Advanced diagnostics: a PASCAL interactive system

by NEIL H WHITE and GEORGE M HAYLETT

Abstract: Monitoring and controlling the execution of PASCAL programming can be a difficult task. Work carried out as part of an Alvey project aims to simplify this. The paper outlines the design philosophy and objectives of an advanced interactive diagnostics system. The results are only obtained from the initial steps of implementation. Research is still continuing to improve the system.

Keywords: data processing, programming languages, diagnostics, PASCAL programming.

An advanced interactive diagnostics system is suitable for high-level programming languages but the particular implementation described here is designed to monitor and control the execution of PASCAL programs. The PASCAL Interactive Diagnostics System (PIDS) is currently being implemented on a Perq 1 computer running a variant of the Unix operating system. The project is within the Alvey programme.*

The term diagnostics has been adopted in recognition of the higher level of problem orientation that the system enjoys over traditional debuggers, and as an indication that the subject area has advanced considerably since the days of the octal dump and program trace. This diagnostics system, although still in its early stages, goes far beyond the common traditional diagnostic provisions and includes the facilities to display any linked data structure at any level of resolution. It can also control the program's speed of execution even to the extent of running the program backwards to home in on a particular point of interest. Finally, several separate instances of this system may be set up working in parallel with each other. Each such instance would monitor a particular aspect of the program being tested. It can be argued that a high-level language system is incomplete without a diagnostic system that communicates with the user in terms, as far as possible, of his or her own algorithms.

Despite the increasing availability of debugging systems on many currently popular operating systems, sdb[1] under Unix for example, their use is still restricted to a small number of users. In some circumstances this is attributable to the limited functionality of the debugging system, or the difficulty experienced by the user when using such a system. Yet, given the higher level of sophistication available in many such systems, addressing some of the inadequacies of their forerunners, they still do not enjoy widespread use.

In general, a failed program's output may not provide the programmer with sufficient insight into the cause of the problem or the specific area of concern. In such cases, a frequently adopted strategy is what might be termed 'wolves and fences'. The metaphorical fences are erected by the programmer in an attempt to isolate the wolves, or bugs, in specific areas of the code. This strategy most often manifests itself in the form of inserted print statements, where certain variable values might be displayed. This is obviously time consuming — requiring an edit, recompilation and link — and based on some intuitive insight that may initially be flawed.

Clearly the process involved requires interrogation of data and some monitoring of program control flow — tasks ideally suited to a diagnostics system. It is important that the system performs such functions unobtrusively, and easily so that the user can extract the required information. The method of the system's display is another key issue.

Problem orientation

The term 'problem orientation' was introduced in an earlier paper[2] as a concept to explain the need for an advanced diagnostics system. Problem orientation is a

Computer Science Department, University of Keele, Keele, Staffs. ST5 5BG, UK.

*Alvey project SE/068

very crude measure of how well-suited a particular level of description is to the problem being solved by a computer program. For example, a certain high-level language may be well-suited to the problem of accessing a certain kind of database. The algorithms created by the programmer are then particularly easy to translate into program statements and data representations. The resulting machine code is unlikely to reflect these algorithms — the programmer's view of the solution — as elegantly as the high-level language does. The process of programming algorithms is two-fold. First, the programmer translates the algorithms into a programming language. Second, a compiler translates this program into a machine code version which is subsequently executed. This process can be viewed on a line of problem orientation as shown in Figure 1.

The task of programming is represented by the path from A to B. The difficulty of this task is represented by the distance from A to B. If the language used was less suitable it would have a lower problem orientation for the task in hand and the distance would correspondingly increase. One could imagine different stages of the compilation process producing several representations of the program through some intermediate language into assembly language and finally into machine code. Clearly, each such representation is further from the programmer's algorithms and it would be successively more difficult for the programmer to translate the algorithms into these representations. The reason for this apparently obvious discussion is to consider what happens when a fault occurs while the program is running. The fault manifests itself at the lowest problem orientation level. The fault might announce itself as:

```
Illegal memory access at@239AF
```

The programmer, on the other hand, would much prefer something along the lines of:

```
The pointer 'link' in the record 'index.node' has been used at line
100 but does not point at an existing data item
```

The translation from the low-level symptom into a high-level description is performed by a diagnostics system which operates almost in reverse to a compiler. Depending upon the sophistication of the diagnostics system, a translation will occur up to some point along the line B to C in Figure 1. In other words, a simple diagnostics system will provide a description of the program in terms of some language with a lower problem orientation than the high-level programming language originally used. The programmer now has to make the effort of delving beyond the point B of Figure 1. This is exactly the effort that was supposed to be avoided in the first place by using that programming language.

