1981

A Model for Representing Families of Programmed System

Walter F. Richy

Report Number:
80-356

A Model for Representing Families of Programmed System

by

Walter F. Tichy

Computer Science Department

Purdue University

West Lafayette, Indiana 47907

CSD-TR 356

A Model for Representing Families of Programmed Systems

Walter F. Tichy

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

ABSTRACT

A general model for multi-version, multi-configuration programmed systems is presented. It is used to evaluate the data representations underlying various programming support environments. The model is based on AND/OR graphs and allows the compact description of system families. The *hierarchical model*, the *relational model*, and the *sequential release model* are defined as subclasses of the general model. It is shown that the hierarchical and relational representations are essentially the same.

The concept of the *well-formed configuration* is developed as a refinement of the general model. This concept yields the basic rules for interface control and configuration composition. The model is also employed to state basic requirements for the data bases on which programming support environments should be built.

January 8, 1981

A Model for Representing Families of Programmed Systems

Walter F. Tichy

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

1. Introduction

Programming support environments (PSE's) have recently been stressed as a way to significantly improve software production methods and software quality [Ker79a, Hab79a, Bux80a, Rid80a]. A PSE provides a rich set of sophisticated tools that support or automate various tasks during software development and maintenance. An important feature of a successful PSE is that it is well coordinated with a high-level programming language, so that the PSE and the language complement and enhance each other.

Unfortunately, research in PSE's has not kept pace with advances in programming languages. This is vividly demonstrated by comparing the following two recent documents: The requirements for the Ada programming language [DoD77a] are specific and precise, whereas the requirements for the Ada programming support environment [Bux80b] are quite the opposite. There is an impressive body of knowledge about programming language concepts, documented in numerous books, journals, and conference proceedings (see, for example [Els73a]). In contrast, there is not nearly as much quality material published on software development environments (Unix [Ker79a] being a notable exception). We do not want to argue here why this is so, but the conclusion should be obvious: If we want to tackle the software crisis by means of programming support environments, we are in desperate need of useful concepts related to them. We need concepts for modeling, classification, and evaluation, and we need to come up with new ideas and prototypes. This paper makes a start in that direction by presenting a model which can be used to compare the data representations underlying a wide range of programming support tools and

environments, as well as providing a suitable conceptual basis for further research and development. The model is not limited to software, but can be applied to hardware, firmware, documentation, and test configurations as well.

Before we discuss our model in detail, let us make two observations concerning programming environments. First, a common way of introducing a particular environment is to explain how to write and run a program or how to prepare a document. Thus, the environment is described by the functions it performs. The data structures involved are hardly mentioned, because they are usually simple text files. We take a different approach. We shall first discuss the data structures for representing programmed systems, and then the operations for manipulating them. The reason is that the data structures determine to a large degree what *kinds* of systems can be handled at all. We shall see that most existing PSE's severely constrain the structure of the systems they can handle effectively.

The second observation concerns the phenomenon that all software systems evolve into families of related versions and configurations, a fact widely ignored by existing support environments. A few examples of system families are in order. Compilers are typically ported to different environments, resulting in large families. Consider, for instance, Pascal [Wir71a] and C [Ker78a], which are available on a wide range of architectures. The same is beginning to happen to some operating systems: Versions of UNIX [Ker79a] and Thoth [Che79a] run on significantly different machines. The portability of programs and, in particular, of software tools is also a major goal of the Ada language and support efforts [Bux80b]. 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.

The porting of programs to different environments is not the only cause for multiple versions. Repair, enhancement, and customization are additional ones [Bel79a]. Of these, repair seems to be the easiest one to deal with: If a particular version is corrected, one can safely throw away the old one. Unfortunately, this is not so if a large user community has come to depend on the old version,

forcing a system's administrator to maintain "obsolete" versions.

Enhancement and customization are powerful forces that cause new versions to arise almost spontaneously. Users always seem to apply a given system in unexpected ways or unforeseen situations, adding improvements, bells and whistles, and frequently changing the characteristics of a system. And so the modification cycle goes on.

An obvious 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, weapons system, etc., that will adequately serve the whole user community [Par79a]. 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.

The existence of system families has long been acknowledged by Parnas and others [Par76a, Par79a, Hab76a, Coo78a, Lov80a]. Unfortunately, all current programming language designs and most existing PSE's still ignore or skirt the issue of multiple versions. This will become more obvious in Section 3.

