A PROGRAM FOR EXACT SYNTHESIS OF THREE-LEVEL NAND NETWORKS

Marek A. Perkowski, Jiuling Liu *,

Department of Electrical Engineering, Portland State University,
P.O.Box 751, Portland, Oregon 97207,
tel: (503) 725-3806x23 (office)

* *Mr. Liu's research was partially funded by the IEEE/ACM Design Automation Award.*

ABSTRACT

The paper describes a program for exact minimization of three-level combinational functions from NAND (NOR) gates. This algorithm generalizes the well-known approaches of TANT synthesis in the following ways: the function is multi-output, it includes don't cares, any subset of variables can be available in only complemented form, or in both affirmative and complemented forms. The number of PP-implicants that can be used for exact minimum solution is reduced as a result of proving some theorems.

1. INTRODUCTION

Currently, the synthesis of *PLA circuits* (two level NOR/NOR structures) is well understood and several sophisticated and efficient algorithms for this task exist, including Espresso [2,29,30], Mini [12], Umini [4], McBoole, and other. Also, synthesis of *multi-level networks* composed of various kinds of logic gates becomes quickly a popular research area in which good and practically applicable results have been obtained [3,7,32,22]. Program MIS II from U.C. Berkeley is widely used for optimization of such networks. However, our own experience with MIS II is that in many cases the optimal result, that can be found using a more sophisticated hand design method, is not generated, even by trying various design scripts and for relatively small circuits like single cells of iterative networks. Moreover, MIS II cannot be used for the synthesis of asynchronous Finite State Machines since it can not remove static hazards from the circuit.

It was proven in a recent paper by Tsutomu Sasao [33] that *three levels of logic are enough to produce an optimal network for "most (statistically) of Boolean functions".* This important result should, in our opinion, *stimulate renewed interest in the design of circuits with three levels.* One of the advantages of such circuits is that the theory of their design is *quite close to the PLA minimization theory,* so that all concepts and efficient algorithms recently introduced for two-level logic and used in the mentioned above programs can be easily modified for the three-level network synthesis as well. Next, it is a common practical experience in modern integrated VLSI design automation systems that, because of restricted abilities of macro-cell generators and placement/routing programs, the logic minimizers are needed for *various logic design styles,* including those other than PLA.

Three level networks (TLN) have the advantage that the TLN design for function *f* can never be worse than the corresponding PLA in terms of the number of gates and connections (gate inputs). Usually, the TLN design is better. It is faster and it also permits for better incorporating of *fan-in and fan-out constraints* than standard cell realizations of two-level logic corresponding to PLA. Three level networks are important from the speed optimization point of view, since three levels is the minimum number of logic levels necessary to realize an arbitrary logic function from inclusive gates (even PLA has three levels, including inverters). The algorithms for hazardless synthesis of such networks have been also proposed which makes them useful for the design of asynchronous state machines.

TLN networks that will be introduced in this paper are generalizations of the *Three level And-Not Networks with True inputs (TANT)* of McCluskey and Gimpel [16,10]. Several algorithms to minimize TANT networks have been published, and some of them have been realized as computer programs. The relevant papers by Mc Cluskey [16], Gimpel [10], Kulpa [14], Chakrabarti [15], Choudhury [6], Koh [13], Lee [15], Frackowiak [8], Vink [34,35], and Perkowski [18,19] are available. A closely related topic of negative gate network minimization is presented in [20,21]. In those papers only [35,18] deal with incompletely specified functions, only [18] discusses multioutput functions, and only [35,18] discuss the case in which certain variable inputs are available in both forms.

The algorithms described in this paper are not only more general than the above algorithms but are also more efficient than the published exact algorithms. In Vink's algorithm, the size of the function is essentially expanded by increasing of the number of variables available in both forms. Also, additional permissible implicants (PP-implicants) are generated for those prime implicants that contain don't cares.

In this paper, we will present the program TLN-MINI for the multioutput, incompletely specified functions where any subset of input variables can be available in both affirmative and complemented forms, or only in one of them. The cost to be minimized is any weighted cost of a gate cost and connection cost, which is a more realistic assumption from the point of view of modern technologies than the gate cost used in most of the well-known papers.

The networks discussed below use NOR gates. When one wants to use NOR gates in the realization, the corresponding Boolean function *f* must be first transformed to its dual function f^D, to which next the presented methods can be used without modification. Finally, in the netlist created by TLN-MINI the symbols of NAND gates are replaced with the symbols of NOR gates.

It is assumed that a subset of variables is available in an affirmative form, the so called **positive variables.** Some subset of variables is available in a **complemented form,** the so called **negative variables. The available literals** are the literals that are available. For instance, if x is both a positive and negative variable then literals x and $\bar{x}$ are available. The set of available literals is defined by the user as the input data to the program.

To simplify the explanation, in the sequel, we will explain first the basic method, in which only positive variables are available and a function is single output and completely specified. Next generalizations of this method are presented.

2. THE ALGORITHM FOR TANT NETWORK MINIMIZATION FOR COMPLETELY SPECIFIED SINGLE OUTPUT FUNCTIONS WITHOUT COMPLEMENTED INPUTS

2.1. THE CONCEPTS, DEFINITIONS AND THEOREMS

The following example will illustrate how TANT optimization leads to network reduction.

