Multidimensional Tree-Structured File Spaces

Douglas N. Kimelman
Department of Computer Science
University of Manitoba
Winnipeg Manitoba Canada

ABSTRACT

Development projects are often based on large collections of information. This information is typically maintained in a set of files. Current file system and database structures are inadequate for storing and manipulating this information.

This paper defines a new class of high-level data structures called "m-dimensional n-ary tree-structured spaces" (also called "md-nt spaces"), and discusses their use in the organization of the files of development project file systems.

The md-nt space organization allows the primary file system structure to represent relationships between files which are difficult or impossible to represent with conventional structures. A fundamental aspect of this organization is that arbitrary semantics, such as implications in terms of a software module's nesting, or in terms of its position within a sequence of revisions, may dynamically be associated with this structure. This enables the primary file system structure to become the basis for automatic system functions such as the recompilation of source code, the propagation of sets of changes to all appropriate files, or the storage and regeneration of revisions of a file. With conventional systems, such operations are accomplished in a manual, ad-hoc, and error-prone fashion, or via mechanisms which are external to the primary file system structure, and hence contrary to the goal of an integrated environment.

This paper also discusses some of the considerations in the design and implementation of such a file system. A prototype implementation is described, and an evaluation of the effectiveness of md-nt spaces in a software development environment is presented.

INTRODUCTION

Computer systems are often the primary facilities underlying development projects in a number of different areas. Such areas include software development, document preparation, image processing, graphics generation, circuit board layout, VLSI chip design, and many other areas of design and manufacturing.

These projects are often based on large collections of information. This information is typically maintained in a set of files.

For software development, the information which is to be managed for the programming and maintenance stages ranges from very high-level definitions, to high-level textual "source code" from which the "end product" is automatically generated, to intermediate representations, to the target machine code.

Document preparation may require source text containing formatting control words, displayable formatted output which may be marked up by reviewers, and phototypesetter data streams.

VLSI design may require that logic diagrams, circuit schematics, and geometry layouts all coexist, along with block diagrams, high level specifications, and the results of various analyses of a circuit.

In addition, for projects in any of these areas, this variety of information is often accompanied by information relating to project planning and scheduling, directions and procedures for processing files, characteristics and attributes of various files, relationships between files, and logs and histories of activities related to various files.

Further, there are typically numerous evolutionary versions of the various files. Some are archived and maintained only for the sake of posterity. Others are still active. For software systems, there might be three or four different releases and sub-releases in production, others being tested, and yet others in early development. Often, a progression of revisions for a given file exists within a given release of a system. For documents, a number of drafts may exist for each edition of a document.

As well as evolutionary versions of a file, numerous alternative variations of a file may exist. For a software development system, these may depend on which machine and system will eventually act as host to the resulting code. Variations may also exist according to which set of selectable features have been chosen.

Throughout this paper, software development projects will be used as an example of projects which have relatively demanding and reasonably well formalized information organization requirements.

INADEQUACIES OF CURRENT STRUCTURES

Current file system structures are inadequate for storing and manipulating development project information.

The UNIX file system [Ritchie and Thompson 1974] (UNIX is a Trademark of Bell Laboratories) exhibits a number of the inadequacies which are common to contemporary file systems. UNIX attempts to represent many different properties of a collection of files with a single hierarchical structure. Pathnames such as '/ u / smith / mydbms / cmds / source / 68k / newversion / util.c' are not uncommon. With systems such as UNIX, the file hierarchy becomes extremely cluttered, the manipulation of files becomes quite cumbersome, and the maintenance of large programs becomes quite complex and error-prone. In some cases, UNIX utilities such as RCS [Tichy 1982] or SCCS [Rochkind 1975] are used to maintain all of the versions of a particular source file in a single space-efficient archive (incorporating a differential base+delta storage scheme). In these cases, maintenance procedures are further complicated by the fact that these versions must be accessed via special-purpose commands. The versions can not be manipulated by the standard UNIX utilities. UNIX also makes use of "makefiles" [Feldman 1975] which describe dependencies between files, and procedures for processing and updating files. Such dependency information is external to the primary file system structure, and thus is potentially redundant. As well, the standard utilities are not aware of such information. Thus this information is not applied in many situations where it might be of further use.

Current database systems are better able to represent various relationships between a number of files in a uniform fashion, but are too cumbersome to be effective in a development project environment.

A number of authors have addressed the inadequacies of current file and database systems. Thall's KAPSE for the Ada Language System [Thall 1982] (Ada is a trademark of the Department of Defense, Ada Joint Program Office) formalizes a UNIX-like hierarchy by explicitly specifying directories as being either groupings of subordinate files, or groupings of revisions of a file, or groupings of variations of a file due to such factors as additional features or different target machines. As well, Thall formalizes the notions of file attributes and secondary associations between files. He provides automatic selection of file variations based on a desired set of attribute values.