2. A General Model for Representing Families of Programmed Systems

At the heart of any programming support environment must be some model of the structure of the system that is to be developed and maintained. The model determines to a large degree what kinds of tools can be applied to what kinds of systems. We present a general model for multi-version, multi-configuration system families, which has the following properties:

a) Multiple versions: Configurations as well as primitive components may be represented in any number of versions.

b) Sharing: Configurations as well as primitive components may be shared among other configurations without restriction. This applies to individual versions as well as groups of versions.

c) Completeness: All versions of all configurations and primitive components may be represented in a compact form.

d) Generality: Hardware, firmware, software, documentation, test data, etc., and versions thereof, can all be combined into configurations. (This is a consequence of c).)

e) Well-Formed Configuration: As a refinement, the concept of a well-formed configuration is defined with respect to the interfaces associated with the components.

The model can also be used to evaluate the representations underlying existing programming support environments (see Section 3).

2.1. Introduction to the Model

Our model is based on AND/OR graphs. AND/OR graphs are applied in problem solving methods in Artificial Intelligence [Nil71a] and for analyzing serial-action work in industry [Rig69a]. However, AND/OR graphs will be used quite differently here.

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 Fig. 1).

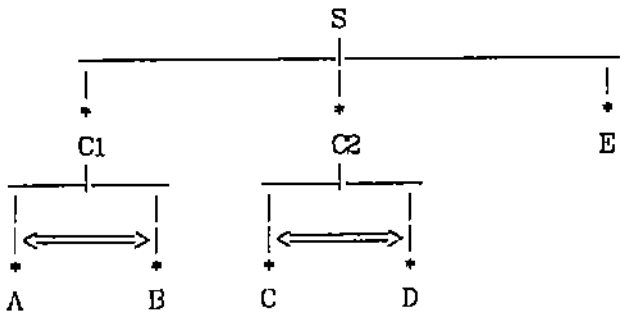

Fig. 1: An AND/OR graph with one OR node and two AND nodes.

The node with the label *S* is called an OR node since it allows a choice of three alternatives. The nodes *C1* and *C2* are called AND nodes, since their successors need to be combined to form a complete configuration. In the diagram, the AND nodes are marked with the symbol "<=====>" linking all their successors. Obviously, an AND node implies an integration process; this corresponds to a link-editing process for pure software configurations, an assembly process for pure hardware configurations, and a loading process for hardware/software configurations. An OR node represents a group of versions -- one may choose one (or several) of its successor nodes.

Formally, AND/OR graphs are directed, labeled, acyclic graphs 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.)

We note two important special cases. First, if a graph has no OR nodes with at least two successors, then we have a single system (no alternatives possible). This may be diagrammed as a single AND node with only primitive components as successors, or as several, cascaded AND nodes. Second, if every node in our graph has at most one predecessor (i.e., our graph is a tree or a forest), then the various configurations do not share common components. Thus, the fact that we have a graph rather than a tree structure allows us to represent the sharing and reuse of subcomponents. For example, a situation like the following is impossible to realize in a tree (except by copying whole subtrees).

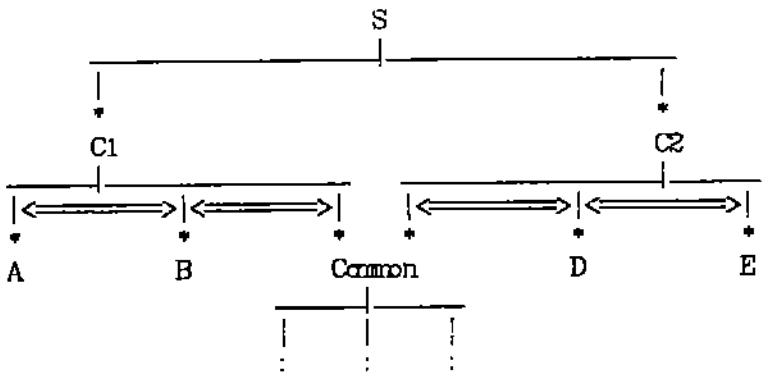

Fig. 2: Configurations C1 and C2 share configuration Common.

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 software module cannot be subdivided. (It is usually assigned to a single programmer.) However, each module normally evolves in a sequence of revisions that are incremental changes to some initial version. These revisions are ordered by their creation time. This situation is diagrammed with an OR node:

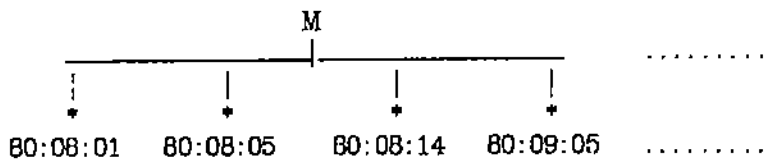

Fig. 3: Revisions of a software module

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.