Example 2.1.1. The function presented in the Karnaugh map from Fig. 2.1a can be minimized as the following Boolean expression: $F = \bar{a}\,\bar{c} + \bar{a}\,\bar{d} + b\,\bar{d}$. Its NAND-based PLA realization (so called **PLA-expression**) has 7 gates and 12 connections. We will say that the cost is (7, 12) - the **gate cost** is 7 and the **connection cost** is 12. In our algorithms the **total costs,** that can be any weighted sums of these two costs, will be minimized. The corresponding optimal expression for TANT network is $FT = \bar{a}\,\overline{cd} + b\,\bar{d}$, and can be realized by the network from Fig. 2.1b, which has one gate and one connection less than the previous realization.

One can observe from the above example that in TANT networks the first level realizes a logical sum, the second realizes a product and the third one the negation of the variables' product. TANT network minimization problem consists in finding the Boolean expression that minimizes the total cost. It means that the synthesis method should *minimize simultaneously* the second and third levels.

Definition 2.1. An **available kernel,** $K(P)$, (or simply a **kernel)** of a *product of literals P* is the product of those literals from *P* which belong to the set of **available literals.**

Example 2.1.2. Assuming that the available literals are a, b, $\bar{c}$, and $\bar{d}$, the available kernel of product $ab\bar{c}d$ is $ab\bar{c}$.

Definition 2.2. The kernel $k1$ includes kernel $k2$, which is denoted by $k1 \supseteq k2$, when all the minterms that are covered by $k2$, are covered by $k1$ as well. A set of kernels $K1$ includes another set of kernels $K2$, which is denoted by $K2 \subseteq K1$, when $(\forall\, k_i \in K2)(\exists\, k_j \in K1)\,[\,k_i \subseteq k_j]$ It is obvious that $k2 \subseteq k1$ if and only if all the literals contained in $k1$ are also contained in $k2$.

Example 2.1.3. Kernel abcde $\subseteq$ acd since the literals of the second kernel are all contained in the first kernel.

Definition 2.3. The **positive kernel** of a product of literals is the product of those literals which are positive.

Example 2.1.4. The positive kernel of $ab\bar{c}\,\bar{d}$ is ab.

The kernel is a *cube,* so that definitions of *cube calculus operations* [2,3,7,12,29,30,31], such as *sharp* (#), *absorption,* and *intersection* are applicable to it.

It can be seen from Definitions 2.1 and 2.3 that when all variables are available only in an *affirmative form, then the available kernel of the product of literals is also a positive kernel.*

Definition 2.4. A **permissible expression** is a Boolean expression of form $P = H\,\overline{T_1}\,\overline{T_2}\,\cdots\,\overline{T_m}$ where both H and T_i are available kernels. H is called the **head of permissible expression** and each T_i is called a **tail kernel** while each $\overline{T_i}$ is called a **tail factor.**

Definition 2.5. A **permissible implicant (PP-implicant) of function** *f* is a permissible expression which implies *f* (covers a subset of true minterms of *f*).

Example 2.1.5. A Boolean expression $b\,\bar{d}$ is the prime implicant of the function from Example 2.1.1, and $b\,\bar{d}$, $b\,\overline{bd}$, $b\,\overline{ad}\,\bar{c}$, $b\,\overline{ad}\,\overline{cd}$ are some of PP-implicants of this function.

Definition 2.5a. A head of a PP-implicant of function *f* is called the **second level group.** The set of all second level groups is denoted by H_f. A tail kernel of a PP-implicant is called the **third level group.** The set of all third level groups is denoted by T_f.

Example 2.1.6. For the TANT network of the function *f* from Example 2.1.1, realized in Fig. 2.1b, products *b*, and 1 are the second level groups, while products *a*, *b*, and *cd* are the third level groups. 1 is the head of the PP-implicant $\bar{a}\,\bar{c}$.

Definition 2.6. A **permissible realization** for function *f* is the inclusive sum of the set of PP-implicants which cover all minterms of the function. An ***optimal permissible realization* for function** *f*, denoted by $OPR(f)$, is such a permissible realization that its corresponding TANT network has the minimum total cost.

Definition 2.7. A **prime permissible implicant,** pp_f-implicant for short, is a permissible implicant that is not included in a prime implicant and if a tail factor is removed from it then the resulting expression is no longer a PP-implicant. The set of all pp_f-implicants is denoted by PP_f.

Example 2.1.7. For function *f* from Example 2.1.1:
$PP_f = \{\bar{a}\,\bar{c},\ \bar{a}\,\bar{d},\ \bar{a}\,\overline{cd},\ b\,\bar{d},\ b\,\bar{a}\,\overline{cd},\ b\,\overline{ad}\,\overline{cd},$
$b\,\overline{bd},\ b\,\overline{ab}\,\overline{cd},\ b\bar{a}\,\overline{bcd},\ b\,\overline{abd}\,\overline{cd},\ b\,\overline{ad}\,\overline{bcd},\ b\,\overline{abd}\,\overline{bcd}\}$.

Definition 2.8. A **principal PP-implicant,** pc_f-implicant for short, is such a pp_f-implicant that its tail kernels do not contain variables from its head. The set of all pc_f-implicants is denoted by PC_f.

Example 2.1.8. For function *f* from Example 2.1.1:
$PC_f = \{\bar{a}\,\bar{c},\ \bar{a}\,\bar{d},\ a\,\overline{cd},\ b\,\bar{d},\ b\,\bar{a}\,\overline{cd},\ b\,\overline{ad}\,\overline{cd}\}$.

Definition 2.9. A **maximal** pc_f**-implicant,** mp_f-implicant for short, is such a pc_f-implicant that is not included in other pc_f-implicants. The set of all mp_f-implicants is denoted by M_f.

