Post Mortem Debugger for Oberon

Markus Hof
Institute for Computer Science (System Software)
Johannes Kepler University, Altenbergerstrasse 69, A-4040 Linz
hof@ssw.uni-linz.ac.at

Abstract

The Oberon environment [Reis92] offers only limited possibilities for inspecting run time information, as well as for determining the cause for an exception. Only variables of basic types such as INTEGER or CHAR can be examined. This paper presents a post mortem debugger that allows also the inspection of structured types as RECORD or ARRAY. It supports views on all local and global variables of any type. Run time types are supported as well as dynamic arrays. It is possible to follow variables of type POINTER in order to traverse complex data structures. Additionally, the trap position in the source code can be viewed.

Special attention has been paid to make the debugger easily portable. The debugger has been ported to several Oberon systems (SPARC, RS6000, Windows and PowerMac). This required only minor changes.

Another goal was to embed the debugger into the Oberon environment as smoothly as possible. This results in a look and feel typical for Oberon. Many features of the Oberon system (text frames, text elements) have been reused, resulting in small and compact code.

Keyword Codes: D.2.5; D.2.6; D.4.9

Keywords: Programming techniques, Testing and Debugging; Programming Environments Operating Systems, Systems Programs and Utilities

1. Introduction

A major drawback of the Oberon system is its lack of a debugger. In Oberon there are only three means for getting information about a running program:

- The command *System.State* (showing part of the global variables of a module).
- The command *System.ShowModules* (showing the list of all loaded modules).
- The trap viewer (showing the active procedures on the stack as well as part of their variables).

The problem for successful debugging is buried within the words 'part of'. Only variables, for which there exists reference information in the object module, are shown. This reference information is created by the compiler. Unfortunately, the compiler does not generate this information for all variables. Structured types such as arrays or records are omitted. Therefore variables of these types cannot be shown. As in Oberon pointers may only reference arrays or records, pointers can only be shown as the address of the object they reference.

The idea of this work is to eliminate these limitations with the help of a new debugging tool. This tool displays the following information:

- The loaded modules.
- The procedures on the stack.
- Every global and local variables of all modules and all active procedures (regardless of its type). Active procedures are procedures with an active stack frame on the current stack (stack during a trap).
- The position in the source code corresponding to the program counter for all procedures on the stack.

Besides the above mentioned requirements, some general design goals were set, which partially contradict each other. The main objective was to find an optimal balance between the different goals:

- As simple as possible.
- Minimal use of memory.
- No interference with normal work. No additional delays.
- Conformity with the Oberon environment and usage conventions. Use interface with commands, viewers and text elements.
- No modifications of the Oberon system and the Oberon compiler which are not absolutely necessary.
- Avoid dependencies on the underlying system as far as possible, to make the debugger portable.

These goals were met to a huge degree. The debugger needs almost no memory on the Oberon heap, needs only one minor change in the compiler and is built on top of the Oberon environment using text elements as well as viewers. As a result of making the debugger as simple as possible, it needs only about 20 kBytes of disk space (depending on the underlying operating system).

The debugger presented in this paper is not a tracing debugger which allows the execution of a program step by step, but a post mortem debugger. A post mortem debugger only allows the user the inspection of machine states present after a trap occurred.

2. How to Use the Debugger

We start explaining the usage of the debugger with the help of an example. Afterwards we describe its different commands and its other abilities in depth.

2.1 An Example

The debugger displays values of variables as well as the trap position within the source code. To illustrate some basic features of the debugger we introduce a small example program. The program does not make any sense but it suits well for a demonstration. If the module Demo is loaded the body is executed. The body calls the procedure P which again generates a trap with the Oberon statement HALT.

```
MODULE Demo;
VAR
     ch: CHAR;
     r: RECORD f1, f2: REAL END;

     PROCEDURE P;
          VAR i: INTEGER;
     BEGIN
          i:=ORD(ch);
          HALT (99)           (* <---- Inspect variables at this point *)
     END P;

BEGIN
     ch:='A';
     r.f1:=1.0; r.f2:=3.14;
     P
END Demo.
```

Listing 1: Sample program.

