CR-103840

A Formal Theory of Cubical Complexes

by

J. Paul Roth, E. G. Wagner and G. R. Putzolu

Formal Report

September 1, 1968 to April 30, 1969

Contract 952341

IBM Watson Research Center

Yorktown Heights, N. Y. 10598

Work partially performed for the Jet Propulsion Laboratory, California Institute of Technology, sponsored by the National Aeronautics and Space Administration under Contract NAS7-100.

MAY 1969

A Formal Theory of Cubical Complexes

Summary

I. Finishing touches were put to the Multiple Output Minimization algorithm MOM, for logic circuits of two levels, although additional improvements to the APL program will be made. A paper on MOM has been accepted for presentation at the Hawaii International Conference on System Sciences January 1969.

II. Work has continued on the development of F-notation, a high-level language for the description of algorithms and programs. A formal syntax for the F-notation has been developed, which enables one to contrast this language with e.g. LISP.

III. An algorithm and APL program have been developed for the diagnosis of failures of cyclic logic circuits, termed the C-algorithm or C-alg. The C-algorithm is a generalization of the D-algorithm, which worked for acyclic circuits. The running time of C-alg is very satisfactory: in the APL time-sharing system, based on a S/360 mod 50, we have been able to compute tests for failures in actual logic circuits of about 50 blocks within less than a minute (including time for simulating the test); furthermore running time appears to be strikingly independent of size. Running time does seem indeed to increase with size but, because of certain features of the algorithm (a "plunge" of the D-chain to a Primary Output PO is made rather than an orderly march of all ends of a D-chain to the PO's as

in the D-algorithm), the number of logical blocks does not seem to be a very significant parameter. "C-alg complexity" is rather measured in terms of the "interwovenness" of the loops of the circuit. The C-algorithm seems assured of practical usage.

The number of single stuck-at-1, stuck-at-0 failures is twice the number of lines of the circuit whereas the number of multiple failures, due to a variety of physical phenomena such as cracks, is very much larger. A strategy has been devised to compute a set of tests, fairly close to a minimum, which tests all stuck failures such that, with high probability, it also tests all multiple failures, of a given general category.

An F-program of C-alg is included in this report.

I. The Multiple Output Minimization Algorithm MOM for 2-Level Logic Circuits.

Work on MOM was mostly done during the last period [RWLP 68]. A paper summarizing this work [RWL 69] was prepared and accepted for presentation at the Hawaii International Symposium on System Sciences January 1969. This summary is included in this report as appendix A. In particular it includes new and much simplified characterizations of the notions of singular complex and the set Z of prime singular cubes.

An improved #-algorithm for computing the prime singular cubes has been developed, which will be incorporated in the APL program MOM. This improvement should significantly improve its running time and space requirements. Essentially the computation is made to be iterative rather than recursive.

II. Development on F-Notation

1. A formal syntax.

Two approaches to the representation of algorithms for switching theory were employed earlier in this project: the calculus of α-objects and F-notation. Considerable effort and progress are being made to join these two representations with the goal of producing a very general, powerful, convenient and well-founded "language" for writing algorithms.

As presented earlier the α-object calculus was precise and formal but was neither easy to use nor concise. The F-notation, on the other hand, was easy to use and concise but it was neither precise nor formal. Present efforts are directed toward combining the best of both approaches. In particular we are employing a (formalized) version of the syntax of the original F-notation and we are employing an extended version of the α-object calculus to provide precise, formal semantics to go with this syntax. The result will be the "official" version of the F-notation. We believe that this "language" will be of great aid in accurately and concisely describing switching theory algorithms in a manner which will both make them easy to understand, easy to program, and easily accessible for mathematical study and verification. At the same time this "language" should also be useful in a wide variety of other disciplines--any discipline, in fact, in which it is necessary to describe algorithms which manipulate strings, lists, (finite) sets, tuples, etc. .

Work on the development of this formalized F-notation has progressed to the point where we have begun the writing of a report on the formal syntax and semantics of the language. We plan to embed this report in a larger report which will contain examples of the application of the language both to specific switching theory algorithms and, as in our earlier paper on the α-object calculus, to building up the fundamental data structures of switching theory.

We also anticipate including a section on various informal modifications of the language which facilitate the writing of algorithms in disciplines in which the usual informal notation is not the same as that of F-notation (e.g. in formal F-notation we use prefix notation while in arithmetic one generally uses infix notation (i.e., using infix notation we write "a+b" while in prefix notation this would be written "+(a,b)")).

III. Logic Testing

Most logic circuits, as developed on cards, etc., as part of a computer or electronic system using integrated circuits are cyclic, that is, have feedback. To date practical algorithms for computing tests to detect failures of logic circuits have been restricted to acyclic circuits. We have been able to extend the D-algorithm to compute tests for cyclic circuits, this algorithm has been programmed in APL and its efficiency and failure-coverage appear to be satisfactory. A description of the problem, previous results, the new algorithm, termed the C-algorithm, examples of the workings thereof and new problems will now be given.

INTRODUCTION: A heuristic algorithm C-alg for the computation of tests for failures in sequential or cyclic circuits is described. It is an extension of the D-algorithm [ ] which for combinational circuits computes tests for failures if they exist. First the problem is described. A logic circuit S is formed by interconnection of combinational logic blocks (like NOR or NAND). Given S and given that one of its logic blocks has experienced a failure, the problem is to find a test T consisting of a sequence of input patterns to the primary inputs of S such that for the corresponding sequence of output patterns, there is a difference between the correct output and the output for the failing circuit. Illustrations of the operations of the D and C-algorithms are given. Next a formal description of the models used in the C-algorithm are described. Then a high-level description of the C-algorithm is given in F-notation, an algorithmic language. Finally brief comments are made concerning the efficiency of an APL version of the algorithm.

1. Description of Problem

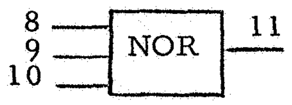
A logic circuit S is formed from combinational logic blocks, like NOR's, NAND's, OR's, AND's, by interconnecting a set of these blocks together by identification of inputs of one with outputs of others, in arbitrary fashion.

Given a logic circuit S and given that one of its logic blocks has experienced a failure F in some prescribed manner, the problem is to find a test T consisting of a sequence of input patterns such that for the corresponding sequence of output patterns there is a difference between the correct output and the output of the failing circuit.

First we will illustrate the D-algorithm which computes a test for combinational (acyclic) circuits.

2. Illustration of D-Algorithm

The basis for the algorithm for diagnosis of cyclic (or sequential) circuits is the D-algorithm version II as described in [1] for acyclic (combinational) circuits. We briefly recapitulate a slight extension of this via an example. First let us consider an individual logic block, a NOR block denoted N with its input and output lines labeled 8, 9, 10, 11 as shown in Fig. 1.



8	9	10	11
1			0
	1		0
		1	0
0	0	0	1

Figure 1 - A NOR Circuit and its Logical Description

The logical description of its behavior is depicted by the table on the right side of Fig. 1 the first row indicating for example that when line 8 is 1, output line 11 is 0, regardless of the value of the other input lines. Suppose the circuit of Fig. 1 is embedded in the logic circuit of Fig. 2.

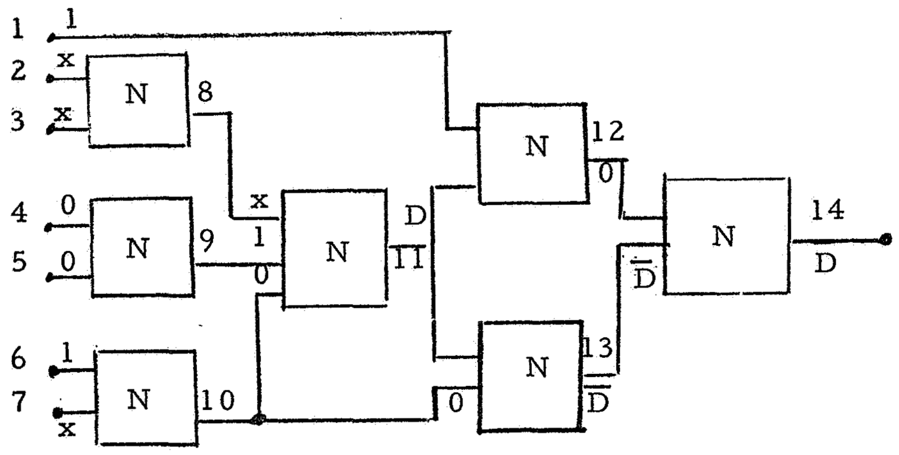


Figure 2 - Illustration of Operation of D-Algorithm

and assume that line 11 has failed by being stuck-at-0. This is denoted by assigning the value D (D for "diagnose") to line 11. Here D is to be thought of as a variable distinguishing between the case of the occurrence or non-occurrence of this failure. When D = 1 this particular failure has not occurred; when D = 0, it has occurred. One defines initially a test cube tc consisting of x's in all coordinates except line 11,

xxxxxxxxxxDxxx.