Theorem 2.1. The tail kernels of a mp_f-implicant are included in all the tail kernels of pc_f-implicants included in this mp_f-implicant.

Example 2.1.9. For function *f* from Example 2.1.1: $M_f = \{\bar{a}\,\overline{cd},\ b\,\overline{ad}\,\overline{cd}\}$.

The tail kernels ad and cd of a mp_f-implicant $b\,\overline{ad}\,\overline{cd}$ are included in the tail kernel d of the pc_f-implicant $b\,\bar{d}$. They are also included in the tail kernels a and cd of $b\,\bar{a}\,\overline{cd}$.

Definition 2.10. An **augmented** pp_f**-implicant,** ap_f**-implicant** for short, is such a pp_f-implicant that it is not a pc_f-implicant. The set of all ap_f-implicants is denoted by AP_f.

Example 2.1.10. For function *f* from Example 2.1.1:
$AP_f = \{b\,\overline{bd},\ b\,\overline{ab}\,\overline{cd},\ b\,\bar{a}\,\overline{bcd},\ b\,\overline{ab}\,\overline{bcd},\ b\,\overline{abd}\,\overline{cd},\ b\,\overline{ad}\,\overline{bcd},\ b\,\overline{abd}\,\overline{bcd}\}$.

Definition 2.11. A **necessary** ap_f**-implicant,** na_f**-implicant** for short, is such a ap_f-implicant that all of its tail factors can be shared by other pp_f-implicants of a different head. The set of all na_f-implicants is denoted by NA_f. An **unnecessary** ap_f-implicant is the ap_f-implicant which is not an na_f-implicant. It is called an una_f**-implicant.**

Theorem 2.2. *If the* una_f*-implicant is not selected to an exact* $OPR(f)$ then at least one exact optimum solution is not lost.

Example 2.1.11. For function f from Example 2.1.1: $NA_f = \phi$.

Definition 2.12. A **necessary pp_f-implicant, np_f-implicant** for short, is such a pp_f-implicant that is not a una_f-implicant. The set of all np_f-implicants is denoted by N_f.

From Definitions 2.7, 2.8, 2.10, 2.11 and 2.12 one can conclude that $N_f = PC_f \cup NA_f$.

Example 2.1.12. For function f from Example 2.1.1: $N_f = PC_f$.

The next theorem results directly from the property $A\,\overline{B} = A\,\overline{AB}$ and the definitions of pc_f-implicants and na_f-implicants.

Theorem 2.3. Every ap_f-implicant can be generated from a PP-implicant by addition of certain variables contained in its head to a subset of its tail kernels.

Theorem 2.4. The kernels of all prime implicants of function f are sufficient as the heads of the pp_f-implicants.

Example 2.1.13. The prime implicants of function f from Example 2.1.1 are: $\{\bar{a}\,\bar{c},\ \bar{a}\,\bar{d},\ b\,\bar{d}\}$. The heads of these implicants are $\{1,\ b\}$ which are the heads of the pp_f-implicants from Example 2.1.7.

Theorem 2.5. The kernels of all prime implicants of function $\bar{f}$ (the *complement of function f*) are sufficient as tail kernels of the mp_f-implicants of f.

Example 2.1.14. Prime implicants of the complement of function f from Example 2.1.1 are $\{a\,\bar{b},\ ad,\ cd\}$. The heads of these implicants are $\{a,\ ad,\ cd\}$ which are tail kernels of the mp_f-implicants from Example 2.1.9.

2.2. THE ALGORITHMS FOR THE MINIMIZATION

The algorithm is divided into two parts. In the first stage the set N_f for function f is found. Then the optimal permissible expressions $OPR(f)$ are generated from it. In the second stage the existing set covering algorithm is adapted by making only small modification to the way how the cost function is calculated. For the reason of explanation, the algorithms given below are simplified: for instance in reality the steps 3 and 4 of Algorithm 2.2 are done concurrently to improve efficiency.

2.2.1. THE ALGORITHMS FOR THE SET N_f GENERATION

Algorithm 2.1. Generation of set N_f of implicants.

1. Using Algorithm 2.2 generate the set M_f of maximal implicants.
2. Using Algorithm 2.3 generate the set PC_f of principal implicants.
3. Using Algorithm 2.4 generate the set N_f of necessary implicants.

Algorithm 2.2. Generation of set M_f of maximal implicants.

1. Generate the set of all prime implicants of function f.
2. Find all positive kernels of all the prime implicants generated in step 1, put them in set H_f.
3. If necessary, calculate the complement $\bar{f}$ (see [36]). Generate the set of all prime implicants of $\bar{f}$.
4. Find all positive kernels of all the prime implicants found in step 3, put them in set T_f.
5. Select one element H from set H_f as the head and delete the variables contained in this head from each element in T_f, creating a new set T_f'.
6. Using Algorithm 2.5 generate an mp_f-implicant that has head H and put it to set M_f.
7. Repeat steps 5 and 6 until all the elements in H_f have been selected.

Algorithm 2.3. Generation of set PC_f of principal implicants.

1. Copy set M_f to PC_f.
2. Select an mp_f-implicant from M_f.
3. Make one possible combinational deletion of variables from one of its tail kernels.
4. Check whether the new permissible expression with deleted tail kernels is still a pp_f-implicant. If yes, append this new pp_f-implicant to set PC_f.
5. Repeat steps 2 and 3 until all the possible combinational deletions of variables from all tail kernels have been applied (this is a *tree search* process with backtracking).
6. Repeat steps 2 to 4 until all the mp_f-implicants in M_f have been processed.

