

/MECHANICAL TRANSLATION OF SOFTWARE REQUIREMENTS SPECIFICATIONS:
FROM ENTITY-RELATIONSHIP-ATTRIBUTE TO WARNIER-ORR

by m

DONALD E. WOLFE

B.S., The Ohio State University, 1977

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

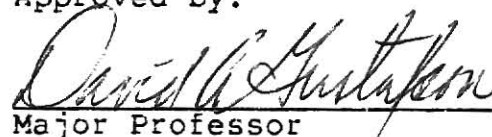
Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1984

Approved by:


Major Professor

LD
2668
R4
1984
W64
C. 2

A11202 665459

CONTENTS

1.	Chapter One.....	1
1.1	Introduction.....	1
1.2	Definitions and Restrictions.....	5
1.3	Literature Review.....	7
1.4	Summary in Relation to the Literature.....	11
2.	Chapter Two.....	13
2.1	Requirements.....	13
2.2	Design Decisions.....	16
3.	Chapter Three.....	18
3.1	Design Overview.....	18
3.2	Hierarchy Diagrams.....	19
3.3	Data Structures.....	24
4.	Chapter Four.....	28
4.1	Implementation.....	28
4.2	Testing.....	30
5.	Chapter Five.....	31
5.1	Conclusions.....	31
5.2	Possible Extensions.....	33
	Appendix A -- Process-language BNF specification.....	35
	Appendix B -- User's Manual.....	36
	Appendix C -- Source Code.....	41
	References.....	58

LIST OF FIGURES

Figure 1.	Masters-project WO-diagram.....	20
Figure 2.	ERAIN WO-diagram.....	21
Figure 3.	PLorder WO-diagram.....	22
Figure 4.	WOedit WO-diagram.....	22
Figure 5.	WOout WO-diagram.....	23

1. Chapter One

1.1 Introduction

The generation of software from requirements specifications with absolute fidelity to those specifications is an important goal of software engineering. Other goals such as increasing productivity or improving the man-machine interface are also worthwhile, but they must be viewed as strictly secondary to the goal of providing correct answers to data processing problems. In order to be able to engineer the correct solutions consistently, the discipline is increasingly turning to the computer itself to help manage software generation. Certainly a major factor behind this is the fact that the problem has consistently proven itself to be beyond our unaided ability to solve. Software engineering would benefit greatly if programs could be generated directly from requirements specifications in the same way that compilers generate machine instructions from today's high-level languages.

To the extent that program generation can be automated, software engineering will be freed from its current preoccupation with managing the software "production line" and allowed to address its other pressing concerns. Automatic program generation from requirements specifications is thus a legitimate and important goal of

software engineering. This goal immediately establishes a common concern between software engineering and the automatic programming (AP) subfield of artificial intelligence. ⁽¹⁾

Unfortunately program generation from unconstrained requirements specifications is not yet feasible due to various reasons, most notable of which is the fact that we simply cannot yet describe in suitable detail the thought processes that go into creating algorithms. Some progress has been made by workers in AP, but they are still woefully short of software engineering's goal. The current state of the art in AP depends on the constraints that one is willing to accept -- whether the starting point in the program generation process is a very high level language (VHLL) specification ⁽¹⁾ or a natural language problem statement, whether the AP system must be fully generalized or if it may be limited to a specific problem area, etc. The programs that can be automatically generated today by completely general AP systems (pattern-matching algorithms for example) are still almost toys compared to the programs that software engineering requires. The programs that can be generated by more restricted knowledge-based systems are more sophisticated, but still inadequate from software engineering's viewpoint. On the other hand, software engineering does not yet appear to understand the problem of

designing systems well enough to benefit from sophisticated algorithm generators even if they were available. Hopefully a mutually beneficial synergy will result between these two fields in the future.

Mechanical translation of requirements specifications is a useful intermediate goal from software engineering's viewpoint since it gives the user additional tools with which to control the software generation process. This pragmatic justification by itself is certainly sufficient to warrant additional work in this area. The "horror" stories that appear in various trade magazines attest to the urgent need for improved tools to help manage software generation. Any mechanical aid that would help reduce the occurrence of missed deadlines and delivered products that don't satisfy the user's requirements would be of real benefit.

As a useful side effect, mechanical translation of requirements specifications would also aid the ultimate goal of automatic programming by high-lighting the issues that must be addressed before that will become possible. The sheer number of competing methodologies illustrates the fact that we don't yet really know the best way to turn a requirements specification into software, or even the best way to create the specification in the first place. Howden's survey ⁽²⁾ for example lists seven different requirements methodologies and nine different design

methodologies -- without being exhaustive. Perhaps there isn't a "best" way to deal with software specifications. Perhaps some methodologies are better than others for a certain class of problems while others are more appropriate for a different class, just as some programming languages are more appropriate than others in a given situation. Hopefully the effort required to create software capable of transforming requirements specifications into design specifications and design specifications into software will help to clarify what advantage (if any) a given methodology has when compared with the others.

The goal of this project was to translate an Entity-Relationship-Attribute (ERA) requirements specification into a Warnier-Orr hierarchical design specification. ERA was selected as the source specification primarily because it fit in with several other Masters' projects being worked on at the same time. Warnier-Orr was selected as the target specification primarily because of personal preference; in an earlier software engineering course it had impressed me as the methodology that would have been most appropriate for dealing with business data processing problems, i.e. input-output intensive file/data-base processing programs.

1.2 Definitions and Restrictions

The term "requirements specification" will be used in this paper to mean the specification of a computational process in terms of what the end-user requires, while the term "design specification" will be used to mean the more detailed specification of a process in terms that reflect how the process will actually be implemented so as to accomplish the goals of the requirements specification. Normally any reference as to "how" a goal is to be accomplished is inappropriate in a requirements specification. For example, a requirements specification may legitimately specify that an output file should be ordered based on a specified key, but it would be inappropriate for it to specify the sorting algorithm to be used. Similarly, a requirements specification should specify the input and output characteristics of a process, but it is inappropriate for it to specify the format (or even the existence) of any intermediate data that may be necessary for the process. Having stated this rule, it is now conceded that there are occasionally cases when a requirements specification may legitimately specify details of how a process is to be implemented; these special cases arise when the requested process will be embedded into an existing system. Excessively detailed requirements specifications are to be avoided for the practical reason

that they will constrain the design specification unnecessarily, and possibly prevent the achievement of the best solution to the problem being addressed.

The Entity-Relationship-Attribute specification that was defined to be the input to this project requires some additional explanation. To begin with, there is no such thing as an Entity-Relationship-Attribute methodology in the sense that there is a Warnier methodology; the ERA specification referred to above and through the remainder of this paper is simply one instantiation of a specification that might be derived by appealing to the concepts associated with the "Entity-Relationship" mode of thought. This approach derived originally from attempts to abstract database models (and is still most active in the database area), but is now beginning to be applied in other areas. The basic concept is extremely general, simply that any system can be described as a collection of entities (and associated attributes) together with their relationships to other entities. The specification that served as input to this project was simply a definition of a computational system where the entities were defined as computational processes (Activity) and data items (Input, Output, Input/Output) and the relationships were the connections between them. Comments about the ERA specification are in reference to this particular instantiation. There are

doubtless many other "ERA" specifications that are equally valid from a theoretical standpoint and that may be either better or worse than this one based upon the standards of evaluation used in this paper.

The selection of Warnier-Orr as the target methodology suggested that the range of problems to be dealt with be somewhat restricted. Warnier's methodology specifically addresses itself to programs for manipulating files of records as opposed to non-record oriented programs, e.g. operating systems. While this methodology could be stretched to cover applications other than file/data-base processing, these other types of problems are not its "natural" realm. Discussions and subjective evaluations in this paper will all be in the context of appropriateness to file/data-base processing, which is where the Warnier-Orr methodology is most suitable.

1.3 Literature Review

Manna and Waldinger's article (3) attempted to illustrate the current capabilities of the automatic programming effort by first discussing general techniques, and then applying them in the context of a program synthesizer that they are developing. The sample programs they included in their article dealt with sets and operations thereon, had only a small amount of intrinsic knowledge about the problem

domain, and worked with a VHLL input styled somewhat like the Kent Recursive Calculator. Their approach relied heavily on the reasoning ability of the program synthesizer since it incorporated very few heuristics about the problems to be solved. They concluded, "We hope we have managed to convey ... the promise of program synthesis, without giving the false impression that automatic synthesis is likely to be immediately practical."

Given the obvious superficial similarity between this project as a specification-transformation and the program-transformation approach to AP, one might hope for some help from that area. Unfortunately Partsch and Steinbruggen's ⁽⁴⁾ survey of work in this specific area indicated that no one is currently investigating the program-transformation approach to the class of programs that this project is concerned with. Virtually all of the work surveyed dealt with either non-numerical functions written in LISP (in one of its various incarnations) or matrix-processing programs in FORTRAN or Pascal.

Barr and Feigenbaum's discussion of automatic programming ⁽¹⁾ indicated that while a fully generalized AP system may not be feasible in the near future, suitable restriction of the problem domain may make specialized systems realizable. Of particular interest was Protosystem I, an automatic-programming system being developed at the M.I.T. Laboratory

for Computer Science. This system is reported to be currently capable of generating reasonably optimized PL/I programs and the associated IBM OS/360 Job Control Language statements for input/output intensive batch-oriented problems. The current input to this program generator is the program statement in a very high level language, SSL.

Hammer and Ruth ⁽⁵⁾ provided an evaluation of automatic programming today by comparing and contrasting the two paradigms of VHLL and knowledge-based systems. Beyond the discussion of AP capabilities however, they also pointed out the very real benefits that this research will have for other areas of computer science (such as software engineering). Some of these benefits include the development of improved specification languages (as a result of better understanding exactly what must be specified), a better understanding of the programming process itself, and a more rigorous taxonomy of reasonable program structures.

The generality of the Entity-Relationship methodology (as well as its strong attachment to the data-base discipline) was illustrated by the papers presented at the International Conference on Entity-Relationship Approach to Systems Analysis and Design. Teichroew et al ⁽⁶⁾ asserted that their Problem Statement Language was based on the concepts of entities, relationships, and attributes, and then reported on a project aimed at modeling various design methodologies

(including Warnier's) in PSL. Solvberg ⁽⁷⁾ attempted to show that the Entity-Relationship approach could be used to attempt to harmonize different world-views rather than forcing different groups to compromise on a view that was not totally natural to either one (for example, the view that software designers have of the enterprise's information store versus the view of the user community).