When activated, *Debugger.Source* displays the source code with the caret placed at the trap position.

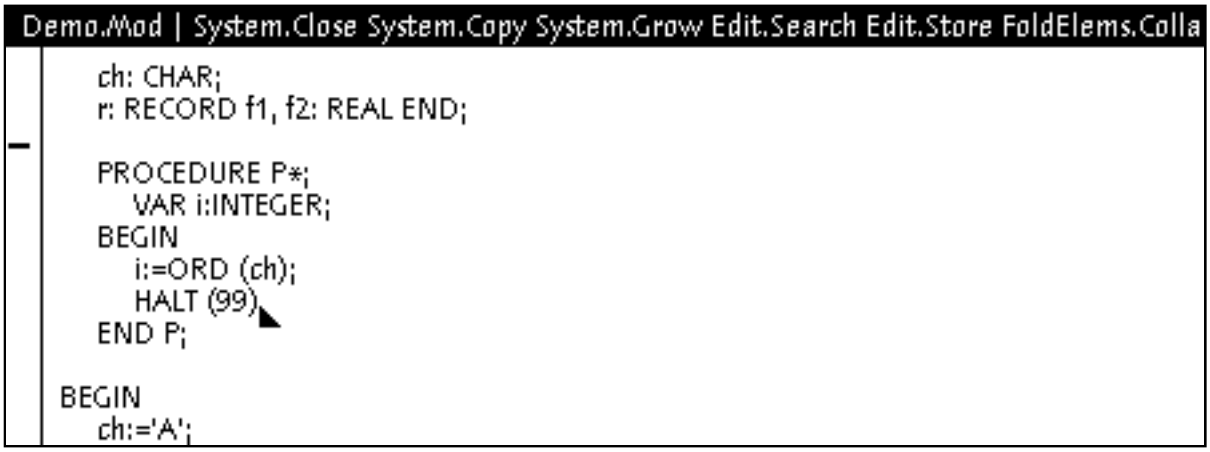


Figure 1: Caret indicating trap position

One can view the values of variables by using the command *Debugger.Data*. One can see the global as well as the local variables of a module.

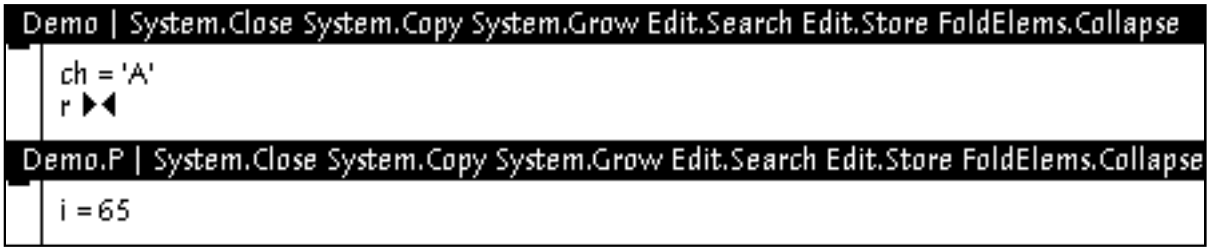


Figure 2: Two data viewers showing a global and a local scope

Variables of a structured type can be expanded to display the values of their components.