It can be argued that the task of a compiler is to give the illusion of working with a machine that 'understands' a high-level language. This is only half of the story and, without a diagnostics system which is capable of traversing the full path from C back to B, this illusion can be cruelly shattered. The diagnostics system described here has this ideal as its minimum aim. By inference and prompting from the user, it is intended that, when requested, it may be possible to do even better than that. By providing representations created with a knowledge of the programmer's use of the programming language's basic data types the system attempts to go some way along the route from B to A. These various levels of a diagnostic assistance are illustrated in Figure 2.

User interface

The nature of the user's interaction with the system must feature strongly in any design philosophy. The Perq 1 has a high resolution (758 × 1024 pixel) bit-mapped display, a puck/tablet and a window management system.

PIDS works in one of two modes at any one time: display or control. In display mode, interrogation of data can be performed, both as a static enquiry by the user, or as a continual display, provided by the PIDS, as the

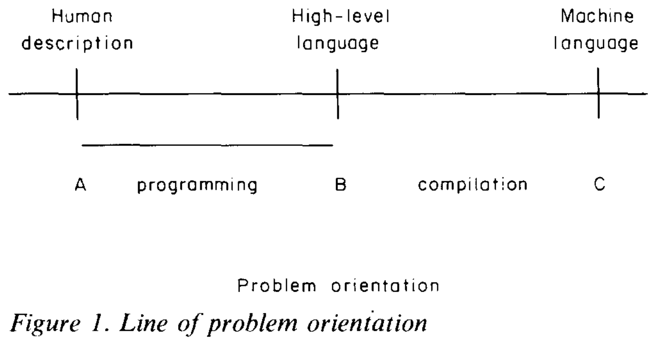


Figure 1. Line of problem orientation

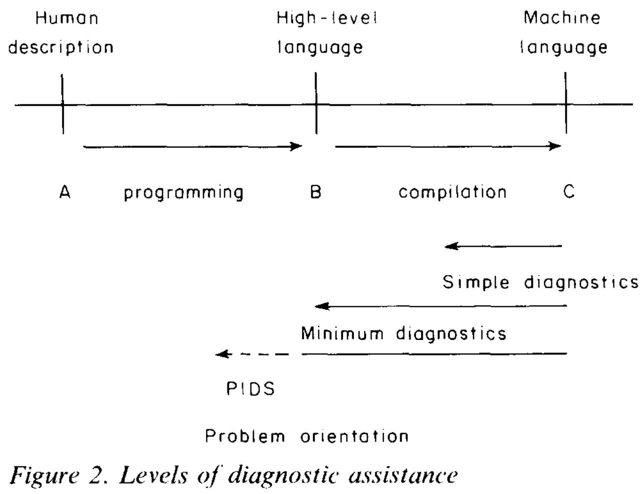


Figure 2. Levels of diagnostic assistance

program runs. All interrogation of data is most obviously done in source language terms — the user should not be exposed to a new set of syntactic conventions unique to the diagnostic system. Full advantage must be taken of the ability to display data graphically, with windowing and zoom techniques, under puck control, being employed for large objects or structures. In control mode, the program execution can be influenced and effected.

Static menus, showing the available command for the applicable mode, will be continually displayed. This avoids the tedium of nested levels of pop-up menus and the need to remember in which menu each command lives.

Interrogation of data

At any point in a program's execution, the user may wish to examine the state of particular extant variables. In order to provide such a facility the overhead to PIDS might be high, since full compiler symbol table information is required, not simply to locate a data object's core address, but also to reflect PASCAL's strong typing.

A data object in PASCAL need not have a name associated with it. PASCAL heap objects are created dynamically and anonymously. They are accessed only via pointers. PIDS must provide both an easy method of data interrogation for individual items on the heap, and also appreciate the structure or shape these objects form when linked together.

PIDS is based on a PASCAL Run-Time Diagnostics System[3]. PRTDS solved this naming problem by automatically assigning a unique name to each newly created heap object. The simplest form of unique name is an ascending integer, which also confers the advantage of automatically showing the chronological ordering of the heap objects. This strategy has worked well in the past and is adopted by PIDS.

