1975

Proceedings of the 13th Annual Allerton
Conference on Circuit and System Theory

SPECIFYING A DESIGN LANGUAGE FOR DIGITAL SYSTEMS

TREVOR MUDGE
University of Illinois, Urbana, Illinois

ABSTRACT

Criteria for specifying a design language are proposed. The similarity these have with the tenets of structured programming is pointed out. A design language is specified using these criteria, which can describe networks of asynchronous logic modules. The process of translating the design language into networks of modules is outlined. An example design is presented. Finally, the design language is shown to satisfy the proposed criteria.

INTRODUCTION

In an attempt to formalize the design process for large digital systems, many researchers have suggested the use of design languages. Examples of such languages and advocacy of their advantages can be found in references 1, 2, 3 and 4. However, using a design language (DL) does not necessarily facilitate the design process. A poorly specified language can encumber the design process and fail to guide it away from design faults. This paper shows that through careful specification a DL can be created for a specific application (in this case the design of networks of asynchronous logic modules) so that by working within the syntax of the language, the designer is forced to formulate his design in a manner that allows him enough freedom to describe any flowchartable process, while at the same time limiting his freedom to describe faulty designs (in this case networks which hang-up). Furthermore this is achieved without the imposition of a complex syntax.

The process of specification is aided by a set of criteria that the DL should satisfy. The underlying motivation is to specify a DL which enables the user to design digital systems efficiently and with the minimum of design errors. Prevention rather than cure is the guideline.

The DL specified translates onto a set of asynchronous logic modules to produce the control structure of a digital system. Actions in the data structure of the system are assumed to be representable as register transfers. The asynchronous modules used by the design language are the W (wye) module, the S (sequence) module, the J (junction) module, the U (union) module, the D (decode) module and the I (iterate) module. These are discussed at length in the literature (references 5, 6, 7) and have been used in paper designs to illustrate their viability in constructing the control structures of complex digital systems (reference 8). The reader is assumed to have some familiarity with these modules.

This work was supported by the Joint Services Electronics Program (U.S. Army, U.S. Navy, and U.S. Air Force) under Contract DAAB-07-72-C-2059.

A design error, or an ill formed design, is considered to have occurred if the DL describes a network of the above modules, which during the course of normal operation will eventually hang-up or deadlock.

SPECIFYING A DESIGN LANGUAGE

The first step towards specifying a DL is the formulation of a description of the class of designs that the DL is required to produce. The DL should then be specified by a set of syntax rules that satisfy the following three conditions.

C1) The language resulting from the set of syntax rules should include the class of designs that the DL is intended to produce.

C2) The language should not include ill formed designs.

C3) The language should be specified in such a way that the user can easily avoid making syntatical errors.

The requirement that C1 be true of the language is obvious. However it is necessary to check that specifying a DL to satisfy C2 and C3 does not result in it failing to satisfy C1. Given a description of the class of designs that the DL is required to produce and a set of syntax rules, it should be possible to prove whether or not the language resulting from the syntax rules includes the required class of designs. As an example, the DL presented in the next section is required to produce designs which control any flowchartable process, making it suitable for the design of the control structure of a large class of general purpose digital hardware. Examination of the syntax rules shows that such designs are indeed produced within the limits of the syntax rules.

The requirement that C2 be true of the language specified means that a syntax analyser will implicitly check for design faults. If the characteristics of an ill formed design (in our case networks which hang-up) are identified, a set of syntax rules can be formulated which produce designs in the language which are never ill formed. Given a set of conditions which define a well formed design and a set of syntax rules which define the DL it should be possible to prove that the DL includes only well formed designs.

Condition C3 is not a condition that can be shown to have been satisfied by mathematical demonstration as C1 and C2 can. This is because it is a qualitative rather than quantitative condition. A loose characterization might be to say that given two DLs satisfying C1 and C2, the one with the simpler syntax rules more nearly satisfies C2. Whether this makes the user's task easier is open to debate; however, it certainly makes a syntax analyser's task less complex.