The TRW "Software Productivity System" [Boehm et al. 1982] incorporates a hybrid "master database" which consists of a relational database coupled with a hierarchical file system and a version control system.

Cheatham's "Program Development System" [Cheatham 1981] for the ECL Programming System uses a relational database with a fixed set of attributes to store the modules of a software development project.

Goldstein and Bobrow's "Personal Information Environment" [Goldstein and Bobrow 1980] for the SMALLTALK system is based on "layered networks", in which each element of an unordered set of contexts is a linear array of partial module networks.

In general, none of these systems fully combines a powerful generally applicable structure, with high level operations, and with formal semantics for automating various file oriented procedures.

Many language-directed systems such as CADES [McGuffin et al. 1979], Mentor [Donzeau et al. 1980], and "The GANDALF Software Development Environment" [Habermann and Notkin 1982] concentrate on the use of a hierarchy, possibly with labelled edges, to represent (with very fine granularity) the abstract structure and some of the semantics of a particular item such as a program, or a module, or a document. These systems devote little attention to the more global organization of these trees within a development project environment.

This paper provides a brief overview of a file system structure which is currently being developed by the author. The major contributions of this research are: a formal high-level organization for previously ad-hoc collections of files, a powerful means of viewing and manipulating these files, the formal association of semantics and consequences with file system structures, and the integration of traditionally distinct areas such as the primary file system structure, version control, and configuration management. These advances will also be applicable in areas outside of software development, such as document preparation, VLSI design, graphics, and others.

PROPOSED FILE SYSTEM STRUCTURE

The structure developed as a result of this research is one in which each file of a file system is regarded as a point in a "multidimensional n-ary tree-structured space" (also called an "m-dimensional n-ary tree-structured space", or an "md-nt space").

Each point in an md-nt space has m coordinates, one for each dimension. Where the axes of a conventional space are linear, each axis of a tree-structured space is a tree (or, in fact, a singly-rooted network). Each coordinate of a point, then, is a pathname derived from the axis tree for the corresponding dimension. As an example, a point in a four dimensional space, with the pathname 'compiler . parser .

treemgr . addnode' as its coordinate in the 'function' dimension, and with the pathname 'vax . 11_750' as its coordinate in the 'host' dimension might be identified by

```
    [ function : compiler . parser . treemgr . addnode ;
      version : release1 . subrelease2 ;
      host : vax . 11_750 ;
      phase : source . definitions ].
```

Such a space may be regarded as a cartesian product of the m n-ary trees which are its axes.

For a particular software development project, a space could be defined to have a dimension in which a point's coordinate reflects the function of the corresponding file, and a dimension in which a point's coordinate reflects the version of the file. The axis for the function dimension might be

```
                         ...
                        /
                   compiler          ...
                 /     |     \
           scanner   parser   coder
                     /     \
           ...    pda      treemgr      ...
                          /    |    \
             ...  create   adopt   remove  ...
```

and the axis for the version dimension might be

```
              0
            /   \
           1      2
         /   \
       1.1   1.2                  .
```

(Note that the values of the version axis which are below the second level of the tree, e.g. 1.2, are automatically qualified relative to the second level of the tree, e.g. 1, when they are displayed). A point (or file) in such a space would be

```
[ function : compiler . parser . treemgr . create ;
  version : 1.2 ]  .
```

Although the space considered in this section has a single axis for each dimension, this need not always be the case. It is possible, in general, for one subspace to have a different axis in a given dimension than another subspace. Thus it is possible, for example, for different functions to have different versions. The issues of varying axes and orthogonality are discussed in another paper currently being prepared by the author.

The points of an md-nt space may be connected. Each point would have a set of links in each dimension, which would connect it to its children in that dimension. For the space being considered here, children of the point

```
[ compiler ; 1 ]
```

in the 'function' dimension would be

```
[ compiler . scanner ; 1 ]
[ compiler . parser  ; 1 ]
[ compiler . coder   ; 1 ] .
```

In the 'version' dimension, children would be

```
[ compiler ; 1.1 ]
[ compiler ; 1.2 ] .
```

Such a structure may be traversed in the conventional fashion, by moving from one point to the next, along one of the links from a parent to a child. Each such move can be taken along any one of the dimensions of the space. For example, in order to get from the point

```
[ compiler ; 1 ]
```

to the point

```
[ compiler . parser . treemgr ; 1.2 ]
```

one could move, in the 'function' dimension, from

    [compiler ; 1]

to

    [compiler . parser ; 1]

to

    [compiler . parser . treemgr ; 1]

and then, in the 'version' dimension, to

    [compiler . parser . treemgr ; 1.2] .

