Browsing, Grazing and Nibbling Persistent Data Structures

A. Dearle, Q. Cutts and G. Kirby
University of St Andrews

ABSTRACT

Here we describe a browser that provides a two and a half dimensional viewing mechanism for persistent data structures. The browser is an adaptive program which learns about its environment; this knowledge is stored in the persistent object store. It achieves this by making use of a compiler that is a dynamically callable data object in the environment. Other novel features of the design of the browser include the use of an event–driven software architecture in which all applications are programmed in a passive object-oriented style.

1. INTRODUCTION

The requirement to examine data structures often arises in computer applications. This requirement may be satisfied by a tool known as a browser. Such a tool, the PS-algol object browser, is discussed in [4]. It has proved useful for traversing the data structures found in persistent object stores, often permitting insight to be gained into the behaviour of complex and highly dynamic systems. It has also been of great value in debugging data structures such as the program graphs of the intermediate language, PAIL [3].

In this paper, after an initial discussion of the PS-algol object browser, a new object browser is described. In addition to the functionality of the former, the latter is capable of displaying the topology of complex data graphs. Like the PS-algol browser, the new one is an adaptive program that learns about its environment incrementally. It does this by dynamically creating programs, compiling them and linking them into the running program. The browser also utilises an event–driven software architecture [2]; this architecture and the browser's interaction with it are described fully below.

2. THE PS-ALGOL OBJECT BROWSER

The PS-algol object browser may be used to traverse arbitrary graphs. When the browser encounters an object, it interrogates its rule base to see if it has a procedure capable of displaying that kind of object. The interrogation of the rule base involves matching on type using structural equivalence. If the browser finds such a procedure, it is applied to the object supplied as a parameter. If, however, the browser does not find a rule for the object, it must produce one. This is achieved by automatically generating the source of a browsing procedure for that type of object. The source is then compiled using a compiler that is a first class data object in the environment. The resultant procedure, keyed by the type of the object, may then be added to the rule base. Finally, the procedure is applied as described above and the browsing continues.

The procedures produced by the PS-algol object browser all present the user with a menu that permits the fields of objects to be interrogated. Primitive objects, such as integers, are displayed using procedures written at browser construction time. For example, if an object of the class:

structure *x*(**int** *a* ; **string** *b* ; **pntr** *c*)

is encountered, the menu shown in Figure 1 will be displayed on the user's screen.

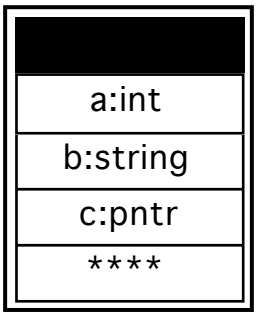


Figure 1**:** A menu for a structure type.

The structure of the menu indicates to the user the shape of the encountered object. The fields of the menu, namely "a:int","b:string" and "c:pntr", are all light-buttons. When "clicked" with the mouse, the value associated with the corresponding field of the structure is displayed. In the case of the pointer field, *c*, the menu is replaced on the screen by another which displays the object referred to by field *c* of the original object. This process may be visualised by a stack of menus being placed on the screen, with only the topmost menu being visible at any time. The last light-button (marked "****") permits the user to return to the object that was displayed immediately before the current one, or in the case of the first object to finish browsing. This operation is akin to popping the stack of menus.

