TRENDS IN INFORMATION SYSTEMS

An Anthology of Papers from Conferences of the IFIP Technical Committee 8 'Information Systems' to Commemorate their Tenth Anniversary

edited by

B. LANGEFORS
Formerly at the University of Stockholm
Stockholm, Sweden

A. A. VERRIJN-STUART
University of Leiden
Leiden, The Netherlands

G. BRACCHI
Politecnico di Milano
Milan, Italy

1986

A Data Model for Programming Support Environments and its Application

Walter F. Tichy

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

A critical issue in programming support environments is the data base that stores all project information. This paper presents a model that can be used for analyzing and designing such data bases. The model represents systems as families consisting of multiple versions and configurations. It is based on AND/OR graphs and has the *hierarchical model*, the *relational model*, and the *sequential release model* as subclasses.

A refinement of the model yields the concept of the *well-formed configuration*. This concept establishes the basic rules for interface control and system composition. A generalization of the model leads to a data base structure that is a directed, attributed graph. This idea is illustrated by presenting design and implementation of a data base for the sequential release model.

1. Introduction

Programming support environments (PSE's) have recently been stressed as an approach to improve programmer productivity and software quality [1, 2, 3, 4, 5]. A PSE provides a rich set of sophisticated tools that support or automate various tasks during software development and maintenance. The tools operate on a common data structure, namely the data base that stores all information associated with a project. As in all software designs, the selection of an adequate data structure is crucial for a successful PSE. The design of that data structure is the subject of this paper.

An important observation for PSE data bases is that all large software products evolve into families of related versions and configurations. The existence of system families has long been acknowledged by Parnas and others [6, 7, 8, 9], yet all current programming language designs and most existing PSE's still ignore or skirt the issue. A few examples of system families are in order.

The most common situation when multiple versions arise is during program maintenance. In order to correct or enhance a software system, a subset of the modules must be modified. Normally this takes more than one iteration, resulting in several revisions per affected module. Some sequences of modifications turn out to lead into the wrong direction, making it necessary to back up to an earlier point. If one did not store the intermediate revisions, programmers have to "undo" the changes they made since the backup point, or "redo" some changes to regain the backup point from the initial revision. Both processes can be extremely difficult and time consuming.

 Reprinted from *Automated tools for information systems design*, H.-J. Schneider and A.I. Wasserman, eds., Proceedings of the WG8.1 Working Conference, Orleans, 26-28 January 1982, pp. 31-48.

Now suppose that the maintenance project is finished, resulting in a new system version. At this point one can usually discard the intermediate revisions that lead from the initial to the new revision. The initial revision, however, can often not be thrown away if a large user community depends on it. Thus, a system's administrator is forced to maintain "obsolete" versions [10].

The porting of programs to different environments is another cause for multiple versions. Compilers are typically ported to different environments, resulting in large families. Consider, for instance, Pascal [11] and C [12], which are available on a wide range of architectures. The same is now possible with some operating systems: Versions of UNIX [1] and Thoth [13] run on significantly different machines. The portability of programs is also a major goal of the Ada language and support efforts [3]. However, it is naive to assume that a program will execute correctly in every environment as long as it is written in a portable language. In reality, all kinds of minor and major changes are necessary, causing a single system to branch out into many parallel versions.

Enhancement and customization are additional, powerful forces that cause new versions to arise almost spontaneously. Users always apply a successful system in unexpected ways or unforeseen situations. Invariably, this requires improvements, bells and whistles to be added. Soon the system starts evolving away from its original characteristics, new errors creep in, and so the modification cycle goes on.

The naive approach to the problems of multiple versions is to eliminate them altogether. Unfortunately, this is not a viable approach. System families arise in response to widely differing demands. We shall never be able to write the all-encompassing compiler, operating system, telecommunications system, etc., that will adequately serve all user communities. On the other hand, the ad hoc approach of constructing a new, unique program for every user group is too costly. *We need to economize by building system families whose members share common parts.* In other words, we need to learn how to deal effectively with system families.

In Section 2 we present a model that has been designed specifically for multi-version programmed systems. This model leads to the concept of the well-formed configuration. Subclasses of the model are discussed in Section 3. Section 4 describes design and implementation of a small, multi-version PSE data base.

2. The AND/OR Graph Model for Families of Programmed Systems

Our model is based on AND/OR graphs [14]. An AND/OR graph is a directed, acyclic graph in which each node is either a leaf (without successors), an AND node, or an OR node. AND nodes and OR nodes must have at least one successor. When a node has a single successor, it can be viewed either as an OR node or an AND node.

Leaf nodes

The leave nodes are primitive objects in our model and represent program modules, intermediate code, documentation fragments, test data, etc.

OR nodes

OR nodes represent *version groups*. Successors of an OR node are considered equivalent according to some criterion. Thus, an OR node implies a choice -- one may choose one (or several) of its successors.

AND nodes

AND nodes represent *configurations*. All successors of an AND node must be combined to form a complete configuration. Thus, an AND node implies an integration process; this corresponds to a link-editing process for pure software configurations, a loading process for hardware/software configurations, and an assembly process for pure hardware configurations.

