BIT 20 (1980), 163–174

AN INTERACTIVE PROGRAMMING SYSTEM FOR PASCAL

JERKER WILANDER

Abstract.

Interactive program development tools are being increasingly recognized as helpful in the construction of programs. This paper describes an integrated incremental program development system for Pascal called Pathcal. Pathcal contains facilities for creation, editing, debugging and testing of procedures and programs. The system facilities are all Pascal procedures or variables and because of this allows the programmer to program the system in itself.

Keywords: Incremental programming, programming systems, debugging, Pascal.

Introduction.

Program development is today commonly supported by an interactive time-sharing system. A more advanced type of interactive program development tool is an incremental system. The basic programming cycle in an incremental system is not the execution of a program but rather a statement or a declaration. This property together with the ever present "database" of values and declarations constitutes an incremental system.

The incremental system is the foundation of the next generation of programming support, the "programming environment". A programming environment contains all the tools in one integrated system and it should support the terminal by e.g. allowing editing of incorrect statements. It should provide the facilities that the operating system monitor usually provides, to allow programmability of all facilities. In a sense the programming environment could be viewed as a programming language oriented monitor which is incrementally extensible. The most important example of this kind of system is INTERLISP [1], but APL [2] has also gone a long way in this direction. For a more conventional looking language, EL1, the ECL system [3,4] could be mentioned. The IBM PL/I checkout compiler [5] is an example of a commercial system for a conventional language that comes close to being an incremental system. For teaching purposes in computer science, a Pascal like system called Basis has been developed [6]. The basis language is a small subset of Pascal. In that system one is

Received Dec. 5, 1979. Revised April 11, 1980.

This work was supported by The National Swedish Board of Technical Development (STU) under contract dnr. 78-4167.

permitted to enter declarations, set variables, call procedures and edit procedures. The MENTOR system [7] contains many facilities for editing and program manipulation. That system does not contain any facilities for program execution.

This paper describes a programming system for Pascal [8] called Pathcal. The Pathcal systems is an attempt to provide a significant subset of the facilities provided in INTERLISP for a compiler oriented language.

The programming system Pathcal.

The goal of the Pathcal project has not been to develop another programming language, but to create, within the framework of the chosen language, an interactive programming environment. The project was intended to produce experience and ideas on how to construct an interactive programming system for the Algol family of programming languages. The experiences gained are giving more knowledge about what tools are useful in interactive programming and how they should be implemented. There are additional facilities in the system but these are not yet fully developed or tested and will not be described here.

A programming environment contains several subsystems such as an editor, a debug system and a pretty printer. In Pathcal these are tightly coupled, to allow invocation of one subsystem from another. All subsystems are Pascal procedures, although some may have privileges that are not allowed in standard Pascal.

It is possible to use the Pathcal system in a manner similar to conventional programming. One may enter the program from the terminal, compile it and execute it. In addition, there is a set of facilities in Pathcal which are not normally available in conventional systems.

Incremental execution.

Incremental execution gives the programmer the opportunity to use statements of the language and get them executed directly. A secondary effect of this is that the system becomes fairly easy to learn. One may try most facilities in the system and the language without having to write whole programs.

One language system.

Pathcal has only Pascal as command language for the subsystems. This is true except in the screen oriented text-editor, which is controlled by the cursor control keys, rubout etc. The one-language system idea is also apparent in that error messages are given in terms of the programming language. This means that it is possible to obtain the code of the incorrect statement, expression or declaration.

One advantage with this approach is that one can write loops over commands without learning an additional syntax.

Incremental program development.

With Pathcal, procedures can be developed one at a time to form programs. It is thus possible to construct and test modules of the program one at a time and later

combine them into a system. One may edit and test single procedures in an existing program. It is possible to test procedures that contain calls to procedures not yet defined. The program may be developed "top down", "bottom up" or the most critical portion first according to the preference of the programmer.

Model of the terminal-session.

Pathcal maintains a model of the terminal session and stores information about earlier interactions and their results. There are procedures in the system for re-execution and editing of previously given statements. One may also compare the results from a test example before and after editing a procedure.

Continuation after errors.

When an error occurs during test execution, the program will not abort. Instead it is interrupted, with the procedure and variable environments retained. During this interrupt, editing of procedures or use of any other Pascal statement is permitted. If a procedure is edited during an interrupt, the new version will be used in the future. It is particularly useful to be able to continue after an error in the case of interactive programs. One single error in the program or the type-in might then require a major effort when restarting the program, if it is not possible to repair the error and continue execution.