Such a menu may be produced by the (slightly simplified) segment of PS-algol code shown in Example 1, which may be stored in the persistent store and used by any program. The procedure takes a pointer to an instance of a structure of some class as a parameter; this class is statically unspecified. In practice, the browser will ensure that this procedure is always supplied with a pointer to the structure class declared in the first line of the procedure. The first line of the procedure (after **begin**) merely serves to declare the relevant type, in the local context. The procedure constructs two vectors, one of strings and one of procedures. These are supplied to the procedure *menu* which generates a procedure that will place a menu on the screen at the coordinates specified when it is called. The procedure returned by ‘menu’ is finally called in the third to last line of the example, after checking that the pointer passed to the procedure is of the expected class.

```
let traversex = proc( pntr p )
begin
    structure x( int a ; string b ; pntr c )          ! Declare the structure class
                                                      ! which this procedure
displays.
    let return = proc() ; {}                          ! An empty procedure

    let strings = @1 of string [   "a:int", ! Declare a vector of strings
                                   "b:string",        ! with lower bound 1
                                   "c:pntr" ,         ! for the menu entries.
                                   "****" ]

    ! Next declare a vector of procedures - the menu actions.
    let procs = @1 of proc()   [ proc() ; write p( a ),    ! Display the integer a.
                                 proc() ; write p( b ),    ! Display the string b.
                                 proc() ; Trav( p( c ) ) , ! Browse the object c
                                 return ]                  ! Return - do nothing.

    let this.menu = menu(  "x",          ! The title.
                           strings,      ! The entries - a vector of strings
                           procs )       ! The actions - a vector of procedures.

    if p is x
    then this.menu( 20,20 )              ! Display menu at source position 20,20.
    else Error()                         ! Take some error action.
end
```

Example 1**:** A procedure to display objects of class x.

3. EXPERIENCE WITH THE PS-ALGOL OBJECT BROWSER

The PS-algol browser was originally designed to aid the debugging of abstract program graphs. Since then it has proved an indispensable tool for navigating around the persistent object graph. However, it does have one major drawback, namely the limitation of only displaying one object at a time. The PS-algol object browser may, therefore, be considered to give a one dimensional view of two dimensional data structures. This has the additional undesirable consequence that, using the browser, it is impossible to discern the difference between the data structures shown in Figure 2. When viewed by the PS-algol object browser, both these data structures will cause a potentially infinite list of menus of the form shown in Figure 3 to be placed on the screen.

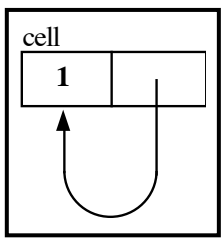


Figure 2(a)**:** A data structures.

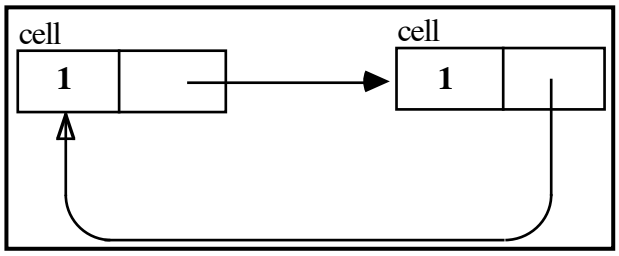


Figure 2(b)**:** Another data structure.

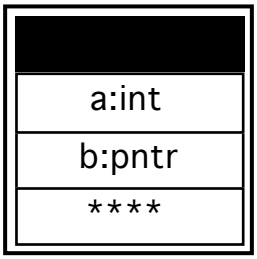


Figure 3**:** The menu for the data structure in Figure 2.

A more desirable situation would be to present the user with diagrams similar to the ones shown in Figures 2 showing the topology of the graphs being browsed, whilst still retaining the ability to "click" on fields to discover the values associated with them. A new browser, which is the subject of this paper, has been constructed which satisfies these requirements. Before describing this browser, we first describe the event–driven software architecture which the browser uses heavily.

4. AN EVENT–DRIVEN SOFTWARE ARCHITECTURE

The Event-Driven Software Architecture [2] provides the applications programmer with a number of tools useful for programming applications for a bitmapped workstation. From a user's (as opposed to a programmer's) point of view the most obvious of these is the window manager. The *window manager* provides typical window services to the applications programmer. Another module, the *interactive window manager*, provides users with a menu-based interface to these services.

The architecture also provides applications programmers with an event-driven scheduling mechanism. This allows applications to be programmed in a passive style, each application being suspended until being re-awakened by some external event. The event distribution mechanism is described below.

4.1. Notifiers

The notifier concept is fundamental to the design of the architecture. A notifier can be thought of as an event distribution procedure. An event, which may be a mouse button selection, mouse movement or key depression, is passed to a notifier, which then determines

whether the thread of control should be passed to any one of a number of registered applications.