The major use of heap objects in PASCAL is to create linked structures, where an object contains pointers to other heap objects. In this way, linked lists and trees can be created, manipulated and modified.

Algorithms for the 'pretty-printing' of lists and trees are well known, yet some prior knowledge of the nature of the structure is required. Furthermore, the display of some more complex structures does not necessarily evolve into a combination of tree and list displays. Indeed, often the user has a very personal visualization of the structures involved.

PIDS associates with each object type definition a 'display tag', denoting whether any structures formed of such elements are classed as lists, trees or something more complex.

The tag is initially set according to PIDS idea of what representation is required. The assumptions PIDS makes in forming such a judgement are based on the symbol table information associated with the objects definition. Consider the following PASCAL type definitions:

```
type node = record
  left, right :↑node;
  key: integer
  end;

 link = record
  next, last :↑link;
  key: integer
  end;
```

In the node definition, left and right would be construed as branches within a tree, whilst next and last within links imply a doubly linked list structure. If PIDS cannot make an evaluation based on the object's type name or component names, it will assume some default display tag. The user has the opportunity to fine tune the name triggers associated with each display tag, and the ability to reassign the display tag if PIDS gets it wrong.

In the case of more complex structures, the algorithms involved necessarily become more complex. Even given PIDS ability to display such structures, using adapted standard algorithms, problems still exist in reconciling the PIDS representation and the user's visualization. Where significant deviations exist, the user should have the opportunity to influence the PIDS representation by applying further constraints upon the display construction, over and above those imposed by the algorithm.

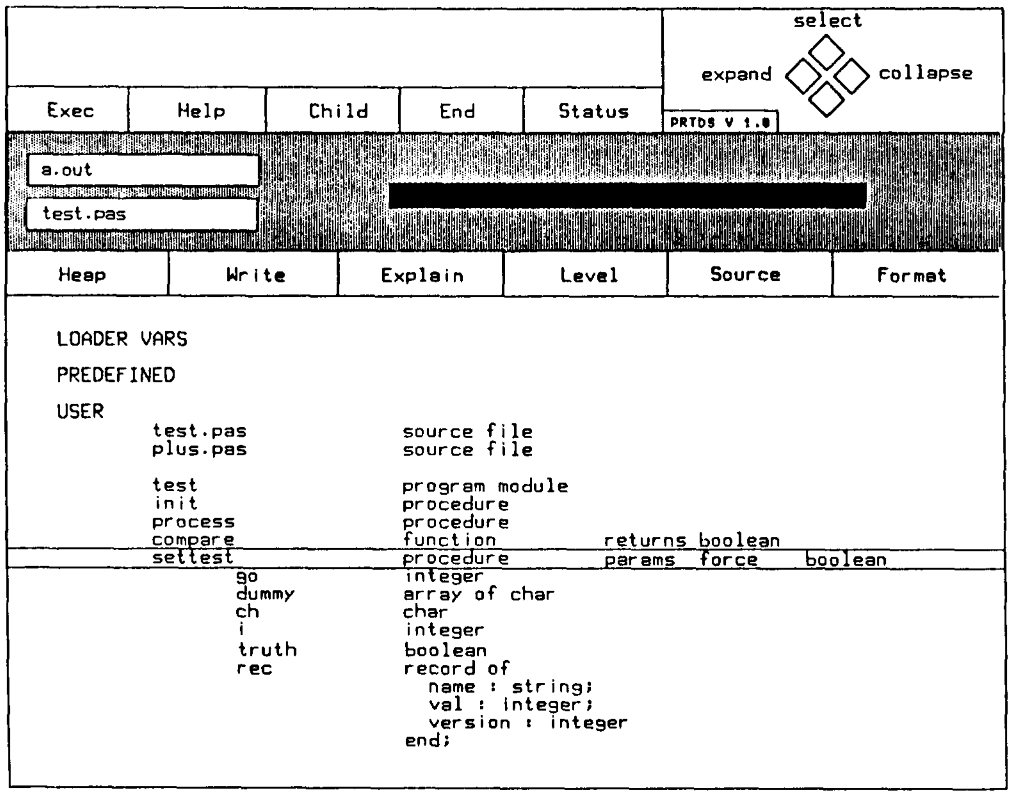
In displaying a nominated heap structure, PIDS first builds up as full a representation as possible, subject to certain system constraints. This display is then fully shown in the available window, at the scale required for it to fit. Using zoom, windowing and clipping techniques the user may then browse around the structure, expanding or contracting the view and panning in any direction.