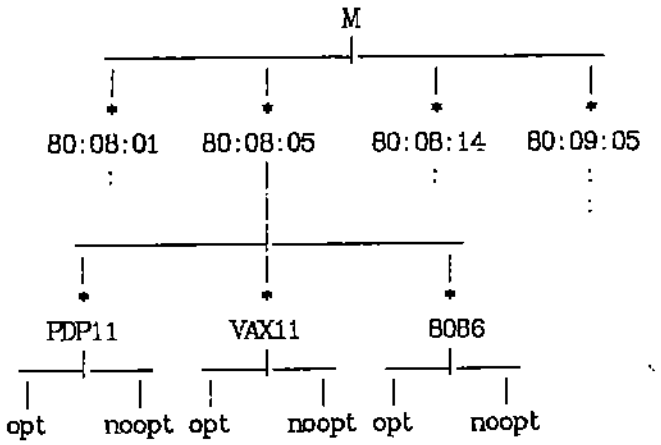

Fig. 4: Revisions, target versions, and optimized versions

Suppose that we would like to add documentation to our module, for example a general description and some implementation details. 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 source code.

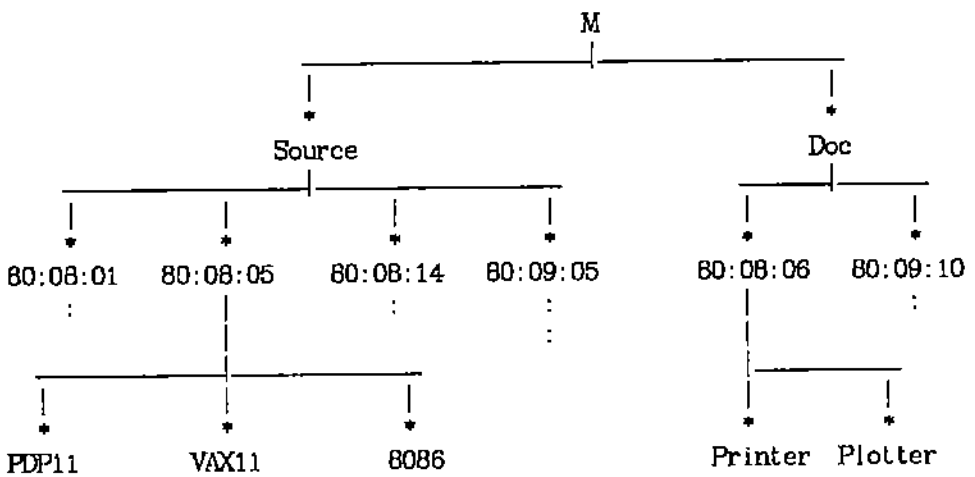

Fig. 5: M has two alternatives, source code and documentation

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. Since the semantics of an OR node are to choose at least one successor, we can even model the view that a program and its documentation form an entity.

It should be clear that the AND/OR graph can be applied to hardware as well. The decomposition into subgraphs may, however, have somewhat different appearances. For example, there may be components that have no revisions, like screws or other standard parts. There may also be additional documents, 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. This can all be managed with AND/OR graphs.

2.2. The Selection Problem

A single AND node may actually represent a number of possible configurations if some of its successors are OR nodes. This leaves us with the problem of selecting a specific configuration. Consider the following example.

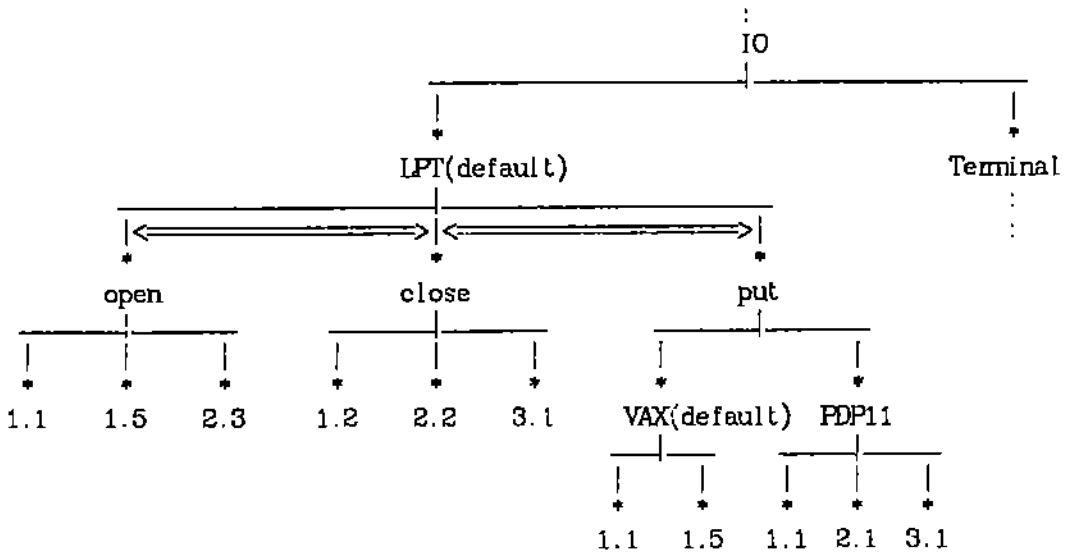

Fig. 6: Several versions of an I/O subsystem.