Structure editor.

With a structure editor, program editing is performed in terms of the programming language. One manipulates the program in procedures and statements instead of lines and characters. A typical structure editor operation is the insertion of the code of one procedure into another. An advantage with a structure editor is that only those parts that really are affected by the edit are changed, which means that breakpoints and similar structure above the ordinary code may remain intact. With structure editing it is natural to include programmability of the editor.

An application of the Pathcal system.

To make the system description more concrete, we will follow the development of a small Pascal procedure.

The specification of the problem is:

Write a procedure HISTOGRAM that prints a histogram on the terminal. The procedure has three parameters.

1. A: MEASURE; MEASURE is an array of measurement values of type real.
2. LEN: INTEGER; LEN is the length of A (actual length).
3. WIDTH: INTEGER; This is the desired width of the histogram.

The program should find a suitable scaling factor to adjust to the width of the screen.

Before each interaction the Pathcal system prints a number followed by a prompt character. The prompt characters differ according to the situation. Greater than (>) indicates the top level (i.e. the level the user usually is connected to), colon (:) indicates a breakpoint or an error interrupt. The number is incremented after each interaction. When using procedures in Pathcal that use the session model, i.e. that refer to earlier interactions, the number is given as an argument.

The problem will be solved through three routines, which will be developed successively. The first procedure only prints one line of the histogram. The second calculates the scale factor for the histogram while the third is the main procedure that calls the other two. The third procedure (HISTOGRAM) makes a pass through the measurements data and calls the printing routine for each item.

First declare the type of the array.

```
1> TYPE MEASURE=ARRAY <1.60> OF REAL;
  MEASURE
```

All statements have a value even in those cases when they do not have one in Pascal. Here the value is the name of the declared type.

Define the procedure that prints one line of the histogram.

```
2> PROCEDURE PRINTLINE(W:INTEGER);
  VAR I : INTEGER;
  BEGIN
    FOR I := 1 TO W DO WRITE('*');WRITTEN
  END;
  PRINTLINE
```

Test printline once to verify that it works. One line with 15 asterisks should get printed.

```
3> PRINTLINE(15);
  ***************
NIL
```

After the asterisks a NIL was written which is the value of the procedure. All procedures have a value, which is the value of the last statement executed. A statement has as value the value of the last expression executed. In this case it is a write statement, which happens to return NIL.

Define the function that calculates the maximum value in an array.

```
4> FUNCTION MAXVAL(VAR B:MEASURE;N:INTEGER):REAL;
  VAR I : INTEGER;
MAX : REAL;
  BEGIN
    MAX := B[1];
    FOR I := 2 TO N.DO
```

```
IF MAX < B[I] THEN MAX := B[I];
    MAXVAL := MAX
    END;
  MAXVAL
```

Now the function MAXVAL will be tested. One of its actual parameters is not declared, and because of this the program execution will be interrupted after an error message when the error is encountered.

```
5> MAXVAL(A,60);
  Identifier not declared.
  A
```

The primary problem in a situation like this is to find out where the execution stopped. The backtrace procedure (BT) prints the names of the currently active procedures. The list is written in calling order with the most recently called procedure first.

```
6: BT
MAXVAL
SYSBLOCK
```

The active procedures are MAXVAL and SYSBLOCK. MAXVAL was called in interaction 5 and SYSBLOCK is the always present "procedure" that surrounds all other procedures. In more complicated cases this list is longer. The Pathcal system builds the stack frame for the procedure before checking the actual parameters. Another procedure called BTV will display all variables and their current values.

To correct the error, a declaration for A should be inserted. After that, A may be given a value. The present problem is that this declaration should be inserted in SYSBLOCK but MAXVAL has already been entered. Of course, A could be declared in this position (MAXVAL) but with the disadvantage that the declaration and the value would disappear upon exit from MAXVAL.