Warnier's two books combined to enunciate an almost algorithmic approach for translating a requirements specification ("Unit of Achievement" or UA in his terminology) into appropriate programs. Logical Construction of Programs ⁽⁸⁾ (LCP) dealt with the mechanics of deriving a program from specified data structures, while Logical Construction of Systems ⁽⁹⁾ (LCS) dealt with the larger questions of the design of optimal bases of data and systems of programs to deal with an organization's data processing requirements. It is precisely this concern with addressing the organization's entire base of data that Orr ⁽¹⁰⁾ cited as the major advantage the Warnier-Orr methodology has over its competitors.

Steward ⁽¹¹⁾ attempted to downplay the competing claims of the proponents of the various design methodologies, and stressed the common goals and compatibilities of the different approaches. To illustrate this, he demonstrated a technique for converting data-flow diagrams into Warnier-Orr

hierarchical diagrams and vice versa. Accomplishment of this required some notational extensions to both forms, which had the nice side effect of increasing the usefulness of each. Warnier-Orr diagrams acquired indicators of the data elements "moving" from activity to activity -- which provided a graphic illustration of the extent of coupling in a given design. Data-flow diagrams acquired control information indicating when different flows were mutually exclusive -- which certainly helps to explicate control flow when this technique is used.

Griffiths' article ⁽¹²⁾ surveyed seven major methodologies in an attempt to identify both their immediate usefulness based upon their existing capabilities and their prospects for long-term value based upon their fundamental assumptions.

1.4 Summary in Relation to the Literature

Griffiths' evaluation consistently favored the "data-structured" methodologies of Warnier and Michael Jackson over their competitors primarily because they derive their problem solutions (programs) directly from the problem statement (data). He generally favored Jackson's Structured Design methodology over Warnier's because of its greater ability to handle the "structure clashes" that occur when radically different data organizations must be processed together, but he remarked, "...it may be wondered,

especially since the mechanisms used to resolve the structure clashes are themselves complex, whether the LCP/LCS emphasis on simple data structures at the system level -- in MJSD [Michael Jackson Structured Design] terms the design of the system with the fewest possible structure clashes -- is not more profitable than the MJSD approach which, by concentrating on program design is willing to accept a situation where many clashes have to be resolved."

It should be explicitly noted here that this project does not implement Warnier's methodology, it merely attempts to translate an ERA specification into a Warnier-Orr style diagram. Warnier's methodology by definition uses the data's structure to derive the program's structure. As soon as an ERA specification descends below its uppermost level where the program's global input/output requirements are stated, it dictates a priori elements of the program's logical structure which may or may not be in harmony with the program structure that would be obtained by derivation from its data definitions. This does not however invalidate this project. On the contrary, the project's handling of Warnier-Orr style diagrams actually coincides with the actual usage of this technique by many practitioners. A good example of this is Steward's article; he used Warnier-Orr style diagrams simply to take advantage of their clear hierarchical structure and largely ignored Warnier's

methodology for their derivation.

Steward's article provided an alternate basis for this project with its assertion that translations between the various design methodologies are both possible and reasonable. As it turned out, the translation between the two methodologies addressed in this project was relatively straightforward. The real problems encountered arose when an effort was made to derive the program control indicators normally associated with Warnier-Orr diagrams. These control indicators (alternation, iteration, etc.) were not explicitly given in the ERA Activity description. Neither could they be inferred from an Activity's data linkages since no syntax was specified for indicating optional or mutually exclusive input/output items. This lack is currently addressed by the provision for post-translation editing and updating of the design specification, as will be detailed the the following chapters.

2. Chapter Two

2.1 Requirements

The first requirement was that of basic functionality; the system had to read in ERA specifications and produce Warnier-Orr style diagrams. This requirement, however, was not really as straight forward as it sounded. The fact that

this project is part of a larger software engineering research tool meant that its environment would be subject to change. This is especially true in relation to the ERA specification itself, which has already undergone several minor changes during the past year. In order to protect the system as much as possible from these changes, a minimal subset of the ERA specification necessary for this project's purposes was defined and documented (see the User's Manual in Appendix B) and any unrecognized data in the input text file is simply passed over without comment.

The second requirement was that the system would run under the UNIX* operating system, and be written in either C or Pascal. UNIX was chosen because its currently expanding popularity combined with its availability on a variety of different computers seems to indicate its long-term viability as an operating system. C language was an obvious candidate because of its association with UNIX, and Pascal was considered a possibility because of its current popularity with various academic institutions (including Kansas State).

The third requirement was to use the information available in the ERA specification to impose a suitable structure on

* UNIX is a trademark of AT&T Bell Laboratories

the Warnier-Orr hierarchy. Part of the project actually consisted of defining exactly the subset of the available information that was appropriate for the purpose at hand.

The fourth requirement was to maintain a relatively high degree of modularity so as to minimize the impact of future changes to the system and/or its input. Given the prototypical nature of the software engineering tool that this project is a part of, this was clearly an important requirement. It also had the benefit of aiding the development of this project itself. Individual functions could be developed without impacting the rest of the system, and then integrated with the other functions after testing and debugging.

The fifth requirement was to provide a simple interface whereby additional functions could be added to the system at a future time. This was important due to the severely limited amount of code that would be generated by any one individual project. It was hoped that provision of such an interface might encourage future projects to also use the same conventions so that ultimately a significant body of software capable of working together would be generated.

2.2 Design Decisions

The first requirement had little real impact on the logical structure of the system; it merely defined the input and output. The format of the Warnier-Orr diagram produced by the system followed the pattern of showing the data-flow along the branches of the hierarchy (as suggested by Steward). Given the nature of an ERA specification, the data names associated with each Activity were easily available. Their availability on the hierarchy chart provided a nice visual means of assessing the coupling present in the design.

All of the modules in this system were written in C-language in compliance with requirement two, and because C is by definition available on any computer that runs UNIX while Pascal is not. C also by virtue of its close association with UNIX has standard access to various system functions, while the interface between Pascal and UNIX is not so well defined.

The third requirement suggested the creation of a directed graph to represent the specification, since that would make it easier to manipulate. The encapsulization of functions (PLin and PLOut) to automatically build the graph from a text file (and vice versa) made this complex data structure feasible in spite of the high modularity of the system.

The second, fourth, and fifth requirements combined to recommend the use of a structured text string to pass information between the various modules in the system. The wide range of utilities that UNIX provides to support text files makes them a natural choice for a communication medium between modules running under it.

The ERA requirements specification has no syntactical element to indicate either iteration of an Activity or that an Activity might never occur. Therefore it will be initially assumed for translation purposes that every Activity occurs exactly one time. If this is not the case, then the process-language text must be updated subsequent to the initial processing of the ERA specification. An interactive editor was written specifically to facilitate this necessary function.

Since Warnier's methodology seems entirely adequate for dealing with the class of programs it chooses to address, this project did not embellish upon his minimal set of control structures. In LCP he stated ('), "All programs, however complex, can be built up by nesting, in hierarchical order, repetitive and alternative structures." In keeping with this view, the only control specifications supported by this system are repetition and alternation, the second of which automatically implies a repetition factor of "0 or 1 times". Also in line with Warnier's methodology, an

alternative processing path must have its alternative specified (even if it is a null procedure). This is analogous to the COBOL formulation:

IF <condition> THEN <action> ELSE NEXT SENTENCE.

The editor function will enforce this by generating a dummy operator (prefixed with "!") in cases where an alternative operation does not exist.

3. Chapter Three

3.1 Design Overview

The most salient feature of the design is the process-language text string which provides a central communication medium for all the modules in the system. Its structure is detailed in a later section on "Data Structures". It contains all of the information that the system needs to generate Warnier-Orr style diagrams for a given specification.

The internal representation of the process-language is a directed graph (digraph) consisting of "operator" and "argument" nodes. The digraph's information content is identical to that of the process-language text from which it is derived. Its structure is likewise detailed in the "Data Structures" section.

The process-language specification is critical since it is ultimately intended that the information contained therein combined with that in the data-dictionary will be sufficient to fully determine a correct implementation of the requirements specification.

"Standard" function PLin reads the process-language string from the input file (usually stdin) and builds the internal operator-argument graph, while POut converts the internal graph into process-language text and writes it to the output file (usually stdout).

3.2 Hierarchy Diagrams

3.2.1 Overview of the Entire System The system was originally conceived of as the two modules ERAin and WOout communicating via the process-language interface. The module POrder was added later so as to relieve the user from the necessity of having to physically order the ERA Activities into their logical sequence. The module WOedit was created to facilitate on-line viewing and updating of the process-language text in a more user-friendly environment than that provided by the standard UNIX editor. This system overview is diagrammed in Figure 1.

3.2.2 ERAin This module reads the ERA specification from standard input and builds operator and argument nodes for each ERA Activity. It then uses the "subpart_is"

	<ERA_specification
	-ERAIN
	(1 time)
	>PL_specification
	<PL_specification
	-PLorder
	(1 time)
	>PL_specification
<ERA_specification	
-Masters_project-----	
(1 time)	
>Warnier-Orr_diagram	
	<PL_specification
	-WOedit
	(0 or 1 times)
	>PL_specification
	(+)
	-!WOedit
	<PL_specification
	-WOout
	(1 time)
	>Warnier-Orr_diagram

Figure 1. Masters-project WO-diagram

relationships that were coded to rearrange the operator nodes. It continues to make passes over the graph rearranging operators until it makes one complete pass during which it could not rearrange any operators. It then writes the specification to standard output via PLout. The gross structure of this module is diagramed in Figure 2.