This fragment 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.

This diagram compactly represents 3*3*(2+3)=45 configurations. In large families, there are easily thousands of configurations that can be created by arbitrary selection of offsprings of OR nodes. Relatively few of them will actually work together. Those are selected with defaults or "cutoff" release numbers. A cutoff release number (or date) selects at each node the revision with the number (or 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 expressed with *IO.Terminal*. Notations for cascading those selections are easily invented.

An additional selection mechanism involves the labeling of OR branches. The labels can then 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 of optimized/unoptimized versions. An example is *IO:PDP11.nonopt*.

2.3. Completeness

The completeness of our model is shown by construction. Note that any configuration can be reduced to a list of primitive components. If there are versions of primitive components, rename each version so that it is a primitive component by itself. Do the same thing for versions of configurations. For each of the renamed configurations, determine the primitive components and make them successors of a single AND node with the name of the configuration. Finally, make all those AND nodes successors of a single OR node. The result is a totally "flat" graph structure with three levels: one OR node at the top, AND nodes at the second level, and primitive parts at the bottom. A refinement of this form will preserve the information that certain primitive components and configurations are versions of each other. This is done by grouping versions as successors of free-standing OR nodes.

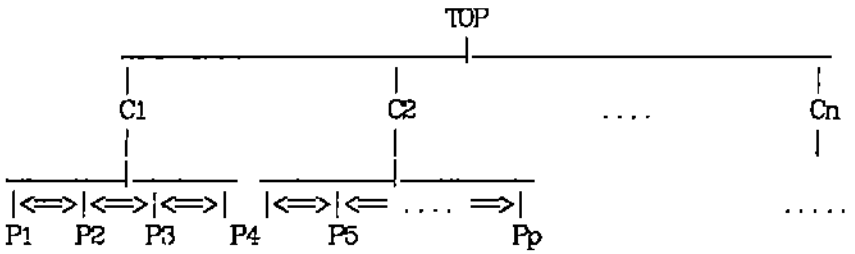

Fig. 7: Normal form for representing all configurations.

2.4. Well-Formed Configurations

A refinement of our model yields the concept of the *well-formed configuration*. This concept is defined in terms of the interfaces between the components of a configuration. It forms the foundations for interface control and system composition. Interface control is important in multi-person projects for establishing and maintaining consistent interfaces between components. System composition is the activity of building new configurations from existing components. Our concept is a generalization of the conditions on system structure presented in [Tic80a] and [Hab80a].

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 those data types, operations, data structures, etc. exported from a node. An example of provided facilities are the visible interfaces of Ada packages [Ich80a]. The required facilities are those types, operations, data structures, etc. that must be imported into the node.

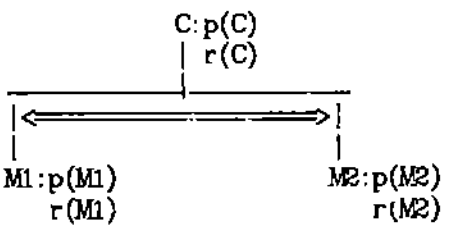

Fig. 8: A configuration with two components; interfaces attached.

The set of facilities provided by a node N is denoted as p(N), the set of required facilities as r(N)*. 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

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

We define two nodes to be compatible if they have the same interface:

Two nodes, N and M, are ***compatible*** if and only if

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

This definition applies to arbitrary nodes; it will be especially interesting for nodes that are successors of the same OR node. Compatible nodes are fully interchangeable.

Similarly, 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

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

* We could link the interfaces into our diagram with some extra OR nodes, but that would only clutter the picture.

We can now define *well-formed nodes*. There are different (recursive) definitions for each node type (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 [Tic79a].)

B. An OR node R with direct successors K1,..,Kn ($n \geq 1$) is well-formed if and only if

i. R is free of contradictions;

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

(Since only one Ki 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 K1,..,Kn ($n \geq 1$) is well-formed if and only if

i. S is free of contradictions;

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

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

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

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

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. They are applied

in search algorithms that compose well-formed configurations automatically. 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 [Tic79a].