The many underlying similarities between the requirements of a good procedure oriented programming language and a DL are highlighted by C1 and C3. These two conditions are inherent in the philosophy of structured programming. In the case of procedure oriented programming languages the class of programs of interest are those with flowchartable control logic. The "Structure Theorem" (reference 9) guarantees that by using the three types of structure f THEN g, IF p THEN f ELSE g and WHILE p DO f (f and g procedures and p a predicate), any flowchartable control logic can be represented. Hence C1 is satisfied for procedure oriented programming languages even when the restricted constructs required for structured programs are used. Whether C3 is satisfied is arguable, as noted above; however, the above three constructs allow the programmer to formulate his programs in a systematic top down fashion (reference 9) which has received wide acceptance as a methodology which is suitable for program formulation by humans, and hence which tends to reduce programmer errors. On the other hand, C2 is not true for procedure oriented programming languages, as certain control logic can be indicated by the programmer which is not executable (ill formed). This is particularly easy if unrestrained use of the GO TO statement is allowed; thus the requirement of structured programming that GO TOs be forbidden, or else used in some very restricted fashion.

A DL is specified in the next section, and at the end of the section it is shown to satisfy the above three criteria.

A DESIGN LANGUAGE

The Syntax

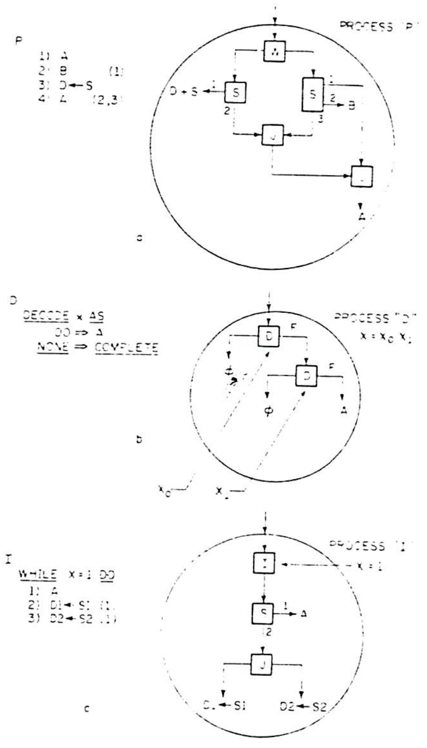
The syntax of the DL is given in the appendix. Several illustrative examples of the use of the DL are shown in figure 1. The networks of modules that these examples translate to are shown alongside. It should be clear from the appendix that the language is block structured. There are three types of blocks (PROC, DPROC, and WPROC) and an example of each is shown in figure 1.

Figure 1(a) shows a block named "P" which describes the following process. Upon the activation of process "P", "A" is initiated together with the register transfer action "D $\leftarrow$ S" (move the contents of data cell S to data cell D). When "A" is completed process "B" is to be initiated. When both "D $\leftarrow$ S" and "B" are completed process "A" is to be reinitiated. Finally, with the completion of "A", process "P" is considered completed. This is an example of a PROC type block. Notice the integers to the left of the statements. These are used in the parentheses on the right of the statements to indicate sequencing information. For example, statement 4 says that process "A" is to be initiated when the processes indicated in statements 2 and 3 are both completed. No parenthesized integers to the right of a statement indicates it is to be initiated immediately the process described by its block is initiated.

Figure 1(b) shows an example of a DPROC type block. This indicates how the control is to branch according to the state of some external variables (X). The reserved word NONE stands for the union of all these conditions of the external variables not explicitly listed (i.e. $X_0 \vee X_1 = 1$). The reserved word COMPLETE indicates that if NONE is true the empty process ($\emptyset$) is to be completed. Then the process described by block "D" is considered completed.

Figure 1(c) shows an example of a WPROC type block. In this block process "A" is to be initiated first. When it is completed the register transfer processes "D1 $\leftarrow$ S1" and "D2 $\leftarrow$ S2" are both to be initiated. When they are both completed the whole block, starting with A, is to be reinitiated as long as the predicate "X=1" holds true. As soon as the predicate is no longer true, and both "D1 $\leftarrow$ S1" and "D2 $\leftarrow$ S2" have been completed, the process "I" is then considered completed.

Figure 1



Space limitations prevent a more complete exposition of the DL. For this the reader is referred to reference 10. However, before closing this subsection on the DL's syntax, a few more comments are in order.

There are only three types of statements in the DL. The IO type, which indicates external asynchronous communications with the control structure. The REG-TRF type, which indicates register transfers in the data structure (e.g. statement 3, figure 1(a)). Finally, the PROC-CALL type, which indicates a process which is described by the block having the same identifier as the PROC-CALL statement (e.g. statement 1, figure 1(a)). Notice the analogy with subroutine calls in procedure oriented programming languages. Natural top down structuring of the design also results from this last type of statement.