In most systems, programs are written as active objects. Specifically, they read input by continually polling the keyboard and mouse until an event is received. In a notifier system, however, applications are passive, in that they are called by the notifier when an event occurs which concerns them. If all applications are written in this style, they will never be trapped in busy loops. As all applications are input-driven, a radically different style of programming is required. It should be noted that some applications are not well suited to this paradigm – for example, procedures that get some input, process it, and return a result.

Any application wishing to run in the system must first register with a notifier by passing it two procedures. These two procedures constitute a notification. The first, called *examineEvent*, determines whether an event is relevant to the application; the second, called *processEvent*, is the action required when such an event occurs. A notifier will pass the thread of control to the second procedure if an event is deemed important to the application by the first procedure. When a notification is submitted to a notifier, a third procedure is returned which, when called, will remove the notification from the notifier.

A notifier is comprised of two procedures: an *addNotification* procedure that allows new notifications to be registered with a notifier, and a *distributeEvent* procedure which takes an event as a parameter. The *distributeEvent* procedure permits events to be passed into a notifier. Since the *distributeEvent* procedure of a notifier and the *processEvent* procedure of a notification both have the same type, notifiers may be arranged into a notification hierarchy.

4.1.1. Events

An external event can currently take two forms:

1. **structure** mouse(**cint** X.pos, Y.pos ; **c*cbool** the.buttons)

which is the structure returned by the PS–algol *locator* function, and,

2. **structure** Chars(**string** chars)

which is used to encapsulate keyboard events.

Instances of the structure class "mouse" are used to encapsulate events relating to mouse movement. The fields of this structure class permit the position of the mouse and the state of the buttons to be obtained. The second structure class "chars" is used to encapsulate keyboard events. This class is provided so that keyboard events may be injected into the PS-algol infinite union **pntr**.

4.2. The event monitor

The event monitor is the only active application in the architecture and provides events to the top-level notifier of the notification hierarchy. An event monitor is a simple loop which gathers all events and passes them to the *distributeEvent* procedure of the notifier at the top level of the notification tree. The procedure that generates an event monitor takes two parameters: the first is a procedure which returns a boolean value, determining whether the event monitor should terminate, and the second is the *distributeEvent* procedure of the top-

level notifier.

Once a top-level notifier has been initialised, and some applications have been registered with it, the event monitor passes any input to it using its *distributeEvent* procedure. Subsequently, any registered applications, when called, may add extra elements to the list within the notifier, or remove themselves from it.

4.3. The window manager

The window manager is responsible for controlling a collection of overlapping windows. When the window manager is initialised, a package of procedures is returned which create and manipulate windows within the system. An interactive window manager is also available, which provides a convenient user interface to the window system.

In the system, a window manager is an instance of a structure containing procedures to create windows, delete them, manipulate them within the window manager display area (*e.g.* move them around, bring them to the front or the back), and resize them. The window manager also allows windows to be iconised, opened (de-iconised), and made current.

Unlike most window managers, the window manager does not manage a physical device but a window. This allows window managers to be created inside windows being managed by another window manager. In order to instantiate this sequence, a function is provided to create a window for a physical device. Although this feature sounds rather esoteric, we will show its utility in the construction of the browser.

When the window creation procedure is called, an instance of a window is returned. The window structure contains procedures that allow the manipulation of: the size of the window, its title, the application associated with the window, the style and position of the icon associated with the window, the cursor associated with it, and the graphical contents of the window.

Windows contain a default application — an empty procedure — when they are created; this procedure may be changed later. The system has a notion of a current window, which will normally take all character input, and only the application therein will be active. The current window is the only one whose application has an entry in the notifier; when another window is made current, the previous entry is removed from the notifier, and the new application registered. This prevents the notifier's internal list of notifications from becoming too long.

4.4. The interactive window manager

The interactive window manager provides an interface to the procedures made available by the window manager package. For manipulating windows when partially obscured, a background menu is provided. This contains the following options :

Delete	removes the window clicked on, and its associated application.
Move	moves the window around until another click occurs.
Push/Pop	brings a window to the front if it is not there already; if it is, then it is put to the back.
Quit	quits the interactive window manager.