In summary, AND/OR graphs provide a basis for the compact representation of all versions of all possible configurations in a system family. Individual versions can be selected by specifying a few parameters and defaults. The concept of well-formed configuration is important for the system designer and yields the basic rules for interface control and system composition.

3. Comparison of other Models for Representing System Families

In this section, we analyze the representations underlying some existing software tools. The comparison concentrates on what kind of AND/OR graph representations the tools permit. We shall see that they place severe restrictions on the shape of the graph. We distinguish the following 4 major models. (Example implementations or proposals are noted in parenthesis.) More detail can be found in [Tic80b].

1) The Hierarchical Model (Algol68C [Cle75a], Simula67 [Bir76a], Mesa [Mit78a], Ada [Ich80a]),
2) The Relational Model ([Bel77a] and [ITT80a]),
3) The Sequential Release Model (SCCS/MAKE [Roc75a, Fel79a]).
4) The AND/OR graph model ([Coo78a, Hab80a, Tic80a, Bux80b]).

The major disadvantage of the hierarchical and relational models is that they make it impossible to express the fact that two components are versions of each other. All versions of a given component must be described separately, even if they are structurally identical and differ only in, say, the release numbers of the modules. Consequently, it is impossible within the hierarchical and relational models to build tools that work on groups of versions rather than individual components.

The first two models also imply a rather inefficient representation, since

even the slightest change in a configuration leads to a duplicate description of the complete configuration. In terms of our general AND/OR graph model, these limitations are due to the fact that there are no OR nodes in these models.

The sequential release model allows primitive components (modules) to exist in a number of revisions. Configurations that are structurally identical and whose modules differ only in the revision numbers can be represented with a single, generic description. Thus, the problem of having to duplicate large numbers of configurations for every minor change is avoided. However, it is still not possible to indicate that two slightly different configurations are actually versions of each other. This is due to the fact that the sequential release model permits no internal OR nodes.

The general AND/OR graph model on the other hand 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 4) permits an AND/OR graph in its full generality.)

3.1. The Hierarchical Model

The hierarchical model is characterized by an AND/OR graph that has no OR nodes at all. Thus, multiple versions cannot be represented. Each AND node denotes a single configuration. Every configuration, even if only slightly different from another one, must be described separately. Similarities among configurations cannot be expressed.

There are two subclasses of the hierarchical model: one is a strict AND tree, the other a (loopfree) AND graph. The AND tree allows no sharing of nodes whatsoever. This is a serious problem in large system families because it results in massive copying and updating problems. However, the AND tree is well suited to block-structured languages: The nesting of blocks can be expressed directly

with an AND tree. This fact has been exploited in Algol68C [Cle75a] and Simula67 [Bir76a], where inner blocks can be extracted from a program, compiled separately, and linked in later. A mechanism that stores intermediate symbol tables and enforces the scope rules of block structure guarantees that the (single) configuration is well-formed.

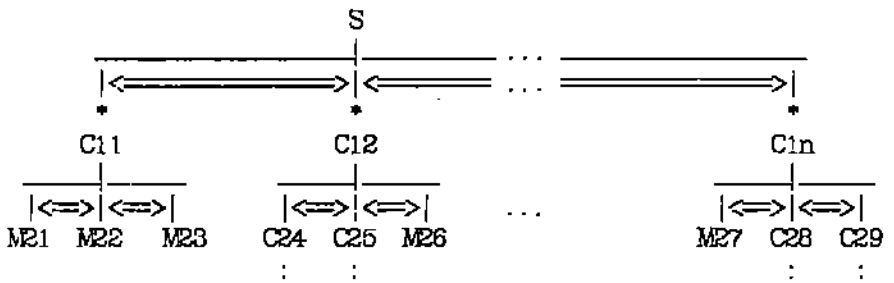

Fig. 9: A fragment of an AND tree.

In the simplest, nontrivial form, the AND tree has two levels: The root is the name of the configuration, and the second level consists entirely of leaves (modules). This form it is often used in linker command files or in parameters to compiling and linking commands. Clearly, not much can be done with such a primitive representation.

The MESA [Mit78a] and Ada [Ich80a] languages permit the representation of AND graphs. This can be used to express the sharing of common components among several configurations. Elaborate rules on scope and compilation order ensure well-formedness. However, since there are no OR nodes, multiple versions of primitive components and configurations cannot be described within these languages. We consider this to be a major flaw of the Ada programming language.

