private User-defined types with user-defined operations and implementations.

The difference between node and class deserves some elaboration. The following IDL fragment declares the class `Customer` to contain the subclasses `Commercial_c` and `Government_c`; `Customer` has the attributes `Name` (a `String`) and `Location` (a node type named `Address`); further, a `Government_c` node contains the attribute `Agency_name`, and a `Commercial_c` node contains no attributes specific to its type.

```
Customer ::= Commercial_c | Government_c;

Customer => Name      : String,
```

```
                    Location : Address;
    Government_c => Agency_name : String;
    Commercial_c => ;
```

Customer is an IDL class (since it has subclasses), and **Commercial_c** and **Government_c** are nodes (since they have no subclasses). An instance of **Government_c** has the attributes **Name**, **Location**, and **Agency_name**; one of **Commercial_c** has **Name** and **Location**. (Although this example's class has only two nodes as subclasses, in general a class may have any number of classes, nodes, or both as subclasses.)

An IDL programmer can declare an attribute as **Seq of Customer** (which is a class), but he must often deal with an individual element of that sequence as a node and not a class. For instance, he can create a new instance of a node which is a member of a class, but cannot create a new instance of a class directly; i.e., he can create a new **Commercial_c** or a new **Government_c**, but cannot create a new **Customer**. IDL provides a function which returns the node type for situations in which a program must determine the node type of a class instance.

IDL specifications are organized into *structures*, where each structure has an associated *root node*. All elements declared within the structure must be reachable from the root node, i.e. every declared element must be contained, directly or indirectly, within the root node. Often the root node has only one attribute which is a sequence of a class, and the rest of the IDL specification describes the class.

An IDL programmer defines one or more *processes* in his IDL specification; each process contains one or more *ports*. A port is declared for either input or output and is associated with one structure. For the case mentioned above, where the root node of a structure is a sequence of classes, the structure read by the port is analogous to a traditional file, and the sequence of classes analogous to a series of records in the file. The process is analogous to a traditional program: it reads in data, performs some operations on it, and writes it out in a different form.

A given node or class may have different attributes in different structures; a node may be declared in one structure, then *inherit* those attributes in a second structure which declares additional attributes for it. Since ports are associated with structures, one may think of a "view" of a node as being the the set of attributes declared for that node within the structure associated with a given port.

One way IDL programs utilize a node with different attributes for different structures is for the common case of reading data, adding information to it, and writing it out with the added information. In semantic analysis, for instance, a compiler can read in nodes from syntactic analysis via a port defined for a syntactic structure, add semantic information to the nodes with attributes defined in a semantic structure, and write out the semantic version through a port associated with the augmented structure. An advantage of IDL is that the semantic definitions can inherit the syntactic declarations, instead of re-declaring attributes used in both phases.

At runtime, nodes are represented internally by the union of all declared attributes; this allows the use of the node to change without changing its runtime representation. This version of the node's structure is called the *invariant* for that node; it should be noted that the invariant does not necessarily have the same structure as any one port's "view" of that node.

4 The Problem

This section describes the tools available for displaying instances of IDL structures before IDLVIEW was available, and what problems there are with using them.

4.1 Tools for Viewing the AER

The ASCII External Representation (AER) represents IDL instances as sequences of ASCII characters; sets and sequences are enclosed in identifying pairs of symbols, and structures used in more than one place are labelled at one instance and referred to by label thereafter. Snodgrass describes the AER and some programs designed for viewing it [13]. Three of those he mentions are reviewed in this section – IDLFORMAT, TREEWALK, and TREEPR.

The AER can be examined with a standard text editor. However, the format is intended primarily for communicating IDL structure instances between programs, not for reading by programmers. The basic problems for run-time viewing are:

The size of an AER file

Since useful IDL structures normally have thousands of nodes and their order in an AER file is not designed for the programmer to read, the user has much confusing text to search through.

Label and reference usage

Labels are a way of referring to a node without having to duplicate the information in it. The location of references, however, is fixed by the IDL routines which write the AER file, and are inconvenient for a programmer to resolve.

The AER format

The AER is not designed to be read; attribute/value pairs tend to be on their own lines, but no lines are indented. When an attribute is a sequence, for example, AER lists its elements before the next attribute for that node. This is convenient if the programmer happens to want to look at that particular sequence in relation to that node, but makes it hard to see the rest of the

attributes in the node containing the sequence. This is especially true since the only delineation for sequences are the enclosing angle bracket characters, and tracking a specific pair of them through many text lines is difficult.

IDLFORMAT improves on this by formatting an AER file. All attributes of a node are indented to the same level, as are elements of sets and sequences. Sequences of strings, which normally occupy lines of text too long to fit on a screen, are broken into viewable lines and indented.

The result is workable for small IDL instances, but for viewing significant structures it does not give the user enough help. For example, the AER file lists members of a sequence declared for an attribute within the node containing the attribute; to skip uninteresting sequences, the reader must must keep track of indentation over many lines of text. Also, the indentation level gets so deep that the text cannot fit on a screen (or even a page).

TREEWALK provides interactive viewing of an IDL instance, one node and tree level at a time. Commands are provided to "walk" the directed graph which is the IDL data. The labelling used in the AER file to reduce duplication is put to use in TREEWALK to reduce the amount of information the programmer must view at one time. For example, a sequence is presented as a series of labels rather than as a series of complete nodes; the programmer can then choose which element of the sequence he wishes to view, rather than having to find one in a large display.

TREEWALK works reasonably well for a programmer interested in only a small part of the entire instance. Like AER and IDLFORMAT, however, it does not show the directed-graph nature of the instance.

TREEPR (pronounced "Tree Print") reads an AER file and prints the IDL instance represented as a directed graph. An attribute which is a sequence has lines drawn to the elements of that sequence; an attribute which is a node has a line drawn to the representation of that node. To address the problem of large instances, TREEPR supports options for either producing output which can be taped together into one large diagram or for producing output which fits on one page at a time. It also supports dot-matrix or laser-printer output. There is no option for selecting which portion of an AER file to print.

None of these tools show the values of IDL structures while a program is in progress (without changing code to produce output at the point where viewing is

desired); all of them operate from AER, which is external output from an IDL program.

4.2 Using DBXTOOL to View IDL Instances

This section describes what a programmer must know and do to examine IDL instances using the Sun Microsystems DBXTOOL debugger, including the overall DBXTOOL environment, some necessary facts about IDL run-time representations, and an illustrative example of a DBXTOOL session.

4.2.1 The DBXTOOL Environment

When DBXTOOL is invoked, it creates a display window on the Sun screen with four subwindows:

1. A window which displays source code; upon reaching a breakpoint, the displayed source changes to include the line at which the breakpoint was set.

2. An area of on-screen *buttons* containing DBXTOOL commands. Common commands such as **print**, **run**, **step**, etc. appear here. A programmer executes the command by positioning the cursor on a button and clicking the right-hand mouse button.

3. A window for display of commands and their resulting output.

4. A window for variables which are updated each time a breakpoint is reached.

Items such as variables, lines, and filenames are selected by positioning the cursor on them in the source window and clicking the right-hand button. Certain DBXTOOL commands take arguments; the on-screen buttons use the currently selected item as an argument. For example, the programmer can select a source line, then click the on-screen button **stop at** to set a breakpoint at that source line. DBXTOOL determines how to treat the selection from the command; e.g. **stop at** interprets the current selection as a line number, but **print** (which prints a variable's value) interprets the current selection as a variable name.

The commands **print** and **print *** display variable values; the latter, if its argument is a pointer, displays the item pointed to.

For names which cannot be conveniently selected, the programmer can use the conventional keyboard interface. To do so he must position the cursor in the command subwindow; his input and the resulting output appear there.

To reference the member of a C struct from a pointer variable, the C language requires a dash and an angle bracket between them (making an "arrow", e.g. ptr->member). To save keystrokes for the programmer, DBXTOOL allows use of a period in this situation (ptr.member); this is the same syntax as for separating a struct variable from a member field (var.member). When DBXTOOL echoes the variable being displayed, it displays periods and arrows the way the C language requires them.

Standard C variables are displayed in expected formats: ints in decimal, floats in decimal with fractions, pointers in hexadecimal. If a variable is declared as a pointer to a char, DBXTOOL displays both its address and the string found at that address.

C structs are displayed in their entirety; that is, DBXTOOL displays all fields. DBXTOOL even displays structs within structs, but displays only the addresses of unions and of pointers to structs.

4.2.2   Some IDL Run-Time Representation Facts

This subsection describes most of the details a programmer needs to know about IDL's implementation of data structures to use DBXTOOL to display them.

IDL Integers, Rationals, and Strings are represented by C ints, floats, and *chars, respectively. Booleans are represented by char variables, with "true" and "false" represented by their normal C meanings.

IDL nodes are represented by C structs, and IDL classes are represented by C unions. The union for a class has one member for each subclass represented; that member is a pointer to a struct which contains a member for each field of the subclass.

An IDL class, in fact, has no IDL run-time representation; a variable declared as a class in the program is a node at run-time. Each node struct has an internal struct named IDLhidden, which contains the field TypeID. Each node type of any IDL specification has a unique value for TypeID; both the IDL run-time and IDLVIEW use the value to determine a node's type.

```
Structure customers Root customer_list Is

customer_list => list : Seq Of customer;

customer ::= commercial_customer | government_customer;

customer => name            : String,
            location        : address,
            customer_number : Integer,
            balance         : Rational,
            contacts        : Set of String;

address =>   city : String,
             state : state_code;
state_code ::= NC | SC;
state_code => ;

commercial_customer =>  types : Set Of industry_code;

industry_code => code : integer;
```

Figure 1: An IDL Specification Fragment

4.2.3 An Example DBXTOOL Session

Consider Figure 1, an IDL specification fragment. For this illustration, let us assume a programmer is debugging IDL code based on a specification containing this fragment, that he has declared a variable thisCL of type customer_list, and that he has reached a breakpoint where he wants to examine this variable.

Since customer_list is a node, thisCL is a pointer to a C struct which contains the member list. To look at the value of list, the programmer needs to know that sequences are represented by linked lists of pointers, and that each pointer points to a *cell* containing (1) a pointer to the next element and (2) the element's value. (There are other representations of sequences, but we will only consider this one.) The programmer can view the cell with the following command:

```
(dbxtool) print *thisCL.list
*thisCL->list = {
        next = 0x20ad8
        value = <union>
}
```

DBXTOOL requires the programmer to name the member of the union **value** to be displayed; he cannot view the **value** field without naming a member, nor does DBXTOOL have any way of telling the programmer what members exist. Since the component element of the sequence is **customer**, any member of the sequence could be either a **commercial_customer** or a **government_customer**. To determine the type, the programmer needs to view the IDL run-time internal variable which contains the type identifier. For this he enters:

```
(dbxtool) print thisCL.list.value.IDLclassCommon.IDLhidden
thisCL->list->value.IDLclassCommon->IDLhidden = {
        TypeID  = 8
        Touched = 0
        Shared  = 0
}
```

Now the programmer knows the **TypeID** of his sequence member. The definition of this number as a certain node type resides in the "include" file generated when the IDL specification was compiled. The statement in that file will be of the form:

```
#define K<node name>  8
```

The following Unix command will produce a list of all such constants, which the programmer might print and keep around for just this purpose:

```
grep 'define K' <include file name>
```

Looking at the resulting list, the programmer could see that **8** is the definition for the constant named **commercial_customer**. The union member for this variant is therefore named **Vcommercial_customer**, and the following command displays its value:

```
(dbxtool) print *thisCL->list.value.Vcommercial_customer
*thisCL->list->value.Vcommercial_customer = {
    IDLhidden                = {
                TypeID  = 8
                Touched = 0
                Shared  = 0
        }
    name                = 0x20004 "Itty-Bitty Machine, Inc."
    location            = 0x2a504
    customer_number     = 1
    balance             = 11.109999656677246
    contacts            = 0x2a51c
    types               = (nil)
}
```

Now the programmer has some useful information. The values for the name, customer_number, and balance fields all appear in this display.

To display the value for the location field, however, the programmer needs to enter one command to see the value of the location attribute, and another to determine the value of the state field.

```
(dbxtool) print *thisCL.list.value.Vcommercial_customer. \
location
*thisCL->list->value.Vcommercial_customer->location = {
    IDLhidden = {
                TypeID  = 6
                Touched = 0
                Shared  = 0
    }
    city        = 0x2001d "Chapel Hill"
    state       = <union>
}
(dbxtool) print \
*thisCL.list.value.Vcommercial_customer. \
location.state.IDLclassCommon
*thisCL->list->value.Vcommercial_customer-> \
location->state.IDLclassCommon = {
    IDLhidden = {
                TypeID  = 2
                Touched = 0
                Shared  = 0
    }
}
```

The programmer can then look at the list generated with the grep command, mentioned above, and determine that a TypeID of 2 indicates a state_code of NC.

Since list is declared to be a sequence of customer, the location field of the value member of the list must be specified as a member of a union identified with the node type of that member of the union (in this case, with Vcommercial_custo-mer).