The x's indicate that no conditions have been yet imposed upon these lines. The "activity vector" A consists of the successors 12, 13 to line 11. One of these lines must be chosen to "drive" (D-drive) the D to a primary output (here line 14). In our program (DALG III) line 13 is chosen. In order to allow the "D-chain" to propagate through block 13, a 0 must be imposed on line 10 and if this is done then line 13 is assigned the value $\overline{D}$ which means, under these conditions that line 10 is a 0, that $\overline{D}$ = 1 if line 11 is stuck-at-0 and $\overline{D}$ = 0 otherwise Blocks 11 and 13 are to be thought of as "D-chained" together.

In order to be sure that line 10 is 0 either line 6 or line 7 must be assigned the value 1 but this decision is not made at this part of the algorithm, called D-drive. The activity vector now becomes 12, 14.

Again the last one is chosen, and in order to drive the $\overline{D}$ through this block line 12 must be made an 0, yielding a D on line 14, yielding the test cube:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
tc =	x	x	x	x	x	x	x	x	x	0	D	0	$\overline{D}$	D

The subalgorithm D-drive has now reached a primary output, line 14, and thus D-drive terminates. The next subalgorithm to be executed is called "consistency" or CDRIVE; in this sub-algorithm internal lines, here 10 and 12, which have been assigned values, are "justified" by suitable choice of signals on primary input lines. For line 12 to be a 0, we assign primary input line 1 to the value 1.

As for line 10 being 0, this may be achieved by line 6 (or line 7 or both) being assigned the value 1. The "test cube" then consists of:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
tc =	1	x	x	x	x	1	x	x	x	0	D	0	$\underline{D}$	D

Finally line 11 being maintained at a D means that in the "good" circuit either line 8 or 9 must be assigned the value 1 (so that 11 will be 0 in the good circuit): we choose line 8. Finally, line 8 has the value 1 only when both input lines 4 and 5 are 0. Thus the final test cube is:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
tc =	1	x	x	0	0	1	x	x	x	0	D	0	$\underline{D}$	D

Since lines 1 through 7 are primary inputs the actual test pattern is defined by:

1	2	3	4	5	6	7
1	x	x	0	0	1	x

Here the x's may be assigned arbitrary values, independent of each other; thus the test cube actually prescribes 8 individual tests for this failure.

This summary does not cover the complexities of the D-algorithm as it actually exists in D-ALG in the APL implementation. Elaborate tracking procedures and decision-making machinery are involved. Furthermore the D-algorithm actually solves the combinational diagnosis problem in that if a test exists it is proven that the D-algorithm will actually compute one.

3. Illustration of C-Algorithm

We now extend the D-algorithm approach to the sequential case. Before giving a formal description of the cyclic diagnosis algorithm, we will discuss an example illustrating the workings of the algorithm.

D-propagation. Consider the circuit shown in Fig. 3, consist-

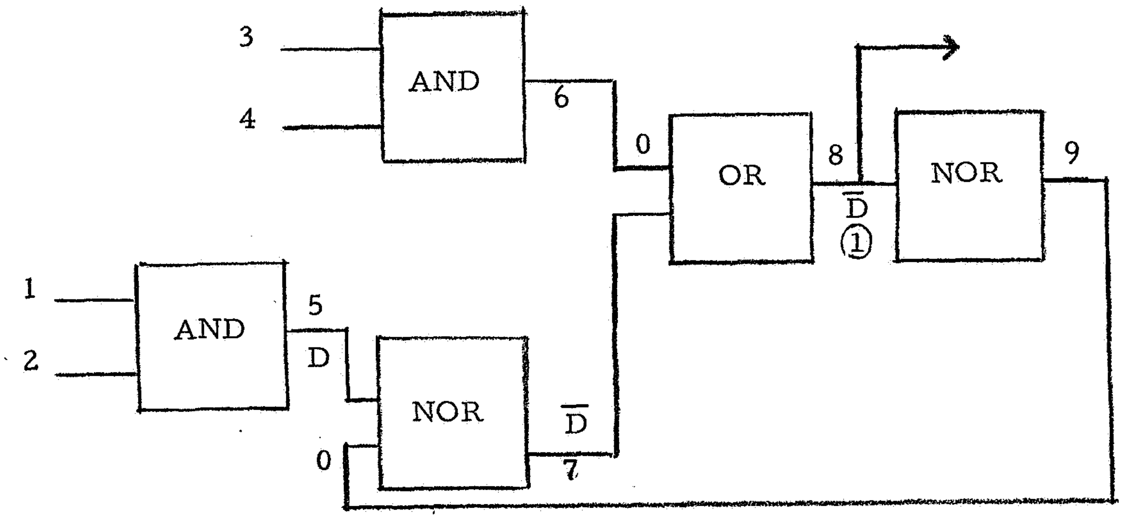


Figure 3 - D-Propagation in a Sequential Circuit.

ing of five logical blocks, two ANDS, two NORs, and an OR interconnected as shown. The lines are labeled with integers as shown, the four primary inputs being labeled 1, 2, 3, 4. Assume in this circuit that the failure under consideration is line 5 stuck at 0. A D is associated with this line and the first task is to "propagate" this D to a primary output, which is possible in this case. To "get through" the NOR block with output line 7, line 9 must be placed in the value 0 (this will be later on justified in the consistency operation). This line 7 assumes the value $\underline{D}$ and line 8 is a primary output so that the D-drive (or D-propagation) ceases operation.

Consistency. We must now justify the imposition of the signals on lines 9 and 6, in a previous time frame. For line 9 to have the value 0 it is necessary that the value of the input to this block, line 8, should be 1. (This, however, is clearly not an inconsistency, as it would be in the combinational case, but the proper interpretation is that one of the signals occur in a "previous" time frame.) We circle this $\textcircled{1}$ in the diagram to indicate that it occurs in the past in Fig. 4. This may be achieved by line 6 having the value 1 (avoiding the feedback loop) which in turn would require that both lines 3 and 4 also have the value 1. Finally the value D on line 5 (indicating that line 5 is stuck-at-0) is maintained by having lines 1 and 2 at values 1 in the present time frame. In Fig. 4 the values that were imposed in the previous time frame are encircled. Thus the test, thus far generated, using two time frames is:

	1	2	3	4	5	6	7	8	9
Prior Time Frame	x	x	1	1	x	1	x	x	x
Present Time Frame	1	1	0	x	$\underline{D}$	0	$\underline{D}$	D	0

We now take the primary input coordinates 1, 2, 3, 4 and "uniformize" them in the following fashion:

XX11	1111
110X	1101

For all lines for which in one time frame the value is c = 1 or 0 and the other is x we change x to this common value (to avoid races and hazards). If we simulate the test we can verify that it is a valid test. This is one intuitive explanation of how the C-algorithm works. We now obtain the same test using a combinational iterative model. We obtain this model by first cutting the feedback loops to render the circuit acyclic. Then as many copies of this circuit are produced as is desired, one copy for each time frame; the pseudo-output of the ith cut line