In all variants of the hierarchical model, change is handled in the following way. Whenever one of the modules is edited, the new revision replaces the old one. Saving of the old version must happen outside the model. As an example, consider Ada: the recompilation rules that must be obeyed after a modification effectively discard all old versions.

A straightforward extension of the hierarchical model is to include compiled versions of the primitive parts. This is needed to save recompilation times of unmodified modules. The technique is used in the SDC system [Not79a], the MAKE program [Fel79a], and the Mesa language implementation.

This situation can be modeled with our previous AND graph, where the leaf nodes are replaced with OR nodes. Each OR node has two branches, one for the source version, the other for the object code version. The object code version is automatically kept up to date with the source version.

Note, however, that we have gained nothing besides avoiding redundant compilations. Since the object code and source version of a single OR node are essentially the same, we have not added an extra degree of freedom. Thus, we still have the problems of the hierarchical model: (1) no representation of multiple versions, (2) duplication of information for nearly identical versions, (3) no sharing of subsystems in the tree case, and (4) documentation must be handled separately.

3.2. The Relational Model

A system family with several configurations can be represented using a boolean matrix. The primitive components are listed across the top of the matrix, and the configurations down the side. The entry in row i, column j is set to 1 if and only if part j is included in configuration i.

	M1	M2		Mn
C1	1	1		
C2		1		1

Fig. 10: A relation with two configurations.

Clearly, each row can be modeled with a single AND node. The resulting structure is a flat graph with 3 levels. The top level is a single OR node, branching out to the AND nodes for the matrix rows. The bottom level consists of the

primitive components. Since these can be shared among several AND nodes, we have an acyclic graph, not a tree.

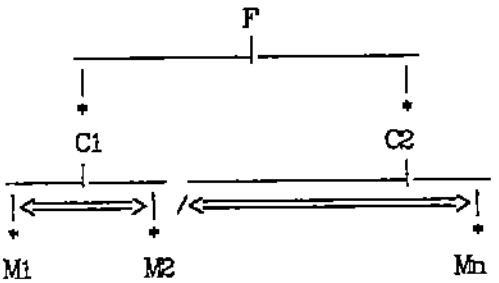

Fig. 11: The relation of Fig. 10 represented as an AND/OR graph.

The problem with the matrix is that it is huge and sparse. Fortunately, a few simple refinements can eliminate much of the sparseness (see, for example [Bel77a] and [ITT80a]). These refinements involve the placing of version numbers into the matrix, and a decomposition of the matrix into several relations. However, note that each configuration stands for a single, fixed version. Generic configurations cannot be entered.

Using our AND/OR graph to analyze the resulting structure, we note that these refinements lead to a deeper graph of cascaded AND nodes, but no additional OR nodes besides the one on top. Compare now the hierarchical model. Recall that that model is characterized with a pure AND graph. If we make all "free" nodes in that graph (i.e. nodes without predecessor) to successors of a single, artificial OR node, we see that *there is no essential difference between the hierarchical and relational models*: they both result in the same AND/OR graph. This is one of the major conclusions we can draw from the AND/OR model.

Both the hierarchical and the relational model suffer from the fact that there are no internal OR nodes. As an example for the kinds of difficulties this may cause, consider the update problem. Suppose we modify an existing source module M, thereby introducing a new revision M'. To incorporate the revision, we have to collect all configurations that contain M, make copies of them, and replace M with M' in the copies. If M is a frequently used module, this may result in nearly doubling the size of the relations or the graph! Since revising

programs happens rather frequently, this representation is clearly unacceptable.

3.3. The Sequential Release Model

In the sequential release model, a primitive component is stored as a number of revisions which are numbered sequentially or marked with a creation date. A particular version of a configuration is selected by means of a cutoff release number or a cutoff date. Out of the potentially numerous revisions of each primitive component, the one with a date or number less than or equal, but closest to the cutoff is chosen.

The sequential release model has the following AND/OR graph. The top node is an OR node whose successors are strictly AND nodes. The AND nodes may then be cascaded for several levels. The nodes at the next to lowest level consist of OR nodes which branch out into different revisions.

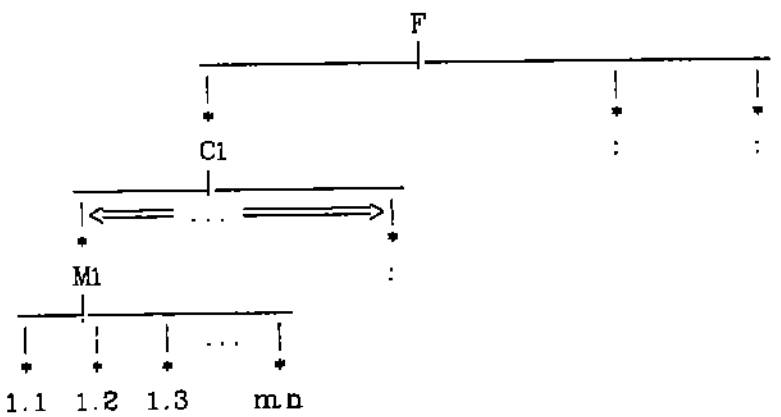

Fig. 12: Example for the sequential release model.