Algorithm 2.4. Generation of set N_f of necessary implicants.

1. Select one pc_f-implicant from PC_f.
2. Make one possible combinational addition of variables contained in its head to a subset of its tail kernels in order to create a new candidate for a np_f-implicant to N_f.
3. Using Algorithm 2.6 check each tail factor with added variables TA to see if it could be shared by other np_f-implicants. If yes, append this new np_f-implicant to N_f.
4. Repeat steps 2 and 3 until all the possible combinational additions of variables have been applied.
5. Repeat steps 1 to 4 until all the pc_f-implicants from PC_f have been processed.

Algorithm 2.5. Generation of the set of mp_f-implicants that have head H.

1. Find the set Z, which consists of those false minterms that are covered by the head H. It is done by an *intersection operation* of the head H with the set of all false minterms.
2. Copy Z to ZC.
3. Select one false minterm from set Z.
4. Select one tail kernel from those created in Algorithm 2.2.
5. Using intersection operation, verify if this kernel covers the selected false minterm. If no, repeat steps 3-5.
6. Using *sharp operation* remove the false minterms covered by this tail kernel from the set ZC.
7. Repeat steps 3 to 6 until ZC becomes empty.
8. Select another set of tail kernels which also covers all the false minterms in Z. Repeat steps 2 to 7 until all such selections have been tried.
9. Select one of the sets created above that contains most variables in its tail kernels. Combine the selected head with the complements of the tail kernels to generate an mp_f-implicant. Return it as the value.

Algorithm 2.6. Checking the augmented tail factor.

1. Select one of the already generated np_f-implicants from N_f, $NP1$, which has the head different from those selected before. If all the already generated np_f-implicants from N_f have been processed, end the check and return the *'no'* signal.
2. Compare the tail factors of $NP1$ with the checked tail factor TA to see if there is one equal to it. If yes, end the check and return *'yes'* signal.
3. Compare the tail kernels of $NP1$ with the checked kernel $\overline{TA}$ to see if there is one which includes it. If no, repeat steps 1 and 2, otherwise go to the next step.
4. Delete all the variables contained in the tail kernel found in step 3 from the checked factor TA.
5. Check if the head of the selected np_f-implicant contains all the remaining variables left. If no, repeat steps 1 to 4, otherwise return the *'yes'* signal and go to the next step.
6. Copy the selected np_f-implicant and replace the tail factor found in step 3 with the checked tail factor to create a new np_f-implicant if it does not exist.

2.2.2. THE ALGORITHM FOR FINDING OPR FROM THE SET N_f

Since the $OPR(f)$ is found from the set of existing np_f-implicants, our search model is that of the *covering table* [23,24,30] rather than the *covering-closure table* [34,35,1,10]. Preparation of data for this stage requires specifying column factors and row factors for the searching table as well as specifying a proper cost function for this particular case. The column factors and row factors for this problem are the minterms and np_f-implicants, respectively. The standard cost function for state-space search is $fc = c(n) + p(n)$, where $c(n)$ for the covering problem is the total cost of row factors already applied, and $p(n)$ is a prediction of the cost of other row factors still needed to cover the remaining columns. The cost of a single row factor is the cost of circuit realization for its corresponding np_f-implicant. Realization of each np_f-implicant should have gates for the tail factors as well as a gate to realize the logic product of the head and all the tail factors. A gate should be always counted for each tail factor that is not shared. For a shared tail factor the gate should be counted only once for all the factors that share it. The connection cost for the second level is the number of tail factors plus the number of the variables contained in the head. The connection cost for the third level group is the total sum of numbers of variables contained in each tail kernel. The number of the third level connections for the shared tail factors should be also counted only once. The cost of TANT network should include also the cost of the first level group, that is: one gate to realize the logical sum and the connections to this gate. The prediction of the cost $p(n)$ in current algorithm is very simple - just one gate and two connections are added to the cost of those nodes being not solutions since at least one more np_f-implicant should be applied for this node and this is the least possible cost of a single pp_f-implicant. These ideas were used to create the algorithm for the cost function.

Algorithm 2.7.
Evaluation of the cost of the node in the state-space for the *OPR* covering problem.

1. Count one gate for the first level gate cost. Count the number of the np_f-implicants already applied as the connection cost for this gate.
2. Select one of the np_f-implicants already applied and evaluate its cost using Algorithm 2.8. Add the cost of this selected np_f-implicant to the cost already evaluated.
3. Repeat step 2 until all the applied np_f-implicants have been evaluated.
4. If it is not a solution node, add one gate and two connections to the cost evaluated above.

Algorithm 2.8. Evaluation of the cost of applied np_f-implicants in a node.

1. Count one gate for the realization of the logical product in the second level gate. Count the number of variables contained in head and the number of tail factors as the connection cost for this gate.
2. For each tail factor check if it is also contained in the np_f-implicant already evaluated for the current node. If no, count one gate for the third level gate and count the number of the variables contained in this tail kernel as the connection cost of this gate.

3. THE ALGORITHMS FOR THE FUNCTION OF NOT ONLY AFFIRMATIVE VARIABLES

3.1. THE ALGORITHM FOR THE FUNCTION OF SOME VARIABLES IN ONLY COMPLEMENTED FORM

Certain transformation should be first performed for a function in which some variables are available only in a complemented form: those variables that are only in a complemented form are replaced with new variables available only in affirmative form. It will be illustrated by the following example.