4.5. The tile manager

The architecture also provides a tile manager which may be used to manage windows. A tile manager has similar functionality to a window manager, the difference between them being that tiles, unlike windows, may not overlap. The panel items briefly discussed below all make use of tile managers to manage the graphical resources they use.

4.6. Panel items

A number of panel items such as light-buttons, choices, sliders, menus, etc. are available as predefined applications. In the browser, we will only make use of light-buttons and sliders. A light-button associates an area of a window with a procedure. When a light-button is selected, the user receives visual feedback: the button is highlighted, and the associated procedure is executed. Sliders also associate an area of a window with a procedure. However, they permit real values to be chosen from within a specified continuous range.

4.7. Menus

The architecture provides a procedure that generates pop-up menus. However, like the procedure supplied by the PS-algol system, this menu only remains on the screen whilst one of the procedures registered with it is active. In the browser, a menu is required that will stay on the screen indefinitely. This kind of menu may be constructed using windows and light-buttons.

5. A 2TMD OBJECT BROWSER

Let us assume that the browser is called with a pointer to the data structure shown in Figure 2. Initially, two objects are displayed to the user: a panning tool and a menu representing the first object in the data structure.

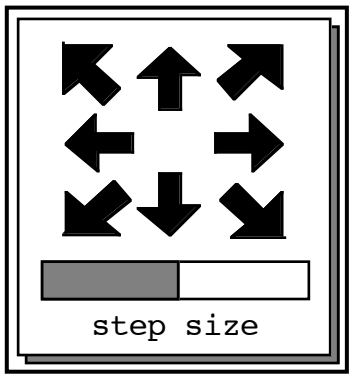


Figure 4: The panning tool.

5.1. The panning tool

Using the browser, the user's screen represents a view onto a conceptually infinite surface. Visual representations of objects are placed on this surface by the browser. The panning tool, common to many applications of this kind, permits users to move the view port around the surface. The panning tool may be considered to be attached to the viewport since it does not

itself move when the view changes. The panning tool permits the view to be panned in one of eight directions. A slider is provided in the panning box controlling the distance traveled across the surface each time one of the arrows is clicked.

5.2. The first menu

The other object on the screen immediately after calling the browser is a representation of the object passed to the browser as a parameter. In the case of the data structure shown in Figure 2(b), the menu will look like the one shown in Figure 5.

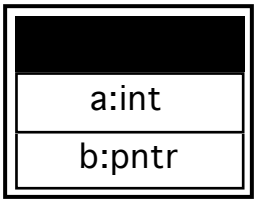


Figure 5**:** First menu for the data structure shown in Figure 2(b).

Unlike the menus provided by the first PS-algol object browser, this menu will remain on the screen until explicitly deleted. Like the panning tool, this menu has the functionality of a window and may be pushed and popped, moved, and deleted. These functions are provided by the interactive window manager initiated by the browser. The title bar of the menu also provides access to these functions via the first, second, and third mouse buttons, respectively.

The other fields of the menu are implemented by light-buttons. If the field marked "b:pntr" is clicked, a new menu will appear on the screen showing the object referred to by field b of the object. This change is shown in Figure 6.

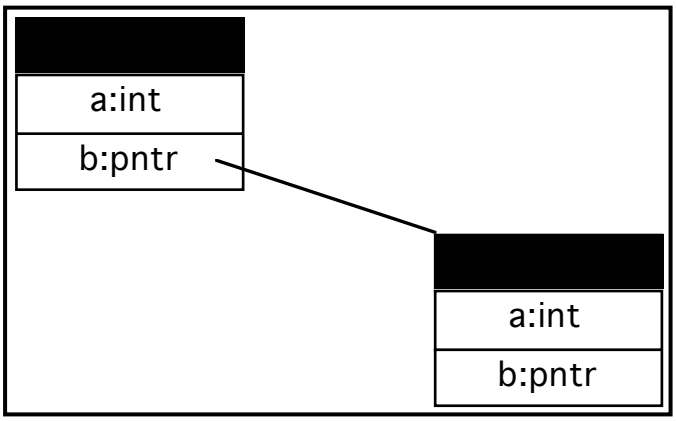
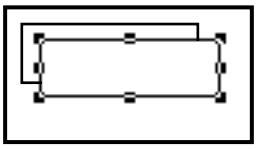


Figure 6**:** Two menus being displayed showing object references.