Printing out an entire set or sequence requires one command for each member. The following set of commands and responses shows how the programmer views the attribute contacts, which is declared as a Set of String.

```
(dbxtool) print *thiscustomer.Vfederal_customer.contacts
*thiscustomer.Vfederal_customer->contacts = {
        next  = 0x2a6b4
        value = 0x200cf "Mr. Harpo"
```

```
        }
(dbxtool) print *thiscustomer.Vfederal_customer.contacts \
.next
*thiscustomer.Vfederal_customer->contacts->next = {
        next  = 0x2a6c0
        value = 0x200d9 "Mr. Chico"
        }
(dbxtool) print *thiscustomer.Vfederal_customer.contacts \
.next.next
*thiscustomer.Vfederal_customer->contacts->next->next = {
        next  = 0x2a6cc
        value = 0x200e3 "Mr. Groucho"
        }
(dbxtool) print *thiscustomer.Vfederal_customer.contacts \
.next.next.next
*thiscustomer.Vfederal_customer->contacts \
->next->next->next = {
        next  = (nil)
        value = 0x200f9 ""
        }
```

The last element shows two special cases: the value of (nil) for next indicates
that there are no more elements in this set, and the display for the value attribute
indicates that its string is empty.

Note how the programmer must retype much of the "variable name" for each
command. In this example, the procedure is made simpler by application to a set
of strings (which DBXTOOL displays directly). If the component elements of a set
are nodes (or even worse, classes), the programmer needs an additional command
to view each member, since DBXTOOL automatically displays only the hexadecimal
value of a pointer variable.

IDLVIEW was written to eliminate the tedium required to view run-time IDL
data. To use DBXTOOL on IDL code, the programmer must type many commands
to view even part of his overall structure, know how each of the IDL types are
represented, and determine the node type of class attributes by searching a generated
file for their type constants.

5   IDLView External Design

A program's external design is a combination of the program requirements and the constraints placed on the program by hardware and other resources. This section describes the external design of IDLVIEW; the User's Guide in Appendix A describes its use in greater detail.

5.1   Hardware and Base Software Decisions

IDLVIEW was planned as an interactive graphics tool, with IDL structure elements appearing in boxes connected by lines to other boxes and the directed-graph nature of the structure instance appearing as part of the display. IDLVIEW currently performs non-graphics output, but does have workstation features such as a separate output window and a cursor-and-mouse user interface.

The single biggest external design requirement was use of a Sun workstation. The actual requirement was that the system support both graphics output and IDL, but the choice was clear. The University of North Carolina at Chapel Hill has over 75 Sun workstations, most of them available to Computer Science students; there was no other reasonable choice.

Using an existing debugging program was another primary consideration; doing so provided many features which IDLVIEW did not have to implement, such as use of compiler symbol table information, setting of breakpoints, displaying source files during debugging, display of scalar variables, and display of run-time stack information. Since the hardware choice was clear, the debugger choice was also indicated; DBX (or its SunTools implementation DBXTOOL) is the most advanced and most commonly used debugger on Sun workstations.

Moreover, DBX contains an interfacing feature making it especially suitable for extension. This is the `call` command; it allows the programmer to execute a subprogram with an interactive command, passing the address of a program variable as an argument. This single feature might have been sufficient to cause the choice of DBX over another debugger, for with it we could write our own subprograms to execute commands to interface with our tool without modifying the debugger itself.

5.2   Window Management Software

With clear choices for the hardware and debugger, we next considered software packages for window management. At the time, there were two to choose from: SunTools and X Windows.

X Windows is the newer of the two systems, with continuing development involving MIT and Digital Equipment Corporation [5]. It features portability across different hardware and "transparent" operation across networks (with programs managing different parts of a window application running on different machines). Unfortunately for IDLVIEW's immediate development, only a low-level library of routines was available at the time the choice had to be made, and not all of that was installed and running at our location. Some features we wanted to use, such as pop-up panels, scrollbars, and automatic refresh on window exposure, had to be implemented by the application rather than being managed by the windowing software. We were sure that X would have these features eventually, but it was not certain how long it would be before a stable set of such tools were available or whether the first set available would become a standard.

The SunView (formerly SunWindows) library, on which the SunTools environment is based, did not have some of these problems. Its use was well established and its user interface library in place. Its features helped make IDLVIEW's user interface similar to other SunTools programs. The following features were all supported by SunView and used by IDLVIEW: scrollbars, text subwindows, pop-up panels, cursor and mouse positioning, window exposing, hiding, resizing, and closing, static and dynamic menu generation, menu item selection, different menus for frame and window background, pull-right menus, and a different cursor and icon for the display window. An additional advantage was that use of SunView enabled use of DBXTOOL, the SunTools version of DBX, which was also in widespread use on Sun systems.

We chose SunTools over X Windows, then, knowing that we might abandon the older system later. We were reluctant to risk working out the problems of a newer system which lacked features we wanted, preferring to spend our time on other problems.

5.3 Overall Goals and Program Structure

While developing both external and internal design for IDLVIEW, we kept three overall goals in mind: execution speed, expandability to a graphics version, and avoiding changes to the user's code.

The speed of an interactive tool is an obvious issue, but how fast is fast enough is a subjective judgement. In general, we feel that a programmer is willing to wait longer for more and/or better information (up to a point); however, a debugger is limited in the level of information it can display. A good debugger answers questions like "what are the values of variables at this point in the running program?" and cannot yet answer ones like "is there a bug in this routine?"

In debugging, the programmer must interpret the answers the debugger gives. We feel this interpretation is easiest (as we already mentioned) when information is presented at the same level as the source code; in other words, a debugger's job is to present run-time information so that compile-time constructs can be made to work correctly. If the programmer writes in C, the debugger should present information at the level of C source code – decimal values for integers and floating point numbers, strings displayed within double quotes (as DBXTOOL does), **structs** displayed as series of fields. Likewise, if the programmer writes in IDL, he should be able to debug by viewing IDL structures in much the same way as he programs with them.

Since a programmer typically performs many debugger actions to produce the information necessary to find bugs, he has good reason to expect each action to complete quickly. This and the interactive nature of symbolic debugging make debuggers more sensitive to speed constraints than some other interactive programs. Elapsed time spent calculating additional display must be considered carefully, because a programmer will get impatient while debugging.

Typical IDL structures are so large that significant time could be spent just calculating their displays. We decided to limit IDLVIEW's display to items selected by the programmer, rather than generating displays he might never want to see. We also made programmer efficiency a goal of our user interface, to reduce elapsed time for input as well as output.

The future graphics version gave us the goal of structuring the design (and the code) so that changing to graphics output would involve minimal change to the rest of the program. We purposely chose a window management package with graphics

capability, so it was up to us to ensure that the graphics version development would be an upgrade instead of a rewrite.

Our goal of requiring no changes to user code resulted from our conviction that a programmer should not have to change code after it is debugged. Occasionally, limitations brought about by IDL and DBXTOOL could have been alleviated by requiring such changes, but we resisted the temptation.

These and other considerations led us to a program structure where library routines, collectively called IDLVLIB, are linked into the user's code and a separate Unix process, called IDLVIEW, controls the display window. This offers the following advantages:

- Most subprogram names and global variables are separate from the user's code. Since IDLVIEW must have access to the user's memory space, the alternative was to link all of IDLVIEW's code with the user's program. Although name conflicts are fairly simply resolved by requiring the user to change his names, it would be a major annoyance for the programmer to have to do even once.

- All window management restrictions are removed from the user's code. For instance, SunView programs are not allowed to trap certain Unix signals – the window management software depends on trapping those signals itself. If all of IDLVIEW was linked with the user's program and the user needed to handle these Unix signals, it is not clear how the conflict could be resolved.

- A separate process for window management allows the possibility of having the window manager and debugged code run on different machines. Towards this end, we used Unix sockets for interprocess communication (since sockets can operate across a Unix network, while Unix pipes cannot).

- A future version of IDLVIEW may be able to take advantage of the separate processes to allow debugging commands and display calculations to proceed simultaneously.

- Since the majority of IDLVIEW code is in a different address space than the user's program, it is harder for either to corrupt the other's address space (the user's code, after all, is being debugged).

- Both memory and disk requirements are reduced; IDLVLIB is 20 kilobytes, IDLVIEW is over a megabyte; if IDLVIEW were all linked with the user's
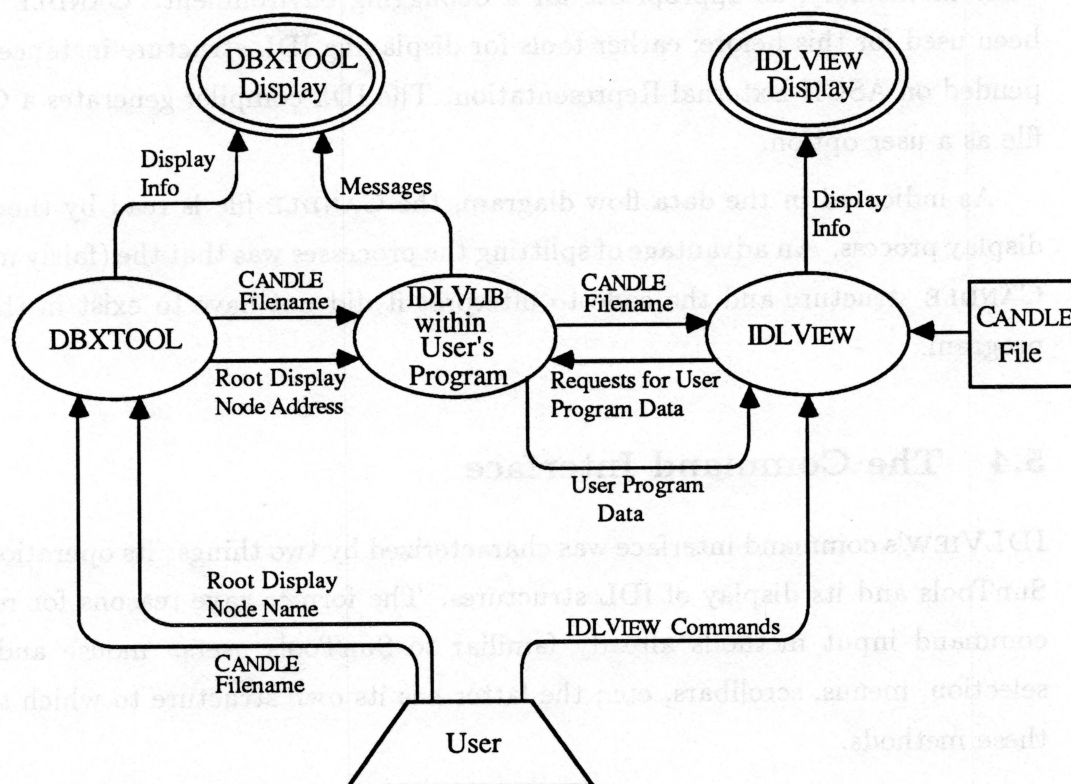
Figure 2: Overall IDLVIEW Data Flow Diagram

code, each process and image file being debugged on any one system would be that much larger.

Figure 2 is an overall IDLVIEW data flow diagram. Ovals represent *processes*, which transform data; boxes represent *data stores*, which are storage places for data; lines represent flow of data. Trapezoids represent user input, and double ovals represent displayed output. Although a process is often a program or a subroutine, it could be multiple programs or parts of different programs or groups of subroutines; although a data store is often a file, it could be an internal data structure, parts of different data structures, or a single program variable. A data flow diagram's purpose is to show how data changes and moves within a system; the system shown is presumed to "already be running", and no chronology or logic is indicated by the symbol placement.

The CANDLE (Common Attributed Notation for the Description LanguagE) representation of the user's IDL structures enables IDLVIEW to interpret IDL struc-

tures in memory, as appropriate for a debugging environment. CANDLE had not been used for this before; earlier tools for displaying IDL structure instances all depended on ASCII External Representation. The IDL compiler generates a CANDLE file as a user option.

As indicated in the data flow diagram, the CANDLE file is read by the window display process. An advantage of splitting the processes was that the (fairly involved) CANDLE structure and the code to interpret it did not have to exist in the user's program.

5.4 The Command Interface

IDLVIEW's command interface was characterized by two things: its operation under SunTools and its display of IDL structures. The former gave reasons for retaining command input methods already familiar to SunTools users: mouse and cursor selection, menus, scrollbars, etc.; the latter has its own structure to which to apply these methods.

The decision to use DBXTOOL as a base has already been discussed. Two commands were added: (1) IDLView, creating and initializing an IDLVIEW window, and (2) shownod, specifying a node in the target program for display. DBXTOOL's call command accepts names of program variables as arguments, translating them to their addresses in the user's address space.

DBXTOOL's interface also allows definition of on-screen *buttons* for command execution. The programmer can define a button to execute a command, and optionally have the command receive as its argument a variable selected in the source code window. This interface minimizes typing, which in turn speeds debugging. IDLVIEW's commands have button equivalents, with shownod[1] receiving a program variable as an argument.

After the programmer has begun execution of his code, he uses the IDLVIEW command to create the display window. It prompts for the name of the CANDLE file (in the DBXTOOL window), reads that file, and DBXTOOL resumes normal operations.

When the programmer wants to display a node, he uses the shownod command to begin an IDLVIEW *session*. During the session, the programmer uses the menu

---

[1]The text contained within buttons in DBXTOOL is limited to seven characters, otherwise we would have spelled this command out as shownode.

options of the IDLVIEW display window to examine selected parts of the IDL structure as they are at that program breakpoint. During the session, DBXTOOL does not respond to commands; another option ends the session and returns DBXTOOL to its normal command mode. The programmer may use as many sessions as he likes during one invocation of his program under DBXTOOL; the previous session output remains available.