The similarities between the DL and a procedure oriented language such as Algol are obvious. Nevertheless there are several important differences. Firstly there is no parameter passing by the PROC-CALL type statements (this would be analogous to subroutine parameter passing). This was done for simplicity and there is no conceptual reason why the DL could not be extended to incorporate this facility. Parameters could be data cells or other blocks, so that a block could be shared by several similar tasks which operate on different data cells. (The translator would have to add switches to the data structure so that different sections of the data structure could be switched to the same section of control structure for processing). Secondly an Algol like language is a sequence of statements whose order of execution is important. In the DL the order in which the statements appear within their blocks, and also the order in which the blocks appear is unimportant. This is because the result of translating a design in the DL onto the modules can be viewed as a directed graph whose nodes are the modules. The position of an arc in the graph is not dependent upon the order in which it is placed into the graph.

The Translation Process

It is appropriate at this point to make some qualitative comments about the translation process. A more in depth quantitative treatment is given in reference 10.

The translation process is quite simple, as can be deduced from figure 1. DPROCs translate to networks of D modules, WPROCs translate to networks of W, S and J modules headed by an I module, and PROCs translate to networks of W, S and J modules. The intra block connections of W, S and J modules are determined by the sequencing or order information given in the parentheses to the right of the statements. The order information can be easily expanded to form a matrix representing the binary relation "is the successor of". This is done below for the block of statements given in figure 1(a). If m(i,j)=1 then the statement labelled i "is the successor of" the statement labelled j.

$$m = \begin{array}{c|ccccc|l} & 0 & 1 & 2 & 3 & 4 & \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & \\ 1 & 1 & 0 & 0 & 0 & 0 & \\ 2 & 0 & 1 & 0 & 0 & 0 & \\ 3 & 1 & 0 & 0 & 0 & 0 & \\ 4 & 0 & 0 & 1 & 1 & 0 & \leftarrow J \\ \hline & \uparrow & & & & & \\ & W & & & & & \end{array}$$

Each statement except 0 and those having no successors (e.g. 4 in the above) translates to a three port S module. Columns with n+1 ones imply n W modules and rows with n+1 ones imply n J modules. A similar relational matrix for inter block connections can be formed and the assignment of U modules deduced.

These matrices, the inter block matrix and the intra block matrices can also be used to complete the syntax analysis. The context free grammar in the appendix which describes the DL's syntax, does not give a complete description of the syntax. Several other constraints must be imposed, so that C2 is satisfied, which cannot be conveniently expressed by a set of production rules. Firstly there are to be no directed cycles in the networks of modules described by the DL. Secondly whenever a process is shared (e.g. A in figure 1(a)), implying the use of a U module, descriptions in the DL which allow the possibility of both input links to the U module to be simultaneously active are to be excluded from the language. (The reason for these constraints are explained more fully in the last subsection of this section). These additional constraints can be checked for by various operations with the relational matrices.

As an example: directed cycles in a block can be identified by forming m, m^2,..., m^{k-1} (k = number of statements in the block) and checking for ones in their main diagonals. Due to the block structure of the DL, the relational matrices are usually small (typically less than 10 X 10), making matrix manipulations quite feasible.

An Example Using The Design Language

The design example presented here is a single instruction computer called SIM (Single Instruction Machine) which performs the single three address instruction:

```
SUBTST A, B, P = A ← C(A) - C(B)
                 IF C(A) = 0 THEN
                 PC ← C(P)   (PC is the program counter)
```

Its control structure is described in the DL below.

```
SIM
WHILE RUN = 1 DO
    1) DECI
    2) FETCH  (1)
    3) EXEC   (2)
```

Comments:
This is the basic instruction fetch and execute cycle. It continues as long as the RUN button is on. Furthermore, it tests the interrupt flag (DECI) before each cycle.

```
DECI
DECODE INT AS
        1 = INTR
        0 = COMPLETE
```

```
INTR
    1) MAR ← IPA
    2) DR ← IP
    3) M ← DR  (1,2)
```

INTR handles the interrupt. IPA holds the address at which the interrupting data is to be stored. IP holds the interrupting data. The interrupt is only for inputting data.

```
FETCH
    1) MAR ← PC
    2) DR ← M     (1)
    3) PC ← INC   (1)
```