3.2.3 PLorder This module reads the process-language specification from standard input via PLin, which constructs an appropriate digraph. It then makes one pass over the graph reordering operator-nodes based upon their input-

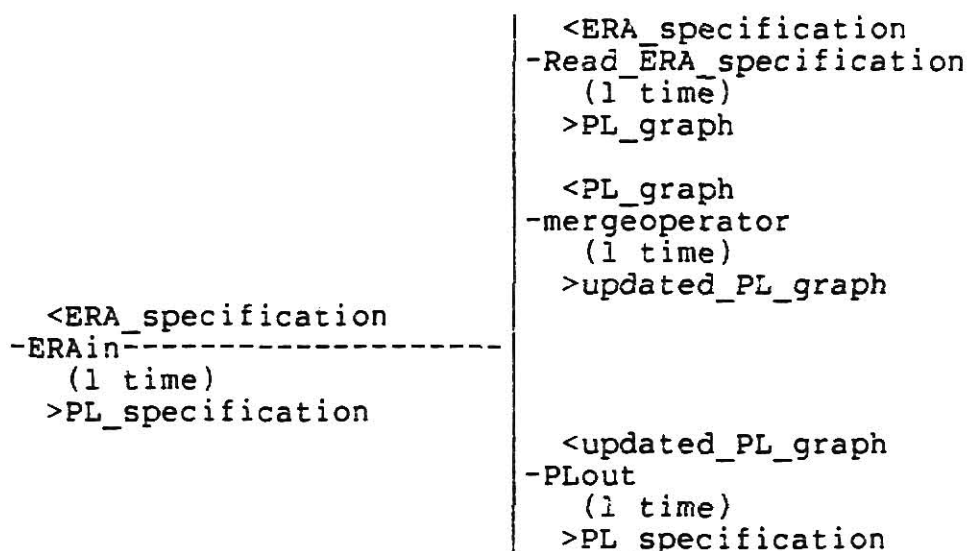


Figure 2. ERAin WO-diagram

output arguments. Within each level (a brace encompasses one level in a Warnier-Orr diagram), all operators whose input is not generated by another operator on that level is moved to the beginning (top) of that level. All operators whose output is not consumed by another operator on that level is moved to the end (bottom) of that level. It then writes the modified process-language specification to standard output via PLout. The gross structure of this module is diagramed in Figure 3.

3.2.4 WOedit This module facilitates interactive viewing and updating of the specification. The format of the displayed information is Warnier-Orr in the same style used by WOout, but in this case only a "window" covering the "current" operator and its suboperators is displayed. Commands are supplied to allow the operator to change the

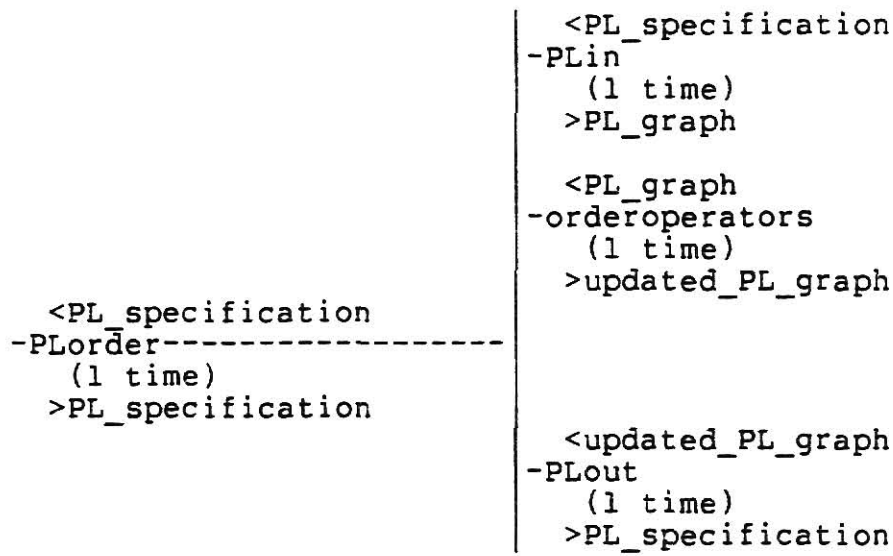


Figure 3. PLorder WO-diagram

"current" operator, display the current "window", update repetition counts, indicate alternative operators, and to update the process-language text on secondary storage. The gross structure of this module is diagramed in Figure 4.

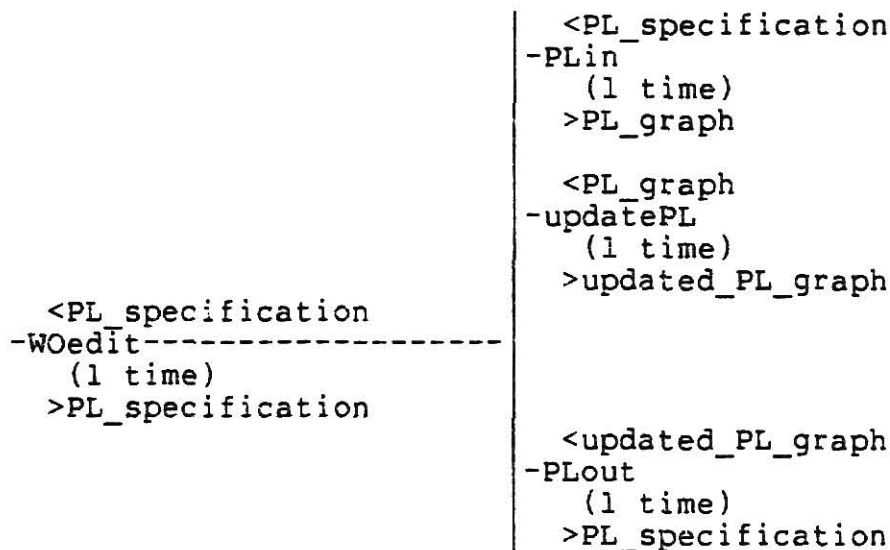


Figure 4. WOedit WO-diagram

3.2.5 WOout This module reads the process-language specification from standard input via PLin. The prune function then uses the input parameters (or its defaults) to break excessively deep operator nestings so that the output diagram will fit on a page. The recursive function printoperatorlist then writes the operator-argument digraph to standard output in the style of a Warnier-Orr diagram. The gross structure of this module is diagramed in Figure 5.

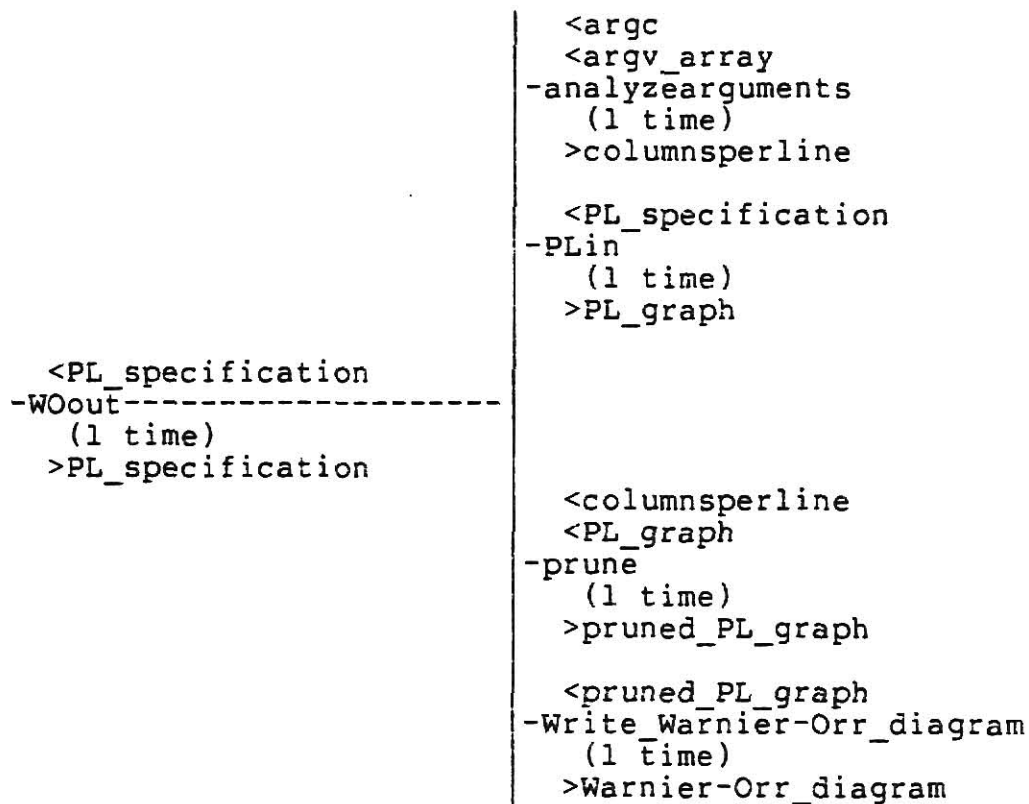


Figure 5. WOout WO-diagram

3.3 Data Structures

Although it is just a text string, the construction of the process-language specification is important since it defines the interface between all modules in the system. The Backus-Naur-Form syntax for the process-language specification is shown in Appendix A. The basic building-block out of which the process-language strings are built is the process-unit, which is strictly analogous to the ERA Activity (or the data-flow activity node). The process-unit is comprised of exactly one operator, a variable number of input arguments, and a variable number of output arguments. The colon is a reserved character; it is used to separate the operator's text from its repetition count. The comma is a reserved character that is used to separate elements in the argument lists. The left and right parentheses are reserved characters that are used to group operators and arguments. The operator and optional arguments are represented by text strings that do not contain any reserved characters. These elements are grouped to form process-units in the following manner:

```
(( <output_args> ) <operator> : <#_times> ( <input_args> ))
```

If the repetition count is equal to 1, it and the colon are omitted. If either argument list is empty, the parentheses enclosing that list are omitted. Note that both of the

above omissions are optional in that the system will correctly process repetition counts of "1" and null argument lists. These are merely abbreviations used by the system when it builds process-language strings. Process-language text output by PLorder or WOedit will never contain repetition counts of "1" or null argument lists even if the original process-language text they read contained them.

Alternation is handled by the creation of a fake Activity which has all of the alternative Activities as its subparts. This fake Activity is flagged by a special three-character sequence that immediately precedes its operator. This sequence consists of an or-bar followed by a two-digit number, e.g. "|01".