The seeming limitation of suspending DBXTOOL operations during a session actually allows much of IDLVIEW's usefulness. During a session, subprograms within IDLVLIB communicate via Unix socket with IDLVIEW itself. When the user requests display of another portion of the IDL structure, the display program sends messages requesting the necessary information from the user's program, and the IDLVLIB routines return it. DBXTOOL is suspended because the IDLVLIB routines which control half the communications are executing.

An alternative was to dispense with the concept of a session, and require the user to enter a DBXTOOL command to complete communication tasks with IDLVIEW; however, the programmer would need to understand when such communications were taking place, and would need to execute additional commands. A session also guarantees consistency of displayed information; if a programmer could display a node, run through one or more breakpoints, and then request expansion of an attribute of that node, the node might have ceased to exist or changed so that the attribute no longer existed.

During an IDLVIEW session, there are two commands made available by positioning the cursor on an item in the window and selecting a menu option: **expand attribute** and **more detail**.

The **expand attribute** option is the basic IDLVIEW user operation. The user selects a program variable which is a *root display variable* for a directed graph structure with the **shownod** command; all other boxes displayed in that session are derived from that root with **expand attribute**. The programmer displays more of the structure by expanding one or more attributes of the root, then expanding attributes of nodes displayed by expansion of the root, etc. In this way, the programmer displays just the parts of his IDL structure he wants to see.

The **more detail** option provides display of the internal representation of the data, rather than how it is interpreted by run-time routines. Some bugs are more easily found this way; if the values in a given structure instance appear meaningless,

and the hexadecimal representation of its address is "32323130", the programmer might guess that the address had somehow been overwritten by the ASCII codes for "2210".

The **select port** option (available by invoking a menu from the display window frame) provides another menu; this one lists the IDL ports declared for the process being debugged. By selecting one of them, the user limits the attributes displayed for any node to those which are declared for that node under the structure associated with the selected port.

The method for selection of items in the IDLVIEW window is not the same as in other SunTools applications. In DBXTOOL, for instance, the user positions the cursor on the name of the variable in the source display window and clicks the mouse button; a reverse-video block indicates the selected variable. To do something with the selection, the user clicks the appropriate on-screen button – **print**, **print ***, etc.

In IDLVIEW, the user positions the cursor on the desired item, clicks the right button, and a menu appears with a list of possible options. The user does not have to select the variable with one click and the action with another.

The DBXTOOL method is appropriate to its environment; for example, a programmer can select a variable and then repeatedly run to a breakpoint and print the value without having to move the cursor outside the window area containing the buttons.

IDLVIEW, however, does not have the goal of supporting such repetitive operations, and has fewer options. Once an attribute is expanded (the equivalent of the DBXTOOL **print** command), there is no point in expanding it again within that session – its value cannot have changed. We therefore eliminated the slight inefficiency of the DBXTOOL method for single selections.

5.5   The IDLView Display

The example display in Figure 3 shows a representation of an IDLVIEW display window. The top line (which is normally in reverse video) names the window and indicates that an IDLVIEW session is *active*, i.e. the user has selected a node from his program to be displayed and can now select attributes of that node to be displayed further.

```
IDLView    Active
(1) customer_list is ROOT DISPLAY NODE
    --------------------
 =o=    list <customer>   (1 element)

(2) commercial_customer is element 1 of (1).list
    --------------------
        maintenance_number  111222333
        name                "Itty-Bitty Machine, Inc."
        location            [address]
        customer_number     1
        balance             11.11
        contacts            {String}   (2 elements)
        types               {industry_code}   (uninitialized)
```

Figure 3: An Example IDLVIEW Display

On the left side of the window is a SunTools *scrollbar*; at the top and bottom of the scrollbar are *scroll buttons*; the grey area between them contains a white *bubble* which indicates the portion of text currently visible in the window. In this case, the bubble indicates that the first 75 percent of the text available is currently displayed.

Within the window itself are two *boxes*, one for a `customer_list` node and one for its attribute `list` (which is a node of type `commercial_customer`). Near the upper left corner of the window is the IDLVIEW cursor (two parallel lines with a circle between them), designed for easy selection of text in the window. The cursor reverts to the default shapes when positioned on the window frame or the scrollbar.

The box numbers appear in the upper left-hand corner of their portions of the display. The node identifier for the box (i.e. the type of node, in this case `customer_list` and `commercial_customer`) appears after the number. The user evidently displayed a variable from his program of type `customer_list`, then positioned the cursor over the `list` attribute and invoked the **expand attribute** menu option; this caused display of the elements in `list` (which is a `Seq of customer`).

The attributes of a box appear below its header. Each attribute type has its own identifying punctuation:

Type	Punctuation	Example in Figure 3
Integer	number without decimal	customer_number

Rational	number with decimal	`balance`
String	characters in double quotes	`name`
Node or Class	name in brackets "[]"	`address`
Set	name in braces "{}"	`contacts`
Seq	name in angle brackets "<>"	`types`

(The character pairs used to indicate nodes/classes, sets, and sequences are the same as those used in the ASCII External Representation.)

The header line indicates the box's source. The first box in this example is the first box of an IDLVIEW session, and so is labelled `ROOT DISPLAY NODE`. The second box was produced by expanding the `list` attribute in the first box; since `list` is a sequence, the source portion of the header identifies it as `element 1`, and also shows the name of the attribute from which it was derived. (If the expanded attribute had been a node instead of a set or sequence, the "`element <number>`" portion of the header would not have appeared.)

One can think of the box numbers as serving a function similar to lines between boxes in the graphics version. Where the graphics version could have a line connecting an attribute to the box containing that attribute when expanded, in the current version an expanded attribute display includes the number of the box from which it was expanded.

The remaining piece of information on the screen is the number of elements in each set or sequence; this appears to the right of the appropriate attribute. In cases where a set, sequence, or node is not yet initialized (as distinct from having no elements or attributes), (`uninitialized`) appears in that space (as in the example for the attribute `types`).

Each element of a set or sequence appears in its own box; if the elements are IDL basic types, the header information appears as usual and the value appears below the header. For nodes which have no attributes, only the header appears.

The display provides as much clear information as space allows, using symbols to reduce the amount of display where this does not interfere with clarity. The use of text to identify non-basic types was considered (such as `Set Of` instead of "{}", but such text would be more easily confused with meaningful text and uses more space than the delimiting characters. We felt that the tradeoff of forcing the user

to learn these conventions at first was worth the clarity it afforded later.

5.6   Summary List of Current Features

In summary, here is a list of the salient external design features. IDLVIEW:

- runs on Sun workstations, using DBXTOOL as a basic debugger;

- adds commands to DBXTOOL for creation of a display window and display of a selected IDL node;

- uses SunView to provide an interface similar to other SunTools applications;

- permits conversion to a graphics version with minimal changes;

- manages the display window with a separate Unix process, communicating with a set of library routines linked with the user's code via Unix sockets;

- operates in *sessions*: the user explores an IDL structure during one session, and may use multiple IDLVIEW sessions during one debugging run;

- provides four commands selected by menu option in the display window: **expand attribute, more detail, select port,** and **end session**;

- uses CANDLE to interpret the representations of the IDL structures in memory;

- uses SunTools standard options in most places, with a few changes to improve programmer efficiency;

- displays structure instances with the dual goals of clarity and economy;

- displays only items selected by the programmer; and

- requires no changes to user code.

6 Internal Design and Implementation

The internal design of a program is the bridge between its function and its implementation, including high-level data structures, overall program structure, and the interface with external software entities (such as SunView, DBXTOOL, and CANDLE). The implementation of a program is the code written to satisfy the external design (representing the program's function) and internal design decisions.

This section describes IDLVIEW's internal design, with some detail about the implementation chosen for that design.

6.1 Dividing IDLVLIB and IDLVIEW Functions

One of the reasons for using a separate display window management process was to lessen restrictions on and conflicts with the user's program. It therefore made sense to keep IDLVLIB, the library linked with the user program, as simple as function allowed. The following jobs were allocated to IDLVLIB with this in mind:

- *Initialize* IDLVIEW

  This includes creating a socket for communication, creating a process to run IDLVIEW, and obtaining the name of the CANDLE file and sending it to IDLVIEW.

  Creation of a child process by a program being debugged under DBXTOOL requires special consideration, since a Unix *fork* creates two processes running the same image file and DBXTOOL refuses to operate on such an image; the trick is to ensure that DBXTOOL does not regain control of its program until the child process has replaced the running image with another one. This is one reason the user must wait 20 or more seconds for the IDLView command to complete before DBXTOOL resumes operation after initializing IDLVIEW.

- *Start an* IDLVIEW *Session*

  A session begins with the display of a node; IDLVLIB gets the node's address from the user and DBXTOOL; the IDL run-time system maintains a 16-bit integer which is the identifier for the node at that address, and IDLVLIB

sends the address and the node identifier to IDLVIEW. (IDLVLIB cannot send the entire node because the node's length is not available from the run-time system. IDLVIEW obtains the length from the CANDLE structure and requests the entire node in a separate communication).

IDLVLIB then waits for and responds to messages through the socket until a message terminates the session. IDLVIEW sends requests of the following types:

- *Send Memory Block*
  Given an address and a length, IDLVLIB sends that block of memory to IDLVIEW. IDLVLIB does no interpretation of the bytes sent.

- *Send String*
  Given an address, IDLVLIB calculates the string length and sends the string as a memory block.

- *Count Linked-list Elements*
  Given an address, IDLVLIB counts the number of set or sequence elements starting at that location and sends the count to IDLVIEW.

Except for the sub-task of inter-process communication, which is discussed in its own section below, this is a complete list of IDLVLIB functions. Our success in keeping them to a minimum made implementation faster and easier, since it contained most of the processing and features in IDLVIEW. For instance, a graphics version would not require changes to IDLVLIB since no display information is kept there.

This left "everything else" for IDLVIEW. The major functional areas are listed here and discussed in the sections which follow:

- Managing communications between the processes.

- Interfacing with SunView: display, menus, and translation of cursor position to displayed item. This also includes translating user commands into their correct functions.

- Tracking displayed information.

- Using CANDLE to interpret user memory locations as IDL data structures.

Figures 4 and 5 show the overall data flow of IDLVLIB and IDLVIEW, respectively. (Data flow diagram conventions are the same as for Figure 2, discussed on page 25.)

IDLVLIB within
User's Program

Initialize
IDLVIEW

CANDLE Filename

CANDLE Filename

DBXTOOL

show
node

IDLVIEW

root display
node address

root display node
address, node id

user address,
block length

memory
block

send
block

set or sequence
address

# of elements

count
elements
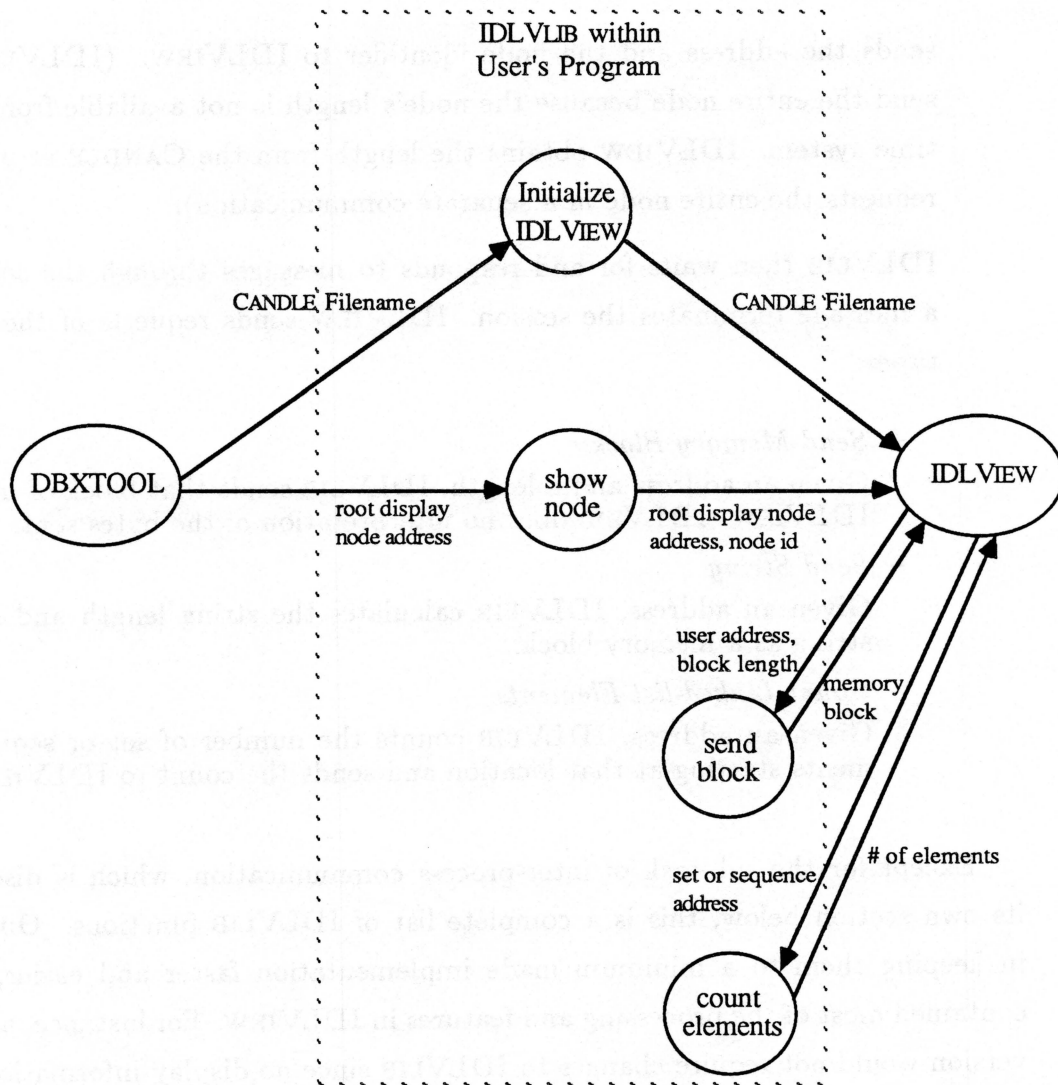
Figure 4: IDLVLIB Functional Data Flow

IDLVIEW

CANDLE
File

Initialize
Window

Internal
CANDLE

Structure Info

Expand
Attribute

item type,
length

CANDLE
Filename

user address,
CANDLE address

new display

screen position

current
port

IDLVLIB
within
User's Program

root display,
node address,
node id

Start
Session

Displayed
Data

Select
Port

user address,
length

user address
node id

address, memory
representation

memory
block

session active
display

set or sequence
address

Get
Box

Display
Box

Show
More
Detail

# of elements

box

box
display

detail
display

IDLVIEW
Display

current port
display

Figure 5: IDLVIEW Functional Data Flow

6.2 Communication Between IDLVIEW and IDLVLIB

The concept of an IDLVIEW session serves several purposes. In its external design, IDLVIEW's display during a session is limited to a static point in the program. The importance of this arises from the nature of IDLVIEW attribute expansion: you must start with an existing attribute to display more attributes. If IDLVIEW allowed the programmer to execute code between the display of an attribute and its expansion, confusing changes to the node or the attribute might have occurred which might not even be visible to the programmer. There are other ways to solve this problem, but this is the simplest and fits the debugging environment.