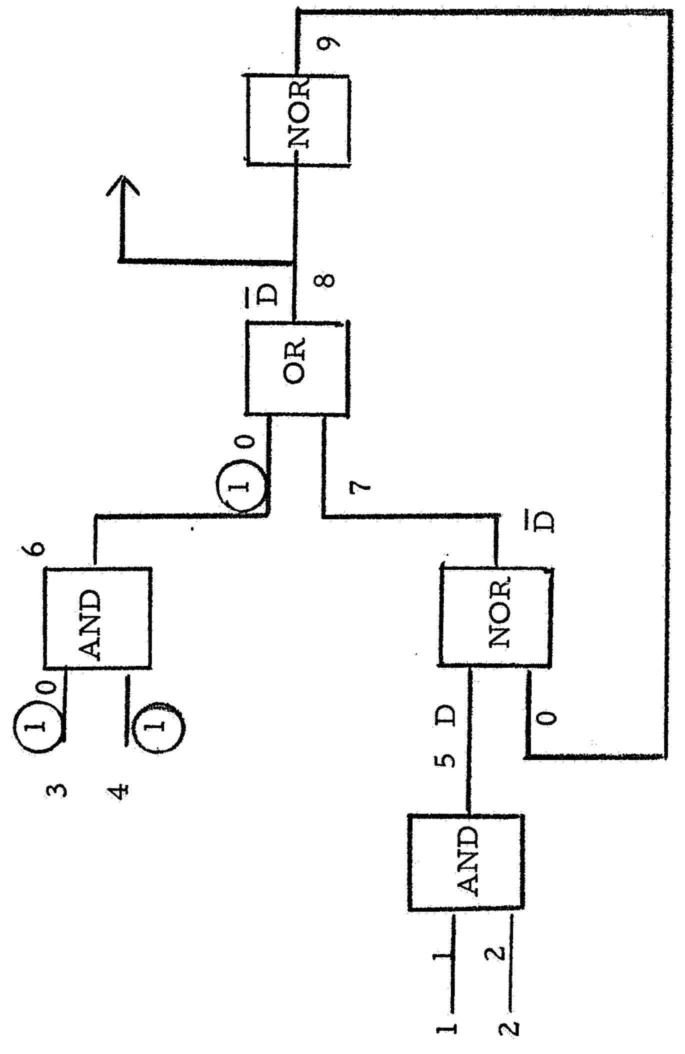


Figure 4 - Consistency Operation in a Sequential Circuit S.

in the jth time frame joined to the pseudo-input of the ith line in the (j + 1)st time frame.

Thus cutting the feedback loop labeled 9 in Fig. 3 and making two copies of it and joining as described, we get the combinational iterative model shown in Fig. 5. Here the nodes corresponding to the first time frame are labeled 1 through 9, with 9_0 the label for the pseudo-input from a prior time frame; the nodes corresponding to the next time frame are 1' through 9'.

We have thereby transformed the sequential or cyclic circuit diagnosis problem into a combinational or acyclic diagnosis problem with particular side conditions. One of these is that line 9_0 must remain at value x, that is, the test being developed cannot use any information from line 9_0. Furthermore the single failure of line 5 in the original circuit becomes in the iterative model a double failure of lines 5 and 5'. Also line 3 cannot be "read" as a PO.

A test is computed in this model in the following way: In order to generate the D on line 5' primary inputs 1' and 2' are set at value 1. To initiate the D-DRIVE, this D "passes through" NOR-block 7' by requiring that line 9 be given the value 0 (to be justified in the consistency part of the algorithm). Thus a $\overline{D}$ is attached to line 7' to go through block 8', which is a primary output so that the D-DRIVE ceases.

The consistency part of the algorithm then requires that line 6' set equal to 0 be justified and this is arbitrarily done by setting primary input line 3' equal to 0. It is then recorded that another choice could still be made to achieve this result. Next line 9 set equal to 0 must be justified. This is done by setting line 8 equal to 1 which in turn is justified by assigning line 6 equal to 1 which is justified by making both primary inputs 3 and 4 equal to 1, completing the test cube. We now have:

9_0	1	2	3	4	5	6	7	8	9
x	x	x	1	1	x	1	x	1	0

1'	2'	3'	4'	5'	6'	7'	8'	9'
1	1	0	X	D	0	$\overline{D}$	$\overline{D}$	$\overline{D}$

We then complete this test by "eliminating" the x's on the primary input patterns to 0's and 1's attempting to minimize changes from one time pattern to the next.

If we now reinterpret the obtained input pattern C_e, as a sequence:

```
1235
1111
1101
```

of two input patterns on S, we find the same test we have already obtained.

4. Formal Description of Models Used in C-Algorithm

We are given a Sequential circuit S, as described in section 1. By means of a program [DPR], we select out certain lines as feedback loops in a "natural" manner so that their cutting renders the circuit acyclic. The terminus of each cut line pointing toward the cut is termed a pseudo-output; those leading away, pseudo-inputs; the sets of these lines are denoted respectively SO and SI. Thus a combinational circuit C is obtained as shown on the two top right diagrams of Fig. 7.

The primary inputs PI of S itself and pseudo-inputs SI constitute the primary inputs of C. The primary outputs PO of S itself and the pseudo-outputs SO constitute the primary outputs of C. Now if we wish to compute a test of length n we make n copies of C and join the appropriate pseudo-inputs of the ith to the pseudo-outputs of the i + 1) st, to form the iterative combinational circuit C_e as shown in the center of Fig. 7.

This model as can easily be seen is isomorphic with the sequential circuit S' shown at the bottom of Fig. 5 wherein a unit delay Δt is inserted in each of the specially selected feedback loops with no delay associated with the combinational part of the circuit and the n input changes are separated in time by the amount Δt. Here is associated a sequence of n primary input patterns and n corresponding output patterns produced by a sequential operation of S'.

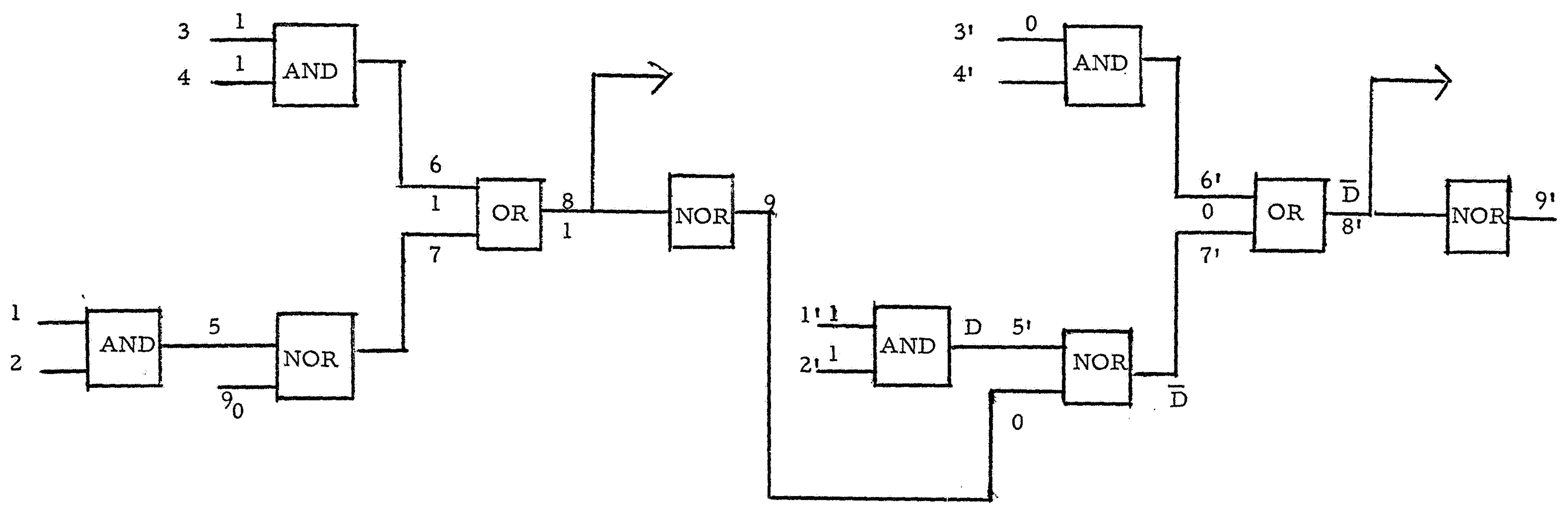


Figure 5 - Obtaining the Same Test Using a Combinational Iterative Model C_e.