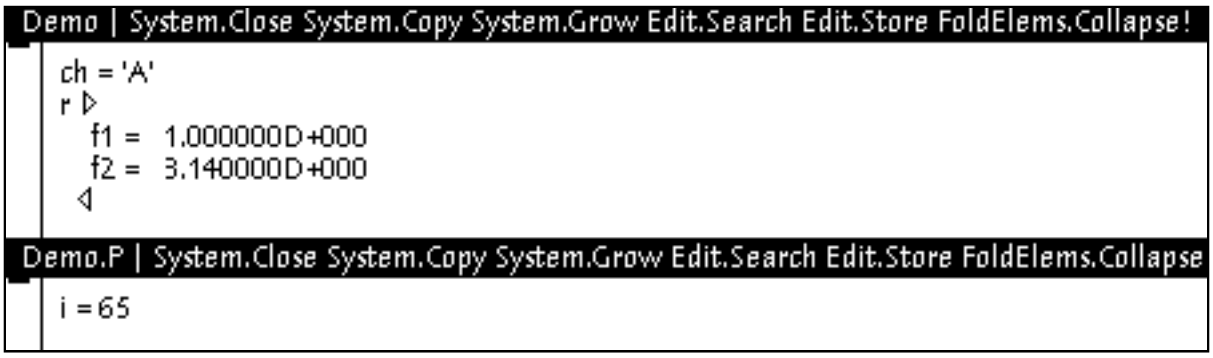


Figure 3: Expanded data viewers

2.2 Commands

Like other Oberon tools, the debugger is not a program for itself, but a collection of modules which export several commands. There are two types of commands:

- Context insensitive commands. They modify the state of the debugger and may be called at any time.
- Context sensitive commands. They give information regarding the system state during the most recent trap.

2.2.1 Context Insensitive Commands

Context insensitive commands display and change the state of the debugger. They typically do not take any argument:

- *Debugger.Install*: This command installs the debugger. The state of all further traps will be saved for later debugging.

  If a trap occurs, the debugger saves the machine state and returns the control to the Oberon system. Using commands, the saved information can later be inspected by the user.
  As the saved information is not stored on disk but only in memory, the information will be lost if either the Oberon system is terminated or another trap occurs. Therefore only the latest trap may be debugged. As the debugger has only minimal memory requirements and introduces no new delays, it is a possibility to start it by default together with the Oberon system.
- *Debugger.Unload*: This command unloads the debugger. No more debugging can be done, further traps are not recorded any more and all used memory is released.

2.2.2 Context Sensitive Commands

Context sensitive commands display information about the machine state at the latest trap:

- *Debugger.ShowModules*: Shows a list of all currently loaded modules (see section 3)
- *Debugger.ShowStack*: Shows the list of procedures which were active during latest trap.
- *Debugger.Source*: Shows the source code of the procedure specified as a parameter. The caret marks the current program position in this procedure. The procedure has to be a procedure which was active during the latest trap. If you specify the procedure at the bottom of the stack, the caret marks the trap PC.
- *Debugger.Data:* Shows all variables of the procedure or module specified as parameter.

2.3 Data Viewers

Data viewers display the values of local or global variables. They are opened with the command *Debugger.Data.* A data viewer is a normal text viewer. All commands applicable to text viewers are valid for data viewers as well.

Variables of basic types, strings, procedure variables and NIL-pointers are displayed with their names and their values. For example:

```
i = 100
ch = 'C'
str = "a string"
real = 3.14159265D+000
bool = TRUE
set = {3, 17}
proc = Texts.Notify
ptr = NIL
```

Variables of structured types (arrays, records) or pointers are displayed with their names and a pair of fold elements that can be expanded to show the components of these types (Fold elements are distributed with the standard Oberon environment). For example:

```
rec ▶ ◀
arr ▶ ◀
ptr^ ▶ ◀
```

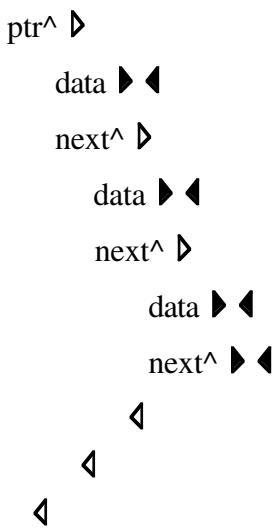
Clicking on one of the triangles shows the components of the corresponding variable. For example, clicking onto the ▶ after 'arr' leads to:

```
rec ▶ ◀
arr ▷
    0 = 17
    1 = 3
    2 = 25
  ◁
ptr^ ▶ ◀
```