5.3. Object placement

At this point in the design of the browser, a problem was encountered, namely where should new objects be placed on the screen. This problem was solved with reference to that paragon of good interface design – MacDraw [4]. In MacDraw, when an object is selected, like so,

it is possible to duplicate it. By default, this causes a copy of the object to be placed to the right and down from the original object like so:



If, however, the duplicated object is moved to another position and another duplication performed, the next object will be placed relative to the new object in the same position as the second object was placed relative to the first.

This object placement strategy is used for displaying objects in the browser. The user may at any time move objects to any position he or she chooses; alternatively if the screen becomes too cluttered, objects may be removed from the screen.

5.4. Discovering relationships

Another design decision that emerged was how the browser should actually behave. Consider the graph for the data structure in Figure 2(b), shown partially displayed in Figure 6 and in its final form in Figure 7. The dilemma concerns whether or not to show relationships between objects already on the screen and new objects. In the case of this example, whether to draw a line showing the relationship between the second object and the first when the second is object is displayed.

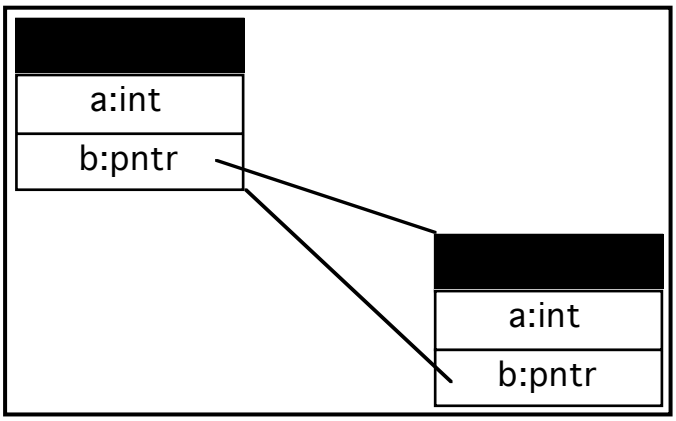


Figure 7**:** Two menus being displayed showing all object references.

The decision made in the browser was to allow relationships only to be displayed by discovery. The browser does not display object references unless the user has clicked the field that makes the reference. As a consequence of this decision, the user may easily see where he or she has been already.

5.5. Universes

The need to provide separate universes was recognised early in the design of the browser. Consider a universe populated by three objects called "Graham", "Quintin" and "Al".

Suppose we want to look at all the objects referenced by "Graham", "Quintin" and "Al", respectively. Viewing all these objects at the same level of visual abstraction may lead to confusion, with the user unable to discern which objects are associated with which. Such a situation is likely to arise if the user's screen is too cluttered. This situation may be viewed as shown in Figure 8.

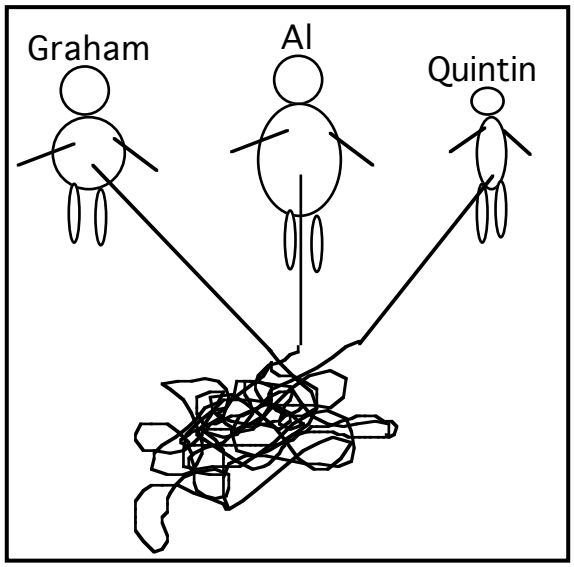


Figure 8: Shared internal references.