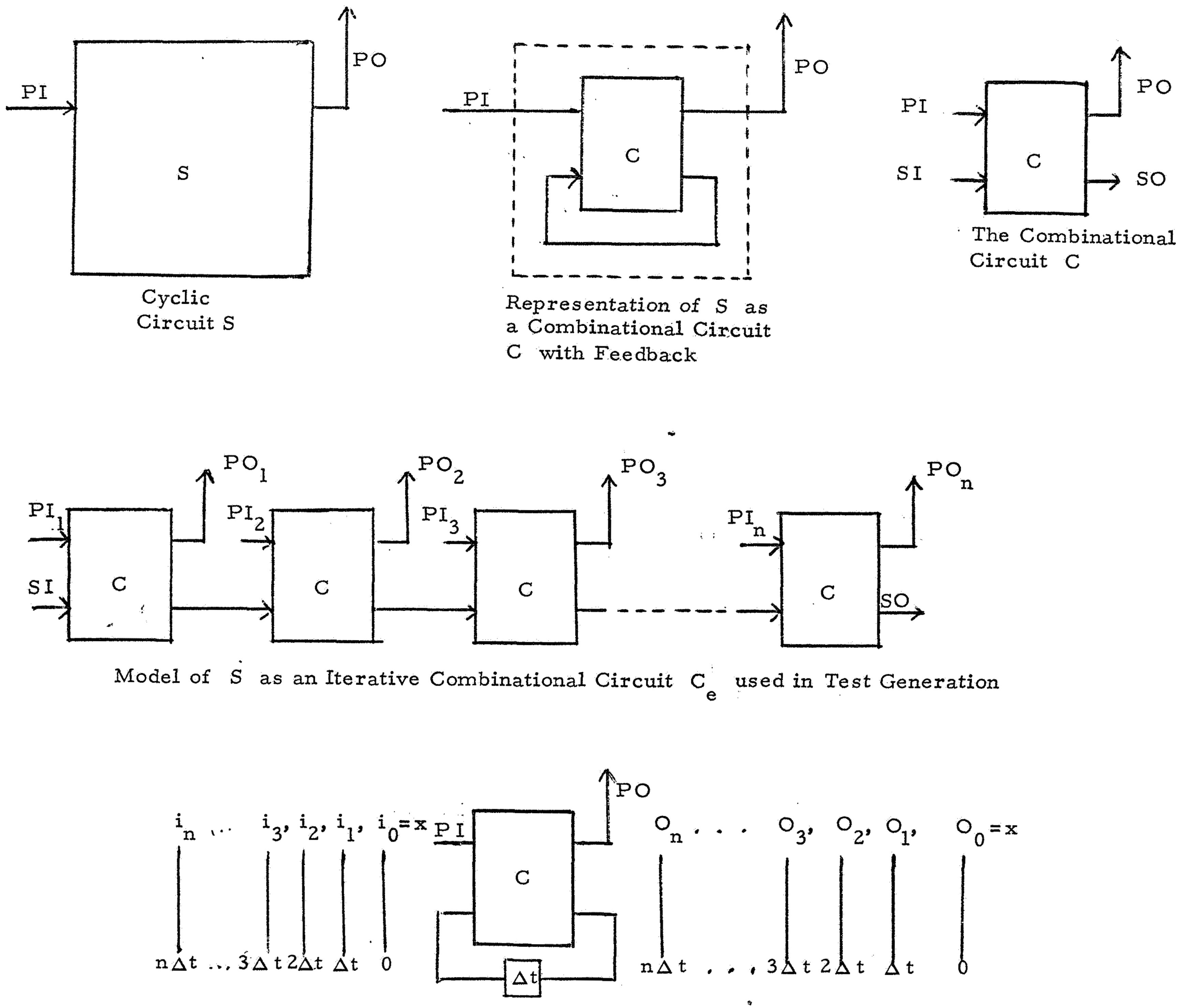
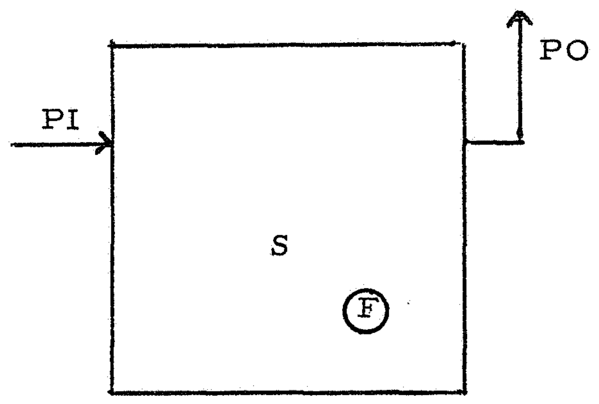

PI
S
PO
Cyclic Circuit S
PI
C
PO
Representation of S as a Combinational Circuit C with Feedback
PI
SI
C
PO
SO
The Combinational Circuit C
PI_1
SI
C
PO_1
PI_2
C
PO_2
PI_3
C
PO_3
PI_n
C
PO_n
SO
Model of S as an Iterative Combinational Circuit C_e used in Test Generation
i_n ... i_3, i_2, i_1, i_0=x
$n\Delta t$ ..., $3\Delta t$ $2\Delta t$ Δt 0
PI
C
PO
Δt
O_n . . . O_3, O_2, O_1, O_0=x
$n\Delta t$ . . . $3\Delta t$ $2\Delta t$ Δt 0

Figure 7 - Sequential Circuit with Delays Equivalent to the Model of S.

In C_e there will be a failure in every copy (assuming that the failure is not transient). Thus to a single failure in S corresponds a multiple failure in the iterative acyclic model C_e. Fig. 6 depicts this correspondence, drawing attention also to the fact that no values 0 or 1 may be assigned to the pseudo-inputs SI of the first copy of C in C_e. In other words in developing a test we can never assume a particular value on the pseudo-inputs for the first copy of the circuit in C_e and likewise no value can be encorporated from the pseudo-outputs of the last one.



Sequential Circuit S
With Single Failure F

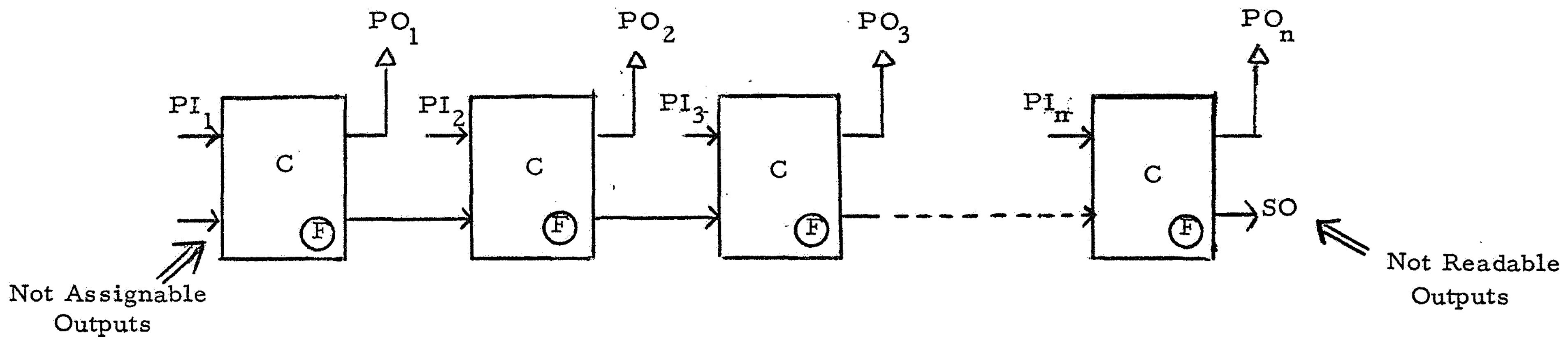


Figure 6 - Combinational Iterative Circuit with Multiple Failure F F F... F .

IV. F-Notational Description of the C-Algorithm.

In this section we describe the C-algorithm in (a semi-informal version of) the F-notation F/I. The description is given in a series of levels--the higher (earlier) levels giving the overall structure of the algorithm, the lower (later) levels giving increasing amounts of detail. The description is not given in complete detail, that is, we do not specify how the data objects are to be represented nor do we precisely define the primitive operations. What we have attempted to do is to carry the description down to the level just above that at which specific choices of notation and implementation must be made. That is, we have tried to give the basic structure of the algorithm (what must be done) while avoiding fixing on a particular way of doing something that might be done in several ways. We do not claim, of course, that the description given here is the only possible description of the algorithm in F/1, but only that we have largely avoided arbitrary detail. Perhaps the most significant example of the use of high level description in this presentation in order to avoid arbitrary detail is the handling of the "tree search" part of the algorithm. Rather than specify precisely how the tree search is to be handled we have merely indicated one, what information is relevant to the tree search and two, when and where in the algorithm this information is employed to make decisions as to what to do next. This is done largely by means of naming a "repository" for the information (denoted variously by the variables: bl, bl', blone, newbl, etc. as appropriate) and by indicating the operations which feed, update, and extract data from this repository.