As an example, suppose we have a system *S* with three configurations *C1*, *C2*, and *E*. Suppose furthermore that configuration *C1* consists of components *A* and *B*, configuration *C2* of components *C* and *D*, and configuration *E* is primitive (i.e., a single component). This situation can be diagrammed in the following way (see Figure 1).

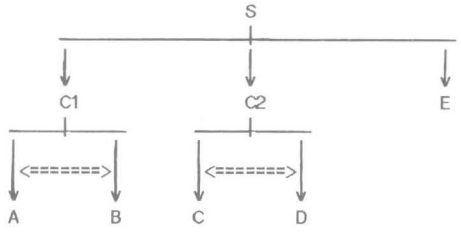

Fig. 1: An AND/OR graph with one OR node and two AND nodes.

The node with the label *S* is an OR node since it allows a choice of three alternatives. The nodes *C1* and *C2* are AND nodes, since their successors need to be combined. In the diagram, the AND nodes are marked with the symbol "<====>" linking their successors.

Note that we are dealing with a directed graph, not merely a tree or a forest. In a directed graph, a single node may have several predecessors. This permits the modeling of *component sharing*. For example, a situation like in Figure 2 is impossible to realize in a tree (except by copying whole subtrees).

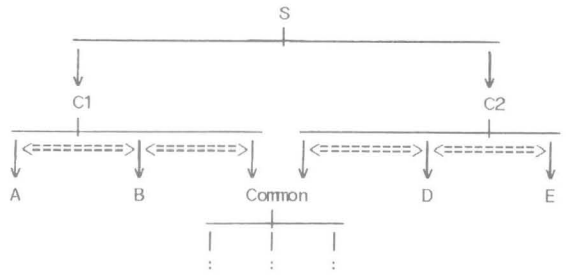

Fig. 2: Configurations C1 and C2 share node Common.

AND and OR nodes may be intermixed freely. Thus, one can form version groups out of primitive nodes, configurations, and even other version groups. Likewise, configurations may consist of primitive nodes, configurations and version groups. This reflects the *orthogonality* of the concepts of version group and configuration.

We shall now demonstrate with a few more examples how AND/OR graphs can be used to represent various types of hardware/software systems, including their documentation and test data. This will be accomplished by attaching special significance to the branches emanating from AND nodes and OR nodes.

In our model, a program module cannot be subdivided. However, each module normally evolves in a sequence of revisions that are incremental changes to some initial version. The revisions are ordered by their creation date. This situation is diagrammed with an OR node (see Figure 3).

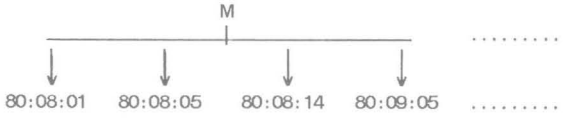

Fig. 3: Revisions of a software module.

Suppose furthermore that our compiler is capable of generating code for the PDP-11, the VAX-11, and the Intel 8086 from any of the revisions. Assume that in each case the compiler may generate optimized or non-optimized code. This is represented with two more levels of OR nodes (see Figure 4).

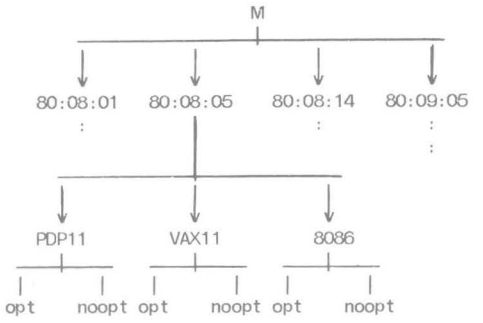

Fig. 4: Revisions, target versions, and optimized versions.

Now suppose that we would like to add documentation to our module, for example a general description and some implementation decisions. That is quite easily done by adding yet another OR node, this time on top. Note that the documentation may go through several revisions, just like source code. It may even be compiled for several output devices, for example for the terminal, the line printer, the photo typesetter, etc. Thus, the structure for documentation is similar to the one for implementation (see Figure 5).

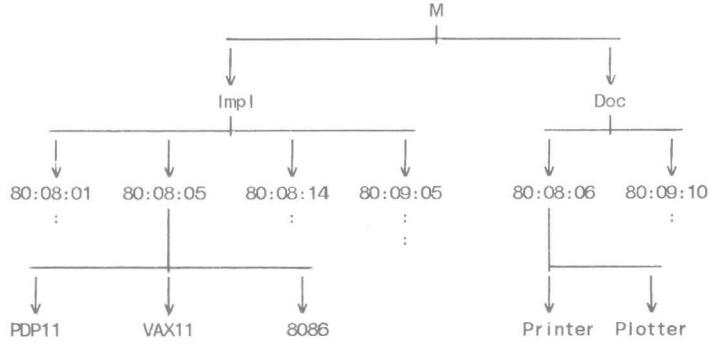

Fig. 5: M has two alternatives, implementation and documentation.

