79-1959

A UNIVERSAL FLOWCHARTER

D. Harel, P. Norvig, J. Rood, T. To
Higher Order Software, Inc.
Cambridge, Massachusetts

Abstract

A software tool has been developed to automatically produce flowcharts and concordances of user source programs. These flowcharts represent structured programming flow of control constructs, as opposed to traditional flowcharts which represent assembly level flow of control. The Flowcharter uses a table driven parse technique, so programs in virtually any language can be processed by changing tables. These tables consist of BNF rules for the language, where each rule is augmented by directives indicating what plotting actions to take, and what tokens to include in the concordance.

I. Introduction

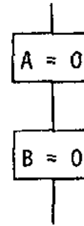
It has been shown[1,2] that structured programming techniques and formal documentation standards can be useful in expediting the software development process. An automated tool to support structured program development is the Universal Flowcharter, (UFC) which was developed for NASA's MUST system[3].

Although other automatic flowcharting programs exist, the UFC is unique for several reasons: it produces structured flowcharts, it is universal, and it produces concordances of program identifiers. Of particular interest is that execution flow is assumed to return in line at the completion of every control construct. This simplifies the representation of programs considerably. Structured flowcharts are described in Section II. The operation of the UFC as a universal* translator is discussed in Section III. Source programs are also analyzed for concordance information, as described in Section IV. Some details of implementation are given in Section V, and finally, some directions for future work appear in Section VI. Additional information[4,5] is available from HOS, Inc.

II. Structured Flowcharts

One approach to Structured Programming proposes that programs should be written using three major control constructs[6]: sequence, selection, and iteration**. Figures 1-3 show how these constructs are represented in structured flowcharts and in traditional flowcharts.

STRUCTURED TRADITIONAL (format varies)



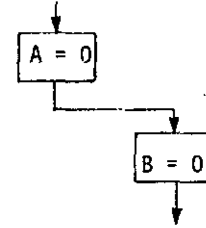
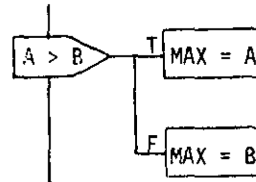


Fig. 1 SEQUENCE



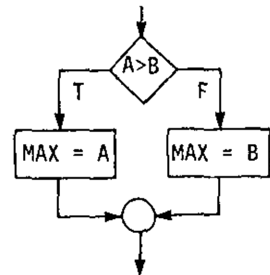
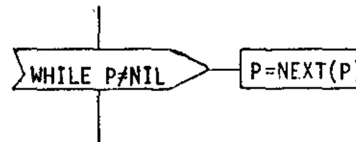


Fig. 2 SELECTION



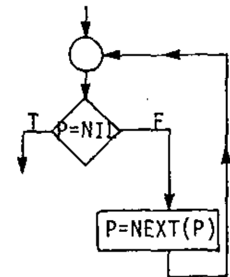


Fig. 3 ITERATION

Structured flowcharts have a number of constraints and conventions which make them easier to read than traditional flowcharts. For example, statements in sequence (Fig. 1) are always lined up in the same column, connected by lines going straight down the page. Similarly, alternate cases of a selection (Fig. 2) are lined up, indented one column to the right of the box containing the selection condition. There can be one, two, or more cases to select from. The multi-case selection construct (e.g., PASCAL case statement) is neater with structured flowcharting than with the traditional approach. The iteration construct shown in Fig. 3 reflects the way loops are usually written in structured programming, with the loop body indented to the right.

Flow of control in the program is always represented by lines that flow down the page (sequence), to the right (iteration) or down and then right

* Universal means that a program written in any language can be translated into a common language of flowcharts.
**In fact, Bohm and Jacopini[7] have proven any program can be written this way.

Released to AIAA to publish in all forms.

(selection), but never up or to the left. Therefore there is no need for arrows on lines, as the direction is unambiguous. We have the simple convention "go right when you can, down if you must, and back when you can't." When we reach a box with no lines leading out, control returns to the nearest box above and to the left, and then continues. This is similar to the idea of returning from a subroutine, and has no explicit flow line associated with it. Traditional flowcharts have no constraints on the placement of boxes, and lines may lead in any direction, turn several corners, merge together, and even cross one another. This "rats nest" approach makes it hard to follow the flow of control.