The main meat of the algorithm is given in the first three levels of description (and primarily in the third level). However one critical algorithm, dalg, which appears for the first time at the third level is further described at the fourth and fifth levels.

The written description which accompanies the F-algorithms is purely informal and is presented in order to explicate the intention behind the formal F-algorithms. In principle all that is required to make the algorithm completely rigorous is a precise definition for each of the primitive-algorithms (or functions) and the primitive arguments. Our claim is that that could be done in many different, reasonably straightforward, ways to produce a complete algorithm which would indeed be an algorithm for solving the sequential diagnosis problem discussed earlier in this report.

1st Level Description:

```
[ t = calg(s, f, pmax)
($v_1$ = form-acyclic-circuit(s, f, pmax))
( t = find-test ($v_1$))]
```

The above formula says that the algorithm calg has three arguments: a sequential (cyclic) circuit s, a failure f in that circuit, and a bound pmax on the number of time frames which may be employed in finding the test. The first statement says that these arguments are employed to form a acyclic circuit from s (by cutting feedback lines) and the second statement says that this result is then to be used to find a test. This is, of course, a very high level description and as such indicates very little about the algorithm.

2nd Level Description:

```
[ t = form-acyclic-circuit(s, f, pmax) ≡
(ac = cut(s))
( t = (s, f, pmax, 0, 0, ac))]
[ t = find-test(s, f, pmax, p, q, ac) ≡
($v_1$ = form-acyclic-circuit-chain(s, f, pmax, p, q, ac))
($v_1$ = NO-ACC  |  t = NOTEST)
($v_2$ = drive-to-form-test ($v_1$))
($v_2$ = CANT-DRIVE |  t = NOTEST)
( t = simulate-test ($v_2$))]
```

This second level description consists of two formulas each explicating one of the two statements of the first level description.

The first formula describes the algorithm, form-acyclic-circuit. The first statement in this algorithm forms an acyclic circuit denoted ac from the sequential circuit by employing an algorithm called cut (which cuts the feedback lines in s). [We do not give any details on the algorithm cut since many different choices of algorithm would suffice.] The second statement indicates the output of this algorithm--namely it says that the result v_1 indicated in the first level description will be a sixtuple as indicated (and from the first level description we know that this sixtuple will be the input to the subalgorithm find-test).

The second formula describes the algorithm find-test. The first step (first statement) is to form a chain of p + q + 1 copies of the acyclic circuit ac. If this cannot be done then this step will produce the result NO-ACC. If NO-ACC is produced then the overall result is that there is NO-TEST. However if the chain is produced then the algorithm drive-to-form-test is applied to it to produce a candidate for a test. If no test can be found then the output of this subalgorithm is CANT-DRIVE and the overall result of the formula is again NO-TEST. However if a candidate is found then it is checked by the sub-algorithm simulate-test to see that it is a valid test (exactly what is to be done if it is not a valid test is spelled out at the next level of description).

<u>3rd Level Description</u>:

```
[t = form-acyclic-circuit-chain (s, f, pmax, p, q, ac) ≡
    (p + q > pmax | t = NO-ACC)
    (acc = iterate (ac, p, q))
    (dcf = pdcf (acc, f, p, q))
    (bl = record (dcf))
    ( t = (s, f, pmax, p, q, ac, acc, bl))]

[t = drive-to-form-test (s, f, pmax, p, q, ac, acc, bl) ≡
    ((u, bl') = dalg (acc, bl))
    (u = NOTEST ∧ (p + q ≥ pmax) | t = CANT-DRIVE)
    ((p', q') = step (p, q, pmax))
    ( u = NOTEST ∧ (p + q < pmax) |
      t = find-test (s, f, pmax, p', q', ac))
    ( t = (s, f, pmax, p, q, ac, acc, u, bl')]

[t = simulate-test (s, f, pmax, p, q, ac, acc, u, bl') ≡
    ( v = sim (s, f, u))
    ( v = 1 | t = u)
    (w_1 = drive-to-form-test (s,f, pmax, p, q, ac, acc, bl')
    (w_1 = CANT-DRIVE | t = NOTEST)
    ( t = simulate-test (w_1))]
```

The third level description explicates the three subalgorithms employed in the description of the algorithm <u>find-test</u>.

The first formula describes the algorithm, form-acyclic-circuit-chain. The description is given completely in terms of primitives, that is, functions and algorithms which are not further explicated formally within F/I in this treatment. Interpreting the formula we see that if $p + q > pmax$ then the result is NO-ACC (since this means that the bound, pmax, on the number of time frames has been exceeded). If $p + q \leq pmax$ then we go on to the second statement which says that we form the acyclic circuit chain, acc, using the primitive-algorithm iterate which connects together $p + q + 1$ copies of the acyclic circuit ac. The next step is to apply the primitive-algorithm pdcf which is intended to develop the initial information, dcf, needed to test for the failure f in acc, in particular this algorithm finds the d-cube of failure associated with f in acc. Finally the last statement records this result in bl which, as mentioned earlier, serves in this presentation as a "repository" of all information needed to control the flow of the algorithms. The result of the algorithm is then the indicated 8-tuple.

The second formula in the third level description describes the algorithm drive-to-form-test. The first statement in this algorithm employs the algorithm dalg (described at the fourth and fifth levels in this treatment) to try to form a test u and to record the steps taken in that process as bl'. If no test can be found then dalg will put out the result

NOTEST. If this happens and the bound, *pmax*, on the number of time frames has been reached then the result of the algorithm is *CANT-DRIVE* which signifies that no test can be found with the given bound. If dalg produces *NOTEST* but the bound *pmax* has not yet been reached then new values for *p* and *q* are chosen by means of the primitive algorithm *step*, and the (second level) algorithm *find-test* is employed using these new values (denoted by *p*' and *q*') of *p* and *q*. While finally, if the value of *dalg* is not of the form (*NOTEST*, *bl*') then the result of the algorithm is the indicated 9-tuple.

The final formula in the third level description describes the algorithm called *simulate-test*. The first step in this algorithm is to check out the proposed test *u* (developed by *drive-to-form-test*) by means of the primitive algorithm *sim*. If *u* is a valid test for failure *f* in sequential circuit *s* then *sim* (s,f,u) takes the value 1. When this happens the result of the algorithm is thus validated, test *u*. If the proposed test *u* is not valid the algorithm then iterates to form another proposed test using the above described algorithm *drive-to-form-test* and itself.

4th Level Description:

```
[ (u, bl') = dalg (acc, bl) ≡
            |(bdrv?(bl) = 1 ∧ conbk? (bl) = 1
            | (u, bl' ) = consistency-drive (tc(bl), acc,
                              conbk(bl)))
             (bdrv?(bl) = 1 ∧ conbk?(bl) = 0 | (u,bl')
                           = (NOTEST, bl))
             ($v_1$ = select-actives (acc, bl))
             ($v_1$ = NOTEST |   (u, bl') = (NOTEST, bl))
             ($v_2$ = drive-to-test-cube ($v_1$))
             ($v_2$ = NOTEST | (u, bl) = (NOTEST, bl))
             ($v_2$ = (NODRIVE, bl") | (u, bl')
                  = dalg (acc, bl"))
             ($v_3$ = consistency-drive ($v_2$)) ]
```

The fourth level description given here presents a more detailed explication of the algorithm dalg. It will be noted from looking at the higher level descriptions that this algorithm can be entered under a variety of conditions. In particular the dalg algorithm may be entered for the first time, or after it has already proposed a number of tests which turned out not to be valid. Depending on what has happened before, the algorithm does different things. Thus the first two statements in the description of dalg are checks on the repository bl to see what it should do. The function brdv? in the first statement checks to see if the algorithm was last performing a "B-drive" (finding appropriate inputs to force the driving of a given d to the outputs) if this is the case and if there are still alternative ways to try doing the "B-drive" as indicated by <u>conbk?(bl) = 1</u> then the <u>consistency-drive</u> algorithm is employed to try these out using the appropriate information from the repository <u>bl</u> as selected by the primitive-algorithm <u>conbk</u>. On the other hand, if the algorithm was last performing a "B-drive" but there are not alternative left to investigate (indicated by <u>conbk?(bl) = 0)</u> then the result is (NOTEST, bl) indicating that no test can be produced. If though the algorithm was not last doing a "B-drive" but is rather doing a "D-drive" (trying to drive a d signal from the failure to an output) then it will go immediately to the third statement in the algorithm. Then, as shown by the fifth level description given below, the algorithm will select an active line on which to attempt to drive the d signals using the algorithm <u>select-actives</u>. If there are no more active lines available the result will be NOTEST, otherwise the algorithm <u>drive-to-test-cube</u> is employed to drive the d from this active line to an output thus forming a test cube. The performance of this subalgorithm will result either in the result NOTEST indicating that no test can be produced, or in the result NO-DRIVE in which case the contents of the repository <u>bl</u> are suitably updated and <u>dalg</u> is applied to the resulting new arguments (this happens when there is no way to drive from the particular active line chosen and a new active line must be selected), or finally, the drive may be completely successful in which case a test-cube is indeed produced and then the consistency-drive algorithm is employed to perform the "B-drive" operation.