Alternatively, one could move, in the 'function' dimension, from

    [compiler ; 1]

to

    [compiler . parser ; 1]

and then, in the 'version' dimension, to

    [compiler . parser ; 1.2]

and then, in the 'function' dimension, to

    [compiler . parser . treemgr ; 1.2] .

Thus, an md-nt space may also be regarded as a colored directed graph, in which an edge is colored according to the dimension in which it links its initial and terminal vertices.

The points of an md-nt space can be projected in various ways, in order to provide a number of different views of the entire space. When performed in the context of a suitable view, tasks such as extracting all of the source files for a particular release of a software system, or extracting all of the releases for a particular source file, which can be quite complicated with conventional file systems, become straightforward.

For the space being considered here, the projection of the points onto a single hierarchy, by version within function, could be displayed textually as

```
      . . .
     compiler
             ; 0
             ;    1
             ;        1.1
             ;        1.2
             ;    2
         scanner
                 ; 0
                 ;    1
                 ;        1.1
                 ;        1.2
                 ;    2
         parser
                 ; 0
              . . .
      . . .                          .
```

An alternative projection, by function within version, could be displayed textually as

```
0
         ; . . .
         ; compiler
         ;     scanner
         ;     parser
         ;         pda
         ;         treemgr
         ;           . . .
         ;
    1
             ; . . .
             ; compiler
             ; . . .
    . . .                     .
```

An md-nt space can be sliced across various dimensions, along various coordinates in other dimensions, in order to yield a multidimensional subset of its points. Projections and slices allow the suppression of file system detail which is extraneous in a given situation. Thus, tasks such as the manipulation of the source files for one release of a software system in isolation from the source for all of the other releases, or the manipulation of all of the releases of a particular source file as a group, are greatly simplified.

For the space being considered, a slice across the version dimension, along the function coordinate 'parser', would be

```
                parser;0
                 /    \
           parser;1   parser;2
            /    \
    parser;1.1  parser;1.2      .
```

The projection of this slice onto the version axis would appear simply as

```
              0
            /   \
          1       2
        /   \
      1.1   1.2                .
```

A particular meaning, or "semantic", which a user intuitively associates with a certain kind of link between the files of an md-nt space, may be made known to the system. For the example being considered here, the "revision-of" semantic could be associated with the version axis by a command such as

```
C: attach rev-of to version
```

in order to inform the system that the kind of link which exists between

```
[ parser ; 1 ]   (call it 'P')
```

and

```
[ parser ; 1.1 ]   (call it 'CV')
```

means that "'CV' is a revision of 'P'".

As a result, certain kinds of actions, or "consequences", could be performed automatically by the system on files connected by such links. For example, any changes made to 'P' could automatically be applied by the system to 'CV' as well. As another example, for economy of storage space, the system might elect to store only the differences between the contents of 'P' and the contents of 'CV', rather than storing the entire contents of 'CV' as a separate file.

PROTOTYPE

A preliminary prototype for the structural aspects of the md-nt file space organization has been implemented as a user-mode layer above the UNIX file system. Each point of an md-nt file space is stored as a single UNIX file. All information concerning the structure of the space is stored in another UNIX file. The "structure file" and the "point files" are kept in a single UNIX directory.

A formalized extension of the UNIX command language is provided by an interpreter which has been implemented as a layer above the UNIX shell, using LEX and YACC [Johnson and Lesk 1978]. The language includes commands for manipulating the points of an md-nt space, and will include a powerful regular-expression sub-language for identifying the points of a space. Commands concerning the structure of the space are processed directly by the interpreter, and their actions are reflected in the underlying UNIX files. Ordinary UNIX commands, including those which deal with files of the multidimensional space, are simply passed on to the shell, after any point references are expanded and translated into UNIX file names.

Currently, the representation for the structure of the space, which is stored in the structure file, is a simple set of multi-linked nodes. Each point of the space is represented by a single node. Each node has a set of links which identify its parent, its first child, and its next sibling, in each dimension. Other representations being considered include: a simple hashed table of nodes, which contains no structural information, but which is augmented by a set of linked nodes representing the various axes; or, a representation which is based on an extended relational database.

In order to allow more effective experimentation with the semantic aspects of md-nt spaces, a more complete integration with the UNIX environment must be undertaken. This integration could be achieved by an interface implemented as a layer above the standard file access library. This would present md-nt spaces at a more fundamental level within the system.