A more desirable situation would be for the data structures referenced by each of the objects to be displayed in a new, logically separate universe. Figure 9 shows such a separation, and here it is clear which objects are referenced by "Graham","Quintin" and "Al" respectively. Presenting objects in separate universes makes it easier to compare data structures and provides the user with the ability to logically partition the view space in any way he or she chooses.

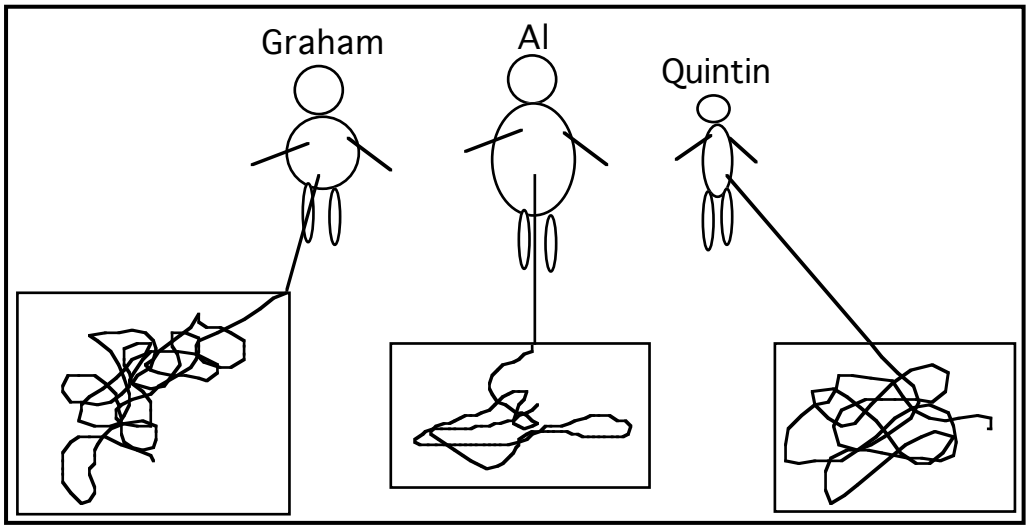


Figure 9**:** Internal references in separate universes.

In the 2™D object browser, each universe is represented by a window. Like the whole display, this window may be thought of as a viewport onto an infinite surface on which objects are placed. Objects displayed in this new universe may, of course, open up yet more new universes. Figure 10 shows a snapshot of the browser just after a new universe has been opened onto the original data structure shown in Figure 2(b).