This is the FETCH routine. MAR is the memory address register. PC is the program counter. DR is the data register. Note the parallelism between statements 2 and 3.

```
EXEC
     1) IR ← DR
     2) MAR ← IRA      (1)
     3) DR ← M         (2)
     4) OUTPUT DR1     (3)
     5) A ← DR         (3)
     6) MAR ← IRB      (3)
     7) DR ← M         (4,5,6)
     8) DR ← SUB       (7)
     9) MAR ← IRA      (7)
    10) OUTPUT DR2     (8)
    11) M ← DR         (8,9)
    12) DEC            (10,11)
```

This executes the 3 address instruction. IRA holds the address A, IRB holds the address B. OUTPUT DR1 and DR2 are links out of the control structure. These indicate to the external environment that the DR is loaded with data which could be read out to some external device. Since the system is asynchronous, SIM will not continue until the OUTPUT ports receive an acknowledge signal from the external environment.

```
DEC
DECODE DR AS
        0 ... 0 = BR
        NONE   = COMPLETE
```

DEC checks to see if the result of the subtraction is zero.

```
BR
    1) DR ← PC
    2) PC ← IRP   (1)
    3) MAR ← PC   (2)
    4) M ← DR     (3)
    5) PC ← INC   (3)
```

If the result is zero, BR handles the branch of control. C(PC) is loaded at the branch address and PC is loaded with the branch address.

The resulting data structure is shown in figure 2. Symbols such as I represent identity operators. The one indicated moves data from the program counter to the memory address register when requested to do so by the S module corresponding to the DL register transfer statement "MAR ← PC" (see FETCH block statement 1).

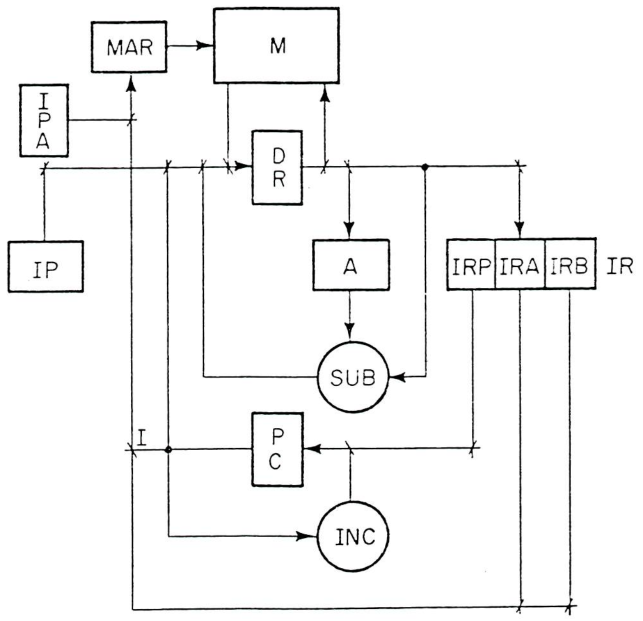


Figure 2

When a control structure is hooked up to a particular data structure the determinism of the total system must be verified. The presentation of a systematic method to check for determinism is outside the scope of this paper. In the above example the determinism was checked by visual examination of the register transfer statements in the DL description. (This would clearly be inadequate in a much larger design).

The block structure of the design is shown in figure 3, together with a close-up of the inside of one of the block (INTR). Figure 4 shows the realization of the most complex block (EXEC) in terms of the modules.

<u>The Design Language And The Three Specification Conditions</u>

In this subsection some informal arguments show that the DL presented above satisfies the specification conditions C1, C2 and C3. (More formal arguments can be found in reference 10).

The DL is required to produce designs which control any flowchartable process. It was noted earlier that the following logical structures were sufficient to represent any flowchart logic.