Another reason for IDLVIEW sessions is a limitation of the DBXTOOL interface on programs communicating with other programs. IDLVLIB is part of the user's address space, not part of DBXTOOL itself; since DBXTOOL normally has the user's program stopped, the only way any IDLVLIB subprogram can be executed is by user command (as described in section 5). A session provides for IDLVLIB to be invoked when a session is started and remain running until the session is over; if it didn't, then each time the user entered an IDLVIEW command to expand an attribute, he would need to enter a DBXTOOL command to invoke the IDLVLIB routine which sent the necessary information to IDLVIEW.

The normal approach to this problem in a stand-alone application would be to receive an interrupt when communications were received on the socket connection. While the user's program is under DBXTOOL control, however, it runs (and can receive interrupts) only when the user gives a DBXTOOL command for it to run. Since the programmer would normally want IDLVLIB and IDLVIEW communication to proceed with his program halted, this solution is not workable. Requiring a DBXTOOL user command in order to complete communications would be confusing.

Since DBXTOOL commands are disallowed during a session, IDLVLIB routines can remain running and service IDLVIEW requests for information as necessary. This simplifies overall design since IDLVIEW can send requests for information at any time; the program does not have to determine ahead of time what information may be needed, nor attempt to determine the state of interprocess communications at a given time.

Unix socket communication provides different *protocols*, but only the *STREAM* type is suitable for IDLVIEW (the *DATAGRAM* protocol, for instance, preserves

message boundaries but does not guarantee delivery; the application must track messages and ask for retransmission if necessary). Stream sockets resemble regular Unix file input and output, where messages are streams of bytes.

We next considered how to represent information in a byte stream; the two choices were bytes copied directly from the user program's memory, and some form of ASCII.

An ASCII form offered some advantages. It would facilitate a later version in which IDLVIEW runs on one machine while the user's program runs on a machine with a different architecture. Test input for IDLVIEW could be created with a text editor. Control characters would be available to help the receiving process interpret the byte stream.

However, converting internal representations to ASCII would require that CANDLE's information about the structures in memory be interpreted in IDLVLIB, greatly complicating its job. Also, information about the binary representation would have to be maintained in IDLVIEW anyway, since IDLVIEW displays information such as hexadecimal representations of data and user addresses.

With this in mind, we settled on the "memory block" as the basic unit of interchange between the two programs. IDLVLIB's central function is transferring blocks of user's memory to IDLVIEW without interpreting them, and IDLVIEW requests specific blocks based on user actions and the CANDLE structure.

Since the bytes can assume any value and the message lengths may vary, there is the problem of delimiting messages. We used a two-message structure: messages from both IDLVLIB and IDLVIEW consist of a fixed-length header and an optional, varying-length additional message. If the additional message is being sent, its length is specified in the header.

The same message header is used by both IDLVIEW and IDLVLIB; together with the "additional message" field, the structure is as follows:

msg type	constant identifying the header message
node id	IDL constant identifying a node type
addr	address in user's memory
block length	length of requested memory block
add'l length	length of additional message to follow
add'l msg	additional message bytes (optional)

We now examine the messages exchanged by the two programs in the course of a debugging session.

After the user enters the IDLView command and IDLVLIB has started the IDLVIEW process, IDLVLIB sends the first communications message; it contains the name of the CANDLE file. After IDLVIEW has read it, it returns a message ending the communications session. The pair of messages is as follows:

sent by IDLVLIB	
msg type	CDLMSG
node id	(unused)
addr	file name address
block length	(unused)
add'l length	length of CANDLE file name
add'l msg	CANDLE file name

sent by IDLVIEW	
msg type	ENDCMM
node id	(unused)
addr	(unused)
block length	(unused)
add'l length	(unused)
add'l msg	(none)

While IDLVIEW is inactive (i.e. when there is no session in progress), it is waiting for a message from IDLVLIB indicating the start of a session. The only message currently implemented which does this is the one sent after a shownod command; it sends the node identifier of the node to be displayed. Its message fields are:

sent by IDLVLIB	
msg type	FSTNODMSG
node id	identifier for node to display
addr	address of node to display
block length	(unused)
add'l length	(unused)
add'l msg	(none)

From this point until the end of the session, IDLVIEW controls the communi-

cation by sending requests for information to IDLVLIB. IDLVLIB waits for each request, returns the information, and waits for the next request; when the user ends the session, IDLVIEW sends a message ending the communications. There are three kinds of information IDLVIEW requests: blocks, strings, and linked-list counts.

A block is an area of the user process' memory which length is calculated by IDLVIEW. For instance, the FSTNODMSG received by IDLVIEW (described above) allows IDLVIEW to calculate the length of the corresponding node and request that the memory containing it be sent by IDLVLIB to IDLVIEW for display. The messages exchanged for this are:

sent by IDLVIEW	
msg type	BLKREQMSG
node id	(unused)
addr	address of node in user's memory
block length	length of block in user's memory
add'l length	(unused)
add'l msg	(none)

sent by IDLVLIB	
msg type	BLKMSG
node id	(unused)
addr	address of node in user's memory
block length	length of node requested
add'l length	length of block sent
add'l msg	requested block

Block requests are not limited to nodes. For instance, when an attribute which is a set is expanded, the information available to IDLVIEW is the address of the linked-list header node; the header contains the address of the first element and the address of the second element's header node. IDLVIEW obtains this header node by requesting it as a block from IDLVLIB; then IDLVIEW can obtain the set element with the address in the node.

A string request is similar to a block request; the difference is that IDLVIEW does not know the length of the string it is requesting. The String type in IDL is implemented as a sequence of bytes with a null byte terminator; on receiving a request for a string, IDLVLIB determines its length and returns the string as a

memory block. The message sequence is:

sent by IDLVIEW	
msg type	STRREQMSG
node id	(unused)
addr	address of string
block length	(unused)
add'l length	(unused)
add'l msg	(none)

sent by IDLVLIB	
msg type	BLKMSG
node id	(unused)
addr	address of string
block length	(unused)
add'l length	length of string returned
add'l msg	string

A linked-list count is requested by IDLVIEW for set or sequence attributes of nodes which it displays; IDLVLIB calculates the desired count and returns it with the following sequence:

sent by IDLVIEW	
msg type	LLSCNTREQ
node id	(unused)
addr	address of linked list
block length	(unused)
add'l length	(unused)
add'l msg	(none)

sent by IDLVLIB	
msg type	LLSCNT
node id	(unused)
addr	(unused)
block length	(unused)
add'l length	size of integer
add'l msg	size of linked list at given address

The simplicity of the message structure served IDLVIEW's development well, especially in cases where additional functions were added. Low-level communication routines serve higher-level routines by reading a header, allocating memory for and reading an additional message if necessary, and returning the information read to the calling routine. These routines do not interpret the message contents, other than the field indicating the length of an additional message. To add a new type of message to the system, no low-level changes are necessary.

6.3 Using SunView

The overall workstation environment is complicated to manage; different processes control different windows, the keyboard and mouse buttons have different meanings depending on cursor position, windows overlap and so have a kind of "third-dimensional" position, etc. SunView's approach to this environment is to take main-line control itself, and to *notify* application code under specified conditions.

For example, IDLVIEW creates a window with customized menus and performs user-defined operations based on menu selections. One SunView library call creates the menu, specifying subprograms to be executed for each menu option; another SunView call creates the window and associates the new menu with it. Then IDLVIEW calls a SunView subprogram which acts as a "main program loop" and does not return until the window is destroyed. SunView invokes the subprogram associated with a given menu option when that option is selected by the user.

Without such a method, the application must interpret all the asynchronous events in the window environment. For instance, cursor shape depends on cursor position, changing for different portions of windows under the control of different processes; each time the cursor moves, some code must track the cursor position and determine if the cursor display must be changed. Just as Sun's overall workstation manager tracks which programs control which windows, it is reasonable to have SunView control global events for each process.

IDLVIEW is therefore split into portions; one portion executes at startup and initializes the window environment, and each other portion executes in response to an event. The events are:

- reading information from the socket connection

- expanding a specified node attribute
- displaying more detail about a selected item
- setting a selected port as the default
- ending the IDLVIEW session
- destroying the window on user command

SunView provides several kinds of windows and subwindows. *Panels* are subwindows which can be created with text, on-screen buttons, user input areas, etc. *Canvases* are graphics-oriented subwindows with provisions for drawing lines as well as displaying text. *Text subwindows* allow browsing and editing of files, with default options for saving the file, reading another file, and so forth.

IDLVIEW uses a text subwindow, which simplifies tracking of display information by treating the display as lines and columns of characters rather than as pixels. Since it does not provide enough function for the graphics version, IDLVIEW's program structure minimizes dependence on the window type. For example, IDLVIEW contains a subprogram to display a string; it currently contains one line which calls a SunView routine with nearly identical function. This way, the implementers of the graphics version can write their string display function in that subprogram, and will not have to change the subprogram call in each place in IDLVIEW which displays a string.

The menu generation procedures are some of the best documented and easiest to use in the SunView library. We took advantage of this for the `Select Port` option, in which the user selects a port from those declared in the IDL source code. IDLVIEW reads the names of the ports for the process being debugged from CANDLE, generates a menu containing those names, and attaches it as a pull-right menu from an option of the IDLVIEW frame menu. The alternative was to have the user type the name of the port; so far no IDLVIEW function except the CANDLE file name (and possibly a root display variable name) requires any typing.

6.4 Tracking the Display – Boxes and Box Attributes

IDLVIEW stores and displays many different kinds of data types: nodes, sets, sequences, and the IDL basic types; more support will eventually be provided for private types. We sought a data structure which could unify the handling of these

types to the extent that they would depend on common functions (such as being selected by cursor position) and have common characteristics for IDLVIEW to manage (such as their location on the screen).

We named the result of using such a unifying structure the "double-funnel effect"; multiple data types going into the single structure, and out of the single structure come multiple displays. This reminded us of setting two funnels with their small ends together, one of them gathering a large stream into a small one and the other one spreading it back out again.

The external design has already introduced the concept of a display *box*, which represents an IDL element on the screen; internally, IDLVIEW maintains one data structure for each display box and one for each attribute of each box (called a *box attribute*). The central programming task of IDLVIEW may be thought of as tracking boxes and box attributes. The flexibility gained by using separate data structures will make it easier to implement future features such as two-dimensional graphics layout, user-assisted layout, and user-specified "collapsed" attributes.

For passing both boxes and box attributes among subprograms as parameters, we borrow the concept of a *handle* from SunView and other systems; a handle is a value (usually an address or integer) which represents one instance of a given object. Each handle is unique, and is used to pass the associated structure as a parameter in subprogram calls. In IDLVIEW, lower-level routines provide storage allocation for new structures, assignment of handles, and conversion of handles to structure pointers. These structures do not qualify as "opaque", since their internal structure is known to subprograms; we call them "translucent" since their allocation, storage, and handle conversion are limited to a few subprograms. The advantage to translucent structures is that the methods for the hidden operations can be changed without changing IDLVIEW code in general.

Although a box handle may be implemented as a pointer to the box structure in a given address space – it fills the qualifications of being unique and of reasonable length – we did not implement it that way and do not recommend doing so. Future IDLVIEW versions may want to obtain a new copy of a node, for instance, and replace the old box with the new one. If the box structure address were used for the handle, it would be difficult to either find all the places which used that handle (to change it) or to fit the new box into the area of memory where the old one was. IDLVIEW's box and box attribute handles are indices into arrays which contain

types to the extent that they would depend on common functions (such as being selected by cursor position) and have common characteristics for IDLVIEW to manage (such as their location on the screen).

We named the result of using such a unifying structure the "double-funnel effect"; multiple data types going into the single structure, and out of the single structure come multiple displays. This reminded us of setting two funnels with their small ends together, one of them gathering a large stream into a small one and the other one spreading it back out again.