The procedure stackset moves the execution environment to the desired position. When all necessary commands have been executed in this new environment, execution may be resumed at the original point.

```
7: STACKSET(SYSBLOCK);
SYSBLOCK
```

Notice that the prompt character changes to !. This demonstrates that the execution environment has been moved from the standard environment.

```
8! VAR A:MEASURE;
A
```

A top level declaration like this is never inserted into the program text, but only in the current execution of the procedure. To achieve the latter, the source has to

be edited. In this case the procedure should not be edited because the declaration should be inserted into SYSBLOCK.

```
9! FOR I := 1 TO 60 DO A[I] := 0.5*I; (* Initialize A *)
  30.0
```

This statement is not included in the program either. Instead it is executed and the array A is initialized. Correction of errors during execution can create inconsistencies. These can occur when the user changes declarations on variables whose values already have been used. This problem is in practice not very serious, because the main reason the user at all continues the execution is to find more errors. Changed declarations are indicated by the system.

Return to the original execution.

```
10! OK;
```

Resume execution at the point of error, but now with all declarations inserted. The execution should now proceed correctly.

```
11: RETTO;

30.0
```

At last, the result appears which was requested in interaction no. 5.

```
12> A[17] := 45;
  45.0
```

Change input data to MAXVAL for another test.

Instead of rewriting all of the statement maxval(a,60) one may refer to an earlier interaction and execute that code. This facility is useful in cases where a lot of typing has to be performed, e.g. interaction 9.

```
13> REDO(5);
  45.0
```

Define the procedure HISTOGRAM.

```
14> PROCEDURE HISTOGRAM(VAR A:MEASURE;
                        LEN, WIDTH:INTEGER);
  VAR MAX:REAL;
    I : INTEGER;
  BEGIN
    MAX := MAXVAL(A,LEN);
    FOR I := 1 TO LEN DO
      PRINTLINE(ROUND(A[I]/MAX*WIDTH+0.5))
  END;
HISTOGRAM
```

Test all procedures together.

```
15> HISTOGRAM(A,5,10)
**
****
******
********
**********
NIL
```

Change input and redo the test.

```
16> A[5]:=100;
100.0
```

Re-execute interaction 15 which is two back from the current interaction.

```
17> REDO(-2);
*
*
*
*
**********
NIL
```

At this point the procedure has been tested in an elementary way and now further test data could be applied. Test data may easily be modified to construct a relatively complete test.

The HISTOGRAM module now contains three separate procedures. If this is considered inappropriate, they may be combined into a single procedure using the structure-editor. The insert procedure performs this combination.

The insert procedure knows how procedures are inserted into other procedures.

```
18> INSERT(HISTOGRAM,MAXVAL);
HISTOGRAM

19> INSERT(HISTOGRAM,PRINTLINE);
HISTOGRAM
```

We now want to verify that the insertion worked as expected, and that the module has the correct structure. The pretty print procedure gives an indented printout of the procedure text. The parts of the printout that are marked #*n*# where *n* is a number are abbreviations of the actual code in that position. They are only abbreviations in the sense that "behind" them the actual code could be found. This facility makes it easy to view the procedure at different levels of abstraction.

```
20> PP(HISTOGRAM);
PROCEDURE HISTOGRAM(VAR A: MEASURE;
                    LEN, WIDTH:INTEGER);
  VAR
    MAX:REAL;
    I:INTEGER;
  FUNCTION MAXVAL(VAR B:MEASURE; N:INTEGER):REAL;
    VAR #1#
  BEGIN #2#;
    FOR I := 2 TO N DO #3# ;
    #4# END;
  PROCEDURE PRINTLINE(W:INTEGER);
    VAR #5#
  BEGIN
    FOR I := 1 TO W DO WRITE('*');
    WRITELN
  END;
  BEGIN MAX := #6# ;
    FOR I := 1 TO LEN DO #7#
  END;
```

The "print depth" of the pretty print is restricted by how deep the nesting and how long the sequence of statements and expressions are. The print level is program settable.

Redo the test to verify that the edit gave a functionally correct procedure.

```
21> REDO(15);
*
*
*
*
**********
NIL
```

Find out what happened in interaction no. 5.

```
22> HISTORY(5);
  5. MAXVAL(A,60);
     Value= 30.0
```

This example shows how an incremental system with the set of powerful tools found in Pathcal gives an extremely convenient programming environment.

Only simple usage of the debugging, editing, pretty-print and session support tools have been demonstrated. This relatively small set of utility functions are the most important building blocks of Pathcal. Among the things not described are breakpoints and the text editor of the system.

Description of the system.

Pathcal works as if one executed the program statement by statement and declaration by declaration, though with the difference that one is not required to obey a strict ordering between statements and declarations.