The process-units are in turn grouped into process-unit strings by an outer pair of parentheses. A process-unit string may be preceded by an operator; this indicates that this string is an expansion of the named operator. As an example of the above rules, consider the following ERA specification:

Activity : Pl	Activity : Pl.1	Activity : Pl.2
input : I	input : I	input : temp
output : O1	output : O1	output : O2
output : O2	output : temp	
subpart_is : Pl.1		Activity : Pl.3
subpart_is : Pl.2		input : temp
subpart_is : Pl.3		output : O2

ERain would translate this into the following process-

language specification:

```
((O1,O2)P1(I))  
P1((O1,temp)P1.1(I))((O2)P1.2(temp))((O2)P1.3(temp))
```

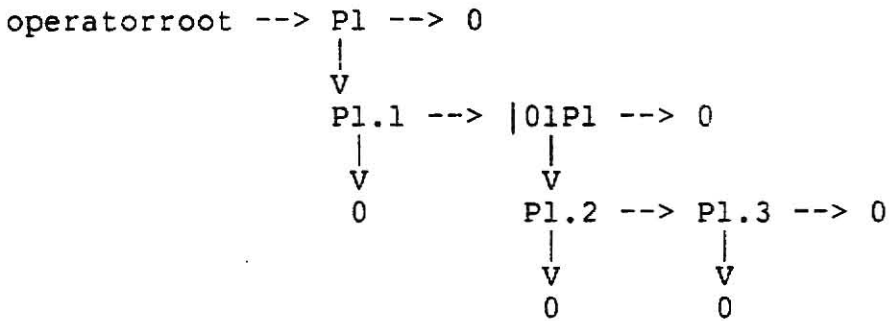
If WOedit subsequently was used to indicate that P1.2 and P1.3 were actually alternative processes (which seems reasonable from their input/output characteristics), the process-language specification would then become:

```
((O1,O2)P1(I))  
P1((O1,temp)P1.1(I))(|O1P1))  
|O1P1(((O2)P1.2:0 or 1(temp))((O2)P1.3:0 or 1(temp)))
```

The operator-argument digraph is the internal representation of the process-language specification. It is composed of "operator" and "argument" nodes. The operator-node consists of two character strings, a counter, and four pointers. The character strings are named "oprtext" and "oprtimes", and correspond to the operator and repetition fields of the process-unit. The count field is named "oproprcount" and is the count of suboperators that occur in the expansion of this operator. Two pointers, "oprnext" and "oproprroot", may point to other operator-nodes; two pointers, "oprinput" and "oproutput", may point to argument-nodes. The argument-node consists of one character string corresponding to the name of the argument it represents, and a pointer

named "argnext" that is used to chain these nodes together.

Each process-unit maps into an operator-node. The operator-nodes corresponding to the process-units in a process-unit string are chained into a list by the oprnext field. The operator-nodes corresponding to the one or more unprefixd process-unit strings are formed into another list (again chained via the oprnext field), and the first of these is pointed to by a root pointer -- called operatorroot in all of the modules in the system. Each operator-node points to the list (if any) that represents its expansion via the oproprroot field, and has the number of operator-nodes in this list noted in its oproprcount field. If the same operator occurs more than once in the design and has an expansion, these separate operator-nodes will point to the same operator-node list to represent that expansion -- thus the data structure is a digraph rather than a tree. Each argument named is mapped into an argument-node. Where multiple input or output arguments exist for a given operator, they are chained into a list via the argnext field. Each operator-node points to its associated input and output arguments via the fields oprinput and oproutput respectively. The argument-node lists, unlike the operator-node lists, are never pointed at by more than one parent operator-node. The operator-nodes corresponding to the above example would be organized as follows:



Recursion is not simulated by the operator-node digraph, since this structure was specified as having no cycles. The first recursive invocation points at another operator-node where oprtext again names the recursive routine, but the oproprroot field of this suboperator is zero. The processing module must recognize that recursion has occurred (if this is important to it) by detecting the reoccurrence of the same operatornode.oprtext in the path it is tracing. A specific pointer back to the first instance of the recursive procedure could be implemented, but this was not done at this time since there was no clear need for this facility.

4. Chapter Four

4.1 Implementation

Currently all modules function independently of each other, but they can be conveniently executed as one entity (except for WOedit) through the use of pipes. Since the input to all modules (other than ERAin) and the output of all modules

(other than WOout) is the process-language text string, the relative ordering and optional inclusion of any individual module is largely at the user's discretion. If for example the user did not want PLorder to attempt to order Activities based upon input/output dependencies, then that module could simply be left out of the "pipe-chain".

Unlike the other modules in the system, WOedit reads/writes the PL-specification from/to a text file other than stdin/stdout. This is because it must use stdin for communicating with the terminal user. This text file can be supplied as a parameter when WOedit is invoked; if it is not, the module will prompt the user to enter the file's name.

If it was necessary for efficiency purposes, the modules could be easily collapsed together since they all deal with the same internal data structure, the operator-argument digraph.

In order to ease maintenance and to facilitate addition of new functions in the future, all common definitions, data structures, and functions are located in separate text files and "#include"d at compile time. Listings of these files are included in Appendix C. It should be noted that there are strong dependencies between these files; there are multiple files just for convenience and not because each

file is logically self-contained. Inclusion of "structures" requires the inclusion of "defines", inclusion of "PLin" requires the inclusion of "defines", "structures", and "functions", etc.

An effort was made to insure that the system would be portable to other UNIX environments. All character data is coded in its graphic form so that recompilation should correct for whatever internal representation is used on a given machine (ASCII, EBCDIC, etc.). All modules were screened by the "lint" utility, and most of its complaints were resolved. Those remaining are not easily resolved and are probably not critical, such as the complaint when malloc is called to obtain storage for a structure rather than a simple character string. This warning is issued in spite of the fact that a cast is used when the returned pointer is assigned to a pointer variable, and that the number of bytes requested is determined by an appropriate "sizeof".

4.2 Testing

Testing of the system consisted of running specifications for various types of programs through the system. It was noted that the implementation was not especially robust when some types of input errors were encountered, but since the task of editing the ERA specification was explicitly assigned to another project being developed at the same time

this was not viewed as a significant failing.

The "standard" chess specification worked correctly, although the output was of limited utility due to the lack of a perceptible structure in that specification. A specification of an actual batch accounting system provided more useful output.

The hierarchy diagrams included in chapter three of this report were generated using this system. One modification was made to the original output produced by the system in the case of the diagram for WOout. This was to reverse the order of the suboperators "PLin" and "analyzearguments" to reflect their sequence in the actual code. Actually these two functions are completely independent and their order of execution is arbitrary, so the diagram produced by the system was equally correct.

5. Chapter Five

5.1 Conclusions

This project has satisfied all of its initial requirements. It reads in ERA requirements specifications and generates Warnier-Orr style design requirements based upon them. The system is written in C and runs under UNIX. The system can infer some before-after ordering rules from input-output linkages and apply them to the generated Warnier-Orr diagram

if desired. The system is highly modular, both externally (where each major function resides in a separate module) and internally (as exemplified by the common modules PLin and PLout). An interface has been provided to facilitate future enhancements of the system, namely the process-language text as a common medium of communication between functions.

Beyond satisfying the basic requirements, additional functions have been implemented through the WOedit module. This module provides for reasonable on-line viewing of the potentially large Warnier-Orr diagrams through its "window" display. It also provides a way for the user to provide iteration and alternation notations even these could not be reasonably inferred from the ERA specification.

If agreement could be reached on the exact semantics of some of the currently unprocessed keywords in the ERA specification, they might allow additional structuring of the process-language. One example of this is the entity "Periodic_function". In the sample specification that we were given as a reference point it was used to represent two very different kinds of processing; in one case it represented an activity that was triggered by an event totally asynchronous to the system (timer interrupts) while in the other case it was triggered by data flows within the system (a function often referred to as a "demon" in the literature). Asynchronous events do not seem to fit

smoothly into Warnier's methodology, but demons are data driven processors and certainly could be incorporated. Demons should either be coded as Activity entries or (if it is felt that their role is really different than that of the Activity) a new type of entry should be created.

A method for specifying more information about the conditions governing control and/or data flow would allow more structuring of the Warnier-Orr diagram by ERAin. While it may be undesirable to inject too many formal requirements into the ERA syntax, it does seem reasonable that at the requirements stage the requestor of the system should indicate if some outputs were mutually exclusive. Optional inputs and outputs to processes are so common that a method for indicating that an input is not required or that an output might not be generated should be formally incorporated into the ERA syntax. In ERA's present definition this information could admittedly be tacked onto the specification in the free-format text of "Assertions", but this does not seem satisfactory for such an important piece of information.

5.2 Possible Extensions

One easy extension would be to provide modules to translate between data-flow diagrams and the process-language text. The mapping between the two is almost trivial; the only

interesting thing here would involve the actual data-flow format. Perhaps a useful interface to a graphics device could be implemented.

A more ambitious extension would be to use a data-dictionary to infer additional (not specified in the ERA input) activities, and to insert them into the process-language stream. This offers the opportunity to begin to utilize Warnier's methodology in earnest. The procedures outlined in LCP offer some guidance in how this should be done. Each field in the output file would give rise to a process-unit. These process-units would be gathered together as an expansion of a higher-level operator reflecting a logical record or a subset thereof. After this process-unit pyramid has been expanded to encompass the entire output file, it should be possible to integrate this with the superstructure created by the ERA input file. The most obvious problem would be the question of how to deal with mismatches between the program structure derived from the data-dictionary and the program structure specified in the ERA input. One could take the attitude that such a mismatch indicated an bad ERA specification, but it is not clear that this is always the case. At the very least, such a project would be instructive in the relative merits of the different methodologies.

6. Appendix A -- Process-language BNF specification

```
<process_text> ::= <process_string>
                  | <process_text><process_string>
<process_string> ::= (<process_unit>)
                  | <operator_text>(<process_list>)
<process_list>  ::= <process_unit>
                  | <process_list><process_unit>
<process_unit>  ::= (<operator>) | (<output_args><operator>)
                  | (<operator><input_args>)
                  | (<output_args><operator><input_args>)
<input_args>    ::= (<argument_list>)
<output_args>   ::= (<argument_list>)
<argument_list> ::= <argument> | <argument>,<argument_list>
<argument>     ::= <text_char> | <argument><text_char>
<operator>      ::= <operator_text>
                  | <operator_text>:<repetition>
<repetition>    ::= <text_char> | <repetition><text_char>
<operator_text> ::= <text_char> | <operator_text><text_char>
<text_char>     ::= A | B | C | D | E | F | G | H | I | J | K
                  | L | M | N | O | P | Q | R | S | T | U | V
                  | W | X | Y | Z | a | b | c | d | e | f | g
                  | h | i | j | k | l | m | n | o | p | q | r
                  | s | t | u | v | w | x | y | z | . | ,
                  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
                  | + | - | _ | $
```

7. Appendix B -- User's Manual

7.1 ERA Syntax

Since a Backus-Naur-Form specification of the ERA syntax was generated and used by several different people engaged in various projects centering around this type of specification, it is assumed that the reader has access to this document. This appendix will merely document any non-obvious assumptions made by ERAin in its processing of the input specification.

ERAin's parser delimits input strings using the normal UNIX definition for "white space", i.e. blanks and new-line characters. Commas and colons are reserved characters and may not be embedded in any character string that will be processed by this system.