Note that OR nodes with documentation and implementation branches are different from the OR nodes we have seen so far. Up to now, OR nodes only combined equivalent implementations. Documentation and implementation are also equivalent, but in a different sense: they describe the same object, one giving the specification, the other the implementation. Recall that we

defined OR nodes as representing an inclusive-or relation. Thus, we can even model the view that documentation and implementation form an entity.

Clearly, an OR branch for documentation can be added wherever desirable. For example, one may add documentation to the revisions in the form of a "change log." One can also associate documentation with higher-level nodes to supply a general overview, a user's manual, or the requirements specification.

The AND/OR graph can be applied to hardware as well. However, the decomposition into subgraphs may have a somewhat different shape. For example, there may be components that have no revisions, like bolts or other standard parts. There may be additional document types, like circuit schematics or instructions for the assembly of certain configurations. Hybrid configurations consisting of both hardware and software are best represented with AND nodes. For instance, if a particular program is to be stored in a specific PROM, then both components should be successors of the same AND node. A combination of hardware and software is permissible anywhere in the graph. For instance, we may want to indicate that a certain operating system can run on several machine models, or that some software components have to be distributed over specific nodes in a network.

All these different interpretations are actually overloading our simple AND/OR graph model. The three basic node types are no longer sufficient. For building intelligent software tools, we need additional node types. The types indicate the semantics associated with a given node. Software tools can then take advantage of that information. We shall come back to this idea in Section 4.

2.1. Generic Configurations and the Selection Problem

A single AND node may actually represent a number of possible configurations if some of its successors are OR nodes. Such an AND node represents a *generic configuration* and is therefore called a *generic AND node*. Generic AND nodes are important for avoiding the combinatorial explosion of the number of AND nodes.

Consider Figure 6, which describes the I/O subsystem of some larger family. It has two major versions, one for the line printer (*LPT*), and one for the terminal (*Terminal*). The *LPT* version is a configuration consisting of three components: *open*, *close*, and *put*. The modules *open* and *close* exist as a sequence of revisions, labeled with release numbers. The node *put* has two machine specific versions, one for the *VAX* and one for the *PDP11*. Each of those has again several revisions.

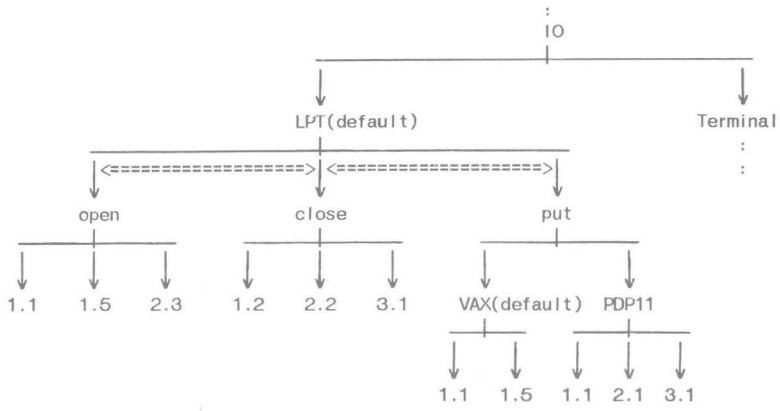

Fig. 6: Several versions of an I/O subsystem.

This diagram compactly represents 3*3*(2+3)=45 configurations. Without the generic node *LPT*, we would need 45 structurally identical AND nodes. If we add just one more revision to module *open*, the number of configurations increases to 60, and we would need 15 more AND nodes. By contrast, with generic nodes we have to add only a single offspring to module *open*. This example should demonstrate that the lack of generic configurations may lead to serious bookkeeping problems as the number of modules and revisions increases[1].

In large families, there are easily thousands of configurations that can be created by arbitrary selection of offsprings at OR nodes. Relatively few of them will actually work together. The problem is how to select the proper ones. One possibility is to use "cutoff" release numbers, "cutoff" dates, and defaults. A cutoff release number (date) selects at each node the revision with the number (date) that is less than or equal, but closest to the cutoff. In the above example, *IO:2.3* would select the configuration {*open:2.3*, *close:2.2*, *put:VAX:1.5*}. This is consistent with the practice of defining releases to be the newest revisions of all components at a given date. Note also the application of two user-specified defaults (*LPT* and *VAX*). The default for release numbers and dates should correspond to the newest one for each component. We also need a mechanism to specify "symbolic" release numbers like *current*, *experimental*, *stable*, etc. Of course, it must be possible to override the defaults to express something like: "I want the default for everything, except that I need the *Terminal*-version of *IO*." This is specified with *IO.Terminal*. Notations for cascading those selections are easily included.

[1] The *cardinality* of a node, i.e., the number of versions represented by it, is computed as follows. (1) The cardinality of a leaf node is 1; (2) the cardinality of an AND node is the product of the cardinalities of its offsprings; (3) the cardinality of an OR node is the sum of the cardinalities of its offsprings (assuming that exactly one alternative must be chosen).