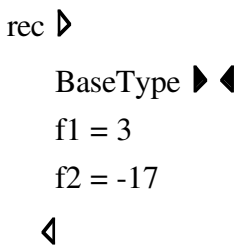
Nested structured types are display as nested fold elements. For example, a record with two fields of array type is displayed as:

```
rec ▷
    arr1 ▶ ◀
    arr2 ▷
        0 = 41
        1 = 35
      ◁
  ◁
```

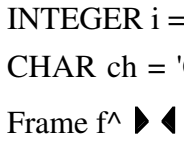
Data structures linked via pointers are displayed with nested fold elements as well. For example:



A record that is an extension of another record is displayed with fold elements for its base type to reveal the components of its base type. For example:



On demand, the debugger can also display the types of the variables (commands *Debugger.ShowTypes* and *Debugger.HideTypes*). For example:



2.4 Limitations

There are some limitations inherent to the structure of the debugger (see also section 3). There are the following restrictions:

- Debugging modules for which there is no source code. It is possible to debug such modules with two restrictions. First, the command *Debugger.Source* does not work. Secondly, when looking at variables of that module or a variable of a type defined in that module, one only gets variables and record fields exported by this module.
- In order to use as little memory as possible, not all information is copied during a trap (for details see section three). Therefore when viewing global variables and variables on the heap, one does not necessarily see the values these variables had at the time of the trap, but rather their current values. Thus the values of these variables should not be changed during debugging.

3. Implementation

To show variables together with their actual values, the debugger needs information about the location of the variables in memory as well as about their type. It gets this information from data saved during a trap and from the symbol table built by the compiler.

3.1 Saving the trap state

For a post mortem debugger, the standard technique of saving the trap state is to make a complete copy of the memory used by the application, after a trap occurred (e.g. core dump under UNIX). With the Oberon system this is more difficult. In Oberon there are no separate applications with their own address space, but all modules share the same heap. This leads to two problems:

1. The debugger would have to copy the whole Oberon heap, the stack, the global data segments of all modules and the processor registers. On small machines (e.g. 4 MB) there is usually not enough memory to copy the heap (which is usually 2MB large). Even using virtual memory, there would be extensive swapping slowing down the computer considerably.
2. A post mortem debugger that stores a core dump as e.g. under UNIX, saves the state of the application, terminates it and returns to the operating system (shell). In Oberon, a user program is not a closed application but an extension of the Oberon system. Terminating the program should therefore not terminate the system but return control to the main event loop of Oberon. Only the state of the program but not the state of the system should be saved.

To avoid the first problem, only the absolutely necessary information should be saved when a trap occurs. Certainly, the debugger has to save the current stack and registers (all together hardly more than four to five kBytes) since they surely will be overwritten if one continues to work with the Oberon system. Since any module may be debugged, the debugger cannot determine in advance which data segments have to be saved. Therefore, the debugger would have to save the data segments of all modules. This would increase memory consumption considerably. Of course, if the global data of a module is not saved, there is the danger that the user can destroy the contents of the data after the trap. But this is under the control of the user: If one does not use a module after the trap, its global variables and the heap data referenced by these variables remain unchanged.

If you continue to work with a module after a trap occurred, the debugger will display the current data instead of the data current during the latest trap.

The second of the above problems is solved by the Oberon system. The standard trap procedure does not terminate the system after a trap. We only have to make sure that the stack is copied during the trap. As there may be pointers on our copy of the stack which reference objects on the heap, we have to ensure that objects referenced by such pointers are not freed by the garbage collector. We ensure this by including our stack copy into the mark phase of the garbage collector.

To speed up debugging of local variables and to enable debugging of modules without available source code, the stack is scanned for stack frames, using the reference information included in the object file. This corresponds to the information visible in the standard trap viewer of the Oberon system.