The keywords recognized by ERAin are follows:

Activity : The character string following this keyword is used as the name of the appropriate process-language operator. No repetition field is appended, so it defaults to "1 time".

input : The character string following this keyword is assumed to be an input argument of the preceding Activity.

output : The character string following this keyword is assumed to be an output argument of the preceding

Activity.

`subpart_is` : The character string following this keyword is assumed to be the name of an Activity that is defined as a sub-activity of the preceding Activity. An operator node will be built for the named sub-activity even if it is not formally named by another Activity keyword.

7.2 Executing the System

7.2.1 ERain This module has no command-line arguments. It simply reads the ERA specification from standard input and writes the process-language specification to standard output.

7.2.2 PLorder This module has no command-line arguments. It simply reads the process-language specification from standard input and writes the modified process-language specification to standard output.

7.2.3 WOedit This module facilitates the interactive editing of the process-language specification. You may specify the file name where the process language text exists as a parameter on the command line, e.g. "WOedit <filename>". If you do not, it will prompt you to enter the file name as soon as it begins to execute.

Currently implemented commands are as follows:

alternate (abbreviated "a") requires two arguments which are the names of the two nodes that are to be made alternatives to each other. If there is no second node (i.e., one procedure is optional, but there is no alternative processing when it is not used) the second argument is entered as "!". In this case, WOedit will create a dummy alternative procedure with a name that is the name of the first operator prefixed with a "!".

display (abbreviated "d") requires no arguments. It requests WOedit to display the "window" consisting of the current operator and its sub-operators (if any).

goto (abbreviated "g") requires one argument which is the name of the operator that is to be made the "current" operator. WOedit will search the operator graph below and lateral to the "current" operator, if it cannot find the named operator a diagnostic message will be issued and the first operator in the graph (the root operator) will be made the current operator. The special argument "root" may be used to return to the graph's first operator.

Quit (abbreviated "Q") requires no arguments. It requests WOedit to terminate immediately without updating the

process text file.

repeat (abbreviated "r") requires one argument which is the text string that is to be inserted in the repetition field of the current operator. A maximum of eight characters is allowed.

switch (abbreviated "s") requires two arguments which are the names of the two suboperators of the current operator that are to be interchanged.

write (abbreviated "w") requires no arguments. It requests WOedit to terminate after writing the updated process language text back to the file that it was originally read from.

7.2.4 WOout This module allows the use of run-time parameters to specify the dimensions of the Warnier-Orr diagram that it will write. The option "-c<number>" allows the user to specify the width of a column on the output diagram. The option "-l<number>" allows the user to specify the width of the page the output diagram will be printed on. Column-width defaults to 32 and page-width defaults to 132.

7.2.5 Process-chaining Since standard input and standard output are used exclusively for all modules except for WOedit, it will often make sense for the user to connect these processes via pipes. The following command structure

is one that the author has found useful in testing:

```
cat <era.file(s)> | ERAin | PLorder | tee <pl.file> | WOout
```

This produces (on standard output) a listing of the Warnier-Orr structure obtained from the input ERA specification(s) and leaves a copy of the process-language text in <output.file> where it may be subsequently edited via WOedit if necessary.

8. Appendix C -- Source Code

defines

```
#define ARGUMENTSTRING 32
#define FALSE 0
#define MAXSTRING 32
#define OPERATORSTRING 32
#define OPRTIMESSTRING 8
#define TRUE 1
```

structures

```
struct operatornode {
    struct operatornode *oprnext;
    struct operatornode *oproprroot;
    struct argumentnode *oproutput;
    struct argumentnode *oprinput;
    short oprprcount;
    unsigned POutflag : 1;
    unsigned WOutflag : 1;
    unsigned waspruned : 1;
    unsigned isbranch : 1;
    char oprtext[OPERATORSTRING+1];
    char oprtimes[OPRTIMESSTRING+1];
};
struct argumentnode {
    struct argumentnode *argnext;
    char argtext[ARGUMENTSTRING+1];
};
```


functions

```
struct operatornode *buildoperatornode ()
{
    struct operatornode *newopr;
    newopr = (struct operatornode *) malloc(sizeof(struct operatornode));
    newopr->oprnext = newopr->oproprroot = 0;
    newopr->oproutput = newopr->oprinput = 0;
    newopr->PLoutflag = newopr->WDoutflag = newopr->waspruned = newopr->isbranch = FALSE;
    newopr->oprtimes[0] = '1';
    newopr->oprtimes[1] = newopr->oprtext[0] = '\\0';
    return(newopr);
}

deleteoperatornode (root, victim) struct operatornode **root, **victim;
{
    if (*root == *victim) {
        *root = (*victim)->oprnext;
        free(*victim);
        *victim = 0;
    }
    else {
        struct operatornode *scanpointer = *root;
        while (scanpointer->oprnext != *victim) scanpointer = scanpointer->oprnext;
        scanpointer->oprnext = (*victim)->oprnext;
        free(*victim);
        *victim = scanpointer;
    }
}

mergeoperator (candidate, thisoperator) struct operatornode *candidate, *thisoperator;
{
    int mergecount = 0;
    do {
        if (candidate != thisoperator && candidate->oproprroot != thisoperator)
            if (thisoperator->oproprroot)
                mergecount += mergeoperator(candidate, thisoperator->oproprroot);
            else if (!strcmp(candidate->oprtext, thisoperator->oprtext)) {
                thisoperator->oproprroot = candidate->oproprroot;
                thisoperator->oproprcount = candidate->oproprcount;
                if (!thisoperator->oprinput)
                    thisoperator->oprinput = candidate->oprinput;
                if (!thisoperator->oproutput)
                    thisoperator->oproutput = candidate->oproutput;
                mergecount++;
            }
        } while (thisoperator = thisoperator->oprnext);
    return(mergecount);
}

strcpy (s1, s2) /* string copy function */
char *s1, *s2; /* copies s2 into s1 */
{ while (*s1++ = *s2++); }

strcmp (s1, s2) /* string difference function */
char *s1, *s2; /* returns <0 if s1<s2, 0 if s1==s2, >0 if s1>s2 */
{
    for (; *s1 == *s2; s1++, s2++)
        if (*s1 == '\\0') return(0);
    return(*s1 - *s2);
}
```

PLin

```
struct operatornode *PLin (file) FILE *file;
{
    /* written by Donald E. Wolfe; June 1984 */
    struct operatornode *thisoperator=0, *lastmainopr=0, *lastsubopr=0, *operatorroot=0;
    int c = 0, mergecount;
    while (c != EOF) {
        int i = 0, level = 0;
        char string[MAXSTRING+1];
        while (c != EOF && level == 0) {
            switch (c = getc(file)) {
                case '(':
                    string[i] = '\0';
                    thisoperator = buildoperatornode();
                    if (operatorroot) lastmainopr = lastmainopr->oprnext = thisoperator;
                    else lastmainopr = operatorroot = thisoperator;
                    strngcpy(thisoperator->oprtext, string);
                    level = 1;
                case 'O':
                    i = 0;
                case ' ':
                    string[i] = ' ';
                    break;
                default:
                    if (i < MAXSTRING) string[i++] = c;
            }
        }
        while (c != EOF && level) {
            switch (c = getc(file)) {
                case '(':
                    level++;
                    string[i] = '\0';
                    if (level == 2) {
                        thisoperator = buildoperatornode();
                        if (lastmainopr->opprroot) lastsubopr->oprnext = thisoperator;
                        else lastmainopr->opprroot = thisoperator;
                        lastsubopr = thisoperator;
                        lastmainopr->opprcount++;
                    }
                    else if (string[0] == '(') strngcpy(thisoperator->oprtext, string+1);
                    else if (string[0] != '(') strngcpy(thisoperator->oprtext, string);
                    i = 0;
                    string[0] = '\0';
                    break;
                case ')':
                    level--;
                case ',':
                    string[i] = '\0';
                    if (string[0] != '\0')
                        if (level >= 2) { struct argumentnode *newarg;
                            newarg=(struct argumentnode *) malloc(sizeof(struct argumentnode));
                            newarg->argnext = 0;
                            strngcpy(newarg->argtext, string);
                            if (thisoperator->oprtext[0] == '\0')
                                if (thisoperator->oproutput) currentarg->argnext = newarg;
                                else thisoperator->oproutput = newarg;
                            else if (thisoperator->oprinput) currentarg->argnext = newarg;
                            else thisoperator->oprinput = newarg;
                            currentarg = newarg;
                        }
                    else if (string[0] == '(') strngcpy(thisoperator->oprtext, string+1);
                    else strngcpy(thisoperator->oprtext, string);
                    i = 0;
                    string[0] = '\0';
                    break;
                case ':':
                    string[i] = '\0';
                    strngcpy(thisoperator->oprtext, string);
                    i = 0;
                default:
                    string[i++] = c;
            }
        }
    }
}
```

```
    }  
  }  
}  
for (thisoperator=operatorroot; thisoperator; thisoperator=thisoperator->oprnext)  
  if (thisoperator->oprtext[0] == '\\0') {  
    strngcpy(thisoperator->oprtext, thisoperator->oproprroot->oprtext);  
    strngcpy(thisoperator->oprtimes, thisoperator->oproprroot->oprtimes);  
    thisoperator->oproutput = thisoperator->oproprroot->oproutput;  
    thisoperator->oprinput = thisoperator->oproprroot->oprinput;  
    thisoperator->oproprcount = thisoperator->oproprroot->oproprcount;  
    lastsubopr = thisoperator->oproprroot;  
    thisoperator->oproprroot = thisoperator->oproprroot->oproprroot;  
    free(lastsubopr);  
  }  
do {  
  mergecount = 0;  
  for (thisoperator=operatorroot; thisoperator;  
       thisoperator=thisoperator?thisoperator->oprnext:operatorroot) {  
    mergecount += c = mergeoperators(thisoperator, operatorroot);  
    if (c) deleteoperatornode(&operatorroot, &thisoperator);  
  }  
} while (mergecount);  
return(operatorroot);  
}
```