5th Level Description:

```
[ t = select-actives (acc, bl) ≡
        (actives = actives (acc, bl))
        (actives = ∅ ∧ dbk?(bl) = 1
                |t = select-actives (acc, dbk(bl)))
        (actives - ∅ ∧ dbk?(bl) = 0
                |t = NOTEST)
        (line = select-element-of (actives))
        (bl"  = update-actives-in-bl (bl, actives, line)
        (   t = (acc, bl", line))]
[ t = drive-to-test-cube (acc, bl, line) ≡
        (ppdc = produce-prim-d-cubes (acc, line, bl))
        (ppdc = ∅ | t = (NODRIVE, bl))
        (pdc  = select-an-element-of (ppdc))
        (newbl = update-ppdc-in-bl (bl, ppdc, pdc, line))
        (tc' = tc (bl) ⊓ pdc)
        (tc' = ∅ | t = drive-to-test-cube (acc, newbl, line))
        (newerbl = update-test-cube-in-bl (newbl, tc', line))
        (tc'⊓po (acc) ≠ ∅ | t = (tc', acc, newerbl)
        (t = (NODRIVE, newerbl)
[ (u, bl') = consistency-driv (tc, acc, bl) ≡
      (bltwo = set-bdrv-equal-one(bl))
      (b = produce-b(tc, acc, bltwo))
      (b ⊂ pi(acc)  | (u, bl') = (tc, bltwo))
      (inline = select-line-from(b - pi (acc)))
      (new-b = (b - inline) ∪ predecessors(inline, acc))
      (blthree = update-b-in-bl(new-b, bltwo)
      ((u,bl) = drive-tc(tc, acc, blthree, inline))]
[ (u, bl') = drive-tc(tc, acc, bl, inline) ≡
      (set-psc = form-primitive-singular-cubes(inline, acc, bl))
      (set-psc = ∅  ∧ conbk?(bl) = 0 | (u, bl) = dalg(acc, reset(bl)))
      (set-psc = ∅ | dalg(acc, conbk(bl))
      (new-set-psc = (set-psc) - psc)
      (newbl = update-bl-with-psc(bl, psc, new-set-psc))
      (newtc = tc ⊓ psc )
      (newtc = ∅ | (u, bl) = drive-tc(tc, acc, newbl, inline))
      (newer-bl = update-bl-with-new-tc(newbl, new-tc))
     ((u, bl) = consistency-drive(new-tc, acc, newer-bl))
```

The fifth level description describes the algorithms <u>select-actives</u>, <u>drive-to-test-cube</u>, and <u>consistency-drive</u> in terms of primitive-algorithms and algorithms already described at a higher level. Note that the algorithm <u>consistency-drive</u> is described using two formulas--this allows us to go into greater detail without resorting to an additional level of description.

The first formula describes the algorithm select-actives. The first step is to perform the primitive-algorithm actives which is intended to find the active lines (outputs of circuit elements which have d's on one or more inputs) on which the algorithm has not as yet tried to drive the d. (The reason that bl is an argument of actives is that bl contains the information as to which lines have already been tried). If there are no active lines that have not already been tried, (so that actives = ∅) and if there is a way to "back up" the search (as indicated by dbk?(bl) = 1) then the algorithm "backs up" and computes select-actives(acc, dbk(bl)). If actives = ∅ and it is not possible to "back up" then the result is NOTEST. However, if there are active lines still to be tried then one of these lines is selected by the primitive-algorithm select-element-of(actives) and the repository bl is updated by the primitive-algorithm update-actives-in-bl to indicate that this line has been tried and the final output is then the indicated triple.

The algorithm drive-to-test-cube employs the output triple of the algorithm select-actives to drive a d to a primary-output of acc along the line line. The first step is to produce the primitive d cubes associated with the line line that have not already been tried. This is done with the primitive-algorithm produce-prim-d-cubes (again, bl contains the information as to which d-cubes have already been tried). If all the primitive d-cubes associated with line have already been tried (as indicated by ppdc = ∅) then the result is (NODRIVE, bl); otherwise some primitive d-cube is selected using the primitive-algorithm select-an-element-of(ppdc). Next the repository bl is updated to record this selection using the primitive-algorithm update-ppdc-in-bl. Then the selected d-cube is intersected with the latest (partial) test-cube tc(bl) recorded in bl . (Note, the recording of tc in bl was initiated by the statement (bl = record(dcf)) in the third level description of form-accyclic-circuit-chain.) If this intersection (interface) is empty (i.e., if tc' = ∅) then the algorithm tries again using newbl in place of bl. However if tc' ≠ ∅ then this new (partial) test-cube is stored in the repository using the primitive-algorithm update-test-cube-in-bl. The algorithm then checks to see if the just produced test-cube drives all the way to the primary outputs po. If this is the case then the result is the indicated triple (which becomes the input to consistency-drive); while if it does not then the result is (NODRIVE, newerbl) which, (as indicated by the fourth level description) will cause reentry of dalg with the repository updated so as to employ this new (partial) test-cube.

The purpose of the algorithm consistency-drive is, given a test-cube tc, to attempt to find a way to set the inputs of the circuit acc so as to realize this test-cube. In this algorithm we again use bl (and variants thereon) to designate the repository for the information build up in the "tree search" for a way to set the inputs. The first step in the algorithm is to record, in bl, that the algorithm is doing the consistency-drive (see beginning of fourth level description of dalg). This is done by means of a primitive-algorithm called set-bdrv-equal-one. The next step is to produce b, the collection of lines which have signals specified on them but whose predecessors must now have signals specified by the algorithm. If all the elements of b are primary inputs (i.e., if b $\subset$ pi(acc) then we are done and (u, bl') = (tc, bltwo). Otherwise the algorithm picks an element of b which is not a primary input (in order to attempt to set the correct signals on its predecessors to bring it to the specified state). When this line, called inline, is chosen we then update the repository to record this selection. Finally we apply the algorithm drive-tc in order to set the values on the inputs of the element which drives the line inline.

The algorithm drive-tc starts by producing the set of primitive singular cubes associated with the line in-line what have not already been tried. If this set is empty (if they have all been tried) and there is no way to continue the "tree search" (as indicated by conbk?(bl) = 0) then this means that there is no way to set the primary inputs so as to realize the given test-cube. When this happens the repository is reset so as to start the algorithm dalg over again to find a new candidate for a test cube (in particular it is at this point that the repository is set so that bdrv?(bl) $\neq$ 1--see discussion of dalg). If, on the other hand, the set set-psc is empty but there are alternatives to try on the consistency drive then the primitive algorithm conbk(bl) "backs up" the data in the repository to the appropriate point and dalg is then employed (with bdrv? = 1) to continue the consistency drive. Finally, if the set set-psc is not empty then the primitive-algorithm select-element-of(set-psc) chooses an element of set-psc and updates the repository to indicate this selection. The selected element psc is then interfaced with the existing test-cube tc to attempt to form a new test-cube newtc specifying the signals on lines closer to the primary inputs. If the attempt is unsuccessful (if tc and psc are incompatible) we will get $tc \sqcap psc = \emptyset$ in which case drive-tc is applied to the new contents of the repository (in order to try the remaining members, if any, of set-psc). If the attempt is successful then the repository is updated to include the new test cube, newtc, and the algorithm applies consistency-drive in order to complete the process of providing input signals that will give rise to the desired test-cube.