Example 3.1.1. For the function from Fig. 3.1a the PLA-expression is: $f = \bar{b}\,\bar{c} + d\,\bar{a}\,\bar{b} + b\,\bar{a}\,\bar{d}$. If variable b is available only in the complemented form, a new variable $u = \bar{b}$ is introduced. After replacing b with $\bar{u}$ the function is transformed to a new form from Fig. 3.1b where the variables a, u, c, d are all available in the affirmative form. For this new function all the algorithms from this section can be directly applied. The $OPR(f)$ is found:

$f = \bar{a}\,\bar{d}\,\overline{uc} + u\,\overline{\bar{u}c} + u\,d\,\bar{a}$.

Next by substituting the original variable b for $\bar{u}$ and $\bar{b}$ for u, one gets the OPR with original variables:

$f = \bar{a}\,\bar{d}\,\overline{\bar{b}\,c} + \bar{b}\,\overline{\bar{b}\,c} + d\,\bar{b}\,\bar{a}$ (Fig. 3.1c).

From the above example the algorithm for the function with some variables available only in complemented form can be found, as the following modification to the algorithms from section 2.2.

Modification 3.1. For the function of some variables available only in a complemented form the Algorithm 2.1 should start with the following step:

0. Transform the function to a new function, of only affirmative input variables, by replacing the negative variables with new affirmative variables.

Also, after the OPR is found the following step is executed:

**. Substitute original complemented form variables to the corresponding replacement variables in the OPR found from the set covering problem to get the OPR for the original function.

3.2. THE ALGORITHM FOR THE FUNCTION OF SOME VARIABLES AVAILABLE IN BOTH AFFIRMATIVE AND COMPLEMENTED FORMS

The algorithms for the function of only affirmative variables can be adopted for the function of some variables available in both forms, when the following modification is introduced.

Modification 3.2. In the searching process for the OPR step 1 in Algorithm 2.8 should be modified as follows:

1A. Check for each tail kernel if it contains only one variable and this variable is available in both forms. If yes, count no cost for this tail factor. Otherwise check if this tail factor is also contained in the np_f-implicant already evaluated for the current node. If no, count one gate for the third network level gate and count the number of the variables contained in this tail kernel as the connection cost for this gate.

Another modification is to do all processing for available kernels instead of positive kernels.

4. THE ALGORITHM FOR THE INCOMPLETELY SPECIFIED FUNCTION

For the function containing don't cares, the OPR can not always be found among the implicants of the N_f set defined by Definition 2.12. This can be illustrated by the following example.

Example 4.1. For the function from Fig. 4.1 the corresponding set N_f can be found according to the algorithms for the function without don't cares: $N_f = \{\bar{a}\,\bar{b}\,\bar{c},\ \bar{c}\,\bar{d},\ \bar{a}\,\bar{c}\,\overline{bd},\ a\,\bar{d}\}$. The *OPR* found from this N_f is $f = \bar{a}\,\bar{c}\,\overline{bd} + a\,\bar{d}$. If the don't cares are treated in this functions as false_minterms, another form of *OPR* can be generated with the same algorithms: $f = \bar{a}\,\bar{c}\,\overline{bd} + ab\,\overline{bd}$. The cost of the second expression is one gate less than that of the first one. In the second expression, however, the item $ad\,\overline{bd}$ is not a pp_f-implicant defined in Definition 2.7, since it is included in the prime implicant $a\bar{d}$. We call this item an **additional implicant.** Therefore, for an incompletely specified function some additional implicants should be generated in order to find *OPR*. The generation of additional implicants is actually the generation of additional heads.

Definition 4.1. An **additional head** is the kernel of an implicant being included in certain prime implicant.

Example 4.2. The additional heads for the function f from Example 4.1 are $\{ab, ac, abc\}$ which are the kernels of $ab\,\bar{d}$, $ac\,\bar{d}$ and $abc\,\bar{d}$, respectively. All these implicants are included in prime implicant $a\,\bar{d}$.

Definition 4.2. A **separate minterm** included in a prime implicant is such a minterm that its kernel is not included in a kernel of any other minterm that is covered by the same prime implicant.

Example 4.3. For minterms included in the prime implicant a of the function f from Fig. 4.2a, the separate minterms are ad and abc.

Theorem 4.1. It can be proven that no additional head need to be generated for an np_f-implicant, if one of the following conditions is satisfied for the pp_f-implicant:

1. It is included in a prime implicant that contains no don't cares.
2. It is included in a prime implicant which has no tail factors.
3. It is included in a prime implicant which has the same head as one of the minterms included in this prime implicant.
4. It is included in a prime implicant in which the number of separate minterms included in this prime implicant exceeds the number of the tail factors of the pp_f-implicant.
5. It contains only don't cares.

Example 4.4. No additional heads should be generated for implicants covered by the prime implicant a of function f from Fig. 4.2a. No additional heads should be generated for implicants covered by the prime implicant $a\,\bar{d}$ of the functions from Fig. 4.2b, c, d, since:

1. The prime implicant $a\bar{d}$ in (b) contains no don't cares.
2. The prime implicant $a\bar{\bar{d}}$ in (c) has the same head as the separate minterm $a\,\bar{b}\,\bar{c}\,\bar{d}$ included in it.
3. The number of separate minterms covered by the prime implicant $a\bar{d}$ in (d), which is two, exceeds the number of its tail factors, which is one.