The arguments for adopting structured over traditional flowcharts are similar to those supporting structured over traditional programming, and include discipline, clarity, and modularity. We trust the reader is familiar with these arguments, and agrees that structured flowcharts merit further exploration.

In Figures 1-3 we have seen distinctly shaped structured flowchart boxes for iteration and selection. These shapes are used only in those contexts. We have also seen the rectangle, used for statements not related to flow of control (e.g., assignment statements). These are the three most inportant constructs. In Appendix I we describe the constructs chosen to correspond to the structured programming concepts of procedure statements, block statements, parallel processing, and non-deterministic processing. There is also a provision for off page connections in the case of programs too big to fit on one page. Constructs that would be indented over the right margin are labeled and continued later on a separate page. These eight constructs were carefully chosen to represent structured programs in a clear, easily readable format.

We have presented the argument that those who code using structured programming conventions would want to use the UFC. In addition the following also appears to be true. Hamilton and Zeldin have observed, "when the automatic structured flowcharter was first introduced, many programmers were converted to structured programmers over night, since this tool not only helped to enforce structured programming, but it also saved the programmers the work of manually producing a flowchart."[8]

On this note, let us now turn to a more detailed description of the Flowcharter itself.

III. A Universal Translator

The UFC can be compared to a compiler - it has a high level source program as input, and produces structured flowcharts instead of machine instructions as output. However, the UFC can also be compared to a compiler-compiler, because it is designed to work on any arbitrary source language. These two aspects of the UFC will be discussed in this section, followed by a discussion of lexical problems.

UFC as a Compiler

Producing a flowchart can be seen as a translation process from source language to "box language." In the UFC, this is done as a two pass compilation. First the input program is translated into what we call the Universal Flowchart Language (UFL). Strings in this language are composed of meta-symbols, which indicate what boxes and lines to print, interspersed with text from the source program, which will be printed inside the boxes. We speak of the constructs as templates, which are filled in with source text.

For example, if we were presented with the input string:

"IF A > B THEN MAX = A ELSE MAX = B" (4)

we would use the meta-symbol template for a two-case selection statement:

(→ /- -/ ; /- -/) (5)

to produce the UFL string:

(A > B → /-T-/ MAX = A; /-F-/ MAX = B) (6)

To specify this translation process, we describe the source language by a set of rules in Backus-Naur Form (BNF), with a corresponding set of UFL translation rules. The rules to effect the above translation are shown here:

BNF Rules	UFL Rules	
stmt: = IF test then stmt else	(2 → 3 4 5)	(7)
stmt: = ident = expression	1 2 3	(8)
test: = ident relop ident	1 2 3	(9)
then: = THEN	/-T-/	(10)
else: = ELSE stmt	; /-F-/ 2	(11)

The numbers in the UFL translation rules are gaps in the template, to be filled in by the UFL translation of the corresponding token on the right hand side of the BNF rule. For example, rule (9) has the UFL translation "1 2 3." This means the UFL translation will consist of the same three tokens as the source input. In contrast, rule (7) adds the meta-symbols "(", "→", and ")" to the source statement, or looked at another way, it places portions of the source into the selection template, "(→)".

The process of filling in templates can recurse to arbitrary depth. For example, the entire UFL string (6) might be nested inside a DO loop, which is nested inside a PROGRAM. The UFL rule for PROGRAM would return this deeply nested UFL translation and would mark the end of the translation process.

When the translation is completed, the next step is to print the flowchart which the nested template represents. This is a fairly straightforward process. Each template has a particular construct associated with it. For example, /-ℓ-/ causes the enclosed text (ℓ) to be plotted as a label, and (→ ;) gives rise to a selection statement with two alternatives. Thus, UFL template (6) would be plotted as shown on the left of Figure 2. Each level of nesting of UFL templates corresponds to a level of indentation in the resulting flowcharts. The only exception to this rule is that each procedure or subroutine is

plotted on a separate page, even if its definition is nested inside another construct.