The external design has already introduced the concept of a display *box*, which represents an IDL element on the screen; internally, IDLVIEW maintains one data structure for each display box and one for each attribute of each box (called a *box attribute*). The central programming task of IDLVIEW may be thought of as tracking boxes and box attributes. The flexibility gained by using separate data structures will make it easier to implement future features such as two-dimensional graphics layout, user-assisted layout, and user-specified "collapsed" attributes.

For passing both boxes and box attributes among subprograms as parameters, we borrow the concept of a *handle* from SunView and other systems; a handle is a value (usually an address or integer) which represents one instance of a given object. Each handle is unique, and is used to pass the associated structure as a parameter in subprogram calls. In IDLVIEW, lower-level routines provide storage allocation for new structures, assignment of handles, and conversion of handles to structure pointers. These structures do not qualify as "opaque", since their internal structure is known to subprograms; we call them "translucent" since their allocation, storage, and handle conversion are limited to a few subprograms. The advantage to translucent structures is that the methods for the hidden operations can be changed without changing IDLVIEW code in general.

Although a box handle may be implemented as a pointer to the box structure in a given address space – it fills the qualifications of being unique and of reasonable length – we did not implement it that way and do not recommend doing so. Future IDLVIEW versions may want to obtain a new copy of a node, for instance, and replace the old box with the new one. If the box structure address were used for the handle, it would be difficult to either find all the places which used that handle (to change it) or to fit the new box into the area of memory where the old one was. IDLVIEW's box and box attribute handles are indices into arrays which contain

addresses. To replace one box with another, we would replace the address in the array, and all handles referencing that box would automatically reference the new box.

We took particular care with the representation of a box's screen position in IDLVIEW. SunView maintains the text in a text subwindow as one long string, and provides library calls for determining information about it in terms of lines, etc. IDLVIEW tracks display position of an object as an integer specifying its position in the single text string; however, this will not be adequate for a graphics version. Screen position is therefore treated as a separate "hidden" structure, and functions which deal with that structure directly are limited. When the graphics version is implemented, the structure can be changed to contain x and y positions (and whatever else is necessary) with limited changes to the code.

The data fields stored for a box are:

- *box type* – node, string, integer, rational, boolean, or unknown

- *box number* – number displayed identifying this box

- *source box attribute* – the handle for the attribute which was expanded to produce this box

- CANDLE *pointer* – a pointer to the CANDLE type of which this box is an instance

- *local block pointer* – points to the copy of user's memory block in IDLVIEW

- *user address* – the address in user's memory of the memory block

- *screen position* – the position of the box on the screen.

- *first box attribute* – first pointer in a linked list of attributes for this box. The final pointer in the list is null.

- *element number* – integer representing the position of this box in a sequence or set

- *next element box* – handle of the box containing the next element in a set or sequence

- *linked list address* – user's address of the linked list set or sequence cell. IDL implements sets and sequences linked lists with "cells" which contain a pointer to the next element and a value (or a pointer to the value) of the current element.

- *attribute name width* – width in characters of the widest attribute name (plus 1) for those attributes being displayed in this box.

- *detail flag* – set if detail has been displayed for this box

The data fields stored for a box attribute:

- *box attribute type* – atomic (i.e., one of the IDL basic types), set, sequence, node, or unknown

- *source box* – handle of the box containing this attribute

- *destination box* – handle of box this attribute has been expanded to

- *value pointer* – pointer to the memory representation of this attribute; this points into the memory block for the containing box.

- *user address* – address of the attribute's value in user's memory

- CANDLE *pointer* – pointer to CANDLE structure representing the type of which this attribute is an instance

- *next box attribute* – handle of the next box attribute for this box

- *screen position* – screen position of this box attribute

- *linked list count* – number of elements of this attribute if it is a set or sequence

- *detail flag* – set if detail has been displayed for this box attribute.

When a root display node identifier is received from IDLVLIB and is found in CANDLE, a box is created for it; the block of memory representing the node is obtained from IDLVLIB, a box number is assigned, and the known fields filled in.

A box attribute structure is then created for each of the attributes in the node (filtering out any attributes not known to the current port). Additional information is obtained for any attributes requiring it; strings, for instance, are represented in nodes by pointers, and IDLVIEW obtains the value of each string from the user's program so it can be displayed with other node information.

Once all the information is obtained, IDLVIEW displays the box. The program could display the information while receiving it, but to do so would tie the receipt of data to its display. A later version of IDLVIEW might obtain a node, compare it to a version from a previous session, and update the existing display. If we had mixed the

functions of receiving and displaying, that feature would have been more difficult to implement. We were not concerned about the time elapsed while information was received and the display calculated, and our lack of concern was justified: the limiting factor on display speed is how fast SunView scrolls text, not the time taken to calculate the display.

When the user invokes the **expand attribute** menu option, SunView provides the x and y coordinates of the cursor and the text lines and columns currently displayed; IDLVIEW calculates which item is under the cursor at that display point and takes appropriate action. If the item is a node, set, or sequence attribute, it obtains the user's address from the box attribute information and re-executes the algorithm for obtaining and displaying a box.

6.5   Using CANDLE

CANDLE is an IDL structure which describes a user's IDL structure; that is, CANDLE's nodes, classes, and other IDL types describe the characteristics of the IDL structure declared by the user. Each data structure's name, type, component layout in memory, and source code position (among other things) is specified by CANDLE. This is analogous to a compiler that is written in the language it compiles.

The following small portion of two parts of the CANDLE specification shows declarations for the **Class** node (the IDL comments are not from the Candle source):

```
Class => rep_enumerated: Boolean,  --true if set or sequence
         rep_nodeId: Integer,       --unique id for class
         rep_allowedOps: Set Of ClassOperation;
                               -- operations allowed on class

Class => sem_allattributes: Seq Of Attribute, -- attributes
         sem_ancestors: Set Of Class,  --superclasses
         sem_subclasses: Set Of Class; --subclasses
```

The *instance* of CANDLE generated by the IDL compiler contains a **Class** node for each class declared in the user's IDL specification, each with the above characteristics (and more). IDLVIEW uses this instance to interpret user memory in terms of the user's IDL specification.

For instance, the **rep_nodeId** field is the identifier IDLVIEW matches to determine the type of a node in the user's memory. When the user specifies a variable

as a root display node, DBXTOOL translates it into an address; IDLVLIB sends that address and the integer at that address (as the IDL node identifier) to IDLVIEW; IDLVIEW searches through the set of nodes in the user's CANDLE instance for a matching node identifier.

When a node is found, its length is calculated from CANDLE information and the memory block which is the node transferred via IDLVLIB. IDLVIEW then visits each of the attributes of that node (in sem_allattributes, shown above); the attribute's type, name, offset from the node address, etc. are used to interpret the block of memory transferred from the user's program.

As mentioned before, a node at run-time is implemented as the union of all the attributes declared for it in a given IDL specification; this representation of a node is called its *invariant* structure. Interpretation of the memory block from the user's program uses the invariant structure for the corresponding node.

To build a box for a node "viewed" through a port, IDLVIEW visits each attribute in the invariant, then searches for that attribute in the current port. If the attribute is found, it is processed normally; if not found, it is skipped.

In an early version, IDLVIEW visited each attribute for a port and searched the invariant for a match. Different ports' views of a structure, however, do not always have their attributes in the same order, so this algorithm sometimes caused node attributes to appear in a different order for different ports. IDLVIEW, therefore, uses the order of attributes from the invariant.

The only difficulty we found using CANDLE was its size: its IDL source is over 35 pages long, and contains much information not germane to IDLVIEW. Since attributes for nodes could be found in different files and different places in one file, we finally made a reference guide we call the "Candle map"; this contains all the class and node declarations in alphabetical order by name from all the Candle source files. With this, we could look up all the attributes for a given node at one time, look up all the subclasses for a class at one time, and determine whether a given entity was a node or a class. XREF, another IDL tool currently under development, will produce such "concordances" for IDL structures automatically.

7   Conclusions and Future Work

This section outlines our conclusions about IDLVIEW and discusses possible improvements.

7.1   IDLVIEW's Current Status

We are pleased with the current version of IDLVIEW as a debugging tool. The user interface is simple and requires no typing for normal operations. The speed with which IDLVIEW displays nodes on command is quite satisfactory; the current limitation appears to be the scrolling speed of a SunTools window. The display matches the appearance of IDL structures as they are declared. Complete information is presented without "noise" characters which interfere with the programmer's view of important data. Port "views" allow the user to reduce the display in a way related to the IDL structure.

There are limitations to IDLVIEW (covered in the next section); we did provide it to 12 graduate students in an advanced translator class, who gave us useful comments and bug reports.

One speed issue we were unable to solve is the time it takes to create the IDLVIEW display window. The problems are:

1. It takes a noticeable period for SunView to create a window.

2. Interesting IDL structures have large CANDLE description files, and it takes a noticeable time for IDLVIEW to read them.

3. The IDLVLIB code which creates the IDLVIEW process must first fork its own executable image; because DBXTOOL will not operate on an image being run in two different processes, it is necessary for the IDLVLIB process to execute a Unix *sleep()* function call while the IDLVIEW process gets started [15].

Fortunately, these operations occur once or twice during a debugging session, when IDLVIEW is getting started; it would be worse if we had run into speed problems for operations which occurred during, say, commands to display nodes. An improvement

to be considered would be to have IDLVIEW run as a process independent of the user program, rather than being forked by it, and to have IDLVIEW be able to use a CANDLE file for more than one session. The user would then have to wait only the first time a program is debugged for a workstation session (unless the IDL structure were changed). After initialization, IDLVIEW's speed is quite satisfactory.

The goal of allowing for easy expansion to a graphics version has also been met. The following design features would make that implementation easier:

Inter-process communications protocol – The simplicity of this protocol makes it easy to add more functions.

Screen position tracking – The separate data structures maintained for each display item will make it easier to control the display of each item in the graphics version.

Storage of displayed items – Storing the binary copy of the user's memory provides maximum flexibility for future display features.

User interface – The user interface can remain similar in the graphics version.

The developers of the graphics version will be able to concentrate on the problems of directed graph display rather than on obtaining run-time data and tracking it, just as this IDLVIEW version was able to concentrate on IDL structure display as opposed to the debugging functions already provided by DBXTOOL.

We have been pleased enough with the current version, however, to doubt whether the graphics version will be worthwhile. There are non-graphics ways to provide some of its useful features; for instance, instead of a graphics window with boxes in a two-dimensional space, IDLVIEW could provide a way for the user to specify that an expansion was to be made in another window. With user control of the window placement, we may be able to forego solving the two-dimensional layout problem but still provide the user tools for arranging the output to suit himself.

We also solved IDLVIEW's problems without requiring changes to the user's code. There were problems for which this solution suggested itself; the linker for the C compiler, for instance, provides no way to force a library of routines to be linked into a program unless at least one of them is called from the program. Since none of the subprograms in IDLVLIB are called from the user's program, we had to find another way to force the linker to include them.

7.2 Limitations of the Current Version

This version of IDLVIEW has the following limitations:

- Booleans attributes are unsupported. The IDL compiler encodes the CANDLE instance using an assumption which is erroneous in the Sun Workstation environment; when this incompatibility is corrected, IDLVIEW will support Booleans.

- Private Type attributes may not be expanded.

- Only the default implementations and sizes of Integers and Rationals are supported.

- Only the linked-list implementation of sets and sequences are supported.

- Neither a Set nor a Sequence may be used as a Root Display Node. Doing so was not part of the program external design requirements, but IDLVIEW users have since often requested it. The IDL run-time system currently has no identifier on sets and sequences the way it does on nodes, so the shownode command cannot be used. A "showset" command might be implemented; it would send IDLVIEW information about the first element in the node which would enable IDLVIEW to obtain display information for it.

- There is a 20,000 character limit on the display window. This is a SunView limitation; the value could be set to another fixed value easily.

- The width of the workstation screen limits the length of an individual display line.

- Only C is supported as the target language. The IDL run-time system implements IDL instances differently for different languages, and IDLVIEW was written for the C implementation.

- The user may debug only the first process declared in the IDL source code.

7.3 Future Graphics Version Features

Figure 6 illustrates the following display features that might be implemented in the graphics version of IDLVIEW:

1. *Nodes* – These are displayed with headers at the top and attributes enclosed in their own boxes below.