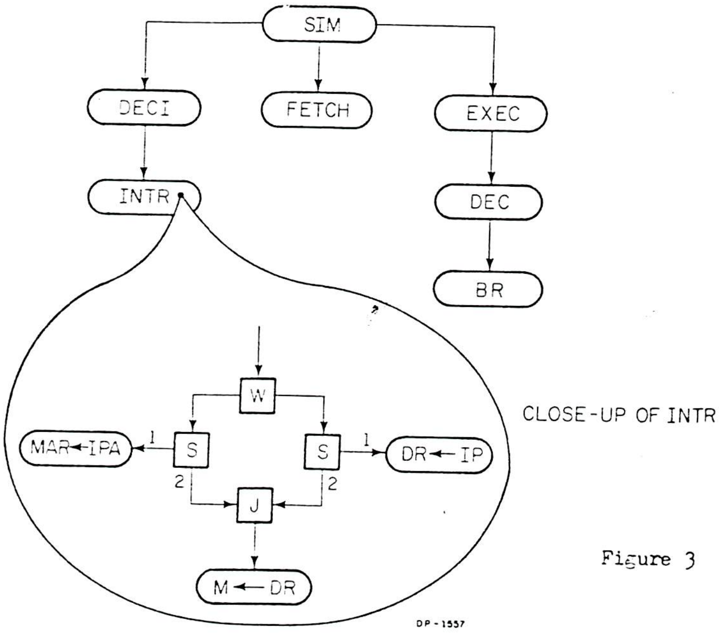


Figure 3

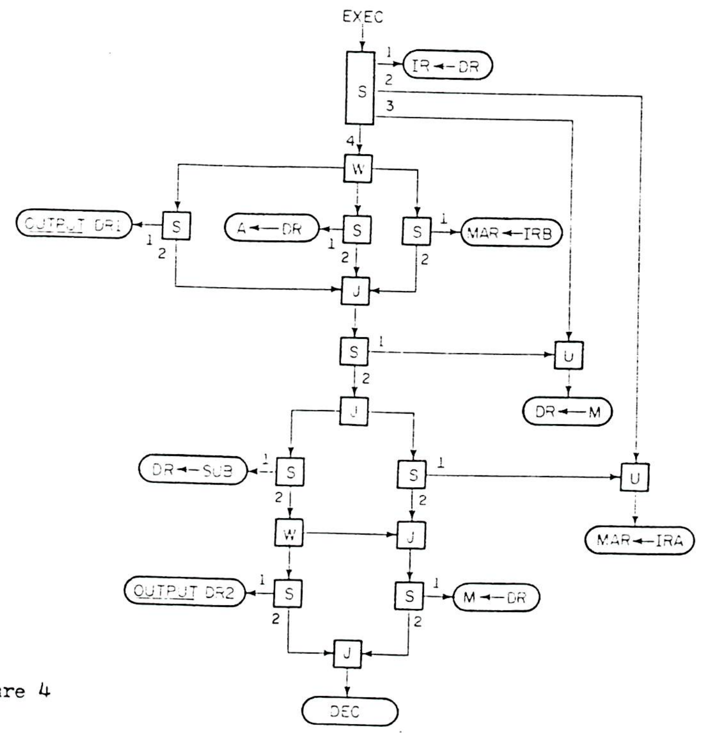


Figure 4

1) f THEN g
2) IF p THEN f ELSE g
3) WHILE p DO f

These three logical structures exist in the DL; 3) exists explicitly - the WHILE block (WPROC), 2) exists in a more general form - the DECODE block (DPROC), and 1) exists in a more general form - the sequencing or order information. Hence C1 is satisfied by the DL, since the language resulting from the set of syntax rules includes the class of designs that the DL is required to produce.

In their paper "Asynchronous Control Networks", Bruno and Altman (see reference 6) present a set of criteria that any network of W, S, J, D and I modules must conform to, to ensure that it cannot hang-up. If the network of modules is viewed as a single component directed graph:

1) I modules must be 2-way articulation points.
2) D modules must be 3-way articulation points.

When the I and D modules are removed the remaining components are composed of W, S and J modules.

3) These components must be circuit free.
4) Their precedence graph must be circuit free.

(The precedence graph of a network N of the above modules describes the relative order with which the output links of N can be activated (references 6 and 7)).

The control networks described by the DL include an additional module, the U module. This invalidates criteria 1) and 2). However, in reference 10 it is shown that because of the restricted use of the U module by the DL, any network of W, S, J, D, I and U modules described by the DL is well formed provided an associated network of W, S, J, D and I modules satisfies the above criteria. It is also shown that if the design in the DL is syntactically correct, the separate blocks for I and D modules (WPROCs and DPROCs) together with the fact that entry to a block is always thru a single link, imply criteria 1), 2) and 4) are satisfied by the associated network. Lastly, it is shown that checking for 1), 2) and 4) in a directed graph corresponds to recognizing the context free language which is the DL, a computationally much more efficient procedure. Criterion 3) is satisfied by a syntactically correct design, as is the additional constraint, the U condition. (The U condition requires that both input links to a U module never be active simultaneously). It was noted in the subsection on the translation process, that these two remaining conditions were checked for by the manipulation of small matrices. Hence the DL satisfies C2, since it does not include ill formed designs.