UFC as a Compiler-Compiler

The translation process as described above is a straightforward, well understood procedure. It could be implemented by any of several techniques described in the literature, and is an easy task - for one particular source language.

However, the flowcharter was required to be universal, to accept any reasonable source language. This puts severe constraints on the spectrum of possible designs. The UFL must borrow from the design of Compiler-Compilers.

Compiler-Compilers, or Translator Writing Systems, have long been in use as software development tools. Typically, they allow the user to write a BNF description of the language to be processed, and to associate a semantic action with each syntax rule. The semantic action can be arbitrary, and might involve any of various things: output some machine instructions, make symbol table entries, update the location counter, invoke a macro substitution, etc. This gives the user power, but responsibilities as well. It is easy to write actions that introduce obscure but non-trivial errors.

The UFC makes this process less error prone by limiting the semantic action to one possible operation - outputting translations. There are actually two translations for each BNF rule, the UFL translation described above, and the concordance information which will be described in Section IV.

To satisfy the requirements of universality, the UFC simulates an automaton which can process any language. The algorithm used is LR(1) parsing,[9] which means the input is scanned from Left to right, producing a Rightmost parse, looking at 1 input token at a time.

Conceptually, the components of the automaton are a pointer to the input stream, a stack of intermediate results, a number indicating the current state of the system, and a table of actions to perform. This table can be constructed directly from the BNF rules and translations.

The action to perform at a given instance depends on the current state and the input token. There are only two major actions: shifting the input token onto the stack, and reducing the stack according to one of the BNF rules. These two actions are repeated in a loop that is terminated by one of the two auxilary actions: accepting the input string, and then passing the stack to the printing routine, or rejecting the input and printing an error message.

To be more specific, to shift means to take the input token, push it onto the stack, shift the pointer to the next token, and enter a new state. A reduction is the process of replacing the right hand side of a BNF rule by the corresponding left hand side. In our framework, this involves making the UFL translation, i.e., filling in the template. An example should make this clearer. Consider parsing string (4). The first four actions would be shifts, leaving the stack looking like this:

		(12)
ident	B	
relop	>	
ident	A	
keyword	IF	

The next action would be to reduce by rule (9). The UFL translation is "1 2 3," which means concatenate the top three elements of the stack, yielding:

		(13)
test	A > B	
keyword	IF	

After some more actions, we reach this state:

		(14)
else	; /-F-/ MAX = B	
stmt	MAX = A	
then	/-T-/	
test	A > B	
keyword	IF	

Finally, reducing by rule (7) yields:

stmt	A > B → /-T-/ MAX = A; /-F-/ MAX = B)	(15)

This is an example of the basic UFL template for a selection statement, "(→ ;)", with the appropriate source text filled in. We also have filled in the UFL template for a label "/- -/" with the letters T and F. When the parse is completed, the accept action will cause this template to be printed as shown on the left of Figure 2 in Section II.

The important point is the fact that this table-driven algorithm can accept any input language for which LR(1) grammars can be written. (Fortunately, virtually all languages have an LR(1) grammar.) To change from one language to another, we merely change tables, writing new translation rules. It is easier and less error-prone to write these rules than it is to write arbitrary subroutines, as would typically be done in a compiler-compiler environment. Although preparing tables is non-trivial, it need be done only once for each language. After the initial effort, producing a flowchart is as simple as compiling a program.

UFC as a Lexical Analyzer

The discussion above made the assumption that the input could somehow be broken into tokens, which were presented to the automaton one at a time. This process is called lexical analysis, and while easier than parsing, designing a lexical analyzer is not trivial. Not surprisingly, designing a universal lexical analyzer is considerably more complex.

For the UFC, it was necessary to make certain assumptions about the domain of languages to be processed, and these assumptions place limits on the universe. For example, the analyzer starts with the basic concepts of blanks, end of file, end of line, strings of digits, and strings of letters. Thus, a language where three blanks has a different meaning than four, or where "ABC" should be interpreted as two tokens, could not be handled by the UFC.