PLout

```
PLout (file,operatorroot,recursivecall)
FILE *file; struct operatornode *operatorroot; int recursivecall;
{
    /* written by Donald E. Wolfe; June 1984 */
    struct operatornode *thisoperator;
    struct argumentnode *argument;
    do {
        if (!recursivecall) {
            fprintf(file,"\n(");
            if (argument = operatorroot->oproutput) {
                fprintf(file,"%s",argument->argtext);
                while (argument=argument->argnext)
                    fprintf(file,"%s",argument->argtext);
                fprintf(file,")");
            }
            fprintf(file,"%s",operatorroot->oprtext);
            if (operatorroot->oprtimes[0] != '1')
                fprintf(file,"%s",operatorroot->oprtimes);
            if (argument = operatorroot->oprinput) {
                fprintf(file,"%s",argument->argtext);
                while (argument=argument->argnext)
                    fprintf(file,"%s",argument->argtext);
                fprintf(file,")");
            }
            fprintf(file,")");
        }
        if ((thisoperator=operatorroot->oprprroot) && !thisoperator->PLoutflag) {
            fprintf(file,"\n %s(",operatorroot->oprtext);
            do {
                fprintf(file,"(");
                if (argument = thisoperator->oproutput) {
                    fprintf(file,"%s",argument->argtext);
                    while (argument=argument->argnext)
                        fprintf(file,"%s",argument->argtext);
                    fprintf(file,")");
                }
                fprintf(file,"%s",thisoperator->oprtext);
                if (thisoperator->oprtimes[0] != '1')
                    fprintf(file,"%s",thisoperator->oprtimes);
                thisoperator->PLoutflag = TRUE;
                if (argument = thisoperator->oprinput) {
                    fprintf(file,"%s",argument->argtext);
                    while (argument=argument->argnext)
                        fprintf(file,"%s",argument->argtext);
                    fprintf(file,")");
                }
                fprintf(file,")");
            } while (thisoperator = thisoperator->oprnext);
            fprintf(file,")");
            PLout(file,operatorroot->oprprroot,recursivecall+1);
        }
    } while (operatorroot = operatorroot->oprnext);
}
```

ERain

```
#include "stdio.h"

#include "defines"
#include "structures"
#include "functions"
#include "PLOut"

addargument (currentoperator, ioflag) struct operatornode *currentoperator; char ioflag;
{
    struct argumentnode *newarg, *temppointer;
    newarg = (struct argumentnode *) malloc(sizeof(struct argumentnode));
    newarg->argnext = 0;
    getstring(newarg->argtext, ARGUMENTSTRING);
    if (ioflag == 'i') if (temppointer = currentoperator->oprinput) {
        while (temppointer->argnext != 0) temppointer = temppointer->argnext;
        temppointer->argnext = newarg;
    }
    else currentoperator->oprinput = newarg;
    else if (temppointer = currentoperator->oproutput) {
        while (temppointer->argnext != 0) temppointer = temppointer->argnext;
        temppointer->argnext = newarg;
    }
    else currentoperator->oproutput = newarg;
}

getstring (string, limit) char string[]; int limit;
{
    int c, i = 0;
    while ((c = getchar()) != ' ');
    while (1) {
        switch(c) {
            case ' ':
            case '\0':
                string[i] = '\0';
                return;
            default:
                if (i < limit) string[i++] = c;
        }
        c = getchar();
    }
}

main () {
    /* written by Donald E. Wolfe; June 1984 */
    struct operatornode *currentoperator=0, *currentsubopr=0, *operatorroot=0;
    int c = 0, chainsmerged, i = 0;
    static char Activity[] = "Activity", input[] = "input",
        output[] = "output", subpart_is[] = "subpart_is";
    char string[MAXSTRING+1];
    for (i=0; i<MAXSTRING; i++) string[i] = 'x';
    string[MAXSTRING+1] = '\0';
    i = 0;
```

```
while (c != EOF) {
    c = getchar();
    switch (c) {
        case ':':
            string[i] = '\0';
            if (!strngdif(string,Activity)) {
                if (!currentoperator) currentoperator = operatorroot = buildoperator();
                else currentoperator = currentoperator->oprnext = buildoperator();
                getstring(currentoperator->oprtext,OPERATORSTRING);
            }
            if (!strngdif(string,input)) addargument(currentoperator,'i');
            if (!strngdif(string,output)) addargument(currentoperator,'o');
            if (!strngdif(string,subpart_is)) {
                if (currentoperator->oproprroot)
                    currentsubopr = currentsubopr->oprnext = buildoperatornode();
                else currentsubopr = currentoperator->oproprroot = buildoperatornode();
                getstring(currentsubopr->oprtext,OPERATORSTRING);
            }
        case 'O':
            string[i] = 'x';
            i=0;
        case ' ':
            string[i] = '\0';
            break;
        default:
            if (i && (string[i] == '\0')) {
                string[i] = 'x';
                i = 0;
            }
            if (i<MAXSTRING) string[i++] = c;
    }
}
do {
    chainsmerged = 0;
    for (currentoperator=operatorroot; currentoperator;
        currentoperator=currentoperator?currentoperator->oprnext:operatorroot)
    {
        chainsmerged += i = mergeoperator(currentoperator,operatorroot);
        if (i) deleteoperatornode(&operatorroot,&currentoperator);
    }
} while (chainsmerged);
PLout(stdout,operatorroot,0);
}
```

PLorder

```
#include "stdio.h"

#include "defines"
#include "structures"
#include "functions"
#include "PLin"
#include "PLOut"

orderoperators (root) struct operatornode *root;
{
    struct operatornode *oprinroot=0, *oprinlast=0, *oproutroot=0, *oproutlast=0;
    struct operatornode *thisoperator, *oprscanner;
    struct argumentnode *argument, *argscanner;
    for (thisoperator=root; thisoperator; thisoperator=thisoperator->oprnext) {
        int nolinkage = TRUE;
        if (thisoperator->oproprroot) orderoperators(thisoperator->oproprroot);
        for (argument=thisoperator->oprinput; nolinkage && argument;
            argument=argument->argnext)
            for (oprscanner=root; nolinkage && oprscanner; oprscanner=oprscanner->oprnext)
                if (oprscanner != thisoperator)
                    for (argscanner=oprscanner->oproutput; nolinkage && argscanner;
                        argscanner=argscanner->argnext)
                        if (!strcmp(argargument->argtext,argscanner->argtext))
                            nolinkage=FALSE;
        if (nolinkage) /* then move this operator to the input list */
            if (oprinlast) oprinlast = oprinlast->oprnext = thisoperator;
            else oprinlast = oprinroot = thisoperator;
        else if (oproutlast) oproutlast = oproutlast->oprnext = thisoperator;
            else oproutlast = oproutroot = thisoperator;
    }
    if (oproutlast) oproutlast = oproutlast->oprnext = 0;
    for (thisoperator=oproutroot, oproutroot=0; thisoperator;
        thisoperator=thisoperator->oprnext) { int nolinkage = TRUE;
        for (argument=thisoperator->oproutput; nolinkage && argument;
            argument=argument->argnext)
            for (oprscanner=oproutroot; nolinkage && oprscanner;
                oprscanner=oprscanner->oprnext)
                if (oprscanner != thisoperator)
                    for (argscanner=oprscanner->oprinput; nolinkage && argscanner;
                        argscanner=argscanner->argnext)
                        if (!strcmp(argargument->argtext,argscanner->argtext))
                            nolinkage=FALSE;
        if (nolinkage)
            if (oproutlast) oproutlast = oproutlast->oprnext = thisoperator;
            else oproutlast = oproutroot = thisoperator;
        else if (oprinlast) oprinlast = oprinlast->oprnext = thisoperator;
            else oprinlast = oprinroot = thisoperator;
    }
    if (oproutlast) oproutlast->oprnext = 0;
    if (oprinlast) oprinlast->oprnext = oproutroot;
    else oprinroot = oproutroot;
    /* oprinroot now points to the head of a unified (possibly reordered) list. */
}
```

```
/* because we dare not move the physical location of the first item in the */
/* list, if oprinroot != root we must swap the contents of the nodes. */
if (oprinroot != root) {
    struct operatornode temp;
    temp.opproprrroot = root->opproprrroot;
    root->opproprrroot = oprinroot->opproprrroot;
    oprinroot->opproprrroot = temp.opproprrroot;
    temp.oproutput = root->oproutput;
    root->oproutput = oprinroot->oproutput;
    oprinroot->oproutput = temp.oproutput;
    temp.oprininput = root->oprininput;
    root->oprininput = oprinroot->oprininput;
    oprinroot->oprininput = temp.oprininput;
    temp.opproprrcount = root->opproprrcount;
    root->opproprrcount = oprinroot->opproprrcount;
    oprinroot->opproprrcount = temp.opproprrcount;
    strngcpy(temp.oprttext,root->oprttext);
    strngcpy(root->oprttext,oprinroot->oprttext);
    strngcpy(oprinroot->oprttext,temp.oprttext);
    strngcpy(temp.oprtimes,root->oprtimes);
    strngcpy(root->oprtimes,oprinroot->oprtimes);
    strngcpy(oprinroot->oprtimes,temp.oprtimes);
    for (thisoperator=oprinroot; thisoperator->oprnext != root;
        thisoperator=thisoperator->oprnext) ;
    if (thisoperator == oprinroot) { /* oprinroot points at root */
        thisoperator->oprnext = root->oprnext;
        root->oprnext = thisoperator;
    }
    else {
        thisoperator->oprnext = oprinroot;
        temp.oprnext = root->oprnext;
        root->oprnext = oprinroot->oprnext;
        oprinroot->oprnext = temp.oprnext;
    }
}
}

main () {
    /* written by Donald E. Wolfe; June 1984 */
    struct operatornode *operatorroot;
    operatorroot = PLin(stdin);
    orderoperators(operatorroot);
    PLout(stdout,operatorroot,0);
}
```