The sequential release model allows a compact representation of structurally identical configurations. It is the model of choice for a rapidly evolving system family with few configurations, because the addition of new revisions can be handled automatically and does not complicate the representation. The model is easily refined to accommodate compiled and linked versions as well. An example for its use is a combination of the tools SCCS and MAKE [Roc75a, Fel79a]. We have not found an adequate solution for the problem of well-formedness in this model.

A disadvantage of the sequential release model is the fact that it allows no internal OR nodes, except for revisions. Thus, all configurations that can be derived from a single AND node are structurally identical. Structural differences, however slight, must be pushed all the way to the top. It is therefore not possible to indicate that two slightly different configurations are similar or versions of each other. In addition, this causes duplication of information. Another problem caused by the lack of internal OR nodes is that documentation cannot be tied in conveniently.

3.4. The AND/OR graph model

The representation proposed by Cooprider [Coo78a] allows internal OR nodes, yet still poses a number of problems. Cooprider took a very general view in that he did not distinguish between different types of versions. We found this view unsatisfactory, because it implies that the programming environment has little or no knowledge about the kind of objects it maintains. This puts an extra burden on the programmer in that he must repeatedly reprogram how a given object should be handled. This can be avoided by including more type information about the objects.

Habermann [Hab80a] and Tichy [Tic80a] allow almost complete AND/OR graphs, except that revisions, derived versions, documentation, and combinations of hardware and software are not treated sufficiently, and the concept of well-formedness needs refinement. The selection problem is not addressed adequately. The questions of exploiting the information in the graph for automatic configuration generation, search and selection functions, and other novel software tools are still open.

Another example for system families can be found in the Stoneman document [Bux80b]. Since this paper lists only the general requirements, it is somewhat difficult to predict a future design resulting from it. We note that internal OR nodes are explicitly prescribed by the requirement that "configurations ... may ... exist in version groups". However, this statement in itself does not require that configurations may consist of version groups rather than individual components. Another drawback is that the requirements preclude successive

OR nodes (versions of versions), because version groups are not "objects". (Only "objects" may belong to a version group.) Compare Fig. 4, 5, and 6 of Section 2 for examples where successive OR nodes are used for different types of grouping. "Partitions" introduce an additional class of OR nodes, mostly used for top-level administrative divisions.

4. General Criteria for Programming Support Environments

In this section, we shall state a number of properties that are desirable for data bases on which programming support environments are built. These properties are derived from our AND/OR graph model. As such, they are general in that they do not depend on a particular host system or development organization. We feel that the following list is the minimum that a modern PSE data base should provide.

I. *Full AND/OR Graph*

From the discussion in the previous section, it is clear that a full AND/OR graph should be permitted. In particular, we need internal OR nodes to express version groups of configurations and primitive components, as well as version groups of version groups. AND nodes should permit arbitrary configurations of individual components (configurations and primitive components) as well as version groups. The data base should not be limited to software, but should also store firmware, test programs, test data, documentation, and hardware descriptions. Sharing of common components and version groups among configurations and other version groups should be possible without duplication of information. A rich and flexible set of selection mechanisms should be available to choose individual versions.

II. *Customization of the AND/OR Graph*

If a development group decides to work with a submodel of the general model, then this should be possible without complications or performance penalties due to the general model. For example, if a programmer develops a single-version program using the hierarchical model, he should not be bothered with extra complexity or generality. Using the sequential release

model should also result in a significant simplification.

Conversely, scaling up from a simple model to a more general one should be possible at any time.

The AND/OR graph will take different shapes for different organizations, for hardware and software, for different languages and language processors. A set of standard arrangements with certain default structures for documentation and test beds should be provided.

III. Suppression of Detail

The data base for a large system family may become extremely complex, and there must be mechanisms to suppress unwanted detail. Certain standard components like documentation and test data need not be shown always. Derived versions like precompiled and prelinked sub-configurations should usually be suppressed, so that the user can think and work in source representations only. This implies that the data base must reliably handle storage, deletion, and regeneration of derived versions.