Appendix A

An Algorithm and a Program for the Multiple-Output 2-Level Logic-Minimization Problem

By J. Paul Roth*, E. G. Wagner* and Leon S. Levy#
*IBM Research Center, Yorktown Heights, N. Y.
#IBM Research Center and University of Pennsylvania

Abstract: An algorithm and program is given for the problem of finding a minimum-cost multiple-output logic circuit of two levels which realizes a given function mapping strings of labelled bits into strings of labelled bits. This is a classical problem currently having application to implementation of logic circuits in large-scale integration. A notation and calculus is devised involving singular cubes, covers and complexes for efficient treatment of the problem. A succinct description of the algorithm itself is given in a newly defined functional notation, termed the F-notation. A computer program for this algorithm following the "F-description" has been written for the IBM System/360 APL interactive system.

1. Why the problem is technologically interesting. The problem of economical implementation of logic circuits having two levels of logic has had continuing application in the design of data processing machines ranging in complexity from EAM machines through the 1401 to STRETCH [ERW 61]. For space applications see [RP 69]. Cost minimization for multiple-outputs is an outstanding problem in logic. Muller's encoding

[54] of the multiple output problem into an "equivalent" single-output problem served to allow the formulation [M 65] RP 69] of useful approximations although handicapped by the required addition of abundances of don't-care cubes. Bartee's tag techniques [B 61] was another approach.

2. New Mathematics. A function G mapping labelled strings of 0's and 1's into labelled strings of 0's and 1's may be described as a "function table", listing all the pairs of such correspondences. More compact is our singular notation: here a function is described by an ensemble or singular cover or s-cover of singular cubes or s-cubes consisting of pairs of labelled strings of 0,1, x separated by a vertical line segment. We shall illustrate. Suppose that the input lables (variables) are 1, 2, 3, 4 and output lables a, b, c. Then the array on the left of Fig. 1 would define a function G in conventional fashion, while the array on the right would be an equivalent s-cover.

Fig. 1: A singular cover and its interpretation.

1 2 3 4	a b c
0 1 1 0	1 x 1
1 1 1 1	1 x 1
1 1 1 0	1 x 1
0 0 0 0	x 0 x

1 2 3 4	a b c
x 1 1 x	1 x 1
0 x 0 0	x 0 x

Work performed for the Jet Propulsion Laboratory, California Institute of Technology, sponsored by the National Aeronautics and Space Administration under Contract NAS-7-100.

The left portion of a singular cube is termed the *input part* and the right, the *output part*. Thus

$$\begin{array}{cccc|ccc} 1 & 2 & 3 & 4 & a & b & c \\ x & 1 & 1 & x & 1 & x & 1 \end{array}$$

specifies that when conditions or variables 2 and 3 are 1, the value of the output variables a and c are 1, *regardless* of the values of 1 and 4; this cube makes no specification for the value of variable b (since it has the value x) for these input conditions. A similar interpretation holds for the other singular cube given. Those input conditions not defined by the singular cover are termed *don't care conditions*. In our model, however, the output part of all s-cubes will consist of 1's and x's alone, with the don't-care conditions also explicitly listed in this form. Thus a function F is defined by s-covers C and D of the *care* and *don't care* conditions respectively.

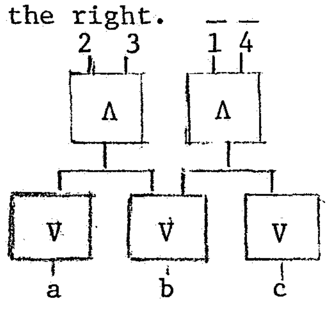
We say that s-cube $\alpha = a_1 \ldots a_r | b_1 \ldots b_s$ *contains* s-cube $\beta = c_1 \ldots c_r | d_1 \ldots d_s$ if $a_i = c_i$ or x and $d_j = c_j$ or x; we denote this condition $\alpha \sqsubset \beta$. We also say that α is a *face* of β. Thus $(01x1|1xx) \sqsubset (01xx|1x1)$. A *vertex* consists of a singular cube $a_1 \ldots a_r | b_1 \ldots b_s$ wherein all a_i's are 0 or 1 and all but exactly one b_j is x, which one is 1. An s-cube defines the set of all vertices contained in any of its cubes. s-cover A is said to *contain* s-cover B if all vertices of B are contained in cubes of A. If A contains B we write $B \sqsubset A$. Covers are partially ordered by the relation $\sqsubset$. Let F be a function defined by s-covers C and D of care and don't-care conditions; we shall say that s-cover E realizes F if $C \sqsubseteq E \sqsubseteq C \cup D$. Let a cost be associated with each s-cover, of a very general type, (cf. [RW 69]). An example would be any linear function of the number of inputs and outputs not equal to x.

Problem: Given F find a realization of minimum cost. E.g., if F is given by the s-covers C,D shown on the left of Fig. 2 then M is a minimum realization.

Fig. 2: A problem, its minimum solution M a logic circuit of M is shown on the right.

	1	2	3	4	a	b	c
C	x	1	1	x	1	1	x
	0	x	1	0	x	1	1
D	0	0	0	0	1	1	1

1	2	3	4	a	b	c	
x	1	1	x	1	1	x	
0	x	x	0	x	1	1	M



The multiple-output extraction algorithm MX is a method for solving this problem. To define MX the following definitions are made. As we have seen, each cover C defines an ensemble V of vertices. The cover C also defines an ensemble of cubes termed a singular complex or s-complex K being the (unique) s-cover K, maximal under set-theoretic inclusion, whose ensemble of vertices is V. Clearly, $V \sqsubseteq C \sqsubseteq K$. An s-cube of an s-complex K is said to be prime if no other cube of K contains it as a face.

MX is described in an algorithmic language called F-notation [RW 69]. This describes MX as a recursively linked ensemble of F-functions. One of the first functions in MX is the F-function #-algorithm, #alg $(C \cup D)$, which computes the set Z of all prime cubes of the complex K defined by s-cover $C \cup D$. A prime cube e is said to be an extremal if it covers a vertex of C (a care vertex) which no other cube of Z does. It can be shown that every minimum M contains e or one of its faces. A second F-function $\underline{E}$ (C,D,S), where S is a partial solution initially empty (ϕ), computes the set E of extremals.

If $E \neq \phi$, the "distinguished" faces F of E are extracted from K and added to S and D, to form a new complex K # F and complex defined by s-covers C-F and $D \cup F$. F-function $\underline{E}(C-F, D \cup F, S \cup F)$ is then executed again, with the new arguments shown. This time a "less-than" operation is performed: given s-cubes u and v we say $u < v$ if, roughly speaking, (cf [RW 69]) cost (u) $\geq$ cost (v) and the vertices of u are contained in those of $v \cup D$. It can be shown that where u, v belong to K, and $u < v$, there is a minimum not containing u. Thus u is dropped from consideration. Thus, in the next and subsequent executions of $\underline{E}$, Z is "appropriately" reduced from its non-maximal cubes under < to form Z = < (D,S,Z).

Hence a new extraction problem is defined and this process is repeated either until a solution is obtained or until no new extremals appear. In this case a branching function $\underline{B}$ is executed and a minimum finally obtained, possibly through recursive execution of B. For fuller explanation see [RW 69].

3. APL-program. A computer program for MX following the "F-description", has been written for the IBM System/360 APL interactive system [FI 68] using the Iverson notation. This program is an interactive one providing typewriter printout of intermediate solutions. During execution of B, for example, manual selection may speed convergence.

Fig. 3 - A geometrically represented problem.

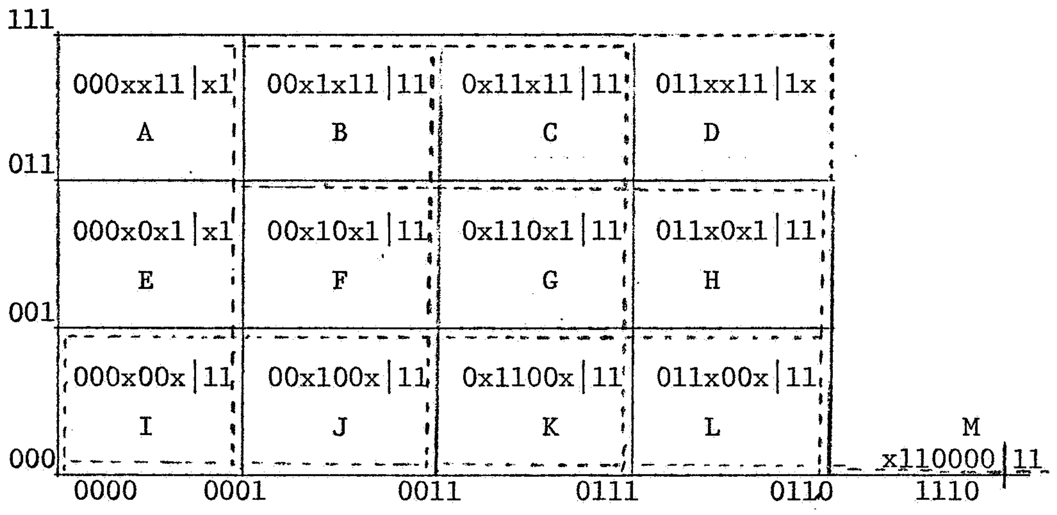


Fig. 4 - A minimum solution.