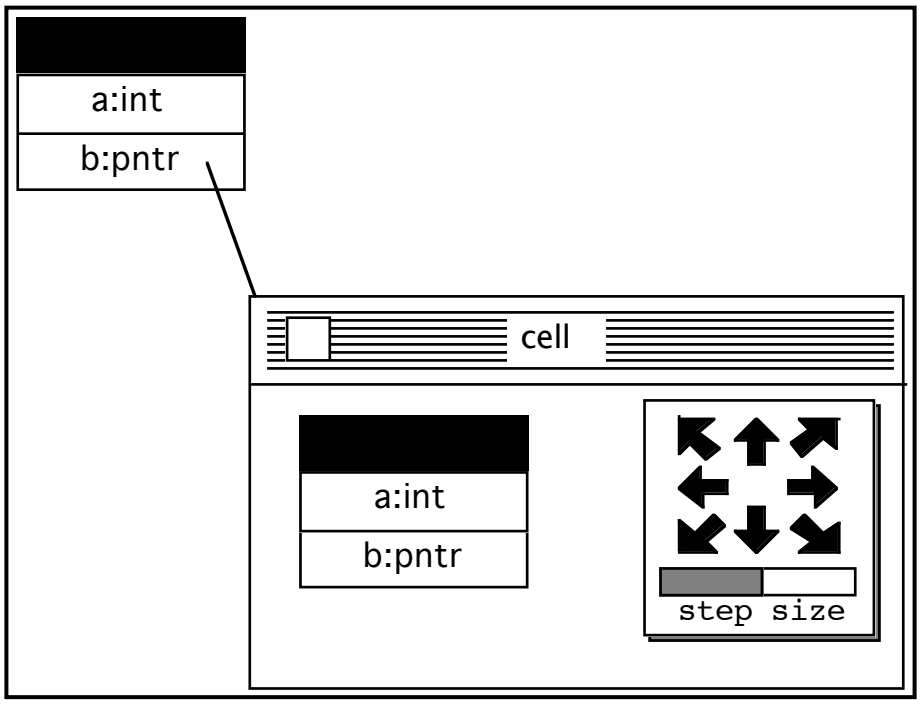


Figure 10**:** Separate universes in separate windows.

The light-buttons in the menus permit three choices to be made with the three mouse buttons (on the Sun workstations being used to develop the current version). They correspond to: viewing the object referenced by the field in the same universe, viewing it in a new universe

and deleting it.

6. CONSTRUCTION OF THE BROWSER

The browser is entirely implemented in PS-algol. It may be considered to be implemented in three pieces: the man-machine interface, the transient visual object manager and the adaptive persistent knowledge base. Each of these parts is orthogonal to the others and will be dealt with separately.

6.1. The man-machine interface

All of the man–machine interface is provided using the event-driven software architecture (EDSA). Most of the components are "off the shelf", that is they are provided by the architecture as predefined applications. The only exceptions to these are the menus and the panning tool. The menus are implemented as windows containing a tile manager which manages the menu entries. This permits menus to be implemented which may overlap, be moved around, pushed and popped etc.

Although the window manager provided by EDSA manages windows and permits line drawing on windows, it does not support the infinite plane concept described earlier. Therefore, a data structure must be maintained by the browser to manage the positions of on-screen objects and the relationships between them.

6.2. The visual object manager

The visual object manager (VOM) is responsible for maintaining information on: the position of all objects on the screen; the known interdependences between objects on the screen; the relationship between screen objects and objects from the data structure being traversed. Notice that this problem is very similar to the problems encountered in constructing a Persistent Object Management System [5].

The objects managed by VOM are arbitrary pointers and windows. It is not obvious how these objects may be sorted, so they are stored unsorted. This means that all the searches for objects are linear. This does not represent a problem since the number of objects in a universe should never be too large. VOM therefore maintains a list of the following structures:

> **structure** keeper(**pntr** object, window, refersTo, referredToBy)
>
> where
>
> *object* is a pointer to the data structure object being represented on screen
> *window* is a window (menu) on screen representing the object
> *refersTo* is a list of windows referred to by this window
> *referredToBy* is a list of windows that refer to this one

The VOM provides a number of functions that maintain and operate on this data structure. These include:

> *WinOnScreen* which returns the window associated with an object,
> *ObjOnScreen* which returns an object associated with a window,
> *addObject* which provides the manager with information about a new

object,
removeObject which removes an object from the manager, and
ObjectIsReferenced which returns true if one object refers to another.

6.3. The adaptive knowledge base

The 2™D object manager, like the PS-algol object browser, is an adaptive program [7]. That is, it learns about the object universes in which it operates. The reason for this approach is that in most programming and database systems, there are a potentially infinite number of types which may occur in the system. This represents a problem when writing a program to browse over them. In general, one cannot write a static program to anticipate all of the types that may occur without resorting to some magic or a second level of interpretation. Generally, object-oriented programming languages avoid this problem by resorting to a combination of conventions and dynamic typing. For example, one solution to this problem would be for every instance of a class to have a print method. This is not a safe solution to the problem since a print method may be overwritten by a method which performs a completely different function.

The browser maintains and uses a table which is used to store the procedures that display particular classes. This table contains procedures, each of which is capable of displaying a different type of object. A representation of this type is used to index the table. Whenever a suitable display procedure cannot be found by the browser, a procedure is called to generate the necessary compiled code. Since the class of any object may be discovered, it is easy (but not trivial!) to synthesize a procedure to display an object of that class. Such a procedure was shown in Example 1. This procedure may be compiled using the callable compiler provided by PS-algol, and the resulting code entered into the table for future use. This stage is a combination of dynamic linking and memoising.