2. *Location window* – The rectangle in the upper right-hand corner of the display contains a small representation of the parts of the structure examined so far. A smaller rectangle within that one shows the user what part of the entire explored structure is currently displayed in the main window. The user moves the "view" of displayed items by "grabbing" an edge of smaller rectangle with the mouse and cursor and "dragging" it to a different part of the entire display.
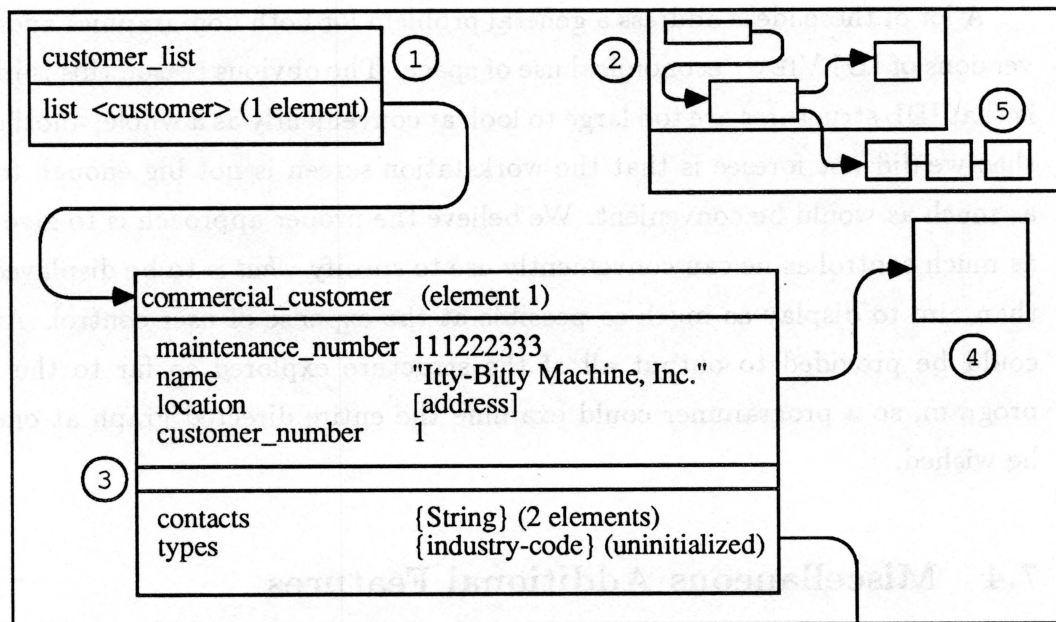
Figure 6: IDLVIEW Graphics Version Display Features

3. *Collapsed attributes* – The user has an option to collapse selected attributes, both for one node and for each instance of a selected node type. A small blank space remains on the screen to remind the user that a collapsed attribute exists there.

4. *Collapsed nodes* – The user has an option to "collapse" entire nodes; this constricts them to a small rectangle on the display screen, leaving more room for other items.

5. *Set of nodes* – Sets and sequences appear as strings of boxes linked together. In the figure, the set appears in the location window, but not in the main display window.

The idea of "collapsed" attributes within a node is connected with user-specified display formats, which is also being considered. The problems mentioned in Section 2 concerning "writing code to debug code" would not apply if the user-specified format were limited to the placement of attribute names and values [9]. For a user interface, perhaps a menu option could display the default (or current) description and allow the user to change it using the mouse and cursor; there would be a need, however, for saving such a definition so the user does not have to define it again in a future debugging run.

Another possibility for user-controlled display is that of node scaling: allowing the user to specify how large the boxes which contain display items are to be.

A lot of these ideas address a general problem for both non-graphics and graphics versions of IDLVIEW: economical use of space. The obvious reason this is important is that IDL structures are too large to look at conveniently as a whole; another reason that we did not foresee is that the workstation screen is not big enough to look at as much as would be convenient. We believe the proper approach is to give the user as much control as he can conveniently use to specify *what* is to be displayed, rather than aim to display as much as possible at the expense of user control. An option could be provided to output all of the structure explored so far to the TREEPR program, so a programmer could examine the entire directed graph at one time if he wished.

7.4   Miscellaneous Additional Features

Here we list possible features to add to IDLVIEW, with brief discussions of their use and/or problems.

Display of declaration source – The user selects a display item and a menu option, and receives a display of an IDL source file at the point where the selected item was declared. This could be complicated, since the source declaration for a given item could be spread over different places in one or more files, and it might not be useful to display them all. Possibly IDLVIEW could display a re-created source-style declaration composed from the run-time representation; this would be easier to do but would not provide such niceties as comments and context near the actual declaration.

Freezing a display window – The user selects a box from a previous session and requests an update on the displayed item; IDLVIEW displays the current value of that item in an additional box next to the old one. Alternately, IDLVIEW re-displays the box and changes the appearance of items within the box which have changed value.

Identify all instances of one node – IDLVIEW indicates all instances of a given node type by highlighting them (e.g. reverse video).

"Stacked" set/sequence display – Instead of displaying a set or sequence of items across the screen, display them using an arrangement such as in Figure 7.

```
 ╭──────────╮ ╭──────╮ ╭───────╮ ╭──────╮
 │ previous │ │ next │ │ first │ │ last │
 ╰──────────╯ ╰──────╯ ╰───────╯ ╰──────╯
 ┌──────────────────────┬─────────────────────────────────┐
 │ (2) commercial_cust  │ element                         │
 │ (1) <.list>          │ 1 of 15                         │
 ├──────────────────────┴─────────────────────────────────┤
 │ maintenance_number  111222333                          │
 │ name                "Itty-Bitty Machine, Inc."         │
 │ location            [address]                          │
 │ customer_number     1                                  │
 │ balance             11.11                              │
 │ contacts            {String}                           │
 │ types               {industry-code} (uninitialized)    │
 └────────────────────────────────────────────────────────┘
```

Figure 7: A "Stacked" Set or Sequence Display

This displays one element of a set or sequence at a time, and gives the user on-screen buttons for choosing display of another. Perhaps this could be a display option, to be chosen by the user for selected sets and sequences[1].

Middle mouse button to expand attributes – The middle button could invoke the **expand attribute** option automatically, eliminating the step of selecting it from the menu. Since expansion of attributes is IDLVIEW's most common operation, this would increase the user's efficiency.

In conclusion, we completed a tool which displays run-time instances of IDL structures within a debugging environment; the display is close to the form in which the programmer produces his IDL code. The tool can be enhanced to display the instances graphically with minimum modification. We used the existing debugger DBXTOOL for standard debugger functions, SunView for our SunTools interface and window mangement, and CANDLE for interpreting IDL run-time data. Our user interface is designed for IDLVIEW's specific needs, and as such differs slightly from other SunTools applications. IDLVIEW's display is designed to provide maximum information with a minimum of characters. Features to enhance the proposed graphics version have been explored, the problems its implementation is likely to encounter have been listed and possible solutions suggested.

---

[1]Our thanks to James Coggins for suggesting this design.

References

1. Beander, B. VAX DEBUG: An Interactive, Symbolic, Multilingual Debugger, in **Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging**. Johnson, M.S. (Ed.), Association for Computing Machinery, (August 1983), 173-9.

2. Brown, G. P., Carling, Richard T., Herot, Christopher F., Kramlich, D. A., Souza, P. Program Visualization: Graphical Support for Software Development. *Computer* (August 1985), 27-35.

3. Brown, M.H., Sedgewick, R. A System for Algorithm Animation. *Computer Graphics* 18, 3 (July 1984), 177-186.

4. Cargill, T. A. The Blit Debugger. *Journal of Systems and Software* 3 (1983), 277-284.

5. Gettys, J., Newman, R., Fera, T. D. Xlib – C Language X Interface, Protocol Version 10. MIT Project Athena, (November 1986).

6. Herot, C.F., Brown, G.P., Carling, R.T., Friedell, M., Kramlich, D., Baecker, R.M. An Integrated Environment for Program Visualization. *Automated Tools for Information Systems Design* (August 1982), 237-259.

7. Isoda, S., Shimomura, T., Ono, Y. VIPS – A Visual Debugger. *IEEE Software* 4, 3 (May 1987), 8-19.

8. Mateti, P., Radack, G.M. Integrating Data Structure Diagrams into Source Level Debuggers, in **ACM Computer Science Conference Proceedings**. Association of Computing Machinery, (February 1986), 407.

9. Myers, B. **Displaying Data Structures for Interactive Debugging**. Master's Thesis, University of California, (1980).

10. Reiss, S. P. Pecan: Program Development Systems that Support Multiple Views, in **Proceedings, Seventh ICSE, IEEE Computer Society**. IEEE, IEEE, (August 1984), 324-333.

11. Reiss, S. P., Pato, J. N. Displaying Program and Data Structures, in **Proceedings 20th Hawaii International Conference of System Sciences**. CS Press, Los Alamitos, CA, (1987), 391-401.

12. Rowe, L. A., Davis, M., Messinger, E., Meyer, C., Spirakis, C., Tuan, A. **A Browser for Directed Graphs**. Technical Report UCB/CSD 86/292, Computer Science Division (EECS), University of California, (April 1986).

13. Snodgrass, R. Displaying IDL Instances. *SIGPlan Notices* (1987).

14. Snodgrass, R. *The Interface Description Language: Definition and Use (forthcoming)*. Computer Science Press, Rockville, MD, (1988).

15. Debugging Tools for the Sun Workstation. Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043, (February 1986).

A User's Guide

A.1 Introduction

IDLVIEW provides run-time display of IDL data structure instances using SunView[1] and DBXTOOL. The user:

- compiles his code for debugging,
- links IDLVIEW routines with his own code,
- creates an IDLVIEW window with a DBXTOOL command,
- executes some portion of his code, stopping at a DBXTOOL breakpoint, and
- selects a variable containing an IDL node for display.

He can then use the mouse to select attributes of that node to be expanded; an attribute which is a node expands into its own attributes, and an attribute which is a set or sequence into its elements.

The reader of this manual is assumed to be familiar with IDL, with the basic use of DBXTOOL (such as use of buttons displayed on the screen), and with use of a Sun workstation running SunTools.

The Sun Microsystems manual *Windows and Window Based Tools* (part number 800-1287-03) is the introductory manual for Sun windows in general, and *Debugging Tools for the Sun Workstation* (part 800-1325-03) describes DBXTOOL.

A.2 The Display

Figure 8 shows parts of an IDL specification, and Figure 9 a representation of an IDLVIEW display based on that specification.

At the top of the display is the window header, which gives the name of the window and indicates that IDLVIEW is *active* (as explained with the concept of an IDLVIEW *session*, below). At the left of the window is a SunTools *scrollbar*;

---

[1]SunView is a set of libraries used by programs termed SunView applications; SunTools is a graphics workstation environment under which SunView applications run; SunWindows is an older version of SunView.

```
customer_list => list : Seq Of customer;

customer ::= commercial_customer | government_customer;
customer =>          name : String,
                 location : address,
          customer_number : Integer,
                  balance : Rational,
                 contacts : Set of String,
                    types : Set of industry_code;

commercial_customer => maintenance_number : Integer;
```

Figure 8: An IDL Specification Fragment

the black areas at the top and bottom of the scrollbar are *scroll buttons*, and the grey area between them contains the *bubble* indicating the amount of text currently appearing in the display window. In our case, the bubble indicates that the first 75 percent of the text is currently displayed.

The window itself contains 2 *boxes*; a box is an IDLVIEW display of either a node or an element of a set or sequence. The top line header for each box shows:

- the box number in parentheses; boxes are numbered sequentially as they are created. In this example, boxes 1 and 2 are shown.

- the type of the item displayed; for a node, this is the node type. In this example are node types customer_list and commercial_customer.

- the origin of the displayed item; in our example, the first box shows the node displayed first in this IDLVIEW session, indicated by the label ROOT DISPLAY NODE. The second box shows the expansion of the attribute of the first box; the header shows which element of the sequence is represented by the box (element 1), which box contains the attribute expanded ((1)), and which attribute in the source box was expanded (.list).

Underneath the header are the value(s) for a displayed item. For a node this is the name of each attribute of the node, along with either the attribute's value (for basic types) or its type. In the first box above, we see that a customer_list node has only one attribute, and it is of type Seq of customer (indicated by the angle

```
┌─────────────────────────────────────────────────────────────────┐
│ IDLView    Active                                                 │
├─┬───────────────────────────────────────────────────────────────┤
│▲│                                                                 │
│ │ (1) customer_list is ROOT DISPLAY NODE                          │
│ │      --------------------                                       │
│═□│     list <customer>  (1 element)                               │
│ │                                                                 │
│ │ (2) commercial_customer is element 1 of (1).list               │
│ │      --------------------                                       │
│ │     maintenance_number  111222333                              │
│ │     name                "Itty-Bitty Machine, Inc."             │
│ │     location            [address]                               │
│ │     customer_number     1                                       │
│ │     balance             11.11                                   │
│ │     contacts            {String}  (2 elements)                  │
│▼│     types               {industry_code}  (uninitialized)        │
└─┴───────────────────────────────────────────────────────────────┘
```

Figure 9: A Typical IDLVIEW Display

brackets around **customer**). In the second box, the following are basic types with their values displayed accordingly:

String (enclosed in quotes)   : **maintenance_expires, name**
Rational (numeric with decimal) : **balance**
Integer (numeric, no decimal)   : **maintenance_number**

Attributes of non-basic types are indicated by:

Node   : [brackets]
Set   : <angle brackets>
Sequence : {braces}

Although the **list** attribute of the first node was declared as a sequence of a class (**customer**), IDLVIEW automatically determines the node type of its element **commercial_customer** and displays it in the header of the expanded attribute.

The following are the non-basic types in the second box of our example:

node of type **address**   : **location**
Set of String   : **contacts**

Set of node of type `industry_code` : `types`

IDLVIEW displays the number of elements of each set and sequence attribute. It also indicates set, sequence, or node attributes which are uninitialized (as opposed to having no elements or attributes); in our example, the `types` attribute is uninitialized.

When an expanded set or sequence is made up of a basic type, its boxes have no attribute name. If the user expanded the attribute `contacts` above, the following boxes would appear on the display:

```
(3) String is element 1 of (2).contacts
    ------------------------------------

    "Mr. Wright"

(4) String is element 2 of (2).contacts
    ------------------------

    "Ms. Management"
```

For nodes which have no attributes at all, IDLVIEW displays only a header.

Displayed text scrolls in the window, automatically adjusting for window size. A standard SunTools scrollbar is available on the left-hand side of the window; its use is explained in Section A.5 in this User's Guide.