Monitoring of control flow

The user must have the opportunity of monitoring the control flow of a program. Traditional techniques have involved the ability to set breakpoints, single stepping through the source code and the supply of profiling information. More recent advances in diagnostic techniques include the ability to set conditional breakpoints and the trapping of exceptions.

These techniques, though included in PIDS, are essentially postmortem in nature and the strategy adopted not unlike the 'wolves and fences' approach (though with significant time savings), the object of the exercise being to isolate the point at which the error occurred.

A more natural approach would be to zoom-in on the 'wolf' dynamically, as the program continues to run. PIDS will later incorporate a 'throttle' control directly affecting the speed of program execution, with the ability



```
LOADER VARS
PREDEFINED
USER
         test.pas           source file
         plus.pas           source file

         test               program module
         init               procedure
         process            procedure
         compare            function          returns boolean
         settest            procedure         params  force   boolean
                  go        integer
                  dummy     array of char
                  ch        char
                  i         integer
                  truth     boolean
                  rec       record of
                              name : string;
                              val : integer;
                              version : integer
                            end;
```

Figure 3. Display of PIDS operating in control mode

to run the program backwards i.e. undo instructions. Obviously, the program will give only the appearance of being 'in reverse' and the problems associated with providing such a facility are related to the sheer weight of information required. It is proposed that two techniques be employed:

- Full 'dumping' of all program state information at strategic points in the program's execution, either by user or PIDS control.
- After each dump, maintenance of a list of 'change vectors' will be employed, one for each flow of control change or overwriting of a store or register value. These can then be examined together with the machine code to 'undo' a program statement.

The overheads may be high, both in terms of execution speed and store requirements, but these are not seen as mitigating against the advantages that would be conferred.

Multiple invocations

It was stated earlier that PIDS should be unobtrusive, impinging as little as possible the thought processes of the user concerned with debugging. Thus, the user should have the ability to examine source code, observe and effect the control flow of the running program, whilst monitoring various aspects of the behaviour.

In a window management environment this is most obviously achieved by dedicating a particular window to a particular task. Thus, the PIDS philosophy encapsulates the ability to invoke a 'master' PIDS, concerned with program control flow, and thereafter spawning child PIDS dedicated to other aspects of the debugging process. Examples of such windows are given below.

Figure 3 shows a display of PIDS operating in control mode. The top right portion of the display indicates the commands associated with each puck button. The user may select an item currently displayed and either 'expand' or 'collapse' the information. In this way the chosen section of the program may be displayed in more detail or an overview of a greater portion of the program may be provided.

Figures 4 and 5 show a display of a complex data structure within PIDS operating in 'display mode'. Display mode operates either upon heap objects or stack objects — that is, data created both dynamically and statically. The black rectangle in Figure 4 is a two-dimensional 'thumb bar' representing how much of the current structure is currently displayed at the present scale. The scale is affected by the 'expand' and 'compress'