Note that the AND/OR graph is only a model. We do not advocate AND/OR graphs as the interface to the user. It seems that a representation in the form of a module interconnection language as proposed in [DeR76a] is more appropriate. However, the conceptual simplicity of the AND/OR graph makes it possible to study a large number of issues in an abstract form, such as search and synthesis algorithms for configurations, interface control, and the automation of configuration management.

5. Conclusions

We developed the AND/OR graph model as a tool for abstractly describing families of programmed systems. Three submodels were defined and used to evaluate the representations underlying various software tools. The model yielded basic requirements for the data base on which programming support environments are built. An overview of future research was given.

Acknowledgements. This research was supported in part by International Telephone and Telegraph during the author's stay at the ITT Programming Technology Center.

References

Bel79a. 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).

Bel77a. 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).

Bir76a. Birtwistle, G., Enderin, L., Ohlin, M., and Palme, J., "DECsystem-10 Simula Language Handbook Part 1," C8398, Swedish National Defense Research Institute (March 1976).

Bux80b. Buxton, John N., *Requirements for Ada Programming Support Environments ("Stoneman")*, US Department of Defense (February 1980).

Bux80a. 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).

Che79a. 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).

Cle75a. Cleveland, J.C., *The Environ and Using Facilities of Algol 68C*, Computer Science Department, UCLA (April 1975). Modeling and Measuring Note, No. 33.

Coo78a. Cooprider, Lee W., *The Representation of Families of Programmed Systems*, PhD thesis, Carnegie-Mellon University, Department of Computer Science (1978).

DeR76a. DeRemer, Frank and Kron, Hans H., "Programming-in-the-Large vs. Programming-in-the-Small," *IEEE Transactions on Software Engineering* **SE-2**(2) pp. 80-86 (June 1976).

DoD77a. DoD,, "Department of Defense Requirements for Higher Order Computer Programming Languages, Revised Ironman," *SIGPLAN Notices* **12**(12)(Dec. 1977).

Els73a. Elson, Mark, *Concepts of Programming Languages*, Science Research Associates, Inc. (1973).

Fel79a. Feldman, Stuart I., "Make - A Program for Maintaining Computer Programs," *Software – Practice and Experience* **9**(3) pp. 255-265 (March 1979).

Hab79a. Habermann, A. Nico, "An Overview of the Gandalf Project," in *CMU Computer Science Research Review 1978-1979*, Carnegie-Mellon University (1979).

Hab76a. 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).

Hab80a. 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).

ITT80a. ITT,. *CMSS3 Users's Manual*, International Telephone and Telegraph (1980). Document No. 211ITT26366-PC

Ich80a. Ichbiah, Jean D., *Reference Manual for the Ada Programming Language*, United States Department of Defense (July 1980).

Ker79a. Kernighan, Brian W. and Mashey, John R., "The UNIX Programming Environment," *Software – Practice and Experience* 9(1) pp. 1-15 (Jan. 1979).

Ker78a. Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, Prentice-Hall (1978).

Lov80a. Love, Tom, *Configuration Control for Telecommunications Systems*, personal communication. 1980.

Mit78a. Mitchell, James G., Maybury, William, and Sweet, Richard, *Mesa Language Manual*, Technical Report, Xerox Palo Alto Research Center (Feb. 1978).

Nil71a. Nilsson, Nils J., *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill (1971).

Not79a. Notkin, David S. and Habermann, A. Nico, *SDC Documentation*, Carnegie-Mellon University, Department of Computer Science (1979).

Par76a. Parnas, David L., "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering* SE-2(1) pp. 1-8 (Mar. 1976).

Par79a. Parnas, David L., "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering* SE-5(2) pp. 128-138 (March 1979).

Rid80a. Riddle, William E., "Panel: Software Development Environments," pp. 220-224 in *Proceedings of COMPSAC 80*, IEEE Computer Society Press (Oct. 1980).

Rig69a. Rigney, Joseph W. and Towne, Dougals M., "Computer Techniques for Analyzing the Microstructure of Serial-Action Work in Industry," *Human Factors* 11(2) pp. 113-121 (April 1969).

Roc75a. Rochkind, Marc J., "The Source Code Control System," *IEEE Transactions on Software Engineering* SE-1(4) pp. 364-370 (Dec. 1975).

Tic79a. 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).

Tic80a. Tichy, Walter F., *Software Development Control Based on System Structure Description*, PhD Thesis, Carnegie-Mellon University, Department of Computer Science (Jan. 1980).

Tic80b. Tichy, Walter F., *Configuration Control Tools in the 1980's*, Technical Report, ITT Programming Technology Center (1980).

Wir71a. Wirth, Niklaus, "The Programming Language Pascal," *Acta Informatica* 1 pp. 35-63 (1971).