An additional selection mechanism involves the labeling of OR branches. The labels serve as criteria for global selection. For example, suppose that some of the branches emanating from OR nodes are labeled *basic*, *intermediate*, or *advanced*, indicating the obvious qualities about the three choices. Assume that these labels are spread through a large graph. Then one can select a desired configuration by simply requesting, for instance, the *basic* branch wherever there is a choice. (Note that this is similar to the global selection by cutoff date.) This technique is also convenient for specifying the target machine or optimized/unoptimized versions. An example is *IO:PDP11:nonopt*.

2.2. Well-Formed Configurations

An extremely important issue in multi-person projects is interface control: to establish and maintain consistent interfaces between the numerous components. The concept of the *well-formed configuration*, defined in this section, forms the basis for interface control. Our concept is a generalization of the conditions on system structure presented in [15] and [16].

We start by associating an interface with every node in our graph. An interface consists of two sets: the provided facilities and the required facilities. The provided facilities are the data types, operations, data structures, etc. exported from a node. An example of provided facilities are the visible interfaces of Ada packages [17]. The required facilities are the types, operations, data structures, etc. that must be imported into the node.

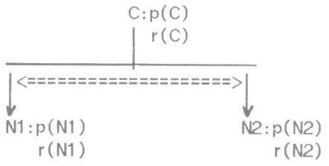

Fig. 8: A configuration with two components; interfaces attached.

The set of facilities provided by a node N is denoted as $p(N)$, the set of required facilities as $r(N)$[2]. We have to make sure that a facility mentioned in the provided set of a node does not also occur in the required set. This leads to the following definition.

A node N is *free of contradictions* if and only if

$$p(N) \cap r(N) = \phi$$

[2] We could link the interfaces into our diagram with some extra OR nodes, but it is more convenient to think of them as node attributes. Compare Section 4.

We define node M to be upward compatible with node N if M provides at least what N provides, and requires not more than what N does. That means that M can be used instead of N, but not vice versa.

Node M is *upward compatible* with node N if and only if

$$p(M) \supseteq p(N) \text{ and } r(M) \subseteq r(N)$$

Similarly, two nodes are compatible if they have the same interface. Thus, compatible nodes are interchangeable.

The nodes M and N are *compatible* if and only if

$$p(M) = p(N) \text{ and } r(M) = r(N)$$

The last two definitions apply to arbitrary pairs of nodes. They will be especially interesting for OR nodes.

We can now define *well-formed nodes*. There are different, recursive definitions for each node class (leaf, AND, and OR nodes).

A. A leaf node is well-formed if and only if it is free of contradictions.

(Since a leaf node usually corresponds to a given source module, we have to make sure that the source actually satisfies the interface. Techniques for implementing that have been presented in [18].)

B. An OR node R with direct successors $K_1, \ldots, K_n$ $(n \geq 1)$ is well-formed if and only if

i. R is free of contradictions;

ii. There exists at least one direct successor $K_i (1 \leq i \leq n)$ of R which is well-formed and upward compatible with R.

(Since only one K_i needs to satisfy condition ii, we can add documentation to OR nodes without problem, or make configurations versions of each other although they have different interfaces.)

C. An AND node S with direct successors $K_1, \ldots, K_n$ $(n \geq 1)$ is well-formed if and only if

i. S is free of contradictions;

ii. All K_i $(1 \leq i \leq n)$ are well-formed;

iii. $p(K_i) \cap p(K_j) = \phi$ if $i \neq j$ (freeness of conflicts)

iv. $p(S) \subseteq \bigcup_{i=1}^{n} p(K_i)$

v. $r(S) \supseteq (\bigcup_{i=1}^{n} r(K_i) - \bigcup_{i=1}^{n} p(K_i))$

Since configurations correspond to AND nodes, we say that a configuration is well-formed if its AND node is well-formed. The basic conditions given above are precisely those which must be checked when a configuration is built from a set of components. The conditions can also be used

to construct the interfaces of newly created AND nodes and OR nodes if a system designer is composing new system versions interactively. They are applied in search algorithms that compose well-formed configurations automatically, as discussed by [16]. The interfaces can also be used to assess proposed interface changes by analyzing the effects for each node. Finally, interface changes can be carried out by propagating the modifications to all affected nodes, as described in [18]. The required algorithms and their complexities are currently being explored.

3. Comparison of other Models for Representing System Families

In this section, we analyze the data models underlying some existing software tools. The comparison concentrates on what kind of AND/OR graph structures the tools permit. We shall see that most of them place severe restrictions on the shape of the graph. We distinguish the following 4 major submodels. (Example implementations or proposals are noted in parenthesis.) More detail can be found in [19].

a) The Hierarchical Model (Ada [17], Simula67 [20], Mesa [21]),

b) The Relational Model ([22] and [23]),

c) The Sequential Release Model (SCCS/MAKE [24,25]).

d) The AND/OR graph model ([9,15,16]).

The hierarchical model imposes a partial ordering on the program modules, and multiple versions are not permitted. The result is an AND/OR graph without any OR nodes. In the relational model, configurations are specified as lists of components in rows of a single, large matrix or several, cascaded matrices. Again, no multiple versions are permitted. Because of the lack of OR nodes, both the hierarchical and relational models are essentially equivalent. Versions cannot be specified, which makes it impossible to build tools within these models that work on version groups rather than individual components. Generic configurations are also lacking, which leads to the combinatorial explosion of the number of configurations.