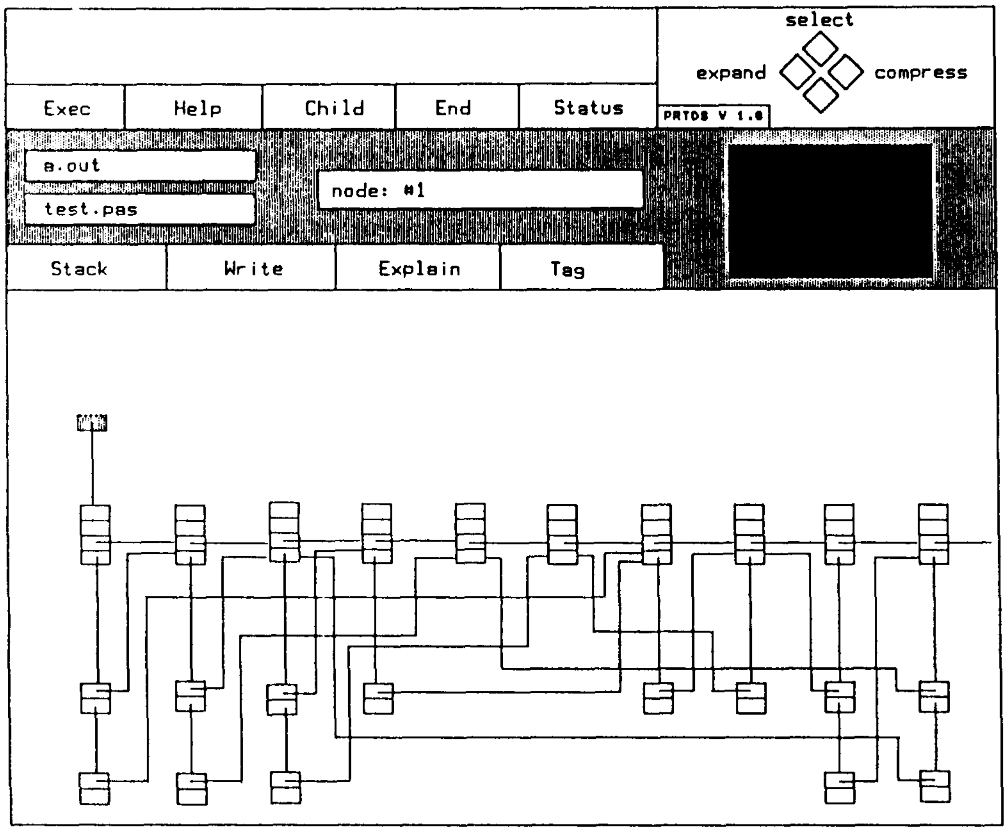
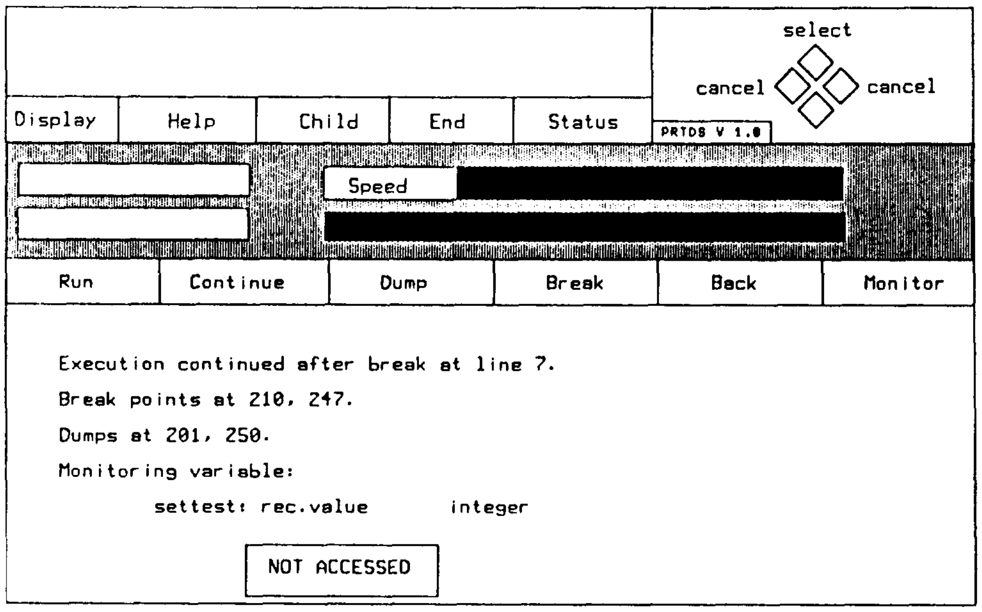


Figure 4. Display of complex data structure within PIDS operating in 'display mode'



```
Execution continued after break at line 7.
Break points at 210, 247.
Dumps at 201, 250.
Monitoring variable:
        settest: rec.value       integer

           NOT ACCESSED
```

Figure 5. Another display of complex data structure within PIDS operating in 'display mode'

puck buttons. The square is also movable within the structure.

Conclusion

As yet there is not a working example available to demonstrate PIDS. Since work is still continuing with PIDS, further results will be published in a later issue.

References

1 **Bourne, S R** *The Unix System* Addison-Wesley (1982)

2 **White, N H and Bennett K H** 'Run-time diagnostics in PASCAL' *Software-Practice and Experience* (April 1985) pp 359–367 Wiley

3 **White, N H and Bennett K H** 'A PASCAL run-time diagnostics system' *Software Practice and Experience* (November 1985) pp 1041–1056 Wiley □