A	0	0	0	x	x	1	1	x	1
δB	0	0	x	1	x	1	1	1	x
δC	0	x	1	1	x	1	1	x	1
D	0	1	1	x	x	1	1	1	x
H	0	1	1	x	0	x	1	1	1
I	0	0	0	x	0	0	x	1	1
K	0	x	1	1	0	0	x	1	1
M	x	1	1	0	0	0	0	1	1

Appendix B

In this section of the appendix we offer the present version of the formal syntax for abstract F-notations. In this context "abstract" means that no specific semantics are given (that is, the set R, the relation =, and the function are not specified). In the forthcoming report on F-notations we shall present the abstract semantics for such abstract F-notations and we will describe a specific F-notation, the F-notation F/1, in detail.

Formal Syntax of F-Notations

An F-notation is specified by giving the following sets and mappings:

$\underset{\sim}{F}$	an infinite set of symbols called the set of <u>function-symbols</u>.
α:	$\underset{\sim}{F} \to \underset{\sim}{N}$ ($\underset{\sim}{N}$ = the non-negative integers) called the <u>arity-function</u>; if $F \varepsilon \underset{\sim}{F}$ then $\alpha(F)$ is called the arity of F and F is said to be of arity $\alpha(F)$.
$\underset{\sim}{R}$	an infinite set of symbols called the set of <u>data-representatives</u>.
=	an equivalence relation on $\underset{\sim}{R}$, called <u>equals</u>.
$\underset{\sim}{D}$	the set of equivalence classes of $\underset{\sim}{R}$ as determined by =, called the set of <u>data-objects</u>.
$\underset{\sim}{P}$	a finite subset of F called the set of <u>primitive-function-symbols</u>.

σ a function which assigns to each P_i in P a total function from $\underset{\sim}{D}^{\alpha(P_i)}$ into $\underset{\sim}{D}$ (for $\alpha(P_i) = 0$ $\sigma(P_i)$ is some particular element of D), σ is called the primitive-function-assignment.

V an infinite set of symbols called the set of variables.

Where:

1. The sets $\underset{\sim}{F}$, $\underset{\sim}{R}$, and $\underset{\sim}{V}$ are disjoint.

2. For each $n \geq 0$ the set of elements of $\underset{\sim}{F}$ of arity n is infinite.

(Note: For all "practical purposes", "infinite" can be taken to mean, "very large".)

We define the set of F-expressions as follows:

1. If $v \varepsilon V$ then $v \varepsilon E$.

2. If $F \varepsilon F$ and $\alpha(F) = 0$ then $F \varepsilon E$ (we say then that F is an F-constant or a 0-ary free-F-expression.

3. If $r \varepsilon R$ then $r \varepsilon E$.

4. If $F \varepsilon F$ and $\alpha(F) = n > 0$ and $v_1,\ldots,v_n \varepsilon V$ then $F(v_1,\ldots,v_n) \varepsilon E$ (we say then that $F(v_1,\ldots,v_n)$ is an n-ary free-F-expression).

5. If $F \varepsilon F$ and $\alpha(F) = n > 0$ and $x_1,\ldots,x_n \varepsilon E$ then $F(x_1,\ldots,x_n) \varepsilon E$.

We define the set S of F-statements as follows:

1. If $v \in \underset{\sim}{V}$ and $e \in \underset{\sim}{E}$ then $(v = e)$ is an <u>execution-statement</u>.

2. If e_1, e_2 and $e_3 \in \underset{\sim}{E}$ and $v \in \underset{\sim}{V}$ then

   $(e_1 = e_2 | v = e_3)$

   and

   $(e_1 \neq e_2 | v - e_3)$

   are <u>conditional-statements</u>.

3. If S is either an execution-statement or a conditional-statement then s is an <u>F-statement</u>.

Given an F-expression $e \in E$ let $V(e)$ denote the set of elements of V occurring in e. Given an F-statement s let $V(s)$ denote the set of elements of $\underset{\sim}{V}$ occurring in s. Given a string $\rho = s_1 \ldots s_n$ of F-statements let $V(\rho) = \cup_{l=1}^{n} V(s_i)$.

We define the set of S-formulas as follows:

Let e be either an F-constant $F \notin P$, or a free-F-expression $F(v_1, \ldots, v_n)$ where, again $F \notin P$. Let Λ denote the null string. And let $v \in V$, $v \notin V(e)$.

1. $(v = e \equiv \Lambda$ is an S-formula.

2. If $(v = e \equiv \rho\ )$ is an S-formula and $(w = e_1)$ is an F-statement where $w \notin \{v\} \cup V(e) \cup V(\rho)$ but $v(e_1) \subset V(e) \cup (V(\rho) - \{v\}\ )$ then $(v = e \quad \rho \ (w = e_1))$ is an S-formula.

3. If $(v = e \equiv \rho)$ is an S-formula and

$(e_1 = e_2 | v = e_3)$ [resp. $(e_1 \neq e_2 | v = e_3)$ is an F-statement where

$$V(e_1) \cup V(e_2) \cup V(e_3) \subset V(e) \cup (V(\rho) - \{v\})$$

then

$$(v = e \equiv \rho(e_1 = e_2 | v = e_3))$$

[resp. $(v = e \equiv \rho(e_1 \neq e_2 | v = e_3))$]

is an S-formula.

We define the set of F-formulas as follows:

If $(v = e \equiv \rho)$ is an S-formula and $(v = e_1)$ is an F-statement such that

$$V(e_1) \subset V(e) \cup (V(\rho) - \{v\})$$

then

$$[\, v = e \equiv \rho(v = e_1)\,]$$

is an F-formula.

Given an F-formula $[v = e \equiv \rho]$ when $e = F$, an F-constant, or $e = F(v_1,\ldots,v_n)$ a free-F-expression, define

$$H([\, v = e \equiv \rho\,]) = \{F\}.$$

(Given an F-formula f we call $H(f)$ the head-symbol of f).

Given a string $f_1 \ldots f_n$, $n>1$, of F formulas, define

$$H(f_1 \ldots f_n) = H(f_1 \ldots f_{n-1}) \cup H(f_n).$$

By an F-algorithm we mean a string of F-formulas $f_1 \ldots f_n, n \geq 1$, such that, for $k = 1,\ldots,n-1$,

$$H(f_1 \ldots f_n) \cap H(f_{n+1}) = \phi.$$

REFERENCES

[M 54] Muller, Application of Boolean Algebra to Switching Circuit Design and Error Detection, IRE Trans. on Electronic Computers, Vol. EC-3, No. 33, pp. 6-12.

[B 61] Bartee, Computer Design of Multiple Output Logical Networks, IRE Trans. on Elec. Computers, Vol. EC-10, pp. 21-30.

[ERW 61] Ewing, Roth, Wagner, Algorithms for Logical Design AIEE Communications and Electronics, No. 56, Sept. 1961, pp. 450-458.

[M 65] Miller, Switching Theory, Vol. I, Combinational Circuits, John Wiley and Sons, Inc., N. Y. 1965.

[FI 68] Falkoff, Iverson, APL 360 User's Manual, International Business Machines Corporation, Yorktown Heights, N. Y. 10598.

[RP 69] Roth, Perlman, Space Applications of a Minimization Algorithm, IEEE Trans. on Aerospace and Electronic Systems. Submitted for publication.

[RW 69] Roth, Wagner, A Calculus and an Algorithm for a Logic Minimization Problem together with an Algorithmic Notation, IBMJ of R and D., also published as RC 2280.