A.3 Setting Up To Run IDLView

IDLVIEW runs on a Sun workstation under SunTools. After ensuring that the version of SunTool is compatible with IDLVIEW, the programmer compiles his code for debugging with DBXTOOL, links the IDLVIEW library routines with his code, and initializes DBXTOOL with IDLVIEW commands.

A.3.1 Checking the SunTools Version

You must be running SunTools version 1.0 or later; this version uses walking menus instead of stacking menus, besides having other features IDLVIEW requires. The easiest way to determine what version is running is to look at the menu obtained by pushing the right-hand button on the frame of a normal window; if the `resize` and `move` options have arrows on them, then your version of SunTools is able to run

IDLVIEW; if they do not, you need to change the default version. IDLVIEW gives no error message indicating that the SunTools version is incompatible.

To change the default version:

1. Place the cursor on the Sun background (i.e. outside all windows) and push the right-hand button to obtain the general Sun menu.

2. Select the "DefaultsEditor" option . A window appears on the screen for editing the SunTools defaults.

3. The first set of defaults presented includes a rotating option for "walking menus". Move the mouse button to the small circle made of two small arrows next to that option and click the right-hand button. The displayed choice changes from "stacking menus" to "walking menus", and a message appears indicating compatibility with version 1.0 or higher.

4. Move the cursor to the "save" button at the top of the window and click the right-hand button. This saves the current version of SunTools as the default, so you don't have to select this option each time you log in.

You must not start DBXTOOL for IDLVIEW use until after this option is set, i.e. starting DBXTOOL, setting the option, and then starting IDLVIEW will still fail.

A.3.2   Initializing DBXTOOL

To facilitate use of IDLVIEW during debugging, the following DBXTOOL commands are recommended (and assumed in the remainder of this document) to be executed at the start of a DBXTOOL session:

```
alias IDLView "call sndcdlwdw(\"\")"
alias shownod "call sndfstnod(!:1)"
button ignore IDLView
button expand shownod
```

DBXTOOL executes these lines automatically on being invoked if they are placed in the file ".dbxinit" in either the working directory or the user's root directory (it uses the one in the working directory if both are present)[2].

---

[2] A sample .dbxinit file may be copied from /usr/softlab/misc/dbxinit.

A.3.3 Compiling and Linking Your Code

When compiling IDL code, use the idlc option `-C <filename>` to produce a CAN-DLE file. A common convention for the CANDLE file name is to use the name of your IDL file with "Cdl" replacing the "idl" suffix.

Compile your "C" routines with the C compiler "-g" option.

To link with the IDLVIEW library routines, add the following to the command line which links your program[3] :

```
/usr/softlab/lib/idlvlib.o /usr/softlab/lib/idlvlib.a
```

For instance, your command line might appear as:

```
cc -g main.o userlib.a /usr/local/lib/libidl.a \
   /usr/softlab/lib/idlvlib.o /usr/softlab/lib/idlvlib.a
```

You are now ready to run the linked program as an image under DBXTOOL.

A.4 Initializing IDLVIEW

On starting DBXTOOL, two buttons appear for operating IDLVIEW: `IDLView` and `shownod`.

IDLVIEW may be initialized at any time after DBXTOOL has executed some of the program to be debugged (i.e. you cannot initialize IDLVIEW immediately after entering DBXTOOL). Then, clicking the `IDLView` button creates the IDLVIEW display window.

The `IDLView` command causes IDLVIEW to prompt for the name of the CANDLE file in the DBXTOOL command window; after the filename is entered, IDLVIEW creates its display window, reads the CANDLE file, and returns keyboard control to DBXTOOL. After entry of the CANDLE file name, 20 seconds or more may pass while the window is created and IDLVIEW reads your CANDLE file; wait until the (`dbxtool`) prompt appears before continuing.

Should you ever restart the program by clicking the `run` button or executing the `run` command, you must re-initialize IDLVIEW with the `IDLView` command, creating a new IDLVIEW window.

---

[3]This assumes the location of the IDLVIEW library routines is `/usr/softlab/lib/`; change as necessary at your installation.

The IDLVIEW window has all standard SunTools frame options available: close (or open), move, resize, expose, hide, redisplay, and quit. They are selected the same way as for other SunTools applications: by putting the cursor on the frame of the window and pushing the right-hand button. See the SunTools documentation for further information on these options.

A.5   Using IDLView

Once IDLVIEW is initialized, you can continue to interact with DBXTOOL normally: set breakpoints, view files, execute code, etc. To have IDLVIEW display a node[4]:

1. select the name of the variable which contains the node with the cursor and the right-hand mouse button (the same as selecting a variable for the "print" button)

2. click the **shownod** button

The node appears in the IDLVIEW window, the display window header indicates that IDLVIEW is *active*, and the DBXTOOL command window displays a message indicating that IDLVIEW is active. This is the beginning of an IDLVIEW *session*, and until the session is finished DBXTOOL does not respond to keyboard input.

Now move the cursor into the window where the node is displayed; it changes to a shape suited for selecting items in the window. In IDLVIEW, selection is accomplished by placing the cursor on the desired item, pushing the right-hand mouse button (which produces a menu), and selecting a menu option for the desired item. This is different from some other selection mechanisms; DBXTOOL, for example, has the user click a button to select the desired item, then click the button a second time to select the action to perform on it.

There are two options available in the menu obtained within the window; both use the position of the cursor over either the attribute name or the type name of the intended attribute:

1. **expand attribute**: display the entirety of an attribute which is a node, set, or sequence. If the attribute is a node, it expands to its own box with its own

---

[4]At runtime, IDL classes are all resolved into different types of nodes; this manual refers to such entities as nodes, whereas a programmer may think of some of them as classes.

attributes; if the attribute is a set or sequence, it expands to a series of boxes, each containing one element of the set or sequence. Attributes displayed on expansion are themselves available for expansion.

In our example in figure 9 above, the second box displayed was produced by selecting **expand attribute** on the attribute in the first box.

As long as the end of the current text appears in the window, additional text causes the window to scroll upwards. If the end of current text does not appear in the window due to user positioning, additional text does not cause scrolling.

2. **more detail**: display more detailed information about the selected item; what information is displayed depends on what kind of item is selected:

Integer	:	address and hexadecimal value
Rational	:	address and hexadecimal value
String	:	address and number of bytes
Node	:	address and number of bytes
Set	:	address of first cell and first node
Seq	:	address of first cell and of first node

Place the cursor on the frame of the IDLVIEW window and push the right-hand button, and two options appear in addition to the normal ones for a frame:

1. **end session**: ends the IDLVIEW session, displays a double line at the bottom of the current display, and returns keyboard control to DBXTOOL. Another IDLVIEW session may be started with another **shownod** command from DBXTOOL; only attributes of the current session may be expanded.

2. **select port**: This is a "pull-right" menu, indicated by the arrow in its menu box. Select the option, then move the cursor to the right: another menu appears; this one has the option **reset to invariant** and an option for each port declared in the IDL process being debugged.

Selecting a port causes nodes to be displayed with only the attributes declared for the IDL structure associated with that port. Selecting the invariant, which is the default, causes nodes to be displayed with all attributes declared for them

in all structures. The title of the IDLVIEW window indicates the currently selected port, and the node header indicates the port which was active at the time the node was selected. A port may be selected at any time after the CANDLE file is read, whether or not an IDLVIEW session is in progress.

A.6   The Scrollbar

A standard SunTools feature included with the IDLVIEW window is the scrollbar on the left-hand edge. This contains a grey area and a white *bubble*; the grey area represents the entire text, and the bubble indicates the portion of the entire text which is currently visible in the text window; e.g. if the bubble occupies the top one-quarter of the grey area in the scrollbar, then the portion of the text in the window is the top one-quarter of the text.

To move the text within the window, place the cursor on the scrollbar; clicking the left-hand button moves toward the end of text, clicking the right-hand button moves toward the start of text, and clicking the middle button positions the text so that the middle of the bubble appears at the location of the cursor.

The distance between the top of the scrollbar and the cursor determines the amount of text movement the left and right-hand buttons produce: the greater the distance, the greater the amount of movement. To position a particular line at the top of the window, place the cursor in the scrollbar opposite that line and click the left-hand button.

Unlike some SunTools programs, IDLVIEW updates the scrollbar each time any command is executed which adds text; many programs only update the scrollbar when the cursor is moved onto the scrollbar. The only time you need to position the cursor on the IDLVIEW scrollbar to update it is after you have resized the window.

A.7   Removing IDLView Windows

Occasionally a bad address or other problem will cause IDLVIEW to become inoperable. In these cases, it is necessary to destroy the IDLVIEW display window with a Unix command, since the `exit` option of the frame's menu is unavailable. To do this, enter the command:

```
kill -9 <process_identifier>
```

using the process identifier displayed in the DBXTOOL window when IDLVIEW was created or finding it with the Unix "ps" command.

A.8   Possible Problems and Solutions

The combination of DBXTOOL and IDLVIEW does not handle an invalid user address gracefully. Should DBXTOOL get interrupted by the system because of a bad address, either while running the user's code, IDLVIEW library routines, or while exchanging information with the IDLVIEW display window, a "bad address" message appears in the DBXTOOL window and DBXTOOL cannot continue execution. As a side effect, that IDLVIEW display window can no longer be used to communicate with DBXTOOL, and may in fact have to be removed with the Unix "kill" command (see Section A.7 in this User's Guide).

More rarely, a bad address causes both IDLVIEW and DBXTOOL to quit functioning; in this case, place the cursor in the DBXTOOL command window, use the CShell interrupt character in the DBXTOOL command window to regain control, and restart your debugging session with the **run** command.

Another case where the interrupt character is useful is if you inadvertently enter the **quit** option which destroys the IDLVIEW display window during an IDLVIEW session. DBXTOOL will not respond to normal keyboard input, so use the interrupt character to restore control to the keyboard.

A.9   Current Limitations

The following list summarizes IDLVIEW's limitations; more complete explanations of each item follow the list.

- Booleans are unsupported.
- Private types cannot be expanded.
- Only default implementations of Integers and Rationals are supported.
- Only linked-list implementation of Sets and Sequences are supported.
- Neither a Set nor a Sequence may be used as the root display element.
- There is a limit of 20,000 displayed characters.
- The width of the workstation screen limits the length of an individual display line.

- Only C may be used as the target language.

- Only the first Process declared in the IDL source may be debugged with IDLVIEW.

Booleans are not supported due to an IDL compiler incompatibility with the Sun C compiler. IDLVIEW correctly handles nodes which do not have Booleans, even if other nodes in the program do contain Booleans. If you attempt to use nodes which contain Booleans, IDLVIEW may display bad values for other attributes and/or generate bad addresses which cause a trap in DBXTOOL.

Private type attributes are identified in a node's display, but cannot be expanded further.

Only the default representations of Integers and Rationals are supported: 32-bit integers and 32-bit floating point Rationals. IDLVIEW does not determine the implementation before attempting to display them, so using other representations will cause unpredictable displays and bad address traps.

Only the linked-list implementation of sets and sequences is supported. Attempts to use others cause unpredictable results.

You cannot display a set or sequence as a root display element. Sets and sequences may only be displayed as expansions of node attributes. Most often, attempts to do this produce a message indicating unknown node type; occasionally IDLVIEW believes it is a valid node, and displays garbage and/or generates a bad address which causes a trap in DBXTOOL.

There is currently a limit of 20,000 characters on the IDLVIEW display window. When the display reaches its limit, it simply stops displaying any more characters; there is no error message. There is no separate limit on the number of characters in a string or the number of attributes in a node.

IDLView "clips" output at the edge of the display window, so the maximum display length is the width of the workstation screen. How many characters that is depends on the typeface being used.

Only IDL programs with the C target language may be debugged with IDLView; this is because the run-time implementations are different for other languages. Attempts to use a different target language will produce unpredictable results.

A.10   Tricks of the Trade

This section describes ways of doing specific operations in IDLView which may be useful but not obvious.

IDLVIEW may be used with DBX instead of DBXTOOL, as long as DBX is run in a SunTools window. The button commands described in the section on setting up do not apply; enter the commands "IDLView" and "shownod <variable>" in place of using the buttons. This is also the way to display variables which do not appear in the DBXTOOL source display window.

You can avoid typing the name of the CANDLE file at startup by including it in the alias of the "IDLView" command. If your CANDLE file name is "test.Cdl", for instance, you can make the alias command line:

```
alias IDLView "call sndcdlwdw(\"test.Cdl\")"
```

In this case, IDLVIEW will not prompt for the name of the CANDLE file.

The environment variable IDLVIEW_BINARY, if defined, will be used as the name of the IDLVIEW executable file. This is useful in case the standard file does not exist (or is not the version you wish to use).

If you wish to initialize IDLVIEW at the beginning of your program, you can enter a break at the first executable line with the ".dbxinit" statement stop in main; then, to start your debugging environment, click the buttons run and IDLView.

If you have already expanded an attribute and want to know the box number to which it was expanded, expand it again – the error message gives the box number, which helps locate it in a display containing many items.

A.11   Error Messages

Most IDLVIEW messages appear in the IDLVIEW window near the last place a menu was invoked. If the window is obscured by another window in the place the message would appear, it appears there anyway on top of anything in that space. Each such message appears in a panel with a button labelled "ok"; operations on the workstation screen are suspended until the user acknowledges the message by clicking that button. Some messages appear in the DBXTOOL window; they do not require acknowledgement.