In a conventional programming system, the scheme described would be very expensive. The traversal program would have to recreate the traversal procedures in every invocation. In a persistent programming language, the table may reside in the persistent store and therefore any changes made to the table will exist as long as they are accessible. This has the effect that the browser **never** has to recompile traversal procedures. The program in effect *learns* about new data structures. It does so in a lazy manner, as it only learns how to display the classes that it is actually required to display.

6.4. Overall construction

The three parts of the browser, described above, are largely orthogonal to each other. The interface between the programs stored in the knowledge base and the man-machine interface procedures may be entirely encapsulated in a single procedure — the menu procedure. The menu procedure, which is passed as a parameter to the object display functions, must be dynamically bound to a particular window manager.

An instance of the VOM must be created for every new universe which is placed on the screen, including the first one. This knowledge is transient and is discarded when a universe is removed from the screen. On the other hand, the procedures stored in the knowledge base are stored in the persistent store and will be kept indefinitely.

7. CONCLUSIONS & FUTURE PLANS

Database systems are notoriously hard to manage. Part of this difficulty stems from the lack of good tools to manage them. Persistent data stores commonly contain complex data structures which cannot be described adequately using textual notations. Tools that permit these complex data structures to be viewed graphically are seen as being a viable alternative. A graphical view of a complex data structure may be used to assist managers of data to visualise the effect of change upon that structure.

Similarly, tools are required to allow data managers to change complex data structures. Writing code to make changes is error prone and expensive. It also requires a high degree of training on the part of the manager. As described in this paper, the browser does not provide any facilities for altering data structures. However this browser is merely a prototype, and it is easy to see how the strategy described in this paper could be extended to permit what might loosely be described as "data structure engineering". We expect to start experiments on this in the near future, constructing programs to change data structures directed by user gesture. This may be achieved using the callable compiler in a manner similar to techniques used in the browser. The data structures required to do this are already maintained by the 2™D object browser.

Software engineering environments also suffer from a preponderance of textual interface tools [6]. In a persistent environment, such as that being constructed to support the language Napier [8], the browser is expected to be one mechanism with which a user may navigate a universe of potentially useful code. With a browser, the user may position him(her)self in an environment containing code to be reused. The provision of the first class compiler will permit the user to construct and compile new code which is bound to the code discovered in the database.

ACKNOWLEDGEMENTS

We would like to thank Fred Brown who co-designed the PS-algol object browser, the inspiration for this new browser; Tony Davie for his motivating comments that led us to start work on this browser; Richard Connor for his part in designing the notifier hierarchy and finally to Ron Morrison for his suggestions on, amongst other things, universes.

REFERENCES

1. Cockshott P. & Brown A.L. "CPOMS – The Persistent Object Management System in C", PPRR-13, Universities of St Andrews and Glasgow, Scotland, 1985.

2. Cutts Q. & Kirby G. "An Event–driven Software Architecture", PPRR-48, Universities of St Andrews and Glasgow, Scotland, 1987.

3. Dearle A. "A Persistent Architecture Intermediate Language", PPRR-35, Universities of Glasgow and St Andrews, Scotland, 1987.

4. Dearle A. & Brown A.L. "Safe Browsing in a Strongly Typed Persistent Environment", *The Computer Journal*, 31,6 December 1988, pp. 540-544.

5. *Inside Macintosh*. Apple Computer Inc. Addison Wesley, (1986).

6. Marlin C.D. "Language-specific editors for block-structured programming languages", *The Australian Computer Journal*, 18,2, May 1986, pp.46-54.

7. Morrison R., Dearle A. and Marlin C.D. "Adaptive Data Stores", Australian Joint Artificial Intelligence Conference – Proc. AI'88 Adelaide, Australia, November 1988, pp 135-145.

8. Morrison, R., Brown, A.L.,Connor, R.C. & Dearle, A. "The Napier88 Reference Manual", PPRR-77, Universities of Glasgow and St Andrews, Scotland, 1989.