When declarations are entered into Pathcal they are accumulated to a special block (in the sense of Algol) called the system block. The statements entered are executed immediately. Executable statements are not accumulated to form some kind of main program. The difference between declarations and statements is less accentuated in Pathcal than in Pascal: both are executed. Statements give the result that variables get values or values get printed, and declarations that a specific procedure or variable may be used in the future. Variable values are saved on a stack, which is maintained by the system and which works in the same manner as a conventional stack for a block structured language.

In the system block there are several different declarations, including all procedures, types and variables the user has entered. It contains all subsystems needed for usage of the system e.g. history, edit, and break procedures.

To allow smooth editing (mainly structure editing) the data-type code has been added to Pathcal. Code is a data type for program text in structure form. Variables of type code have as value Pascal statements, procedures, variable declarations etc. This data type makes it possible to manipulate program code from a Pascal program. For example, one is allowed to move code from one procedure to another or collect code from different interactions to form new procedures.

Some of the Pathcal system procedures do not strictly obey the rules of Pascal for procedures. They sometimes allow a variable number of arguments (like READ and WRITE) and sometimes the arguments have a special status (like PP). This latter difference is similar to the procedure parameter of Pascal. Both of these changes are important, but the necessary syntactic modifications of the surface language are small.

The system block includes all user programs, procedures and other declarations. When interacting with the system one is normally connected to the system block. Interaction occurs also in other situations, primarily in the debug package (at a breakpoint or after an error). During a break it is possible, in the same manner as in the system block, to enter new declarations and to assign values etc. The main difference is that declarations are not entered into the system block but into the presently active block, which is the block of the last called procedure.

Conventional techniques compared with Pathcal.

In a conventional interactive programming environment there is little programming support. There is an editor, a compiler and a debugger. In this section a brief comparison with existing interactive systems, in particular the PL/I checkout compiler, will be presented.

The working cycle in conventional interactive program development is edit, compile and execute. In Pathcal the program is defined and tested piecewise. An expected working habit when using Pathcal is to build a database of procedures and later combine them into programs. The PL/I checkout compiler system does not allow partial programs to exist and does not support incremental definition of the program.

Traditional editing is performed using a text editor that knows nothing about the programming language we work with. The editor in Pathcal on the other hand recognizes Pascal symbols, statements and overall program structure.

Usually the compiler is a black box. However, the translator in Pathcal stops when a syntactic error is encountered and permits the programmer to edit the source and resume the translation until another error occurs. This means that there will not be a cascade of error messages because of one single error. After each error one may correct the source of the error, and the corrected program is the object of the parser from then on. The PL/I checkout compiler uses an intermediate form between the Pathcal technique and conventional techniques. This compiler uses a technique for error correction which tries to correct misspellings and other errors. If the programmer accepts the attempted correction no further editing is needed. If an error is encountered, for which the compiler cannot suggest a correct change, one must exit from the compiler, enter the text editor and after editing restart the compiler from the beginning.

In most interactive debugging systems, values of variables can be read and set. However, it is usually neither possible to call procedures in the program nor to enter new declarations. In Pathcal all operations are allowed in a breakpoint or error break, of course including resuming the program execution. It is possible to call any procedure, create new declarations and edit the running program. In the PL/I checkout compiler statements may be added or deleted from the code of the running program, and statements may be executed in "direct mode". No declarations may be entered or deleted in direct mode.

Support of program testing is only rarely found. Those aids that are available are mainly test-data-generators and statement frequency counting programs. Testing in Pathcal allows interactive or programmed comparison of the results. Verification of testing in Pathcal may be enhanced by special test procedures attached to statements in the code. These test procedures are executed when the statements, to which the specific procedure is attached, are reached. The test procedures could be viewed as a programmed precondition. This approach encourages the building of test libraries and verifiers for the programs.

There are some disadvantages with the approach used in the Pathcal system. Most of these disadvantages are considered to be manageable. The most serious difficulties are:

1. The Pathcal system assumes that the program, after development or maintenance, is to be moved into a production environment. This is also the case of the PL/I checkout compiler. This raises the issue of compatibility between the Pathcal system and its production compiler.
2. In a compute bound program efficiency might be critical even when testing the program. This might be remedied partly by specialized compilers within the framework of Pathcal.

Implementation.