WOedit

```
#include "stdio.h"

#include "defines"
#include "structures"
#include "functions"
#include "PLin"
#include "PLout"

isdigit(c) char c;
{
    if (c == '0' || c == '1' || c == '2' || c == '3' || c == '4' || c == '5' ||
        c == '6' || c == '7' || c == '8' || c == '9') return(TRUE);
    else return(FALSE);
}

static char formatopr[] = "%s-%-31.31s%c", formatarg[] = "%s %c%-29.29s%c";

struct operatornode *findoperator(operator,name) struct operatornode *operator;
char name[];
{
    struct operatornode *temppointer = 0;
    do {
        if (!strngdif(operator->oprtext,name)) return(operator);
        else if (operator->oproprroot
            && (temppointer=findoperator(operator->oproprroot,name))) return(temppointer);
        } while (operator=operator->oprnext);
    return(0);
}

printsuboperators(operator,momsname,skipcnt,printcnt) struct operatornode *operator;
char *momsname; int skipcnt, printcnt;
{
    static char leadstring[] = " |";
    char blank = ' ', outflag = '>', inflag = '<', nullchar = '\\0', orbar = '|';
    char trailingchar, workstring[MAXSTRING+1];
    while (skipcnt--) operator = operator->oprnext;
    while (printcnt--) {
        struct argumentnode *argument;
        if (!strngdif(operator->oprtext,momsname))
            printf(formatopr,leadstring,"<<<recursion<<<",blank);
        else {
            if (operator->oproprroot) trailingchar = orbar;
            else trailingchar = blank;
            for (argument=operator->oprinput; argument; argument=argument->argnext)
                printf(formatarg,leadstring,inflag,argument->argtext,trailingchar);
            if (operator->oproprroot) {
                int i;
                if (operator->oprtext[0] == '|') {
                    for (i=1; i<= operator->oproprcount; i++) {
                        printsuboperators(operator->oproprroot,operator->oprtext,i-1,1);
                        if (i<operator->oproprcount) {
                            printf(formatarg,leadstring,blank," ",blank);
                            printf(formatarg,leadstring,blank,"(+)",blank);
                            printf(formatarg,leadstring,blank," ",blank);
                        }
                    }
                }
                else {
                    for (i=0; (workstring[i] = operator->oprtext[i]) != ' '; i++) ;
                    while (i<OPERATORSTRING) workstring[i++] = '-';
                    workstring[i] = '\\0';
                    printf(formatopr,leadstring,workstring,trailingchar);
                }
            }
            else printf(formatopr,leadstring,operator->oprtext,trailingchar);
            if (operator->oprtext[0] != '|')
                if (operator->oprtext[0] == '1')
                    printf(formatarg,leadstring,blank,"(1 time)",trailingchar);
                else {
                    sprintf(workstring,"%s times",operator->oprtext);
                    printf(formatarg,leadstring,blank,workstring,trailingchar);
                }
        }
    }
}
```

```
        for (argument=operator->oproutput; argument; argument=argument->argnext)
            printf(formatarg,leadstring,outflag,argument->argtext,trailingchar);
    }
    if (printcnt) printf(formatarg,leadstring,blank,&nullchar,blank);
    operator = operator->oprnext;
}

strngcat(s1,s2,s3,smax1) char *s1, *s2, *s3; int smax1;
{
    int i = 0, j = 0;
    while ((s1[i]=s2[i]) != ' ' && i < smax1) i++;
    while ((s1[i]=s3[j]) != ' ' && i < smax1) i++, j++;
    s1[smax1] = '\0';
}

updatePL (root) struct operatornode *root;
{
    struct operatornode *currentopr = root;
    short done = FALSE;
    char arg1[MAXSTRING+1], arg2[MAXSTRING+1], command[MAXSTRING];
    while (!done) { char commandchar = '9';
        printf("\nready:");
        scanf("%s",command);
        {
            if (!strngdif(command,"alternate") || !strngdif(command,"a"))
                commandchar = 'A', scanf("%s %s",arg1,arg2);
            if (!strngdif(command,"display") || !strngdif(command,"d"))
                commandchar = 'D';
            if (!strngdif(command,"goto") || !strngdif(command,"g"))
                commandchar = 'G', scanf("%s",arg1);
            if (!strngdif(command,"repeat") || !strngdif(command,"r"))
                commandchar = 'R', scanf("%s",arg1);
            if (!strngdif(command,"switch") || !strngdif(command,"s"))
                commandchar = 'S', scanf("%s %s",arg1,arg2);
            if (!strngdif(command,"Quit") || !strngdif(command,"Q")) commandchar = 'Q';
            if (!strngdif(command,"write") || !strngdif(command,"w")) commandchar = 'W';
        }
        switch(commandchar) {
```

```
case('A'):
{
    struct operatornode *altptr[2], *tempptr;
    int i, altnumnext = 0, argfound = 0;
    if (!currentopr->oproprroot) {
        fprintf(stderr, "\nno suboperators present; command ignored");
        break;
    }
    for (tempptr=currentopr->oproprroot; tempptr; tempptr=tempptr->oprnext)
        if (tempptr->oprtext[0] == '|' && isdigit(tempptr->oprtext[1])
            && isdigit(tempptr->oprtext[2])) {
            sscanf(&tempptr->oprtext[1], "%2d", &i);
            altnumnext = altnumnext > i ? altnumnext : i;
        }
    for (tempptr=currentopr->oproprroot; tempptr; tempptr=tempptr->oprnext)
        if (!strngdif(tempptr->oprtext, arg1)) {
            altptr[argfound++] = tempptr;
            arg1[0] = '\0';
        }
        else if (!strngdif(tempptr->oprtext, arg2)) {
            altptr[argfound++] = tempptr;
            arg2[0] = '\0';
        }
    if (argfound < 2) if (argfound==1 && !strngdif(arg2, "!")) {
        altptr[1] = buildoperatornode();
        strngcat(altptr[1]->oprtext, "!", altptr[0]->oprtext, OPERATORSTRING);
    }
    else {
        fprintf(stderr, "\ncant find %s %s request ignored", arg1, arg2);
        break;
    }
    tempptr = buildoperatornode();
    tempptr->oprnext = altptr[1];
    tempptr->oproprroot = altptr[0]->oproprroot;
    tempptr->oproutput = altptr[0]->oproutput;
    tempptr->oprinput = altptr[0]->oprinput;
    tempptr->oproprcount = altptr[0]->oproprcount;
    strngcpy(tempptr->oprtext, altptr[0]->oprtext);
    strngcpy(tempptr->oprtext, "0 or 1");
    altptr[0]->oproprroot = tempptr;
    altptr[0]->oproutput = altptr[0]->oprinput = 0;
    altptr[0]->oproprcount = 2;
    sprintf(altptr[0]->oprtext, "%02d", ++altnumnext);
    strngcat(altptr[0]->oprtext, altptr[0]->oprtext,
        currentopr->oprtext, OPERATORSTRING);
    altptr[0]->oprtext[0] = '1';
    strngcpy(altptr[1]->oprtext, "0 or 1");
    if (strngdif(arg2, "!")) {
        for (tempptr=altptr[0]; tempptr->oprnext!=altptr[1];
            tempptr=tempptr->oprnext) ;
        tempptr->oprnext = altptr[1]->oprnext;
        altptr[1]->oprnext = 0;
        currentopr->oproprcount--;
    }
}
break;
```

```
case('D'):
{
    struct argumentnode *argument;
    int i, printcnt = (currentopr->oproprcount + 1) / 2;
    char blank = ' ', outflag = '>', inflag = '<', orbar = '|', trailingchar;
    char leadstring[2], workstring[MAXSTRING+1];
    sprintf(leadstring, "\n");
    if (currentopr->oproprroot) {
        printsuboperators(currentopr->oproprroot, currentopr->oprtext, 0, printcnt);
        trailingchar = orbar;
    }
    else trailingchar = blank;
    for (argument=currentopr->oprinput; argument; argument=argument->argnext)
        printf(formatarg, leadstring, inflag, argument->argtext, trailingchar);
    if (currentopr->oproprroot) {
        for (i=0; (workstring[i] = currentopr->oprtext[i]) != '\0'; i++);
        while (i<OPERATORSTRING) workstring[i++] = '-';
        workstring[i] = '\0';
        printf(formatopr, leadstring, workstring, trailingchar);
    }
    else printf(formatopr, leadstring, currentopr->oprtext, trailingchar);
    if (currentopr->oprtimes[0] == '1')
        printf(formatarg, leadstring, blank, "(1 time)", trailingchar);
    else {
        sprintf(workstring, "(%s times)", currentopr->oprtimes);
        printf(formatarg, leadstring, blank, workstring, trailingchar);
    }
    for (argument=currentopr->oproutput; argument; argument=argument->argnext)
        printf(formatarg, leadstring, outflag, argument->argtext, trailingchar);
    if (currentopr->oproprroot)
        printsuboperators(currentopr->oproprroot, currentopr->oprtext, printcnt,
            currentopr->oproprcount-printcnt);
}
break;
case('G'):
    if (!strngdif(arg1, "root")) currentopr = root;
    else if (!(currentopr = findoperator(currentopr, arg1))) {
        fprintf(stderr, "\nunable to find %s; currentopr is set to %s", arg1,
            root->oprtext);
        currentopr = root;
    }
break;
case('Q'):
    return(0) /* dont write PL text */;
case('R'):
    arg1[OPRTIMESSTRING] = '\0' /* a little safety device */;
    strcpy(currentopr->oprtimes, arg1);
    printf("\n%s iteration count is set to %s", currentopr->oprtext,
        currentopr->oprtimes);
break;
```

```
case('S'):
{
    struct operatornode *switchptr[2], *tempptr, temp;
    int argfound = 0;
    if (!currentopr->oproprroot) {
        fprintf(stderr, "\nno suboperators present; command ignored");
        break;
    }
    for (tempptr=currentopr->oproprroot; tempptr; tempptr=tempptr->oprnext)
        if (!strngdif(tempptr->oprtext, arg1)) {
            switchptr[argfound++] = tempptr;
            arg1[0] = ' ';
        }
        else if (!strngdif(tempptr->oprtext, arg2)) {
            switchptr[argfound++] = tempptr;
            arg2[0] = ' ';
        }
    if (argfound < 2) {
        fprintf(stderr, "\ncant find %s %s request ignored", arg1, arg2);
        break;
    }
    temp.opoprroot = switchptr[0]->oproprroot;
    switchptr[0]->oproprroot = switchptr[1]->oproprroot;
    switchptr[1]->oproprroot = temp.opoprroot;
    temp.oproutput = switchptr[0]->oproutput;
    switchptr[0]->oproutput = switchptr[1]->oproutput;
    switchptr[1]->oproutput = temp.oproutput;
    temp.oprinput = switchptr[0]->oprinput;
    switchptr[0]->oprinput = switchptr[1]->oprinput;
    switchptr[1]->oprinput = temp.oprinput;
    temp.opoprcount = switchptr[0]->opoprcount;
    switchptr[0]->opoprcount = switchptr[1]->opoprcount;
    switchptr[1]->opoprcount = temp.opoprcount;
    strngcpy(temp.oprtext, switchptr[0]->oprtext);
    strngcpy(switchptr[0]->oprtext, switchptr[1]->oprtext);
    strngcpy(switchptr[1]->oprtext, temp.oprtext);
    strngcpy(temp.oprtimes, switchptr[0]->oprtimes);
    strngcpy(switchptr[0]->oprtimes, switchptr[1]->oprtimes);
    strngcpy(switchptr[1]->oprtimes, temp.oprtimes);
    break;
}
case('W'):
    done = TRUE;
    break;
default:
    fprintf(stderr, "\nunknown command (%s) ignored", command);
}
}
return(1) /* write PL text */;
}
```

```
main (argc,argv) int argc;
char *argv[];
{
    /* written by Donald E. Wolfe; July 1984 */
    struct operatornode *operatorroot;
    FILE *fopen(), *file;
    char filename[15];
    if (--argc) sscanf(*(argv+1),"%s",filename);
    else {
        printf("\n%s: enter the name of the PL file that you wish to edit:",*argv);
        scanf("%s",filename);
    }
    if ((file=fopen(filename,"r")) == NULL) {
        fprintf(stderr,"\n%s: cant read %s",*argv,filename);
        exit(1);
    }
    else {
        operatorroot = PLin(file);
        fclose(file);
    }
    if (updatePL(operatorroot))
        if ((file=fopen(filename,"w"))==NULL) {
            fprintf(stderr,"\n%s: cant write to %s",*argv,filename);
            exit(2);
        }
        else {
            PLout(file,operatorroot,0);
            fclose(file);
        }
}
```