The sequential release model allows program modules to exist as a sequence of revisions. This leads to a graph where OR nodes are permitted only as predecessors of leaf nodes. Configurations that are structurally identical and whose modules differ only in the revision numbers can be represented with a single, generic configuration. However, it is not possible to indicate that two different configurations are actually versions of each other, no matter how slight the differences. This is due to the fact that the sequential release model permits no internal OR nodes.

The general AND/OR graph model has none of these restrictions. Any two configurations can be made versions of each other, and a single, generic description suffices for structurally identical configurations. Structurally similar configurations can be described without duplication of information. Hardware configurations, documentation, test configurations, and test data can be added without problem. (None of the examples listed under point d permits an AND/OR graph in its full generality.)

4. Application of the Model

We noted previously that three node classes (leaves, AND nodes, and OR nodes) are not sufficient for building an intelligent PSE. For example, software tools need to treat revisions of source modules differently from object code or configuration versions. Yet in our basic model, these are all offsprings of OR nodes. We suggested already that types associated with nodes would alleviate the problem, because then the tools can be programmed to treat each type properly. Besides the types, we also need to attach various attributes to the nodes, for recording creation dates, release numbers, selection labels, access lists, interfaces, etc.

A refinement of the AND/OR graph model that permits this information to be represented is the *directed, attributed graph*. Every node in an attributed graph has a type and a set of attributes. The type determines the attribute set associated with a node. In this section, we use attributed graphs to define a PSE data base for the *Revision Control System (RCS)*. RCS is a variant of the sequential release model.

4.1. Design of the RCS Data Base

As a first step, we specify the general structure of the attributed graph. We could use the data declaration facility of a general purpose programming language for that. However, this approach would dictate a large number of representational details which are either wrong or should not be fixed at this point. The same is true for the CODASYL data definition language. The latter also forces some awkward constructions for certain kinds of graphs. Instead, we use IDL (Interface Description Language) [26]. IDL is a language for declaring attributed graphs as abstract data types. It has been used for defining Diana, the intermediate form of Ada programs [27]. IDL satisfies three important requirements. First, IDL is programming language independent. Thus, IDL graphs may be manipulated by tools written in diverse languages. Second, IDL does not prescribe any particular realization -- the graphs are merely conceptual ones. Indeed, we chose a rather uncommon way of implementing IDL graphs, to be discussed in Section 4.2. Third, IDL defines a standard, externally visible ASCII representation for graphs. In this form, the graph can be read by the user, and communicated between arbitrary tools and even arbitrary PSEs on different computer systems. All that is needed are encoders and decoders for porting the contents of a PSE data base from one implementation to the next.

Below is the IDL specification of RCS. The reader need not be familiar with the IDL notation; those aspects of IDL that are used here are informally described as we go along.

```
mode Revision_Constrol_System root RCSnode is

    RCSnode      ::=  Delta | Module | Config;
    -- There are 3 basic node types.

    Delta        =>   RevisionNo      : string,
                      Date            : string,
                      Author          : string,
                      LogEntry        : string,
                      State           : string,
                      Text            : string,
                      Next            : Delta,
                      Branches        : seq of Delta;

    -- This specifies the attributes of nodes of type Delta.

    Module            AccessSet       : set of string,
                      Language        : string,
                      Locks           : set of Lock,
                      Head            : Delta;

    Lock         =>   RevisionNo      : string,
                      Locker          : string;

    -- A lock node indicates which branch is locked for expansions.

    Config       =>   AccessSet       : set of string,
                      ReleaseNo       : string,
                      State           : string,
                      Components      : set of RCSnode;

end
```

Fig. 9: IDL specification for the revision control system.

The first line of the specification indicates that a new data type with the name Revision_Control_System is defined. The root clause gives the starting symbol of the specification. There are 3 basic node types in the graph: *Delta*, *Module*, and *Config*. The delta nodes represent the revisions of a module and are organized in a tree with the initial revision as the root. The tree has a main branch, called the *trunk*, along which the main development occurs. The field *Next* links deltas on the same branch of the tree. A delta may sprout one or more parallel branches. The entries in the field *Branches* point to the first delta on each branch. Figure 10 illustrates an example tree with 3 branches (not counting the trunk). Deltas on the trunk are numbered 1.1, 1.2, ..., 2.1, 2.2, etc. Other branches are numbered *fork*.1, *fork*.2, ... etc, where *fork* is the number of the delta that sprouts the branch. Deltas on a branch are again numbered sequentially, using the branch number as a prefix.

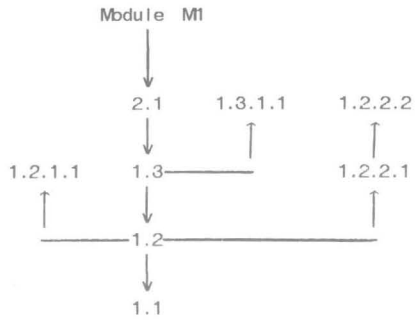

Fig. 10: A revision tree with 3 side branches.