Whether the DL satisfies C3 is a matter of opinion. The following closing remarks of this section give the author's opinion. The block structuring, especially the requirement of separate blocks for WHILE and DECODE statements should help the designer to formulate his design in an error free fashion. Furthermore, the top down structuring of the blocks induced by the PROC-CALL type statements maintains a correspondence between the textual description of the digital machine and its proposed operation that should also facilitate the design process. The only sources of syntactical error not made conspicuous by the form of the syntax are U condition conflicts and the creation of directed circuits thru incorrect use of the order information.

APPENDIX

The syntax is given in extended BNF. Use is made of the following meta-symbols.

```
     A[B]  ⇔  A|AB
   A{B|C}  ⇔  AB|AC
    {A}+   ⇔  A|AA|AAA|...
    {A}*   =  {A}+|∅
```

Non-terminal symbols are sequences of uppercase letters and hyphens. They are separated by blanks. The terminal symbols are underlined. Those that are spelt differently in the text are listed together with their spelling below.

DL	carriage return, line feed	INITIATE	➔
LP	(	ASSIGN	→
RP	)	COMMA	,
LETTER	A,...,Z	ZERO	0
DIGIT	1,...,9	ONE	1

Blanks may be inserted between terminals for ease of reading.

```
PROGRAM ::= {PROC|DPROC|WPROC}+

PROC  ::= DL PROC-ID STAT-LIST
DPROC ::= DL PROC-ID DSTAT
WPROC ::= DL PROC-ID WSTAT STAT-LIST

STAT-LIST ::= {DL LABEL STAT}+
STAT      ::= {PROC-CALL|REG-TRF|IO} ORDER-INFO

PROC-CALL ::= PROC-ID
REG-TRF   ::= NAME ASSIGN NAME
IO        ::= {INPUT|OUTPUT} PROC-ID

DSTAT ::= DL DECODE NAME AS DLIST
DLIST ::= {DL BITS INITIATE ACTION}*DL NBITS INITIATE ACTION

WSTAT ::= DL WHILE PRED DO
PRED  ::= NAME REL {NUM|NAME}

PROC-ID ::= ID
NAME    ::= ID SUB-OPTION
ID      ::= LETTER{LETTER|DIGIT}*
LABEL   ::= NUM RP
NUM     ::= {DIGIT}+

ORDER-INFO ::= SUB-OPTION
SUB-OPTION ::= [LP SUB-LIST RP]
SUB-LIST   ::= NUM{COMMA NUM}*

ACTION ::= COMPLETE|PROC-ID
NBITS  ::= BITS|NONE
BITS   ::= {ZERO|ONE}+

REL ::= useful binary relations
```

REFERENCES

1) Computer Magazine. IEEE Computer Soc., Dec. 1974.

2) Chu, Y. "An ALGOL Like Computer Design Language," Comm. ACM (Oct. 1965), vol. 8, pp. 607-615.

3) Duley, J.R.; and Dietmeyer, D.L. "A Digital System Design Language (DDL)," IEEE Trans. Computers (Sept. 1968), vol. C-17, no. 9, pp. 850-861.

4) Friedman, T.D. "ALERT: A Program to Compile Logic Designs of New Computers," Digest 1st Annual IEEE Comp. Conf. (Sept. 1967), pp. 128-130.

5) Patil, S.S.; and Dennis, J.B. "The Description and Realization of Digital Systems," IEEE Comp. Conf. (Sept. 1972), pp. 223-226.

6) Bruno, J.; and Altman, S. "Asynchronous Control Networks," IEEE Conf. Proceeding 10th Annual Symposium on Switching and Automata Theory (1969), pp. 61-73.

7) Altman, S.M.; and Denning, P.J. "Decomposition of Control Networks," Record of the Proj. MAC Conf. on Concurrent Systems and Parallel Computation (1970), pp. 81-92.

8) Dennis, J.B. "Modular, Asynchronous Control Structures for a High Performance Processor," Record of the Proj. MAC Conf. on Concurrent Systems and Parallel Computation (1970), pp. 55-80.

9) Mills, H.D. Mathematical Foundations of Structured Programming. IBM, 1972.

10) Mudge, T. A Design Language for Modular Asynchronous Control Structures. CSL Report No. R- , Univ. of Illinois, to be published.