The lexical analyzer is tailored to a particular language by supplying lists of keywords, special symbols, comment delimiters, string delimiters, and other constructs. Again, writing lists is seen as a less error prone process than writing actual executable code.

IV. Concordance

Along with flowcharts, the UFC produces a concordance of identifiers in the program. Although only mentioned in passing above, this is another full-fledged translation task, executed along with the UFL parse described in the last section.

As one might expect, this requires a complete set of concordance translations, which are in one-to-one correspondance with the BNF rules (just as the UFL translations are). For example:

BNF Rules	Concordance Translations	
stmt: = IF test then stmt else	- U - U U	(16)
stmt: = ident = expression	A - U	(17)
test: = ident relop ident	I - I	(18)

Each concordance translation has one symbol for each element on the right hand side of the BNF rule. The symbols are:

- \- a syntactic unit containing no identifiers
- U a syntactic unit of unspecified type
- A a variable that is being assigned
- I an identifier of unspecified type
- L a local variable
- S the name of a subroutine

As the parse is proceeding, identifiers are filled into concordance templates, just as text is filled into UFL templates. A concordance template is created for each subroutine, and they all share this form:

NAME (S)	LOCAL VARS (L)
OTHER IDENTIFIERS (I)	ASSIGNED VARS (A)

(19)

Because the UFC allows only template construction as semantic action; there is no symbol table, and hence there is no way of determining the meaning of certain identifiers. For example, in some languages "F(x)" could be either a function invocation or an array reference. In such a case the identifier in question goes into the "other identifiers" section of the template. When all the templates have been filled in, we can form the list of all subroutine names, and intersect it with the "other identifiers" to obtain the list of procedure calls for each subroutine. Similarly, global variables can be found by subtracting local variables and and procedure names form the union of local variables and other identifiers. Other set operations are performed to obtain other interesting classes of identifiers.

V. Specification and Implementation Details

Specification

The UFC was designed using the AXES[10] specification language. AXES is a formal notation for defining systems. With AXES, one can define data types using abstract algebraic specification, functions which relate members of these data types, and control structures which relate functions. The interfaces between these objects can be automatically verified statically.

The purpose of this project was to build a universal flowcharter, but we were asked to apply AXES whenever possible, throughout all phases of development. The project was something of an experiment, and we were continuously observing our performance.

There were a number of factors which made this experiment a rugged test for AXES. It was a difficult system to design, because no program like the UFC had ever been built before, and the requirements were continuously changing. Several different engineers worked on the project. Some were involved throughout, others came on only in the programming stage. These programmers were handed AXES specifications written by someone else, and asked to come up with working code that would interface correctly with other modules.

As delivery dates approached, some designers panicked and decided to start implementing before certain data types and functions were completely specified. Others stuck to the formal methodology and completed the specification process. We found that any errors in implementation were in those functions where the specification was not complete before implementation. Therefore, we judged that AXES was of considerable help in the design and specification of the UFC.

Implementation

The UFC has been implemented as a PASCAL program of approximately 4000 lines. The program needs a partition of at least 100K bytes to run, and plotting a 1000 line program will require about 300K bytes, and run in a few seconds. The program was developed on CDC mainframe machines, (both SCOPE and NOS operating systems) and was rehosted on an IBM-370 with less than one day's work. This is a tribute to the modularity of the program, but mainly to the portability of PASCAL.

Output can be produced by a CALCOMP plotter, or by a lineprinter. Because there are two distinct plotting routines, we chose to translate the UFL templates into a common intermediate form. This is done by a recursive descent parse. The UFL template, which is a linear string, is de-nested into an internal tree form which represents exactly what the flowcharter will look like, with two exceptions: the exact size of each box and connecting line is not specified, and the internal form does not account for off-page connectors. These two details depend on the plotting medium and paper size, and are handled by the routine that does the actual plotting (either lineprinter or Calcomp).

The Flowcharter is currently set up to process programs written in either PASCAL or HAL/S languages. Because it is written in PASCAL, the Flowcharter is therefore capable of flowcharting itself, providing automatic documentation.