Note that the links in the trunk point backwards from the latest delta rather than forwards from the root. This is an optimization to speed up access time. The latest delta on the main branch is the one that is most often used. We therefore store this version intact. All others are stored as a set of differences that will restore the revision given the previous one. To obtain, for instance, revision 1.3, the differences stored in delta 1.3 are applied to revision 2.1. The older the revision is, the more deltas need to be applied. This technique is called reverse deltas. Deltas conserve space, and reverse deltas minimize the average time needed to restore a revision[3].

Unfortunately, reverse deltas do not work so well for side branches. To avoid keeping a complete copy of the newest revision on each branch, we use forward deltas from the branch point. Thus, applying to delta 2.1 first delta 1.3 and then 1.3.1.1 will regenerate revision 1.3.1.1. This is still shorter than regenerating 1.3.1.1 from the root 1.1.

Let us now examine the definition of the *Delta* node more carefully. The text following the symbol "=>" is a record definition. Each field or attribute declaration is composed of an attribute name and a type. *RevisionNo* numbers the deltas as discussed above. The *Date* field records the creation date and time, and the *Author* field stores the identification of the person who created the new revision. The *LogEntry* attribute contains a short note describing the nature of the change that made the revision necessary. The author of the revision is prompted by the data base system to supply the log entry when the revision is deposited. *State* indicates the status of a revision, for example whether it is experimental, stable, or released. Finally, the attribute *Text* contains the actual program text.

The *Module* node contains attributes that are common to all deltas. The *AccessSet* attribute is a set of user names that have write-permission, i.e., those who may create deltas. (Read permission is given to all other users.) The *Language* field records the programming language or

[3] The use of reverse, non-intermixed deltas is one of the chief differences between RCS and SCCS.

document formatting language used. This is needed for automatic system or document generation. The locks make sure that there are no overlapping changes on a branch while somebody is preparing a new revision for it. A *Lock* node records the branch number and locker. The field *Head* points to the most recent revision on the main branch.

Nodes of type *Config* record configurations. The offsprings of this AND node are recorded in the attribute *Components*. Note that this is a set of *RCSnodes*. Thus, members of this set may be modules, configurations, and even deltas. The attribute *AccessSet* defines who may change the node. There are additional fields to record the release number and the state of the configuration. Figure 11 presents an example instantiation of RCS.

4.2. Implementation of RCS

A prototype of RCS has been successfully implemented on a VAX/UNIX system. It uses reverse deltas, but without branches. The purpose of the prototype was to investigate the feasibility of a PSE data base patterned after an attributed graph. A more ambitious project providing a full attributed graph structure (including reverse deltas and branches) is under way. Another project develops ADABASE, a PSE data base specifically designed for the Ada programming language [28]. We wish to report here on the important implementation decisions and the data base operations of the prototype.

An important problem is the representation of the attributed graph. The naive approach would be to place an encoding of the graph into a single file. Since the file is the unit of change, only one programmer at a time can modify the graph. This leads to unacceptable delays since there are usually several people modifying the data base simultaneously. For example, there may be several programmers checking modules in and out for modification, adding documentation and object modules, constructing test configurations, and accessing the data base for interfaces and various other data items.

The opposite approach of placing every node into a separate file may lead to another kind of inefficiency caused by frequent directory lookups. We therefore adopted the approach that one or several nodes may be stored in a single file. Pointers to nodes in the same file can be traversed quickly; pointers to "remote" nodes require a directory lookup or even a network transfer. In that manner one can optimize the data base bandwidth by rearranging the assignment of node groups to files (possibly spread over a computer network). In RCS, one should place each *Module* node together with all its descendants into a single file because these nodes are usually accessed together.

The encoding of the graph is the standard ASCII representation defined in the IDL manual. This simplifies the operations for browsing through the graph. Graphics support for pictorial