A fundamental idea when constructing Pathcal was that all subsystems should use a single internal notation. This internal notation should be "weakly consistent" in the sense that only syntactically correct programs are stored. Semantic correctness is verified only at runtime. This limitation means that we always have well-formed data structures to manipulate internally, which in itself increases the security and the simplicity of the system. The semantic checks are performed during runtime to allow a high level of incrementality. Without this restriction it is difficult to allow procedures and functions to be defined successively. Naturally one may perform full compilation of a program if it is desired but this seems unnatural in an incremental environment. The internal notation of a Pascal expression is the corresponding parse tree. Comments and break points are associated to this internal notation via hash links. The interpreter simulates the Pascal stack and variable look-up mechanism by symbolic search of the stack for names. The efficiency of the current interpreter is fairly low, partly because the symbol table and data representation has not been optimized and because the system is written in Lisp. In spite of this the Pathcal system only required approximately 75% of the cpu time required by the standard compiler [9] and editor to complete the example given above. This performance gain is due to the incremental system, which has no overhead in program startup time and requires fewer compilations. One edit and rerun of the program required 30% less time in Pathcal. A single run of the program required 30% more time with Pathcal. The loop in the HISTOGRAM procedure was approcimately 25 times slower in Pathcal than with the compiled system. It is possible to construct cases where Pathcal performs worse, but the system is not intended to be used for productions runs.

Conslusions.

To-day interactive incremental program development is available only for users of Lisp and APL. This type of program development has been described by E. Sandewall [10].

The experiences with Pathcal show that it is both possible and reasonable to build an incremental programming system for Pascal. Most ideas included in the

system are adjustments of the facilities in Interlisp. Lisp has advantages over "syntax oriented" languages in its simplicity of representation. Code and data have the same internal representation, and one never needs to use a parser after manipulation of code. In APL there is no canonical representation and thus it is necessary to "unparse" the function before editing it and afterwards parse it again. Lisp has fewer data types than Pascal, which allows a more efficient implementation of dynamic type checking. INTERLISP provides a set of command languages for the editor, break-package etc. This implies that the usage of a new utility system requires learning a new syntax. The command languages of INTERLISP are mostly fairly simple to learn, but as soon as one wants to perform a more complicated operation or if one wishes to combine several commands the system becomes more difficult to use. In Pathcal the command language is part of the programming language and thus it is natural to combine commands into user defined commands. The all-inclusive system block, where test data and declarations are kept, has been a very successful technique. This gives convenient testing of single procedures and a good environment for the extension of the system, both by the user and the system implementor.

Acknowledgements.

Kenth Ericson, Jim Goodwin, Gunilla Lönnemark and Erik Sandewall have been most helpful in preparing this report or in the construction of the system.

REFERENCES

1. W. Teitelman, *Interlisp Reference Manual*, Xerox Palo Alto Research Center., Oct. 1978.
2. *Dec system 10*, APLSF Programmer's reference Manual. DEC-10-LPLSA-A-D, 1976.
3. *ECL Programmer's Manual*, Center for Research in computing Technology. Harward University, Cambridge Mass., Dec. 1974.
4. R. M. Balzer, *Language-independent Programmer's Interface*, Information Sciences Institute, University of Southern California. ISI/RR-73-15 March 1974.
5. *OS PL/I Checkout Compiler*, CMS Users Guide. IBM 1976. SC33-0047/2.
6. R. P. van de Riet, *Basis—an interactive system for the introductory course in informatics*. IFIP 77, Gilchrist B. (Ed.), North Holland 1977.
7. G. Huet, G. Kahn, P. Maurice, *Environment de programmation Pascal. Manuel d'utilisation sous SIRIS 7/8*, IRIA Nov 1977.
8. K. Jensen, N. Wirth, *Pascal User Manual and Report*. Lecture Notes in Computer Science (18). Springer Verlag. 1977.
9. E. Kiscki, H.-H. Nagel, *Pascal for the DEC-system* 10, Mitteilung NR. 37, IFI-HH-M-37/76. Institut für Informatik der Universität Hamburg, Nov 1976.
10. E. Sandewall, *Programming in the interactive environment.—The Lisp experience*. Computing Surveys, March 1978.

INFORMATICS LABORATORY
DEPARTMENT OF MATHEMATICS
LINKÖPING UNIVERSITY
S-581 83 LINKÖPING
SWEDEN