SOFTWARE

PRACTICE & EXPERIENCE

VOLUME 12, No. 12

DECEMBER 1982

EDITORS

DAVID BARRON
CHARLES LANG
DAVID HANSON

175 YEARS OF PUBLISHING
1807 1982

JOHN WILEY & SONS
Chichester · New York · Brisbane · Toronto · Singapore
A Wiley-Interscience Publication
SPEXBL 12(12) 1085–1173
ISSN 0038-0644

'FINGERPRINTING'—A TECHNIQUE FOR FILE IDENTIFICATION AND MAINTENANCE

D. R. MCGREGOR AND J. A. MARIANI

Department of Computer Science, University of Strathclyde, Glasgow, Scotland

This short note describes a technique, which we have called 'fingerprinting', to produce a quasi-unique identifier for a file, derived from that file's contents. This allows the identification of identical files with different names and the notification of any changes to a known file.

KEY WORDS File identification Sum check Information content Documentation Programme support environment Source code control

One problem associated with current operating systems is the easy proliferation of multiple copies of programs and data. While this is not only wasteful of space and other resources, it can cause problems when associated documentation has been connected with a program by the program's file name.

The 'fingerprinting' technique described in this short note outlines a solution whereby information about a file is connected to that file by means of the file's contents, as opposed to its name.

The idea is to provide an identifying feature for every file, which is intrinsically distinctive, and analogous (hopefully) to a human's fingerprint. In our current implementation, a fingerprint is a 32-bit number, formed simply by performing a double check-sum on the program file, and providing a unique identifier for the file.

A fingerprint is calculated as follows—given a file with N characters, the double checksum ($s2$) is formed by the algorithm—

```
s1 := 0; s2 := 0;
for i := 1 to N do
begin
  s1 := s1 + c[i];
  s2 := s2 + s1
end
```

where $s1$ and $s2$ are 16-bit integers (overflow is ignored) and $c[i]$ is the ith character of the file.

These 2 16-bit numbers are then stored as a 32-bit fingerprint. From an information theoretical standpoint this algorithm does not generate the best possible checksum. The chance of a clash in fingerprints is therefore somewhat higher than the 'ideal' value (1 in 2^{32}), but the simplicity of the algorithm makes for a simple and very fast fingerprinting program.

Hence, we are using the file's content, as opposed to the file's name as in conventional systems, to identify the file. In a simple fingerprint (FP) system, for example, we can maintain a database of FPS against associated documentation.

Therefore, when the user comes across an undocumented file he can try to match its FP against one in the database in an attempt to obtain any information regarding that file. To build up the FP database, users are free to add new files to the system, as long as they are willing to enter appropriate information regarding that file.

Systems can naturally range upwards in sophistication from the simple one described above. In the case of program files, to ensure correct identification, the FP database could also hold test data and matching output for the program (for example, a Pascal compiler could have a Pascal program as data and the corresponding compiled listing as output) which can be checked with the output of the unknown program (by fingerprints again). In the event of any difference, a more sophisticated comparison of the programs could be called into play.

The fingerprinting technique, as outlined this far, is best used statically—i.e. with programs which are rarely expected to change. The FP database can be extended to handle different versions of a program, while programs (or program versions) which are no longer supported can have the updated version's FP overwrite the old FP.

Static fingerprints can be used to detect any file corruptions. One use FP was put to was as an assurance of correct software transportation. Every file transported was fingerprinted at our installation, and a list of FPs plus a copy of the fingerprinter sent with the software. The fingerprinter was then employed at the destination to confirm the safe transportation of the files involved.

The main use of the FP technique has been in conjunction with the AutoProg system,[1] which essentially (for the purposes of this paper) maintains a library of user source modules accessible by all users but maintained by their owners within ther own file store. In this case, the files can be expected to change, and the fact that a file has been altered can be exposed by occasionally re-fingerprinting all the modules and comparing these dynamic FPs against the FPs as recorded in the AutoProg database.

If modifications are detected, users of the affected modules can be notified and re-compilation

Received 26 July 1982 Revised 21 September 1982

of affected programs can be automatically generated. Similarly, FPs can be extended to detect changes in the associated documentation files.

In this short note we have presented a very simple technique for connecting information about a file with that file's content. This method shares the drawback of associating documentation with a file name, as both a file's name and contents can be transient. However, the FP technique can be usefully applied in the diametrically opposing situations where files are relatively static or where changes to files are required to be trapped, such as the AutoProg system or as a detection mechanism for file corruption.

REFERENCES

1. D. R. McGregor and J. A. Mariani, 'AutoProg—a software development and maintenance system', *IUCC Bulletin*, Summer 1981.

Book Reviews

INTRODUCTORY ALGOL 68 PROGRAMMING, D. F. Brailsford and A. N. Walker, Ellis Horwood, Chichester. No. of pages: 281. Price: £14.00 (hardback), £5·95 (paperback).

Algol 68 is not an easy language to teach. This text on Algol 68 is intended both for undergraduates attending their first course in programming and for computer programmers and system analysts in industry, research and commerce. I am afraid that the book's approach is such that most beginners will soon be put off programming. Even for programmers conversant with Fortran, Basic or Cobol, a gentler introduction to some of the concepts would be helpful.

For a first course in programming I believe it is necessary to have an approach led by a need to write programs in order to solve problems rather than a breadth-first approach through the elements of a programming language. One requires a gentle lead-in to the way that objects are represented in programs rather than know all about objects, then find out how programs are written and finally find out how objects are manipulated within programs. In another context, there is the phrase *Algorithms + Data Structures = Programs* where the operator + evaluates its operands concurrently. In this book the right hand operand of + is evaluated first, then the left hand operand and finally the + is done in order to yield some programs.

Thus, chapter 1 is about objects. It introduces the modes, **real**, **int**, **bool** and **char**, row modes and structured modes. The chapter also mentions heap generators and dope vectors and has a very heavy dose of **refs** including:

ref ref int chain = heap ref int
:=heap int :=123

In my opinion this chapter contains too many new ideas.

For example, consider the very first coverage of arrays. There is one paragraph explaining in general terms that it may be useful to regard a group of objects as forming a composite object and that Algol 68 has row modes and structured modes in order to do this. The next paragraph contains an example of an object of mode []**real** and introduces the notation [,]**real** and [,,]**real** and the final paragraph explains how a [,,]**real** is represented inside a computer. I feel that most of this material should have been left until a later chapter.

Chapter 2 is called 'Program Structure'. It starts with sections on 'primaries', 'expressions', 'unitary clauses' and 'serial clauses'. This is an important chapter because it contains some of the essentials such as assignments, expressions, conditional clauses and loops. However, there is too much theory: there is not enough demonstration of how these constructs are used and put together. For example, at the start of this chapter, the first Algol 68 program of the book is presented. It contains two heap declarations, two calls of *read* and a call of *print*. The second Algol 68 program of the book appears at the end of the section on 'serial clauses': it contains three heap declarations, a call of *read*, an obscure **while** loop and a call of *print*. In between the two programs, the fragments of Algol 68 that are given are not presented as fulfilling some need.

Chapter 2 continues with a discussion of scopes and ranges. Until now heap generators have been used. However, after a discussion of how runtime storage is organized in terms of a stack and a stack pointer, the book points out that heap storage cannot be handled in this fashion and so it introduces **loc**. Abbreviated declarations are then covered.

Chapter 3 takes some elementary problems and produces programs to solve them. The chapter starts by producing many programs to solve a