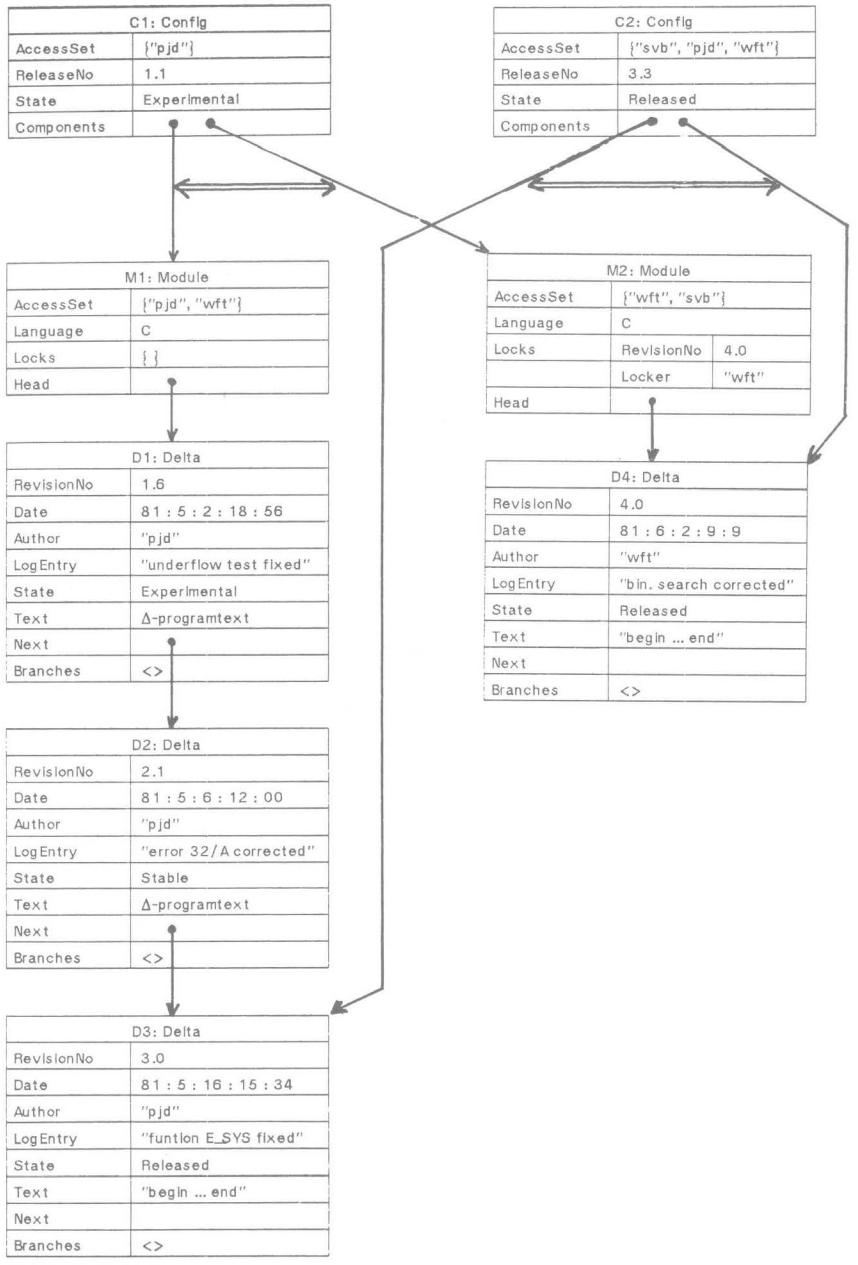

Fig. 11: Example data base for RCS.

rather than textual display is under investigation. The high-level operations for handling revisions are patterned after SCCS [24]. A synopsis of the commands is given below.

Rcs

Rcs modifies or initializes a module or configuration node. There are parameters to expand or shrink the access set, and to specify the other attributes. Special options lock, unlock, or break the lock of a branch. *Rcs* may only be executed by users on the access set.

Co

Co checks out a revision of a module node for update or inspection. If an update is desired, *Co* first locks the corresponding branch (if not already locked). This avoids that two people create overlapping updates to the same branch. (A lock can be released by performing an update with the *Ci* command, or by releasing it, without update, using *Rcs*.)

Co places the retrieved revision into a file in the user's directory for editing or inspection, or sends it to the terminal for perusal. The desired revision may be selected by revision number, symbolic name, creation date, state, or author. The default is the most recent revision.

Ci

Ci appends a new revision to a branch. Only the user who locked the branch in his name may execute it. Normally, the new revision number is obtained by incrementing the number of the latest revision, but the user can specify a higher number explicitly. After successful completion, *Ci* releases the lock.

Rlog

Rlog displays the log messages and other information about module and delta nodes in a variety of formats.

Make

Make compiles a configuration. Normally, it selects the latest revisions of all modules composing it, but the notation suggested in Section 2 can be used to specify other selections by state, symbolic name, cutoff date, or cutoff release number. It is also possible to select specific versions individually. *Make* uses the *Language* field to determine which processor (e.g., compiler or document formatter) to call. It does not attempt to avoid redundant compilations and linkings.

Make could be extended to save redundant compilations and linkings if our graph structure included derived versions. In that case we would need additional offsprings at *Delta* and *Config* nodes for recording object modules and the derivation history. We have omitted these details for clarity. It should also be noted that it is easy to add an OR node for representing versions of configurations.

5. Conclusions

We have introduced a simple and flexible model for representing families of programmed systems. The model allows the sharing of components among configurations, treats configurations and versions completely orthogonally, provides generic configurations, and yields the concept of the well-formed configuration. The model can also be used to compare the data base structures underlying other software tools.

A refinement of the model leads to a directed, attributed graph with several node types. We designed the graph structure for the revision control system and presented a data base implementing that structure. The design and implementation demonstrate that the directed, attributed graph is adequate for developing data bases for programming support environments.

Acknowledgments: Part of this work was done at the ITT Programming Technology Center in Stratford, Conn., and I am especially grateful for comments from Donn Combelic and Tom Love.