Both IDLVIEW display messages and DBXTOOL messages appear in the following list alphabetically by text; DBXTOOL messages are labelled as such. IDLVIEW continues normal operations unless otherwise indicated.

Attribute already expanded to box <destination box>

You tried **expand attribute** on an attribute which is already expanded. Use the box number given to find the expansion.

Attribute type unknown to IDLView;
either Candle is out of date,
or there is an unknown error

You may have tried to perform **shownod** on a variable which is not a node (such as a scalar, set, or sequence), or your CANDLE file is not up-to-date with your program. If you have an invalid CANDLE file, IDLVIEW will have to be restarted.

Bad IDLView message type #<type number>

(DBXTOOL message) Internal error. Contact IDL support personnel. IDLVIEW will have to be restarted.

Cannot expand IDL basic type

You tried **expand attribute** on an IDL basic type: Integer, Rational, String, or Boolean. This option works only on node, set, and sequence attributes.

Could not complete the 'expand' selection

(DBXTOOL message) You have clicked the **shownod** button without making a selection in the source text window. To make a selection, position the cursor on top of the name of a variable containing an IDL node or class and click the right-hand button.

Created IDLView, process #<process id>

(DBXTOOL message) The IDLVIEW display window process has been created. The number is useful should you ever need the "kill" command to destroy it (see Section A.7 in this User's Guide).

`display of unsupported box type`

This is an internal error. Contact IDL support personnel.

`dspbat:  INTERNAL ERROR`
`display of unknown attribute type`

This is an internal error. Contact IDL support personnel.

`dspbat:  INTERNAL ERROR`
`display of unsupported atomic type:<type name>`

This is an internal error. Contact IDL support personnel.

`Error opening Candle file <filename>`

IDLVIEW could not find the CANDLE filename you have given it. If IDLVIEW did not prompt in the DBXTOOL window for a file name, check for a name in the IDLView command of your ".dbxinit" file; otherwise check spelling and pathname. IDLVIEW may have to be restarted.

`Error reading compilation unit;`
`file given for CANDLE may`
`not be proper format`

IDLVIEW encountered some error while trying to read your CANDLE file. Make sure the filename you've given really is a CANDLE file. IDLVIEW may have to be restarted.

`IDLView active...`

(DBXTOOL message) This message appears normally after a **shownod** command, and indicates that an IDLVIEW session is in progress. DBXTOOL does not respond to keyboard commands until the session is ended.

`IDLView is reading the Candle file`

(DBXTOOL message) This message appears normally when IDLVIEW is initialized, and indicates that the IDLView display process is reading the CANDLE file for your program. This may take some minutes, depending on the size of your file.

IDLView: allnewbox: failed malloc

(DBXTOOL message) IDLVIEW has failed in attempting to allocate memory for its internal display structures. Contact IDL support personnel. IDLVIEW will have to be restarted.

IDLView: Error creating window

(DBXTOOL message) An error occurred when IDLVIEW attempted to create the display window. Hopefully a system message accompanies this one; if not, contact IDL support personnel. IDLVIEW will have to be restarted.

IDLView: must have 2 arguments, not <number>

(DBXTOOL message) Internal error. Contact IDL support personnel. IDLVIEW will have to be restarted.

illegal call to idlvlib...

(DBXTOOL message) The routine "idlvlib" should never be called; there are no calls to it in any IDLVIEW code. "Idlvlib" contains no useful code and its results are unpredictable. Its purpose is to force the linker to include the uncalled IDLVIEW library routines – contact IDL support personnel for more information. IDLVIEW will have to be restarted.

INTERNAL ERROR:
allnewbat() failed memory allocation

This is an internal error. Contact IDL support personnel. IDLVIEW will have to be restarted.

INTERNAL ERROR:
Bad message header,
type <internal type number>

This is an internal error. Contact IDL support personnel. IDLVIEW will have to be restarted.

INTERNAL ERROR
chgwdwttl: bad window title change request

This is an internal error. Contact IDL support personnel. IDLVIEW may have to be restarted.

INTERNAL ERROR:
`cnvbathnd() received bad box attribute handle`

This is an internal error. Contact IDL support personnel. IDLVIEW will have to be restarted.

INTERNAL ERROR:
`cnvboxhnd() received bad box handle`

This is an internal error. Contact IDL support personnel. IDLVIEW will have to be restarted.

INTERNAL ERROR
`setboxptr received illegal box number`

This is an internal error. Contact IDL support personnel. IDLVIEW will have to be restarted.

INTERNAL ERROR:
`sndrcvusrmsg: failed additional msg socket read;`
`see dbx window for error code`

This is an internal error. Contact IDL support personnel. IDLVIEW will have to be restarted.

INTERNAL ERROR:
`sndrcvusrmsg: Failed block request for socket write`

This is an internal error. Contact IDL support personnel. IDLVIEW will have to be restarted.

INTERNAL ERROR:
`sndrcvusrmsg: failed socket read;`
`see dbx window for error code`

This is an internal error. Contact IDL support personnel. IDLVIEW will have to be restarted.

INTERNAL ERROR:
`sndrcvusrmsg: failed socket write;`
`see dbx window for error code`

This is an internal error. Contact IDL support personnel. IDLVIEW will have to be restarted.

More detail already displayed for that attribute.

You have tried **more detail** on an attribute for which more detail is already displayed.

More detail already displayed for that box

You have tried **more detail** on a set or sequence element box for which more detail is already displayed.

No attribute at that location

You have tried an attribute operation in a place in the window where there is no attribute. Set and sequence elements which are not nodes have no attributes; for nodes, attributes are below the two header lines.

No box at that location

You have tried an operation in a screen position where there is no box, such as between boxes or to the right of the boxes.

No box or attribute at that location

You have tried an operation in a screen position where there is no box, such as between boxes or to the right of the boxes.

No source for value of IDL basic type

You have tried a **display source** option (currently unimplemented) on a basic type. There is no IDL source available for basic IDL types.

Node is uninitialized

You have attempted an operation on a node which has a null address.

Node type is unknown to IDLView.
Candle may be out of date,
or the variable chosen may not be an IDL node

You have attempted a node operation on a node for which IDLVIEW cannot find a type in CANDLE. It may be a set or sequence, a scalar, or a variable which is not an IDL type.

Not in IDLView session;
You must enter IDLView from dbx

You have tried an IDLVIEW command for which you must be in an IDLVIEW session.

Not in IDLView session;
Use this option to end IDLView session

You have attempted to exit an IDLVIEW session when you are not in one.

Null node address

(DBXTOOL message) You have attempted an operation on a variable in your program which contains a null address.

possibly no image named <filename>

(DBXTOOL message) Somehow the executable image for IDLVIEW is not where it is expected to be. If your installation has it somewhere besides its default directory (**/usr/softlab/bin/**), you can set the environment variable **IDLVIEW_BINARY** to the full pathname for the executable file, and IDLVIEW will use that name instead.

Selected node type not found in this port

The node you have indicated is not declared for this port.

setbatptr received illegal bat number

This is an internal error. Contact IDL support personnel. IDLVIEW will have to be restarted.

Set/Sequence is uninitialized

The pointer for the indicated set or sequence is null. Note that a set or sequence which is uninitialized but has elements will not produce this message.

```
Source display not implemented;
Requesting source for <type name>
from file <IDL file name>
```

Dummy message from (currently unimplemented) IDL source display feature. Indicates where source would be displayed from, if the feature were implemented.

```
syntax error on "sndfstnod)"
```

(DBXTOOL message) (Although this message looks like a typographical error, it is correct) You have executed the **shownod** command without an argument. You need to supply, as an argument, the name of the variable containing the node or class you want to display.

```
That display is from a previous IDLView session.
That attribute cannot be expanded now.
```

You have attempted to expand an attribute from a previous IDLVIEW session. This is not allowed.

```
Unimplemented Llist component type
```

You have attempted to expand a set or sequence of a type not supported by IDLVIEW. Currently supported are: Node, Class, Integer, Rational, String. Currently unsupported are: private types, Boolean.

```
Window already initialized
```

(DBXTOOL message) You have attempted to execute the IDLView command twice in the same program run. If you really wish to re-initialize IDLView, you must execute the **run** command first.

```
window:  Base frame not passed parent window in environment
```

(DBXTOOL message) This is a message from SunTools indicating that window creation did not complete normally. It occurs when you either attempt to initialize IDLVIEW before executing any code, or when you attempt to restart IDLVIEW under some condition that SunTools does not accept. The best action is to destroy the DBXTOOL and IDLVIEW windows and start over.

B 'man' Page for IDLVIEW

The following pages are representations of the on-line *man* information available for IDLVIEW.

NAME

IDLView -- Display IDL structures from within dbx or dbxtool

SYNOPSIS

Used from within dbx or dbxtool -- not available directly from shell

DESCRIPTION

IDLView displays the run-time values of *IDL* instances in a *SunTools* window during debugging with *dbxtool* (1) (or *dbx* (1)). It provides a mouse/menu interface for selecting portions of an IDL instance to display.

Debugging Environment

IDLView use requires SunTools version 1.0 or higher. Change it, if necessary, by enabling 'walking_menus' under the SunView section of SunTools *defaultsedit* (1).

Put the following commands into '.dbxinit':

    alias IDLView "call sndcdlwdw(\'\')"
    alias shownod "call sndfstnod(!:1)"
    button ignore IDLView
    button expand shownod

Compile and Link for IDLView use

Add the following to your *idlc* command line, providing a name for your Candle file:

    -C candlefilename

Compile your program for debugging with dbxtool (cc -g option)

Add the following to the cc command line used for linking your IDL program code:

    /usr/softlab/lib/idlvlib.o /usr/softlab/lib/idlvlib.a

Starting and Using IDLView from dbxtool

Using a Sun workstation running SunTools, use dbxtool to debug your program. When stopped at some breakpoint, click the button 'IDLView' (or enter it as a command), and enter your Candle filename in response to the prompt in the dbxtool command window. Note -- you cannot enter the 'IDLView' command before executing some portion of your own code. In 15-60 seconds, the IDLView display window appears and the 'dbxtool' prompt returns. The 'IDLView' window may be moved, resized, closed, etc. like other SunTools windows.

Once the IDLView window is created, you may continue to interact with dbxtool normally: set breakpoints, examine variables, etc.

To enter an *IDLView session* for viewing IDL variables in the IDLView window, use the cursor and left-hand mouse button to select a variable (in the dbxtool window) representing an IDL node or class, and click the dbxtool button 'shownod' (alternately, enter 'shownod nodename' where 'nodename' is the variable). Dbxtool displays "IDLView active..." and does not respond to typed commands until the session is ended (described below).

The shownod argument must represent a node or class, not a set or sequence.

The IDLView window has a standard SunTools scrollbar on the left edge for positioning the displayed text.

Normally, when through with your last IDLView session, you would delete the IDLView window by selecting the 'quit' option of the frame menu (described below under Menu Options).

Any time you restart your program with the 'run' command, you must use the 'IDLView' command to start another IDLView window. The old window cannot execute any IDLView commands, but does not have to be removed immediately if (for instance) you want to compare its display to something in a newer window. When through with it an old window, you may delete it with the 'quit' frame option as usual.

If the IDLView window should quit responding to commands (i.e. 'hangs'), use the Unix command 'kill -9 <processid>' from another window to get rid of it. The process id is displayed in the dbxtool window when the IDLView command is executed.

MENU OPTIONS

For options within the window, position the cursor on the target item and push the right mouse button to invoke the window menu. For options on the frame, position the cursor on the edge of the IDLView window and push the right mouse button to invoke the frame menu. All standard Suntools frame options are available: Open/Close, Move, Resize, Hide, Expose, Redisplay, and Quit.

expand attribute (option within window)
Expands the display of that attribute: node attributes are expanded into displays of the node's attributes, set and sequence attributes are expanded into displays of each element.

more detail (option within window)
Displays more detail about the selected item, such as its address and/or hexadecimal value.

end session (option on frame)
End the IDLView session; dbxtool resumes normal command mode. Another 'shownod' command may be entered, at the same or a different breakpoint, to enter another IDLView session in the same IDLView window.

select port (option on frame)
select the IDL port to be used to determine what attributes are displayed for nodes. A pull-right menu provides ports to select.

ERRORS

Most messages appear in the window with an 'ok' button for user acknowledgement.

Some messages appear in the dbxtool window; they do not require acknowledgement.

FILES

/usr/softlab/bin/IDLView -- image file
/usr/softlab/lib/idlvlib.a -- IDLView library
/usr/softlab/lib/idlvlib.o -- required for correct link
/usr/softlab/misc/dbxinit -- sample dbxinit file

SEE ALSO

idlc(1), dbx(1), dbxtool(1), suntools(1), defaultsedit(1)
IDLView User's Guide

BUGS

Boolean attributes are not supported.

Private type attributes cannot be expanded.

Only linked-list implementation of sets and sequences is supported.

If the shownod argument translates to non-null illegal address, dbxtool traps on the illegal address and refuses to execute any 'continue' commands. The only thing to do in that case is start the program over with 'run' and start a new IDLView window.

If the IDLView window is killed (with the 'quit' option) while in the middle of a session, dbxtool does not automatically regain control. A workaround for this is to type the 'interrupt' character in dbxtool to regain its control over keyboard input.

IDLView cannot tell for certain if a variable given to 'shownod' is an IDL node. For most cases, it can tell and will report a non-node reference, but it is possible that a non-node address will appear to be a node to IDLView, in which case it will display garbage as the values of IDL basic types.

AUTHOR
Ralph Cook
University of North Carolina at Chapel Hill