Therefore, using the following modifications the algorithm for completely specified functions can be adopted for incompletely specified functions.

Modification 4.1. The modification of step 2 in Algorithm 2.2 for incompletely specified functions looks like this:

2A. Select positive kernels of all the prime implicants found in step 1. Combine these kernels with additional heads generated by Algorithm 4.1 to form set H_f.

Algorithm 4.1. Generation of additional heads.

1. Select a prime implicant of the function.
2. Check if the selected implicant contains don't cares. If no, repeat step 1. Otherwise go to the next step.
3. Generate true minterms contained in the selected implicant.
4. Generate separate true minterms from those found in step 3.
5. Check if the number of separate minterms exceeds the number of tail factors of the selected prime implicant. If yes, go to step 1.
6. Check if there is a separate minterm which has the same head as the selected implicant. If yes, go to step 1.
7. Using Algorithm 4.2 generate the heads of the implicants covered by the *selected implicant* and put them in the set of additional heads.
8. Repeat steps 1 to 7 until all the prime implicants have been processed.

Algorithm 4.2.
Generation of heads of implicants covered by a selected prime implicant.

1. Select a separate true minterm covered by the selected prime implicant and generate its kernel.
2. Generate the kernels of the true minterms or don't cares that are covered by the selected prime implicant, which either include the kernel found in step 1 or are included in it. Put these kernels in the set of additional heads.
3. Repeat steps 1 and 2 until all the separate true minterms have been processed.

5. THE ALGORITHM FOR THE MULTIOUTPUT FUNCTIONS

For a multi-output function, the function corresponding to a single output will be called **the subfunction for this output.** The OPR for total output will always be the accumulation of OPRs for each subfunction. This can be illustrated by the following example.

Example 5.1. The subfunctions f^1, f^2, and f^3 of a three-output function f are:
$f^1(a,b,c,d) = \sum(0,1,2,4,5,6,12,14)$, $f^2(a,b,c,d) = \sum(2,6,8,10,12,14)$, $f^3(a,b,c,d) = \sum(0,1,4,5,8,10)$.
The OPRs for each separate subfunction are:

$$f^1 = \bar{a}\,\overline{cd} + b\,\bar{d}, \quad f^2 = a\,\bar{d} + c\,\overline{cd}, \quad f^3 = \bar{a}\,\bar{c} + a\,\bar{b}\,\bar{d}.$$

The TANT network has cost (14, 24). If one changes the permissible expressions for f 1 and f 2 to another form:

$$f^1 = \bar{a}\,\bar{c} + c\,\bar{a}\,\bar{d} + ab\,\bar{d},\ f^2 = ab\,\bar{d} + c\,\bar{a}\,\bar{d} + a\,\bar{b}\,\bar{d},$$

the TANT network of cost (11, 23) is found.

Comparing the two networks we find that the cost is reduced because of the existance of some implicants which can be shared by several subfunctions. Sometimes the commonly shared implicant is not an pp_f-implicant for all the subfunctions, such as $c\bar{a}\bar{d}$ in Example 5.1. Therefore the pp_f-implicants cannot be created in the generation of N_f for each subfunction with algorithms for single output function. We noticed that the commonly shared implicants contain the minterms which are commonly contained by the subfunctions sharing them. By intersection of sets of minterms of different subfunctions one can easily find their commonly contained minterms. Then this set of minterms can be treated as the minterms of a new independent function, and the new set N_f for this function is generated using the algorithms for the single output function. This new set N_f contains the commonly shared implicants of the above intersections of subfunctions.

The algorithm for the multioutput function is based on the above ideas. With the modification given below the algorithms for the single output function can be applied to the multioutput function as well. Before doing this the definition helpful for understanding this modification will be formulated.

Definition 5.1. For the multioutput function $f = (f^1, f^2, \ldots f^m)$, ($m$ is the number of outputs) the *component function* is defined as a Boolean product of a subset of single output subfunctions: $f = \bigcap_{i \in I} f^i$, where $I \subseteq \{1, 2, \cdots, m\}$.

Modification 5.1.
The Algorithm 2.1 should be modified for the multioutput function as follows.
The generation of N_f can be done by:

1. Generate component function f.
2. Generate M_f with Algorithm 2.2.
3. Generate PC_f with Algorithm 2.3.
4. Generate N_f with Algorithm 2.4 and copy it to N_f.
5. Check if all the component functions have been generated. If no, repeat steps 1 to 4.

When next the covering problem is formulated, the columns of the covering table are the minterms of all subfunctions. For the function from Example 5.1 there should be 20 columns in the covering table, since the total number of minterms in all subfunctions is 20.

6. CONCLUSION AND CURRENT WORK

Program TLN-MINI was written in FORTRAN 77 for VAX-11/780. It takes much less time to generate N_f than to select from it the $OPR(f)$. This is because in the generation most of executions are bitwise operation among memory words, while the cube array of the function is transferred to a vector in memory. In this vector each symbol of the array is represented with only two bits. TLN-MINI has been tried on many Boolean functions, and yielded always correct results. For functions with complemented variables the solutions were always exact. For multi-output functions the number of PP-implicants grows so fast that the exact variant of algorithm is not practical to use for more than 10 total input/output variables.

Presently we are working on improving its applicability [27], however, at the price of sacrifying its exactness. First, subminimal implicants [23,17] and reduced OFF-cubes will be used instead minterms and false minterms, respectively. Then, better cube calculus algorithms for complementation, tautology, supercube, overexpanded cube, disjoint sharp and other will be used [23].