CONCLUSION

Initial experience suggests that the md-nt file space organization can be a powerful and effective component of an integrated development project environment. This organization capitalizes on the considerable degree of orthogonality which is present in the file spaces of most large development projects. However, most such file spaces also include some dependencies. More work is required on handling these dependencies in a uniform fashion within the framework of md-nt spaces.

REFERENCES

[Boehm et al. 1982]
Boehm, B.W., Elwell, J.F., Pyster, A.B., Stuckle, E.D., and Williams, R.D. "The TRW Software Productivity System," Proc. 6th International Conf. on Software Engineering, September 1982.

[Cheatham 1981]
Cheatham, T.E. "An Overview of the Harvard Program Development System," in [Hunke 1981].

[Donzeau et al. 1980]
Donzeau-Gouge, V., Huet, G., Kahn, G., and Lang, B. "Programming environment based on structured editors: The Mentor Experience," INRIA Research Report No. 26, July 1980.

[Feldman 1979]
Feldman, S.I. "Make - A Program for Maintaining Computer Programs," Software Practice and Experience 9(3), March 1979.

[Ferch et al. 1978]
Ferch, H.J., Neufeld, G.W., and Zarnke, C.R. "MANTES User Manual," University of Manitoba, August 1982.

[Goldstein and Bobrow 1980]
Goldstein, I.P., and Bobrow, D.G. "A Layered Approach to Software Design," Xerox Palo Alto Research Center CSL-80-5, December 1980.

[Habermann and Notkin 1982]
Habermann, A.N., and Notkin, D.S. "The GANDALF Software Development Environment," Carnegie-Mellon University, January 1982.

[Hunke 1981]
Hunke, H., ed. "Software Engineering Environments," North Holland, 1981.

[Johnson and Lesk 1978]
Johnson, S.C., and Lesk, M.E. "UNIX Time-Sharing System: Language Development Tools," The Bell System Technical Journal 57(6), July 1978.

[McGuffin et al. 1979]
McGuffin, R.W., Elliston, A.E., Tranter, B.R., and Westmacott, P.N. "CADES - Software Engineering in Practice," Proc. 4th International Conf. on Software Engineering, 1979.

[Ritchie and Thompson 1974]
Ritchie, D.M., and Thompson, K. "The UNIX Time-Sharing System," Communications of the ACM 17(7), July 1974.

[Rochkind 1975]
Rochkind, M.J. "The Source Code Control System," IEEE Transactions on Software Engineering SE-1(4), December 1975.

[Thall 1982]
Thall, R.M. "The KAPSE for the Ada Language System," Proc. of the AdaTEC Conference on Ada, October 1982.

[Tichy 1982]
Tichy, W.F. "Design, Implementation, and Evaluation of a Revision Control System," Proc. 6th International Conf. on Software Engineering, September 1982.

APPENDIX: A SAMPLE FILE MANIPULATION SESSION

This appendix briefly illustrates a way in which the points of an md-nt space might be manipulated in a MANTES-like [Ferch et al. 1982] environment.