A sample input program and resulting flowchart are shown in Appendix II.

VI Future Work

Concordance Structures

The concordance information is gathered in a very simple manner, which does not account for the great diversity of programming languages. Problem areas include:

1) Distinguishing variables from procedures, particularly in languages that allow procedure-valued variables.
2) Allowing non-standard scope rules, such as dynamic binding in LISP.
3) Determining what variables are modified.
4) Recognizing compound names (like P → REC. NAME(I). FIRST).
5) Identifier collision problems.

It is difficult to define a common kernel of features to systematically obtain the necessary concordance information. There are many problem areas that are handled in a wide variety of ways in different languages. For example, using the identifier "F" inside function "F" would refer to a variable in FORTRAN, but would be a recursive call in PL/I. In PASCAL, it would depend on the context (e.g., in "F = N * F(N-1)", the first F refers to the value returned, and the second makes a recursive call).

Unfortunately, it appears that problems like this can only be handled by ad hoc routines to process each case. These routines must be either supplied by the user, or pre-programmed into the UFC, in which case the user would supply a set of flags along with the BNF grammar to indicate which way to handle each problem area.

Note that these problems do not arise in the syntactic UFL translation. This is probably because the theory of languages provides a sound mathematical base upon which all programming langauges build their syntax. There is no such common base for the semantics of languages.

Lexical Analysis

The above discussion assumes we know where individual tokens begin and end. This is not always a good assumption, as programming languages have a wide variety of lexical rules. For example, HAL/S is rare in that it allows comments within character strings. Some languages have column oriented formats. FORTRAN allows the pathological case of blanks within identifiers. The string "+10E+5" is a valid real number in some languages, but not others.

This variety of structure cannot be adequately handled by the few simple lists currently used as input to the lexical analyzer. What is really needed is another set of parse rules, either in LR(1) or regular expression form. This would go a long ways toward making the UFC "more universal," at the cost of an increase in execution time.

Parser Generator

In implementing the UFC it was found that much of the debugging time was spent changing BNF rules and translations, not the actual UFC source code. Although experience is valuable, producing BNF rules and tables is still a difficult operation. Even if a BNF specification is available for a given language, it probably will have to be altered for use by the UFC. This is because of the limited power of the lexical analyzer. Suppose a programming language had these two rules:

array_reference : array-name (exp-list) (24)

fun_reference : fun-name (exp-list) (25)

There would be no ambiguity, because the lexical analyzer would refer to the symbol table to distinguish between array-name and fun-name. The UFC would not be able to differentiate the two types of names, so rules (24) and (25) would cause a conflict. They would have to be replaced by a number of rules to resolve the ambiguity.

The ability to patch in this manner is considered something of a mystical art. Perhaps the most important area for improving the UFC lies not in the Flowcharter itself, but in developing a good tool for generating parse tables, such as the YACC[10] program, which could interface with the UFC. The development of software tools seems to breed a need for more software tools, which indicates there is still much work to be done.

Acknowledgement

The Flowcharter project could not have been completed without the contributions of R. Pankiewicz, D. Burns, and H. McManus.

References

[1] Dahl, O.J., E.W. Dijkstra, C.A.R. Hoare, Structured Programming, Academic Press, London, 1972.

[2] Wirth, N., Systematic Programming: An Introduction, Prentice Hall, 1973.

[3] Straeter, T., et. al., "MUST - An Integrated System of Support Tools for Reasearch Flight Software Engineering." A Collection of Technical Papers, AIAA/NASA/IEEE/ACM Computers in Aerospace Conference, Los Angeles, Nov. 1977.

[4] HOS, Inc., Flowcharter Functional Description, July, 1979.

[5] Harel, D. and R. Pankiewicz, "A Universal Flowcharter," TR-11, HOS, Inc., November 1976.

[6] McGowan, C.L., and J.R. Kelly, Top-Down Structured Programming Techniques, Mason/Charter Publishers, Brown University, NY, 1975.

[7] Bohm, C. and G. Jacopini, Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules, Comm. of the ACM, 9;5 (1966) 366-371.