We used several possible layout styles [27] to implement TLN networks: Three-level NOR/NOR/NOR PLA (called *3PLA);* Nor-based Weinberger layout; AND/OR/INVERT-based Weinberger layout; a single NOR plane of a PLA with feedback connections (as in [33]); OCT tools-based layout from Missisipi State Library, obtained after providing MIS II with the results of TLN-MIN. With some of those tools we are able to investigate the influence of the number of gates in each level on the shape and size of the layout cell. The user can affect the layout's shape as well [27].

We are working also on another related topics: design of TLN network, free from any kind of static or dynamic hazard [27]; incorporating of fan-in and fan-out constraints [27]; efficient methods to solve covering/closure probelms [35,34,1,24]; merging the ideas of TLN design and Exclusive Sum of Products (ESOP) design [11,26,25] into a new concept of *Three Level NOR/EXOR/NOT Networks* that have inverters in levels 1 and 5, and NOR and EXOR gates in levels 2, 3, and 4.

7. REFERENCES.

[1] Biswas, N.N., "Implication Trees in the Minimization of Incompletely Specified Sequential Machines", *Int. J. Electron., (GB)*, No. 2, pp. 291-298, August 1983. [2] Brayton, R.K., Hachtel, G.D., McMullen, C.T., and A.L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Boston, MA, Kluwer, 1984. [3] Brayton, R.K., Camposano, R., De Micheli, G., Otten, R.H.J.M., and J. Van Eijndhoven, *"The Yorktown Silicon Compiler System"*, Chapter 7 in Gajski, D., (ed), Silicon Compilation, 1987. [4] Ciesielski, M.J., Yang, S., and M.A. Perkowski: "Multiple-Valued Minimization Based on Graph Coloring", *Proc. International Conference on Computer Design: VLSI in Computers, ICCD'89*, October 1989. [5] Chakrabarti, K.K., Choudhury, A.K., and M.S. Basu, "Complementary Function Approach to the Synthesis of Three-Level NAND Network", *IEEE Trans. on Comput.*, Vol. C-19, pp. 509-514, June 1970. [6] Choudhury, A.K., Chakrabarti, K.K., and D. Sharma, "Some Studies on the Problem of Three-level NAND Network Synthesis", *Int. Journal of Control*, Vol. 6., No. 6., pp. 547-572, 1967. [7] Dietmayer, D.L. "Logic Design of Digital Systems", *Allyn and Bacon*, Boston, Mass, 1971. [8] Frackowiak, J., "The minimization of hazardless TANT networks", *IEEE Trans. on Comp. Vol.*, C-21., No. 10, pp. 1099-1108, Oct. 1972. [9] Garey, M.R., and D.S. Johnson, "Computers and Intractability. A Guide to the Theory of NP-Completeness." *W.H. Freeeman and Company*, San Francisco 1979. [10] Gimpel, J.F., "The Minimization of TANT Networks", *IEEE TEC*, Vol. EC-16, pp. 18-38, February 1967. [11]