WOout

```
#include "stdio.h"

#include "defines"
#include "structures"
#include "functions"
#include "PLin"

static int columnsperline = 4;
static char formatleadstring[] = "%s%32c%c%c":
static char formatopr[] = "%s-%31.31s%c", formatarg[] = "%s %c%-29.29s%c";

printoperatorlist (operator, momsname, leadstring, skipcnt, printcnt)
struct operatornode *operator; char *momsname, *leadstring; int skipcnt, printcnt;
{
    char blank = ' ', outflag = '>', inflag = '<', nullchar = '\0', orbar = '|';
    char trailingchar, newleadstring[128], workstring[MAXSTRING+1];
    while (skipcnt--) operator = operator->oprnext;
    while (printcnt--) { int i, suboprprintcnt = (operator->oproprcount + 1) / 2;
        if (!strngdif(operator->oprtext, momsname))
            printf(formatopr, leadstring, "<<<recursion<<<", blank);
        else { struct argumentnode *argument;
            if (operator->oproprroot && operator->oprtext[0] != '|') {
                sprintf(newleadstring, formatleadstring, leadstring, blank, orbar, nullchar);
                printoperatorlist(operator->oproprroot,
                    operator->oprtext, newleadstring, 0, suboprprintcnt);
                trailingchar = orbar;
            }
            else trailingchar = blank;
            for (argument=operator->oprinput; argument; argument=argument->argnext)
                printf(formatarg, leadstring, inflag, argument->argtext, trailingchar);
            if (operator->oproprroot && operator->oprtext[0] != '|')
                for (i=1; i<=operator->oproprcount; i++) {
                    printoperatorlist(operator->oproprroot,
                        operator->oprtext, leadstring, i-1, 1);
                    if (i<operator->oproprcount) {
                        printf(formatarg, leadstring, blank, " ", blank);
                        printf(formatarg, leadstring, blank, "(+)", blank);
                        printf(formatarg, leadstring, blank, " ", blank);
                    }
                }
            else {
                if (operator->oproprroot || operator->waspruned)
                {
                    for (i=0; (workstring[i] = operator->oprtext[i]) != '\0'; i++) ;
                    while (i<OPERATORSTRING) workstring[i++] = '-';
                    workstring[i] = nullchar;
                    printf(formatopr, leadstring, workstring, trailingchar);
                    if (operator->waspruned)
                        printf(formatarg, leadstring, '-', "(to be continued)-", trailingchar);
                }
                else printf(formatopr, leadstring, operator->oprtext, trailingchar);
                sprintf(workstring, strngdif(operator->oprtext, "1")?
                    "({s times)": "{s time)", operator->oprtext);
                printf(formatarg, leadstring, blank, workstring, trailingchar);
            }
            for (argument=operator->oproutput; argument; argument=argument->argnext)
                printf(formatarg, leadstring, outflag, argument->argtext, trailingchar);
            if (operator->oproprroot && operator->oprtext[0] != '|')
                printoperatorlist(operator->oproprroot, operator->oprtext, newleadstring,
                    suboprprintcnt, operator->oproprcount-suboprprintcnt);
        }
        if (printcnt) printf(formatarg, leadstring, blank, &nullchar, blank);
        operator = operator->oprnext;
    }
}
```

```
prune (ancestor,predecessor,operator,level)
struct operatornode *ancestor, **predecessor, *operator; int level;
{
    do {
        if (operator->oproprroot)
            if (level == columnssperline) {
                struct operatornode *newpredecessor, *newoperator;
                newpredecessor = buildoperatornode();
                newoperator = newpredecessor->oproprroot = buildoperatornode();
                newpredecessor->oproprcount = operator->oproprcount;
                newpredecessor->oproutput = operator->oproutput;
                newpredecessor->oprinput = operator->oprinput;
                strcpy(newoperator->oprtext,ancestor->oprtext);
                strcpy(newpredecessor->oprtext,operator->oprtext);
                newpredecessor->isbranch = TRUE;
                newpredecessor->oprnext = (*predecessor)->oprnext;
                *predecessor = (*predecessor)->oprnext = newpredecessor;
                newoperator->oprnext = operator->oproprroot;
                operator->oproprroot = 0;
                operator->waspruned = TRUE;
            }
            else prune(ancestor,predecessor,operator->oproprroot,
                operator->oprtext[0]=='|'?level:level+1);
        } while (operator = operator->oprnext);
    }

main (argc,argv) int argc; char *argv[];
{
    /* written by Donald E. Wolfe; June 1984 */
    struct operatornode *currentoperator, *operatorroot;
    int colwidth = 32, linewidth = 132;
    while (--argc && ***++argv == '-') {
        char workstring[3], *parmpointer = *argv;
        if (sscanf(parmpointer,"%c%d",&colwidth)) {
            sprintf(workstring,"%02d",colwidth);
            formatleadstring[3] = workstring[0];
            formatleadstring[4] = workstring[1];
            sprintf(workstring,"%02d",--colwidth);
            formatopr[5] = formatopr[8] = workstring[0];
            formatopr[6] = formatopr[9] = workstring[1];
            --colwidth;
            sprintf(workstring,"%02d",--colwidth);
            formatarg[8] = formatarg[11] = workstring[0];
            formatarg[9] = formatarg[12] = workstring[1];
        }
        if (sscanf(parmpointer,"%-1d",&linewidth)) ;
        columnssperline = linewidth/colwidth;
    }
    operatorroot = PLin(stdin);
    /* To continue chains truncated by insufficient linewidth we are now going to */
    /* fudge the PLtree. The last operator that fit on a line is marked "waspruned". */
    /* A new operator is built immediately after the current one and marked "isbranch". */
    for (currentoperator=operatorroot; currentoperator;
        currentoperator=currentoperator->oprnext) {
        struct operatornode *workpointer = currentoperator /* modified by prune */;
        if (currentoperator->oproprroot)
            prune(currentoperator,&workpointer,currentoperator->oproprroot,2);
    }
    for (currentoperator=operatorroot; currentoperator;
        currentoperator=currentoperator->oprnext) {
        char leadstring[3];
        sprintf(leadstring,"\n");
        if (currentoperator->isbranch) {
            printf(formatarg,"\nDiagram continuation from:","' ',
                currentoperator->oproprroot->oprtext,':');
            currentoperator->oproprroot = currentoperator->oproprroot->oprnext;
        }
        printoperatorlist(currentoperator,leadstring,leadstring,0,1);
        printf("\n");
    }
}
```


References

1. Barr, Avron and Edward A. Feigenbaum, **The Handbook of Artificial Intelligence**, vol. 2, William Kaufmann, Inc., Los Altos, CA, pp.296-380
2. Howden, William E., "Contemporary Software Development Environments", **Communications of the ACM**, vol. 25, no. 5, May 1982, pp. 318-329
3. Manna, Zohar and Richard Waldinger, "Knowledge and Reasoning in Program Synthesis", **Studies in Automatic Programming Logic**, North-Holland Publishing Company, Amsterdam, 1977, pp.141-180
4. Partsch, H. and R. Steinbruggen, "Program Transformation Systems", **ACM Computing Surveys**, September, 1983, pp.199-236
5. Hammer, Michael and Gregory Ruth, "Automating the Software Development Process", **Research Directions in Software Technology** (edited by Peter Wegner), The MIT Press, Cambridge, MA, 1979, pp.767-790
6. Teichroew, Daniel et al , "Application of the Entity-Relationship Approach to Information Processing Systems Modeling", **Entity-Relationship Approach to Systems Analysis and Design**, (edited by Peter P. Chen), North-Holland Publishing Company, Amsterdam, 1980, pp. 15-38
7. Solvberg, Arne, "A Contribution to the Definition of Concepts for Expressing Users' Information Systems Requests", **Entity-Relationship Approach to Systems Analysis and Design**, (edited by Peter P. Chen), North-Holland Publishing Company, Amsterdam, 1980, pp. 381-402
8. Warnier, Jean-Dominique, **Logical Construction of Programs**, H. E. Stenfert Kroese B. V., Leiden Holland, 1974
9. Warnier, Jean-Dominique, **Logical Construction of Systems** Van Nostrand Reinhold Company, 1981
10. Orr, Kenneth T., "Introducing Structured Systems Design", **Software Design Strategies**, (edited by Glenn D. Bergland and Roland D. Gordon), IEEE Computer Society, 1979, pp. 72-82

11. Steward, Donald V., "A Tale of Hope for Anyone Lost in the Software Forest Primeval", **Computerworld**, 19 March 1984, pp. ID1-ID16
12. Griffiths, S. N., "Design Methodologies -- A Comparison", **Software Design Strategies** (edited by Glenn D. Bergland and Roland D. Gordon), IEEE Computer Society, 1979, pp. 189-213

MECHANICAL TRANSLATION OF SOFTWARE REQUIREMENTS SPECIFICATIONS:
FROM ENTITY-RELATIONSHIP-ATTRIBUTE TO WARNIER-ORR

by

DONALD E. WOLFE

. B.S., The Ohio State University, 1977

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1984

The Entity-Relationship-Attribute model provides one possible technique for defining software requirements specifications. This project was to implement a system that would mechanically transform an ERA "keyword:text" style specification into a Warnier-Orr hierarchical specification. To accomplish this an intermediate process-representation syntax was defined as the sole mode of communication between the system's modules. This had the advantage of allowing piece-wise implementation of the modules the comprise the project. It will have the future advantage of providing a convenient interface to facilitate the addition of new modules and functions to this system. An interactive editor was also created to allow the viewing and updating of the Warnier-Orr specification.