References

1. Kernighan, Brian W. and Mashey, John R., "The UNIX Programming Environment," *Software -- Practice and Experience* **9**(1) pp. 1-15 (Jan. 1979).
2. Habermann, A. Nico, "An Overview of the Gandalf Project," in *CMU Computer Science Research Review 1978-1979*, Carnegie-Mellon University (1979).
3. Buxton, John N. and Druffel, Larry E., "Requirements for an Ada Programming Support Environment: Rationale for Stoneman," pp. 66-72 in *Proceedings of COMPSAC 80*, IEEE Computer Society Press (Oct. 1980).
4. Osterweil, Leon J., "Software Environment Research: Directions for the Next Five Years," *IEEE Computer* **14**(4) pp. 35-43 (April 1981).
5. Teitelman, Warren and Masinter, Larry, "The Interlisp Programming Environment," *IEEE Computer* **14**(4) pp. 25-33 (April 1981).
6. Parnas, David L., "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering* **SE-2**(1) pp. 1-8 (Mar. 1976).
7. Parnas, David L., "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering* **SE-5**(2) pp. 128-138 (March 1979).
8. Habermann, A. Nico, Flon, Lawrence, and Cooprider, Lee W., "Modularization and Hierarchy in a Family of Operating Systems," *Communications of the ACM* **19**(5) pp. 266-272 (May 1976).
9. Cooprider, Lee W., *The Representation of Families of Programmed Systems*, PhD thesis, Carnegie-Mellon University, Department of Computer Science (1978).

10. Belady, L.A. and Lehman, M.M., "The Characteristics of Large Systems," pp. 106-138 in *Research Directions in Software Technology*, ed. Peter Wegner,M.I.T. Press (1979).
11. Wirth, Niklaus, "The Programming Language Pascal," *Acta Informatica* **1** pp. 35-63 (1971).
12. Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, Prentice-Hall (1978).
13. Cheriton, David R., Malcom, Michael A., Melen, Lawrence S., and Sager, Garry R., "Thoth, a Portable Real-Time Operating System," *Communications of the ACM* **22**(2) pp. 105-115 (Feb. 1979).
14. Nilsson, Nils J., *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill (1971).
15. Tichy, Walter F., *Software Development Control Based on System Structure Description*, PhD Thesis, Carnegie-Mellon University, Department of Computer Science (Jan. 1980).
16. Habermann, A. Nico and Perry, Dewayne E., "Well-Formed System Compositions," CMU-CS-80-117, Technical Report, Carnegie-Mellon University, Department of Computer Science (March 1980).
17. Ichbiah, Jean D., *Reference Manual for the Ada Programming Language*, United States Department of Defense (July 1980).
18. Tichy, Walter F., "Software Development Control Based on Module Interconnection," pp. 29-41 in *Proceedings of the 4th International Conference on Software Engineering*, ACM, IEEE, ERO, GI (Sept. 1979).
19. Tichy, Walter F., *A Model for Representing Families of Programmed Systems*, Technical Report, Purdue University, Computer Science Department (January 1981).
20. Birtwistle, G., Enderin, L., Ohlin, M., and Palme, J., "DECsystem-10 Simula Language Handbook Part 1," C8398, Swedish National Defense Research Institute (March 1976).
21. Mitchell, James G., Maybury, William, and Sweet, Richard, *Mesa Language Manual*, Technical Report, Xerox Palo Alto Research Center (Feb. 1978).
22. Belady, L.A. and Merlin, P.M., "Evolving Parts and Relations: A Model for System Families," RC-6677, Technical Report, IBM Thomas J. Watson Research Center (1977).
23. ITT,, *CMSS3 Users's Manual*, International Telephone and Telegraph (1980). Document No. 211ITT26366-PC
24. Rochkind, Marc J., "The Source Code Control System," *IEEE Transactions on Software Engineering* **SE-1**(4) pp. 364-370 (Dec. 1975).
25. Feldman, Stuart I., "Make - A Program for Maintaining Computer Programs," *Software -- Practice and Experience* **9**(3) pp. 255-265 (March 1979).
26. Nestor, John R., Wulf, William A., and Lamb, David A., *IDL - Interface Description Language, Formal Description*, Technical Report, Carnegie-Mellon University, Computer Science Department (Feb. 1981).
27. Goos, Gerhard and Wulf, William A., *Diana Reference Manual*, Technical Report, Carnegie-Mellon University, Computer Science Department (March 1981).
28. Tichy, Walter F., *ADABASE -- A Data Base for Ada Programs*, Technical Report, Purdue University, Computer Science Department, in preparation (November 1981).

Area 4: FORMAL METHODS

G. RICHTER, Clocks and their Use for Time Modeling

"Information systems: Theoretical and formal aspects", A. Sernadas, J. Bubenko, jr. and A. Olivé, eds., Proceedings of the WG8.1 Working Conference, Sitges, 16-18 April 1985, pp. 49-66.

P.A.S. VELOSO and A.L. FURTADO, Towards Simpler and Yet Complete Formal Specifications

"Information systems: Theoretical and formal aspects", A. Sernadas, J. Bubenko, jr. and A. Olivé, eds., Proceedings of the WG8.1 Working Conference, Sitges, 16-18 April 1985, pp. 175-189.