Note: '*' is used as a "wildcard" pattern rather than as a symbolic reference to the current file or record, ';' is used as a delimiter of coordinates in point identifiers rather than as a command separator, '--' is used to introduce comments, and ' . . . ' is used to mean "et cetera". Text entered by the user is in lowercase.

```
C: . . .
C: display_axis dim=module

    COMPILER
        SCANNER
        PARSER
            SYMMGR
            TREEMGR
        CODER

C: display_axis dim=version

    TOP
        1
        2
            2.1
            2.2
            2.3
        3

C: use projection=module,version
                  -- causes version within module for display, and module before
                  -- coordinate before version coordinate for file identifiers
C: display f=[compiler;top]

    COMPILER
            ; TOP
            ;     1
            ;     2
            ;         2.1
            ;         2.2
            ;         2.3
            ;     3
        SCANNER
                ; TOP
                ;     1
                ;     2
                ;         2.1
                ;         2.2
                ;         2.3
                ;     3
        PARSER
                ; TOP
                ;     1
                ;     2
                ;         2.1
                ;         2.2
                ;         2.3
                ;     3
            . . .
```

```
C: use projection=version,module
C: display top;compiler

    TOP
            ; COMPILER
            ;     SCANNER
            ;     PARSER
            ;         SYMMGR
            ;         TREEMGR
            ;     CODER
        1
                ; COMPILER
                ;     SCANNER
                ;     PARSER
                ;         SYMMGR
                ;         TREEMGR
                ;     CODER
        2
                ; . . .
            2.1
                    ; . . .
            2.2
                    ; . . .
            2.3
                    ; . . .
        3
                ; . . .

C: use projection=module
C: use f=[module:current; version:*]
                -- note: each coordinate of a file
                -- designator is defaulted or
                -- overridden separately
C: display compiler

    COMPILER
        SCANNER
        PARSER
            SYMMGR
            TREEMGR
        CODER

C: transfer symmgr under scanner
                -- moves the file for module symmgr
                -- under the file for module scanner
                -- (rather than the file for module
                -- parser) FOR EACH version
C: use f=version:3
                -- restrict operations to the slice
                -- with version coordinate '3'
                -- (rather than all versions)
C: create optimizer after coder
                -- just for version 3
                -- (see the display below)
C: use f=version:*
                -- back to dealing with all versions
                -- by default
C: create parser;version:2.1.1 under parser;version:2.1
```

```
C: use projection=module,version
C: display compiler;top
    COMPILER
            ; TOP
            ;     1
            ;     2
            ;         2.1
            ;         2.2
            ;         2.3
            ;     3
        SCANNER
                ; TOP
                ;     1
                ;     2
                ;         2.1
                ;         2.2
                ;         2.3
                ;     3
            SYMMGR
                    ; TOP
                    ;     1
                    ;     2
                    ;         2.1
                    ;         2.2
                    ;         2.3
                    ;     3
        PARSER
                ; TOP
                ;     1
                ;     2
                ;         2.1
                ;             2.1.1
                ;         2.2
                ;         2.3
                ;     3
            TREEMGR
                    ; TOP
                    ;     1
                    ;     2
                    ;         2.1
                    ;         2.2
                    ;         2.3
                    ;     3
        CODER
                ; TOP
                ;     1
                ;     2
                ;         2.1
                ;         2.2
                ;         2.3
                ;     3
        OPTIMIZER
                ;     3
```

```
C: use f=[module:symmgr; version:current]
C: use projection=version
C: display top
                 -- i.e. the versions of symmgr

    TOP
        1
        2
            2.1
            2.2
            2.3
        3

C: list version:3 first/2

    1.    *PROCESS;
    2.     SYMMGR: PROC OPTIONS(MAIN);

C: list version:1 first/2

    1.    *PROCESS;
    2.     SYMMGR: PROC OPTIONS(MAIN);

C: after version:1 first by=.01

    1.    *PROCESS;
    1.01  /* symmgr - symbol table manager package
    1.02   *
    1.03   * routines exported:
    1.04   *    addsym( . . .
    1.05   *    delsym( . . .
     . . .
    1.28   */

C: list version:1 first/4

    1.    *PROCESS;
    1.01  /* SYMMGR - SYMBOL TABLE MANAGER PACKAGE
    1.02   *
    1.03   * ROUTINES EXPORTED:

C: list version:3 first/4

    1.    *PROCESS;
    1.01  /* SYMMGR - SYMBOL TABLE MANAGER PACKAGE
    1.02   *
    1.03   * ROUTINES EXPORTED:
     -- (the changes which inserted 1.01:1.28
     -- were propagated to all subsequent
     -- versions as a result of the
     -- "revision-of" semantic being
     -- associated with axes of the
     -- version dimension)
```

```
C:  . . .
C:   -- now, assuming a somewhat larger space
C: display_axis dim=phase

    TOP
        SPECS
            DFD
        SRC
            DCLS
            BODY
        INTERM
            SYMTAB
            OBJCODE
        TARGET
            LOAD
        DOCN
            PGM_GUIDE
            USER_REF

C: display_axis dim=host

    TOP
        MOTOROLA
            8
                6800
                6801
                6809
            32
                68000
                68010
                68020
        INTEL
            8
                8080
                8085
            16
                8088
                8086
            32
                432
        ZILOG
            8
                Z80
            32
                Z8000

C: use projection=module,host,phase
C: display f=[module:symmgr;
              host:(motorola,zilog).8.*;
              phase:src.*] depth=0

    SYMMGR; MOTOROLA.8.6800; SRC.DCLS
                                 BODY
                       6801; SRC.DCLS
                                 BODY
                       6809; SRC.DCLS
                                 BODY
            ZILOG.8.Z80; SRC.DCLS
                             BODY
```

```
C: use f=[phase:src.*;host:68000]
C: use projection=module,phase
C: display f=[parser.*]

    PARSER
            ; SRC
            ;     DCLS
            ;     BODY
        TREEMGR
            ; SRC
            ;     DCLS
            ;     BODY

C: scan [parser;dcls] f:1 'fixed bin'

    . . .

C: off
```