3.2 Getting Symbol Information

The debugger also needs information on how to interpret the memory after a trap, i. e., it needs the types and the addresses of global and local variables. Most debuggers get this information from special files created by the compiler (reference files). This has several disadvantages:

- For every module compiled an additional file is created.
- The compiler has to be modified to save this information.
- The debugger has to read this file and to build a data structure similar to the symbol table built by the compiler at compile time.
- A reference file is created for every compiled module, even if a module is never debugged.

The biggest of these problem is that in order to create reference files, the Oberon compiler would have to be modified. This would make the debugger less portable, since these modifications would have to be made to the compiler of every other Oberon system the debugger is supposed to run on.

To circumvent these problems a new approach was chosen. The idea was to rebuild the symbol information on demand during debugging, instead of saving it at compile time. By this, one avoids the problems mentioned. Also, debug information is only created if it is actually needed and only for the time it is needed. On the other side, this method has disadvantages as well:

- A slight delay when accessing the symbol information of a new module, since the module has to be recompiled first.
- The source code has to be available. This is not a problem when debugging your own modules.
- A slight modification to the compiler is still necessary.
- Problems may arise, if the source of a module was modified after it was loaded by the Oberon system.

The delay for recompiling a module is so short, that it is only noticeable on slower machines. In addition, the debugger maintains a cache for the symbol information of the most recently used modules. So there are hardly any noticeable delays even on slower machines.

The modifications necessary to the compiler are negligible. We only have to keep a pointer to the symbol table constructed by the compiler in order to prevent the garbage collector from freeing this memory. This can be done by adding a single line to the compiler.

3.3 Putting it All Together

To display a variable with its actual value, the debugger needs the following information

- the static type of the variable
- the effective address of the variable
- the variable's actual run time type

The compiler provides the static type of the variable. The effective address can be calculated from the information provided by the compiler (offset of the variable relative to the base address of the corresponding module or procedure). Knowing the static type as well as its address, the run time type can be computed, as every object with a possible difference between run time and compile time type has run time type information added to the actual value.

4. Porting the Debugger to other Oberon Systems

A debugger is highly dependent on the system it is running on. Therefore it is not possible to write a debugger which is able to run on several systems without modifications. One of the goals with this debugger was to design it as portable as possible.

There are two levels of dependencies where the debugger has to be modified when ported to another system:

- Every Oberon system (except on the Ceres [Ebe87]) runs on an underlying operating system. These systems have specific features, different programming interfaces and possibly different CPU's with different registers. Luckily, all these differences can be hidden in the implementation of the trap handler. The rest of the debugger is machine independent.
- The Oberon systems themselves differ in certain respects. The run time information (about dynamic arrays and dynamic types) is not stored in the same way on different Oberon implementations. Furthermore , the interfaces of low level modules (e.g. Kernel, System, ...) differ in some places on different Oberon systems. These differences are handled in a few procedures which have to be modified when the debugger is ported.

Each of the two levels is handled in a module of its own, giving the modules depending on them a standard interface for all implementations of the debugger.

5. Conclusions

This paper presented a post mortem debugger for Oberon. The strength of this debugger stems from its concept basing strongly on the Oberon environment. The debugger has several principal features:

- It saves only minimal information during a trap. This speeds it up considerably.
- It needs no reference files
- It is small and efficient by using many features of the Oberon system.
- It is easily portable.

We have only given an overview of the debugger, showing its basic ideas, its advantages as well as its disadvantages. We plan to extend the debugger to enable trace debugging, allowing step by step execution and breakpoints. In doing so, the greatest challenge probably is, to embed the trace debugger into the Oberon system, giving it a typical Oberon like look and feel interface.

6. References

[Reis92] Martin Reiser, Niklaus Wirth: Programming in Oberon - Steps beyond Pascal and Oberon; Addison-Wesley 1992.

[Ebe87] Hans Eberle: Development and Analysis of a Workstation Computer. PhD thesis, Swiss Federal Institute of Technology (ETH Zürich), 1987. Number 8431.