Helliwell, M., and M.A. Perkowski, "A Fast Algorithm to Minimize Multi-Output Mixed-Polarity Generalized Reed-Muller Forms", *Proceedings of 25-th Design Automation Conference*, Anaheim, CA, paper 28.2, pp. 427-432, June 1988. [12] S.J. Hong, R.G. Cain, and D.L. Ostapko, "MINI: A heuristic approach for logic minimization", *IBM J. Res. Develop.*, Vol. 18, pp. 443 - 458, Sept. 1974. [13] Koh, K.S., "A Minimization Technique for TANT Networks", *IEEE Trans. on Comp.*, January 1971, pp. 105-107. [14] Kulpa, Z., "Synthesis of Quasi-Minimal Logic Circuits of Many Variables with use of NAND and NOR gates", *M.Sc. Thesis*, Institute of Automatic Control, Warsaw Technical University, 1970. [15] Lee, H-P.S., "An Algorithm for Minimal TANT Network Generation", *IEEE Trans. on Comp.* Vol. C-27, No. 12, Dec. 1978, pp. 1202-1206. [16] McCluskey, E.J. Jr., "Introduction to the Theory of Switching Circuit", *McGraw-Hill*, 1965. [17] Nguyen, L., Perkowski, M., and N.B. Goldstein, "PALMINI - Fast Boolean Minimizer for Personal Computers", *Proc. 24th Design Automation Conference*, June 28-July 1, 1987, Miami, Florida, Paper 33.3. [18] Perkowski, M.A., "Synthesis of multioutput three level NAND networks", *Proc. of the Seminar on Computer Aided Design*, Budapest, 3-5 November 1976, pp.238-265. [19] Perkowski, M.A., "An example of heuristic programming application in the three-level combinational logic design", *Proceedings of the 3rd Symposium on Heuristic Methods*, Polish Cybernetical Society Warsaw, 25 September 1976, Vol. 1. pp. 105-132. [20] Perkowski, M.A., "Minimization of Two-Level Networks from Negative Gates", *Proc. Midwest 86 Conference on Circuits and Systems*, Lincoln, Nebraska, 1- 12 August 1986. [21] Perkowski, M.A., "A Parallel Programming Approach to the Design of Two-Level Networks with Negative Gates", *International Workshop on Logic Synthesis, Research* Triangle Park, North Carolina, May 12 - 15, 1987. [22] Perkowski, M.A., Ming, L.J., and A. Wieclawski, "An Expert System for Optimization of Multi-Level Logic", *IASTED Conference, Applied Simulation and Modeling, ASM '87*, Santa Barbara, CA, May 26 - 29, 1987. [23] Perkowski, M.A., Wu, P., and K. Pirkl, "KUAI-EXACT: A New Approach for Multi-Valued Logic Minimization in VLSI Synthesis", *Proc. 1989 IScAS - International* Symposium on Circuits and Systems, May 9-11, 1989. [24] Perkowski, M.A., Liu, J., and J.E. Brown, "Quick Software Prototyping: CAD Design of Digital CAD Algorithms". In *"Progress in Computer Aided VLSI Design"* (G. Zobrist, editor), *Ablex Publishing Corp*, 1989. [25] Perkowski, M.A., Helliwell, M., and P. Wu, "Minimization of Multiple-Valued Input, Multi-Output Generalized Reed Muller Forms", *Proc. International Symposium on Multi-Valued Logic*, May 1989, People Republic of China. [26] Perkowski, M.A., and M. Chrzanowska-Jeske: "An Exact Algorithm to Minimize Mixed-Radix Exclusive Sums of Products for Incompletely Specified Boolean Functions", *Proc. International Symposium on Circuits and Systems*, 1990. [27] Perkowski, M.A., "Fast Algorithm for Synthesis of Three-Level NOR Networks with Constraints", *Report, Dept. EE.*, PSU, 1990. [28] Perkowski, M.A., "Minimization of Limited-Level NOR/EXOR/NOT Networks with Constraints", *Report, Dept. EE.*, PSU, 1990. [29] Rudell, R.L., and A.L. Sangiovanni-Vincentelli, "ESPRESSO-MV: algorithms for multiple-valued logic minimization, *Proc. IEEE Custom Integrated Circuits Conf.*, 1985. [30] Rudell, R.L., and A.L. Sangiovanni-Vincentelli, "Exact minimization of multiple-valued functions for PLA optimization", *ICCAD-86*, Nov. 1986. [31] Sasao, T., "An algorithm to derive the complement of a binary function with multiple-valued inputs," *IEEE Trans. on Comp.*, Vol. C-34, pp. 131 - 140, Febr. 1985. [32] Sasao, T., "MACDAS: Multi-level AND-OR circuit synthesis using two-variable function generators", *23-rd Design Automation Conference*, Las Vegas, pp. 86-93, June 1986. [33] Sasao, T., "On the Complexity of Three-Level Logic Circuits", Proc. International Workshop of Logic Synthesis, MCNC, ACM SIGDA, May 23-26 1989, paper 10.2. [34] Vink, H.A., Van Dolder B., and J. Al, "Reduction of CC-tables Using Multiple Implication", *Trans. on Comp.*, Vol. C-27, No. 10, October 1978. [35] Vink, H.A., "Minimal TANT Networks of Functions with Don't Care's and Some Complemented Input Variables", *IEEE Trans. Comp.*, Vol. C-27, No. 11., November 1978.

ab \ cd	00	01	11	10
00	1	1	0	1
01	1	1	0	1
11	1	0	0	1
10	0	0	0	0

Figure 2.1a.
Boolean Function to Example 2.1. The Karnaugh map,

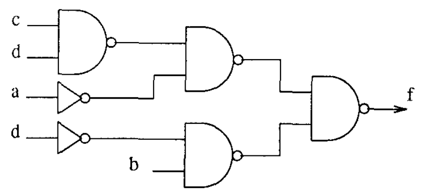


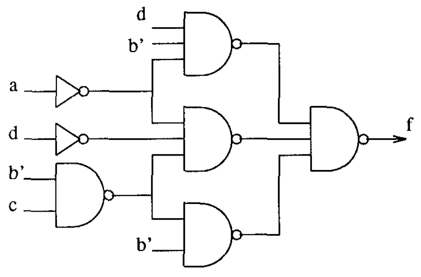
Figure 2.1b.
Boolean Function to Example 2.1.
TANT network which assumes that only positive literals are available.

ab \ cd	00	01	11	10
00	1	1	1	0
01	1	0	0	1
11	0	0	0	0
10	1	1	0	0

(a)

a u \ cd	00	01	11	10
00	1	0	0	1
01	1	1	1	0
11	1	1	0	0
10	0	0	0	0

(b)



(c)

Figure 3.1.
Boolean Function to Example 3.1.1.
(a) - the Karnaugh map,
(b) - the Karnaugh map after replacing b with $\bar{u}$
in which all variables are available in an affirmative form.
(c) the final TLN network after minimization: literal $\bar{b}$ is available,
while NOT gates are used to create the complements of variables a and d.

ab \ cd	00	01	11	10
00	1	1	0	0
01	1	0	0	0
11	1	0	0	1
10	-	0	0	-

Figure 4.1.
The Karnaugh map to Example 4.1.

ab \ cd	00	01	11	10
00	0	1	1	0
01	0	1	0	0
11	-	1	-	1
10	-	1	-	-

(a)

ab \ cd	00	01	11	10
00	0	1	1	0
01	0	1	0	0
11	1	0	0	1
10	1	0	0	1

(b)

ab \ cd	00	01	11	10
00	0	1	1	0
01	0	1	0	0
11	-	0	0	1
10	1	0	0	-

(c)

ab \ cd	00	01	11	10
00	0	1	1	0
01	0	1	0	0
11	1	0	0	-
10	-	0	0	1

(d)

Figure 4.2.
The Karnaugh maps to Example 4.4.