[8] Hamilton, M. and S. Zeldin, Higher Order Software - A Methodology for Defining Software, IEEE Transactions on Software Engineering, March 1976.

[9] Aho, A.V., and J.D. Ullman, The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing, Prentice-Hall, 1972.

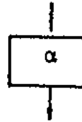
[10] Hamilton, M. and S. Zeldin, "AXES Syntax Description," TR-4, HOS, Inc., December 1976.

[11] Johnson, S.C., YACC - Yet Another Compiler-Compiler. Documents for the PWB/UNIX Time Sharing System, Bell Laboratories, 1977.

APPENDIX I

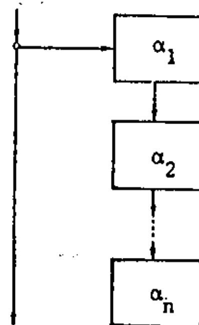
The Eight Control Constructs

(1) ELEMENTARY STATEMENT



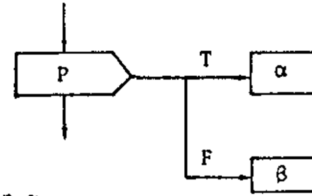
example: x:=x+1

(2) BLOCK STATEMENT



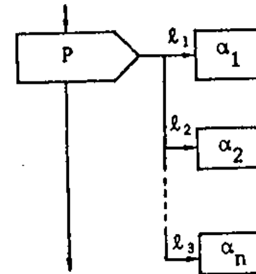
example: *begin*
α_1; α_2;...; α_n
end

(3) CONDITIONAL STATEMENT



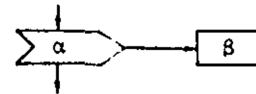
example: if P then α
else β

and similarly



example: case P of
ℓ_1: α_1; ℓ_2: α_2;...; ℓ_n: α_n
esac

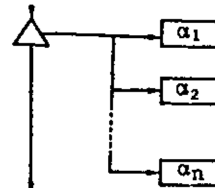
(4) ITERATIVE STATEMENT



example: while P do β od
(while P = α)

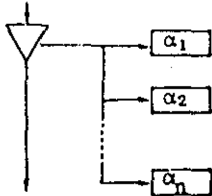
for x=y to z do β od
(for x=y to z = α)

(5) CONCURRENT STATEMENT (parallel processing)



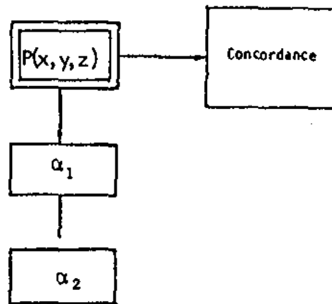
example: cobegin
α_1;
α_2;
⋮
α_n;
coend

(6) CHOICE STATEMENT (non-deterministic choice)



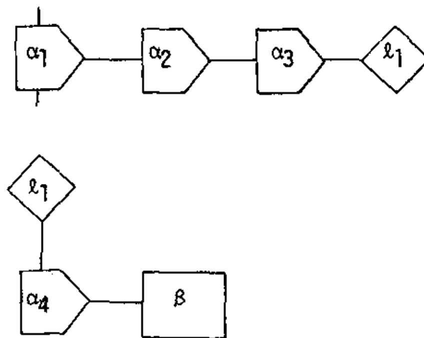
example: choose
α_1;
α_2;
⋮
α_n;
end

(7) PROCEDURE STATEMENT



example: Procedure P(x,y,z)
α_1;
α_2;
End P

(8) OFF-PAGE CONNECTION



example: if α_1 then
if α_2 then
if α_3 then
if α_4 then β

APPENDIX II

Sample Input and Output

As an example we sketch the flowchart of the following procedure statement:

```
Procedure  P(x,y);  x:=x+1; call Q
           if x<0 then
                 while x<0 do x:=x+1 od
               else
                 if x=1 then
                        cobegin
                              x:=x+1;
                              y:=v-1;
                        coend
               else x:=1; fi
         y:=x+1; fi
      z:=y;
end
```